Control Statements

- **Control statements** are expressions used to control the execution and flow of the program based on the conditions provided in the statements. These structures are used to make a decision after assessing the variable.

In R programming, there are 8 types of control statements as follows:

- if condition
- if-else condition
- for loop
- nested loops
- while loop
- repeat and break statement
- return statement
- next statement

➢ if condition

This control structure checks the expression provided in parenthesis is true or not. If true, the execution of the statements in braces {} continues.

Syntax:

```
if(expression){
   statements
   ....
   ....
}
```

Example:

```
x <- 100


if(x > 10){
print(paste(x, "is greater than 10"))
}
```

➢ if-else condition

It is similar to **if** condition but when the test expression in if condition fails, then statements in **else** condition are executed.

Syntax:

```
if(expression){
   statements
   ....
   ....
}
else{
   statements
   ....
   ....
}
```

Example:

```
x <- 5

# Check value is less than or greater than 10
if(x > 10){
print(paste(x, "is greater than 10"))
}else{
print(paste(x, "is less than 10"))
}
```

➢ for loop

It is a type of loop or sequence of statements executed repeatedly until exit condition is reached.

Syntax:

```
for(value in vector){
    statements
    ....
    ....
}
```

Example:

```
x <- letters[4:10]

for(i in x){
print(i)
}
```

Output:

```
[1] "d"
[1] "e"
[1] "f"
[1] "g"
[1] "h"
[1] "i"
[1] "j"
```

➢ Nested loops

Nested loops are similar to simple loops. Nested means loops inside loop. Moreover, nested loops are used to manipulate the matrix.

Example:

```
# Defining matrix
m <- matrix(2:15, 2)


for (r in seq(nrow(m))) {
for (c in seq(ncol(m))) {
    print(m[r, c])
}
}
```

Output:

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

➢ while loop

while loop is another kind of loop iterated until a condition is satisfied. The testing expression is checked first before executing the body of loop.

Syntax:

```
while(expression){
    statement
    ....
    ....
}
```

Example:

```
x = 1
# Print 1 to 5
while(x <= 5){
print(x)
x = x + 1
}
```

Output:

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

➢ Repeat loop and break statement

Repeat is a loop which can be iterated many number of times but there is no exit condition to come out from the loop. So, break statement is used to exit from the loop. **break** statement can be used in any type of loop to exit from the loop.

Syntax:

```
repeat {
   statements
   ....
   ....
   if(expression)  {
      break
   }
}
```

Example:

```
x = 1

# Print 1 to 5
repeat{
print(x)
x = x + 1
if(x > 5){
    break
}
}
```

Output:

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

➢ Return statement

Return statement is used to return the result of an executed function and returns control to the calling function.

Syntax:

return(expression)

Example:

```
# Checks value is either positive, negative or zero
func <- function(x){
if(x > 0){
    return("Positive")
}else if(x < 0){
    return("Negative")
}else{
    return("Zero")
}}
func(1)
func(0)
func(-1)
```

Output:

```
[1] "Positive"
[1] "Zero"
[1] "Negative"
```

➢ Next statement

Next statement is used to skip the current iteration without executing the further statements and continues the next iteration cycle without terminating the loop.

Example:

```
# Defining vector
x <- 1:10


# Print even numbers
for(i in x){
if(i%%2 != 0){
    next #Jumps to next loop
}
print(i)
}
```

Output:

```
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
```

Arithmetic and Boolean Operators and Values

Basic R Operators

Operation	Description
`x + y`	Addition
`x - y`	Subtraction
`x * y`	Multiplication
`x / y`	Division
`x ^ y`	Exponentiation
`x %% y`	Modular arithmetic
`x %/% y`	Integer division
`x == y`	Test for equality
`x <= y`	Test for less than or equal to
`x >= y`	Test for greater than or equal to
`x && y`	Boolean AND for scalars
`x || y`	Boolean OR for scalars
`x & y`	Boolean AND for vectors (vector x,y,result)
`x | y`	Boolean OR for vectors (vector x,y,result)
`!x`	Boolean negation

Types of the operator in R language

- Arithmetic Operators
- Logical Operators
- Relational Operators
- Assignment Operators
- Miscellaneous Operator

Arithmetic Operators

```
# R program to illustrate the use of Arithmetic operators
vec1 <- c(0, 2)
vec2 <- c(2, 3)

# Performing operations on Operands
cat ("Addition of vectors :", vec1 + vec2, "\n")
cat ("Subtraction of vectors :", vec1 - vec2, "\n")
cat ("Multiplication of vectors :", vec1 * vec2, "\n")
cat ("Division of vectors :", vec1 / vec2, "\n")
cat ("Modulo of vectors :", vec1 %% vec2, "\n")
cat ("Power operator :", vec1 ^ vec2)
```

Output:

```
Addition of vectors : 2 5
Subtraction of vectors : -2 -1
Multiplication of vectors : 0 6
Division of vectors : 0 0.6666667
Modulo of vectors : 0 2
Power operator : 0 8
```

R Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description
&	Element-wise Logical AND operator. It returns TRUE if both elements are TRUE
&&	Logical AND operator - Returns TRUE if both statements are TRUE
|	Elementwise- Logical OR operator. It returns TRUE if one of the statement is TRUE
||	Logical OR operator. It returns TRUE if one of the statement is TRUE.
!	Logical NOT - returns FALSE if statement is TRUE

Relational Operators

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

R Assignment Operators

Assignment operators are used to assign values to variables:

Example

```
my_var <- 3

my_var <<- 3

3 -> my_var

3 ->> my_var

my_var # print my_var
```

R Miscellaneous Operators

Miscellaneous operators are used to manipulate data:

Operator	Description	Example
:	Creates a series of numbers in a sequence	x <- 1:10
%in%	Find out if an element belongs to a vector	x %in% y
%*%	Matrix Multiplication	x <- Matrix1 %*% Matrix2

%in% Operator:

Checks if an element belongs to a list and returns a boolean value TRUE if the value is present else FALSE.

Input : val <- 0.1

```
list1 <- c(TRUE, 0.1,"apple")
print (val %in% list1)
```

Output : TRUE

Checks for the value 0.1 in the specified list. It exists, therefore, prints TRUE

Colon Operator(:) :

Prints a list of elements starting with the element before the color to the element after it.

Input : print (1:5)

Output : 1 2 3 4 5

Prints a sequence of the numbers from 1 to 5.

%*% Operator:

This operator is used to multiply a matrix with its transpose. Transpose of the matrix is obtained by interchanging the rows to columns and columns to rows. The number of columns of first matrix must be equal to number of rows of second matrix. Multiplication of the matrix A with its transpose, B, produce a square matrix.

Input :

```
mat = matrix(c(1,2,3,4,5,6),nrow=2,ncol=3)
print (mat)
print( t(mat))
pro = mat %*% t(mat)
print(pro)
```

Output :

```
     [,1] [,2] [,3]     #original matrix of order 2x3
[1,]    1    3    5
[2,]    2    4    6

     [,1] [,2]          #transposed matrix of order 3x2
[1,]    1    2
[2,]    3    4
[3,]    5    6

     [,1] [,2]          #product matrix of order 2x2
[1,]   35   44
[2,]   44   56
```

Functions in R

- Functions are useful when you want to perform a certain task multiple times.

- A function accepts input arguments and produces the output by executing valid R commands that are inside the function.

- In R Programming Language when you are creating a function the function name and the file in which you are creating the function need not be the same and you can have one or more function definitions in a single R file.

Types of function in R Language

- **Built-in Function:** Built function R is sq(), mean(), max(), these function are directly call in the program by users.

- **User-defile Function:** R language allow us to write our own function.

Built-in Function

Example :

```
# Find sum of numbers 4 to 6.
print(sum(4:6))

# Find max of numbers 4 and 6.
print(max(4:6))

# Find min of numbers 4 and 6.
print(min(4:6))
```

Output:

```
[1] 15
[1] 6
[1] 4
```

User-defined Functions

- R provides built-in functions like **print()**, **cat()**, etc. but we can also create our own functions. These functions are called user-defined functions.

Example:

```
# A simple R function to check
# whether x is even or odd
evenOdd = function(x){
if(x %% 2 == 0)
    return("even")
else
    return("odd")
}
print(evenOdd(4))
print(evenOdd(3))
```

Single Input Single Output

- Create a function in R that will take a single input and gives us a single output.
- Following is an example to create a function that calculates the area of a circle which takes in the arguments the radius. So, to create a function, name the function as "areaOfCircle" and the arguments that are needed to be passed are the "radius" of the circle.

Example:

```
# A simple R function to calculate area of a circle
areaOfCircle = function(radius){
area = pi*radius^2
return(area)
}
print(areaOfCircle(2))
```

Multiple Input Multiple Output

Example:

```
# A simple R function to calculate area and perimeter of a rectangle
Rectangle = function(length, width){
area = length * width
perimeter = 2 * (length + width)
# create an object called result which is a list of area and perimeter
result = list("Area" = area, "Perimeter" = perimeter)
return(result)
}
resultList = Rectangle(2, 3)
print(resultList["Area"])
print(resultList["Perimeter"])
```

Function Arguments in R

- Arguments are the parameters provided to a function to perform operations in a programming language.

Adding Arguments in R

- We can pass an argument to a function while calling the function by simply giving the value as an argument inside the parenthesis.

Example:

```
# Function definition To check n is divisible by 5 or not
divisbleBy5 <- function(n){
if(n %% 5 == 0)
{
    return("number is divisible by 5")
}
else
{
    return("number is not divisible by 5")
}
}
# Function call
divisbleBy5(100)
divisbleBy5(4)
divisbleBy5(20.0)
```

Output:

```
[1] "number is divisible by 5"
[1] "number is not divisible by 5"
[1] "number is divisible by 5"
```

Adding Multiple Arguments in R

- A function in R programming can have multiple arguments too.

```
# Function definition To check a is divisible by b or not
divisible <- function(a, b){
if(a %% b == 0)
{
   return(paste(a, "is divisible by", b))
}
else
{
   return(paste(a, "is not divisible by", b))
}}
# Function call
divisible(7, 3)
divisible(36, 6)
divisible(9, 2)
```

Output:

```
[1] "7 is not divisible by 3"
[1] "36 is divisible by 6"
[1] "9 is not divisible by 2"
```

Default Values for Argument

- Default arguments are arguments which take default values when those arguments not passed at the time of function calling.

- Function can have default values of it's parameters. It means while defining a function, it's parameters can be set values.

- Default value in a function is a value that is not required to specify each time the function is called. If the value is passed by the user, then the user-defined value is used by the function otherwise, the default value is used.

Example :

```
# Function definition to check a is divisible by b or not. If b is not provided in
   function call, Then divisibility of a is checked with 3 as default
divisible <- function(a, b = 3){
if(a %% b == 0)
{
   return(paste(a, "is divisible by", b))
}
else
{
   return(paste(a, "is not divisible by", b))
}
}
                              # Function call
                              divisible(10, 5)
                              divisible(12)
```

Output:

```
[1] "10 is divisible by 5"
[1] "12 is divisible by 3"
```

Function as Argument

In R programming, functions can be passed to another functions as arguments.

Example:

```
# Function definition
# Function is passed as argument
fun <- function(x, fun2){
return(fun2(x))
}

# sum is built-in function
fun(c(1:10), sum)
```

Recursive Functions

- Recursion, in the simplest terms, is a type of looping technique.
- Recursion is when the function calls itself. This forms a loop, where every time the function is called, it calls itself again and again and this technique is known as recursion.
- Since the loops increase the memory we use the recursion. The recursive function uses the concept of recursion to perform iterative tasks they call themselves, again and again, which acts as a loop.
- These kinds of functions need a stopping condition so that they can stop looping continuously.
- They break down the problem into smaller components. The function() calls itself within the original function() on each of the smaller components.
- After this, the results will be put together to solve the original problem.

Example: Factorial using Recursion in R

```
rec_fac <- function(x){
    if(x==0 || x==1)
    {
        return(1)
    }
    else
    {
        return(x*rec_fac(x-1))
    }
}
```

Conversion Functions in R

- Sometimes to analyze data using R, we need to convert data into another data type.

- R has the following data types Numeric, Integer, Logical, Character, etc.

- Similarly R has various conversion functions that are used to convert the data type

In R, Conversion Function are of two types:

- **Conversion Functions for Data Types**

- **Conversion Functions for Data Structures**

Conversion Functions For Data Types

There are various conversion functions available for Data Types.

These are:

- **as.numeric()**

  Decimal value known numeric values in R. It is the default data type for real numbers in R.

  In R as.numeric() converts any values into numeric values.

  Syntax:

// Conversion into numeric data type

as.numeric(x)

Example:

```
# A simple R program to convert
# character data type into numeric data type
x<-c('1', '2', '3')

# Print x
print(x)

# Print the type of x
print(typeof(x))

# Conversion into numeric data type
y<-as.numeric(x)

# print the type of y
print(typeof(y))
```

Output:

```
[1] "1" "2" "3"
[1] "character"
[1] "double"
```

- **as.integer()**

In R, Integer data type is a collection of all integers. In order to create an integer variable in R and convert any data type in to Integer we use as.integer() function.

Syntax:

// Conversion of any data type into Integer data type

as.integer(x)

Output:

```
[1]  1.3  5.6 55.6
[1] "double"
[1]  1  5 55
[1] "integer"
```

```
# A simple R program to convert
# numeric data type into integer data type
x<-c(1.3, 5.6, 55.6)

# Print x
print(x)

# Print type of x
print(typeof(x))

# Conversion into integer data type
y<-as.integer(x)

# Print y
print(y)

# Print type of y
print(typeof(y))
```

- **as.character()**

In R, character data is used to store character value and string.

To create an character variable in R, we invoke the as.character() function and also if we want to convert any data type in to character we use as.character() function.

Syntax:

// Conversion of any data type into character data type

as.character(x)

```
x<-c(1.3, 5.6, 55.6)

# Print x
print(x)

# Print type of x
print(typeof(x))

# Conversion into character data type
y<-as.character(x)

# Print y
print(y)

# Print type of y
print(typeof(y))
```

Output:

```
[1]  1.3  5.6 55.6
[1] "double"
[1] "1.3"  "5.6"  "55.6"
[1] "character"
```

- **as.logical()**
  Logical value is created to compare variables which return either true or false.To compare variables and to convert any value in to true or false, R uses as.logical() function.

Syntax:

```
// Conversion of any data type into logical data type
as.logical(x)
```

Example:

```
x = 3
y = 8
# Conversion in to logical value
result<-as.logical(x>y)
# Print result
print(result)
```

- **as.date()**

In R as.date() function is used to convert string into date format.

Syntax:

```
// Print string into date format
as.date(variable, "%m/%d/%y")
```

Example:

```
dates <- c("02/27/92", "02/27/92", "01/14/92", "02/28/92", "02/01/92")
# Conversion into date format
result<-as.Date(dates, "%m/%d/%y")
# Print result
print(result)
```

Conversion Functions For Data Structure

- **as.data.frame()**

Data Frame is used to store data tables. Which is list of vectors of equal length.

In R, sometimes to analyse data we need to convert list of vector into data.frame.

So for this R uses as.data.frame() function to convert list of vector into data frame.

Syntax:

// Conversion of any data structure into data frame

as.data.frame(x)

Example :

```
x<- list( c('a', 'b', 'c'),
c('e', 'f', 'g'), c('h', 'i', 'j'))

# Print x
print(x)

# Conversion in to data frame
y<-as.data.frame(x)

# Print y
print(y)
```

Output:

```
[[1]]
[1] "a" "b" "c"

[[2]]
[1] "e" "f" "g"

[[3]]
[1] "h" "i" "j"

  c..a....b....c.. c..e....f....g.. c..h....i....j..
1                a                e                h
2                b                f                i
3                c                g                j
```

- **as.vector()**

R has a function as.vector() which is used to convert a distributed matrix into a non-distributed vector. Vector generates a vector of the given length and mode.

Syntax:

```
// Conversion of any data structure into vector

as.vector(x)
```

Example:

```
x<-c(a=1, b=2)

# Print x

print(x)

# Conversion into vector

y<-as.vector(x)

# Print y

print(y)
```

Output:

```
a b
1 2
[1] 1 2
```

- **as.matrix()**

In R, there is a function as.matrix() which is used to convert a data.table into a matrix, optionally using one of the columns in the data.table as the matrix row names.

Syntax:

```
// Conversion into matrix
as.matrix(x)
```

Example:

```
# Importing library
library(data.table)
x <- data.table(A = letters[1:5], X = 1:5, Y = 6:10)

# Print x
print(x)

# Conversion into matrix
z<-as.matrix(x)

# Print z
print(z)
```

Output:

```
   A X  Y
1: a 1  6
2: b 2  7
3: c 3  8
4: d 4  9
5: e 5 10
     A   X   Y
[1,] "a" "1" " 6"
[2,] "b" "2" " 7"
[3,] "c" "3" " 8"
[4,] "d" "4" " 9"
[5,] "e" "5" "10"
```

Environment and Scope Issues

R Programming Environment

- Environment can be thought of as a collection of objects (functions, variables etc.).
- An environment is created when we first fire up the R interpreter.
- Any variable we define, is now in this environment.
- The top level environment available to us at the R command prompt is the global environment called R_GlobalEnv.
- **Global environment** can be referred to as .GlobalEnv in R codes as well.
- We can use the ls() function to show what variables and functions are defined in the current environment.
- Moreover, we can use the environment() function to get the current environment.

```
> a <- 2
> b <- 5
> f <- function(x) x<-0
> ls()
[1] "a" "b" "f"
> environment()
<environment: R_GlobalEnv>
> .GlobalEnv
<environment: R_GlobalEnv>
```

- In the above example, we can see that a, b and f are in the R_GlobalEnv environment.
- In the above example, the function f creates a new environment inside the global environment.
- Hence, x is in the frame of the new environment created by the function f. This environment will also have a pointer to R_GlobalEnv.

Scope of variable

Global variables

- Global variables are those variables which exists throughout the execution of a program. It can be changed and accessed from any part of the program.
- However, global variables also depend upon the perspective of a function.
- For example, from the perspective of inner_func(), both a and b are **global** variables.
- However, from the perspective of outer_func(), b is a local variable and only a is global variable. The variable c is completely invisible to outer_func().

Example

```
outer_func <- function(){
b <- 20
inner_func <- function(){
c <- 30
}
 }
a <- 10
```

Local variables

- On the other hand, Local variables are those variables which exist only within a certain part of a program like a function, and is released when the function call ends.

- In the above program the variable c is called a local variable.

- If we assign a value to a variable with the function inner_func(), the change will only be local and cannot be accessed outside the function.

- This is also the same even if names of both global variable and local variables matches.

Example

```
outer_func <- function(){
a <- 20
inner_func <- function(){
a <- 30
print(a)
}
inner_func()
print(a)
}
```

Replacement Functions

To replace specific values in a column of DataFrame there are two methods

- **Using Replace() function.**
- **Using the logical condition.**

Method 1: Using Replace() function.

- replace() function in R Language is used to replace the values in the specified string vector x with indices given in list by those given in values.

Syntax: *replace(list , position , replacement_value)*

- It takes on three parameters first is the list name, then the index at which the element needs to be replaced, and the third parameter is the replacement values.

Example:

```
# List of Names
Names<-c("Suresh","Sita","Anu","Manasa", "Riya","Ramesh","Roopa","Neha")
# Roll numbers
Roll_No<-1:8
# Marks obtained
Marks<-c(15, 20, 3, -1, 14, -2 , 10, 13)
Full_Marks<-c(20, 10, 44, 21, 24, 36, 20, 13)
# df name of data frame Converting the list into dataframe
df<-data.frame(Roll_No,Names, Marks, Full_Marks)
print("Original DF")
print(df)
print("Replaced Value")
#replaces the negative numbers with zeros
data<-replace(df$Marks, df$Marks<0, 0)
print(data)
```

Output:

```
[1] "Original DF"
  Roll_No  Names Marks Full_Marks
1       1 Suresh    15          20
2       2   Sita    20          10
3       3    Anu     3          44
4       4 Manasa    -1          21
5       5   Riya    14          24
6       6 Ramesh    -2          36
7       7  Roopa    10          20
8       8   Neha    13          13
[1] "Replaced Value"
15 20 3 0 14 0 10 13
```

Method 2: Using the logical condition.

Example:

```
# List of Names
Names<-c("Suresh","Sita","Anu","Manasa",
         "Riya","Ramesh","Roopa","Neha")
# Roll numbers
Roll_No<-1:8
# Marks obtained
Marks<-c(15, 20, 3, 11, 14, 16, 10, 13)
# df name of data frame
# Converting the list into dataframe
df<-data.frame(Roll_No,Names,Marks)
print(df)
```

Roll_No	Names	Marks
1	Suresh	15
2	Sita	20
3	Anu	3
4	Manasa	11
5	Riya	14
6	Ramesh	16
7	Roopa	10
8	Neha	13

- We replace specific values in the column.

- data frame **“Sita”** marks are given as **20** let us replace it with **25**.

Syntax: *dataframe_name$column_name1[dataframe_name$column_name2==y]<-x*

Parameters:

- *y:It is the value which help us to fetch the data location the column*

- *x: It is the value which needs to be replaced*

```
# List of Names
Names<-c("Suresh","Sita","Anu","Manasa",
        "Riya","Ramesh","Roopa","Neha")
# Roll numbers
Roll_No<-1:8
# Marks obtained
Marks<-c(15, 20, 3, 11, 14, 16, 10, 13)
# df name of data frame
# Converting the list into dataframe
df<-data.frame(Roll_No,Names, Marks)
df$Marks[df$Names == "Sita"] <- 25
# Column_name1 is Marks which we need to replace
# Print the modified data frame
print(df)
```

```
  Roll_No  Names Marks
1       1 Suresh    15
2       2   Sita    25
3       3    Anu     3
4       4 Manasa    11
5       5   Riya    14
6       6 Ramesh    16
7       7  Roopa    10
8       8   Neha    13
```

MATH AND SIMULATIONS IN R

Math Functions

- **exp()**: Exponential function, base e
- **log()**: Natural logarithm
- **log10()**: Logarithm base 10
- **sqrt()**: Square root
- **abs()**: Absolute value
- **sin(), cos(), and so on**: Trig functions
- **min() and max()**: Minimum value and maximum value within a vector
- **which.min() and which.max()**: Index of the minimal element and maximal element of a vector
- **pmin() and pmax()**: Element-wise minima and maxima of several vectors
- **sum() and prod()**: Sum and product of the elements of a vector
- **cumsum() and cumprod()**: Cumulative sum and product of the elements of a vector
- **round(), floor(), and ceiling()**: Round to the closest integer, to the closest integer below, and to the closest integer above
- **factorial()**: Factorial function

Cumulative Sums and Products

- The functions cumsum() and cumprod() return cumulative sums and products.

```
> x <- c(12,5,13)
> cumsum(x)
[1] 12 17 30
> cumprod(x)
[1] 12 60 780
```

- In x, the sum of the first element is 12, the sum of the first two elements is 17, and the sum of the first three elements is 30.

- The function cumprod() works the same way as cumsum(), but with the product instead of the sum.

Minima and Maxima

- There is quite a difference between min() and pmin().
- The former simply combines all its arguments into one long vector and returns the minimum value in that vector.
- In contrast, if pmin() is applied to two or more vectors, it returns a vector of the pair-wise minima, hence the name pmin.

```
> z
     [,1] [,2]
[1,]    1    2
[2,]    5    3
[3,]    6    2
> min(z[,1],z[,2])
[1] 1
> pmin(z[,1],z[,2])
[1] 1 3 2
```

Sorting

- Sorting of a vector can be done with the sort() function, as in this example:

```
> x <- c(13,5,12,5)
> sort(x)
[1] 5 5 12 13
> x
[1] 13 5 12 5
```

- Note that x itself did not change.
- If you want the indices of the sorted values in the original vector, use the order() function.
- Example:

```
 > order(x)
[1] 2 4 3 1
```

```
> y
    V1 V2
1  def  2
2   ab  5
3 zzzz  1
> r <- order(y$V2)
> r
[1] 3 1 2
> z <- y[r,]
> z
    V1 V2
3 zzzz  1
1  def  2
2   ab  5
```

```
> d
   kids ages
1  Jack   12
2  Jill   10
3 Billy   13
> d[order(d$kids),]
   kids ages
3 Billy   13
1  Jack   12
2  Jill   10
> d[order(d$ages),]
   kids ages
2  Jill   10
1  Jack   12
3 Billy   13
```

Set Operations

R includes some handy set operations, including these:

- union(x,y): Union of the sets x and y
- intersect(x,y): Intersection of the sets x and y
- setdiff(x,y): Set difference between x and y, consisting of all elements of x that are not in y
- setequal(x,y): Test for equality between x and y
- c %in% y: Membership, testing whether c is an element of the set y

```
> x <- c(1,2,5)
> y <- c(5,1,8,9)
> union(x,y)
[1] 1 2 5 8 9
> intersect(x,y)
[1] 1 5
> setdiff(x,y)
[1] 2
> setdiff(y,x)
[1] 8 9
> setequal(x,y)
[1] FALSE
> setequal(x,c(1,2,5))
[1] TRUE
> 2 %in% x
[1] TRUE
> 2 %in% y
[1] FALSE
```