Module 2: R Programming Basics

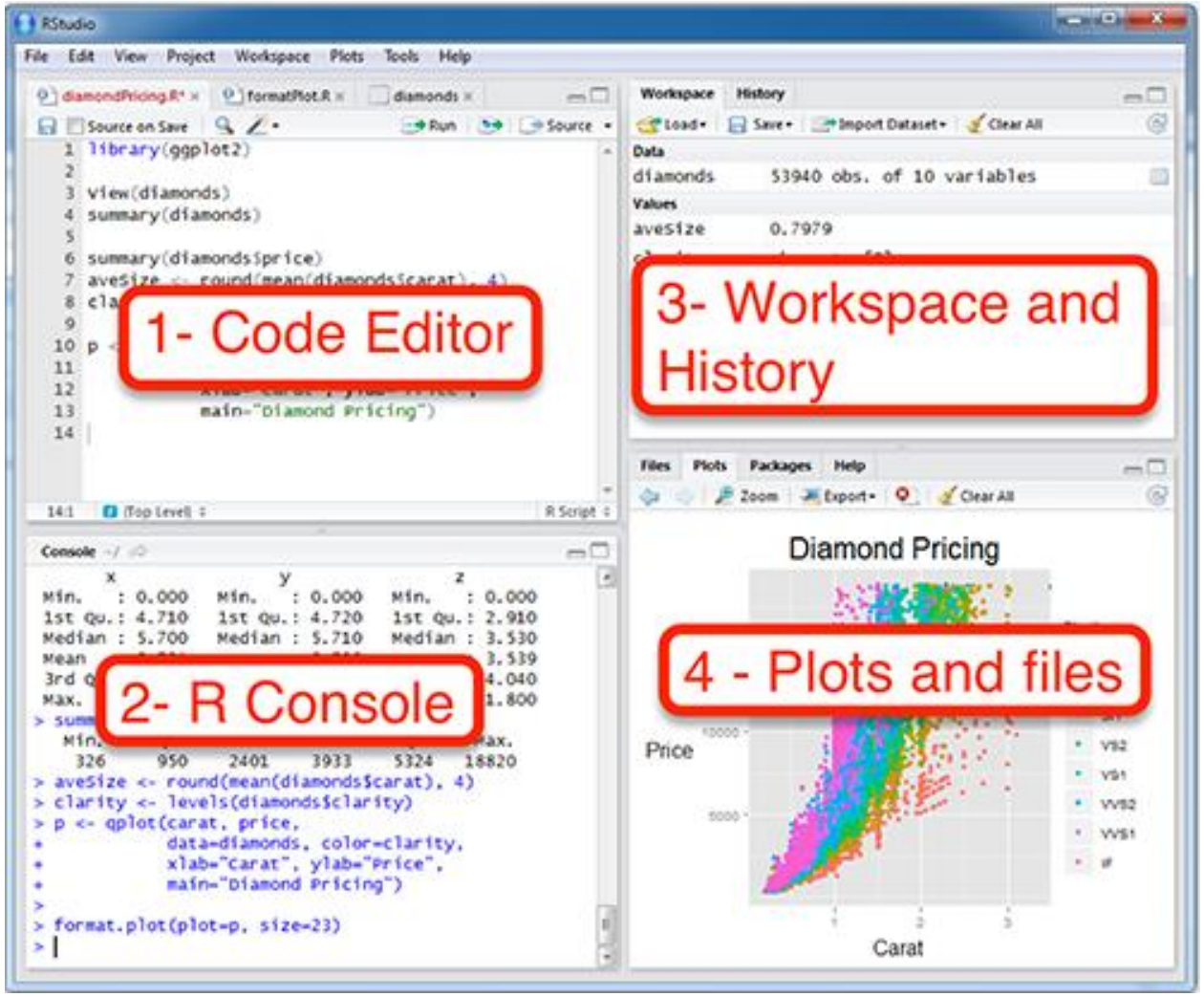
https://cran.r-project.org/doc/manuals/R-intro.pdf

BeVera

RStudio is a four pane work-space for 1) creating file containing R script, 2) typing R commands, 3) viewing command histories, 4) viewing plots and more.

1.Top-left panel:
◦Code editor allowing you to create and open a file containing R script.
◦The R script is where you keep a record of your work. R script can be created as follow: File –> New –> R Script

2.Bottom-left panel:
◦R console for typing R commands



3.Top-right panel:
◦Workspace tab: shows the list of R objects you created during your R session
◦History tab: shows the history of all previous commands

4.Bottom-right panel:
◦Files tab: show files in your working directory
◦Plots tab: show the history of plots you created. From this tab, you can export a plot to a PDF or an image files
◦Packages tab: show external R packages available on your system. If checked, the package is loaded in R.

Importing your data

R allows for the import of different data formats using specific packages that can make your job easier:

Package	Purpose
readr	Importing flay files
readxl	Getting excel files into R
haven	Import SAS, STATA and SPSS data files into R.
RMySQL and RpostgreSQL	Connect to databases, access and manipulate via DBI
rvest	webscraping

Example: mydata=read.csv(file="c:/…..csv", header=TRUE)

https://www.datacamp.com/community/tutorials/r-data-import-tutorial

Manipulating your data

Package	Purpose
tidyr	tidying your data
stringr	string manipulation
dplyr	to learn data frame like objects, solve data manipulation challenges
data.table	perform heavy data wrangling tasks
zoo, xts, quantmod	performing time series analysis

Using packages

1

```
install.packages("haven")
```

Downloads files to computer

1 x per computer

2

```
library("haven")
```

Loads package

1 x per R Session

R is Infinitely Expandable

Applications of R normally use a package; i.e., a library of special functions designed for a specific problem.

Hundreds of packages are available, mostly written by users.

A user normally only loads a handful of packages for a particular analysis (e.g., library(caret)).

Standards determine how a package is structured, works well with other packages and creates new data types in an easily used manner.

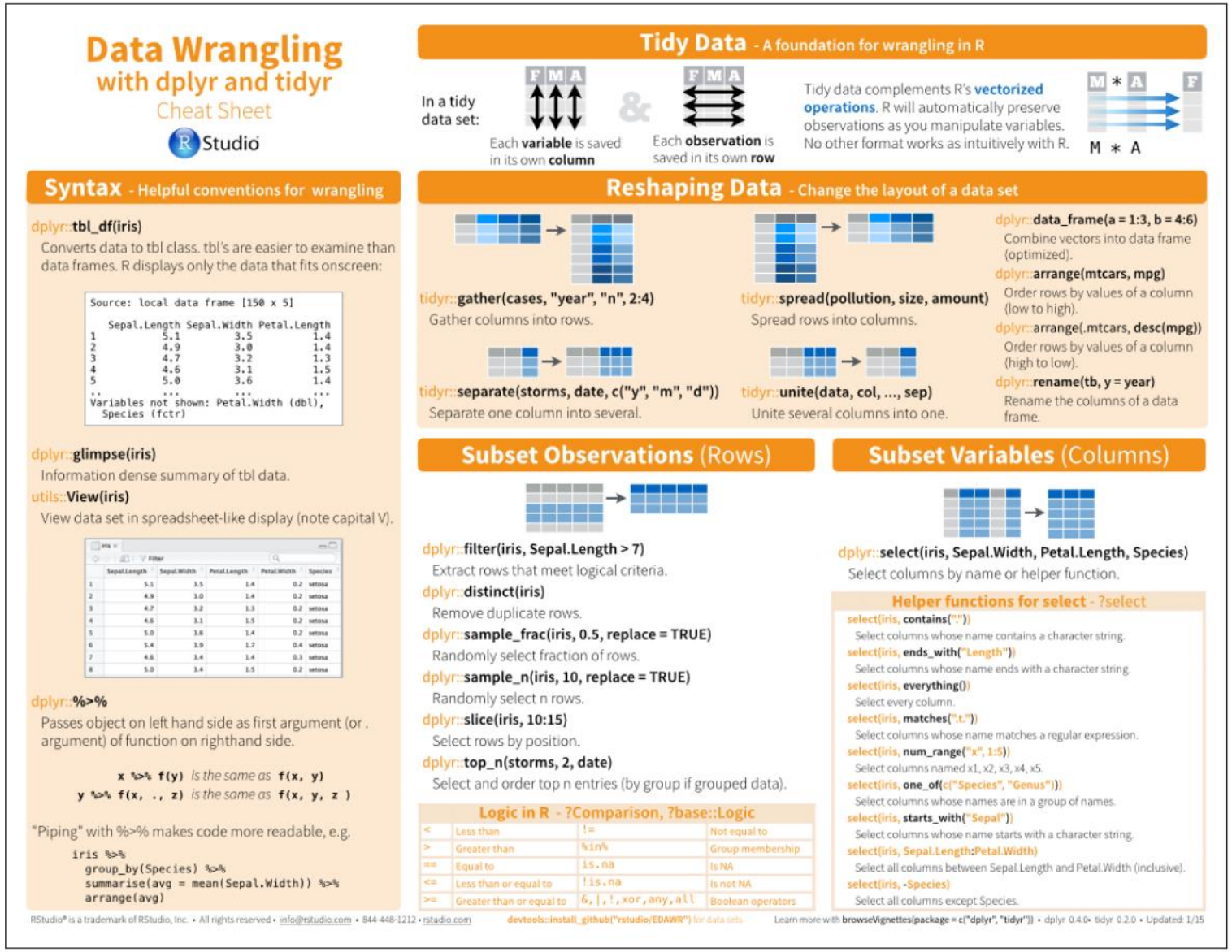
Standardization makes it easy for users to learn new packages.

Additional resources for R programming basics

http://www.sthda.com/english/wiki/easy-r-programming-basics

https://www.rstudio.com/resources/webinars/data-wrangling-with-r-and-rstudio/

https://github.com/rstudio/webinars/blob/master/05-Data-Wrangling-with-R-and-RStudio/wrangling-webinar.pdf



Data Wrangling with dplyr and tidyr

Cheat Sheet

RStudio

Tidy Data - A foundation for wrangling in R

In a tidy data set:

Each **variable** is saved in its own **column**

&

Each **observation** is saved in its own **row**

Tidy data complements R's **vectorized operations**. R will automatically preserve observations as you manipulate variables. No other format works as intuitively with R.

M * A → F

M * A

Syntax - Helpful conventions for wrangling

dplyr::**tbl_df(iris)**

Converts data to tbl class. tbl's are easier to examine than data frames. R displays only the data that fits onscreen:

```
Source: local data frame [150 x 5]

   Sepal.Length Sepal.Width Petal.Length
1           5.1         3.5          1.4
2           4.9         3.0          1.4
3           4.7         3.2          1.3
4           4.6         3.1          1.5
5           5.0         3.6          1.4
..          ...         ...          ...
Variables not shown: Petal.Width (dbl),
  Species (fctr)
```

dplyr::**glimpse(iris)**

Information dense summary of tbl data.

utils::**View(iris)**

View data set in spreadsheet-like display (note capital V).

dplyr::**%>%**

Passes object on left hand side as first argument (or . argument) of function on righthand side.

`x %>% f(y)` *is the same as* `f(x, y)`

`y %>% f(x, ., z)` *is the same as* `f(x, y, z )`

"Piping" with %>% makes code more readable, e.g.

```
iris %>%
  group_by(Species) %>%
  summarise(avg = mean(Sepal.Width)) %>%
  arrange(avg)
```

Reshaping Data - Change the layout of a data set

tidyr::**gather(cases, "year", "n", 2:4)**

Gather columns into rows.

tidyr::**spread(pollution, size, amount)**

Spread rows into columns.

tidyr::**separate(storms, date, c("y", "m", "d"))**

Separate one column into several.

tidyr::**unite(data, col, ..., sep)**

Unite several columns into one.

dplyr::**data_frame(a = 1:3, b = 4:6)**

Combine vectors into data frame (optimized).

dplyr::**arrange(mtcars, mpg)**

Order rows by values of a column (low to high).

dplyr::**arrange(.mtcars, desc(mpg))**

Order rows by values of a column (high to low).

dplyr::**rename(tb, y = year)**

Rename the columns of a data frame.

Subset Observations (Rows)

dplyr::**filter(iris, Sepal.Length > 7)**

Extract rows that meet logical criteria.

dplyr::**distinct(iris)**

Remove duplicate rows.

dplyr::**sample_frac(iris, 0.5, replace = TRUE)**

Randomly select fraction of rows.

dplyr::**sample_n(iris, 10, replace = TRUE)**

Randomly select n rows.

dplyr::**slice(iris, 10:15)**

Select rows by position.

dplyr::**top_n(storms, 2, date)**

Select and order top n entries (by group if grouped data).

Logic in R - ?Comparison, ?base::Logic

<	Less than	!=	Not equal to
>	Greater than	%in%	Group membership
==	Equal to	is.na	Is NA
<=	Less than or equal to	!is.na	Is not NA
>=	Greater than or equal to	&, |, !, xor, any, all	Boolean operators

Subset Variables (Columns)

dplyr::**select(iris, Sepal.Width, Petal.Length, Species)**

Select columns by name or helper function.

Helper functions for select - ?select

select(iris, contains("."))
Select columns whose name contains a character string.

select(iris, ends_with("Length"))
Select columns whose name ends with a character string.

select(iris, everything())
Select every column.

select(iris, matches(".t."))
Select columns whose name matches a regular expression.

select(iris, num_range("x", 1:5))
Select columns named x1, x2, x3, x4, x5.

select(iris, one_of(c("Species", "Genus")))
Select columns whose names are in a group of names.

select(iris, starts_with("Sepal"))
Select columns whose name starts with a character string.

select(iris, Sepal.Length:Petal.Width)
Select all columns between Sepal.Length and Petal.Width (inclusive).

select(iris, -Species)
Select all columns except Species.

devtools::install_github("rstudio/EDAWR") for data sets

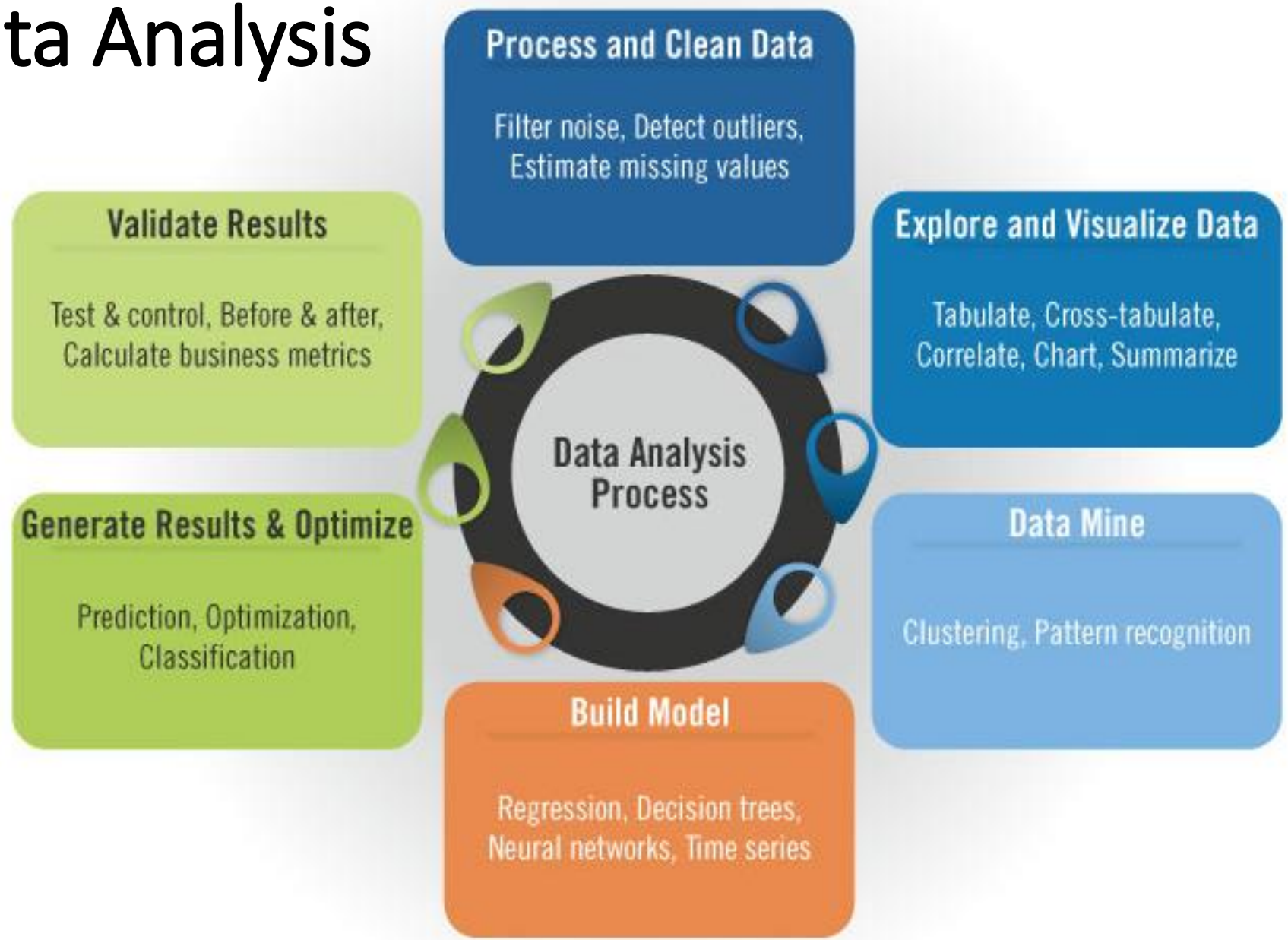
Learn more with browseVignettes(package = c("dplyr", "tidyr")) • dplyr 0.4.0 • tidyr 0.2.0 • Updated: 1/15

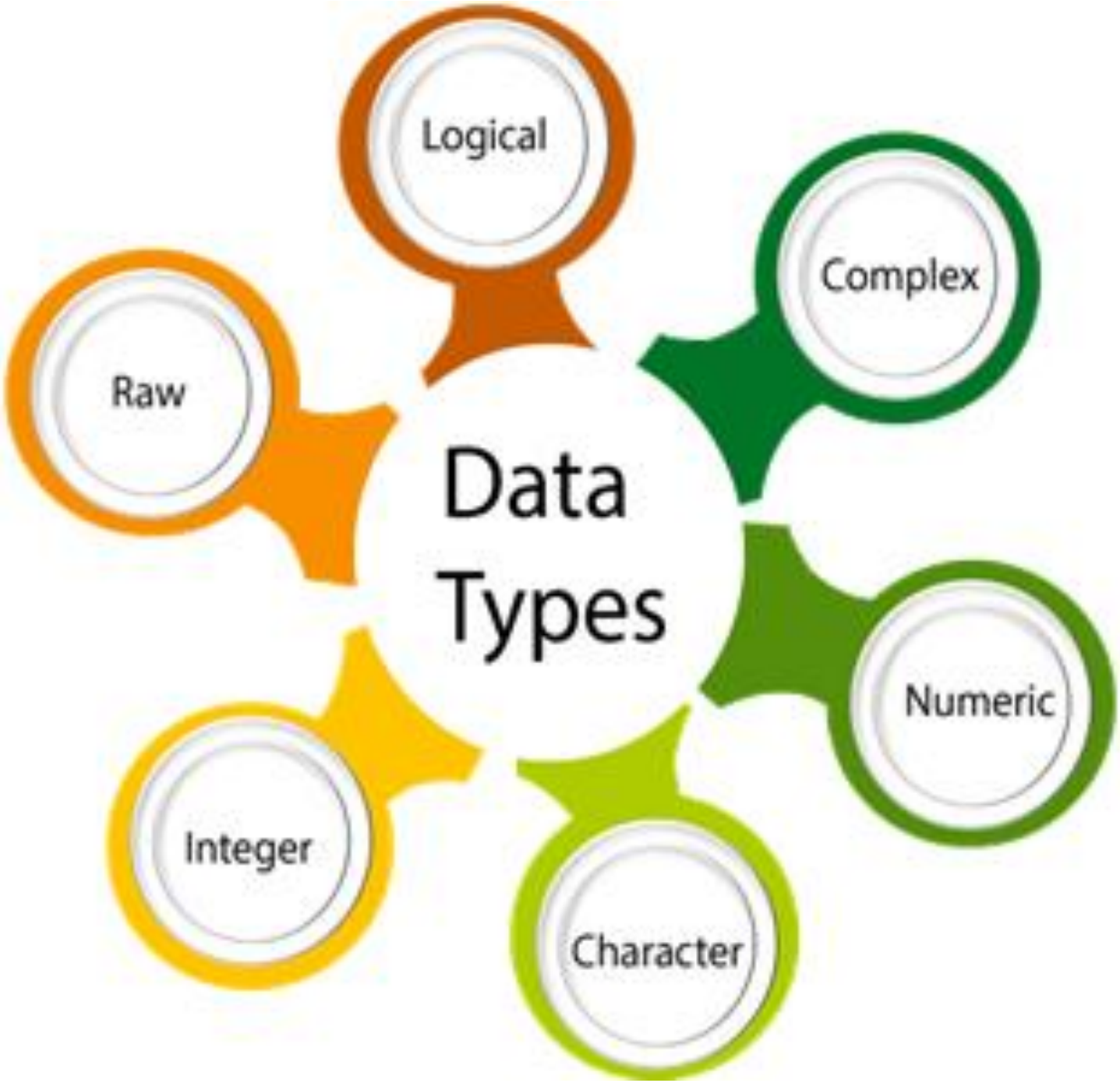
Also available for:

- Base R
- Advanced R
- Data Table
- Devtools
- ggplot2
- R Markdown
- Regular Expressions
- Rstudio IDE
- Shiny

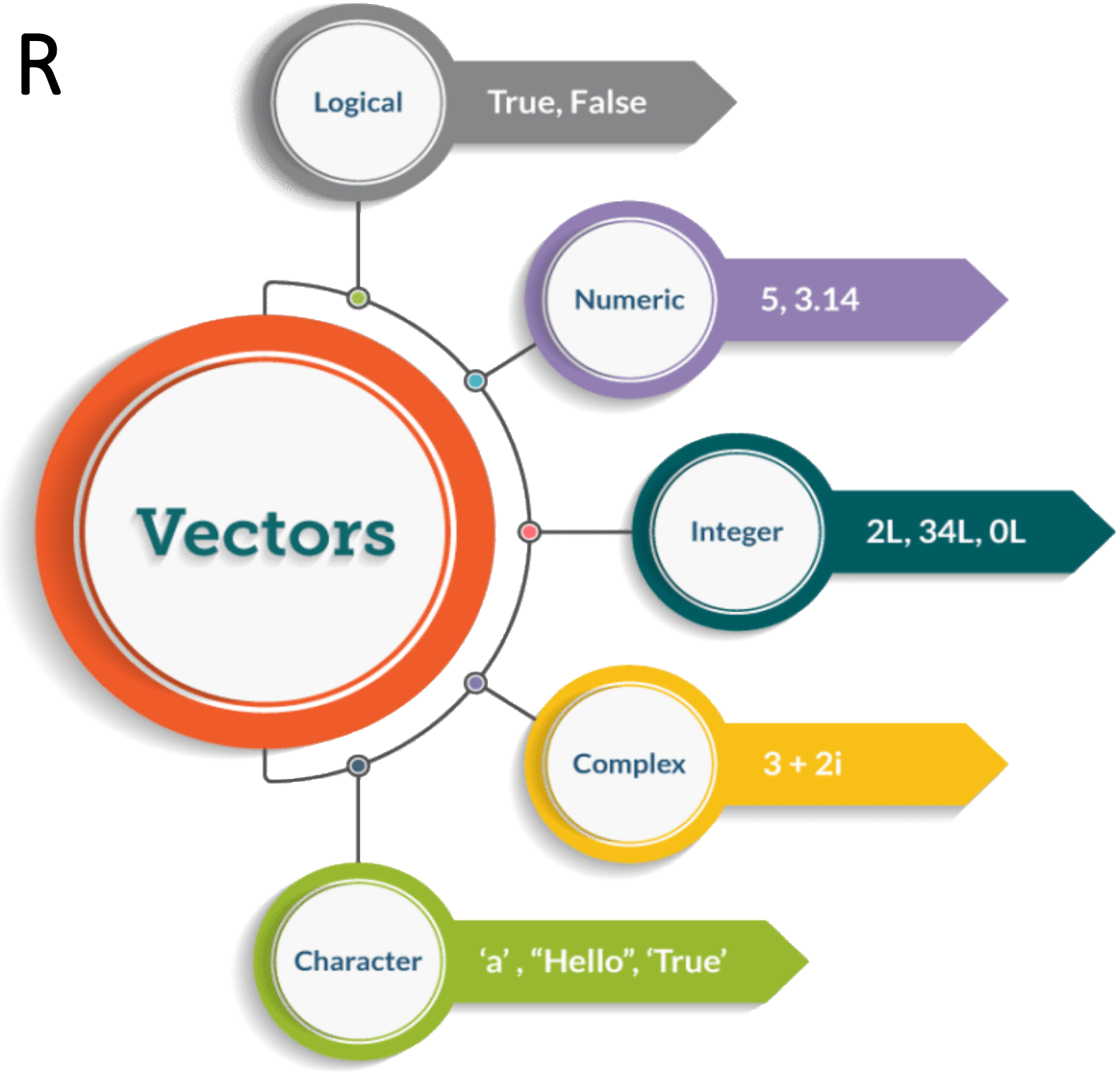
http://www.rstudio.com/resources/cheatsheets/

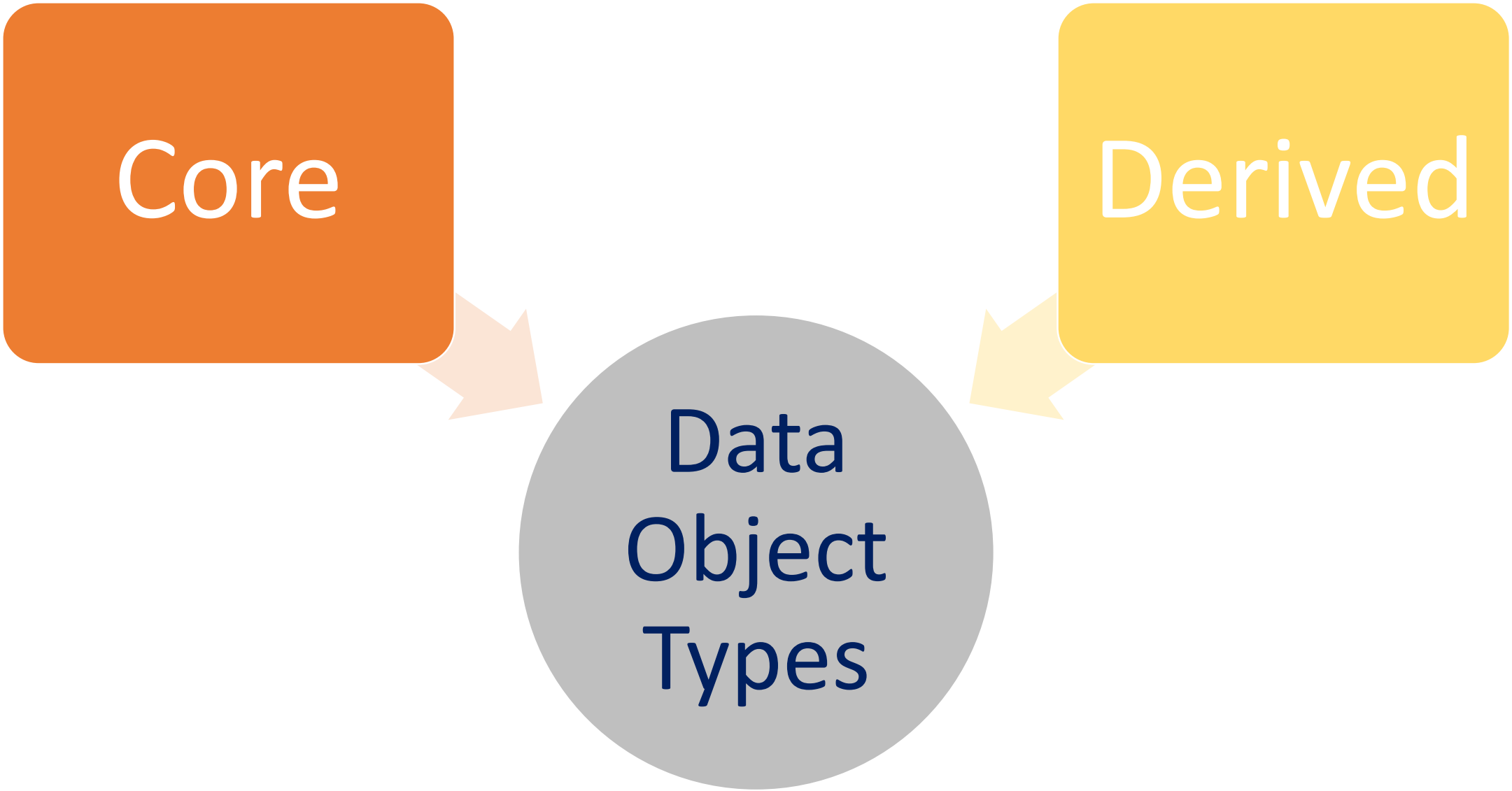
Steps of Data Analysis




Logical
Complex
Raw
Data Types
Numeric
Integer
Character

Data Types in R




Core
Derived
Data Object Types

Declare variables of different types

Example 1:
# Numeric
x <- 28
class(x)
Output: ## [1] "numeric"

Example 2:
# String
y <- "R is Fantastic"
class(y)
Output: ## [1] "character“

Example 3:
# Boolean
z <- TRUE
class(z)
Output: ## [1] "logical"

Variables

- Variables store values and are an important component in programming. A variable can store a number, an object, a statistical result, vector, dataset, a model prediction basically anything R outputs. We can use that variable later simply by calling the name of the variable.

- To declare a variable, we need to assign a variable name. The name should not have space. We can use _ to connect to words.

- To add a value to the variable, use <- or =.

Here is the syntax:

```
# First way to declare a variable:  use the `<-`
name_of_variable <- value

# Second way to declare a variable:  use the `=`
name_of_variable = value
```

In the command line, we can write the following codes to see what happens:

Example 1:

```
# Print variable x
x <- 42
x
```

Output: ## [1] 42

Example 2:

```
y  <- 10
y
```

Output: ## [1] 10

Example 3:

```
# We call x and y and apply a subtraction
x-y
```

Output: ## [1] 32

:

Basic Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^ or **	Exponentiation

```
# Examples:
# A multiplication
3*9
Output: ## [1] 27

# A division
(5+13)/2
Output: ## [1] 9

# Exponentiation
-3^2
Output: ## [1] 9
```

Logical Operators return values inside the vector based on logical conditions.

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Exactly equal to
!=	Not equal to
!x	Not x
x	y
x & y	x AND y
isTRUE(x)	Test if X is TRUE

You can add many conditional statements but we need to include them in a parenthesis. Follow this structure to create a conditional statement:

```
variable_name[(conditional_statement)]
```

```
# Example:


# Create a vector from 1 to 8
logical_vector <- c(1:8)
logical_vector > 6


Output: ## [1]FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE
```

```
# Example:


logical_vector <- c(1:8)
logical_vector[(logical_vector>3) & (logical_vector<5)]


Output: ## [1] 4
```

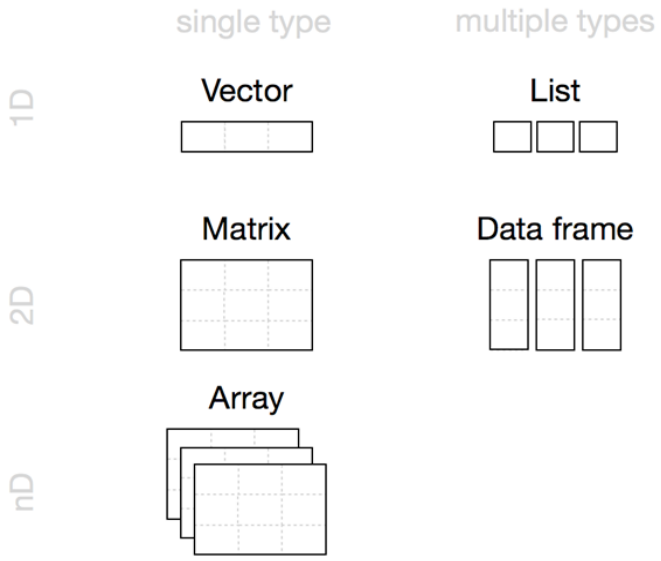
Data Structures in

1. VECTORS
2. MATRIX
3. ARRAY
4. LIST
5. DATA FRAME

Learning objectives of this module:

- What are the three properties of a vector, other than its contents?
- What are the four common types of atomic vectors? What are the two rare types?
- What are attributes? How do you get them and set them?
- How is a list different from an atomic vector? How is a matrix different from a data frame?
- Can you have a list that is a matrix?

Fundamental Data Structures



Vector

- A sequence of numbers or characters, or higher-dimensional arrays like matrices

Matrix

- The basic two dimensional data structure in R is the vector

Array

- If a higher dimension vector is desired, then use the function to generate the n-dimensional object.

List

- An ordered set of components stored in a 1D vector

Data.Frame

- A table-like structure (experimental results often collected in this form)

Factor

- A sequence assigning a category to each index.

Atomic Data Elements: Vectors

- In R the “base” type is a vector, not a scalar.
- A vector is an indexed set of values that are all of the same type. The type of the entries determines the class of the vector.
- The possible vectors data types are:
  - integer
  - numeric
  - character
  - complex
  - logical
- An integer is a subclass of numeric.
- Cannot combine vectors of different modes

Creating Vectors

Assignment of a value to a variable is done with <- (two symbols, no space).

> v <- 1

➢ v

Vectors can only contain entries of the same type: numeric or character; you can't mix them.

- Note that characters should be surrounded by " ". The most basic way to create a vector is with c(x1, . . . , xn), and it works for characters and numbers alike.

> x <- c("a", "b", "c")

> length(x)

[1] 3

Can also generate regular sequences:

seq(from = #, to = #, by = #): allows you to create a sequence from a starting number to an ending number.

rep(x = c(1, 2), each = 3): function allows you to repeat a scalar (or vector) a specified number of times, or to a desired length.

Vector Arithmetic

Numeric vectors can be used in arithmetic expressions, in which case the operations are performed element by element to produce another vector.

```
> x <- rnorm(3)
> y <- rnorm(3)
> x
> x + 1
> c(1, 2, 3, 4, 5) + c(5, 4, 3, 2, 1)
```

More Vector Arithmetic Statistical operations on numeric vectors

- In studying data you will make frequent use of sum, which gives the sum of the entries, max, min, mean.

Function	Example	Result
sum(x), product(x)	sum(1:20)	210
min(x), max(x)	min(1:20)	1
mean(x), median(x)	mean(1:20)	10.5
sd(x), var(x), range(x)	sd(1:20)	5.91608
quantile(x, probs)	quantile(1:20, probs = .2)	20%, 4.8
summary(x)	summary(1:20)	Min = 1.00. 1st Qu. = 5.75, Median = 10.50, Mean = 10.50, 3rd Qu. = 15.25, Max = 20.0

> (i.e. Sum((x - mean(x))^2)/(length(x) - 1))

- A useful function for quickly getting properties of a vector: summary(y)

More Vector Arithmetic Statistical operations on continuous vectors

Function	Description	Example	Result
round(x, digits)	Round elements in x to digits digits	round(c(3.712, 3.1415), digits = 1)	3.7, 3.1
ceiling(x), floor(x)	Round elements x to the next highest (or lowest) integer	ceiling(c(2.6, 8.1))	3, 9
x %% y	Modular arithmetic (ie. x mod y)	8 %% 4	0

Vector Arithmetic Statistical operations on discrete vectors

Function	Description	Example	Result
unique(x)	Returns a vector of all unique values.	unique(c(3, 3, 4,5,12))	3, 4, 5, 12
table(x, exclude)	Returns a table showing all the unique values as well as a count of each occurrence. To include a count of NA values, include the argument exclude = NULL	table(c("x", "x", "y", "z"))	x y z 2 1 1

Example: See the “indexing vectors.R” in the folder

baby.names	baby.city	baby.ages	baby.weight	baby.eyecolor
amy	macon	13	21	brown
brittany	athens	21	22	brown
carol	canton	32	41	green
donna	savannah	6	16	blue
erin	savannah	12	18	blue
fran	atlanta	11	19.4	grey
gigi	atlanta	18	26	brown
helen	athens	16	23	green
irene	macon	17	22	brown
jackie	macon	34	36	brown

Factors in R: Categorical & Continuous Variables

Factors are variables in R which take on a limited number of different values; such variables are often referred to as categorical variables.

In a dataset, we can distinguish two types of variables:

categorical and **continuous**

- In a categorical variable, the value is limited and usually based on a particular finite group. For example, a categorical variable can be countries, year, gender, occupation.

- A continuous variable, however, can take any values, from integer to decimal. For example, we can have the revenue, price of a share, etc..

Categorical Variables

R stores categorical variables into a factor. Let's check the code below to convert a character variable into a factor variable. Characters are not supported in a machine learning algorithm, and the only way is to convert a string to an integer.

Syntax

```
factor(x = character(), levels, labels = levels, ordered = is.ordered(x))
```

Arguments:

- **x**: A vector of data. Need to be a string or integer, not decimal.
- **Levels**: A vector of possible values taken by x. This argument is optional. The default value is the unique list of items of the vector x.
- **Labels**: Add a label to the x data. For example, 1 can take the label `male` while 0, the label `female`.
- **ordered**: Determine if the levels should be ordered.

Nominal Categorical Variable

A categorical variable has several values but the order does not matter. For instance, male or female categorical variable do not have ordering.

```
# Create a color vector
color_vector <- c(‘turquoise', 'red', 'green', ivory', 'black', 'yellow')

# Convert the vector to factor
factor_color <- factor(color_vector)
factor_color
```

Output:

```
## [1] turquoise red green ivory black yellow
## Levels: black turquoise green red ivory yellow
```

Ordinal Categorical Variable

Ordinal categorical variables do have a natural ordering. We can specify the order, from the lowest to the highest with order = TRUE and highest to lowest with order = FALSE.

Example: We can use summary to count the values for each factor.

```
# Create Ordinal categorical vector
day_vector <- c('evening', 'morning', 'afternoon', 'midday', 'midnight', 'evening')
# Convert `day_vector` to a factor with ordered level
factor_day <- factor(day_vector, order = TRUE, levels =c('morning', 'midday', 'afternoon', 'evening', 'midnight'))
# Print the new variable
factor_day
```

Output:

```
## [1] evening morning afternoon midday midnight evening
```

Example:

```
## Levels: morning < midday < afternoon < evening < midnight
# Append the line to above code
# Count the number of occurrence of each level summary(factor_day)
```

Output:

```
## morning midday afternoon evening midnight
## 1 1 1 2 1
```

Continuous Variables

- Continuous class variables are the default value in R.
- They are stored as numeric or integer.
- We can see it from the dataset below. mtcars is a built-in dataset. It gathers information on different types of car. We can import it by using mtcars and check the class of the variable mpg, mile per gallon. It returns a numeric value, indicating a continuous variable.

```
dataset <- mtcars
class(dataset$mpg)
## [1]   “numeric”
```

Lists in R:

- A list is a generic object consisting of an ordered collection of objects.
- Lists are heterogeneous data structures.
- A list is an one-dimensional data structures.
- A list can be a list of vectors, list of matrices, a list of characters and a list of functions and so on.
- Lists are different from atomic vectors because their elements can be of any type, including lists.

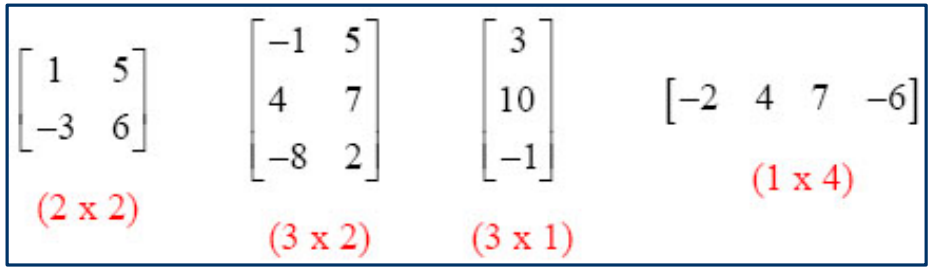
- You construct lists by using list() instead of c():

```
x <- list(1:5, “d", c(FALSE, FALSE, TRUE), c(3.5 7.2))
str(x)
```

```
# Example - Illustrate a List

# The first attributes is a numeric vector
# containing the employee IDs which is
# created using the 'c' command here
empId = c(1, 2, 3, 4)

# The second attribute is the employee name
# which is created using this line of code here
# which is the character vector
empName = c(“Ann", “Sanjan", “Ellison", “Evette")

# The third attribute is the number of employees
# which is a single numeric variable.
numberOfEmp = 4

# We can combine all these three different
# data types into a list
# containing the details of employees
# which can be done using a list command
empList = list(empId, empName, numberOfEmp)

print(empList)
```

Matrix – What is a Matrix?

- A matrix is a 2-dimensional array that has m number of rows and n number of columns. In other words, matrix is a combination of two or more vectors with the same data type.



- You can create a matrix with the function matrix(). This function takes three arguments:
  matrix(data, nrow, ncol, byrow = FALSE)

Arguments:

- data: The collection of elements that R will arrange into the rows and columns of the matrix.
- nrow: Number of rows
- ncol: Number of columns
- byrow: The rows are filled from the left to the right. We use `byrow = FALSE` (default values), if we want the matrix to be filled by the columns i.e. the values are filled top to bottom.

#Print dimension of the matrix with dim()

Defining names of columns and rows in a matrix

```
# Print dimension of the matrix with dim()
  dim(matrix_a)
```

In order to define rows and column names, you can create two vectors of different names, one for row and other for a column. Then, using the Dimnames attribute, you can name them appropriately:

```
rows = c("row1", "row2", "row3", "row4")     #Creating our character vector of row names

cols = c("coln1", "coln2", "coln3")          #Creating our character vector of column names

mat <- matrix(c(4:15), nrow = 4, byrow = TRUE, dimnames = list(rows, cols) )
#creating our matrix mat and assigning our vectors to dimnames
```

```
# Print matrix
  print(mat)
```

Add a Column to a Matrix with the cbind()

- You can add a column to a matrix with the cbind() command.
- cbind() means column binding. cbind() can concatenate as many matrix or columns as specified.

Example:

```
# concatenate c(1:5) to the matrix_a
matrix_a1 <- cbind(matrix_a, c(1:5))
# Check the dimension dim(matrix_a1)
```

Output:

```
## [1] 5 3
```

Example:

```
matrix_a1
```

Output

```
##      [,1] [,2] [,3]
## [1,]    1    2    1
## [2,]    3    4    2
## [3,]    5    6    3
## [4,]    7    8    4
## [5,]    9   10    5
```

Example:
We can also add more than one column. Add the next sequence of number to the matrix_a2 matrix. The dimension of the new matrix will be 4x6 with number from 1 to 24.

```
matrix_a2 <-matrix(13:24, byrow = FALSE, ncol = 3)

##      [,1] [,2] [,3]
## [1,]   13   17   21
## [2,]   14   18   22
## [3,]   15   19   23
## [4,]   16   20   24
```

Example:

```
matrix_c <-matrix(1:12, byrow = FALSE, ncol = 3)
matrix_d <- cbind(matrix_a2, matrix_c)
dim(matrix_d)
```

Output:

```
## [1] 4 6
```

Slice a Matrix: Accessing individual components

We can select elements one or many elements from a matrix by using the square brackets [ ]. This is where slicing comes into the picture.

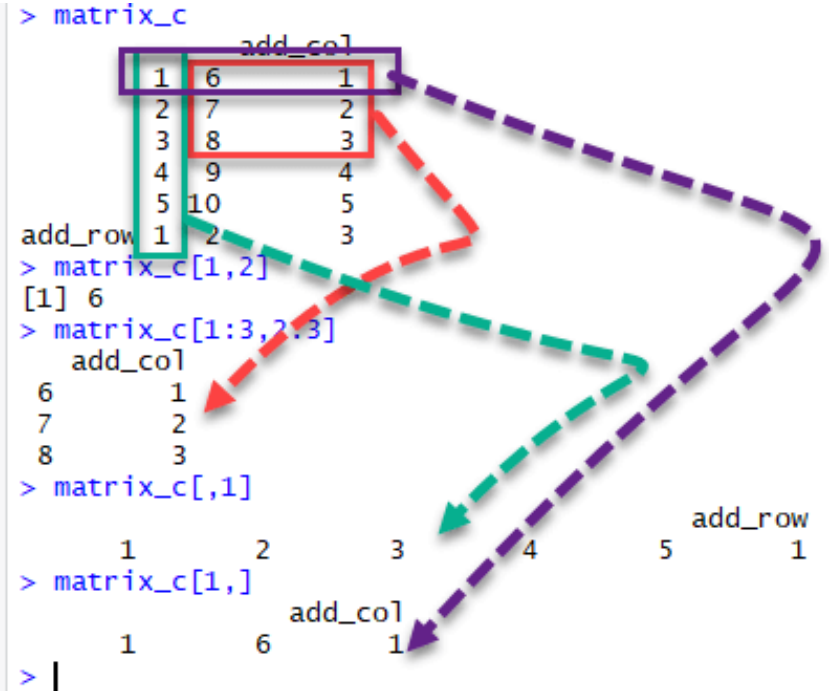
For example:

matrix_c[1,2] selects the element at the first row and second column.

matrix_c[1:3,2:3] results in a matrix with the data on the rows 1, 2, 3 and columns 2, 3,

matrix_c[,1] selects all elements of the first column.

matrix_c[1,] selects all elements of the first row.



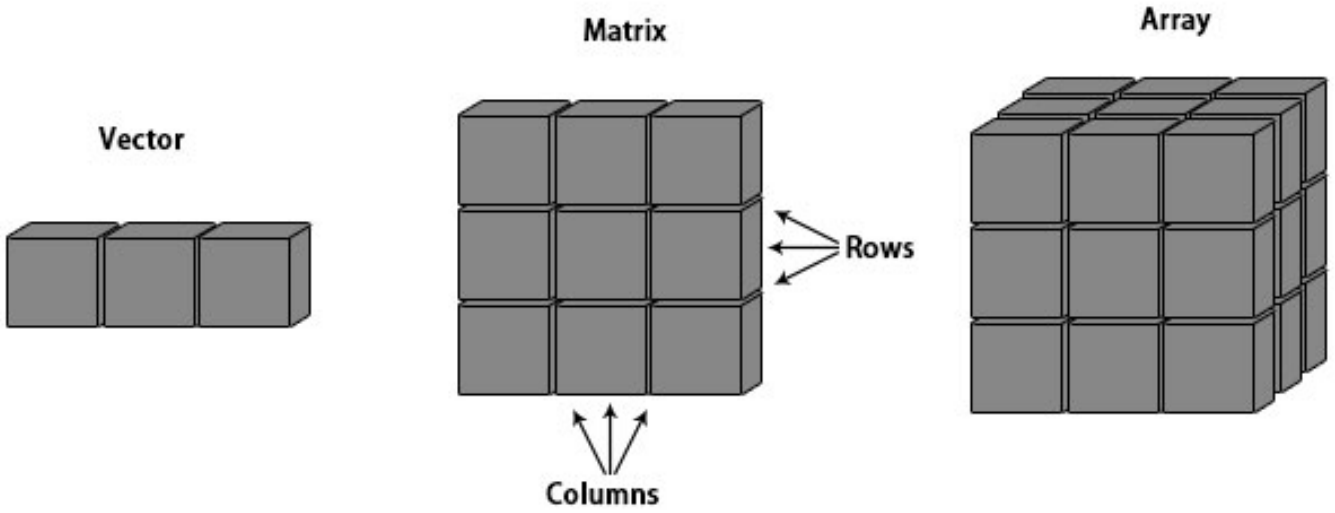
```
# Examples:
> A = matrix( +   c(2, 4, 3, 1, 5, 7),      # the data elements
                  +   nrow=2,               # number of rows
                  +   ncol=3,               # number of columns
                  +   byrow = TRUE)         # fill matrix by rows
> A


> A[2, 3]                                   # element at 2nd row, 3rd column
> A[2, ]                                    # the 2nd row
> A[ ,c(1,3)]                               # the 1st and 3rd columns
> t(A)                                      # transpose of B
> c(B)                                      # deconstruct a matrix
```

Functions for viewing matrices and dataframes and returning information about them.

Function	Description
head(x), tail(x)	Print the first few rows (or last few rows).
View(x)	Open the entire object in a new window
nrow(x), ncol(x), dim(x)	Count the number of rows and columns
rownames(), colnames(), names()	Show the row (or column) names
str(x), summary(x)	Show the structure of the dataframe (ie., dimensions and classes) and summary statistics

Introduction to Arrays in R

- In arrays, data is stored in the form of matrices, rows, and columns.



R Array Syntax

Array_NAME <- array(data, dim = (row_Size, column_Size, matrices, dimnames)

- **data –** Data is an input vector that is given to the array.
- **matrices –** Array in R consists of multi-dimensional matrices.
- **row_Size –** row_Size describes the number of row elements that an array can store.
- **column_Size –** Number of column elements that can be stored in an array.
- **dimnames –** Used to change the default names of rows and columns to the user's preference.

Arguments in Array

The array function in R can be written as:

array(data = NA, dim = length(data), dimname = NULL)

- **data** is a vector that provides data to fill the array.
- **dim** attribute provides maximum indices in each dimension
- **dimname** can be either NULL or can have a name for the array.

How to Create an Array in R

```
# Example:

# Create two vectors of different lengths.
vector1 <- c(2,8,3)
vector2 <- c(10,14,17,12,11,15)



# Take these vectors as input to the array.
result <- array(c(vector1,vector2), dim = c(3,3,2))

print(result)
```

Different Operations on Rows and Columns

Naming Columns And Rows

```
# Example:
# Create two vectors of different lengths.
vector1 <- c(2,8,3)
vector2 <- c(10,14,17,12,11,15)
column.names <- c("COL1","COL2","COL3")
row.names <- c("ROW1","ROW2","ROW3")
matrix.names <- c("Matrix1","Matrix2")


# Take these vectors as input to the array.
result <- array(c(vector1,vector2),dim = c(3,3,2), dimnames = list(row.names,
column.names,  matrix.names))


print(result)
```

```
# Print the third row of the first matrix of the array.
print(result[3,,1])

# Print the element in the 2nd row and 3rd column of the 2nd matrix.
print(result[2,3,2])

# Print the 2nd Matrix.
print(result[,,2])
```

Manipulating R Array Elements

```
# Example:

# Create two vectors of different lengths.
vector1 <- c(1,4,6)
vector2 <- c(2,3,5,6,7,9)

# Take these vectors as input to the array.
array1 <- array(c(vector1,vector2), dim = c(3,3,2))

# Create two vectors of different lengths.
vector3 <- c(6,4,1)
vector4 <- c(9,7,6,5,3,2)
array2 <- array(c(vector1,vector2), dim = c(3,3,2))
```

```
# create matrices from these arrays.
matrix1 <- array1[,,2]
matrix2 <- array2[,,2]

# Add the matrices.
result <- matrix1+matrix2

print(result)
```

Calculations across R Array Elements

- apply()function for calculations in an array in R.
- Syntax apply(x, margin, fun)

Following is the description of the parameters used:

- x is an array.
- a margin is the name of the dataset used.
- fun is the function to be applied to the elements of the array.
- For Example:

- We use the apply() function below in different ways. To calculate the sum of the elements in the rows of an array across all the matrices.

```
# Example:
# We will create two vectors of different lengths.

vector1 <- c(1,4,6)
vector2 <- c(2,3,5,6,7,9)

# Now, we will take these vectors as input to the array.
new.array <- array(c(vector1,vector2),dim = c(3,3,2))
print(new.array)

# Use apply to calculate the sum of the rows across all the matrices.
result <- apply(new.array, c(1), sum)
print(result)
```

R Data Frames:

- Data frames combine the behavior of lists and matrices to make a structure ideally suited for the needs of statistical data. The data frame is a list of vectors which are of equal length.
- A data-frame must have column names and every row should have a unique name.
  - names(), colnames(), and rownames()
- Each column must have the identical number of items.
- Each item in a single column must be of the same data type.
- Different columns may have different data types.
- A matrix contains only one type of data, while a data frame accepts different data types (numeric, character, factor, etc.). This makes it a 2-dimensional structure, so it shares properties of both the matrix and the list.

How to Create a Data Frame

- We can create a data frame by passing the variable a,b,c,d into the data.frame() function. We can name the columns with name() and simply specify the name of the variables.

- data.frame(df, stringsAsFactors = TRUE)

Arguments:

- df: It can be a matrix to convert as a data frame or a collection of variables to join
- stringsAsFactors: Convert string to factor by default

```r
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
str(df)

#> 'data.frame': 3 obs. of 2 variables:
#> $ x: int 1 2 3
#> $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

```r
df <- data.frame(
        x = 1:3,
        y = c("a", "b", "c"),
        stringsAsFactors = FALSE)
str(df)

#> 'data.frame': 3 obs. of 2 variables:
#> $ x: int 1 2 3
#> $ y: chr "a" "b" "c"
```

```
# Example R program to illustrate dataframe

# A vector which is a character vector
Name = c(“Auriel", "Ray", "Asia")

# A vector which is a character vector
Type = c(“O+", “B-”, “A-")

# A vector which is a numeric vector
Age = c(36, 23, 62)

# To create dataframe use data.frame command
# and then pass each of the vectors
# we have created as arguments
# to the function data.frame()

df = data.frame(Name, Language, Age)

print(df)
```

Because a data.frame is an S3 class, its type reflects the underlying vector used to build it: the list. To check if an object is a data frame, use class() or test explicitly with is.data.frame():

```
typeof(df)
#> [1] "list"
class(df)
#> [1] "data.frame"
is.data.frame(df)
#> [1] TRUE
```

You can coerce an object to a data frame with as.data.frame():

- A vector will create a one-column data frame.
- A list will create one column for each element; it's an error if they're not all the same length.
- A matrix will create a data frame with the same number of columns and rows as the matrix.

Merging: Combine data frames using cbind() and rbind()

```
cbind(df, data.frame(z = 3:1))
#> x y z
#> 1 1 a 3
#> 2 2 b 2
#> 3 3 c 1

rbind(df, data.frame(x = 10, y = "z"))
#> x y
#> 1 1 a
#> 2 2 b
#> 3 3 c
#> 4 10 z
```

- When combining column-wise, the number of rows must match, but row names are ignored.
- When combining row-wise, both the number and names of columns must match.
- Use plyr::rbind.fill() to combine data frames that don't have the same columns.

It’s a common mistake to try and create a data frame by cbind()ing vectors together. This doesn’t work because cbind() will create a matrix unless one of the arguments is already a data frame. Instead use data.frame() directly:

```
bad <- data.frame(cbind(a = 1:2, b = c("a", "b")))
str(bad)
#> 'data.frame': 2 obs. of 2 variables:
#> $ a: Factor w/ 2 levels "1","2": 1 2
#> $ b: Factor w/ 2 levels "a","b": 1 2

good <- data.frame(a = 1:2, b = c("a", "b"), stringsAsFactors = FALSE)
str(good)
#> 'data.frame': 2 obs. of 2 variables:
#> $ a: int 1 2
#> $ b: chr "a" "b"
```

Example:

Let's create a factor data frame.

```
# Create gender vector
gender_vector <- c("Male", "Female", "Female", "Male", "Male")
class(gender_vector)

# Convert gender_vector to a factor
factor_gender_vector <-factor(gender_vector)
class(factor_gender_vector)
```

Output:

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

Knowledge Check:

1. What are the three properties of a vector, other than its contents?
   The three properties of a vector are type, length, and attributes.

2a. What are the four common types of atomic vectors?
   1. Logical
   2. Integer
   3. Double (sometimes called numeric)
   4. Character

2b. What are the two rare types?
   1. Complex
   2. Raw

Knowledge Check (cont.)

3. What are attributes? How do you get them and set them?

Attributes allow you to associate arbitrary additional metadata to any object. You can get and set individual attributes with attr(x, "y") and attr(x, "y") <- value; or get and set all attributes at once with attributes().

4a. How is a list different from an atomic vector?

The elements of a list can be any type (even a list); the elements of an atomic vector are all of the same type.

4b. How is a matrix different from a data frame?

Similarly, every element of a matrix must be the same type; in a data frame, the different columns can have different types

5a. Can you have a list that is a matrix?

You can make “list-array” by assigning dimensions to a list.