Social Science Program
R Training

Instructor: Yvonne Phillips

Summer 2022

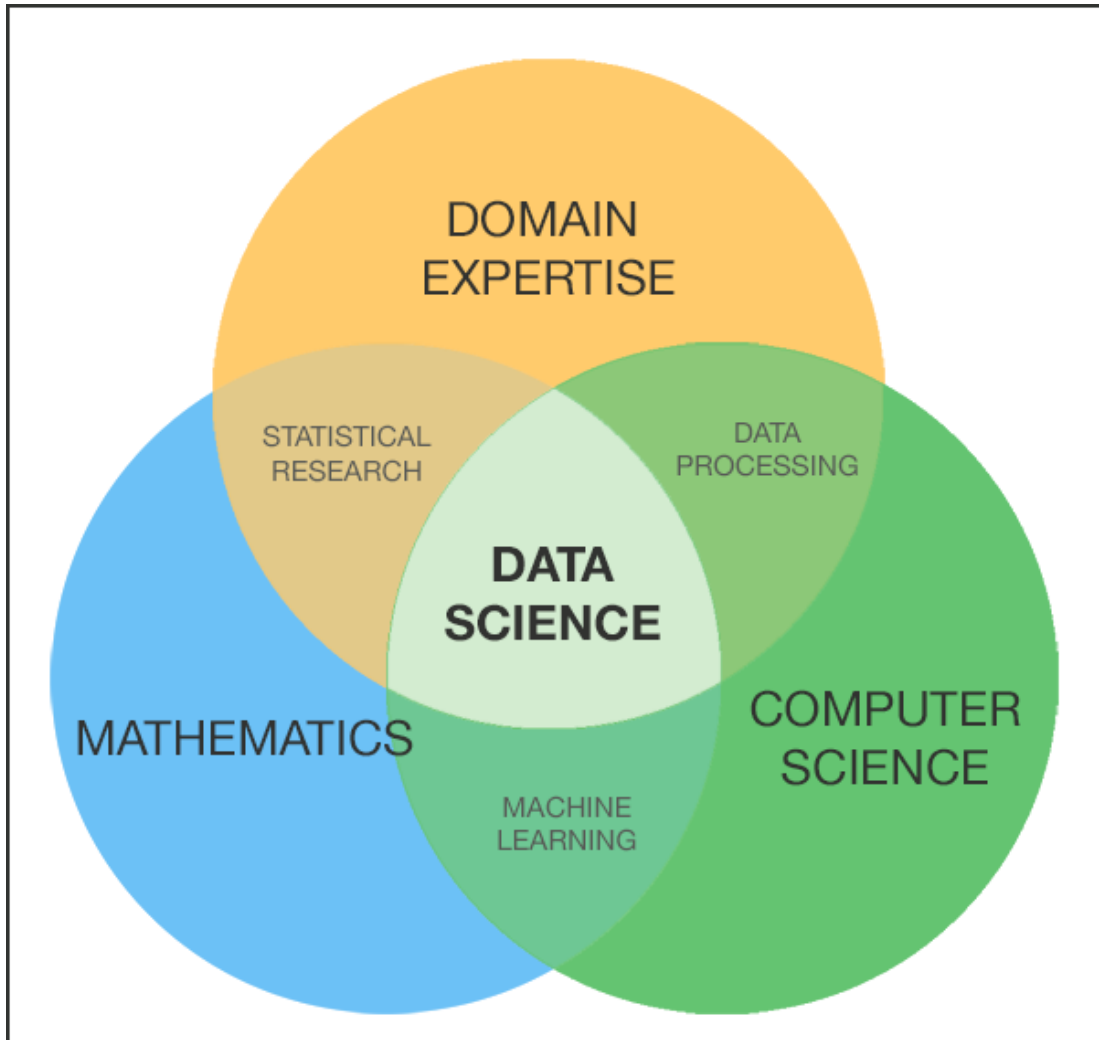
ANALYTICS ENLIGHTENED

Updated 05/10/2022

R is a programming language, particularly in the world of data analysis and data science. This is an introductory course to get you started with RStudio with hands-on examples. In this course, you will learn:

- How to use RStudio, a free and open-source development environment for R,
- Learn the fundamentals of R syntax,
- How to assign and manipulate variables,
- Learn the data types,
- Learn the data structures; vectors, matrices, lists, arrays, and data frames,
- Import data into RStudio,

What is Data Science?



Data Science is the science which uses computer science, statistics and machine learning, visualization and human-computer interactions to collect, clean, integrate, analyze, visualize, interact with data to create data products.

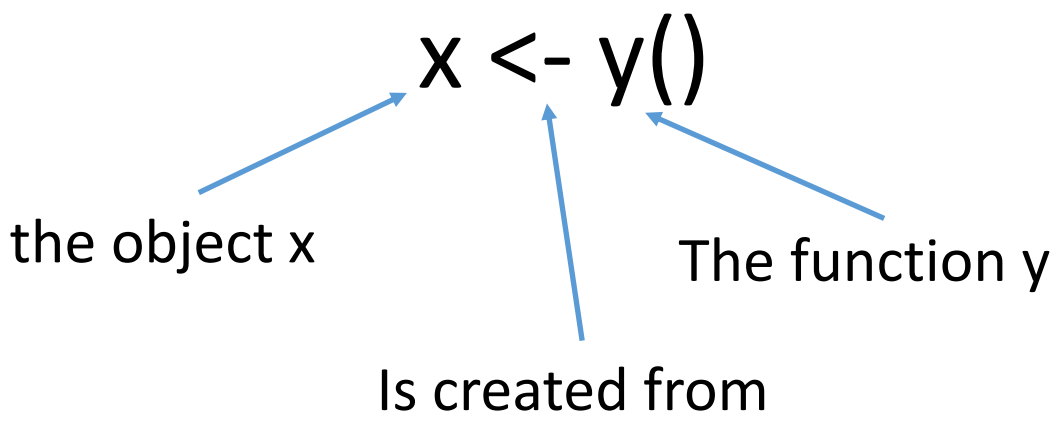
Data science is the extraction of actionable knowledge directly from data through a process of discovery, or hypothesis formulation and hypothesis testing.

Variable Assignment

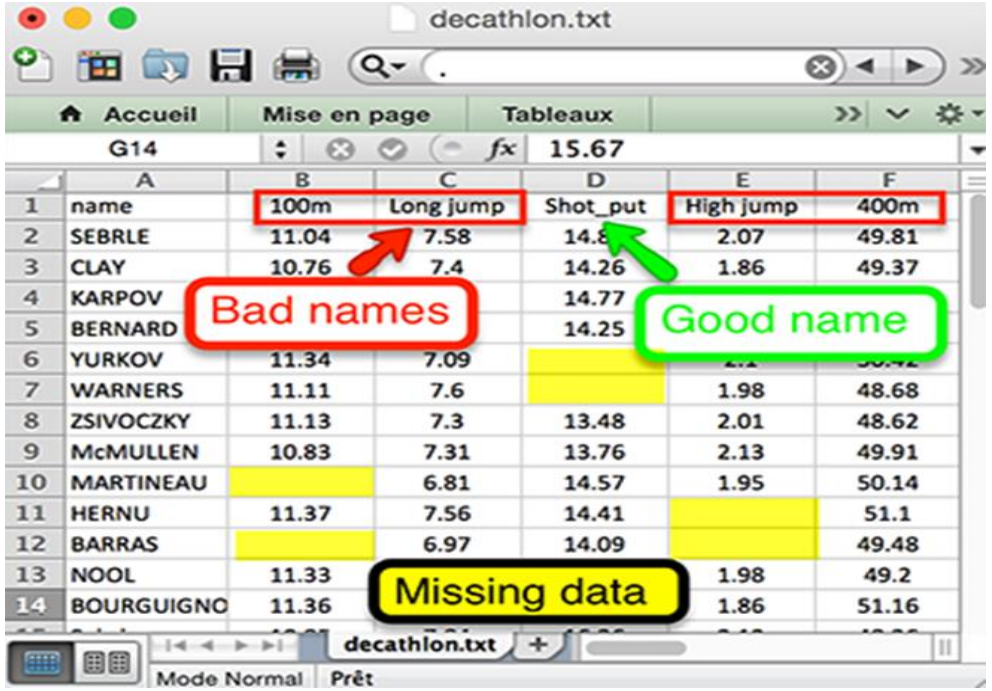
- Objects are named data structures inside of which you store data. An object can be a single digit, a character, a Boolean value, a sentence, a data frame, a list of data frames, a multi-dimensional structure, and so on. You use functions to create objects.
- To declare a variable, we need to assign 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
```



Variables

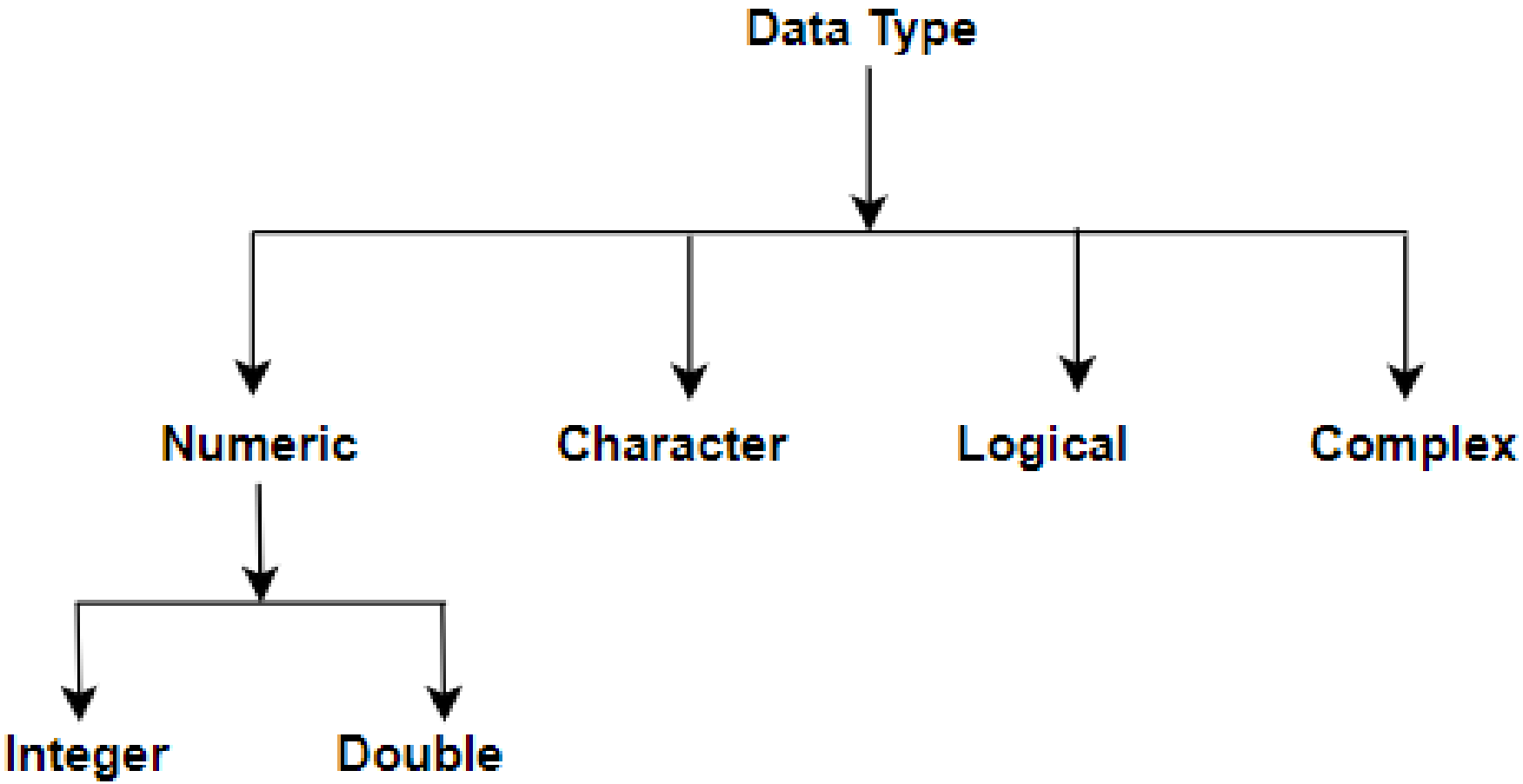


	name	100m	Long jump	Shot_put	High jump	400m
2	SEBRLE	11.04	7.58	14.8	2.07	49.81
3	CLAY	10.76	7.4	14.26	1.86	49.37
4	KARPOV			14.77		
5	BERNARD			14.25		
6	YURKOV	11.34	7.09			
7	WARNERS	11.11	7.6		1.98	48.68
8	ZSIVOCZKY	11.13	7.3	13.48	2.01	48.62
9	McMULLEN	10.83	7.31	13.76	2.13	49.91
10	MARTINEAU		6.81	14.57	1.95	50.14
11	HERNU	11.37	7.56	14.41		51.1
12	BARRAS		6.97	14.09		49.48
13	NOOL	11.33			1.98	49.2
14	BOURGUIGNO	11.36			1.86	51.16

- 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
- A valid variable name consists of letters, numbers and the dot or underline characters.
- The variable name starts with a letter, or the dot should not follow by a number.

Variable Name	**Validity**	**Reason**
var_name2.	valid	Has letters, numbers, dot and underscore
var_name%	Invalid	Has the character “%”. Only dot(.) and underscore allowed.
2var_name	invalid	Starts with a number
.var_name, var.name	valid	Can start with a dot(.) but the dot(.)should not be followed by a number.
.2var_name	invalid	The starting dot is followed by a number making it invalid.
_var_name	invalid	Starts with _ which is not valid

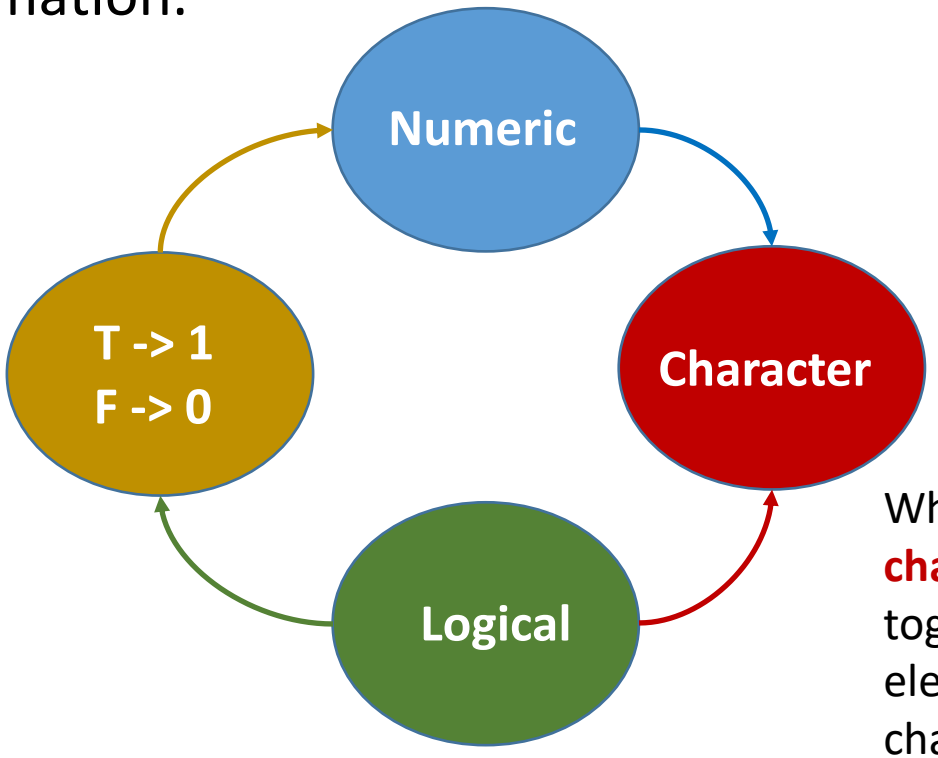
Data Types in R



Coercion rules

R has ways to prevent certain mistakes from happening. For example, if you are trying to create an object, and you are passing the wrong type of data as an argument, then R will convert the value to the correct type, so you can end up creating your object. The correct type is typically the simplest type necessary to represent all the information.

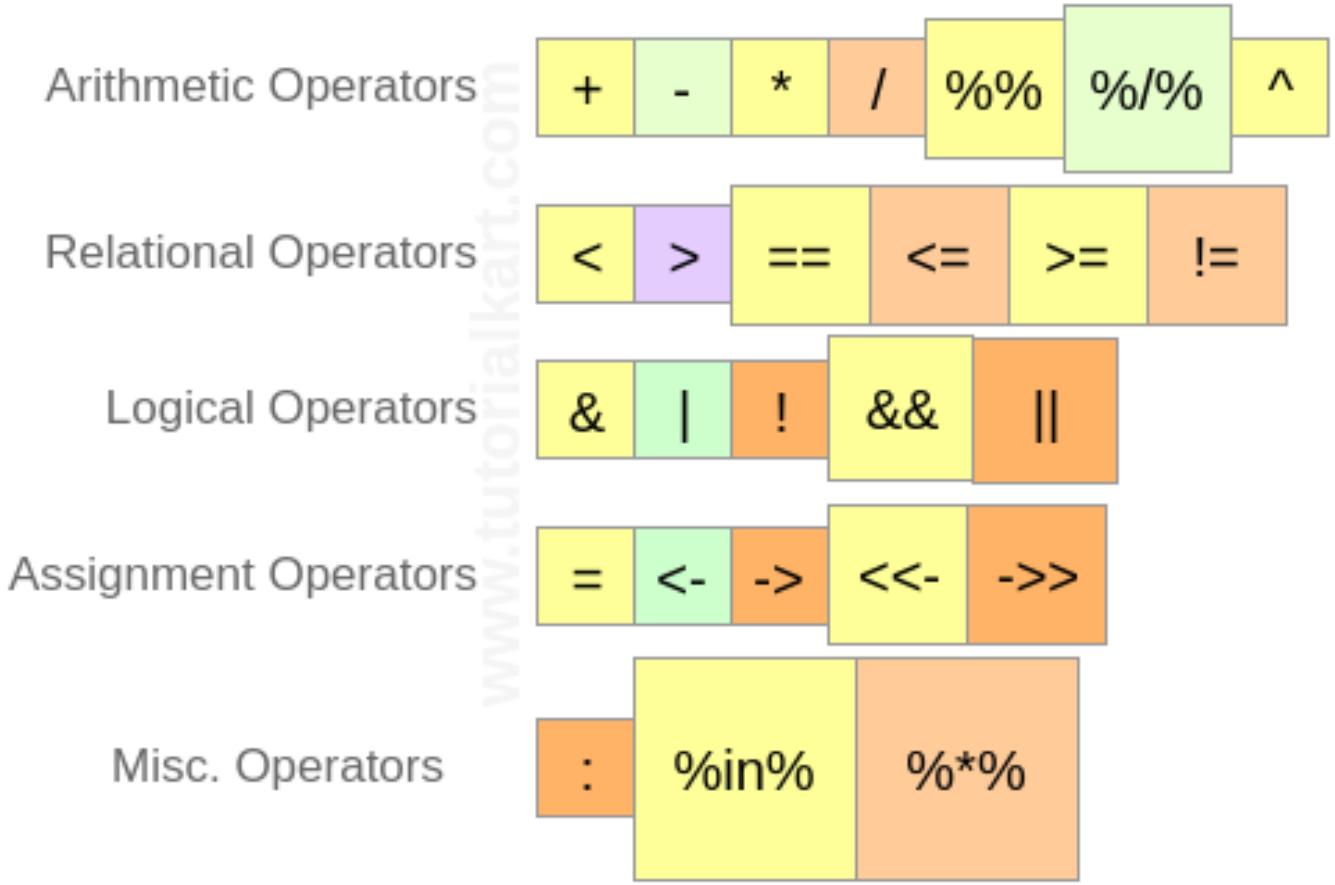
When a **numeric** and a **logical** element are present together, the logical value is converted to a numerical one.
TRUE is encoded as 1
FALSE is encoded as 0



When a **numeric** and a **character** value are present together, the numeric value is converted to a character

When a **logical** and a **character** value are present together, the logical element is coerced to a character.

R Operators: There are four main categories of Operators in R programming language. They are shown in the following picture :



Basic Arithmetic Operators

Arithmetic Operators are used to accomplish arithmetic operations. They can be operated on the basic data types Numericals, Integers, Complex Numbers. Vectors with these basic data types can also participate in arithmetic operations, during which the operation is performed on one-to-one element basis.

Operator	Description	Meaning in Formula	Usage
+	Addition	Add term	a+b
-	Subtraction	Remove or exclude term	a-b
*	Multiplication	Main effect and interactions	a*b
/	Division	Main effect and nesting	a/b
^ or **	Exponentiation	Limit depth of interactions	a^b
%%	Remainder	Amount left that does not entirely go into the divisor	a%%b
%/%	Quotient	quotient is the number of times a division is completed fully	a%/%b

Types

Converting between common data types in R. Can always go from a higher value in the table to a lower value.

`as.logical`	`TRUE, FALSE, TRUE`	Boolean values (TRUE or FALSE).
`as.numeric`	`1, 0, 1`	Integers or floating point numbers.
`as.character`	`'1', '0', '1'`	Character strings. Generally preferred to factors.
`as.factor`	`'1', '0', '1', levels: '1', '0'`	Character strings with preset levels. Needed for some statistical models.

Maths Functions

`log(x)`	Natural log.	`sum(x)`	Sum.
`exp(x)`	Exponential.	`mean(x)`	Mean.
`max(x)`	Largest element.	`median(x)`	Median.
`min(x)`	Smallest element.	`quantile(x)`	Percentage quantiles.
`round(x, n)`	Round to n decimal places.	`rank(x)`	Rank of elements.
`signif(x, n)`	Round to n significant figures.	`var(x)`	The variance.
`cor(x, y)`	Correlation.	`sd(x)`	The standard deviation.

Variable Assignment

```
> a <- 'apple'
> a
[1] 'apple'
```

The Environment

`ls()`	List all variables in the environment.
`rm(x)`	Remove x from the environment.
`rm(list = ls())`	Remove all variables from the environment.

You can use the environment panel in RStudio to browse variables in your environment.

Matrices

`m <- matrix(x, nrow = 3, ncol = 3)`
Create a matrix from x.


`m[2, ]` - Select a row

`m[ , 1]` - Select a column

`m[2, 3]` - Select an element

`t(m)`
Transpose

`m %*% n`
Matrix Multiplication

`solve(m, n)`
Find x in: m * x = n

Lists

`l <- list(x = 1:5, y = c('a', 'b'))`
A list is a collection of elements which can be of different types.

`l[[2]]`	`l[1]`	`l$x`	`l['y']`
Second element of l.	New list with only the first element.	Element named x.	New list with only element named y.

Also see the **dplyr** package.

Data Frames

`df <- data.frame(x = 1:3, y = c('a', 'b', 'c'))`
A special case of a list where all elements are the same length.

x	y
1	a
2	b
3	c

List subsetting


`df$x`


`df[[2]]`

Understanding a data frame

`View(df)` See the full data frame.

`head(df)` See the first 6 rows.

Matrix subsetting


`df[ , 2]`


`df[2, ]`

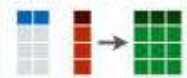

`df[2, 2]`

`nrow(df)`
Number of rows.

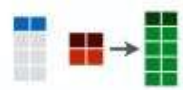
`ncol(df)`
Number of columns.

`dim(df)`
Number of columns and rows.

`cbind` - Bind columns.



`rbind` - Bind rows.



Strings

Also see the **stringr** package.

`paste(x, y, sep = ' ')`	Join multiple vectors together.
`paste(x, collapse = ' ')`	Join elements of a vector together.
`grep(pattern, x)`	Find regular expression matches in x.
`gsub(pattern, replace, x)`	Replace matches in x with a string.
`toupper(x)`	Convert to uppercase.
`tolower(x)`	Convert to lowercase.
`nchar(x)`	Number of characters in a string.

Factors

`factor(x)`
Turn a vector into a factor. Can set the levels of the factor and the order.

`cut(x, breaks = 4)`
Turn a numeric vector into a factor by 'cutting' into sections.

Statistics

`lm(y ~ x, data=df)`
Linear model.

`glm(y ~ x, data=df)`
Generalised linear model.

`summary`
Get more detailed information out a model.

`t.test(x, y)`
Perform a t-test for difference between means.

`pairwise.t.test`
Perform a t-test for paired data.

`prop.test`
Test for a difference between proportions.

`aov`
Analysis of variance.

Distributions

	Random Variates	Density Function	Cumulative Distribution	Quantile
Normal	`rnorm`	`dnorm`	`pnorm`	`qnorm`
Poisson	`rpois`	`dpois`	`ppois`	`qpois`
Binomial	`rbinom`	`dbinom`	`pbinom`	`qbinom`
Uniform	`runif`	`dunif`	`punif`	`qunif`

Plotting

Also see the **ggplot2** package.


`plot(x)`
Values of x in order.


`plot(x, y)`
Values of x against y.

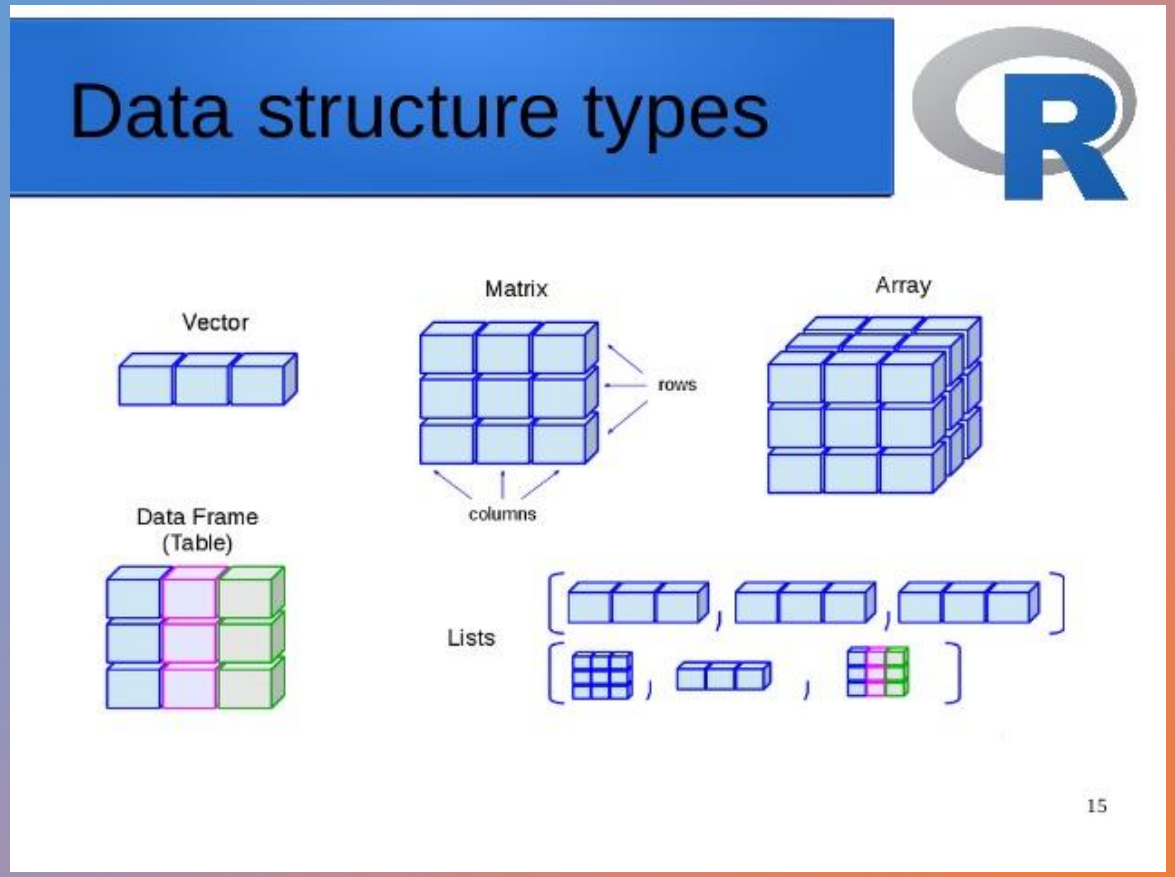

`hist(x)`
Histogram of x.

Dates

See the **lubridate** package.

Data Structures in

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



- ***Vector***: A sequence of numbers or characters, or higher-dimensional arrays like matrices.
- ***Matrix***: The basic two-dimensional data structure with rows and columns,
- ***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.
- An integer is a subclass of numeric.
- Cannot combine vectors of different modes

Creating Vectors: combine (c) and colon (:)

- Vectors can only contain entries of the same type: numeric or character; you can't mix them.
- The most basic way to create a vector is with ***c***(x1, . . . , xn), and it works for characters and numbers alike. Note that characters should be surrounded by " ".

```
> x <- c(1,2,3)              #numeric
[1] 1 2 3
> x <- c("a", "b", "c")      #character
[1] "a" "b" "c"
> typeof(x)
[1] "character"
> length(x)
[1] 3
```

- Use the **:** operator to create and define the vector variable.

```
> v <- (10:15)
> print(v)
[1] 10 11 12 13 14 15
```

Creating Vectors: sequence and replicate

You can also generate regular sequences:

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

```
> seq(from = 0, to = 6, by = 2)
[1] 0, 2, 4, 6
> seq(0, 1, length.out = 11) #desired length of the final sequence (only use if you don't specify by)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

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

```
> rep(c(7, 8), times = 2, each = 2)
[1] 7, 7, 8, 8, 7, 7, 8, 8
> rep(x = 3, times = 10)
[1] 3 3 3 3 3 3 3 3 3 3
> rep(x = 1:3, length.out = 10)
[1] 1 2 3 1 2 3 1 2 3 1
> rep("spades", 2)              #Rep can be used in replicating character string
[1] "spades" "spades"
```

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 <- (1:20)
> print (x)
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

> x + 1
 [1]  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21

> y <- rnorm(20)
> x*y

> c(1, 2, 3, 4, 5) + c(5, 4, 3, 2, 1)
[1] 6 6 6 6 6
```

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)	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(x)

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

Missing Data

Missing data in R appears as NA. NA is not a string or a numeric value, but an indicator of missingness. We can create vectors with missing values.

```
> x1 <- c(1, 5, 9, NA, 7)
> x2 <- c(“y", “D", NA, "NA")
```

NA is the one of the few non-numbers that we could include in **x1** without generating an error (and the other exceptions are letters representing numbers or numeric ideas like infinity).

In **x2**, the third value is missing while the fourth value is the character string “NA”.

To see which values in each of these vectors R recognizes as missing, we can use the **is.na** function. It will return a TRUE/FALSE vector with as any elements as the vector we provide.

```
is.na(x1)
## [1] FALSE FALSE FALSE  TRUE FALSE
is.na(x2)
## [1] FALSE FALSE  TRUE FALSE
```

Our missing value cannot be compared to 0 and none of our values can be compared to NA because NA is not assigned a value–it simply is or it isn't.

- *NA is used for all kinds of missing data*: In other packages, missing strings and missing numbers might be represented differently–empty quotations for strings, periods for numbers. In R, NA represents all types of missing data. We saw a small example of this in **x1** and **x2**. **x1** is a "numeric" object and **x2** is a "character" object.

- *Non-NA values cannot be interpreted as missing*: Other packages allow you to designate values as "system missing" so that these values will be interpreted in the analysis as missing. In R, you would need to explicitly change these values to NA. The **is.na** function can also be used to make such a change:

```
is.na(x1) <- which(x1 == 7)
x1
## [1]  1  5  9 NA NA
```

Selecting Vector Elements

You can select the indexing technique appropriate:

- Use square brackets [ ] to select vector elements by their position

  v[2] – second element of vector v

- Use negative indexes to exclude elements
- Use a vector of indexes to select multiple values.
- Use a logical vector to select elements based on a condition
- Use names to access named elements.

Relational Operators return values inside the vector based on logical conditions. Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value

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

Operator	Description	Meaning in Description	Usage
<	Less than	Is first operand less than second operand	a < b
>	Greater than	Is first operand greater than second operand	a > b
==	Exactly equal to	Is first operand equal to second operand	a == b
<=	Less than or equal to	Is first operand less than or equal to second operand	a <= b
>=	Greater than or equal to	Is first operand greater than or equal to second operand	a > = b
!=	Not equal to	Is first operand not equal to second operand	a!=b
!x	Not a	Is first operant not equal to itself	!a
a&b	a AND b	Is first operant and second operant	a&b
isTRUE(a)	Test if a is TRUE	Is first operant is TRUE	isTRUE(a)

Logical Operators

- These are applicable only to vectors of type logical, numeric or complex.
- All numbers greater than 1 are considered as logical value TRUE.
- Each element of the first vector is compared with the corresponding element of the second vector.
- The result of comparison is a Boolean value.

&	It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE.
|	It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if nay one the elements are TRUE.
!	It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value.

- The logical operator && and || considers only the first element of the vectors and gives a vector of single element as output.

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.

Lists in R:

- A list is a generic object consisting of an ordered collection of objects.
- Lists are heterogeneous data structures.
- A list is a one-dimensional data structure.
- 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.

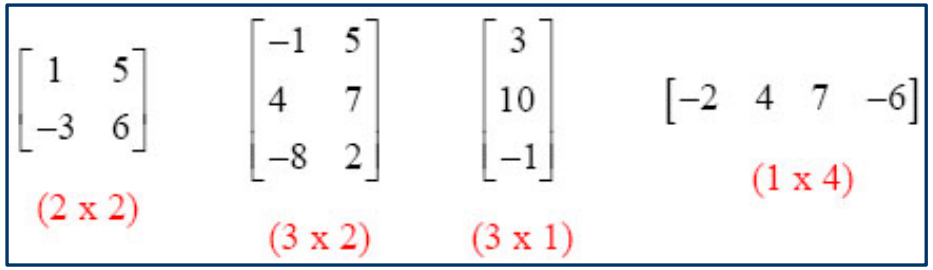
- You construct lists by using list() instead of c():

```
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's 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: When byrow='TRUE' the rows are filled from the left to the right. We use `byrow = FALSE`, if we want the matrix to be filled down 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

matrix_a <- matrix(c(1:12), nrow = 4, byrow = TRUE, dimnames = list(rows, cols) )
#creating our matrix mat and assigning our vectors to dimnames
```

```
# Print matrix
  print(matrix_a)
```

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(13:16) to the matrix_a
matrix_a1 <- cbind(matrix_a, c(13:16))
# Check the dimension dim(matrix_a1)

Output:
## [1] 4 4
```

Example:

```
matrix_a1

Output
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3   13
## [2,]    4    5    6   14
## [3,]    7    8    9   15
## [4,]   10   11   12   16
```

Syntax:

We can also add more than one column.

(matrix name)<-matrix(c(n:m), byrow = FALSE, ncol = (number of desired columns)), where n and m are integers. Byrow=FALSE populates down the column.

Example:

Add a sequence of numbers to the matrix_b matrix. The dimension of the new matrix will be 4x6 with number from 13 to 36.

```
matrix_b <-matrix(c(13:24))
##      [,1] [,2] [,3]
## [1,]   13   17   21
## [2,]   14   18   22
## [3,]   15   19   23
## [4,]   16   20   24
```

Example:

```
matrix_c <-matrix(c(25:36), byrow = FALSE, ncol = 3)
matrix_d <- cbind(matrix_b, matrix_c)
dim(matrix_d)
```

Output:

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

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   13   17   21   25   29   33
[2,]   14   18   22   26   30   34
[3,]   15   19   23   27   31   35
[4,]   16   20   24   28   32   36
```

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.

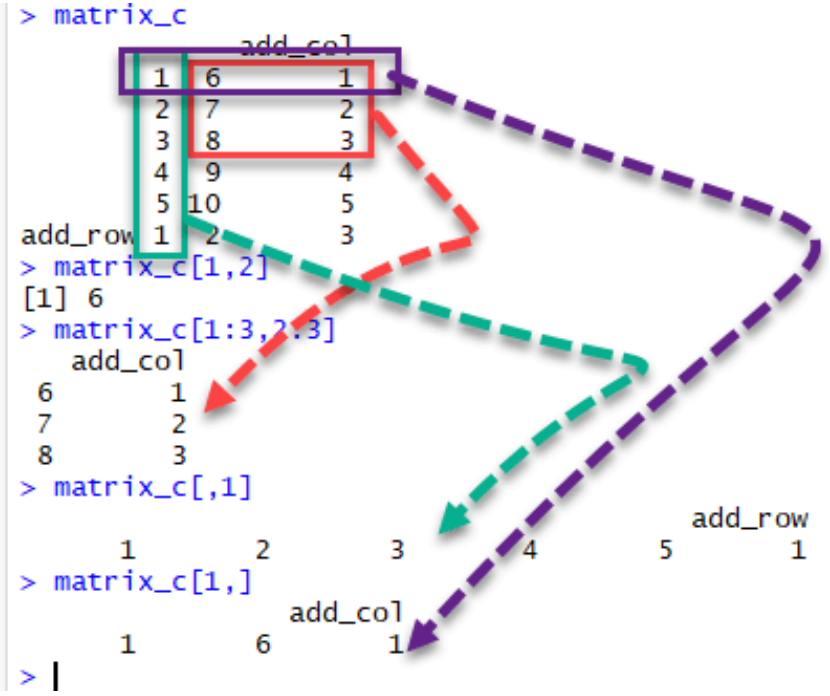
For example:

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.

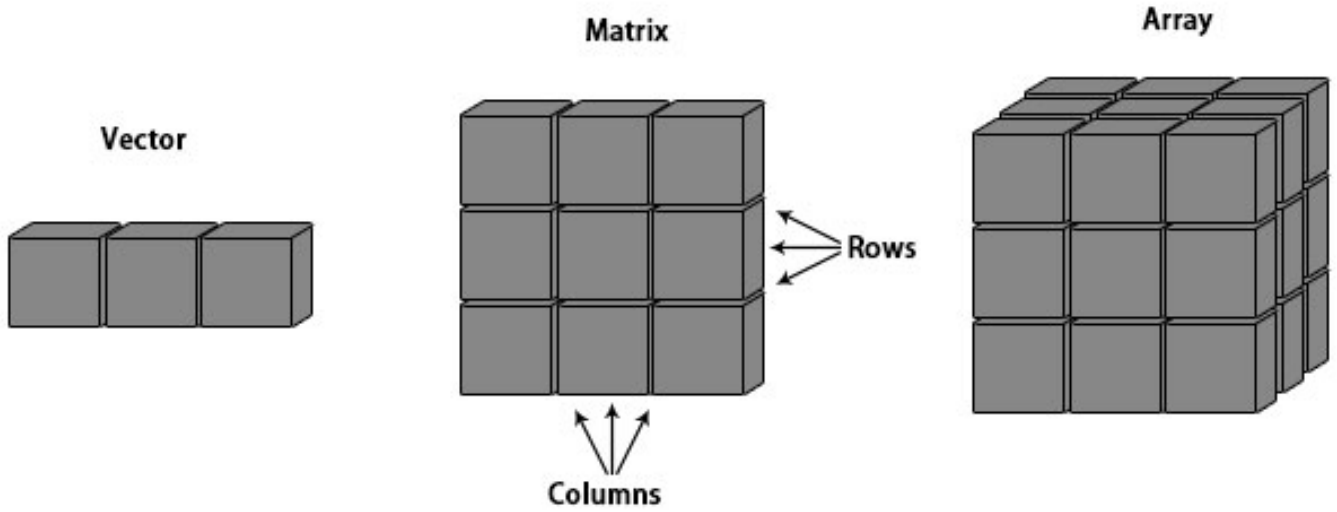


Example:

```
> A = matrix(c(2, 4, 3, 1, 5, 7),          # create a matrix and add 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

> B<- t(A)                                  # Use the t function to transpose of B

> c(B)                                      # Use the c function to deconstruct a matrix
```

Introduction to Arrays in R

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



•One dimensional array referred to as a vector.
•Two-dimensional array referred to as a matrix.

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 = FALSE)

Arguments:

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

- Note, versions prior to R 4.0.0, stringsAsFactors is set to default TRUE. See next slide

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.

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

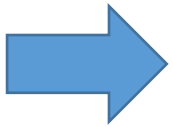
Function	Description
head(x), tail(x)	Print the first few rows (or last few rows)
View(x)	Open the entire object in a new window
nrow(x), ncol(x), dim(x)	Count the number of rows and columns
rownames(), colnames(), names()	Show the row (or column) names
str(x), summary(x)	Show the structure of the data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

How to access components of a data frame?

Assessing like a list: We can also access the components of the list of data frames using indices. To access the top-level components of a list of data frames we have to use a double slicing operator “[[ ]]” which is two square brackets and if we want to access the lower or inner level components of a list, we have to use another square bracket “[ ]” along with the double slicing operator “[[ ]]”..

Using the list, empList, example created earlier:

```
> empList
[[1]]
[1] 1 2 3 4

[[2]]
[1] "An" "Be" "Cm" "Di"

[[3]]
[1] 4
```



```
> empList[[2]]
[1] "An" "Be" "Cm" "Di“

> empList[[2]][2]
[1] "Be"
```

How to access components of a data frame?

Assessing like a matrix: Data frames can be accessed like a matrix by providing index for row and column. We can use either [, [[or $ operator to access columns of data frame.

```
#Extract the first column
> dfPatient[,1]
Name
1 Auriel
2 Ray
3 Asia

#Extract the second row
> dfPatient[2,]
  Name  Type  Age
2 Ray  B-  23

#Extract the first two rows and then all columns
> dfPatient[1:2,]
```

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: chr  "1" "2"
   $ b: chr  "a" "b"

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

	create	change to	check	get names	get dimensions
vector	`c, vector`	`as.vector`	`is.vector`	`names`	`length`
matrix	`matrix`	`as.matrix`	`is.matrix`	`rownames, colnames`	`dim, nrow, ncol`
array	`array`	`as.array`	`is.array`	`dimnames`	`dim`
list	`list`	`as.list`	`is.list`	`names`	`length`
data frame	`data.frame`	`as.data.frame`	`is.data.frame`	`names`	`dim, nrow, ncol`

Using packages

1

```
install.packages("readr")
```

Downloads files to computer

1 x per computer

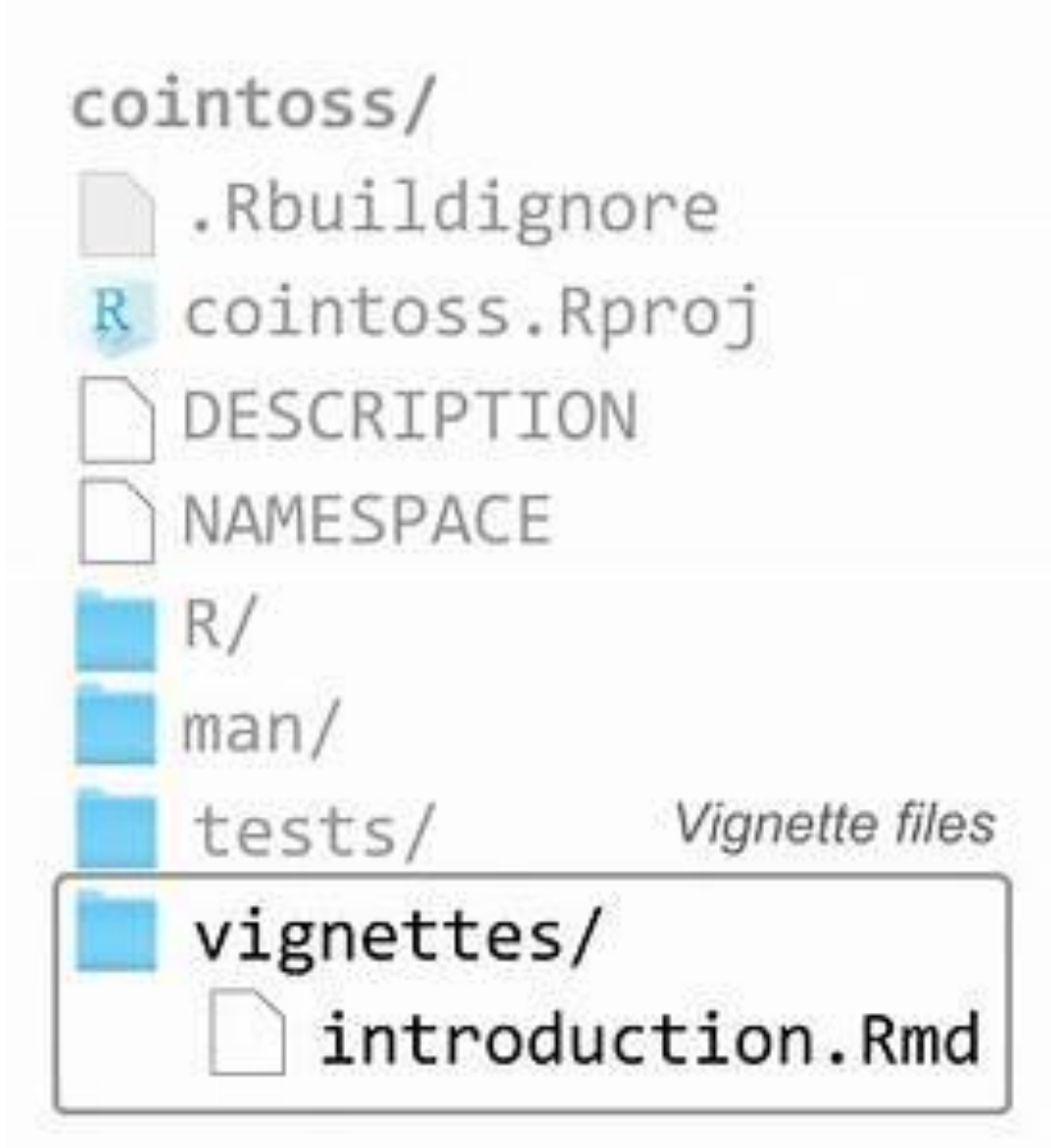
2

```
library("readr")
```

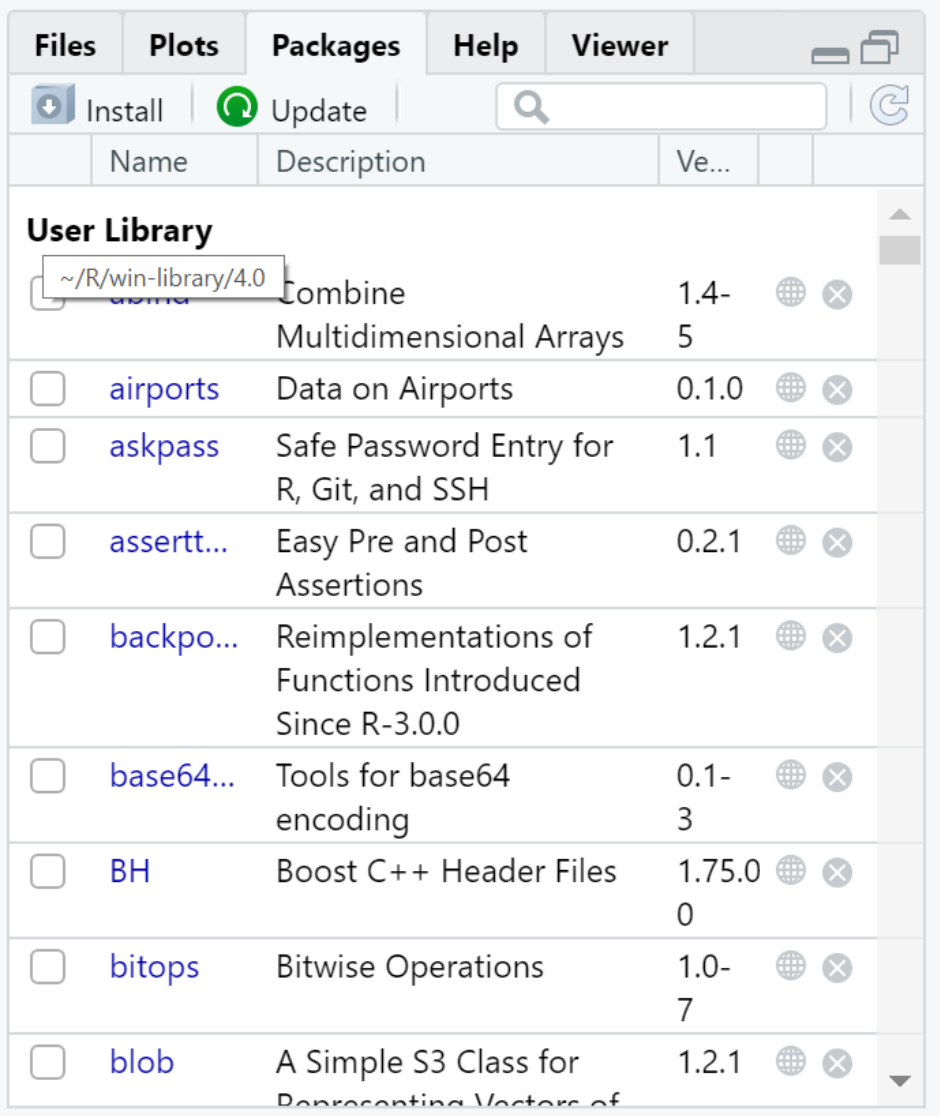
Loads package

1 x per R Session

Vignettes


cointoss/
.Rbuildignore
cointoss.Rproj
DESCRIPTION
NAMESPACE
R/
man/
tests/
vignettes/
introduction.Rmd
Vignette files

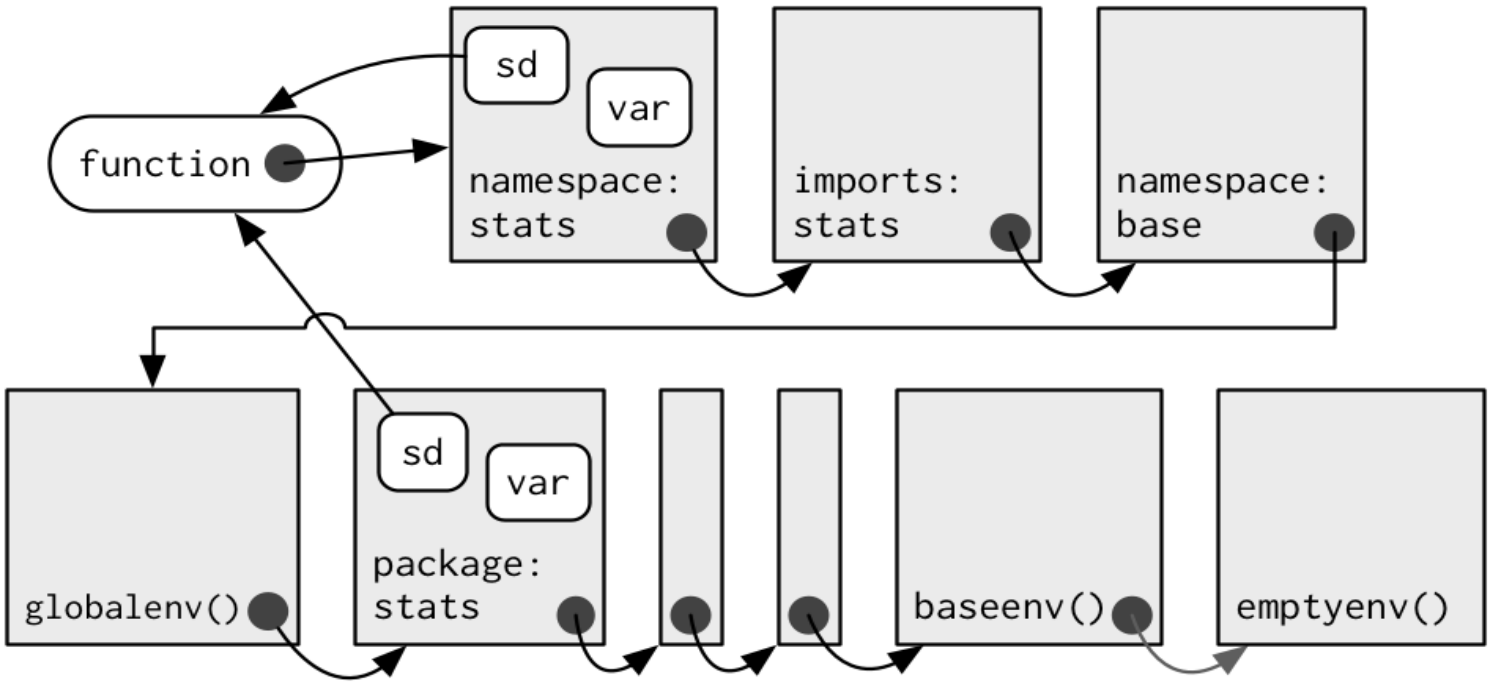
Packages



- **Preloaded packages**: R has a decent functionality before installing new packages. However, R's {base} packages are often outdated, or cumbersome to use.
- **Installing packages:** installing a package saves the package in our User Library for future use. A package needs to be installed only once.
- **Loading packages:** If you want to use the functions in a specific package, you must tell R to load it for you. You only need to load a package once per session. You do that with the library(package.name) function.
- **Distinguishing between functions:** some packages have functions whose names overlap. You can recognize which package a function belongs to by checking the {package.name} next to it during autocompletion.
- **Removing packages:** you can uninstall a package with the remove.packages("package.name") function.

Search Path – list of packages currently loaded into the memory

```
> search( )
 [1] ".GlobalEnv"        "tools:rstudio"         "package:stats"
 [4] "package:graphics"  "package:grDevices"  "package:utils"
 [7] "package:datasets"  "package:methods"    "Autoloads"
[10] "package:base"
```



Downloading files

.sas7bdat

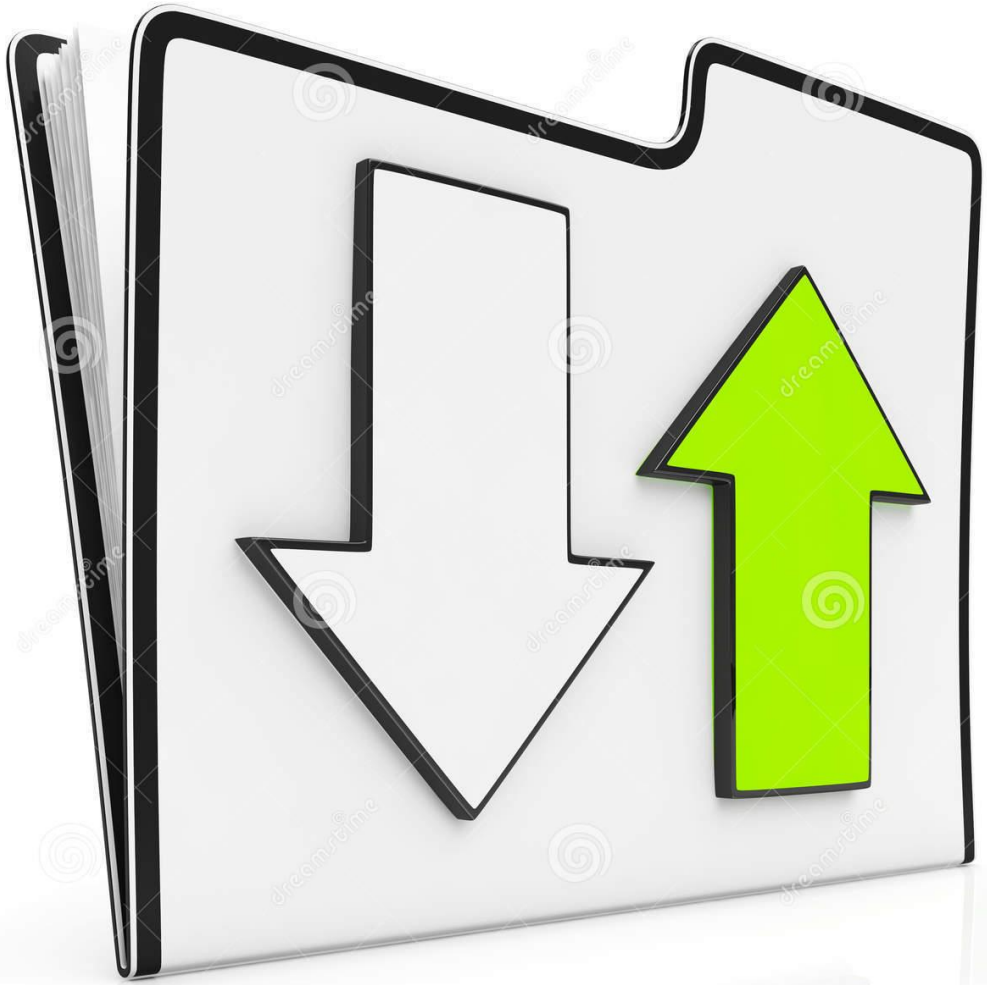
- library("haven")
- SASdata <- read.sas(“example.sas7bdat”)

.xlsx or .xls:

- library("readxl")
- xlsxdata <-read_excel("sheet1.xlsx"). You can add in a sheet argument.

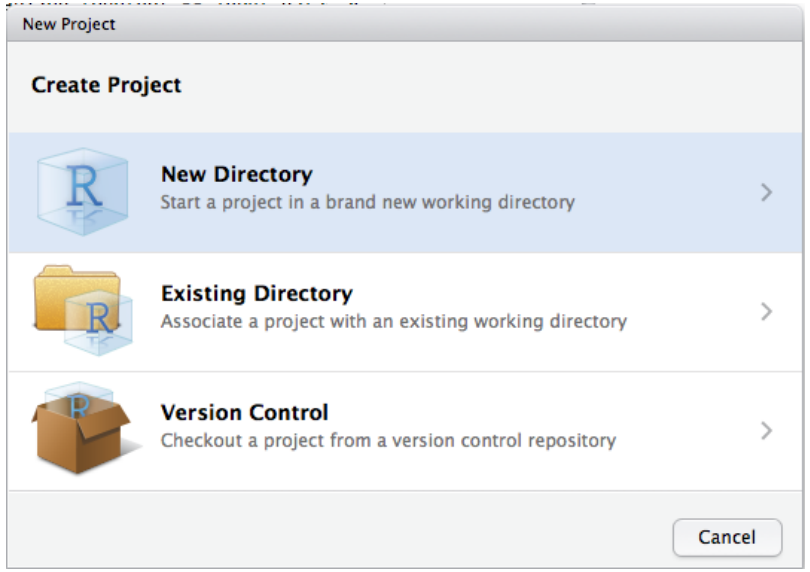
From the internet:

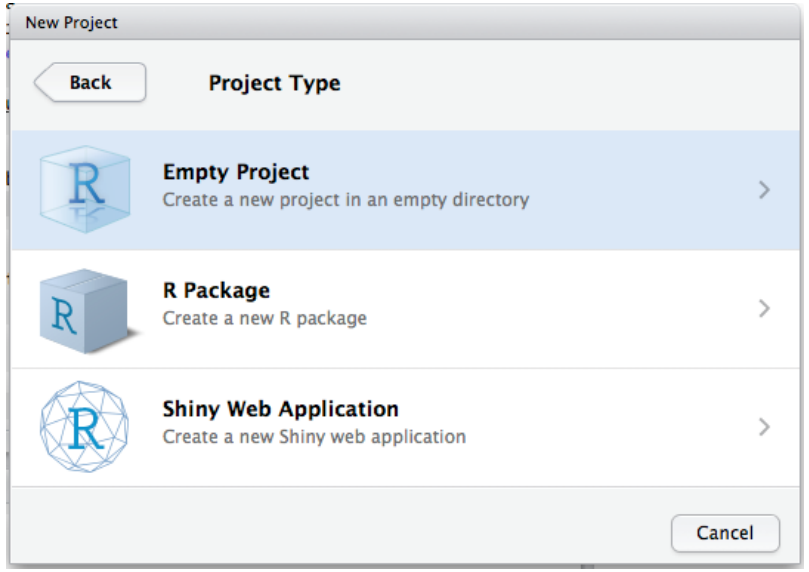
- download.file("http://www..xls", "C:/test/students.xls", method="auto", quiet=FALSE, mode = "wb", cacheOK = TRUE)

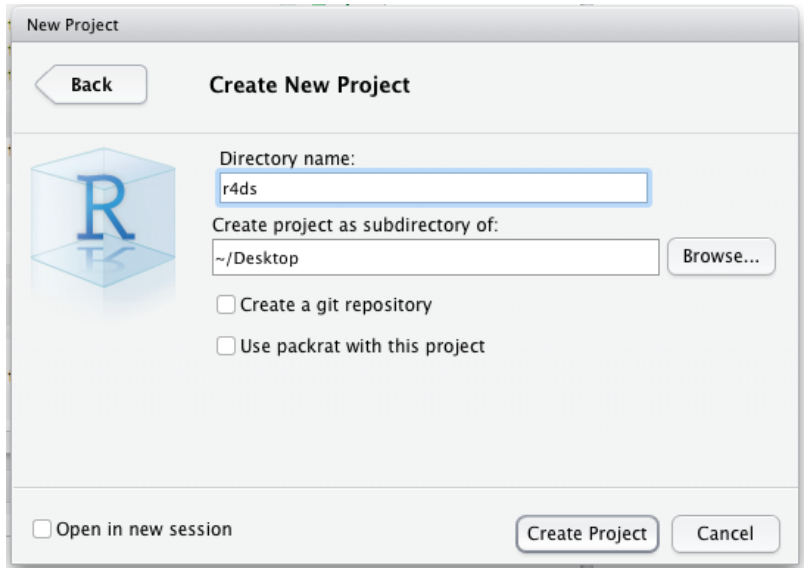


https://www.datacamp.com/community/tutorials/r-data-import-tutorial

R Projects: everything you need is in one place, and cleanly separated from all the other projects that you are working on.







How to Structure your Project Directory

Structure your project directory for efficient management and handling.

1. Data: store all the source files.

2. Script: store all the R scripts and all the files with extensions .Rmd and .R.
    - Files
    - Functions
    - Analysis

3. Output: store all the files you create in your projects such as HTML, plots, and exports.

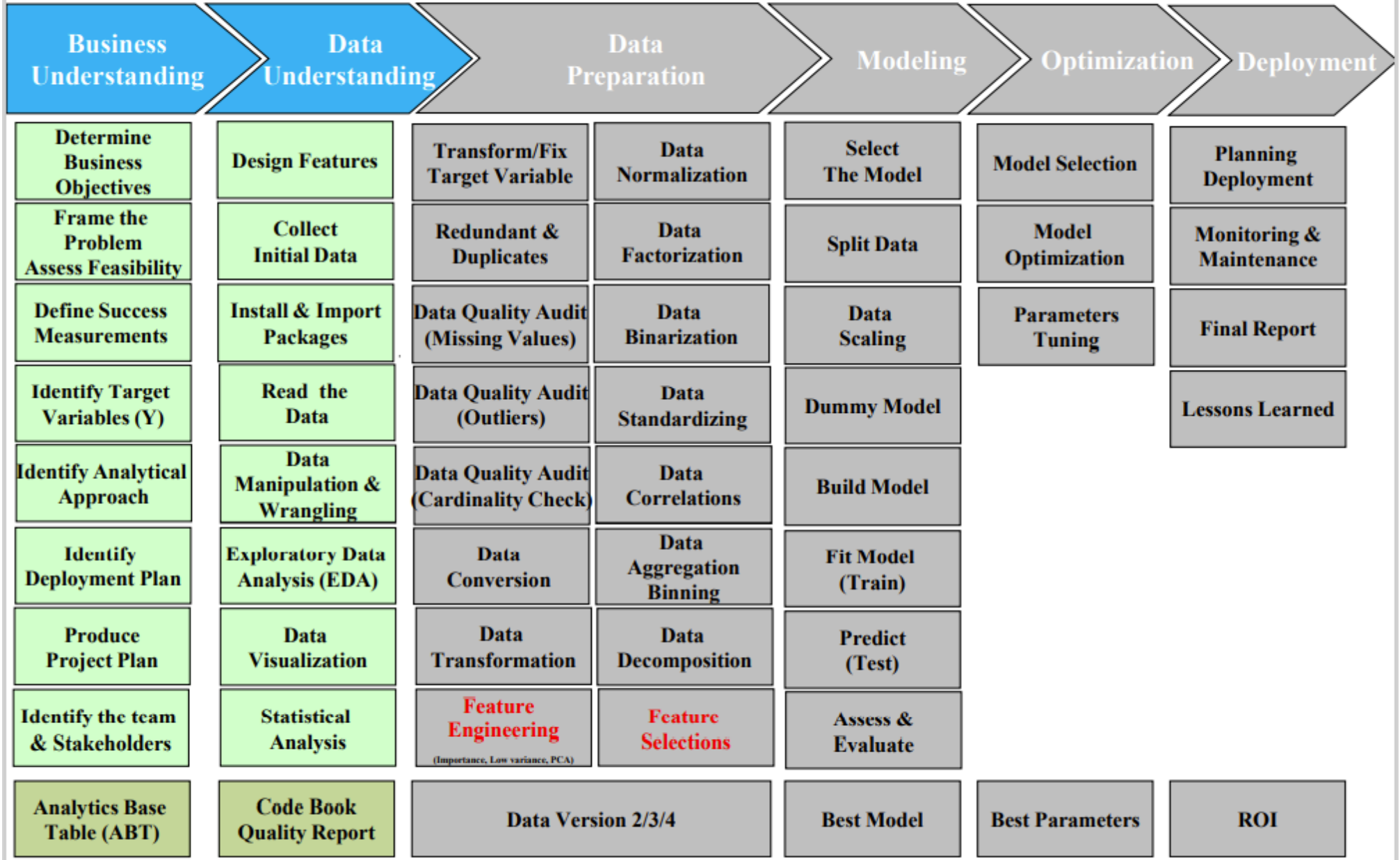
Additional resources for R programming basics

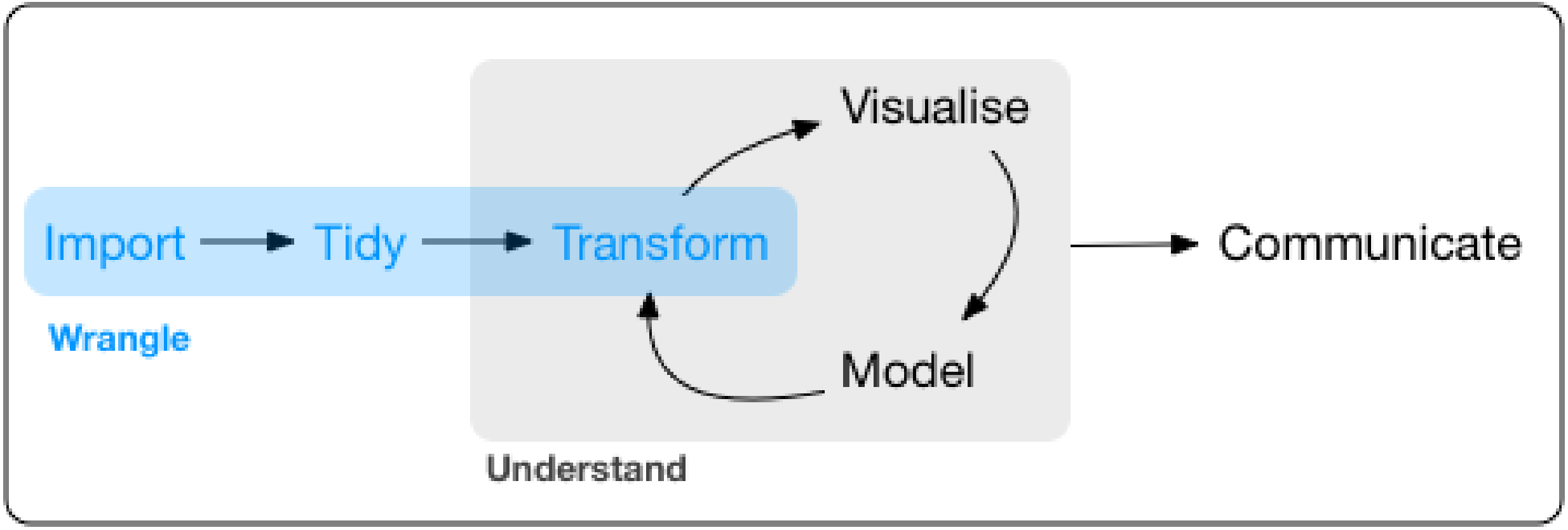
- Easy R Programming Basics: http://www.sthda.com/english/wiki/easy-r-programming-basics
- An Introduction to R: https://cran.r-project.org/doc/manuals/R-intro.pdf
- Packages available in CRAN: https://cran.r-project.org/web/packages/
- R for SAS and SPSS Users: https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxyNHN0YXRpc3RpY3N8Z3g6MWNmZDQ4ZjcwODY2Y2I0Yw

Data Wrangling: learning objectives of this module:

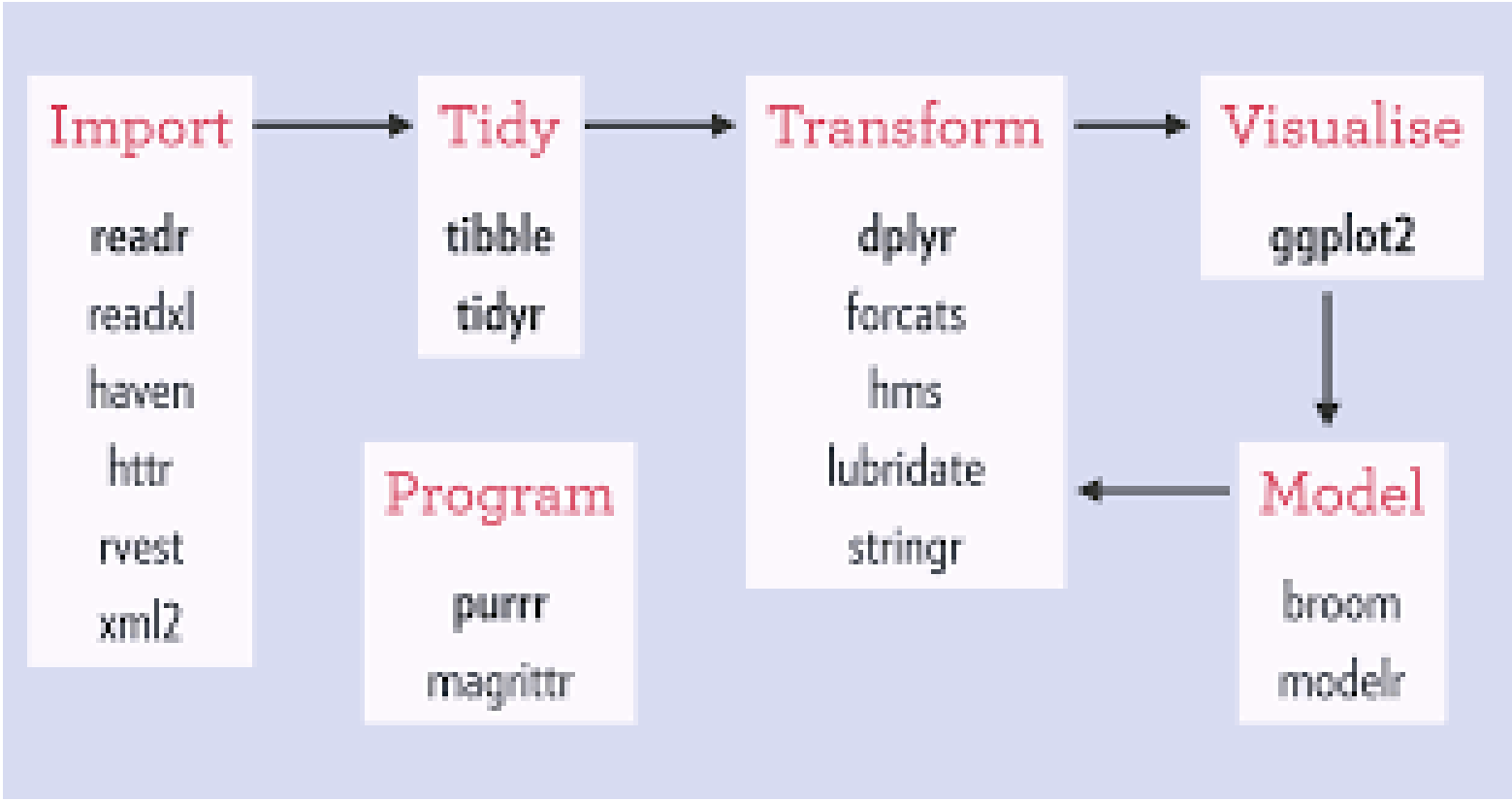
- Data Understanding
- Visit the tidyr package
- Exercise commands
- Introduction to data analysis
- Learn the basic vocabulary of dplyr

Data Scientist Day to Day Activities (CRISP-DM)




Visualise
Import
Tidy
Transform
Communicate
Model
Wrangle
Understand

https://www.tidyverse.org/




```
install.packages("tidyverse")

library("tidyverse")
```

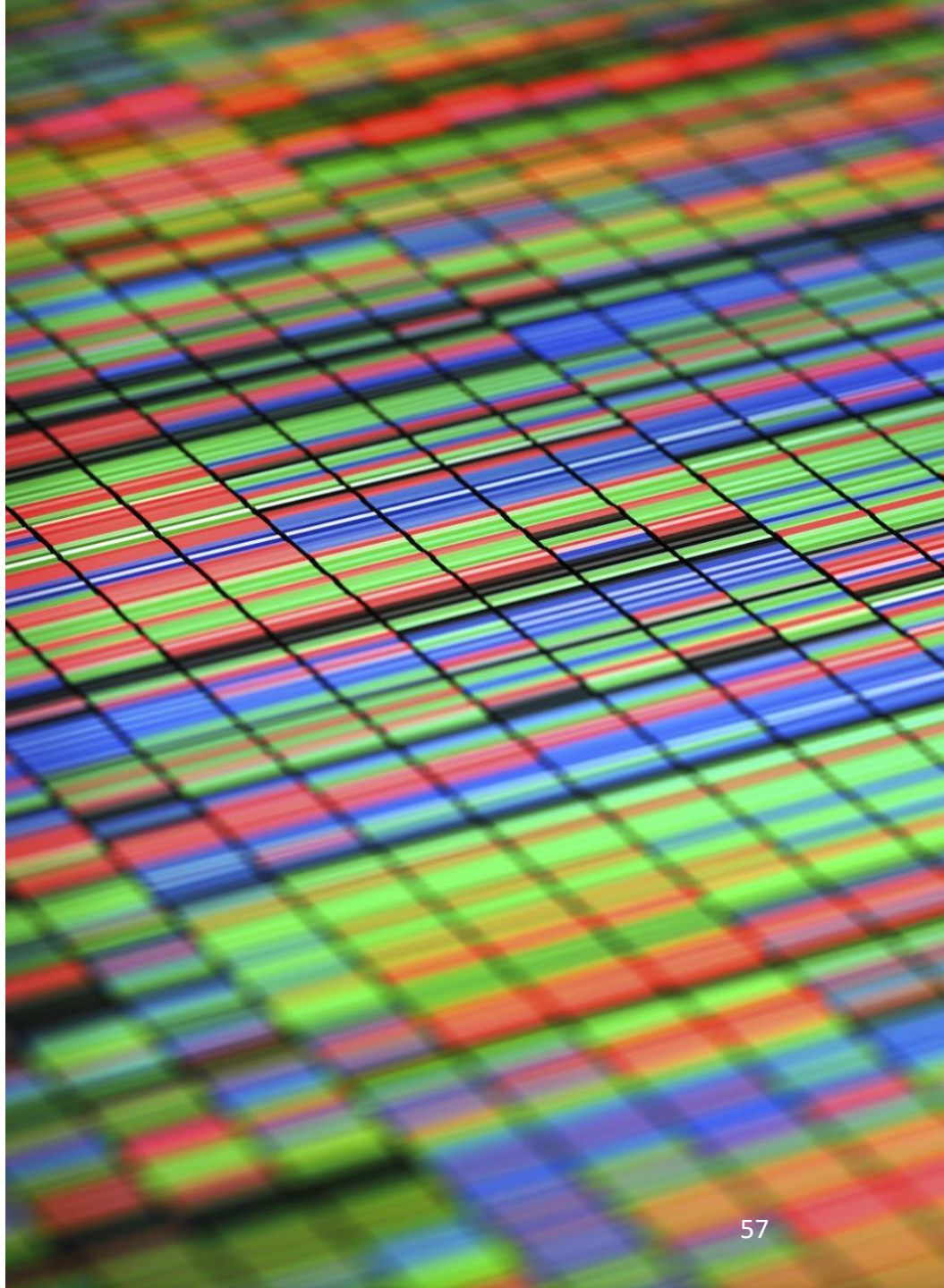
https://github.com/rstudio/master-the-tidyverse/archive/master.zip

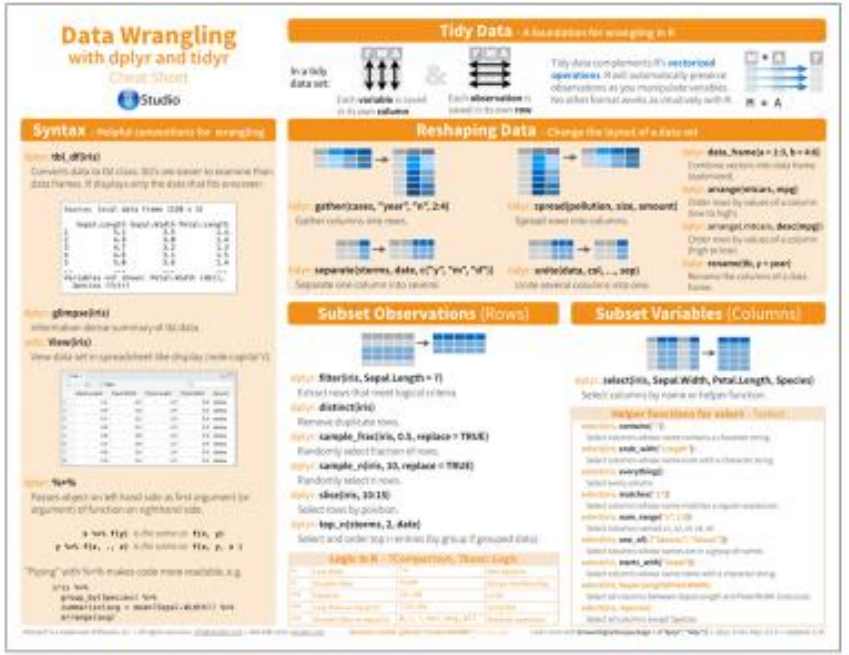
Untidy Data

There are various features of messy data that one can observe in practice.

Here are some of the more commonly observed patterns.

- Column headers are values, not variable names
- Multiple variables are stored in one column
- Variables are stored in both rows and columns
- Multiple types of experimental unit stored in the same table
- One type of experimental unit stored in multiple tables



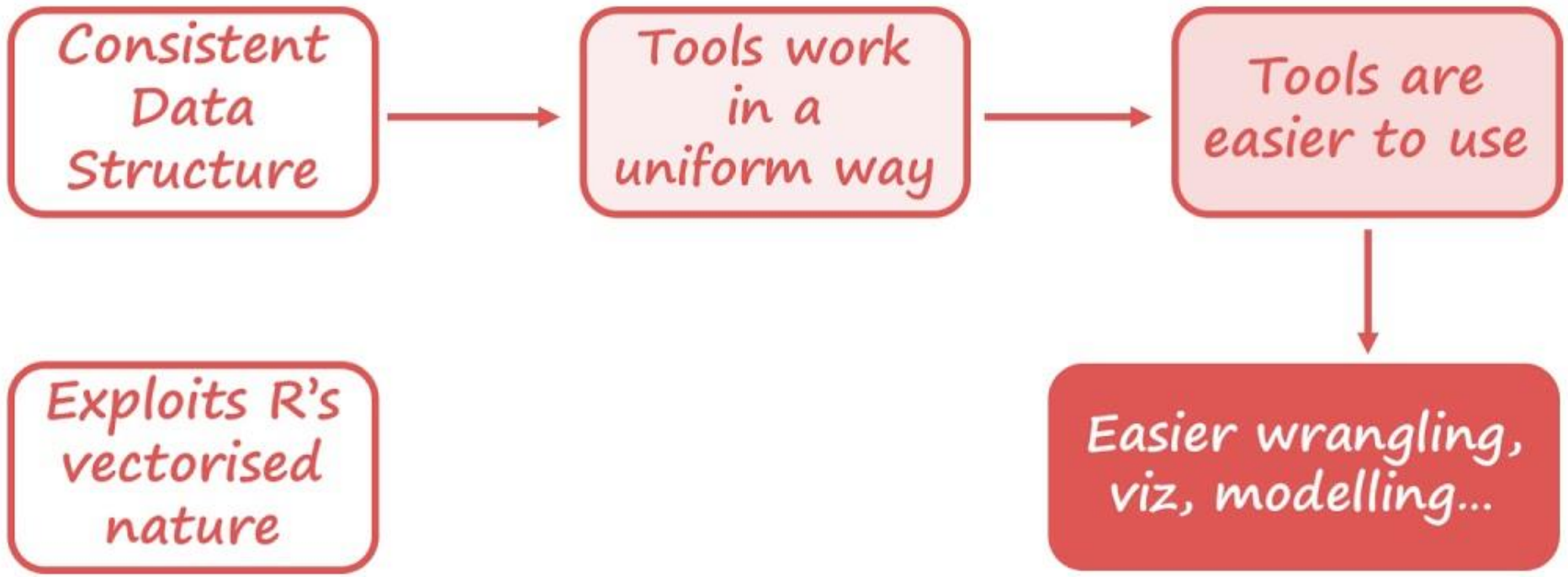


http://www.rstudio.com/resources/cheatsheets/

Tidy Data - A foundation for wrangling in R

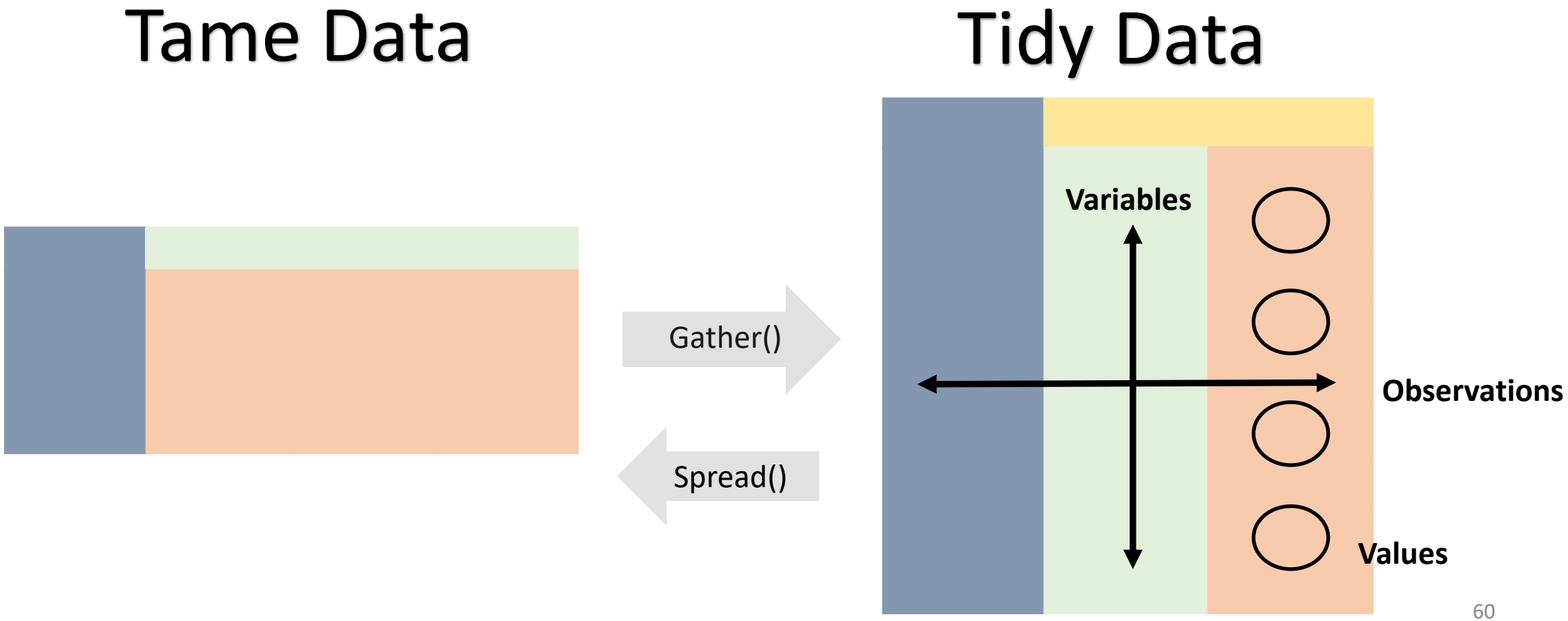
- Syntax - Helpful conventions for wrangling
- Reshaping Data - Change the layout of a data set
- Subset Observations (Rows), Subset Variables (Columns)
- Summarize Data
- Make New Variables
- Combine Data Sets
- Group Data

Why tidy data?



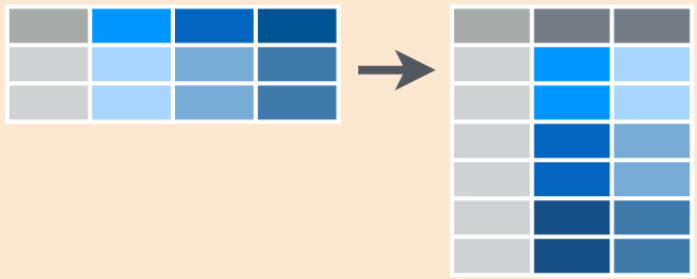
The tidyr package is used to manipulate the structure of your data while preserving all original information, using the following functions:

pivot_longer() data (wide –> long) (formerly gather())
pivot_wider() data (long –> wide) (formerly spread())



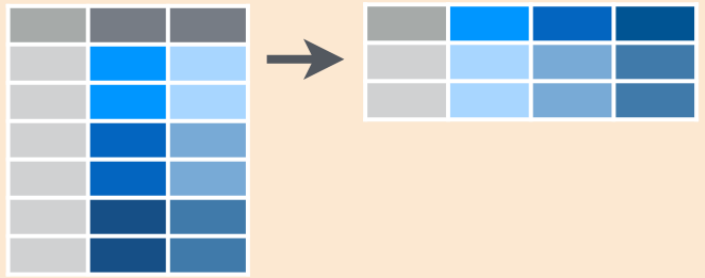
There are four main verbs which are essentially pairs of opposites:

turn columns into rows (gather()),
turn rows into columns (spread()),
turn a character column into multiple columns (separate()),
turn multiple character columns into a single column (unite())



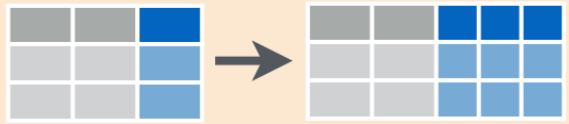
tidyr::**gather(cases, "year", "n", 2:4)**

Gather columns into rows.



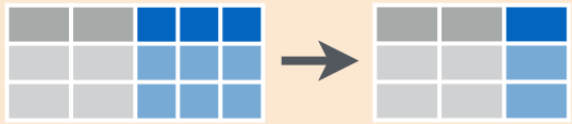
tidyr::**spread(pollution, size, amount)**

Spread rows into columns.



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

Separate one column into several.



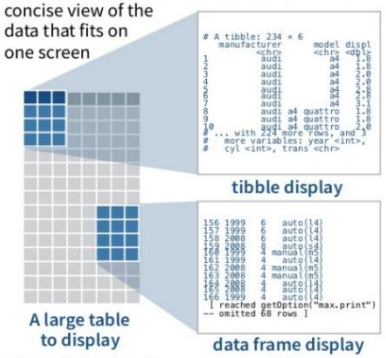
tidyr::**unite(data, col, ..., sep)**

Unite several columns into one.

Tibbles - an enhanced data frame

The **tibble** package provides a new S3 class for storing tabular data, the tibble. Tibbles inherit the data frame class, but improve three behaviors:

- **Subsetting** - [always returns a new tibble, [[and $ always return a vector.
- **No partial matching** - You must use full column names when subsetting
- **Display** - When you print a tibble, R provides a concise view of the data that fits on one screen



- Control the default appearance with options:

  options(tibble.print_max = n, tibble.print_min = m, tibble.width = Inf**)**

- View full data set with **View()** or **glimpse()**
- Revert to data frame with **as.data.frame()**

CONSTRUCT A TIBBLE IN TWO WAYS

tibble(…)
Construct by columns.
tibble(x = 1:3, y = c("a", "b", "c"))

tribble(…)
Construct by rows.

```
tribble( ~x,   ~y,
         1,   "a",
         2,   "b",
         3,   "c")
```

Both make this tibble

```
A tibble: 3 × 2
      x     y
  <int> <chr>
1     1     a
2     2     b
3     3     c
```

as_tibble(x, …) Convert data frame to tibble.

enframe(x, name = "name", value = "value")
Convert named vector to a tibble

is_tibble(x) Test whether x is a tibble.

RStudio

Tidy Data with tidyr

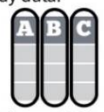
Tidy data is a way to organize tabular data. It provides a consistent data structure across packages.

A table is tidy if:

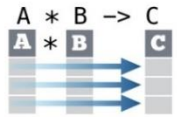
Each **variable** is in its own **column**

Each **observation**, or **case**, is in its own **row**

Tidy data:



Makes variables easy to access as vectors



Preserves cases during vectorized operations

Reshape Data - change the layout of values in a table

Use **gather()** and **spread()** to reorganize the values of a table into a new layout.

gather(data, key, value, …, na.rm = FALSE, convert = FALSE, factor_key = FALSE**)**

gather() moves column names into a **key** column, gathering the column values into a single **value** column.

table4a

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

→

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

key value

gather(table4a, `1999`, `2000`, key = "year", value = "cases")

spread(data, key, value, fill = NA, convert = FALSE, drop = TRUE, sep = NULL**)**

spread() moves the unique values of a **key** column into the column names, spreading the values of a **value** column across the new columns.

table2

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

key value

spread(table2, type, count)

Handle Missing Values

drop_na(data, …**)**
Drop rows containing NA's in … columns.

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
D	3

drop_na(x, x2)

fill(data, …, .direction = c("down", "up"))**
Fill in NA's in … columns with most recent non-NA values.

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
B	1
C	1
D	3
E	3

fill(x, x2)

replace_na(data, replace = list(), …**)**
Replace NA's by column.

x1	x2
A	1
B	NA
C	NA
D	3
E	NA

→

x1	x2
A	1
B	2
C	2
D	3
E	2

replace_na(x, list(x2 = 2))

Expand Tables - quickly create tables with combinations of values

complete(data, …, fill = list()**)**
Adds to the data missing combinations of the values of the variables listed in …
complete(mtcars, cyl, gear, carb)

expand(data, …**)**
Create new tibble with all possible combinations of the values of the variables listed in …
expand(mtcars, cyl, gear, carb)

Split Cells

Use these functions to split or combine cells into individual, isolated values.

separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, convert = FALSE, extra = "warn", fill = "warn", …**)**

Separate each cell in a column to make several columns.

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M
C	1999	212K/1T
C	2000	213K/1T

→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172
B	2000	80K	174
C	1999	212K	1T
C	2000	213K	1T

separate(table3, rate, into = c("cases", "pop"))

separate_rows(data, …, sep = "[^[:alnum:].]+", convert = FALSE**)**

Separate each cell in a column to make several rows. Also **separate_rows_()**.

table3

country	year	rate
A	1999	0.7K/19M
A	2000	2K/20M
B	1999	37K/172M
B	2000	80K/174M
C	1999	212K/1T
C	2000	213K/1T

→

country	year	rate
A	1999	0.7K
A	1999	19M
A	2000	2K
A	2000	20M
B	1999	37K
B	1999	172M
B	2000	80K
B	2000	174M
C	1999	212K
C	1999	1T
C	2000	213K
C	2000	1T

separate_rows(table3, rate)

unite(data, col, …, sep = "_", remove = TRUE**)**

Collapse cells across several columns to make a single column.

table5

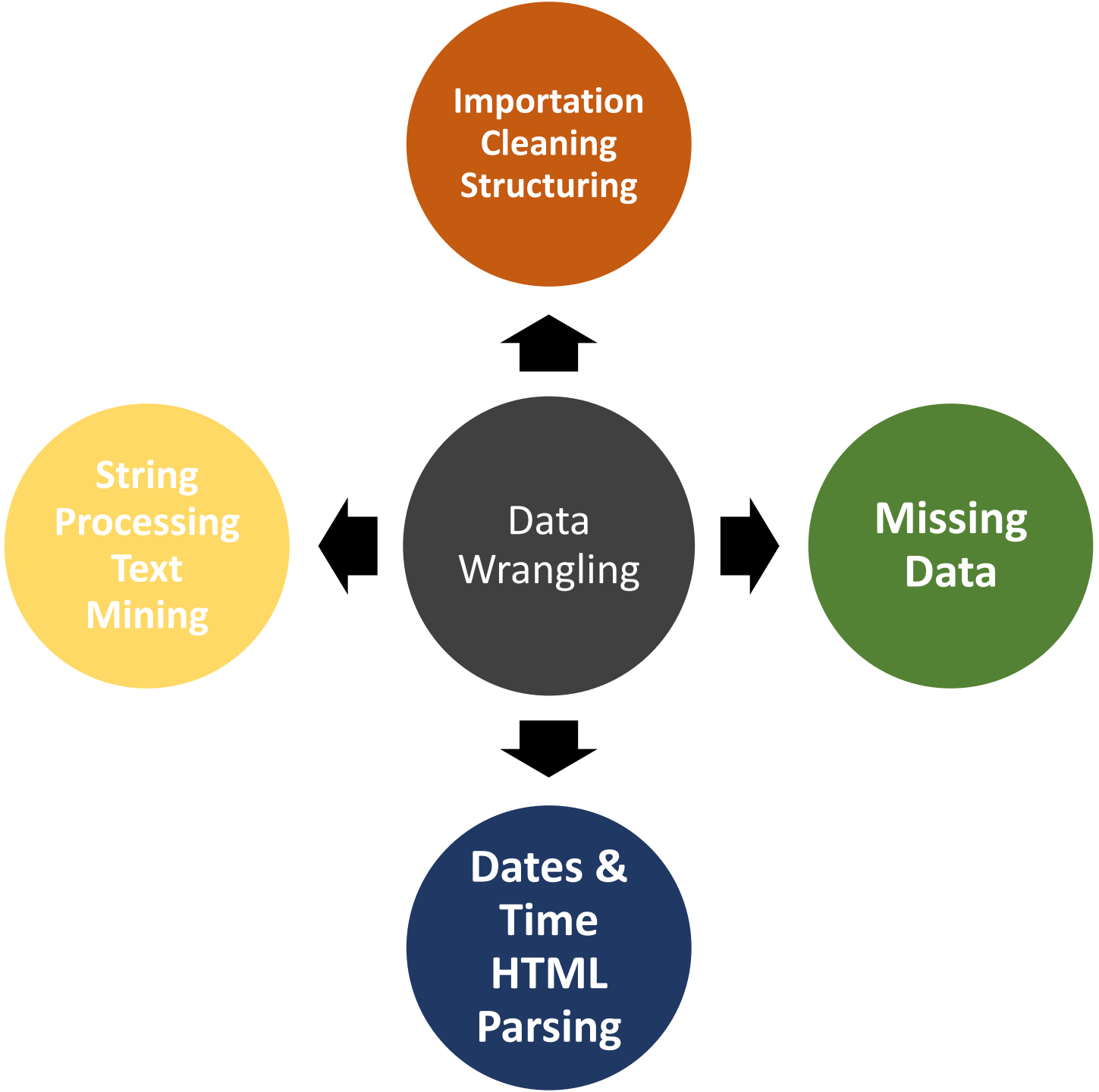
country	century	year
Afghan	19	99
Afghan	20	0
Brazil	19	99
Brazil	20	0
China	19	99
China	20	0

→

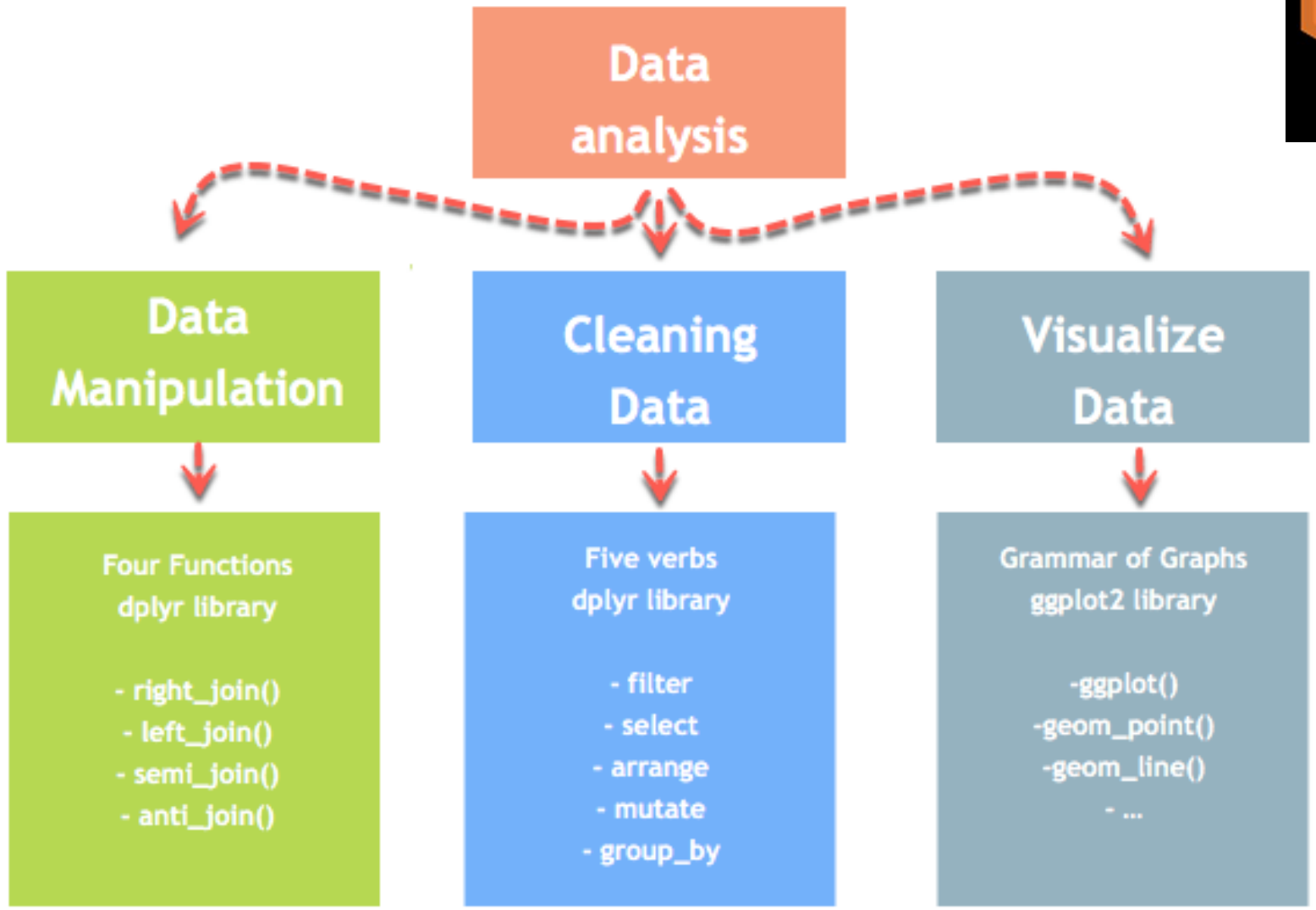
country	year
Afghan	1999
Afghan	2000
Brazil	1999
Brazil	2000
China	1999
China	2000

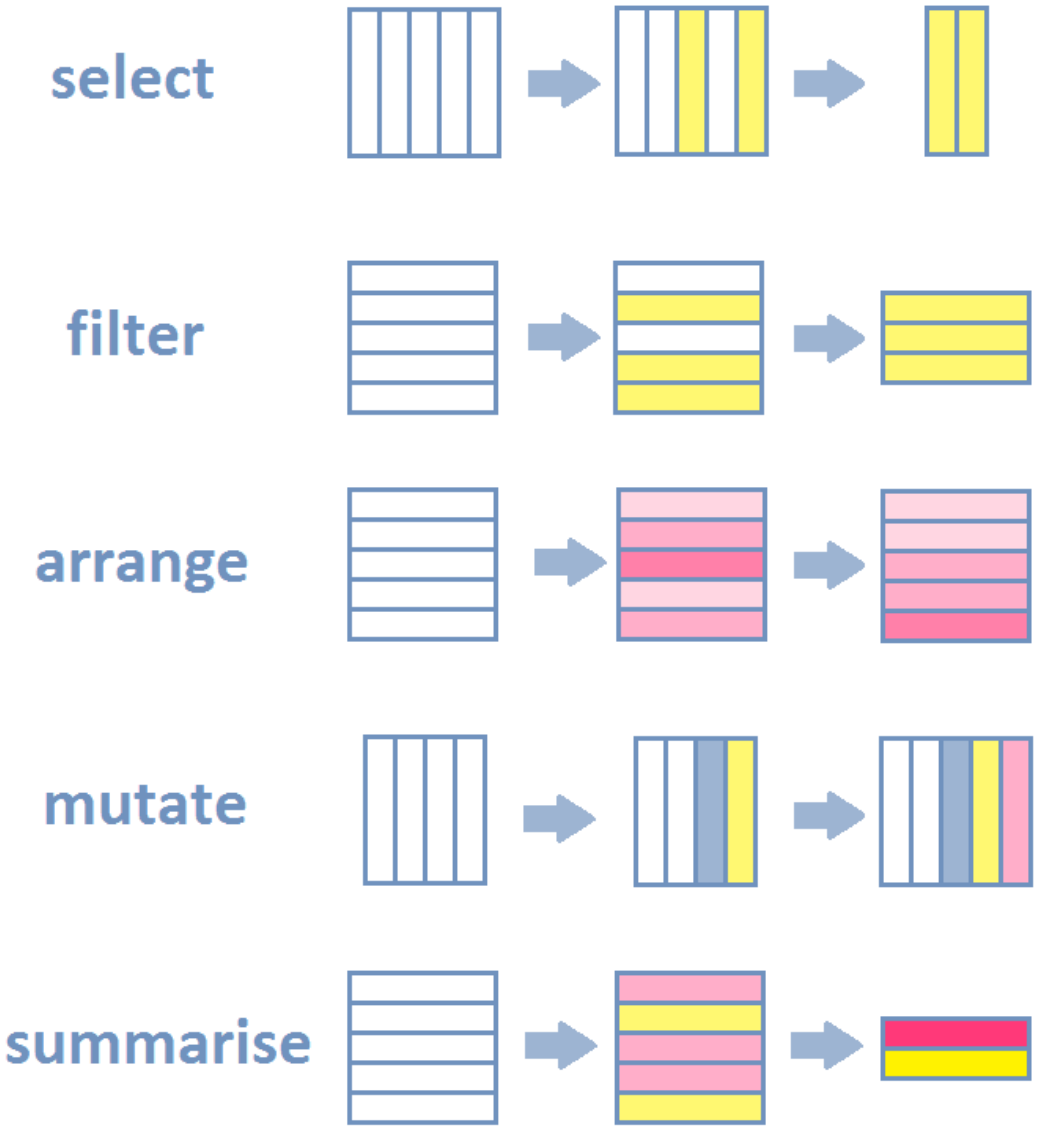
unite(table5, century, year, col = "year", sep = "")

http://tidyr.tidyverse.org/ vignette("tidy-data")


Importation
Cleaning
Structuring
String
Processing
Text
Mining
Data
Wrangling
Missing
Data
Dates &
Time
HTML
Parsing

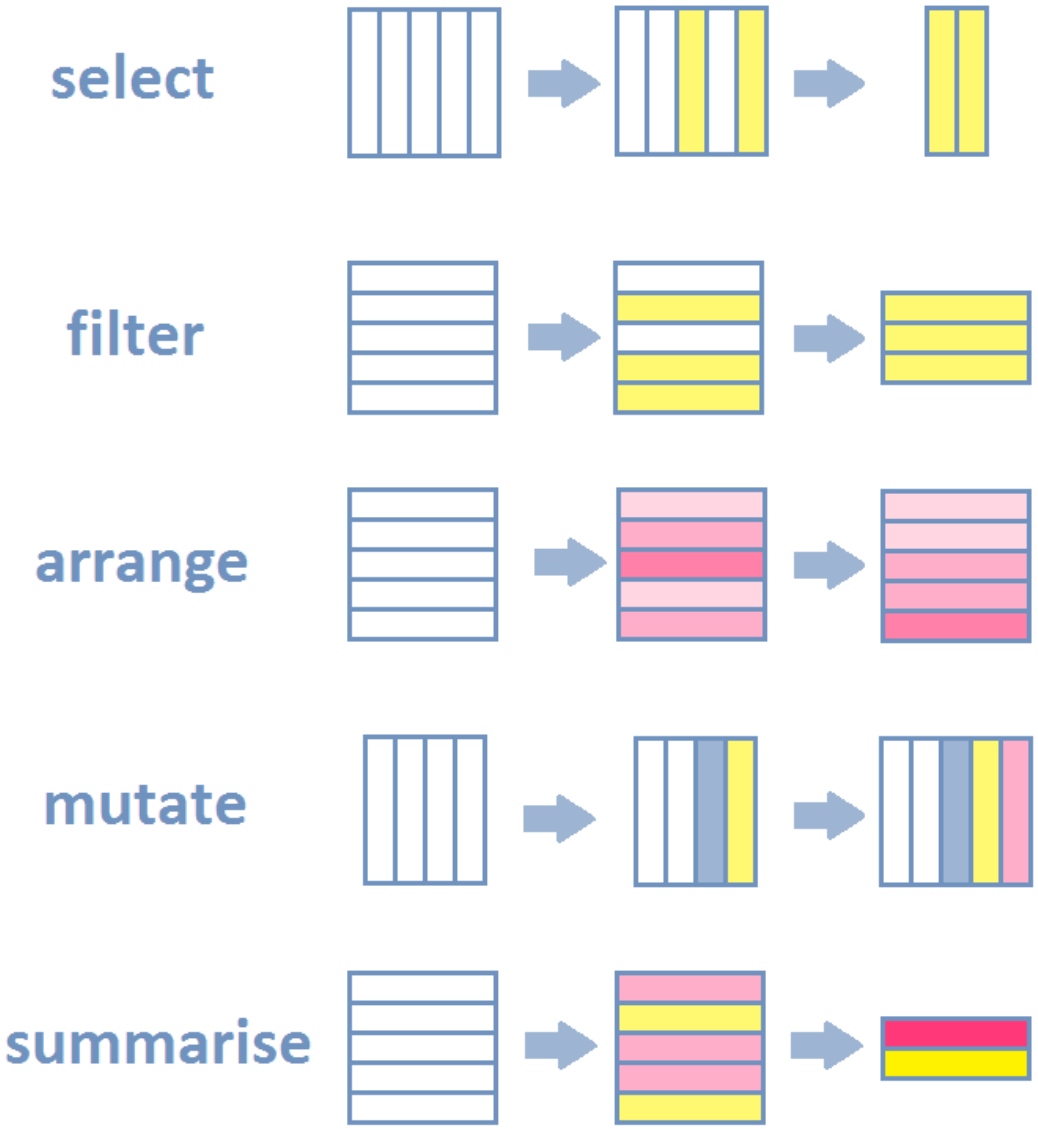
Data Wrangling – Dplyr Package





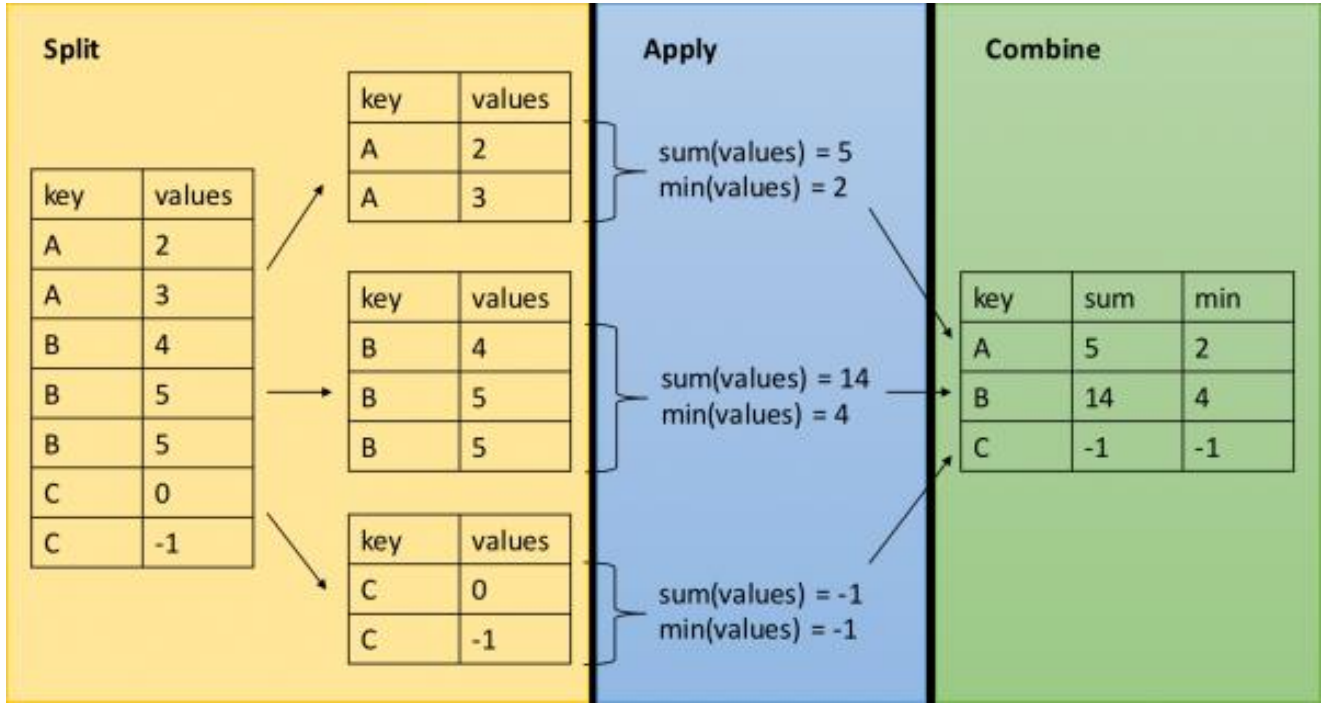
Source: http://perso.ens-lyon.fr/lise.vaudor/dplyr/

- Inspect your tibble (glimpse())
- Select specific columns (select())
- Filter out a subset of rows (filter())
- Reorders rows by one or multiple columns (arrange())
- Change or add columns (mutate())
- Group observations by a grouping variable (group_by())
- Get a summary (in particular per group) (summarise())



Source: http://perso.ens-lyon.fr/lise.vaudor/dplyr/

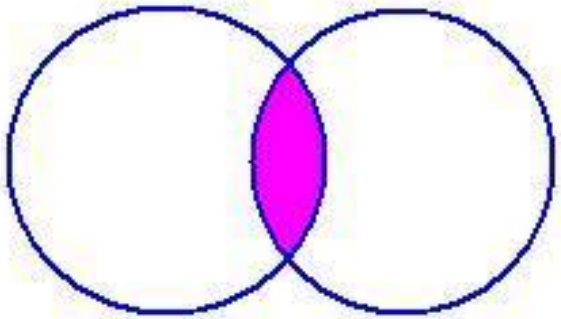
- select(dataframe, column1, column2, …)
- filter(dataframe, logical statement 1, logical statement 2, …)
- arrange(data, variable1, desc(variable2), …)
- mutate(data, newVar1 = expression1, newVar2 = expression2, …)



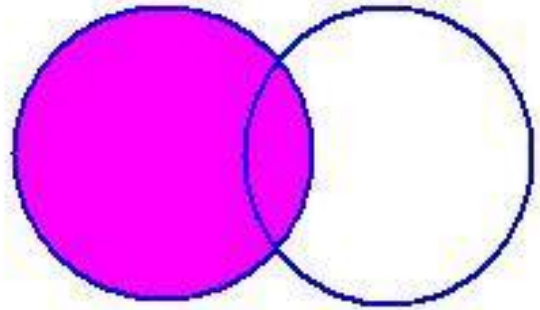
group_by(): group data frame by a factor for downstream commands (usually summarise)

summarise(): summarise values in a data frame or in groups within the data frame with aggregation functions (e.g. min(), max(), mean(), etc…)

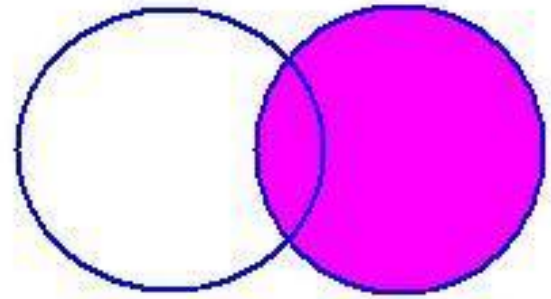
JOINS AND SET OPERATIONS IN RELATIONAL DATABASES



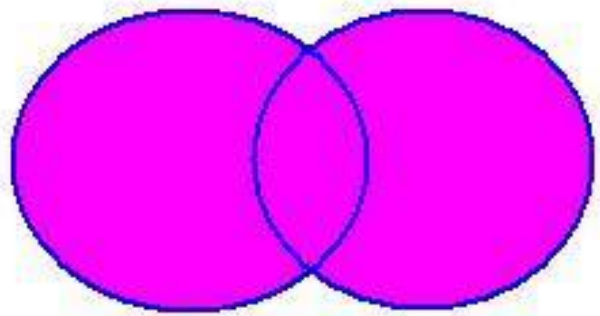
Inner join (result similar to Intersect)



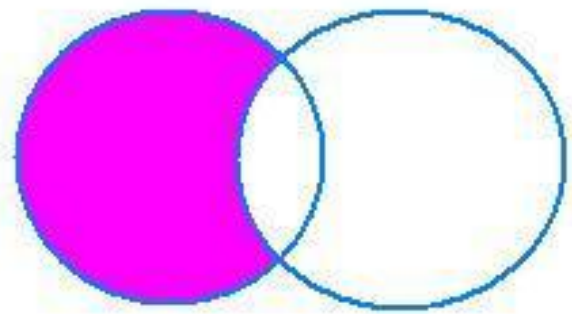
Left outer join



Right outer join

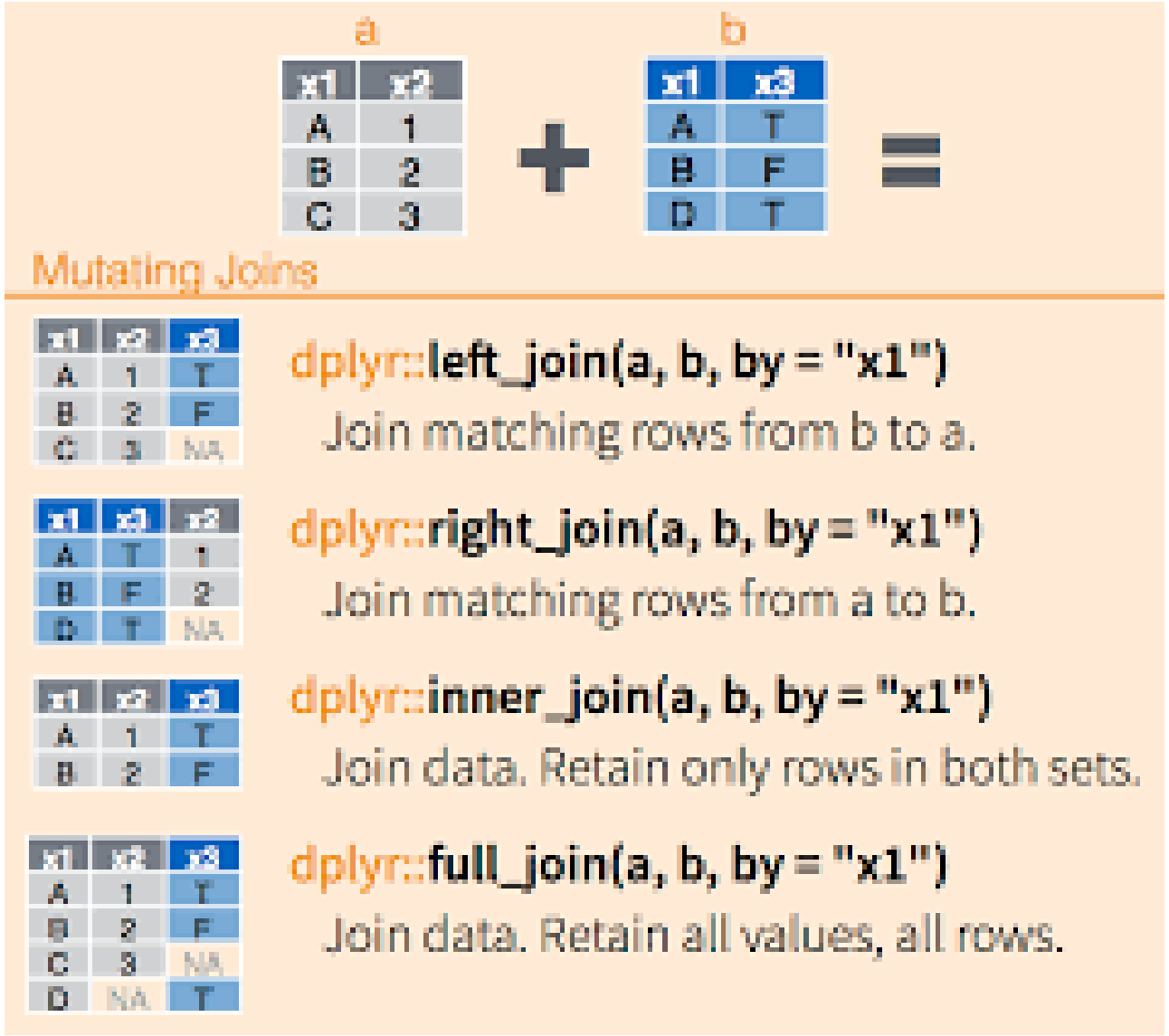


Full outer join



Minus

Functions that allow you to join two data frames together.



Magrittr package: using the pipe operator

- Pipe operators provide ways of linking functions together so that the output of a function flows into the input of the next function in the chain.
- Chaining increases readability significantly when there are many commands. With many packages, we can replace the need to perform nested arguments.
- Specify the dataset first, then “pipe” into the next function in the chain.

```
#1.
dlpyr::select (Tb, child:elderly)

# chaining method
Tb %>% dlpyr::select(child:elderly)
```

```
#2.
x1 <- 1:5; x2 <- 2:6
sqrt(sum((x1-x2)^2))

# chaining method
(x1-x2)^2 %>% sum() %>% sqrt()
```


Any
Questions