# MATH 180 Introduction to Data Science
# Installing R

R can be downloaded from the Comprehensive R Archive Network (CRAN) at

https://cran.r-project.org/

The user can the select the desired installation from the upper right portion of the screen where links are available for Linux, Mac OS 10, and Windows.

Select the appropriate link and follow the instructions for your machine. You only need to install base R.

**General Instructions for Windows:**

A single .exe file will install the base R package.

1. Click “Download R for Windows”
2. Click the base link
3. Click “Download R X.X.X for Windows” (whatever the latest version is)
4. Run the installer after the download is complete
5. The recommended installation is fine (select default options during installation)

**General Instructions for Mac:**

1. Click “Download R for (Mac) OS X”
2. There will be a section header “Latest release:” and under that the R packages will be listed on the left with their corresponding compatible OS versions on the right. For example, R-4.0.2.pkg is compatible with macOS 10.13 and higher. Check your OS version by clicking the Apple in the top left corner of your computer, then “About this Mac,” and then the Version is listed. Using the OS version of your computer, download the most recent compatible R-X.X.X.pkg
3. Once the package is downloaded to your computer, open it
4. You will be prompted through the installation, press continue through several screens, then press install and put in your password.
5. It will probably ask you to move the installer to the trash (which you can do)
6. If you go to the Applications folder in Finder on your Mac, there should now be the application “R” Just open it and you are ready to use R!

## Note about RStudio:

R-Studio is a cross-platform application, also known as an Integrated Development Environment (IDE) with some great features to support R. It is available at:

https://www.rstudio.com/products/rstudio/download/

We will be using R-Studio during the course and you should download it after installing R. R-studio provides a clean interface with many of the R packages pre-installed in it. R-Studio is extremely popular among data scientists, as it provides many useful short-cuts and improvements over the base R interface.

## <u>Installing Packages</u>:

We will make use of several packages for R that come with functions, data sets, and other objects. To install a package, use the `install.packages` command

To access the items in a package in your library, use the library command:

```
> library(MASS)
```

The functions and data sets in the `MASS` package are now available for use during the active session. You will need to use the library command every time you start R as they are only available during the active session.

You can obtain session info including which base and extended packages are currently attached using the `sessionInfo` command.

```
> sessionInfo()

R version 3.6.2 (2019-12-12)
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows 10 x64 (build 17763)

Matrix products: default

Random number generation:
 RNG:     Mersenne-Twister
 Normal:  Inversion
 Sample:  Rounding

locale:
[1] LC_COLLATE=English_United States.1252  LC_CTYPE=English_United
States.1252
[3] LC_MONETARY=English_United States.1252 LC_NUMERIC=C
[5] LC_TIME=English_United States.1252

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] MASS_7.3-51.4 class_7.3-15
```

## Introduction to R

When you open R, it loads a *workspace* into memory from the working directory location (more on this later). This workspace is then updated during the session and can be saved (if desired) when the session is over. The saved workspace will then be loaded when R is opened at a future time.

After opening R, the version is displayed followed by the command prompt **>**.

```
R version 3.6.2 (2019-12-12) -- "Dark and Stormy Night"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)
```

You can create an object using the assignment operator **<-**

The function c (short for concatenate or combine values into a vector or list) can be used to create a vector of values. For example,

```
> X<-c(1,2,3,4)
> X
[1] 1 2 3 4
```

This creates a vector object, X, whose elements are the numbers 1, 2, 3, and 4. This object will remain in the session until it is altered or removed. Typing its name displays the object.

To obtain a list of all the objects in your current workspace, use the ls function (that first letter is a lower-case L)

```
> ls()
[1] "X"
```

If at any time you want to remove an object, use the rm command.

```
> rm(X)
> X
Error: object 'X' not found
> ls()
character (0)
```

We observe the object X has been removed.

## **Help**:

Help files are available by invoking the help system using the ? or help commands. This will provide information on many of the objects encountered in R.

```
> ?var
> help(var)
```

Either of these commands will access the help files on the var function.

## **Basic functionality**:

R can be used as a calculator and has all the basic math functions built in. The standard mathematical operators + – * / ^ should be familiar as well as their order of operation. For example:

```
> 7+3*2
[1] 13
> 7/2+5
[1] 8.5
```

Common functions include

| Command | Output | Command | Output |
|---|---|---|---|
| sin(x) | Sine function | abs(x) | Absolute value |
| cos(x) | Cosine function | log(x) | Natural Log function |
| tan(x) | Tangent function | log10(x) | Log base 10 |
| floor(x) | Floor function | exp(x) | Natural Exponential function |
| ceiling(x) | Ceiling function | sqrt(x) | Square root |

```
> sin(pi/4)
[1] 0.7071068
> exp(2)
[1] 7.389056
> abs(-3)
[1] 3
> floor(3.765)
[1] 3
> ceiling(2.34)
[1] 3
```

## **Continuation**:

When you hit Enter for an expression that is not complete, R will provide a *continuation* prompt for you to complete the expression. This appears as a **+** instead of a **>**. You can then complete the expression and hit Enter. For example:

```
> 7*
+ 8
[1] 56
> sqrt(99
+ )
[1] 9.949874
```

## **Directory Information**:

You can find the directory where R is installed using `R.Home`

```
> R.home()
[1] "C:/PROGRA~1/R/R-36~1.2"
```

and you can find the working directory using the `getwd` command.

```
> getwd()
[1] "C:/Users/bilene/Documents"
```

The working directory is the location where R will:

1. save the current workspace and
2. the location from which it will read files (where the default path is located).

Information about the operating system can be obtained using the `Sys.info` command.

```
> Sys.info()
       sysname        release         version       nodename        machine
     "Windows"       "10 x64"   "build 17763"       "BILENE"       "x86-64"
         login           user  effective_user
    "bilene"      " bilene "      " bilene "
```

To quit R, use the q command. You will be prompted as to whether you want to save the workspace. If you click yes, changes will be saved and available when R is reopened, otherwise, they will be lost.

```
> q()
```

## Introduction to Data Values and Data Objects

### Data Values in R:

There are four basic kinds (modes) of values in R:

1. numeric (6, 4.2, 2.3e10, 3.45e-4)
2. character ("ABC", "hello", "7")
3. logical (TRUE, FALSE or T, F)
4. complex (2 + 4i)

You can use the typeof command to discover the storage mode of an object in R:

```
> Y<-3.6
> Z<-"34"
> typeof(Y)
[1] "double"
> typeof(Z)
[1] "character"
> Z<-2+3i
> typeof(Z)
[1] "complex"
```

**Special Values:**

It is often the case that data sets contain many missing values. R codes these as NA, for Not Available. The NA value can appear in a data object of any type. Many functions have methods for handling missing values (more on this later). Another result that may appear is NaN, which means Not a Number and occurs when R encounters indeterminate calculations such as 0/0. It differs from NA in that it is not missing but rather a type of calculation error. Finally, you may encounter the value NULL, which indicates a *non-value.* This differs from a missing value in that no value is present and none was intended.

### Data Objects:

In R, a *data object* is a collection of values. Many of the data objects in R can only hold one kind of value at a time. These include vectors and matrices. This means a vector cannot have values of both numeric and character mode stored in it at the same time (but one or the other is fine). Data objects come with a set of *attributes* the specify the characteristics of the object. Different objects have different attributes defined for them.

**Vectors:**

You can create a vector (an ordered set of values) in many ways in R including using the c (concatenate) command.

```
> X<-c(1,2,3,4)
> Y<-1:4
> X
[1] 1 2 3 4
> Y
[1] 1 2 3 4
> Z<-c(2,45,NA,1)
> Z
[1] 2 45 NA  1
```

Observe that the third value in the vector `Z` is missing, as indicated by the `NA` value.

Elements of a vector can be accessed by their index/indices or sets of indices:

```
> X<-c(1.4,5.2,34,8)
> X
[1]  1.4  5.2 34.0  8.0
> X[3]
[1] 34
> X[2:3]
[1]  5.2 34.0
```

The length (number of elements) of a vector can be queried using the `length` command.

```
> length(X)
[1] 4
```

Other important ways to create a vector include the `seq` and `rep` functions. In its basic usage, the `rep` function *repeats* a value a specified number of `times`, the value itself may be a vector. The are other ways to use the `rep` function, type `?rep` for details. Examples:

```
> X<-rep(3,5)
> X
[1] 3 3 3 3 3
> X<-rep(c(1,2,3),3)
> X
[1] 1 2 3 1 2 3 1 2 3
```

The `seq` (sequence) function creates a sequence of values spaced by an increment. The function has several modes of operation, and details can be found using `?seq`. The arguments are given by

```
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)), length.out = NULL,
along.with = NULL, ...)
```

| | |
|---|---|
| `from, to` | the beginning and end values of the sequence |
| `by` | the increment of the sequence |
| `length.out` | the length (number of elements) of the sequence |

Examples:

```
> X<-seq(1,11,2)
> X
[1]  1  3  5  7  9 11
> X<-seq(1,10,length.out=11)
> X
 [1]  1.0  1.9  2.8  3.7  4.6  5.5  6.4  7.3  8.2  9.1 10.0
```

The values in a vector can be named. The names attribute is an optional attribute for a vector and can be accessed and manipulated using the `names` function.

```
> X<-c(1,2,3,4)
> X
[1] 1 2 3 4
> names(X)
NULL
```

Note that the Null value indicates that the vector `X` has no names (not that they are missing). We can also use the `names` function to assign names to the values in `X`.

```
> names(X)<-c("first","second","third","fourth")
> names(X)
[1] "first"  "second" "third"  "fourth"
> X
 first second  third fourth
     1      2      3      4
> X["third"]
third
    3
> names(X)[3]
[1] "third"
> names(X)[3]<-"THIRD"
> X
 first second  THIRD fourth
     1      2      3      4
```

## Matrices:

A matrix is a rectangular array of values. All elements in a matrix must be of the same data type.

The `matrix` command can be used to create a matrix of elements. The command accepts the following arguments.

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

You can set the desired number of rows (or columns) using the `nrow` or `ncol` arguments; the `byrow` argument indicates how the matrix is filled, i.e. across the rows or down the columns.

```
> X<-matrix(1:10,nrow=2)
> X
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> X<-matrix(1:10,nrow=2,byrow=T)
> X
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
```

Elements in a matrix have two indices, a row number and a column number. Individual elements and subsets of elements can be extracted by supplying indices or sets of indices. The square brackets [ , ] are structured with row(s) first, then column(s).

```
> X[1,4]
[1] 4
> X[2,3]
[1] 8
```

We can select the second row using

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

and we could select columns 3 and 4 with

```
> X[,3:4]
     [,1] [,2]
[1,]    3    4
[2,]    8    9
```

We can also remove rows or columns by using a – symbol in front of the row(s) or column(s) to remove. For example, we can print X without column 2 by typing

```
> X[,-2]
     [,1] [,2] [,3] [,4]
[1,]    1    3    4    5
[2,]    6    8    9   10

> X[,-c(2,4)]
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    6    8   10
```

removes columns 2 and 4. A similar result applies to the rows of X.

In addition to its length, which is the total number of elements, a matrix also has a `dim` attribute, which specifies its number of rows and columns.

```
> X
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
> length(X)
[1] 10
> dim(X)
[1] 2 5
```

Similar to the names for a vector, a matrix can have row or column names (or both). This information is stored in the dimnames attribute of the matrix. You can access and modify these using the `dimnames` function:

```
> dimnames(X)
NULL
```

You can find or set the row or column names of a matrix using the functions `rownames` and `colnames`.

```
> rownames(X)
NULL
> colnames(X)
NULL
> rownames(X)<-c("Row 1","Row 2")
> colnames(X)<-c("Col a","Col b","Col c","Col d","Col e")
> rownames(X)
[1] "Row 1" "Row 2"
> colnames(X)
[1] "Col a" "Col b" "Col c" "Col d" "Col e"
> X
      Col a Col b Col c Col d Col e
Row 1     1     2     3     4     5
Row 2     6     7     8     9    10
> dimnames(X)
[[1]]
```

```
[1] "Row 1" "Row 2"

[[2]]
[1] "Col a" "Col b" "Col c" "Col d" "Col e"
```

Note that the dimnames are an unfamiliar data object, called a *list*. The first element of this list is the vector of row names, and the second element is the vector of column names. We will discuss lists in more detail shortly.

**Aside:**

Many R functions are *overloaded*, that is, they can be applied to a variety of objects and produce a result appropriate for the input. For example:

```
> X
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
> sqrt(X)
        [,1]     [,2]     [,3] [,4]     [,5]
[1,] 1.00000 1.414214 1.732051    2 2.236068
[2,] 2.44949 2.645751 2.828427    3 3.162278
> Y
[1] 1 2 3 4
> sqrt(Y)
[1] 1.000000 1.414214 1.732051 2.000000
```

In the example above, the `sqrt` function was able to accept either a matrix or a vector as input and its output returned an object of the same type whose elements were the square root of the elements of the original object.

## Lists:

A **list** is a data structure in R that allows you to store a set of *components* that are ordered like a vector, but the components can be of any kind of data structure. For example, a list could contain two vectors and a matrix.

You can create a list using the `list` function. For example,

```
> X<-list(rep(2,3),matrix(1:4,nrow=2),c("hello","bye"))
> X
[[1]]
[1] 2 2 2

[[2]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[3]]
[1] "hello" "bye"
```

which creates a list whose first component is a vector with numeric elements, its second component is a 2x2 matrix and its third component is a vector of character data. Note that each component of the list can have all the attributes associated with its object type (names, dim, etc.).

You can extract a component of a list using double brackets [[ ]]. For example,

```
> X[[1]]
[1] 2 2 2
> X[[2]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

You can also access values in a component using the [ ] single bracket notation. For example,

```
> X[[2]][,2]
[1] 3 4
> X[[3]][1]
[1] "hello"
```

List components can also be names (similar to vectors). These names can be accessed and manipulated using the `names` command.

```
> names(X)
NULL
> names(X)<-c("Vec1","MAT1","Vec2")
> X
$Vec1
[1] 2 2 2

$MAT1
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$Vec2
[1] "hello" "bye"
```

We can now access list components by their name (using the $).

```
> X$MAT1
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

The length of a list is the number of components in the list.

```
> X$MAT1
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> length(X)
[1] 3
> length(X$MAT1)
[1] 4
```

Note that the dimnames of a matrix is a list of values, with the first component being a vector of names for the row variables and the second component being a vector of column names.

The output of many of the functions in R is in the form of a list (sometimes a long one if a lot of different information is provided).

## Factors:

Categorical (Qualitative) data is represented in R as a special data type called a *factor*. The possible values of a factor are called the *levels* of the factor. Some examples include:

| | |
|---|---|
| Treatment: | Drug or Placebo |
| Antibiotic Given: | Antibiotic A, Antibiotic B, Antibiotic C, Antibiotic D, |
| Political Party: | Democratic, Republican, Independent, Other |

You can use the `factor` function to create a factor.

```
> Treatment<-c("Drug","Drug","Drug","Placebo","Placebo","Placebo")
> Treatment
[1] "Drug"    "Drug"    "Drug"    "Placebo" "Placebo" "Placebo"
> Treatment<-factor(Treatment)
> Treatment
[1] Drug    Drug    Drug    Placebo Placebo Placebo
Levels: Drug Placebo
```

Note that the character vector was converted to a vector containing the information coded as a factor with levels Drug and Placebo. In fact, factors are stored internally in R as integers. You can view this using the `print.default` command

```
> Treatment
[1] Drug    Drug    Drug    Placebo Placebo Placebo
Levels: Drug Placebo
> print.default(Treatment)
[1] 1 1 1 2 2 2
```

We see that R maps 1 to the level Drug and 2 to the level Placebo. What if we had wanted 1 to be Placebo and 2 Drug? If you enter a vector of `levels` with the factor function, the resulting factor will order the factors as they appear in this vector (the default is to order them alphabetically). For example,

```
> Treatment<-factor(Treatment,levels=c("Placebo","Drug"))
> Treatment
[1] Drug    Drug    Drug    Placebo Placebo Placebo
Levels: Placebo Drug
> print.default(Treatment)
[1] 2 2 2 1 1 1
```

If your data is *ordinal*, i.e. an ordered factor, this can be accommodated using the `ordered` command which creates an ordered factor. For example, suppose we had a variable Social Class, with ordered levels Lower, Middle, and Upper. We would use:

```
> SoClass<-c("Lower", "Middle","Middle","Upper","Lower")
> SoClass<-ordered(SoClass,levels=c("Lower","Middle","Upper"))
> SoClass
[1] Lower  Middle Middle Upper  Lower
Levels: Lower < Middle < Upper
```

The levels are now indicated as being ordered with Lower < Middle < Upper. Note that the order is the one provided in the `levels` argument. Important: If you don't supply a levels argument, the ordering will be done alphabetically by R.

## Data Frames:

Multivariable data is often stored using a data frame object. These data structures allow the storage of different types of variables in the same object, you can think of a data frame as a generalization of a matrix. Data frames allow you to mix data modes from column to column (they must be the same *within* a column). The `data.frame` command can be used to construct a data.frame.

```
> df_1<-data.frame(Number=c(1,4,2,5),Letter=c("a","r","g","f"),Truth=c(T,F,F,T))
> df_1
  Number Letter Truth
1      1      a  TRUE
2      4      r FALSE
3      2      g FALSE
4      5      f  TRUE
```

Each of the columns in `df_1` is of a different type: numeric, character, logical. Note that the columns of the data frame can have names, which we can access with the `colnames` function.

```
> colnames(df_1)
[1] "Number" "Letter" "Truth"
```

We can access a column of data in a data frame using its column number or by its name (with the $ symbol). For example:

```
> df_1$Truth
[1]  TRUE FALSE FALSE  TRUE
> df_1[,3]
[1]  TRUE FALSE FALSE  TRUE
> df_1[,1]
[1] 1 4 2 5
> df_1$Number
[1] 1 4 2 5
```

This highlights the fact that there are often **many ways** to accomplish the same thing in R (which is a good and bad thing). All the following will pull the number 2 from the `Number` column

```
> df_1[3,1]
[1] 2
> df_1$Number[3]
[1] 2
> df_1[3,"Number"]
[1] 2
> df_1[,1][3]
[1] 2
> df_1[,"Number"][3]
[1] 2
```

Character vectors are converted to factors by the `data.frame` function (because R assumes this is best for analysis). To prevent this, pass the vector to the `data.frame` function in a call to the `I` function; this is the Inhibit Interpretation/Conversion of Objects function and indicates an object should be treated 'as is'. For example:

```
> df_2<-data.frame(Number=c(2,6,1,4),Letters=I(c("abc","fre","hjt","gth")))
> df_2
  Number Letters
1      2     abc
2      6     fre
3      1     hjt
4      4     gth
> df_2$Letters
[1] "abc" "fre" "hjt" "gth"
> typeof(df_2$Letters)
[1] "character"
```

The functions `nrow` and `ncol` can be used to find the number of rows or columns for either matrix or data frame objects. The `length` function can also be used after selecting a row or column to obtain the same information.

```
> X
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
> ncol(X)
[1] 5
> nrow(X)
[1] 2
> length(X[1,])
[1] 5
> length(X[,1])
[1] 2
```

You can also access and modify the row or column names of a data frame using the `rownames` and `colnames` functions. You can view them as a list using the `dimnames` function. See the matrices section above for more details.

We end this lesson with another nice function, `head`. This function displays the first few lines or rows of an object (the default is 6) and is useful when the object is very large, and you don't want to print it in its the entirety. It works on functions as well.

```
> head(matrix)

1 function (data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
2 {
3     if (is.object(data) || !is.atomic(data))
4         data <- as.vector(data)
5     .Internal(matrix(data, nrow, ncol, byrow, dimnames, missing(nrow),
6         missing(ncol)))
```

displays the first 6 lines of the function `matrix`. You can have the function display more or fewer lines with the optional parameter `n`.

As another example,

```
> Y<-matrix(1:100,nrow=10)
> head(Y,4)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1   11   21   31   41   51   61   71   81    91
[2,]    2   12   22   32   42   52   62   72   82    92
```

```
[3,]    3   13   23   33   43   53   63   73   83    93
[4,]    4   14   24   34   44   54   64   74   84    94
```

Don't forget to use the `rm` command to remove any unwanted objects!

## Functions and Packages

R is a powerful language that allows the user to create sophisticated functions that can be used to generate complex results. Base R comes preloaded with more than 1000 functions that allow the user to perform many numerical and graphical tasks. In addition, the R community has created a vast number of *packages*, collections of functions and data objects, that can be downloaded and used to augment base R.

The general form of a function is `fncName`(Arg1, Arg2, …), where `fncName` is the *name* of the function and the `Arg` are the *arguments* of the function which are enclosed in parentheses and separated be commas.

A function can have:

- No arguments
- A set of required arguments
- Optional arguments (arguments that can be changed from default values)

As an example, we first consider the function `c`, which is a function designed to Combine Values into a Vector or List. In its simplest form, we can use the function to make a vector (ordered list) of values, as

```
> X<-c(1,2,3,4)
> X
[1] 1 2 3 4
```

As seen above, the function has taken the values 1, 2 ,3, and 4 and combined them into a vector which we assigned to the object `X`. We then were able to view this vector by typing its name, `X`.

We can access the help available for the function `c` by typing `?c` or `help(c)`. This will open the available help file. In the file we see

```
c(..., recursive = FALSE, use.names = TRUE)
```

Arguments

| | |
|---|---|
| ... | objects to be concatenated. |
| recursive | logical. If recursive = TRUE, the function recursively descends through lists (and pairlists) combining all their elements into a vector. |
| use.names | logical indicating if names should be preserved. |

The user must submit the objects to be concatenated and we also see that the function has two arguments, `recursive`, and `use.names`, that have *default values* (both set to logical values, the first `FALSE` and the second `TRUE`). When we call the function, we are free to change these by setting them to other (logical) values. If we set `use.names = FALSE`, then when the function concatenates the objects, it will discard their names. In our example above, the elements we were concatenating had no names and setting the value to `TRUE` or `FALSE` would have no effect on the outcome.

As another example, the `ls` List Objects function has no required arguments and thus can be invoked as

```
> ls()
[1] "X"
```

This function also has optional arguments which can be seen in the help file on the function.

Let's look at one more built in function, `mean`. This function finds the arithmetic average of a set of numbers. Looking at the help file, we find the function has the form

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

We see that it requires one object, `x`, since this has no default value. Suppose we submit our vector `X` to the `mean` function.

```
> mean(X)
[1] 2.5
```

We get the value we expect, since (1 + 2 + 3 + 4)/4 = 2.5.

What if we set the third value in `X` to `NA` to indicate it is *missing*.

```
X[3]<-NA
> X
[1]  1  2 NA  4
> mean(X)
[1] NA
```

The function now returns the value `NA`, since it cannot find an average with a missing value. We observe the optional argument `na.rm` in the `mean` function. If we set this to `TRUE`, then the function will remove the missing values before calculating the mean.

```
> mean(X,na.rm=T)
[1] 2.333333
```

Now the mean of `X` is given as (1 + 2 + 4)/3 = 7/3. The NA was removed prior to the calculation of the mean.

## Object Naming:

You should avoid defining objects to names that already exist in the namespace. For example, the function `mean` is in the base package (which is automatically attached). What happens if we try to create a vector with this name?

```
> mean<-c(1,2,3)
> mean
[1] 1 2 3
```

Now if we type mean, R prints the vector `mean`, not the function `mean`. The function `mean` is not gone (like it normally would be if we overwrite an object in R), it is just *masked* (more on this later). To see that they are both there we can use the `find` command.

```
> find("mean")
[1] ".GlobalEnv"   "package:base"
```

Now we see that the name mean is associated with two different objects, which can lead to confusion and is not good practice. When you define objects, they are assigned to the global environment. We can remove the object `mean` in the global environment using the `rm` command.

```
> rm(mean)
> find("mean")
[1] "package:base"
> mean
function (x, ...)
UseMethod("mean")
<bytecode: 0x000000000513c9c0>
<environment: namespace:base>
```

We see now that the function `mean` has been restored and is the only object with this name.

## Packages:

One of the strengths of R is the ability of the user to access collections (libraries) of functions and data sets referred to as *packages*. Packages can include functions, documentation, and data sets that the user can access after *attaching* them to the workspace in the R environment.

Most packages need to be downloaded into a local library and then attached to the workspace when the user wants to access them. There are a few packages that come with the base installation of R, some of which are loaded automatically when you open the R environment. To see what packages are currently attached, you can use the `sessionInfo` function.

## Installing Packages

You can install packages from the CRAN (Comprehensive R Archive Network) using the `install.packages` function. For example, we will be using the `dplyr` package extensively during the course. To install this package, type

```
> install.packages("dplyr")
```

You will be prompted to select a secure CRAN mirror. Choose any site and follow the instructions to download the package. Note that for packages not on the CRAN, you can use the `install.gethub` function available in the `devtools` package (we won't discuss this further here).

## Custom Functions

Often times it is handy to write your custom function in R. Functions can take inputs (objects) as arguments, and do some operation for you. You can program it to return a value or an object, or simply have it complete an operation. It has the following basic structure,

```
say_hi <- function() {
      print("Hello, World!")
}

> say_hi()
## [1] "Hello, World!"
```

You can define a function on your script editor (makes it much easier given it will have multiple rows), or via

```
> fix(say_hi)
```

Another example,

```
twice <- function(my_text) {
      print(my_text)
      print(my_text)
}

> twice("Hello, World!")
## [1] "Hello, World!"
## [1] "Hello, World!"

> twice("Another custom text!")
## [1] "Another custom text!"
## [1] "Another custom text!"
```

Lastly,

```
evenorodd <- function(num) {
      if(num %% 2 == 0){
            return("EVEN")
      }
      return("ODD")
}

> evenorodd(5)
## [1] "ODD"

> evenorodd(4)
## [1] "EVEN"
```

## If-else statement:

As with virtually all programming languages, R comes with the ability to write if-else statements. To demonstrate what if-else statements do, define an object,

```
> x <- 2
```

The following is only executed if the x<0 condition is true,

```
if(x < 0){
   print("negative")
}
```

You can combine if with else to tell R what to do if the condition is not satisfied,

```
if(x < 0){
      print("negative")
} else {
      print("positive")
}
## [1] "positive"
```

Or specify multiple ifs with,

```
if(num > 0) {
      print("positive")
} else if (num < 0) {
      print("negative")
} else {
print("zero")
}
## [1] "positive"
```

**For loop:**
A for loop allows you to iterate over objects, or repeat commands. The basic structure looks like the following,

```
for(i in 1:3){
print(i)
}

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

You can loop over a custom vector,

```
myvector <- c('a', 'vector', 'of', 'character', 'values', 'works', 'too!')
for(word in myvector){
print(word)
}
## [1] "a"
## [1] "vector"
## [1] "of"
## [1] "character"
## [1] "values"
## [1] "works"
## [1] "too!"
```

## Matrices, Data Frames, and `dplyr`

Rectangular arrays of data are stored in R as a matrix or as a data frame. Both these data objects have a collection of *attributes* (as do all data objects in the R environment). Understanding how some of these attributes interact with commands in the `dplyr` suite of functions is important.

### Matrices:

A matrix is a rectangular array of values. All elements in a matrix must be of the same data type.

The `matrix` command can be used to create a matrix of elements. The command accepts the following arguments.

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

You can set the desired number of rows (or columns) using the `nrow` or `ncol` arguments; the `byrow` argument indicates how the matrix is filled, i.e. across the rows or down the columns.

Matrix objects have the following attributes.

| | |
|---|---|
| `length` | number of values |
| `mode` | kind of values |
| `dim` | number of rows and columns |
| `dimnames` | row and column names |

You can print the values of the attributes of a matrix object as follows:

```
> X<-matrix(1:10,nrow=2,byrow=T)
> length(X)
[1] 10
> dim(X)
[1] 2 5
> mode(X)
[1] "numeric"
> dimnames(X)
NULL
```

This tells us that the matrix `X` has a total of ten values, which are numeric, stored in a 2 x 5 array and it currently has no row or column names. Here is the object printed:

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

The `dimnames` attribute is a list object with two elements in it: a vector of row names and a vector of column names. Either or both of these can be set to `NULL` which means there are no names in that dimension.

You can retrieve or set the row or column names (or both) using the `rownames`, `colnames`, or `dimnames` functions, respectively. To set the names, pass in a vector of values into the `rownames` or `colnames` functions or a list of two vectors of names into the `dimnames` function as shown in the following examples

```
> rownames(X)<-c("Row 1", "Row 2")
> X
      [,1] [,2] [,3] [,4] [,5]
Row 1    1    2    3    4    5
Row 2    6    7    8    9   10
> colnames(X)<-c("Col 1", "Col 2","Col 3","Col 4","Col 5")
> X
      Col 1 Col 2 Col 3 Col 4 Col 5
Row 1     1     2     3     4     5
Row 2     6     7     8     9    10

> dimnames(X)<-list(c("R1","R2"),c("A","B","C","D","E"))
> X
   A B C D  E
R1 1 2 3 4  5
R2 6 7 8 9 10
> dimnames(X)
[[1]]
[1] "R1" "R2"

[[2]]
```

```
[1] "A" "B" "C" "D" "E"
```

The `letters` and `LETTERS` (built-in) data objects make this process faster. The first object is a vector of the lowercase letters, while the second is a vector of the uppercase letters. Remember, to access an element of a list, we use double brackets [[ ]] instead of the single brackets used for vectors and arrays.

```
> dimnames(X)[[1]]
[1] "R1" "R2"
> dimnames(X)[[2]]
[1] "A" "B" "C" "D" "E"
> colnames(X)<-letters[1:5]
> X
   a b c d  e
R1 1 2 3 4  5
R2 6 7 8 9 10
> dimnames(X)[[1]]<-LETTERS[1:2]
> X
  a b c d  e
A 1 2 3 4  5
B 6 7 8 9 10
```

Remember, we can use row or column names to extract data from a matrix as in the following examples. Note that the `$` operator does not work with matrices.

```
> X["A",]
a b c d e
1 2 3 4 5
> X[,c("b","d")]
  b d
A 2 4
B 7 9
> X["A",c("b","d","e")]
b d e
2 4 5
```

## Data Frames:

Multivariable data is often stored using a data frame object. These data structures allow the storage of different types of variables in the same object, you can think of a data frame as a generalization of a matrix. Data frames allow you to mix data modes from column to column (they must be the same *within* a column). The `data.frame` command can be used to construct a data.frame.

While data frame objects are similar to matrices (at least in your mind), they have many important differences. They are represented internally as lists, and hence the have `mode "list"` (see below).

Data frame objects have the following attributes.

| | |
|---|---|
| `length` | number of variables |
| `mode` | “list” |
| `names` | variable labels |
| `rownames` | row or observation labels |
| `class` | “data.frame” |

To explore the attributes of a data frame, we will examine the `mtcars` data frame, which is available with base R (and should be accessible for you without attaching any packages). This data was extracted from the 1974 *Motor Trend* US magazine and comprises fuel consumption and ten aspects of automobile design and performance for 32 automobiles (1973–74 models). Type `?mtcars` for more information.

We can view the top of the data frame using the `head` command.

```
> head(mtcars)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

You can print the values of the attributes of a data frame object as follows:

```
> length(mtcars)
[1] 11
> mode(mtcars)
[1] "list"
> names(mtcars)
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear" "carb"
> rownames(mtcars)
 [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"          "Hornet 4 Drive"
 [5] "Hornet Sportabout"   "Valiant"             "Duster 360"          "Merc 240D"
 [9] "Merc 230"            "Merc 280"            "Merc 280C"           "Merc 450SE"
[13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"  "Lincoln Continental"
[17] "Chrysler Imperial"   "Fiat 128"            "Honda Civic"         "Toyota Corolla"
[21] "Toyota Corona"       "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
[25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"       "Lotus Europa"
[29] "Ford Pantera L"      "Ferrari Dino"        "Maserati Bora"       "Volvo 142E"
> class(mtcars)
[1] "data.frame"
```

Note the differences between the attributes of a data frame and those of a matrix. For example, the `length` attribute of a data frame refers to the number of variables (columns) in the data frame, not the total number of values stored in the object as is the case for a matrix.

You can also use the `dimnames` command with a data frame, which lists the `rownames` and `names` vectors as its first and second components, respectively.

```
> dimnames(mtcars)
[[1]]
 [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
 [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
 [7] "Duster 360"          "Merc 240D"           "Merc 230"
[10] "Merc 280"            "Merc 280C"           "Merc 450SE"
[13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
[19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
[22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
[25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
[28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
[31] "Maserati Bora"       "Volvo 142E"
[[2]]
```

```
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear" "carb"
```

In addition, you can use the `$` operator with a data frame to extract a column of data:

```
> mtcars$hp
 [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230  66  52
[20]  65  97 150 150 245 175  66  91 113 264 175 335 109
```

Why do we want row and column names? Some functions will use row or column names during the analysis of the data or in the display of the results. This makes less work for the user, who would otherwise have to enter the information in as an optional argument into the function. However, not everyone is a fan of row names. The `dplyr` package is part of the Tidyverse, a suite of integrated packages for data wrangling. Here is their statement on row names:

## Getting Data into R

### Entering Data by Hand:

For small data sets, it may be convenient to enter data by hand using the keyboard. This can be accomplished using the `scan` command (with no arguments). To use this method, type `ObjName<-scan()`. The prompt will change to `1:`. You can enter as many values on a line as you like, with spaces between the values. Hitting Enter will bring up a new line. When you have finished, hit Enter on an empty line to complete the scan. The result is a vector with the entered values. For example,

```
> Data1<-scan()
1: 1 4 6 7
5: 21 34 1 100
9: 3
10:
Read 9 items
> Data1
[1]   1   4   6   7  21  34   1 100   3
```

By default, `scan` expects numeric data. If you want to enter character data, you can set the optional argument `what` to `what=""`. This tells R to expect character data. For example,

```
> Data2<-scan(what="")
1: s d r rtr
5: ff e
7:
Read 6 items
> Data2
[1] "s"   "d"   "r"   "rtr" "ff"  "e"
```

Note that the signal for a new value is whitespace (which may need to be considered depending on what character data is being read).

### Reading in a Data Table

```
Mammal      Water  Protein   Fat   Lactose   Ash
Horse        90.1    2.6      1      6.9     0.35
Orangutan    88.5    1.4     3.5      6      0.24
Monkey       88.4    2.2     2.7     6.4     0.18
```

```
  Donkey      90.3     1.7      1.4      6.2      0.4
  Hippo       90.4     0.6      4.5      4.4      0.1
```

**Reading in a .csv File:**

A CSV (comma separated value) file is a common data storage format. Values of the variables in the file are separated with a comma. In general, CSVs are easy to share across platforms and the structure is non-proprietary. However, they are not compressed and can take up larger amounts of disk space than other formats.

The function `read.csv` is part of base R and is designed to read in .csv files.

## Storing Data:

Saving the entire workspace when it has multiple objects in it can lead to difficulties, especially when the objects in question are large. If you have been working on an object and would like to save a copy of the object, you can use the `save` command. The object can then be retrieved in a later session using the `load` command.

The `save` and `load` commands default to saving and loading files from the working directory. You can view the working directory using `getwd`.

```
> getwd()
[1] "C:/Users/bilene/Documents"
```

This indicated that R will save to and read from this location unless a path is specified. The suffix `.RData` is usually given to the saved file. Here is an example of how to save, remove, and then load a (small) object in R.

```
> Data1
[1]   1   4   6   7  21  34   1 100   3
> save(Data1,file="Data1.RData")
> rm(Data1)
> Data1
Error: object 'Data1' not found
> load(file="Data1.RData")
> Data1
[1]   1   4   6   7  21  34   1 100   3
```

If the object is (very) large, you may want to use a compression method when you save it. This can be done using the `compress` optional argument. Several schemes are available, see `?save` for details.

You can also specify a full path to both the `save` and `load` functions. For example, to save to the folder AML_Sets in the Documents folder, we could use:

```
> save(Data1,file="C:/Users/bilene/Documents/Data1.RData")
```

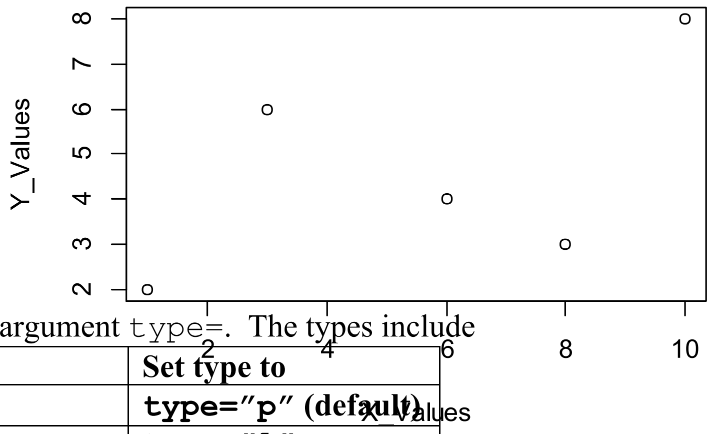
## Introduction to Graphics

### Basic Plots:

```
> demo(graphics)
```

The basic scatterplot can be generated using the `plot` function, whose form is `plot(x, y, ...)` where `x` is a vector of the *x*-coordinates and `y` a vector of *y*-coordinates. Submitting vectors of different lengths will produce an error. As an example:

```
> X_Values<-c(1,3,6,8,10)
> Y_Values<-c(2,6,4,3,8)
> plot(X_Values,Y_Values)
```

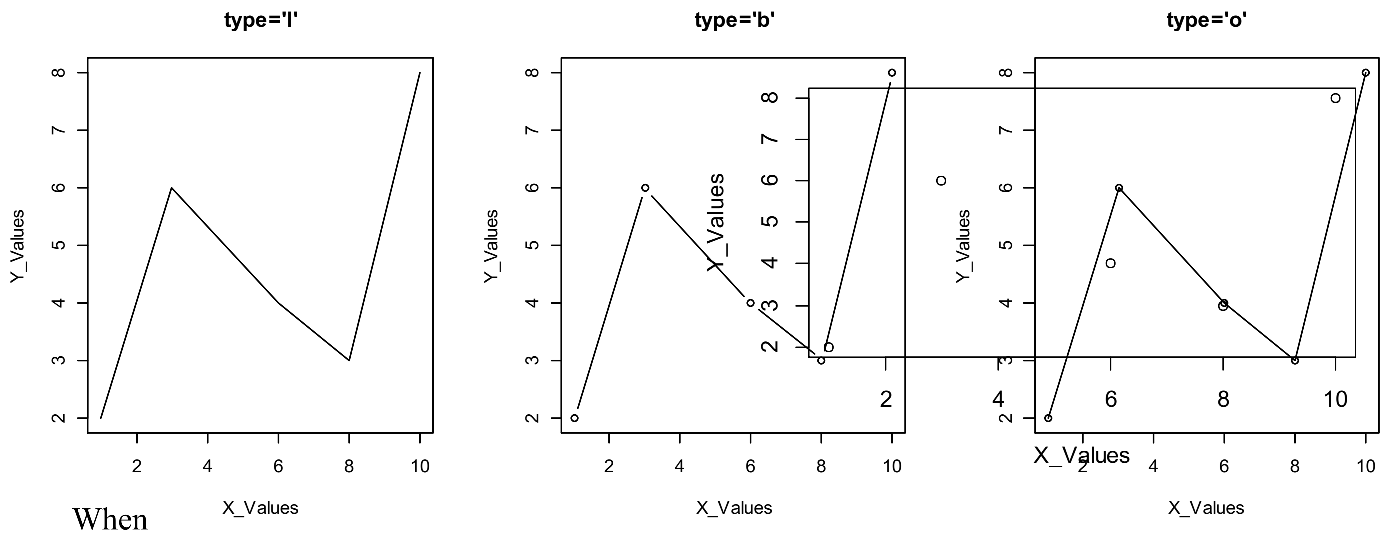


### Plot Types:

You can specify the type of plot using the argument `type=`. The types include

| To Plot | Set type to |
|---|---|
| points | `type="p"` (default) |
| lines | `type="l"` |
| points and lines | `type="b"` |
| lines with points overstruck | `type="o"` |
| empty plot | `type="n"` |

Examples using the same data as above showing the different plot types:



When the data are in a data frame, you can plot using the **~** operator (with the *y*-values on the left and the *x*-values on the right) and the argument `data=` as in the following example:

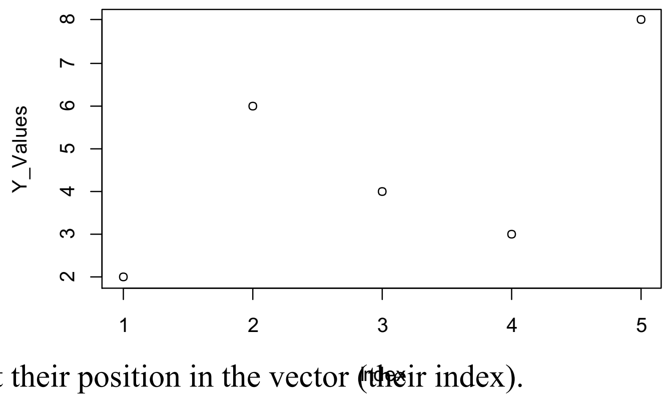
```
> DF_1<-data.frame(X_Values,Y_Values)
> DF_1
  X_Values Y_Values
1        1        2
2        3        6
3        6        4
4        8        3
5       10        8
```

```
> plot(Y_Values~X_Values,data=DF_1)
```

## Other Facts About Plot:

If you submit a vector to the `plot` command, it plots the values against their observation number or index.

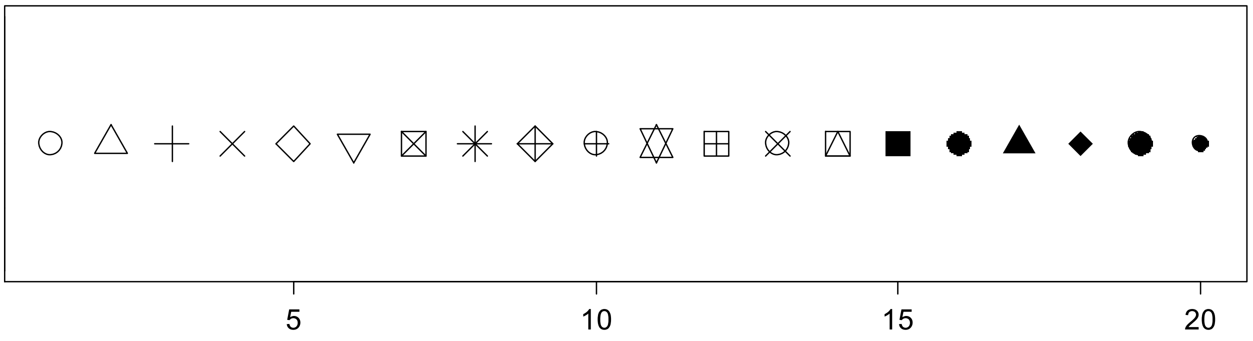
```
> Y_Values
[1]  2 6 4 3 8
> plot(Y_Values)
```



We note that the *y*-values are now plotted against their position in the vector (their index).

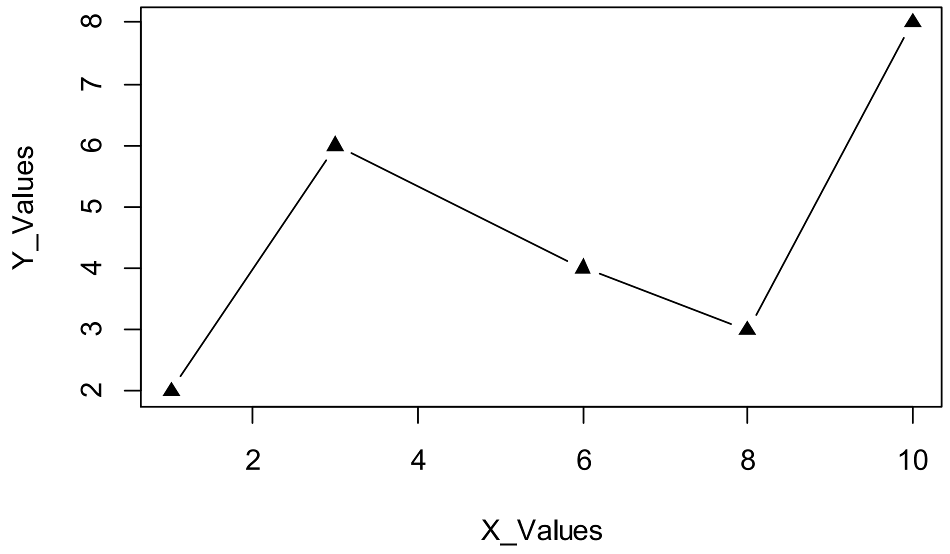
## Plotting Characters:

You can change the plotting character from the default value using the `pch=` argument which can be set to an integer (see below) or provided as a character symbol. The symbols corresponding to the integers 1-20 are



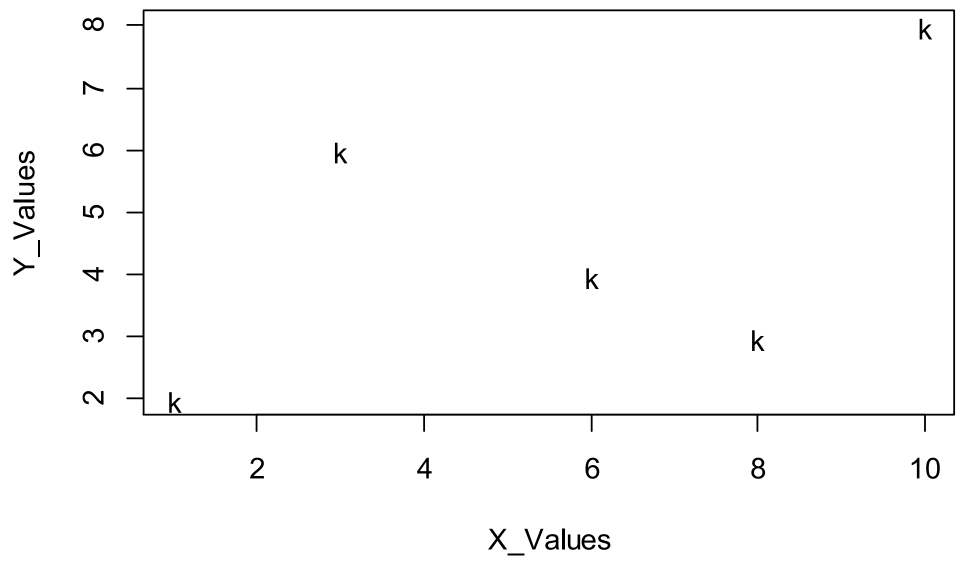
For example:

```
> plot(X_Values,Y_Values,pch=17,type='b')
```



Character strings can also be entered into the `pch` argument, for example:

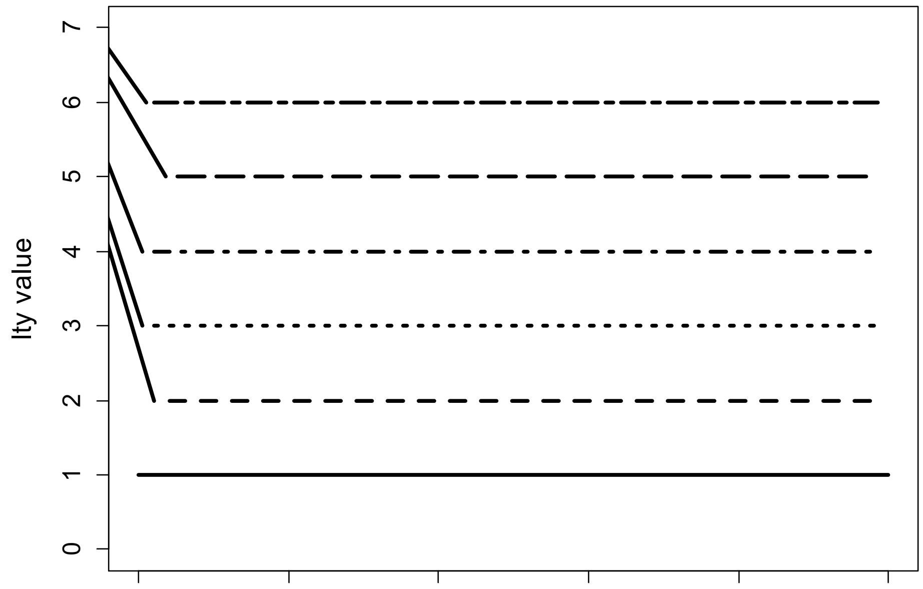
```
> plot(Y_Values~X_Values,data=DF_1, pch="k")
```



## Lines:

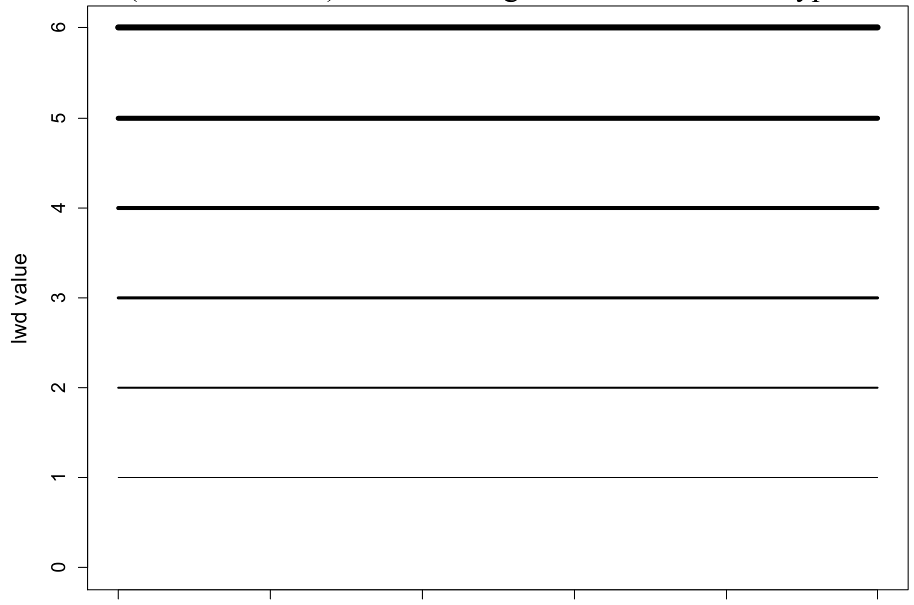
### Type:

If the plot you create involves lines, the *type* of line is controlled with the `lty` parameter. To set it, choose `lty=n` when submitting the plotting command (the default is `lty=1`). The lines available on most devices are shown below (value of *n* is on the *y*-axis).



### Weight:

The *weight* of a line is assigned using the `lwd` parameter. To set it, choose `lwd=n` when submitting the plotting command (the default is 1). Several weights for the same line type are shown below.



## Axis Labels:

When plotting, R defaults to the names of the data objects passed into the command for axis labels. You can choose your own labels using `xlab=` and `ylab=` to set them to the desired character strings. You can also use `xlab=""` or `ylab=""` to specify no label for that axis.

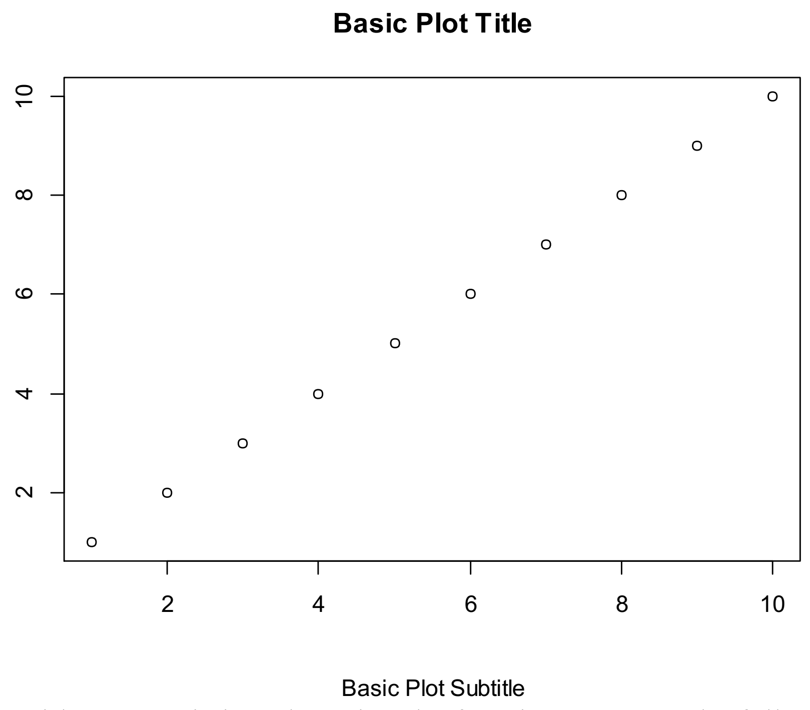
## Axis Limits:

The limits for the *x*- and *y*-axes will be chosen by R automatically. If you would like to specify these values, you can pass them in to the plotting function using `xlim=c(lowx,highx)` and `ylim=c(lowy,highy)`. You are free to specify either or both.

## Titles:

Many plots allow for titles and subtitles. These are passed in as character strings (i.e. in quotes or a character type) into the parameters `main=` for the title and `sub=` for the subtitle when plotting. For example:

```
> plot(c(1:10),c(1:10),main="Basic Plot Title",sub="Basic Plot Subtitle",
+ xlab="",ylab="")
```

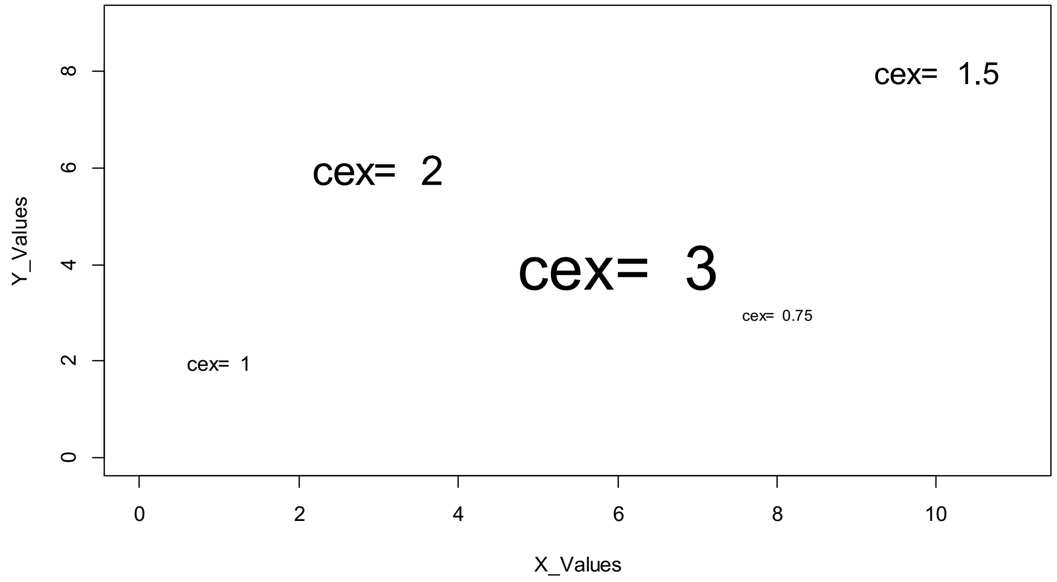


You can also add a title to an existing plot using the function `title`. The following code produces the same plot as above.

```
> plot(c(1:10),c(1:10),xlab="",ylab="")
> title(main="Basic Plot Title",sub="Basic Plot Subtitle")
```

## Text and Symbol Size

You can control the size of text and plotting symbols using the `cex` parameter (character expansion). The default value is 1, and values larger than 1 scale the character up while those less than 1 (and positive) scale the character down. When `cex=2`, for example, the character is twice the standard size. Some Examples:



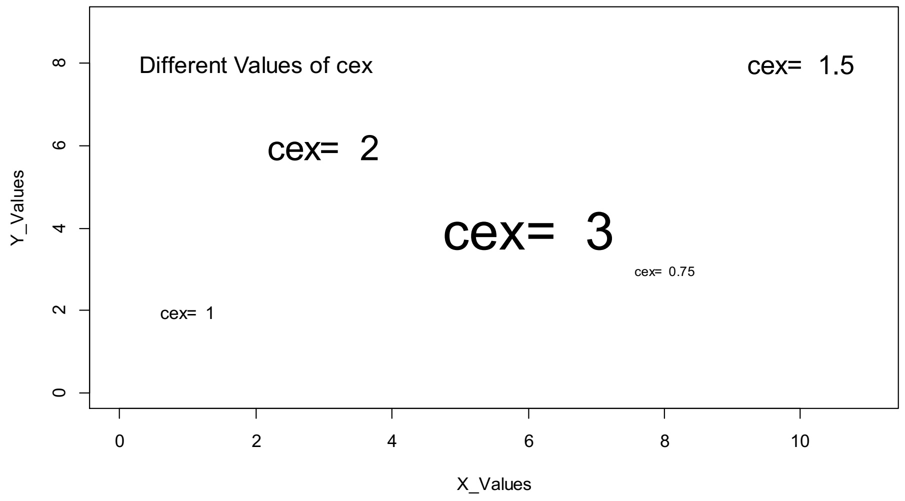
Other aspects of the plot can be controlled using variants of the `cex` parameter. These include:

| | |
|---|---|
| `cex.axis` | The magnification to be used for axis annotation. |
| `cex.lab` | The magnification to be used for x and y labels. |
| `cex.main` | The magnification to be used for main titles. |
| `cex.sub` | The magnification to be used for sub-titles. |

## Adding Text to a Plot

You can add text to a plot using the `text` command. The form is `text(x, y, labels, …)` which gives the *x* and *y* coordinates (vectors for multiple labels) of the vector of character `labels`.

```
> text(2,8,"Different Values of cex",cex=1.3)
```


Different Values of cex
cex= 1.5
cex= 2
cex= 3
cex= 0.75
cex= 1
Y_Values
X_Values