manipalglobal
Academy of Data Science

Operations on Data Types and Data Structures

Table of Content

Introduction

This topic would transition the learner's skill to the next level. It helps the learner in gaining the ability to perform operations such as adding or deleting data items to a vector, list, data frames matrices, and so on. Also, to apply several built-in functions on those data frames.

Learning Objectives

Upon completion of this topic, you will be able to:

- Explain different kind of data types and data structures in detail
- Perform various operations such as appending values and removing elements from a vector, list, data frames, matrices, lists, and so on
- Use built-in commands on data structures such as summary, transpose, structure, and so on

1. Data Types

1.1 Character

Example 1:

Function to type-cast numeric value to a string type:

```
s <- as.character (2.17)
s       # prints the character string
```

Output:

```
[1] "2.17"
```

Example 2:

Function to identify the class of the object **"s"**:

```
class(s)
```

Output:

```
[1] "character"
```

1.2 Integer

Example 1:

A simple program to add 2 numbers (where one number is an integer and other is a decimal value):

```
a=10
b=12.5
c=a+b
print(c)]
```

Output:

```
[1] 22.5
```

Example 2:

Passing a non-decimal string to integer using *as.integer()* function:

```
as.integer("hello")
```

Output:

```
 [1] NA
Warning message: NAs introduced by coercion
```

Passing a non-decimal string to an integer is not recommended.

1.3 Logical

The standard logical operators in R are, "&", "|", "!" (AND, OR, NEGATION)

Example:

The two vectors below (a & b) are assigned logical values TRUE/FALSE and logical operations are performed on both the vectors:

```
a = TRUE
b = FALSE
a & b      #Comment: T&F = F, T&T=T, F&F = F
```

Output:

```
[1] FALSE
a | b          #Comment: T|T =T, T|F=T, F|F=F
```

Output:

```
[1] TRUE
!a
```

Output:

```
[1] FALSE
```

2. Data Structures

2.1 Vectors

Working with Vectors:

- **Adding or appending values to a vector**

Example:

This example demonstrates how **append()** command works on vectors:

```
a = c("Hello", "World" , 1, 2, 3)
append(a,"Hi")
```

The above command will display "Hi" appended along with the original contents of vector (a).

Output:

```
"Hello" "World" "1"     "2"     "3"     "Hi"
```

The contents of vector (a) are printed once again. It is observed that, the last appended value will be missing.

Output:

```
print(a)
[1] "Hello" "World" "1"     "2"     "3"
```

The command used to create a new vector (a) using the append option is:

```
a=append(a,"Hi")
```

Here, an entirely new vector is created with the original values and appended values.

- **String operations on vectors**

Example:

This example demonstrates comparison operator on vectors.

```
mystr = c("b","bc","acdf")
mystr=="a"
          OR
mystr == 'a'
```

Output:

```
False,False,False,False
```

The '= =' operator checks if each element in the left-hand side vector is equal to the element on the right-hand side, and produces a logical vector as a result.

- **Sequence function**

The Sequence function in R generates a sequence of numbers.

Example:

The vector "a" creates a sequence of numbers from 1 to 10.

```
a= 1:10
```

Output:

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

- **Creating a sequence using a stepping factor**

Example 1:

```
a = seq(1,10,2)  #The last argument 2 is the stepping factor
```

Output:

```
[1] 1 3 5 7 9
```

Example 2:

```
a = seq(20,1,-0.5)
```

Output:

```
[1] 20.0 19.5 19.0 18.5 18.0 17.5 17.0 16.5 16.0 15.5 15.0 14.5 14.0
13.5 13.0 12.5
[17] 12.0 11.5 11.0 10.5 10.0 9.5 9.0 8.5 8.0 7.5 7.0 6.5   6.0 5.5 5.0
4.5
[33] 4.0 3.5 3.0 2.5 2.0 1.5 1.0
```

The output starts from 20 and ends at 1.0 with an interval of -0.5.

Example 3:

```
a = seq (10, by = 2, 20)
```

The output starts from 10 and ends at 20 with a step of 2.

Output:

```
[1] 10 12 14 16 18 20
```

- **Repeat Function**

  Repeat function will repeat a number “x”, for n number of times

```
rep(x, ntimes)
```

Example 1:

```
rep(5,10)      #This will repeat the number 5, 10 times.
```

Output:

```
[1] 5 5 5 5 5 5 5 5 5 5
```

Example 2:

```
rep(1:4, 10)  #This will repeat the whole sequence "1234"
ten times.
```

Output:

```
[1] 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
1 2 3 4 1 2 3 4
```

Example 3:

```
rep(1:4, each = 3)  #This command prints each element thrice in the range
1:4.
```

Output:

```
[1]1 1 1 2 2 2 3 3 3 4 4 4
```

Example 4:

```
rep(1:4, each = 3, times = 2 ) # This command prints each element thrice and
the whole sequence is repeated twice.
```

Output:

```
[1] 1 1 1 2 2 2 3 3 3 4 4 4 1 1 1 2 2 2 3 3 3 4 4 4
```

Example 5:

```
rep(1:5, 2)    # 1 2 3 4 5 1 2 3 4 5        # The range of elements are
repeated twice.
```

Output:

```
[1] 1 2 3 4 5 1 2 3 4 5
```

- **Random Numbers**

  The *runif()* command is used to generate random numbers (runif means random uniform).

Example 1:

Command to generate 10 random numbers:

```
a=runif(10)
```

Output:

```
[1] 0.68542857 0.39088707 0.95148238 0.91808714 0.98565325
[6] 0.33171389 0.08166682 0.14584838 0.71889706 0.03982318
```

Example 2:

```
a = runif(10,min=50,max=60) # Generates 10 numbers, randomly starting from
50 and ending at 60.
```

Output:

```
[1] 55.78878 56.55851 51.97267 55.85565 51.29143 50.38351 57.24829 50.61501
51.36529
[10] 59.44149
```

- **Naming a vector elements using "names()" function**

Example 1:

The vector "a" has 3 elements:

```
a=c(1,2,3)
names(a)=c("small","medium","large")
print(a)
```

Output:

```
small medium  large
  1     2       3
```

The names, defined using the **names(a)** vector are assigned to the corresponding vector elements which are demonstrated in the above example.

Example 2:

```
num = c(1:3)
a=c("one", "two" , "three")
names(num)=a
```

Output:

```
>num
One two three
1        2     3
```

- **NA (not available)**

  NA in R represents the Missing values or Not Available values.

Example 1:

```
a=c(1:6)
a[3]=NA  #Comment, the 3rd element in the vector "a" is now replaced by NA
print(a)
```

Output:

```
[1]  1  2 NA  4  5  6
```

- **Identifying the missing values in the vector using the *is.na(<vector_name>)***

Example 2:

```
is.na(a)
```

Output:

```
[1]  1  2 NA  4  5  6
FALSE FALSE  TRUE FALSE FALSE FALSE
```

A new logical vector is created with values such as TRUE/FALSE, where TRUE would be representing missing values (not available) and false would be representing existing values.

- **Omitting NA values**

```
b=na.omit(a)
print(b)
```

Output:

```
[1] 1 2 4 5 6
attr(,"na.action")
[1] 3
attr(,"class")
[1] "omit"
```

This command would omit the NA values from the vector "a". Hence, the value at position 3(NA) has been removed by executing the command **"na.omit()".**

- **Deleting elements from a vector**

```
a=c(1:10)
b=a[-c(5)]
print(b)
```

Output:

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

Vector b now contains all the elements from vector (a) except the 5th element from vector (a). It does not replace it with NA, it is deleted from the vector.

- **Identifying maximum, minimum and median values from a vector**

A vector "a" contains 10 elements. The max, min and median element can be obtained using the commands below:

```
a=c(1:10)
max(a)
min(a)
median(a)
```

Output:

```
> max(a)
[1] 10
> min(a)
[1] 1
> median(a)
[1] 5.5
```

- **Conditional access of vector elements**

  The vector "a" contains 10 elements, the following command is used to find out which of these elements in vector "a" is greater than 5.

```
a=c(1:10)
b=a[(a>5)]   #produces a list of elements greater than 5,     #stored in vector b
a>5 # would result in the generation of a logical vector with #TRUE, FALSE
```

Output:

```
[1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Here, the logical value TRUE means that the condition specified in the command above has been met. Otherwise it returns FALSE.

- **The "which()" function**
  - *which.min(a) #This will print the position of the minimum element in the vector "a"*
  - *which.max(a) #This will print the position of the maximum element in the vector "a"*

Subsetting

The subset function is used to selectively pick elements from an R object.

Working with Subset Function:

- **Using the subset function on a vector**

  The subset function takes in 2 arguments. The example below demonstrates the use of **subset()** function. The first argument is the R data object to be subsetted, and the second argument is a conditional statement.

Example 1:

```
a = 1:20
b = subset(a,a>10)
print(b)
```

Output:

```
[1] 11 12 13 14 15 16 17 18 19 20
```

The subset function has picked only those elements from vector "a" whose elements are greater than 10.

The subset function can also contain logical expressions such as '&' as shown below:

Example 2:

```
a = 1:20
b = subset(a,a<10 & a< 15)
```

Output:

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

It results in a subset which satisfies both the conditions (a<10 and a<15) in the above command.

- **Using the subset function on a data frame**

  USArrests is a built-in dataset in R studio (It is a data frame).

In the example below, the first argument is the data frame which has to be subsetted; the second argument is the condition on which it has to be subsetted, the third argument **select()** is used to select the number of columns to display from the data frame USArrests.

Example 1:

Accessing all the columns based on the condition that the population is greater than 70 million (3rd column), which is assigned to the user-defined data frame *"newarrests"*

```
newarrests = subset(USArrests,UrbanPop >= 70, select =
c(Murder,Assault,UrbanPop,Rape))
print(newarrests)
```

Output:

	Murder	Assault	UrbanPop	Rape
Arizona	8.1	294	80	31.0
California	9.0	276	91	40.6
Colorado	7.9	204	78	38.7
Connecticut	3.3	110	77	11.1
Delaware	5.9	238	72	15.8
Florida	15.4	335	80	31.9
Hawaii	5.3	46	83	20.2
Illinois	10.4	249	83	24.0
Massachusetts	4.4	149	85	16.3
Michigan	12.1	255	74	35.1
Missouri	9.0	178	70	28.2
Nevada	12.2	252	81	46.0
New Jersey	7.4	159	89	18.8
New Mexico	11.4	285	70	32.1
New York	11.1	254	86	26.1
Ohio	7.3	120	75	21.4
Pennsylvania	6.3	106	72	14.9
Rhode Island	3.4	174	87	8.3
Texas	12.7	201	80	25.5
Utah	3.2	120	80	22.9
Washington	4.0	145	73	26.2

Example 2:

```
newarrests = subset(USArrests, UrbanPop >=70 & Rape >= 25,
select=c(UrbanPop,Rape))
```

newarrests is a user defined data frame that contains the subset which meets the above condition specified.

Output:

```
           UrbanPop Rape
Arizona          80 31.0
California       91 40.6
Colorado         78 38.7
Florida          80 31.9
Michigan         74 35.1
Missouri         70 28.2
Nevada           81 46.0
New Mexico       70 32.1
New York         86 26.1
Texas            80 25.5
Washington       73 26.2
```

Example 3:

Sorting the newarrests data frame by the population column.

```
sorted = newarrests[order(newarrests$UrbanPop),]
print(sorted)
```

Output:

```
                UrbanPop  Rape
Missouri        70        28.2
New Mexico      70        32.1
Washington      73        26.2
Michigan        74        35.1
Colorado        78        38.7
Arizona         80        31.0
Florida         80        31.9
Texas           80        25.5
```

```
Nevada          81 46. 0
New York        86 26.1
California      91 40.6
```

Note: For descending ordering of the data frame, use the command decreasing = TRUE.

```
sorted = newarrests[order(newarrests$UrbanPop, decreasing = TRUE),]
print(sorted)
```

- **Set operations**

  `is.element()` - it compares two vectors which returns a logical vector indicating if there is a match or not for its left operand.

Example:

The example below consists of two vectors with a range of values

```
a = c(1:6)
b = c(5:10)
is.element(1,a) # this will return TRUE as element "1" is #present in
vector "a"
```

The above command can also be executed using the below syntax:

```
1 %in% a
is.element(a,b) #this will check if every element in a is present in every
element in b and result in producing a logical  vector
```

- **Union**

  Union will merge the elements of two vectors into a single vector, but the elements will not be repeated.

Example:

```
a = 1:10
b = 5:10
union(a,b) ->c
```

Output:

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

- **Intersection**

  Intersection will select the common elements from vector **a & b** into a new vector.

Example:

```
a = 1:10
b = 5:10
intersect(a,b) ->d
```

Output:

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

- **Any**

  The **any** command will check if there is any element in a vector, which satisfies the given logical condition. The below example demonstrates an example using the **"any"** command:

Example:

```
a = 2
b=c(3,4,5)
c = 1:3
```

```
any(c>b)
```

Output:

```
[1] FALSE
```

- **Sample**

  Sample() command is used to pick random sample of numbers from a vector which is considered to be the source vector.

  The elements are retrieved from the source vector such that each time the ordering will be different.

Example 1:

```
a=c(1:10)
sample(a)
```

Output:

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

To ensure the ordering of the element is same every time, the **set.seed()** command is used.

Example 2:

```
set.seed(100)
sample(a)
set.seed(100)
sample(a)
```

Output:

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

During both occasions, the **sample()** function will produce the sampling from **"a"** in the same order.

Note: *set.seed(100)* can be any number ,instead of 100 any number can be used. However, if the ordering has to be same, one should use the same seed value

Example 3:

```
set.seed(100)
sample(a)
```

Output:

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

The above sampling has produced the same sample ordering as in the example 2.

Example 4:

```
set.seed(50)
sample(a)
```

Output:

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

This example has produced a different sample ordering of elements.

A vector has 5 elements, and a sample of size 10 elements is to be picked, how to do that?

```
a=c(1:5)
b=sample(a,10,replace=T) #replace = T means, the resultant #vector can have
repeated elements in the sample.
```

- **Sorting the elements in a vector in ascending/descending order**

  The default sort order is ascending. This option can be changed to descending order as shown in the below example:

Example:

```
a=c(1:10)
b=sample(a,10)
c = sort(b, decreasing=TRUE)
OR
   c=rev(sort(b))
```

Output:

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

- **Ordering**

  Ordering will give the position of the elements in ascending order. The elements of the vector will not be printed, only the order number will be printed.

Example:

```
b=c(-1,100,2)
order(b)
```

Output:

```
[1] 1 3 2
```

2.2. Matrices

Working with Matrices:

- **Naming the matrix (assigning names to rows and columns)**

  The function **dimnames()** is used for naming the elements in a matrix.

Example:

```
rnames = c("r1","r2") # for naming the row elements
cnames = c("c1","c2") # for naming the column elements
```

```
mymat = matrix(c(1:4),nrow=2,byrow=TRUE,dimnames=list(rnames,cnames))
The rows will be named as r1 and r2 and the columns as c1 and c2
```

Output:

```
    c1 c2
r1   1  2
r2   3  4
```

2.3 Data Frames

Working with Data Frames:

- **Checking the structure of data frame**

  The **str()** command is used to check the structure of data frame.

```
str(df) #df is the name of the data frame, passed as an #argument to the
str() function
```

Example:

The three vectors below (a,b,c) are created, which contains a sequence of numbers and passed to the data frame function

```
a =1:5
 b= 6:10
 c=11:15
 data.frame(a,b,c) -> df
str(df)
```

Output:

```
'data.frame':    5 obs. of 3 variables:
 $ a: int  1 2 3 4 5
 $ b: int  6 7 8 9 10
 $ c: int  11 12 13 14 15
```

Str() function provides information about the number of variables and the type of variables in the data frame.

- **Summary Function**

  The **summary()** function in data frames is used to provide the summary of the data, that is, summary statistics (mean, median, min, max, and so on.)

Example:

```
a =1:5
 b= 6:10
 c=11:15
 data.frame(a,b,c) -> df
summary(df) #df is an example for data frame
```

Output:

```
       a           b           c
 Min.   :1   Min.   : 6   Min.   :11
 1st Qu.:2   1st Qu.: 7   1st Qu.:12
 Median :3   Median : 8   Median :13
 Mean   :3   Mean   : 8   Mean   :13
 3rd Qu.:4   3rd Qu.: 9   3rd Qu.:14
 Max.   :5   Max.   :10   Max.   :15
```

- **Stack() command in data frames**

  The stack command is used to stack all the vectors which are in a data frame, into a single column.

Example:

```
a =1:5
 b= 6:10
 c=11:15
 data.frame(a,b,c) -> df
stack(df)
```

Output:

```
   values ind
1       1   a
2       2   a
3       3   a
4       4   a
5       5   a
6       6   b
7       7   b
8       8   b
9       9   b
10     10   b
11     11   c
12     12   c
13     13   c
14     14   c
15     15   c
```

All the column elements are stacked on top of the other in the above output.

- **Unstack command in data frames**

  The unstack command reverses the above stack command.

```
unstack(stack(df))
```

- **Adding more rows and columns to a data frame**

  The **cbind()** function is used to add more columns to an existing data frame.

Example:

```
a =1:5
 b= 6:10
 c=11:15
 data.frame(a,b,c) -> df
d = 100:104
df=cbind(df,d)
print(df)
```

Output:

```
  a  b  c   d
1 1  6 11 100
2 2  7 12 101
3 3  8 13 102
4 4  9 14 103
5 5 10 15 104
```

"d" is an extra column added to the data frame.

- The **rbind()** command is used to add rows to an existing data frame.

Example:

```
row6 = 1000:1003
df=rbind(df,row6)
           print(df)
```

Output:

```
      a    b    c
1     1    6   11
2     2    7   12
3     3    8   13
4     4    9   14
5     5   10   15
6  1000 1001 1002
```

In the above example, an extra row 6 has been added to the data frame.

- **Viewing a large data frame which does not fit into the R console**

  The **fix** command will present the contents of the data frame in a separate Excel style window.

```
fix (name of the data frame)
```

- **Obtaining the transpose (interchange of rows and columns) of a data frame**

Example:

```
12              13
34   --transpose-->  24
```

`t()` function is used to obtain the transpose of a data frame

Syntax: `t(name of the data frame)`

After a matrix is transposed and stored in another object, the class of that object would be a matrix instead of data frame.

The **as.data.frame(object name)** is used to convert it back to data frame.

- **Omitting the missing values in a data frame (NA)**

Omit() command is used to remove the NA values in a data frame.

```
na.omit(name of the data frame)
```

Example:

In the example below, three vectors are created and passed to a data frame, the third and 4th element in the vector v1 contains NA values.

```
v1 = 1:5
v2=6:10
v3=11:15
v1[3]=NA
v3[4]=NA
mydf = data.frame(v1,v2,v3)
print(mydf)
na.omit(mydf)
```

Output:

```
   v1 v2 v3
1   1  6 11
2   2  7 12
3  NA  8 13
```

```
4  4  9 NA
5  5 10 15  #NA values are present when the data frame

 v1 v2 v3
1  1  6 11
2  2  7 12
5  5 10 15 #NA values are removed after the na.omit()operation
```

Note: In the output seen above, the entire row which contains NA will be ignored.

- **"which()" command on a data frame**

  Demonstration of the **which()** function on USArrests data frame - The rows whose Urban Population is greater than 75 are accessed.

```
USArrest[which(USArrest$UrbanPop>75),]
```

Output:

```
                Murder Assault UrbanPop Rape
Arizona            8.1     294       80 31.0
California         9.0     276       91 40.6
Colorado           7.9     204       78 38.7
Connecticut        3.3     110       77 11.1
Florida           15.4     335       80 31.9
Hawaii             5.3      46       83 20.2
Illinois          10.4     249       83 24.0
Massachusetts      4.4     149       85 16.3
Nevada            12.2     252       81 46.0
New Jersey         7.4     159       89 18.8
New York          11.1     254       86 26.1
Rhode Island       3.4     174       87  8.3
Texas             12.7     201       80 25.5
Utah               3.2     120       80 22.9
```

- **Using the "&" operator in the which() function (extending the above example)**

```
USArrest[which(USArrest$UrbanPop>75&USArrest$Rape>12),]
```

Note: On satisfying the conditions *UrbanPop>75 and Rape>12*, the rows from the USArrests are selected.

Output:

```
             Murder Assault UrbanPop Rape
Arizona        8.1     294       80 31.0
California     9.0     276       91 40.6
Colorado       7.9     204       78 38.7
Florida       15.4     335       80 31.9
Hawaii         5.3      46       83 20.2
Illinois      10.4     249       83 24.0
Massachusetts  4.4     149       85 16.3
Nevada        12.2     252       81 46.0
New Jersey     7.4     159       89 18.8
New York      11.1     254       86 26.1
Texas         12.7     201       80 25.5
Utah           3.2     120       80 22.9
```

2.4 Lists

Working with Lists:

- **Naming the list elements**

  Naming of the list is similar to naming of vectors.

Example:

```
v1 =c("true","false");
v2= 1:5
v3 = c("cat","dog")
my_list=list(v1,v2,v3)
mynames = c("A boolen vector","An integer vector","Animals string")
names(my_list) = mynames
```

- **Checking the contents of my_list:**

```
print(my_list)
```

Output:

```
$`A boolen vector`
[1] "true"  "false"
$`An integer vector`
[1] 1 2 3 4 5
$`Animals string`
[1] "cat" "dog"
```

- **To unlist the list elements the below command is used:**

```
unlist(my_list)
```

`unlist` command is used to change the list into a flat vector.

Output:

```
A boolen vector1   A boolen vector2 An integer vector1 An integer vector2
"true"      "false"         "1"         "2"
An integer vector3 An integer vector4 An integer vector5    Animals string1
  "3"          "4"           "5"         "cat"
   Animals string2
   "dog"
```

- **Accessing the list elements using the names of the elements**

Once the names have been assigned to the list elements, the elements of a particular list can be accessed using the below syntax:

```
print(my_list$"Animals string")
```

Output:

```
[1] "cat" "dog"
```

- **List manipulation (add, delete and update list elements)**

  Adding an item to the list - A new item is added to the existing list by specifying the position of the last element in the list.

  The example below demonstrates the adding of an item to a list.

Example:

```
v1 =c("true","false");
v2= 1:5
v3 = c("cat","dog")
my_list=list(v1,v2,v3)
my_list[4] = "Hello"
```

Output:

```
[[1]]
[1] "true"  "false"
[[2]]
[1] 1 2 3 4 5
[[3]]
[1] "cat" "dog"
[[4]]
[1] "Hello"
```

The item “hello” has been added as the fourth element in the list.

- **Deleting an element from the list**

  NULL command is used to delete an element from the list.

```
my_list[4] = NULL
```

The above statement will remove an element from the end of the list.

Output:

```
[[1]]
[1] "true"  "false"

[[2]]
```

```
[1] 1 2 3 4 5

[[3]]
[1] "cat" "dog"
```

Note: The last element (the 4th element) has been deleted!

- **Mering the lists**

  The **c()** function is used to merge two lists.

  The example below consists of two lists "l1" and "l2"

Example:

```
l1 <- list(10,20,30,40)
l2 <- list("Sun","Mon","Tue","Wed")

merg_list <- c(l1,l2)
print(merg_list)
```

Output:

```
[[1]]
[1] 10

[[2]]
[1] 20

[[3]]
[1] 30

[[4]]
[1] 40

[[5]]
[1] "Sun"

[[6]]
```

```
[1] "Mon"

[[7]]
[1] "Tue"

 [[8]]
[1] "Wed"
```

The final output shows that the two lists have been merged.

Summary

The different data types discussed in this topic are:

- Character
- Integer
- Logical

The different data structures discussed in this topic are:

- Vector
- Matrix
- Data frame
- Lists

