**Week 2 – Data Structures**

This week we will focus on the following:

1. Vectors
2. Matrices
3. Lists
4. DataFrames

**Vectors**

Vectors are the simplest data structures in R. They are sequences of elements of the same basic type. These types can be numeric, integer, complex, character, and logical. In R, the more complicated data structures are made with vectors as building-blocks. Vectors contain what are called indices which can be thought of as the position in which a value is located within the vector.

So how do you define a vector in R?   
1) Use c():

EX: number\_vector <- c(1,2,3,5,8,13,21)

2) Using an operator:

EX: number\_vector <- 1:10

3) Taking a slice from a data frame or matrix:

EX1: price\_vector <- boston\_housing\_df[,boston\_housing\_df$price]

EX2: number\_vector <- number\_mat[,1]

More on this later….

**Vector Types & Operations**

Vectors can contain any type of data in R. This means that there are special operations you can perform on the different types. Here are some that will be important for you to learn.

1. Slicing: Slicing is a way for you to subset your vectors to extract elements out of it. There are a few ways to do it in R:
   1. Vectors are ordered by their **indices**. Since R is a 1 based indexed language the first element in a vector will be 1. You can extract specific indices using the bracket notation:

> fib\_vec[1] #will slice the first element

[1] 1

> fib\_vec[1:10] #will slice the first 10 elements

[1] 1 1 2 3 5 8 13 21 34 55

> fib\_vec[c(1,5,10)] #will slice the 1st, 5th, and 10th elements

[1] 1 5 55

1. Sometimes you will want to know the index position a specific number is located at. Or you may want to know where the specific min or max is located. To do this we will use the function **which().**

> which(fib\_vec == 8)

[1] 6

> which.max(fib\_vec)

[1] 10

> which.min(fib\_vec)

[1] 1

1. To remove an element from a vector simply take the inverse of the indices.

> fib\_vec[-8] #remove the 8th element

[1] 1 1 2 3 5 8 13 34 55

> fib\_vec[-1:-10] #remove the first through 10th element

numeric(0)

> fib\_vec[c(-1,-5,-10)] #remove the first, 5th, and 10th element

[1] 1 2 3 8 13 21 34

1. You can also add, subtract, multiply, or divide vectors. Its important to note when you do this each element is added/subtracted/multiplied/divided by the corresponding element in the second vector. If the second vector is of unequal length, the smaller vector will be recycled to compensate for the different length.

> fib\_vec + catalan\_vec#vector addition

[1] 2 3 7 17 47 140 442 1451 4896 16851

> catalan\_vec - fib\_vec #vector subtraction

[1] 0 1 3 11 37 124 416 1409 4828 16741

> catalan\_vec \* fib\_vec#vector multiplication

[1] 1 2 10 42 210 1056 5577 30030 165308 923780

> catalan\_vec/fib\_vec#vector division

[1] 1.000000 2.000000 2.500000 4.666667 8.400000 16.500000 33.000000

[8] 68.095238 143.000000 305.381818

1. Other useful functions with vectors:
   1. any()
   2. all()
   3. sort()
   4. length()
   5. rep()
   6. seq()

> any(catalan\_vec==1) #any of the values equals 1

[1] TRUE

> all(catalan\_vec==1) #all of the values equals 1

[1] FALSE

> sort(catalan\_vec,decreasing=TRUE)#vector sorting

[1] 16796 4862 1430 429 132 42 14 5 2 1

> length(catalan\_vec) #find the length of the vector

[1] 10

> rep(catalan\_vec,2) #repeat the vector 2 times

[1] 1 2 5 14 42 132 429 1430 4862 16796 1 2 5

[14] 14 42 132 429 1430 4862 16796

> seq(from=1,to=10,by=2) #create a vector from 1 to 10 by 2

[1] 1 3 5 7 9

**Matrices**

Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout. They contain elements of the same atomic types. Though we can create a matrix containing only characters or only logical values, they are not of much use. We use matrices containing numeric elements to be used in mathematical calculations.

A lot of times we will be using high level libraries to model our data and will not be doing matrix calculations directly. However, it is still good to understand how to access matrix components and perform calculations against them. Much like vectors matrix elements can be accessed using the bracket notation with the difference being that they are multi-dimensional. The first element within the brackets is the row and the second is the column. So to access elements you can apply the same logic you did for vectors just by slicing either the rows or columns. Full examples can be found in the Ex2\_matrices.R file in /Examples.

**Data Frames**

Data frames are the most widely used data structure in data science. They can contain multiple columns of differing data types and can be thought of as a collection of rows and columns much like what you would find in an Excel workbook. The reason we covered vectors first is because each data frame column is simply a vector. Further, elements of a data frame can be accessed just as they were with matrices. There are rules that data frame objects must follow:

* The column names should be non-empty.
* The row names should be unique.
* The data stored in a data frame can be of numeric, factor or character type.
* Each column should contain same number of data items.

There are several ways in which you can create data frames, either by using the R function data.frame() directly, or by importing data into R from an outside source.

1. CSV files

> sales\_csv <- read.csv("Week\_2/Data/sales.csv"

+ ,stringsAsFactors=FALSE

+ )

1. Tab or any other delimiter

> sales\_tab\_delim <- read.delim("Week\_2/Data/sales.txt"

+ ,stringsAsFactors=FALSE

+ ,sep = "\t"

+ )

1. Excel files

> sales\_excel <- readxl::read\_excel("Week\_2/Data/sales.xlsx"

+ ,sheet = "sales"

+ )

|  |  |  |
| --- | --- | --- |
| **Function** | **What It Does** | **Example** |
| read.table() | Reads any tabular data where the columns are separated (for example by commas or tabs). You can specify the separator (for example, commas or tabs), as well as other arguments to precisely describe your data. | read.table(file=”myfile”, sep=”t”, header=TRUE) |
| read.csv() | A simplified version of read.table() with all the arguments preset to read CSV files, like Microsoft Excel spreadsheets. | read.csv(file=”myfile”) |
| read.csv2() | A version of read.csv() configured for data with a comma as the decimal point and a semicolon as the field separator. | read.csv2(file=”myfile”, header=TRUE) |
| read.delim() | Useful for reading delimited files, with tabs as the default separator. | read.delim(file=”myfile”, header=TRUE) |
| scan() | Allows you finer control over the read process when your data isn’t tabular. | scan(“myfile”, skip = 1, nmax=100) |
| readLines() | Reads text from a text file one line at a time. | readLines(“myfile”) |
| read.fwf | Read a file with dates in fixed-width format. In other words, each column in the data has a fixed number of characters. | read.fwf(“myfile”, widths=c(1,2,3) |

**Data Frame Sub-setting**

R Data frames are arranged in rows and columns. Data frames contain named columns and can contain unique row names.

Using bracket notation in R we can refer to the rows and columns in R as follows:

Data[row, column]

Typically, when you want to ‘filter’ a column for a specific value(s) you will want to subset the rows of the data. An example of this is shown below:

mean(sales[sales$Customer.Name=='Darrin Van Huff','Profit'])

[1] 5.71275

Take a second to digest what’s going on here. We are calculating the average profit for Darrin Van Huff’s sales. To do this we subset the rows to match his name and subset the Columns to only include the profit column. This will give us a vector of profits so its easy to just wrap this expression with a mean() to return the average profit per transaction. Notice how the `sales` object name followed by the Column name is specified in the rows portion of the sub-set but the column name is wrapped in quotes in the column portion of the sub-set. This is because you need to reference a column to subset the rows against. If the rows had names you could simply specify the row name corresponding to our customer.

**Data Frame Data Types**

Data frames typically contain varying data types. Its important to understand what each is and how they will be used. By now you should be familiar with integers and numeric vectors so we will skip those data types and focus on dates, characters, factors, and NA values (not an actual data type).

**Characters** (strings)

A character in R is any combination of numbers, letters, or other characters (text) wrapped in quotes. Characters typically hold descriptions, keys, raw dates, names, or any other descriptive data. In R we will be using the stringr package to work with and manipulate characters. You will need to familiarize yourself with regular expressions (regex) in order to become proficient with characters in R. The good news is that almost every language uses the same syntax for regex. A cheat sheet can be found in the cheat sheet directory of week 2.

> # Splitting strings to create two new columns

> ## String split fixed will split the product id column into three columns in a matrix by the '-'

> temp\_char <- stringr::str\_split\_fixed(string=sales\_csv$Product.ID,pattern='-',n=3)

> # To recreate our Product ID we just paste the individual vectors together with the '-' seperator

> sales\_csv$Product <- paste(temp\_char[,1], temp\_char[,2],sep='-')

> # Our product number is now just the third column of the matrix

> sales\_csv$Product.Number <- temp\_char[,3]

>

> cat(

+ sales\_csv$Product.ID[1]

+ ,sales\_csv$Product[1]

+ ,sales\_csv$Product.Number[1]

+ ,sep="\n\r"

+ )

FUR-BO-10001798

FUR-BO

10001798

**Factors**

Factors are used to represent categorical data. Factors can be ordered or unordered and are an important class for statistical analysis and for plotting.

Factors are stored as integers, and have labels associated with these unique integers. While factors look (and often behave) like character vectors, they are integers under the hood, and you need to be careful when treating them like strings.

Once created, factors can only contain a pre-defined set value, known as levels. By default, R always sorts levels in alphabetical order.

Factors are created using the factor () function by taking a vector as input.

Sometimes, the order of the factors does not matter, other times you might want to specify the order because it is meaningful (e.g., “low”, “medium”, “high”) or it is required by a particular type of analysis. Additionally, specifying the order of the levels allows us to compare levels.

Examples of basic usage of factors are available in the Data Frame example file and a cheat sheet is available for you in the cheat sheet directory.

**Dates**

When you import data into R dates are usually represented as character strings. Dates must be formatted using as.Date() for R to register the data type correctly. You can apply this function to an entire vector (column) in a data frame. There are other libraries and functions for converting R to date time, but for now we will stick with just dates. In order to correctly convert the character object to date you must first understand how the character object is formatted. Below is a table of conversions for formatting a character to date along with an example. There is a cheat sheet in the cheat sheets directory along with examples in the Data Frame file.

> # Convert the Order.Date string to Date format

>

> # Get the structure of the date column

> str(sales\_csv$Order.Date)

chr [1:9994] "11/8/2016" "11/8/2016" "6/12/2016" "10/11/2015" "10/11/2015" ...

>

> # Check to see if Order.Date is a date

> inherits(sales\_csv$Order.Date, c("Date"))

[1] FALSE

>

> # Using the table in our notes, convert the character to a date obeject.

> sales\_csv$Order.Date <- as.Date(sales\_csv$Order.Date,format='%m/%d/%Y')

>

> # Check to see if our conversion worked

> inherits(sales\_csv$Order.Date, c("Date"))

[1] TRUE

> str(sales\_csv$Order.Date)

Date[1:9994], format: "2016-11-08" "2016-11-08" "2016-06-12" "2015-10-11" "2015-10-11" ...

|  |  |  |
| --- | --- | --- |
| **Conversion specification** | **Description** | **Example** |
| %a | Abbreviated weekday | Sun, Mon |
| %A | Full weekday | Sunday, Monday |
| %b or %h | Abbreviated month | Sep, Nov |
| %B | Full month | September, November |
| %d | Day of the month | 10, 20 |
| 31-Jan |
| %j | Day of the year | 250, 310 |
| 001-366 |
| %m | Month | 09, 11 |
| 12-Jan |
| %U | Week | 35, 45 |
| Jan-53 |
| with Sunday as the first day of the week |
| %w | Weekday | 0, 4 |
| 0-6 |
| Sunday is 0 |
| %W | Week | 21, 27 |
| 00-53 |
| with Monday as the first day of the week |
| %x | Date, locale-specific |  |
| %y | Year without century | 84, 05 |
| 00-99 |
| %Y | Year with century | 1984, 2011 |
| on input: |
| 00 to 68 prefixed by 20 |
| 69 to 99 prefixed by 19 |
| %C | Century | 19, 20 |
| %D | Date formatted %m/%d/%y | 09/10/93, 11/20/93 |
| %u | Weekday | 7, 4 |
| 7-Jan |
| Monday is 1 |

**Lists**

Lists are the R objects which contain elements of different types like − numbers, strings, vectors and another list inside it. Lists are generally used to store a collection of different objects into a single object. Many of the modeling functions you will encounter in your future courses will output lists for the summaries of the models.

Sub-setting lists is a little different that other objects. For lists you will need to use double brackets as shown below and in the example.

> # Create a list containing strings, numbers, vectors and a logical value.

> list\_data <- list("Male", "Female", c(35,42,5), FALSE, 220.5)

>

> # Get the first list element

> list\_data[[1]]

[1] "Male"

> # Get the third list element (a vector!)

> list\_data[[3]]

[1] 35 42 5

> # Get the last element of the vector

> list\_data[[3]][3]

[1] 5

> # We can also name the elements of the list

> names(list\_data) <- c("Male", "Female", "number\_vector","logical", "numeric")

>

> print(list\_data)

$Male

[1] "Male"

$Female

[1] "Female"

$number vector

[1] 35 42 5

$logical

[1] FALSE

$numeric

[1] 220.5