UTSA®

ALVAREZ
College of Business
The University of Texas at San Antonio

# Introduction to Programming in R

## Module 2:

## Data Types

# Learning Objectives

- Creating vectors: numeric, character, and factor
- Filtering, editing, replicating, and creating sequences
- Creating arrays, matrices, data frames, and lists
- Initializing and coercing objects
- Dealing with missing values

# Vectors

- In R, a vector is a one-dimensional object, comprised of elements: e.g. {1, 2, 3, 4, 5}
  - A matrix is a two-dimensional object, and an array is an n-dimensional object.
- The function *c()* (short for combine) creates a vector.
- Create a vector v1 comprised of the numbers 1, 2, …, 10:

*v1 <- c(1,2,3,4,5,6,7,8,9,10)*

- To see the result, execute *print(v1)* on the next line. Alternative just *v1* (without print)*.*
- To see the type of vector v1 is, use *class(v1)* = *"numeric"*
- Or, the operator ‘:’ creates a sequence of consecutive numbers with a unit increment:

v2 = c(1:10)

- See the type of object v2: *class(v2)* = *"integer"*
- Size of vector, use *length().* In R, *dim()* won’t work on a vector.

# Vectors (cont.)

*Can be comprised of:*

- Number (numeric data)

  E.g., *c(1:10)*
- Logical

  *TRUE / FALSE* or *T / F*
- Character

  E.g., R has a predefined character vectors of the English alphabet: *letters* (lowercase) and *LETTERS* (uppercase
- Factor

  E.g., *factor(c("Freshman", "Senior", "Junior", "Sophomore", "Junior", "Senior"))*.

**Logical Vectors**

- Create a logical vector: *c(FALSE, TRUE, TRUE, FALSE, TRUE)* or *c(F, T, T, F, T)*.
- Most simple form of vector, basically binary (1, 0).
  - Contains the least amount of information (yes/no).

# Character Vector

- Multiple *characters* (i.e., letters) form *strings*.
- To see a character vector of the English alphabet:

  *print(letters)* (lowercase) or *print(LETTERS)* (uppercase)
- To double check the type of object letters is:

  *class(letters)* or *class(LETTERS)*
- Create a character vector *c("Hello", "World!")*.
- Suppose you want to combine multiple strings: “Hello” & “World!”

  *paste("Hello", "World!")* ☾ *"Hello World!”*
- By default, *paste()* enters a space separator*.*
  - What if you don’t want that? For e.g., if you had “extra” & “terrestrial”.

    *paste0("extra", "terrestrial")* or *paste("extra", "terrestrial", sep = "")*
  - *sep* = “" tells R not to put a space. sep can be anything, e.g., *sep* = "-".

# Factor Vector

- Categorical data, contains more information than logical, but only a finite (limited) number of categories possible.
- Categories are referred to as *levels*.
- When a numeric vector is converted to a factor, the unique values serve as levels.

- Create a factor vector of 10 students with alternative genders of students, starting with female.

```
f1 <- factor(c("male", "female", "male", "female", "male", "female",     "male",   "female", "male", "female", "male", "female"))
```

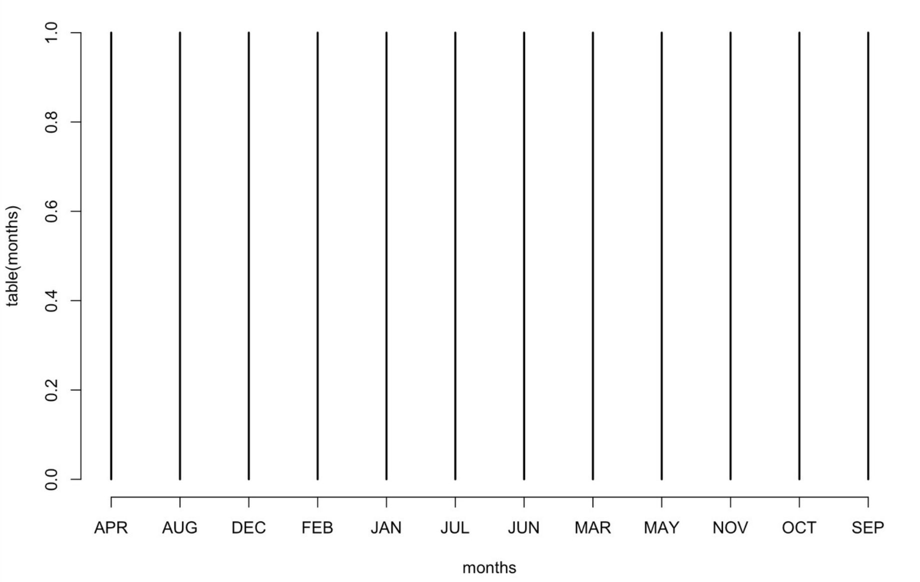
- R will automatically show you the levels when it prints the output, however, to see it without printing the entire vector: *levels(f1)*

# Ordered Factor Vector

- Create a vector of months of the year:

  *months = factor(c("JAN","FEB","MAR","APR","MAY","JUN", "JUL","AUG","SEP","OCT","NOV","DEC"))*

- To see a frequency distribution of this: *plot(table(months))*

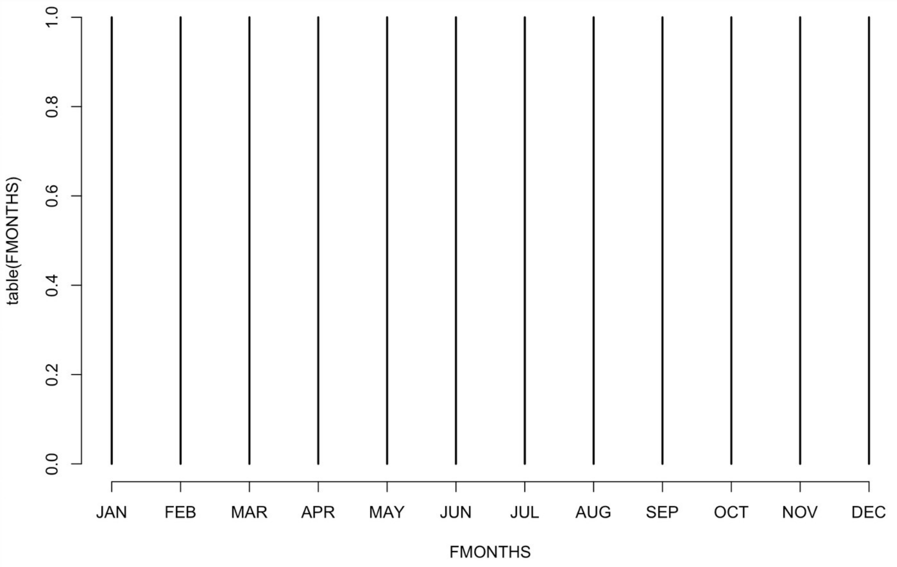


- *How can I fix this ordering? It's alphabetical.*

# Ordered Vector

- Have months as numbers? Yes, but cumbersome!
- We can create *ordered factors*: use argument *ordered = TRUE*.

*factor(months, levels = c("JAN","FEB","MAR","APR","MAY","JUN", "JUL","AUG","SEP","OCT","NOV","DEC"), ordered = TRUE)*

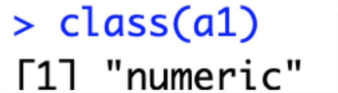


- You can change the order using levels(). For e.g.

*factor(months, levels = c("DEC","FEB",”OCT","APR","MAY","JUN", ”AUG",”JUL","SEP",”MAR","NOV",”JAN"), ordered = TRUE)*

# Vector Manipulation

- Function *c()* can be used to combine elements to existing vector.
- Create vector a <- c(1,2,3,4) and check its class.

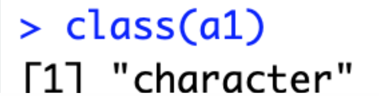


```
> class(a1)
[1] "numeric"
```

- Then, concatenate letters a, b, c, and d to it.

*a1 <- c(1,2,3,4)* then *a1 <- c(a1,"a","b","c","d")*

- Check the class of the updated *a1*. What happened?

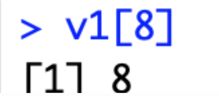


```
> class(a1)
[1] "character"
```

- Remember vectors can be only of a single data type.
  - Be careful! The numeric data became all characters.
  - Characters contain much lesser information than numeric.
  - Can’t compute sum, mean, etc. with character vectors.

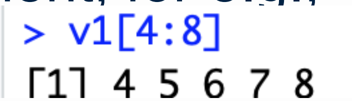
# Subset of Vector by Index

- To obtain a specific element of a vector, use square brackets [].
- Suppose v1 <- c(1:10), and you wanted the element at 8th position.



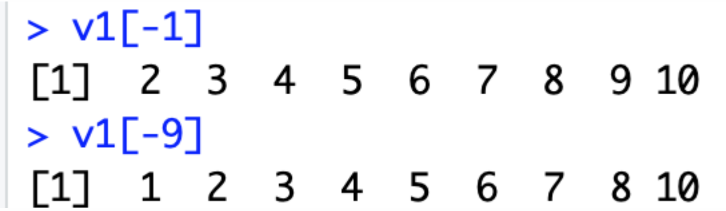
```
> v1[8]
[1] 8
```

- You can also subset a range of element, for e.g., 4th to 8th elements.



```
> v1[4:8]
[1] 4 5 6 7 8
```

- You can also subset excluding elements, using negative sign.
- Removing 1st or 9th elements:

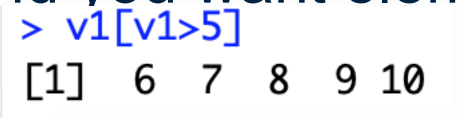


```
> v1[-1]
[1]  2  3  4  5  6  7  8  9 10
> v1[-9]
[1]  1  2  3  4  5  6  7  8 10
```

- Works similarly for character/logical/factor vectors.

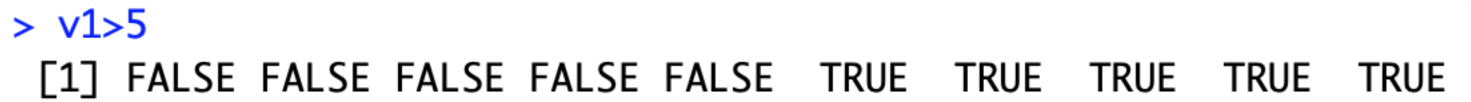
# Subset of Vector by Criteria

- Suppose v1 <- c(1:10) and you want elements larger than 5.



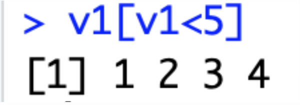
```
> v1[v1>5]
[1]  6  7  8  9 10
```

  - Here, first a vector of logicals is computed using *v1 > 5*.

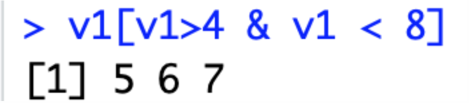


```
> v1>5
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

  - This then serves as indices to get the actual values.
- If you want elements smaller than 5:

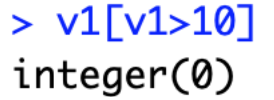


```
> v1[v1<5]
[1] 1 2 3 4
```



```
> v1[v1>4 & v1 < 8]
[1] 5 6 7
```

- If you want elements between 4 and 8.
  - Here we use the & operator which makes sure both criteria holds true.
- If a criteria is false, then 0 is returned:

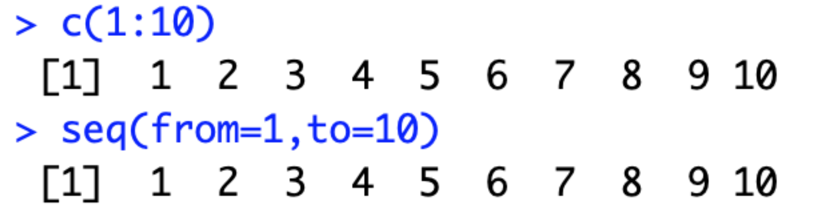


```
> v1[v1>10]
integer(0)
```

# Sequences

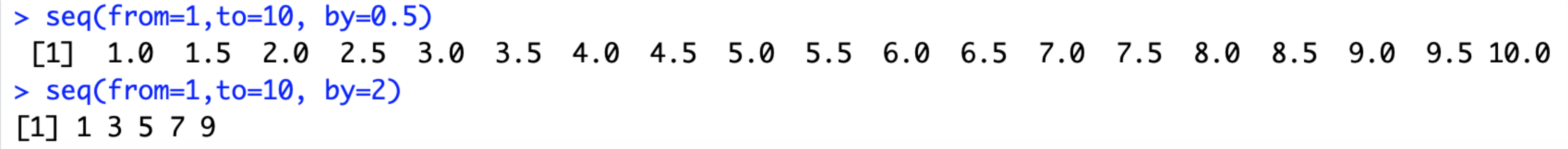
- Instead of using a colon, you can use seq(), which is more generic.

  Try *c(1:10)* and *seq(from=1, to=1*



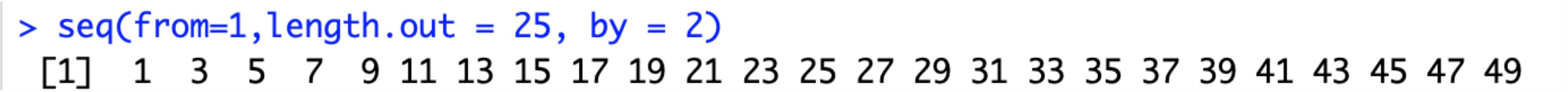
```
> c(1:10)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(from=1,to=10)
 [1]  1  2  3  4  5  6  7  8  9 10
```

- While colon used a unit increment, seq() allows you to define the increment. What if we want to increment by 0.5 or 2.



```
> seq(from=1,to=10, by=0.5)
 [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0
> seq(from=1,to=10, by=2)
[1] 1 3 5 7 9
```

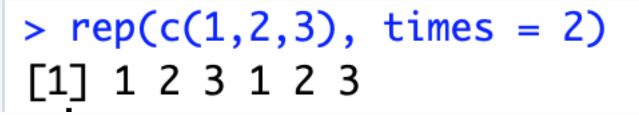
- What if we didn't know “to”, but knew the length? Use *length.out*



```
> seq(from=1,length.out = 25, by = 2)
 [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
```

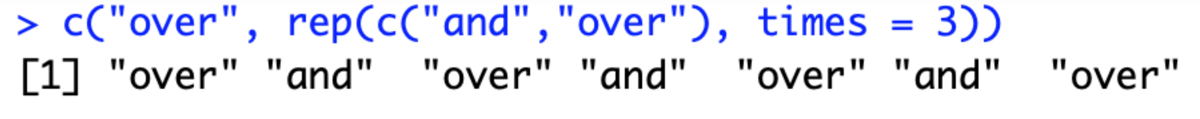
# Replicates

- If you want replicate elements? For e.g., a vector {1,2,3,1,2,3}.
  - Use *rep()* function:



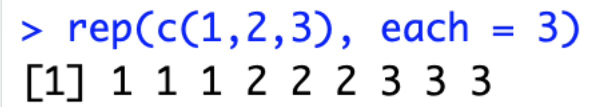
```
> rep(c(1,2,3), times = 2)
[1] 1 2 3 1 2 3
```

- What about {"over", "and", "over”, "and" "over", "and", "over"}?



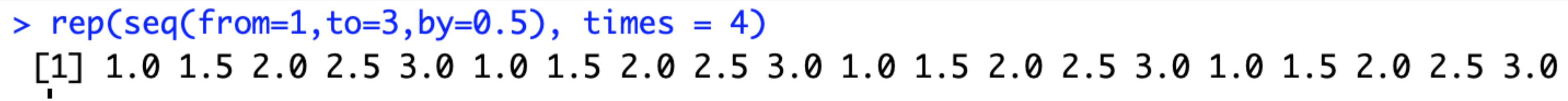
```
> c("over", rep(c("and","over"), times = 3))
[1] "over" "and"  "over" "and"  "over" "and"  "over"
```

- What if you want to collate elements, i.e. {1,1,1,2,2,2,3,3,3}?
  - Use *each()* function*:*



```
> rep(c(1,2,3), each = 3)
[1] 1 1 1 2 2 2 3 3 3
```

- What about something a little more complicated?
  - Create {1,1.5,2,2.5,3,1,1.5,2,2.5,3,1,1.5,2,2.5,3, 1,1.5,2,2.5,3}



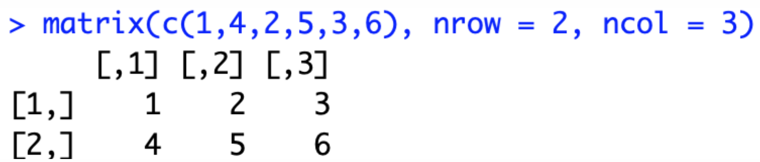
```
> rep(seq(from=1,to=3,by=0.5), times = 4)
 [1] 1.0 1.5 2.0 2.5 3.0 1.0 1.5 2.0 2.5 3.0 1.0 1.5 2.0 2.5 3.0 1.0 1.5 2.0 2.5 3.0
```

# Matrices

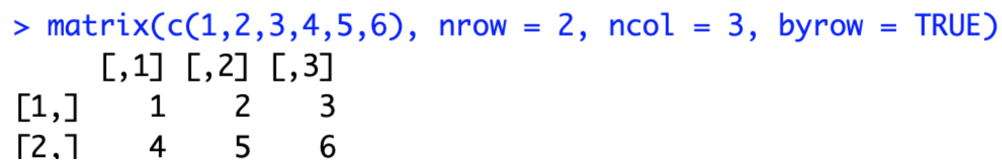
- A matrix is a two-dimensional object, comprised of elements:

  e.g.  or

- A matrix can be decomposed into multiple row or column vectors.
- By default, a matrix is constructed by filling up column-wise.
  - To overwrite this behavior: *byrow = TRUE*
- You must define the number of rows and columns, *nrow* and *ncol*.

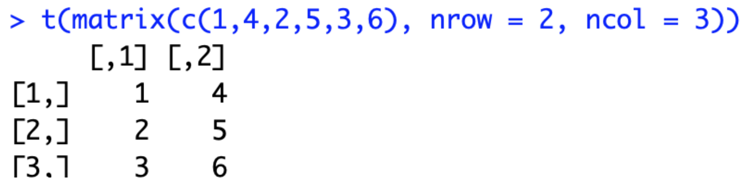


```
> matrix(c(1,4,2,5,3,6), nrow = 2, ncol = 3)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```



```
> matrix(c(1,2,3,4,5,6), nrow = 2, ncol = 3, byrow = TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

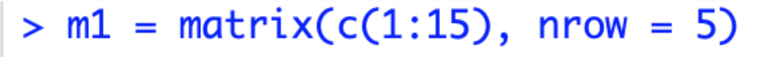
- To transpose the matrix, i.e. : use *t()*.



```
> t(matrix(c(1,4,2,5,3,6), nrow = 2, ncol = 3))
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

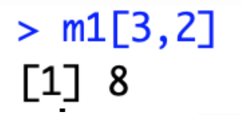
# Subset by Matrix by Index

- Let’s create a matrix:

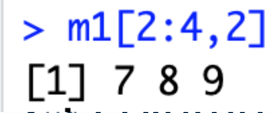


```
> m1 = matrix(c(1:15), nrow = 5)
```

- To extract values, use [], with row first, then column.
- Extract the number 8 from the matrix:
- Extract the numbers 7, 8, and 9:

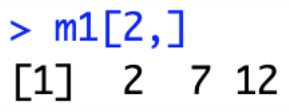


```
> m1[3,2]
[1] 8
```

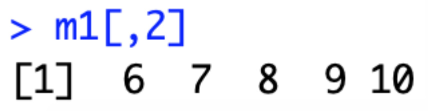


```
> m1[2:4,2]
[1] 7 8 9
```

- Extract the entire 2nd row or the 2 column.
- For matrices *dim()* gives size and *length(* er of e

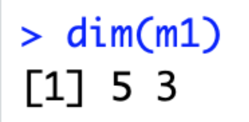


```
> m1[2,]
[1]  2  7 12
```

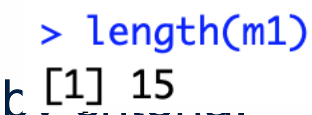


```
> m1[,2]
[1]  6  7  8  9 10
```

- Like vectors, subsets can tracted b



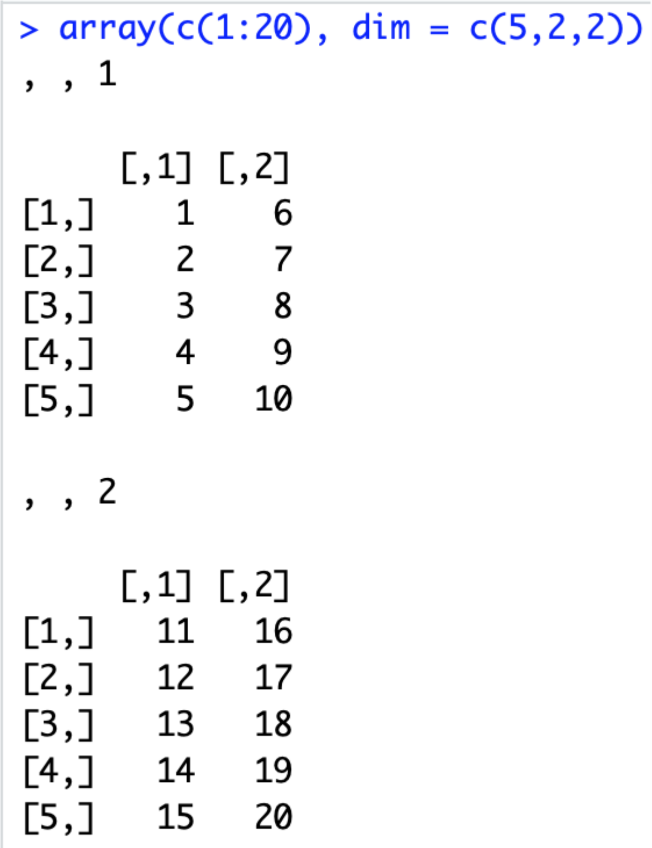
```
> dim(m1)
[1] 5 3
```



```
> length(m1)
[1] 15
```

# Arrays

- A three-dimensional object, comprised of elements.
- Think of arrays as books, where each page has a matrix.
- Useful for applications in medical imaging (CT scans – slices) or storing large related datasets.
- The *array()* function needs the data and the dimensions, *dim(#rows, #cols, #pages)*.
- For e.g., to create 2 matrices layered onto two pages, with numbers 1 to 20, you can use:
- Like vectors and matrices, subsets can be extracted by index or criteria.



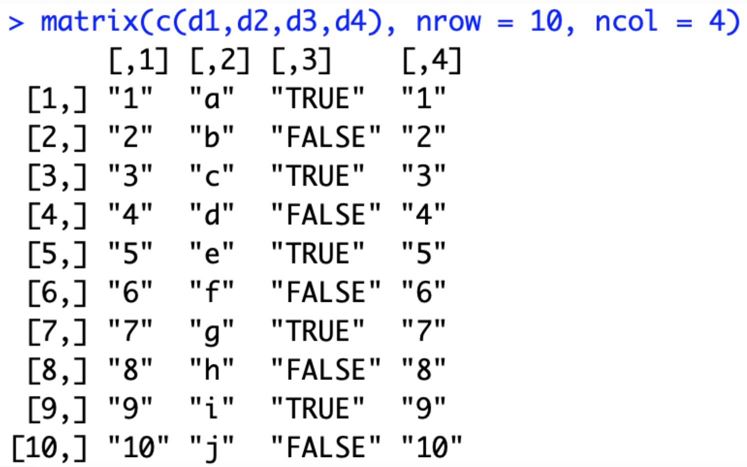
```
> array(c(1:20), dim = c(5,2,2))
, , 1

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

, , 2

     [,1] [,2]
[1,]   11   16
[2,]   12   17
[3,]   13   18
[4,]   14   19
[5,]   15   20
```

# Data Frames

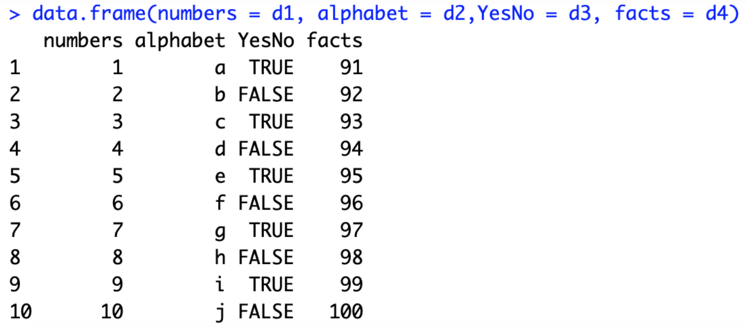
- Similar structure of matrix, but same length variables can be of different data types, i.e., character, numeric, logical, factors, etc.
  - Think of a real-estate dataset: sq. ft., price, age, fire-place, pool…
  - Till now, either separate (hard to manipulate across all variables, for e.g. sort) or combined (matrix - but must be of same data type). Data frame can overcome this!

Num Char Logi Factor

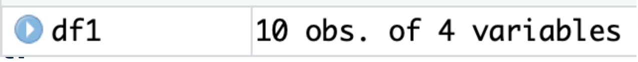
- E.g. make   as separate vectors.
- Try as *matrix(c(d1,d2,d3,d4, nrow = 10, ncol = 4))*
- Try as *data.frame(name1 = d1,…,name4 = d4)*
- Data frame structure: columns - variables, rows - obs.
- If you assign to df1, then you can see in environment.



```
> matrix(c(d1,d2,d3,d4), nrow = 10, ncol = 4)
      [,1] [,2] [,3]    [,4]
 [1,] "1"  "a"  "TRUE"  "1"
 [2,] "2"  "b"  "FALSE" "2"
 [3,] "3"  "c"  "TRUE"  "3"
 [4,] "4"  "d"  "FALSE" "4"
 [5,] "5"  "e"  "TRUE"  "5"
 [6,] "6"  "f"  "FALSE" "6"
 [7,] "7"  "g"  "TRUE"  "7"
 [8,] "8"  "h"  "FALSE" "8"
 [9,] "9"  "i"  "TRUE"  "9"
[10,] "10" "j"  "FALSE" "10"
```



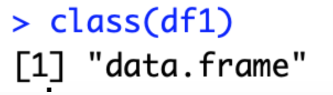
```
> data.frame(numbers = d1, alphabet = d2,YesNo = d3, facts = d4)
   numbers alphabet YesNo facts
1        1        a  TRUE    91
2        2        b FALSE    92
3        3        c  TRUE    93
4        4        d FALSE    94
5        5        e  TRUE    95
6        6        f FALSE    96
7        7        g  TRUE    97
8        8        h FALSE    98
9        9        i  TRUE    99
10      10        j FALSE   100
```



```
df1          10 obs. of 4 variables
```

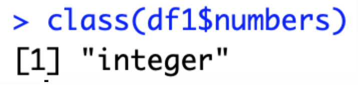
# Data Frames (cont.)

- See class of df1:

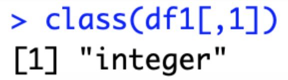


```
> class(df1)
[1] "data.frame"
```

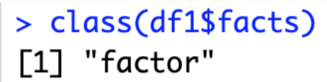
- See class of each variable inside df1 – use $ or [].



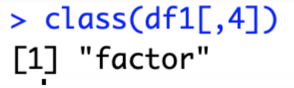
```
> class(df1$numbers)
[1] "integer"
```



```
> class(df1[,1])
[1] "integer"
```

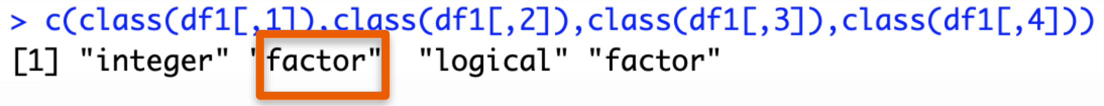


```
> class(df1$facts)
[1] "factor"
```



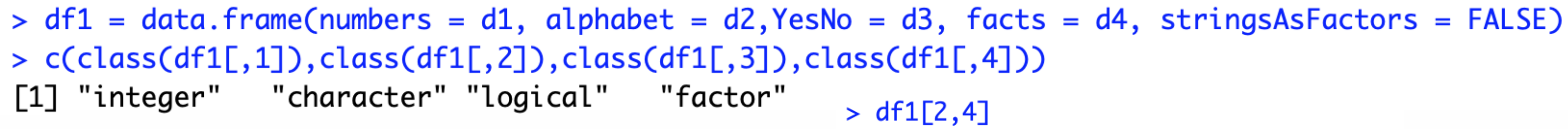
```
> class(df1[,4])
[1] "factor"
```

- See all of them:



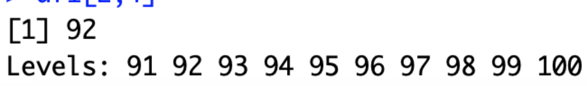
```
> c(class(df1[,1]),class(df1[,2]),class(df1[,3]),class(df1[,4]))
[1] "integer" "factor"  "logical" "factor"
```

- R default: strings are converted to factors when combining to data frame/reading files, etc. Won’t happen with tibbles!
  - Can switch off permanently in *options()*. Be careful!
  - For now, use *stringsAsFactors = FALSE*



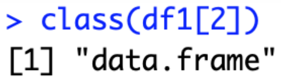
```
> df1 = data.frame(numbers = d1, alphabet = d2,YesNo = d3, facts = d4, stringsAsFactors = FALSE)
> c(class(df1[,1]),class(df1[,2]),class(df1[,3]),class(df1[,4]))
[1] "integer"   "character" "logical"   "factor"
```

- For factors, index will return levels attribute:

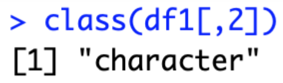


```
> df1[2,4]
[1] 92
Levels: 91 92 93 94 95 96 97 98 99 100
```

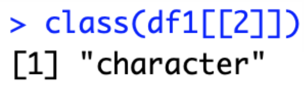
- Referencing in data frames [] vs [[]]:



```
> class(df1[2])
[1] "data.frame"
```



```
> class(df1[,2])
[1] "character"
```

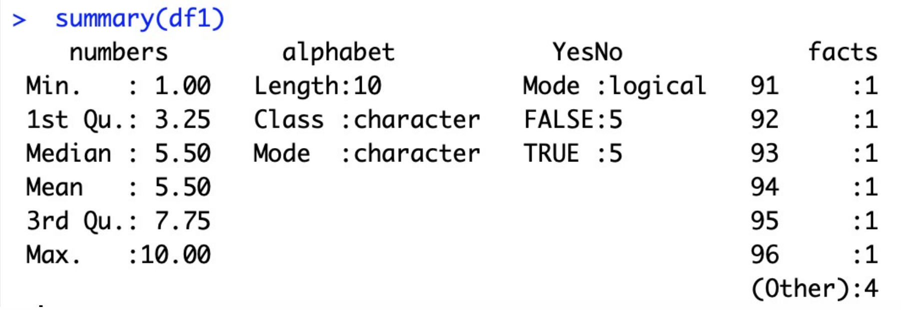


```
> class(df1[[2]])
[1] "character"
```

# Data Frame Functions

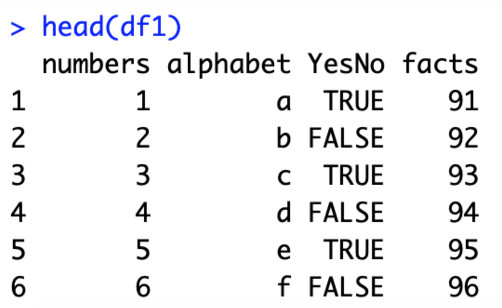
There are some handy functions that you can execute on data frames.

- *summary()* provides concise description of the data.

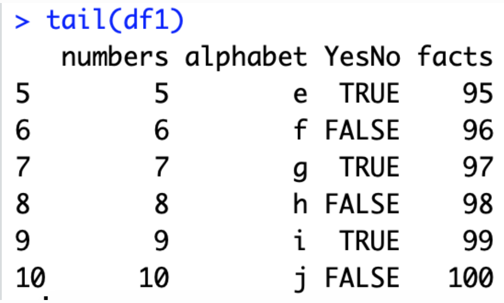


```
> summary(df1)
    numbers        alphabet            YesNo            facts
 Min.   : 1.00   Length:10          Mode :logical   91     :1
 1st Qu.: 3.25   Class :character   FALSE:5         92     :1
 Median : 5.50   Mode  :character   TRUE :5         93     :1
 Mean   : 5.50                                      94     :1
 3rd Qu.: 7.75                                      95     :1
 Max.   :10.00                                      96     :1
                                                    (Other):4
```

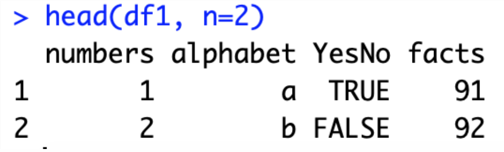
- *head()* and *tail()* shows the first and last 6 rows (observations) of the data, respectively. The number of rows can be changed by *n*.



```
> head(df1)
  numbers alphabet YesNo facts
1       1        a  TRUE    91
2       2        b FALSE    92
3       3        c  TRUE    93
4       4        d FALSE    94
5       5        e  TRUE    95
6       6        f FALSE    96
```

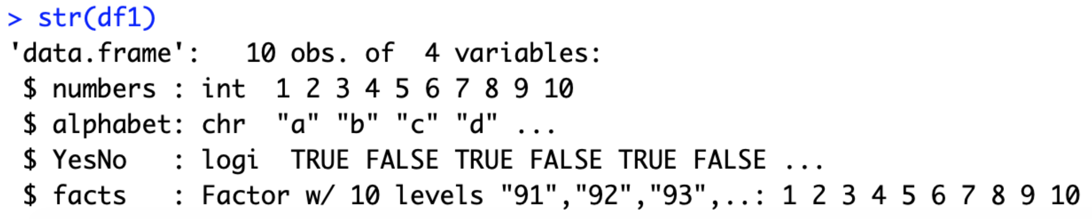


```
> tail(df1)
   numbers alphabet YesNo facts
5        5        e  TRUE    95
6        6        f FALSE    96
7        7        g  TRUE    97
8        8        h FALSE    98
9        9        i  TRUE    99
10      10        j FALSE   100
```



```
> head(df1, n=2)
  numbers alphabet YesNo facts
1       1        a  TRUE    91
2       2        b FALSE    92
```

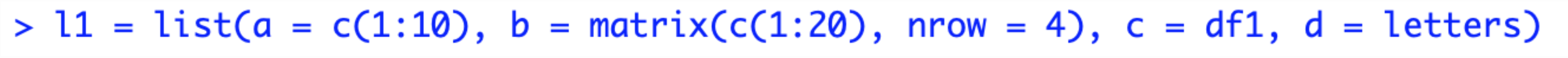
- *str()* shows data, class, names, and size in a compact form.



```
> str(df1)
'data.frame':	10 obs. of  4 variables:
 $ numbers : int  1 2 3 4 5 6 7 8 9 10
 $ alphabet: chr  "a" "b" "c" "d" ...
 $ YesNo   : logi  TRUE FALSE TRUE FALSE TRUE FALSE ...
 $ facts   : Factor w/ 10 levels "91","92","93",..: 1 2 3 4 5 6 7 8 9 10
```

# Lists

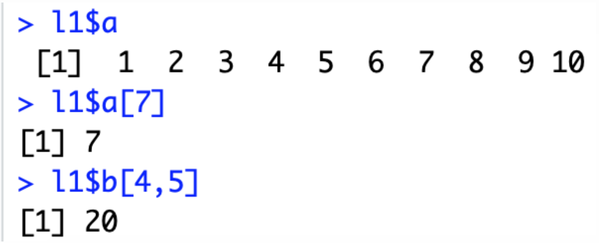
- A “catch-all” object that can be comprised all different data types.
- We are going to combine vastly incompatible objects into a list.



```
> l1 = list(a = c(1:10), b = matrix(c(1:20), nrow = 4), c = df1, d = letters)
```

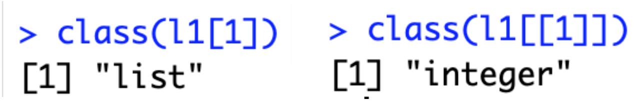
*print(l1)* to see – It works!

- Like data frames, lists can be referenced by $ or [].

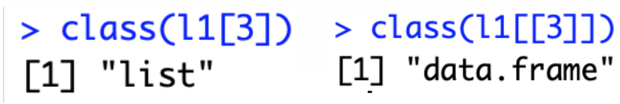


```
> l1$a
 [1]  1  2  3  4  5  6  7  8  9 10
> l1$a[7]
[1] 7
> l1$b[4,5]
[1] 20
```

  - However, [] will return a list. To obtain the original data type, use [[]].



```
> class(l1[1])     > class(l1[[1]])
[1] "list"         [1] "integer"
```



```
> class(l1[3])     > class(l1[[3]])
[1] "list"         [1] "data.frame"
```

# Initializing and Coercing Objects

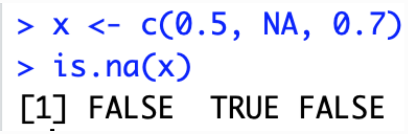
- Conventionally, we initialize objects before assigning values to them.
- Empty objects can be created by *vector()* (default=logical), *matrix()* (default = “NA”), *array()* (default = “NA”), *data.frame()*, and *list()*.
- Specific data type vectors can be created by *logical()*, *character()*, *numeric()*, and *factor().*
- To forcefully convert data types, you can use *as.numeric()*, *as.integer()*, *as.character()*, *as.logical(),* and *as.factor().*
- Like data types, objects can be coerced by *as.vector()*, *as.matrix()*, *as.array()*, *as.data.frame()*, and *as.list()*.

Exercise:

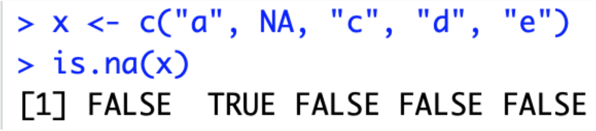
- Create a matrix of 5 rows with 1:15. Convert it to a data frame, vector, list...

# Missing Values

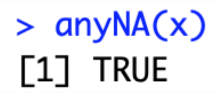
- Missing values are represented by NA (not available) in R.
- Logical function *is.na()* shows if any element is not available.
- Function *anyNA()* returns *TRUE* or *FALSE* for the entire object:
- Some functions can have an argument *na.rm* = *TRUE* that will exclude the missing values from computation.
- Function *na.omit()* removes the entire row (observation) from objects such as matrices and data frames.



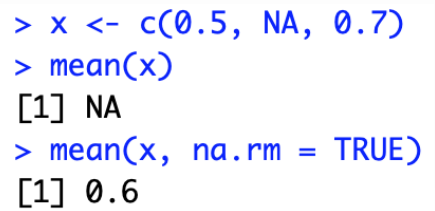
```
> x <- c(0.5, NA, 0.7)
> is.na(x)
[1] FALSE  TRUE FALSE
```



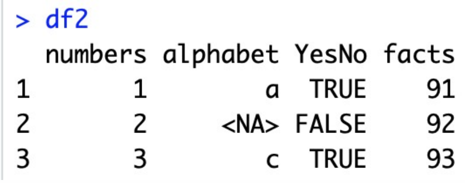
```
> x <- c("a", NA, "c", "d", "e")
> is.na(x)
[1] FALSE  TRUE FALSE FALSE FALSE
```



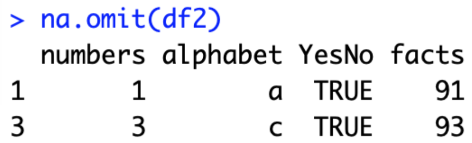
```
> anyNA(x)
[1] TRUE
```



```
> x <- c(0.5, NA, 0.7)
> mean(x)
[1] NA
> mean(x, na.rm = TRUE)
[1] 0.6
```



```
> df2
  numbers alphabet YesNo facts
1       1        a  TRUE    91
2       2     <NA> FALSE    92
3       3        c  TRUE    93
```



```
> na.omit(df2)
  numbers alphabet YesNo facts
1       1        a  TRUE    91
3       3        c  TRUE    93
```