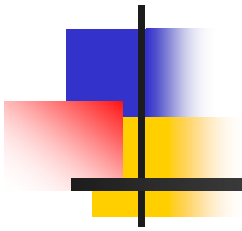
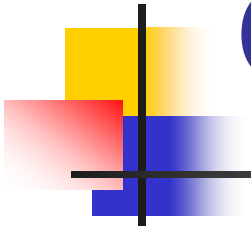


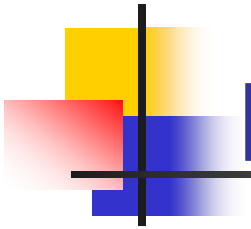
Lecture 2: Data Input

Week-02: Probability and Statistical Methods



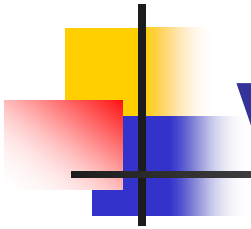
Outline

- Data Types
- Importing Data
- Keyboard Input
- Database Input
- Exporting Data
- Viewing Data
- Variable Labels
- Value Labels
- Missing Data
- Date Values



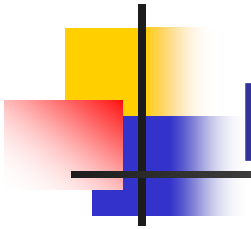
Data Types

R has a wide variety of data types including scalars, vectors (numerical, character, logical), matrices, dataframes, tibble, and lists etc.



Vectors

```
a <- c(1, 2, 5.3, 6, -2, 4) # numeric vector
b <- c("one", "two", "three") # character vector
c <- c(TRUE,TRUE,TRUE,FALSE,TRUE,FALSE)
#logical vector
Refer to elements of a vector using subscripts.
a[c(2,4)] # 2nd and 4th elements of vector
```



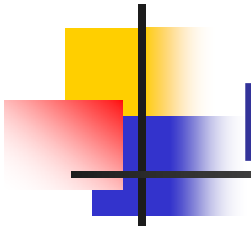
Matrices

All columns in a matrix must have the same mode(numeric, character, etc.) and the same length.

The general format is

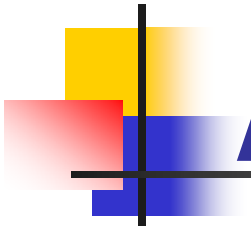
mymatrix <- **matrix(***vector*, **nrow=***r*, **ncol=***c*, **byrow=***FALSE*, **dimnames=list(***char_vector_rownames*, *char_vector_colnames***))**

byrow=TRUE indicates that the matrix should be filled by rows. **byrow=FALSE** indicates that the matrix should be filled by columns (the default). **dimnames** provides optional labels for the columns and rows.



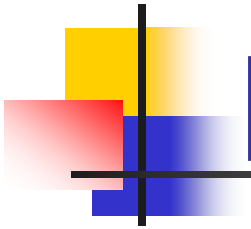
Matrices

```
# generates 5 x 4 numeric matrix
   y<-matrix(1:20, nrow=5,ncol=4)
# another example
   cells <- c(1, 26, 24, 68)
   rnames <- c("R1", "R2")
   cnames <- c("C1", "C2")
   mymatrix <- matrix(cells, nrow=2, ncol=2,
   byrow=TRUE, dimnames=list(rnames, cnames))
#Identify rows, columns or elements using subscripts.
x[,4] # 4th column of matrix
   x[3,] # 3rd row of matrix
   x[2:4,1:3] # rows 2,3,4 of columns 1,2,3
```



Arrays

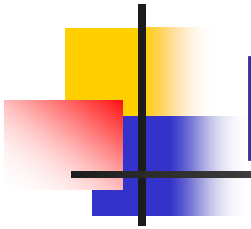
Arrays are similar to matrices but can have more than two dimensions. See **help(array)** for details.



Data frames

A data frame is more general than a matrix, in that different columns can have different modes (numeric, character, factor, etc.).

```
d <- c(1, 2, 3, 4)
e <- c("red", "white", "red", NA)
f <- c(TRUE,TRUE,TRUE,FALSE)
mydata <- data.frame(d,e,f)
names(mydata) <- c("ID","Color","Passed")
  #variable names
```



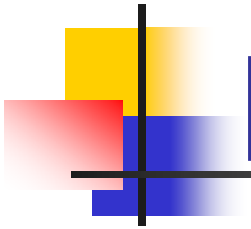
Data frames

There are a variety of ways to identify the elements of a dataframe .

myframe[3:5] # columns 3,4,5 of dataframe

myframe[c("ID","Age")] # columns ID and Age from dataframe

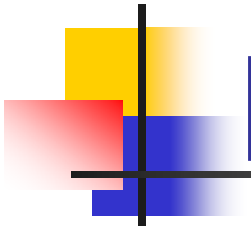
myframe$X1 # variable x1 in the dataframe



Lists

An ordered collection of objects (components). A list allows you to gather a variety of (possibly unrelated) objects under one name.

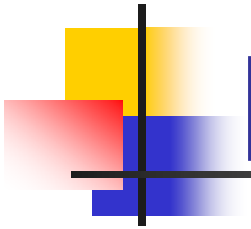
```
# example of a list with 4 components -
  # a string, a numeric vector, a matrix, and a scaler
  w <- list(name="Fred", mynumbers=a,
  mymatrix=y, age=5.3)

  # example of a list containing two lists
  v <- c(list1,list2)
```



Lists

Identify elements of a list using the [[]] convention.

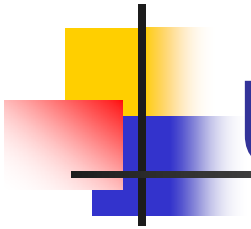
```
mylist[[2]] # 2nd component of the list
```



Factors

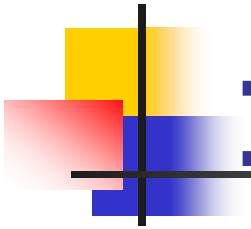
Tell **R** that a variable is **nominal** by making it a factor**.** The factor stores the nominal values as a vector of integers in the range [1... k] (where k is the number of unique values in the nominal variable), and an internal vector of character strings (the original values) mapped to these integers.

```
# variable gender with 20 "male" entries and
# 30 "female" entries
  gender <- c(rep("male",20), rep("female", 30))
  gender <- factor(gender)
# stores gender as 20 1s and 30 2s and associates
# 1=female, 2=male internally (alphabetically)
# R now treats gender as a nominal variable
  summary(gender)
```



Useful Functions

```
length(object) # number of elements or components
str(object)    # structure of an object
class(object)  # class or type of an object
names(object)  # names
c(object,object,...)  # combine objects into a vector
cbind(object, object, ...) # combine objects as columns
rbind(object, object, ...) # combine objects as rows
ls()       # list current objects
rm(object) # delete an object
newobject <- edit(object) # edit copy and save a newobject
fix(object)               # edit in place
```



Importing Data

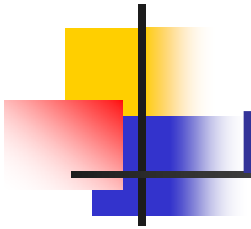
Importing data into **R** is fairly simple.

For Stata and Systat, use the **foreign** package.

For SPSS and SAS I would recommend the **Hmisc** package for ease and functionality.

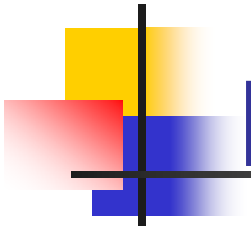
See the **Quick-R** section on **packages**, for information on obtaining and installing the these packages.

Example of importing data are provided below.



From A Comma Delimited Text File

```
# first row contains variable names, comma is
  separator
# assign the variable id to row names
# note the / instead of \ on mswindows systems

  mydata <- read.table("c:/mydata.csv",
  header=TRUE, sep=",", row.names="id")
```

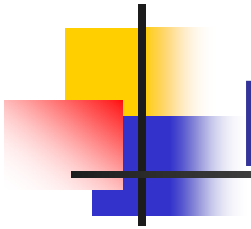


From Excel

The best way to read an Excel file is to export it to a comma delimited file and import it using the method above.

On windows systems you can use the **RODBC** package to access Excel files. The first row should contain variable/column names.

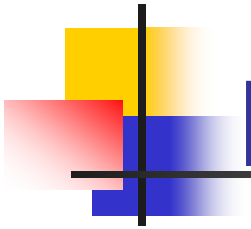
```
# first row contains variable names
# we will read in workSheet mysheet
library(RODBC)
channel <- odbcConnectExcel("c:/myexel.xls")
mydata <- sqlFetch(channel, "mysheet")
odbcClose(channel)
```



Keyboard Input

Usually you will obtain a dataframe by importing it from **SAS**, **SPSS**, **Excel**, **Stata**, a database, or an ASCII file. To create it interactively, you can do something like the following.

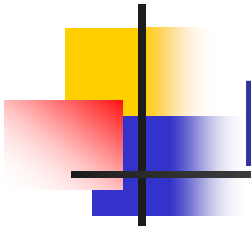
```
# create a dataframe from scratch
age <- c(25, 30, 56)
gender <- c("male", "female", "male")
weight <- c(160, 110, 220)
mydata <- data.frame(age,gender,weight)
```



Keyboard Input

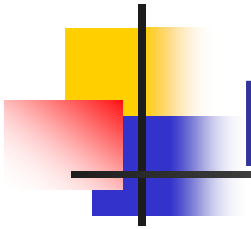
You can also use **R**'s built in spreadsheet to enter the data interactively, as in the following example.

```
# enter data using editor
  mydata <- data.frame(age=numeric(0),
  gender=character(0), weight=numeric(0))
  mydata <- edit(mydata)
  # note that without the assignment in the line
  above,
  # the edits are not saved!
```



Exporting Data

There are numerous methods for exporting **R** objects into other formats . For SPSS, SAS and Stata. you will need to load the **foreign** packages. For Excel, you will need the **xlsReadWrite** package.



Exporting Data

To A Tab Delimited Text File

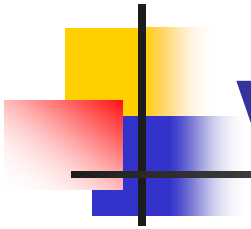
```
write.table(mydata, "c:/mydata.txt", sep="\t")
```

To an Excel Spreadsheet

```
library(xlsReadWrite)
   write.xls(mydata, "c:/mydata.xls")
```

To SAS

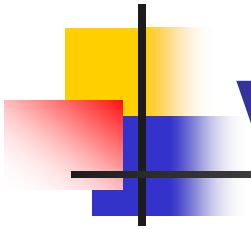
```
library(foreign)
   write.foreign(mydata, "c:/mydata.txt",
   "c:/mydata.sas",   package="SAS")
```



Viewing Data

There are a number of functions for listing the contents of an object or dataset.

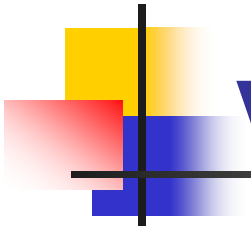
- ls() # list objects in the working environment
- names(mydata) # list the variables in mydata
- str(mydata) # list the structure of mydata
- levels(mydata$v1) # list levels of factor v1 in mydata
- dim(object) # dimensions of an object



Viewing Data

There are a number of functions for listing the contents of an object or dataset.

- class(object) # class of an object (numeric, matrix, dataframe, etc)
- mydata # print mydata
- head(mydata, n=10) # print first 10 rows of mydata
- tail(mydata, n=5) # print last 5 rows of mydata

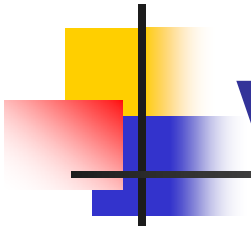


Variable Labels

R's ability to handle variable labels is somewhat unsatisfying.

If you use the **Hmisc** package, you can take advantage of some labeling features.

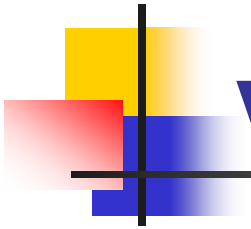
```
library(Hmisc)
  label(mydata$myvar) <- "Variable label for variable
  myvar"
  describe(mydata)
```



Variable Labels

Unfortunately the label is only in effect for functions provided by the **Hmisc** package, such as **describe()**. Your other option is to use the variable label as the variable name and then refer to the variable by position index.

```
names(mydata)[3] <- "This is the label for variable 3"
mydata[3] # list the variable
```

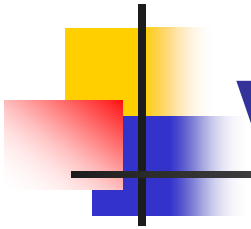


Value Labels

To understand value labels in **R**, you need to understand the data structure factor.

You can use the factor function to create your own value lables.

```
# variable v1 is coded 1, 2 or 3
# we want to attach value labels 1=red, 2=blue,3=green
  mydata$v1 <- factor(mydata$v1,
  levels = c(1,2,3),
  labels = c("red", "blue", "green"))
# variable y is coded 1, 3 or 5
# we want to attach value labels 1=Low, 3=Medium, 5=High
```

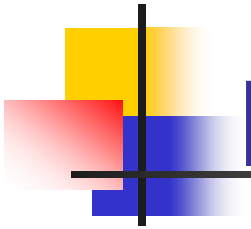


Value Labels

```
mydata$v1 <- ordered(mydata$y,
    levels = c(1,3, 5),
    labels = c("Low", "Medium", "High"))
```

Use the **factor()** function for **nominal data** and the **ordered()** function for **ordinal data**. **R** statistical and graphic functions will then treat the data appropriately.

Note: factor and ordered are used the same way, with the same arguments. The former creates factors and the later creates ordered factors.

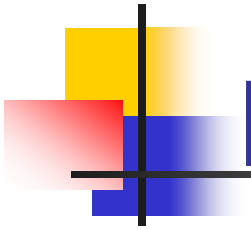


Missing Data

In **R**, missing values are represented by the symbol **NA** (not available) . Impossible values (e.g., dividing by zero) are represented by the symbol **NaN** (not a number). Unlike SAS, **R** uses the same symbol for character and numeric data.

Testing for Missing Values

```
is.na(x) # returns TRUE of x is missing
y <- c(1,2,3,NA)
is.na(y) # returns a vector (F F F T)
```



Missing Data

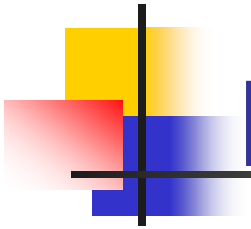
Recoding Values to Missing

```
# recode 99 to missing for variable v1
# select rows where v1 is 99 and recode column v1
mydata[mydata$v1==99,"v1"] <- NA
```

Excluding Missing Values from Analyses

Arithmetic functions on missing values yield missing values.

```
x <- c(1,2,NA,3)
mean(x)           # returns NA
mean(x, na.rm=TRUE) # returns 2
```



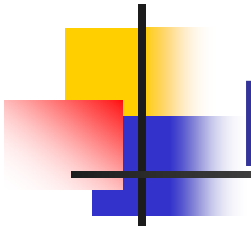
Missing Data

The function **complete.cases()** returns a logical vector indicating which cases are complete.

```
# list rows of data that have missing values
mydata[!complete.cases(mydata),]
```

The function **na.omit()** returns the object with listwise deletion of missing values.

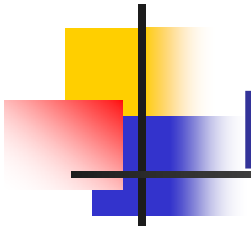
```
# create new dataset without missing data
newdata <- na.omit(mydata)
```



Missing Data

Advanced Handling of Missing Data

Most modeling functions in **R** offer options for dealing with missing values. You can go beyond pairwise of listwise deletion of missing values through methods such as multiple imputation. Good implementations that can be accessed through **R** include **Amelia II**, **Mice**, and **mitools**.



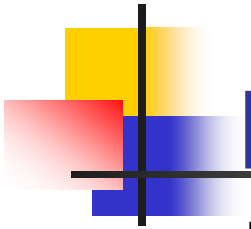
Date Values

Dates are represented as the number of days since 1970-01-01, with negative values for earlier dates.

```
# use as.Date( ) to convert strings to dates
mydates <- as.Date(c("2007-06-22", "2004-02-13"))
# number of days between 6/22/07 and 2/13/04
    days <- mydates[1] - mydates[2]
```

Sys.Date( ) returns today's date.

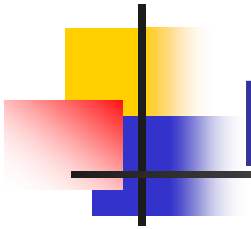
Date() returns the current date and time.



Date Values

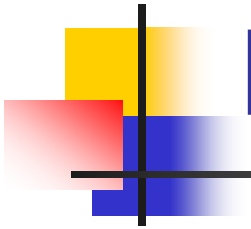
The following symbols can be used with the format( ) function to print dates.

Symbol	Meaning	Example
%d	day as a number (0-31)	01-31
%a	abbreviated weekday	Mon
%A	unabbreviated weekday	Monday
%m	month (00-12)	00-12
%b	abbreviated month	Jan
%B	unabbreviated month	January
%y	2-digit year	07
%Y	4-digit year	2007



Date Values

```
# print today's date
today <- Sys.Date()
format(today, format="%B %d %Y")
     "June 20 2007"
```



References:

Lecture slides: Courtesy of New Jersey Institute of Technology (NJIT)