```
## Data Types in R

# Types(Basic)
# character
# numeric (real numbers)
# integer
# complex (i.e. 1.4i)
# logical (TRUE/FALSE)

## Most basic object is a vector
#   a Vector can only contains onjects of the same class
#   one exception is the list that is represented as a vecotr but allows for
#     different object classes - i.e. list(1, 'Hi', TRUE, 4.5i)

# empty vectors can be created using the vector() function.

# Numbers are generally treated as Numeric Objects (i.e. double precision real numbers)
# If you explicitely want an integet you need to specify it my adding
# an L before the number i.e.  L4

# Special Numbers Inf amnd -Inf i.e. 1/0 == Inf 1 / Inf == 0

# NaN is undefined number (not a number)   i.e. 0 / 0 == NaN
#.  also used for misisng numbers (more on this later)

## Attributes:
```

```
# names, dimnames

# dimenstions ( e.g. martices, arrays)

# class

# length

# Other user defined attibutes / metadate

# attibutes of an object can be ccessed by using the
 attributes() function.

# the c() funciton
x <- c(0.5, 0.6)          ## numeric
x <- c(TRUE, FALSE)       ## logical
x <- c(T, F)              ## logical
x <- c("a", "b", "c").    ## character
x <- 9:29                 ## integer
x <- c(1+0i, 2+4i)         ## complex


#####################
##
## Functions
##

myfunc <- function(x, y) {
 x * y
}

# goes through each x by y
myfunc(10:14,5)
```

```
# [1]  50  66  84 104 126

# goes through each x and y and multiplies them, ranges
 must be the same size
myfunc(10:14,5:9)
# [1]  50  66  84 104 126

#################




# vector
x <- vector("numeric", length = 10)
x
# [1] 0 0 0 0 0 0 0 0 0 0

# Mixing Objects (bring to least common denominator)
y <- c(1.7, "a")     # create two strings "1.7" and "a"
y <- c(TRUE, 2)      # created two numbers 1 and 2
y <- c("a", TRUE)    # creaes two string "a" and "TRUE"

# Explicit. Coercion

x <- 0:6
x
# [1] 0 1 2 3 4 5 6
class(x)
# [1] "integer"
as.number(x)
# Error in as.number(x) : could not find function
 "as.number"
as.numeric(x)
# [1] 0 1 2 3 4 5 6
as.logical(x)
```

```
# [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
as.character(x)
# [1] "0" "1" "2" "3" "4" "5" "6"
# as.complex(x)
[1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i

x <- c("a", "b", "c")
x
# [1] "a" "b" "c"
as.numeric(x)
# [1] NA NA NA
# Warning message:
# NAs introduced by coercion
as.logical(x)
# [1] NA NA NA
as.complex(x)
# [1] NA NA NA
# Warning message:
# NAs introduced by coercion
as.complex(x)
# [1] NA NA NA
# Warning message:
# NAs introduced by coercion

# Lists
x <- list(1, "a", TRUE, 0+4i)
x
# [[1]]
# [1] 1
#
# [[2]]
# [1] "a"
#
# [[3]]
```

```
# [1] TRUE
#
# [[4]]
# [1] 0+4i


## matrices

m <- matrix(nrow = 2, ncol = 3)
m
#      [,1] [,2] [,3]
# [1,]   NA   NA   NA
# [2,]   NA   NA   NA
dim(m)
# [1] 2 3

attributes(m)
# $dim
# [1] 2 3

m <- matrix(1:12, nrow = 3, ncol = 4)
m     [,1] [,2] [,3] [,4]
# [1,]    1    4    7   10
# [2,]    2    5    8   11
# [3,]    3    6    9   12

m <- matrix(1:12, nrow = 4, ncol = 3)
m
#      [,1] [,2] [,3]
# [1,]    1    5    9
# [2,]    2    6   10
# [3,]    3    7   11
# [4,]    4    8   12
```

```
# ceate a matrix by creating the dimension attribute on
 a vector
# first create a vector of 10 numbers
m <- 1:10
m
#  [1]  1  2  3  4  5  6  7  8  9 10

# no apply the dimension attribute of the vector to
 create (tranform) a matrix
dim(m) <- c(2, 5)
m

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

# comman way to make a matrix - using bind (rbind or
 cbind)
x <- 1:3
y <- 10:12
cbind(x, y)
#      x  y
# [1,] 1 10
# [2,] 2 11
# [3,] 3 12

rbind(x, y)
#   [,1] [,2] [,3]
# x    1    2    3
# y   10   11   12


## Data Type Factors
```

```
# factor are for categorical fdata (ordered and
 unordered)
# unordered (male, female)
# ordered (High, Medium, Low)

# Modeling functions (more about these later) lm() and
 glm() uses factors

x <- factor(c('yes', 'no', 'yes', 'yes', 'no'))
x
# [1] yes no  yes yes no
# Levels: no yes

# table to show number of each frequency of the levels
table(x)
# x
# no yes
#  2   3

# unclass removed the class showing the underlying data
 stored without the labels
unclass(x)
# [1] 2 1 2 2 1
# attr(,"levels")
# [1] "no"  "yes"

attr(x,"levels")
# [1] "no"  "yes"

# changing the order of the levels
# note: the default order is based on the alphabetical
 order of the items
#  (no is before yes)
```

```
x <- factor(c('yes', 'no', 'yes', 'yes', 'no'),
 levels = c("yes", "no"))
x
# [1] yes no  yes yes no
# Levels: yes no



## Data Type - Missing Values

# missing values are denoted by NA and NaN undefiend
 mathematical operations.

# is.na() is used to test objects if they are NA

# is.nan() is used to test for NaN

# NA values have a class also, they are integer NA,
 character NA etc.

# Nan value is also a NA but a NA is not an Nan

x <- c(1, 2, NA, 10, 3)
is.na(x)
# [1] FALSE FALSE  TRUE FALSE FALSE

# so NA is not a NaN
is.nan(x)
# [1] FALSE FALSE FALSE FALSE FALSE

x <- c(1, 2, NaN, NA, 3)
# no the NaN and the NA are NA's
is.na(x)
# FALSE FALSE  TRUE  TRUE FALSE
```

```
# but here only the NaN is an Nan and the NA is not an
 NaN
is.nan(x)
# [1] FALSE FALSE  TRUE FALSE FALSE



## Data Types - Data Frames


# Used to store tabular data

# Its a speial type of list having each element hte
 same length

# Each element of the list if effectivly a column and
 the length of each element
# is the number of rows

# unlice mactrices , data frames can store different
 types of objects in the
#  column (like lists) while mtrices must have every
 element the same class

# Data frames also have special attributes called
 row.names

# Data frames are usually created by calling
 read.table() or read.csv()

# Can be converted to a matric by calling data.matrix()

# Can also be freatce using data.frame()
```

```
x <- data.frame( foo = 1:4, bar = c(T, T, F, F))
x
#   foo   bar
# 1   1  TRUE
# 2   2  TRUE
# 3   3 FALSE
# 4   4 FALSE

nrow(x)
# [1] 4

ncol(x)
# [1] 2
```

Data Types - The Names Attribute

```
x <- 1:3
x
# [1] 1 2 3

names(x)
# NULL

names(x) <- c('foo', 'bar', 'norf')
names(x)
# [1] "foo"  "bar"  "norf"

names(x) <- c('One', 'Second', 'Last')
names(x)
# [1] "One"    "Second" "Last"
```

```r
# List can also have names

x <- list(a = 1, b = 2, c = 3)
x

# $a
# [1] 1
#
# $b
# [1] 2
#
# $c
# [1] 3

names(x) <- c('One', 'Second', 'Last')
x

# $One
# [1] 1
#
# $Second
# [1] 2
#
# $Last
# [1] 3


# matrices can have names (can be set using dimnames())

m <- matrix(1:4, nrow = 2, ncol = 2)
m

#      [,1] [,2]
```

```r
# [1,]    1    3
# [2,]    2    4

# assing it using a list where the fist element is the
 row names and the second element is the column names
dimnames(m) <- list(c("a", "b"), c("c", "d"))
m

#   c d
# a 1 3
# b 2 4


##
## Summary Data Types
##

# atomic classes: numeric, logical character, integer,
 complex \

# vectors, lists

# factors

# missing values

# data frames

# names



##
## Reading Tabular Data
```

```
##
## Reading data

# - read,table, read.csv for reading tabular data

# - readLines, for reading lines of a text file

# - source, for reading R code file (invers of dump)

# - dget, for reading in R code file (inverse of dput)

# - load, for reading in saved workspaces

# - unserialize, for reading a R objects in a binary
 form


## Writing Data

# there are analogous functions for writing data to
 files

# - write.table

# - writeLines

# - dump (invers of source)

# - dput (inverse of dget)

# - save

# - serialize
```

```r
## Reading Data Files with read.table
# one of the most commonly used functions for reading
 Data
# it has a few important arguments:

# - file, the name of the file, or a connection

# - header, logical including ig the file has a header
 line

# sep, a string indicating how the columns are
 separated

# colClasses, a character vector indicagting the class
 of each column in the dataset

# nrows, the number of rows in the dataset

# comment.char, a character string indicating the
 comment character

# skip, the number of lines to skip from the beginning

# stringsAsFactors, should character vaeriables be
 coded as factors?.
#  (true by default) *whats this*


# example
data <- read.table("foo.txt")

## R will automatically
```

```
# - skip lines that begin with the #

# - figure out how many rows there are (and how much
 memory needs to be allocated)

# - figure what type of variable is in each colum of
 the table
#   Note: telling R all these things directly makes R
 run faster and more efficiently

# (for read.table) assumes header = false and sep is a
 space
# read.csv is identical to read.table except that the
 default separator is
#  a comma




##
## Reading Large Tables
##

# read the help page on read.table
?read.table

# make a rough estimate of how much memory is required
 to read the file
# if its larger than the amount of ram you have, we
 need to look at a
# different approach
```

```
# set omment.char = "" to tell read.table to read line
 that otherwise
#  wourld be trated as a comment line.


# *** Important ***
# colClassas is you know the data types you can specify
 it here and this could
# double the speed of you read.table
#

# a quick and dirty way to get the datatypes is to look
 at the start of the file
# get the class information
# read the full file with the classes information
 (colClasses =)
initial <- read.table("readtable.txt", nrows = 100)
classes sapply(initial, class) # looping over each of
 the columns
                           #  and calling the class
 function
tabAll <- read.table("datatable.txt",
        colClasses = classes)

# nrows doesnt make R run faster but it does help to
 reduce the memory thats used.




##
##  Know Thy System
##

# in general, when using R with larger datasets, it's
```

```
# useful to know a few thgins about the system.

# - How much memory is available?

# - What other applications are in use?

# - Are there otehr users loggin inro the same system?

# - What operating system?

# - Is the OS 32 or 64 bit?



## Calculating Memory Requirements

# I have a data frame with 1,500,000 rows and 120
 columns, all of which are numeric data.
# Roughly how much memory is required to store this
 data frame?

# 1,500,000 x 120 x 8 bytes/numeric

# = 1440000000

# = 1440000000/2(20) bytes/MB

# = 1,373.29 MB1.34 GB



##
## Textual Data Formats: dput() and dump()
##
```

```
# - Dumping and dputting are useful because the
 resulting textual format
# is editable and in the case of corruptionm
 potentially recoverable

# - Unlike writing out a table or csv, dump and dput
 perserve the
# metadate (sacraficing some readability), so that
 another user
# doesn''t have to speciy it all over again.

# - Textual formatgs can work much better wit version
 control
# programs live subversion or git which can only track
 changes
# meaningfully in the etxt files.

# - Textual formats can be longer-lives; if there is
 corruption
# somewhere in the file, it can be easier to fix the
 problem

# - Textual formats adhere to the "Unix philosophy"

# - Downside: the format is not very space-efficient

##
## dput-ting R Objects
##

# Another way to pass data around is by deparsing the R
 object with dput
#  and resding it back in using dget.
```

```r
y <- data.frame(a = 1, b = "a")
dput(y)
# ## dput-ting R Objects

# Another way to pass data around is by deparsing the R
 object with dput
#  and resding it back in using dget.

# structure(list(a = 1, b = "a"), class = "data.frame",
#.   row.names = c(NA, -1L))

dput(y, file = "y.R")
new.y <- dget("y.R")
new.y
#   a b
# 1 1 a


## Dumping R OBjects

# Multiple objects can be deparsed using the dump
 funciton
#   and read back using source.

x <- "foo"
y <- data.frame(a = 1, b = "a")
dump(c("x", "y"), file = "data.R")
rm(x, y)
source("data.R")
y
#   a b
# 1 1 a

x
```

```r
# [1] "foo"x


## Interfaces to the Outside World

# Data are read in using connection interfaces, Connections can be makde to
#     files (most common) or to other more exotic things


# file, opens a connection to a files

# qzfile, opens a connection to a file compressed wit gzip

# bzfile, opens a connection to a file compressed wit bzip2

# url, opens a connection to a webpage


# File Connections

str(file)
# function (description = "", open = "", blocking = TRUE,
#   encoding = getOption("encoding"), raw = FALSE,
#     method = getOption("url.method", "default"))

# - description is the name of the File

# - open is a code indicating
```

```
#     - "r" read only
#     - "w" writing (and initializing a new file)
#     - "a" appending
#     - "rb", "wb", "ab" reading, writing, or appending
#        in binary mode (Windows)




# Connections
# in generaral, connections are powerful tools that
# let you navigate file or other external objects,
# in pracice, we often don't need to deal with the
#  connection directly

con <- file("foo.txt", "r")
data <- read.csv(con)
close(con)

# is the same as

data <- read.csv("foo.txt")

# no con is not useful in te above, however

# Reading Lines of a Text File

con <- gzfile("words.gz", "r")
x <- readLines(con, 10)
x
# [1] "1010"        "10-point"  "10th"      "11-point"
# [5] "12-point"    "16-point"  "18-point"  "1st"
# [9] "2"           "20-point"
close(con)
```

```r
# writeLines takes a character vector and writes each
# element one line at a time to a text file

## Read direct from Web Site

# this might take time
con <- url("https://jhsph.edu", "r")
r <- readLines(con,50)
head(r)
close(con)




##
## Subsetting - Basics
##

# There are a number of operations that can be
#  used to extra subsets of R objects.

- [ always returns an object of the same class as the
 original;
  can be used to select more than one element (there is
 one exception)

- [[ is used to extract elements of a list or a data
 frame;
  it can anly be used to extract a single element and
 the class
  of the returned object will not necessarily be a list
 of a data frame
```

```
- $ is used to extract elements of a list of data frame
 by name;
    semantics are similar to that of [[.


x <- c("a", "b", "c", "c", "d", "d")
x
x[1]
# [1] "a"

x[2]
# [1] "b"

x[1:4]
# [1] "a" "b" "c" "c"

x[x > "a"]
# [1] "b" "c" "c" "d" "d"

 u <- x > "a"
 u
 # [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE

 x[u]
 # [1] "b" "c" "c" "d" "d"

 # so teo type of index wre used above,
 #  1 is the numeric index
 #  2 is the logcial index


 ##
 ## Subsetting a list
 ##
```

```r
x <- list(foo = 1:4, bar = 0.6)
# returns a list that contains a sequence **
x[1]
# $foo
# [1] 1 2 3 4

# returns just the sequence **
x[[1]]
# [1] 1 2 3 4

x$bar
# [1] 0.6

# same as x$bar
x[["bar"]]
# [1] 0.6

# return a list with the element bar in it **
x["bar"]
# $bar
# [1] 0.6

# Extract multiple elements of a list
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
x[c(1, 3)]

# $foo
# [1] 1 2 3 4
#
# $baz
# [1] "hello"
```

```r
 # you *cannot* use the [[]] or $ when you want to
extract miltiple elements
 # from a list


 # Dynamic access
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
# assing the columns name
name = "foo"
x[[name]]       # computed index
# [1] 1 2 3 4

# assing another columns name
name = "baz"
x[[name]]
# [1] "hello"

x$name  # elemenrt name does not exist!
# NULL

x$foo   # elemenrt foo does exist.
# [1] 1 2 3 4

##
## Subsetting Nested Elements of a list

# The [[ can take an integer sequence

x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
x[[c(1,3)]]
# [1] 14

# same as x[[c(1,3)]]
x[[1]][[3]]
```

```r
# [1] 14


x[[c(2,1)]]
# [1] 3.14




##
## SubSetting Matrices
##

# Matrices can eb subsetting in the usual way with
(i,j)
#   type indices.

x <- matrix(1:6, 2, 3) # num elements , the rows, the
columns
x
#       [,1] [,2] [,3]
#  [1,]    1    3    5
#  [2,]    2    4    6

x[1, 2]
# [1] 3

x[2,1]
# [1] 2

# Indeces can also be missing

x[1, ]
# [1] 1 3 5
```

```
x[, 2]
# [1] 3 4


# By default, when a single element of a matrix is
 retrieved, it is returned
#   as a vector of length 1 rather tham a 1 x 1 matrix.
# This behavior can be turned off by settin
# drop = FALSE

x <- matrix(1:6, 2, 3)
x[1,2]
# [1] 3

# using drop = FALSE to return a martix instead of a
 vector
x[1, 2, drop = FALSE]
#      [,1]
# [1,]    3

 # drop is TRUE by detault and drops the dimension so
 returning a 1
 # dimension object instead of a 2 dimension object
 # you can disable this using the drop = FALSE argment.


# Similarly, subsetting a single column or a single row
 will give you a vector,
# now a matrix (by default)

x <- matrix(1:6, 2, 3)
# returns a vector (usually this is what you want)
x[1, ]
```

```
# [1] 1 3 5

# but if you don't

# use drop = FALSE to get the matrix
x[1, , drop = FALSE]
#      [,1] [,2] [,3]
# [1,]    1    3    5



 ##
 ## Subsetting - Partial Matching
 ##

 # Partial matching of names is allowed with [[ and $.

x <- list(aardvark = 1:5)
x$a
# [1] 1 2 3 4 5

# so $ does partial matching

x[["a"]]
# NULL

# so [[]] does not partial matching by default, we can
 change this with
#   exact = FALSE]

x[["a", exact = FALSE]]
# [1] 1 2 3 4 5
```

```
 ##
 ## Subsetting - Removing Missing Values
 ##

# A Common task is to remove missing values (NAs)

x <- c(1, 2, NA, 4, NA, 5)
bad <- is.na(x)
# output the element that are not NA **
x[!bad]
# [1] 1 2 4 5

# bad aboveis a logical vector having TRUE where the
 value is NA oherwise FALSE
bad
# [1] FALSE FALSE  TRUE FALSE  TRUE FALSE


## Removing NA Values

#  What if there are multiple things and you want to
 take
#  the subset with no misisng values?

** Important / Clever **

x <- c(1, 2, NA, 4, NA, 5)
y <- c("a", "b", NA, "d", NA, "f")

# only where both emelent in x and y are not NA then
 TRUE otherwise FALSE
good <- complete.cases(x, y)
```

```
good
# [1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE

x[good]
# [1] 1 2 4 5

y[good]
# [1] "a" "b" "d" "f"


airquality[1:6, ]
#   Ozone Solar.R Wind Temp Month Day
# 1    41     190  7.4   67     5   1
# 2    36     118  8.0   72     5   2
# 3    12     149 12.6   74     5   3
# 4    18     313 11.5   62     5   4
# 5    NA      NA 14.3   56     5   5
# 6    28      NA 14.9   66     5   6

good <- complete.cases(airquality)

airquality[good, ][1:6, ]
#   Ozone Solar.R Wind Temp Month Day
# 1    41     190  7.4   67     5   1
# 2    36     118  8.0   72     5   2
# 3    12     149 12.6   74     5   3
# 4    18     313 11.5   62     5   4
# 7    23     299  8.6   65     5   7
# 8    19      99 13.8   59     5   8



## Vectorized Operastions
```

```
#  Many operations in R are vectorized making code more
efficient,
#  concise, and easier to read.

x <- 1:4; y <- 6:9
x + y
# [1]  7  9 11 13

x > 2
# [1] FALSE FALSE  TRUE  TRUE

x >= 2
# [1] FALSE  TRUE  TRUE  TRUE

y == 8
# [1] FALSE FALSE  TRUE FALSE

x * y
# [1]  6 14 24 36

x / y
# [1] 0.1666667 0.2857143 0.3750000 0.4444444


##
## Vectorized Matrix Operastions
##

x <- matrix(1:4, 2, 2); y <- matrix(rep(10, 4), 2, 2)
x * y   ## element-wise multiplication
#      [,1] [,2]
# [1,]   10   30
# [2,]   20   40
```

```
x / y
#      [,1] [,2]
# [1,]  0.1  0.3
# [2,]  0.2  0.4

x %*% y   ## true matrix multiplication
#      [,1] [,2]
# [1,]   40   40
# [2,]   60   60
```