```
> gl(5,3, labels = c("one", "two", "three", "four", "five"))
 [1] one   one   one   two   two   two   three three three four  four  four  five
[14] five  five
Levels: one two three four five
> gl(5,1,15)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
 Levels: 1 2 3 4 5
```

The factors thus generated can be combined using the function interaction() to get a resultant combined factor.

```
> fac1 <- gl(5,3, labels = c("one", "two", "three", "four", "five"))
> fac2 <- gl(5,1,15, labels = c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j",
                                "k", "l", "m", "n", "o"))
> interaction(fac1, fac2)
 [1] one.a   one.b   one.c   two.d   two.e   two.a   three.b three.c three.d four.e
[11] four.a  four.b  five.c  five.d  five.e
75 Levels: one.a two.a three.a four.a five.a one.b two.b three.b four.b ... five.o
```

2.7. Strings

Strings are stored in character vectors. Most string manipulation functions act on character vectors. Character vectors can be created using the *c()* function by enclosing the string in double or single quotes. (Generally we follow only double quotes). The *paste()* function can be used to concatenate two strings with a space in between. If the space need not be shown, we use the function *paste0()*. To have specified separator between the two concatenated string, we use the argument sep in the *paste()* function. The result can be collapsed into one string using the collapse argument.

```
> c("String 1", 'String 2')
[1] "String 1" "String 2"
> paste(c("Pine", "Red"), "Apple")
```

```
[1] "Pine Apple" "Red Apple"
> paste0(c("Pine", "Red"), "Apple")
[1] "PineApple" "RedApple"
> paste(c("Pine", "Red"), "Apple", sep = "-")
[1] "Pine-Apple" "Red-Apple"
> paste(c("Pine", "Red"), "Apple", sep = "-", collapse = ", ")
[1] "Pine-Apple, Red-Apple"
```

The *to String()* function can be used to convert a number vector into a character vector, with the elements separated by a comma and a space. It is possible to specify the width of the print string in this function.

```
> x <- c(1:10) ^ 3
> x
 [1]    1    8   27   64  125  216  343  512  729 1000
> toString(x)
[1] "1, 8, 27, 64, 125, 216, 343, 512, 729, 1000"
> toString(x, 18)
[1] "1, 8, 27, 64, ...."
```

The *cat()* function is also similar to the *paste()* function, but there is little difference in it as shown below.

```
> cat(c("Red", "Pine"), "Apple")
Red Pine Apple
```

The *noquote()* function forces the string outputs not to be displayed with quotes.

```
> a <- c("I", "am", "a", "data", "scientist")
> a
[1] "I"         "am"        "a"         "data"      "scientist"
> noquote(a)
[1] I       am      a       data    scientist
```

The *formatC()* function is used to format the numbers and display them as strings. This function has the arguments *digits*, *width*, *format*, *flag* etc which can be used as below. A slight variation of the function *formatC()* is the function *format()* whose usage is as shown below.

```
> h <- c(4.567, 8.981, 27.772)
> h
[1]  4.567  8.981 27.772
> formatC(h)
[1] "4.567" "8.981" "27.77"
> formatC(h, digits = 3)
[1] "4.57" "8.98" "27.8"
> formatC(h, digits = 3, width = 5)
[1] " 4.57" " 8.98" " 27.8"
> formatC(h, digits = 3, format = "e")
[1] "4.567e+00" "8.981e+00" "2.777e+01"
> formatC(h, digits = 3, flag = "+")
[1] "+4.57" "+8.98" "+27.8"

> format(h)
[1] " 4.567" " 8.981" "27.772"
> format(h, digits = 3)
[1] " 4.57" " 8.98" "27.77"
> format(h, digits = 3, trim = TRUE)
[1] "4.57"  "8.98"  "27.77"
```

The *sprint()* function is also used for formatting strings and passing number values in between the strings. The argument *%s* in this function stands for a string to be passed. The argument *%d* and argument *%f* stands for integer and floating-point number. The usage of this function can be understood by the below example.

```
> x <- c(1, 2, 3)
> sprintf("The number %d in the list is = %f", x, h)
[1] "The number 1 in the list is = 4.567000"
[2] "The number 2 in the list is = 8.981000"
[3] "The number 3 in the list is = 27.772000"
```

To print a tab in between text, we can use the *cat()* function with the special character "\t" included in between the text as below. Similarly, if we need to insert a new line in between the text, we use "\n". In this *cat()* function the argument *fill = TRUE* means that after printing the text, the cursor is placed in the next line. Suppose if a back slash has to be used in between the text, it is preceded by another back slash. If we enclose the text in double quotes and if the text contains a double quote in between, it is also preceded by a back slash. Similarly, if we enclose the text in single quotes and if the text contains a single quote in between, it is also preceded by a back slash. If we enclose the text in double quotes and if the text contains a single quote in between, or if we enclose the text in single quotes and if the text contains a double quote in between, it is not a problem (No need for back slash).

```
> cat("Black\tBerry", fill = TRUE)
Black    Berry
> cat("Black\nBerry", fill = TRUE)
Black
Berry
> cat("Black\\Berry", fill = TRUE)
Black\Berry
> cat("Black\"Berry", fill = TRUE)
Black"Berry
> cat('Black\'Berry', fill = TRUE)
Black'Berry
> cat('Black"Berry', fill = TRUE)
Black"Berry
```

```
> cat("Black'Berry", fill = TRUE)
Black'Berry
```

The function *toupper()* and *tolower()* are used to convert a string into upper case or lower case respectively. The *substring()* or the *substr()* function is used to cut a part of the string from the given text. Its arguments are the text, starting position and ending position. Both these functions produce the same result.

```
> toupper("The cat is on the Wall")
[1] "THE CAT IS ON THE WALL"
> tolower("The cat is on the Wall")
[1] "the cat is on the wall"

> substring("The cat is on the wall", 3, 10)
[1] "e cat is"
> substr("The cat is on the wall", 3, 10)
[1] "e cat is"
> substr("The cat is on the wall", 5, 10)
[1] "cat is"
```

The function *strsplit()* does the splitting of a text into many strings based on the splitting character mentioned as argument. In the below example the splitting is done when a space is encountered. It is important to note that this function returns a list and not a character vector as a result.

```
> strsplit("I like Bannana, Orange and Pineapple", " ")
[[1]]
[1] "I"         "like"      "Bannana,"  "Orange"    "and"       "Pineapple"
```

In this same example if the text has to be split when a comma or space is encountered it is mentioned as ",?". This means that the comma is optional and space is mandatory for splitting the given text.

```
> strsplit("I like Bannana, Orange and Pineapple", ",? ")
[[1]]
[1] "I"        "like"     "Bannana"  "Orange"   "and"      "Pineapple"
```

The default R's working directory can be obtained using the function *getwd()* and this default directory can be changed using the function *setwd()*. The directory path mentioned in the *setwd()* function should have the forward slash instead of backward slash as in the example below.

```
> getwd()
[1] "C:/Users/admin/Documents"
> setwd("C:/Program Files/R")
> getwd()
[1] "C:/Program Files/R"
```

It is also possible to construct the file paths using the *file.path()* function which automatically inserts the forward slash between the directory names. The function *R.home()* list the home directory where R is installed.

```
> file.path("C:", "Program Files", "R", "R-3.3.0")
[1] "C:/Program Files/R/R-3.3.0"
> R.home()
[1] "C:/PROGRA~1/R/R-33~1.0"
```

Paths can also be specified by relative terms such as "." denotes current directory, ".." denotes parent directory and "~" denotes home directory. The function *path.expand()* converts relative paths to absolute paths.

```
> path.expand(".")
[1] "."
> path.expand("..")
[1] ".."
> path.expand("~")
[1] "C:/Users/admin/Documents"
```

The function *basename()* returns only the file name leaving its directory if specified. On the other hand the function *dirname()* returns only the directory name leaving the file name.

```
> filename <- "C:/Program Files/R/R-3.3.0/bin/R.exe"
> basename(filename)
[1] "R.exe"
> dirname(filename)
[1] "C:/Program Files/R/R-3.3.0/bin"
```

2.8. Dates and Times

Dates and Times are common in data analysis and R has a wide range of capabilities for dealing with dates and times.

2.8.1. Date and Time Classes

R has three date and time base classes and they are *POSIXct*, *POSIXlt* and *Date*. *POSIX* is a set of standards that defines how dates and times should be specified and "*ct*" stands for "calendar time". *POSIXlt* stores dates as a list of seconds, minutes, hours, day of month etc. For storing and calculating with dates, we can use *POSIXct* and for extracting parts of dates, we can use *POSXlt*.

The function *Sys.time()* is used to return the current date and time. This returned value is by default in the *POSIXct* form. But, this can be converted to *POSIXlt* form using the function *as.POSIXlt()*. When printed both forms of date and time are displayed in the same manner, but their internal storage mechanism differs. We can also access individual components of a *POSIXlt* date using the dollar symbol or the double brackets as shown below.

```
> Sys.time()
[1] "2017-05-11 14:31:29 IST"
> t <- Sys.time()
> t1 <- Sys.time()
> t2 <- as.POSIXlt(t1)
```

```
> t1
[1] "2017-05-11 14:39:39 IST"
> t2
[1] "2017-05-11 14:39:39 IST"
> class(t1)
[1] "POSIXct" "POSIXt"
> class(t2)
[1] "POSIXlt" "POSIXt"
> t2$sec
[1] 39.20794
> t2[["min"]]
[1] 39
> t2$hour
[1] 14
> t2$mday
[1] 11
> t2$wday
[1] 4
```

The Date class stores the dates as number of days from start of 1970. This class is useful when time is insignificant. The *as.Date()* function can be used to convert a date in other class formats to the Date class format.

```
> t3 <- as.Date(t2)
> t3
[1] "2017-05-11"
```

There are also other add-on packages available in R to handle date and time and they are *date*, *dates*, *chron*, *yearmon*, *yearqtr*, *timeDate*, *ti* and *jul*.

2.8.2. Date Conversions

In CSV files the dates will be normally stored as strings and they have to be converted into date and time using any of the packages. For this we need to parse the strings using the function *strptime()* and this returns the date of the format *POSIXlt*. The date format is specified as a string and passed as argument to the *strptime()* function. If the given string does not match the format given in the format string, then it returns *NA*.

```
> date1 <- strptime("22:15:45 22/08/2015", "%H:%M:%S %d/%m/%Y")
> date1
[1] "2015-08-22 22:15:45 IST"

> date2 <- strptime("22:15:45 22/08/2015", "%H:%M:%S %d-%m-%Y")
> date2
[1] NA
```

In the format string "*%H*" denotes hour in 24 hour system, "*%M*" denotes minutes, "*%S*" denotes second, "*%m*" denotes the number of the month, "*%d*" denotes the day of the month as number, "*%Y*" denotes four digit year.

To convert a date into a string the function *strftime()* is used. This function also takes a date formatting string as argument like *strptime()*. In the format string "*%I*" denotes hour in 12 hours system, "*%p*" denotes AM/PM, "*%A*" denotes the string of day of the week, "*%B*" denotes the string of name of the month.

```
> strftime(Sys.Date(),"It's %I:%M%p on %A %d %B, %Y.")
[1] "It's 12:00AM on Thursday 11 May, 2017."
```

2.8.3. Time Zones

It is possible to specify the time zone when parsing a date string using *strptime()* or *strftime()* functions. If this is not specified, the default time zone is taken. The functions *Sys.timezone()* and *Sys.getlocale("LC_TIME")* are used to get the default time zone of the system and the operating system respectively.

```
> Sys.timezone()
[1] "Asia/Calcutta"
> Sys.getlocale("LC_TIME")
[1] "English_India.1252"
```

Few of the time zones are *UTC* (Universal Time), *IST* (Indian Standard Time), *EST* (Eastern Standard Time), *PST* (Pacific Standard Time), *GMT* (Greenwitch Meridian Time), etc. It is also possible to give manual offset from *UTC* as "*UTC+n*" or "*UTC–n*" to denote west and east parts of *UTC* respectively. Even though it throws warning message, it gives the result correctly.

```
> strftime(Sys.time(), tz = "UTC")
[1] "2017-05-12 04:59:04"

> strftime(Sys.time(), tz = "UTC-5")
[1] "2017-05-12 09:59:09"
Warning message:
In as.POSIXlt.POSIXct(x, tz = tz) : unknown timezone 'UTC-5'
> strftime(Sys.time(), tz = "UTC+5")
[1] "2017-05-11 23:59:15"
Warning message:
In as.POSIXlt.POSIXct(x, tz = tz) : unknown timezone 'UTC+5'
```

The time zone changes does not happen in *strftime()* function if the date is in *POSIXlt* dates. Hence, it is required to change to *POSIXct* format first and then apply the function.

2.8.4. Calculations with Dates and Times

If we add a number to the *POSIXct* or *POSIXlt* classes, it will shift to that many seconds. If we add a number to the *Date* class, it will shift to that many days.

```
> ct <- as.POSIXct(Sys.time())
> lt <- as.POSIXlt(Sys.time())
```

```
> dt <- as.Date(Sys.time())
> ct
[1] "2017-05-12 11:41:54 IST"
> ct + 2500
[1] "2017-05-12 12:23:34 IST"
> lt
[1] "2017-05-12 11:42:15 IST"
> lt + 2500
[1] "2017-05-12 12:23:55 IST"
> dt
[1] "2017-05-12"
> dt + 2
[1] "2017-05-14"
```

Adding two dates, throws error. But subtracting two dates gives the number of days in between the dates. To get the same result, alternatively, the *difftime()* function can be used and in this it is possible to specify the attribute *units* = "*secs*" (or "*mins*" or "*hours*" or "*days*" or "*weeks*").

```
> dt1 <- as.Date("10/10/1973", "%d/%m/%Y")
> dt1
[1] "1973-10-10"
> dt2 <- as.Date("25/09/2000", "%d/%m/%Y")
> dt2
[1] "2000-09-25"

> diff <- dt2 - dt1
> diff
Time difference of 9847 days

> difftime(dt2, dt1)
```

```
Time difference of 9847 days
> difftime(dt2, dt1, units = "secs")
Time difference of 850780800 secs
> difftime(dt2, dt1, units = "mins")
Time difference of 14179680 mins
> difftime(dt2, dt1, units = "hours")
Time difference of 236328 hours
> difftime(dt2, dt1, units = "days")
Time difference of 9847 days
> difftime(dt2, dt1, units = "weeks")
Time difference of 1406.714 weeks
```

The *seq()* function can be used to generate a sequence of dates. The argument "*by*" can take many options based on the class of the dates specified. We can apply the *mean()* and *summary()* functions on these sequence of dates generate.

```
> seq(dt1, dt2, by = "1 year")
 [1] "1973-10-10" "1974-10-10" "1975-10-10" "1976-10-10" "1977-10-10"
                                                          "1978-10-10"
 [7] "1979-10-10" "1980-10-10" "1981-10-10" "1982-10-10" "1983-10-10"
                                                          "1984-10-10"
[13] "1985-10-10" "1986-10-10" "1987-10-10" "1988-10-10" "1989-10-10"
                                                          "1990-10-10"
[19] "1991-10-10" "1992-10-10" "1993-10-10" "1994-10-10" "1995-10-10"
                                                          "1996-10-10"
[25] "1997-10-10" "1998-10-10" "1999-10-10"

> seq(dt1, dt2, by = "500 days")
 [1] "1973-10-10" "1975-02-22" "1976-07-06" "1977-11-18" "1979-04-02"
"1980-08-14"
```

[7] "1981-12-27" "1983-05-11" "1984-09-22" "1986-02-04" "1987-06-19"
"1988-10-31"
[13] "1990-03-15" "1991-07-28" "1992-12-09" "1994-04-23" "1995-09-05"
"1997-01-17"
[19] "1998-06-01" "1999-10-14"

```
> mean(seq(dt1, dt2, by = "1 year"))
[1] "1986-10-10"

> summary(seq(dt1, dt2, by = "1 year"))
        Min.              1st Qu.             Median               Mean            3rd Qu.               Max.
"1973-10-10" "1980-04-10" "1986-10-10" "1986-10-10" "1993-04-10" "1999-10-10"
```

The *lubridate* package makes the process of date and time manipulation easier. The *ymd()* function in this package converts any date to the format of year, month and day separated by hyphens. (*Note:* This function requires the date to be specified in the order of year, month and day, but can use any separator as below).

```
> install.packages("lubridate")
> library(lubridate)
> ymd("2000/09/25", "2000-9-25", "2000*9.25")
[1] "2000-09-25" "2000-09-25" "2000-09-25"
```

If the given date is in other formats that is not in the order of year, month and day, then we have other functions such as *ydm()*, *mdy()*, *myd()*, *dmy()* and *dym()*. These functions can also be accompanied with time by making use of the functions *ymd_h()*, *ymd_hm()* and *ymd_hms()* [similar functions available for *ydm()*, *mdy()*, *myd()*, *dmy()* and *dym()*]. All the parsing functions in the *lubridate* package returns *POSIXct* dates and the default time zone is *UTC*. A function named *stamp()* in the *lubridate* package allows formatting of the dates in a human readable format.

```
> dt_format <- stamp("I purchased on Sunday, the 10th of October 2013 at
                                                    6:00:00 PM")
Multiple formats matched: "I purchased on %A, the %dth of %B %Y at %H:%M:%S
```

%Op"(0), "I purchased on %A, the %dth of October %Y at %Om:%H:%M %Op"...

...

Using: "I purchased groceries on %A, the %dth of %Om %Y at %H:%M:%S %Op"

```
> dt_to_convert <- strptime("2000-09-25 7:00:00", "%Y-%m-%d %H:%M:%S")
> dt_format(dt_to_convert)
[1] "I purchased groceries on Monday, the 25th of 09 2000 at 07:00:00 AM"
```

The *lubridate* package has three variable types, namely the "*Durations*", "*Periods*" and "*Intervals*". The *lubridate* package has the functions, *dyears()*, *dweeks()*, *ddays()*, *dhours()*, *dminutes()*, *dseconds()* etc that specify the duration of year, week, day, hour, minute and second in terms of seconds. The duration of 1 minute is 60 seconds, the duration of 1 hour is 3600 seconds (60 minutes * 60 seconds), the duration of 1 day is 86,400 seconds (24 hours * 60 minutes * 60 seconds), the duration of 1 year is 31,536,000 seconds (365 days * 24 hours * 60 minutes * 60 seconds) and so on. The function *today()* returns the current days date.

```
> y <- dyears(1:5)
> y
[1] "31536000s (~52.14 weeks)" "63072000s (~2 years)"    "94608000s (~3 years)"
[4] "126144000s (~4 years)"    "157680000s (~5 years)"
> w <- dweeks(1:4)
> w
[1] "604800s (~1 weeks)"  "1209600s (~2 weeks)" "1814400s (~3 weeks)"
[4] "2419200s (~4 weeks)"
> d <- ddays(1:10)
> d
 [1] "86400s (~1 days)"     "172800s (~2 days)"    "259200s (~3 days)"
 [4] "345600s (~4 days)"    "432000s (~5 days)"    "518400s (~6 days)"
 [7] "604800s (~1 weeks)"   "691200s (~1.14 weeks)" "777600s (~1.29 weeks)"
[10] "864000s (~1.43 weeks)"
```

```
> today() + y
[1] "2018-05-12" "2019-05-12" "2020-05-11" "2021-05-11" "2022-05-11"
```

"Periods" specify time spans according to the clock time. The *lubridate* package has the functions, *years()*, *weeks()*, *days()*, *hours()*, *minutes()*, *seconds()* etc that specify the period of year, week, day, hour, minute and second in terms of clock time. The exact length of these periods can be realized only if they are added to an instance of date or time.

```
> y <- years(1:7)
> y
[1] "1y 0m 0d 0H 0M 0S" "2y 0m 0d 0H 0M 0S" "3y 0m 0d 0H 0M 0S"
                                                "4y 0m 0d 0H 0M 0S"
[5] "5y 0m 0d 0H 0M 0S" "6y 0m 0d 0H 0M 0S" "7y 0m 0d 0H 0M 0S"
> today()+y
[1] "2018-05-12" "2019-05-12" "2020-05-12" "2021-05-12" "2022-05-12"
                                                "2023-05-12"
[7] "2024-05-12"
```

"*Intervals*" are defined by the instance of date or time at the beginning and end. They are mostly used for specifying "*Periods*" and "*Durations*" and conversion between "*Periods*" and "*Durations*".

```
> yr <- dyears(5)
> yr
[1] "157680000s (~5 years)"
> as.period(yr)
[1] "5y 0m 0d 0H 0M 0S"
> sdt <- ymd("2017-05-12")
> int <- new_interval(sdt, sdt+yr)
> int
[1] 2017-05-12 UTC--2022-05-11 UTC
```

The operator "*%--%*" is used for defining intervals and the operator "*%within%*" is used for checking if a given date is within the given interval.

```
> intv <- ymd("1973-10-10") %--% ymd("2000-09-25")
> intv
[1] 1973-10-10 UTC--2000-09-25 UTC
> ymd("1979-12-12") %within% intv
[1] TRUE
```

The function *with_tz()* can be used to change the time zone of a date (correctly handles *POSIXlt* dates) and the function *force_tz()* is used for updating incorrect time zones.

```
> with_tz(Sys.time(), tz = "America/Los_Angeles")
[1] "2017-05-12 06:44:14 PDT"
> with_tz(Sys.time(), tz = "Asia/Kolkata")
[1] "2017-05-12 19:14:29 IST"
```

The functions *floor_date()* and *ceiling_date()* can be used to find the lower and upper limit of a given date as below.

```
> floor_date(today(), "year")
[1] "2017-01-01"
> ceiling_date(today(), "year")
[1] "2018-01-01"
> floor_date(today(), "month")
[1] "2017-05-01"
> ceiling_date(today(), "month")
[1] "2017-06-01"
```

❖ HIGHLIGHTS

- The basic data types in R are Numeric, Integer, Complex, Logical and Character.