Session 1
Getting Started with R

1.1 Overview

The R website defines R as "R is a system for statistical computation and graphics. It consists of a language plus a run-time environment with graphics, a debugger, access to certain system functions, and the ability to run programs stored in script files." For those unfamiliar with R, this may not be too helpful. We can only add that we have found that we can do within R the computations and data plotting available in statistical packages such as SPSS and SAS, and with certain advantages that we will discuss in Section 1.2. Presumably, as we go through the material in this website, we will have a better understanding of what R is and does.

1.2 Why R?

R is a free, open-source, collection of functions contributed by many individuals. It provides everything, including graphics, that is available in statistical packages such as SAS, SPSS, and Systat and it does so with great flexibility. For example, as we shall see in the Session 2 script, when plotting data we have total control over font size, color, the position of the legend, the types of symbols, and a host of other parameters. To take another example, we can use R to calculate power, or to determine the sample size needed to have a specified level of power to detect a given population effect. R also provides access to advanced analytical methods such as logistic regression and mixed-models analysis. Also, in addition to the many functions available, the R language is so designed that we can create our own functions. Once we have written an R function, we only need to invoke a single command with the function name and the appropriate arguments each time we wish to execute the function.

Another important aspect of R is that help is readily available. If we know the name of a function, we can find the details from R's help menu. If we have a topic in mind, such as ANOVA, or multiple regression, or testing contrasts, we can find help in manuals provided by R, in responses on the internet from others who have had a similar question, and at dozens of websites containing lecture notes by many statisticians. We will have more to say about getting help in Section 1.4.

Although R is freely distributed, applicable to a vast variety of data exploration and analysis methods, and flexible to use, there are some potential drawbacks. Learning R is easier than learning most computer languages but is a more difficult task than using a menu. Error messages are sometimes difficult to understand. It is also possible to have an erroneous result without realizing it. However, these possible drawbacks are more than outweighed by the advantages.

1.3 Getting Started

Installing R. Detailed instructions for installing R with either Windows or Mac operating systems may be found at
http://cran.r-project.org/bin/windows/base/rw-FAQ.html#How-do-I-install-R-for-Windows_003f
or
http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html#How-to-install-packages.

These sites also provide other useful information. However, if you want to plunge right in, with the installation, go to *http://cran.r-project.org/mirrors.html*
Find your country listed; for the United States, there are several sites. Any will do, but it is probably fastest to select the one nearest you. Once you've clicked on the site you want, a page will come up and in the upper right there is a box with the statement "Precompiled binary distributions of the base system and contributed packages." Beneath are hyperlinks to download R to either Windows, Mac, or Linux operating systems. Select your operating system, then select "base packages." Then click on the download link for the current version. In Windows 7 (my current operating system), a setup file was installed in the *Downloads* folder of *My Documents*. Double-click on this and select "Run" and follow instructions. Windows may caution you about running a program from an unknown source; you should ignore the warning.

Once the setup is complete, you will have an icon on your screen and also in your start menu, and perhaps in the quick launch section of your task bar, depending on options chosen during the setup. There will also be an R folder within the *Program Files* directory; this in turn will contain many subfolders. It is worthwhile exploring these; helpful manuals are available in R's *bin* directory , as well as many packages (in R's *library* folder) containing functions that will be of use. We will add to those packages as we go through the scripts.

The Working Directory. At this point you should create a working directory. In my case, this is C:\R_Stats\WD. My WD directory also contains a subfolder for the data sets that I intend to use in the sessions that follow, as well as a subfolder for the scripts (sets of R commands and comments) that I use to analyze and plot the data in those files. Of course, you can set up any organization you choose with any names you wish.

There are two basic ways of getting to any functions or objects that you may have saved in your working directory during a previous session. One is to click on the icon on your Start menu or on the screen. Then when the R console appears, at the prompt (>) set the working directory; e.g.,

```
>setwd("C:/R_Stats/WD")
```

(Throughout this session, I will use R's default colors – red for commands and blue for output.) Before going on, we should note several aspects of this command. First, the "setwd" command consists of a function ("setwd") and an object within parentheses (the path in this example), This is typical of R commands although there are instances in which the parentheses are empty, as we shall see later in this session. Second, in naming paths, R uses either *forward* slashes, or double backslashes (\\). Third, R is sensitive to cases. So, for example,

```
> Setwd("C:/R_stats/WD")
```

results in an error message:

```
Error: could not find function "Setwd"
```

The second approach to loading a saved workspace requires you to right click on the icon you will normally use to start R (you may wish to do this on both the Start and screen icons), select "properties", and change the name in the "Start in" line to that of your startup directory. In my case, I have "C:\R_Stats\WD". in my "start in" slot in both the start menu and the screen icon. Once that is done, whenever I click on the icon in either place, I have immediate access to any objects and functions saved previously in

"WD." This is the better method if you, like me, are using the same workspace repeatedly. With either method, you can check on which working directory has been loaded by:

```
> getwd()
```

which in my example responds with

```
[1] "C:/R_Stats/WD".
```

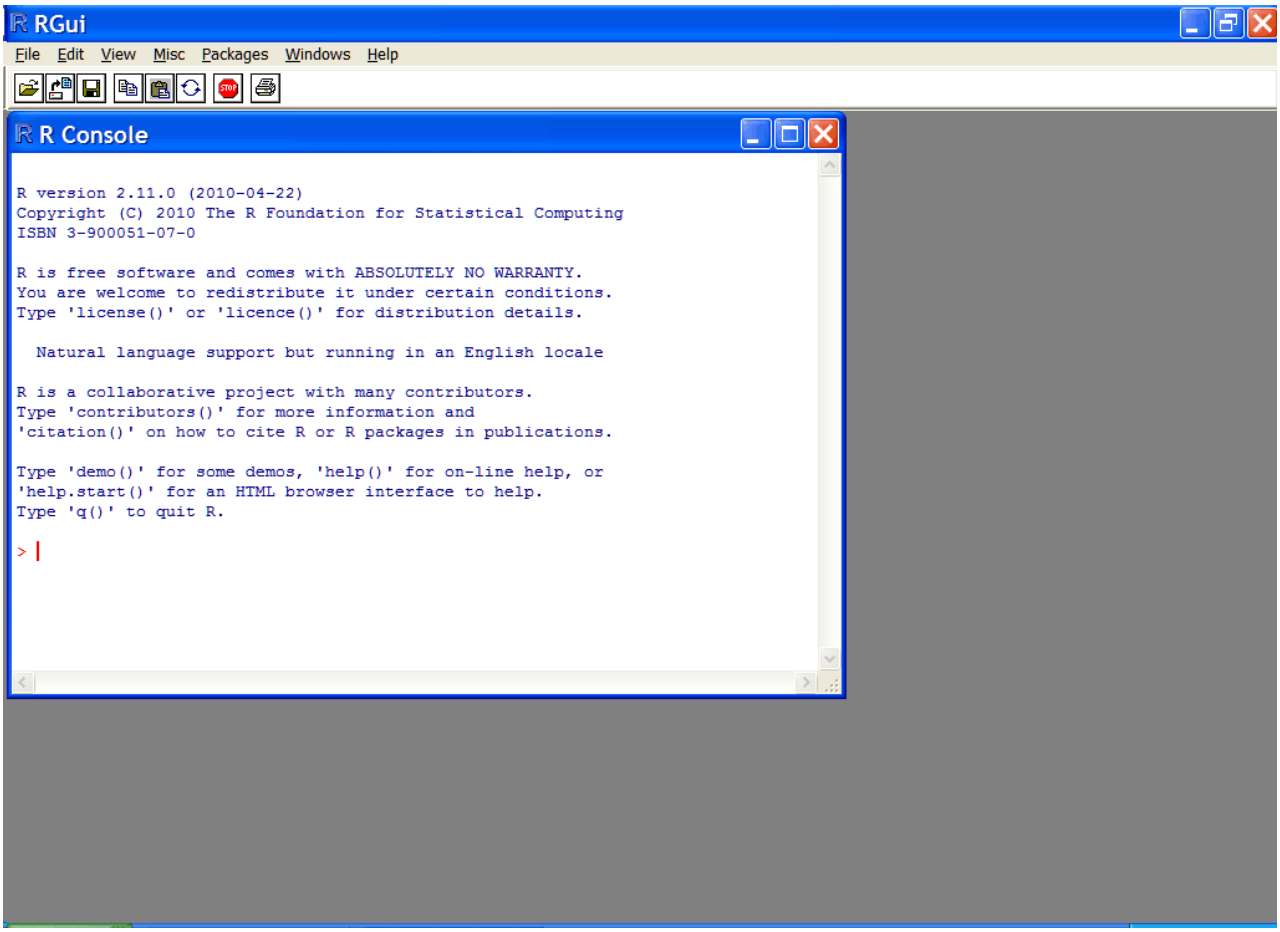
.RProfile. When a session starts, R looks for the system's .RProfile file, a text file that loads the base library and sets environmental variables. R then looks for the user's ,RProfile. This file should be stored in your working directory. While not essential, it provides a way of loading any functions that you find useful. It should not be a copy of the system .RProfile because that will execute commands twice. I found several functions in an .RProfile developed by Kellerman to be useful and copied them to the WD folder on my hard drive. Kellerman's profile can be obtained by the command:

```
>source("http://psych-swiki.colorado.edu:8080/LearnR/uploads/10/Rprofile.3.site")
```

Rather than loading the functions with this statement each time you start R, it (or whatever .RProfile you have) should be stored in your current working directory. To do this with the Kellerman RProfile, enter the URL in your browser, then open the file in Notepad and save it as .RProfile in your working directory (do not omit the dot before the R). Because websites appear and disappear, I have placed two of the functions I found most useful in the Appendix at the end of this introduction as well as in the functions.R script. If you don't wish to download the Kellerman profile, you may use these as the base for your .RProfile, perhaps adding other functions that you have found elsewhere or have written yourself.

The R Console. When you first start R, you will see the following screen. The leftmost icon on the button bar allows us to open a script, a list of commands that can be run as an executable program; we will have more to say about scripts later in this session. The next allows us to load a workspace. The next button saves the workspace. The others permit editing functions (e.g., "copy," "paste"), stopping ongoing computations (handy if you have somehow gotten yourself in an infinite loop), and printing. The menu bar above the buttons has seven pull-down menus that access various commands, some of which are also available from the button bar.

You may have several workspaces, one for each of several projects, within your working directory. The second icon from the left allows you to load any workspace within your working directory, or select "File" and then "Load workspace" from the menu above the icons.



1.4 Getting Help

The R Reference Card. A handy guide to many of the base functions, organized by category (e.g., "Getting help," Input and output", "Math"), is available as a pdf file at *http://cran.r-project.org/doc/contrib/Short-refcard.pdf*
We suggest downloading and printing this. The following is a brief excerpt:

R Reference Card

by Tom Short, EPRI PEAC, tshort@epri-peac.com 2004-11-07
Granted to the public domain. See www.Rpad.org for the source and latest version. Includes material from *R for Beginners* by Emmanuel Paradis (with permission).

Getting help

Most R functions have online documentation.

`help(topic)` documentation on topic
`?topic` id.
`help.search("topic")` search the help system
`apropos("topic")` the names of all objects in the search list matching the regular expression "topic"
`help.start()` start the HTML version of help
`str(a)` display the internal *str*ucture of an R object
`summary(a)` gives a "summary" of a, usually a statistical summary but it is *generic* meaning it has different operations for different classes of a
`ls()` show objects in the search path; specify pat="pat" to search on a pattern

Help Functions. As the excerpt above suggests, there are many commands that provide help. Suppose we wish to have a boxplot of a set of scores and need help. We can use either

>?boxplot

or

>help(boxplot)

An HTML file on your browser responds with detailed information about the function and its parameters, with examples, and a clickable "See also" directing us to related help files.

Now suppose we wanted to do a *t* test in R. We try

> help(t test)

and get an error message"

Error: unexpected symbol in "help(t test"

But try

> apropos("test")

which yielded a list of functions involving "test", including "t.test".

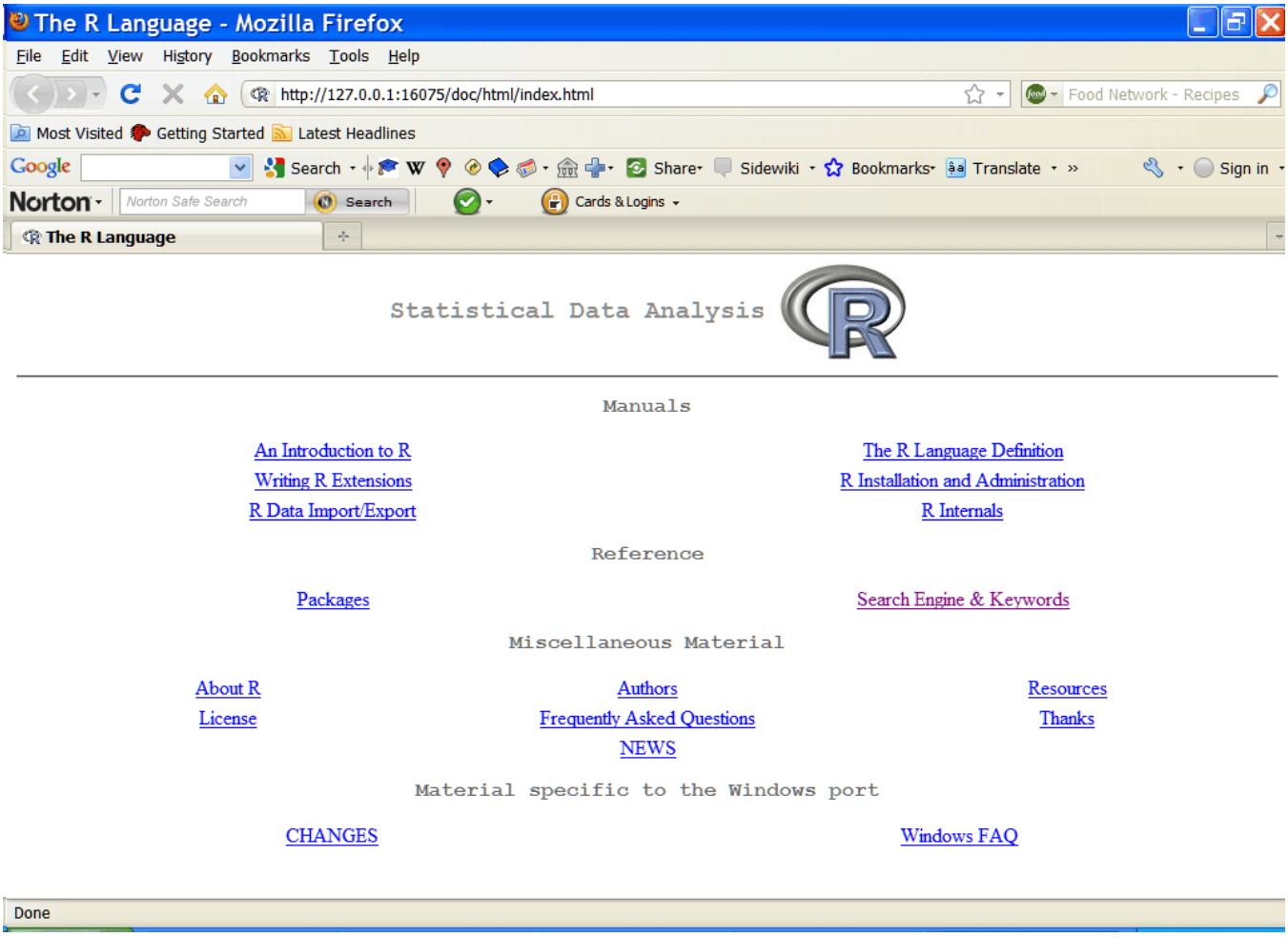
```
 [1] ".valueClassTest"        "ansari.test"
 [3] "bartlett.test"          "binom.test"
 [5] "Box.test"               "chisq.test"
 [7] "cochranq.test"          "cor.test"
 [9] "file_test"              "fisher.test"
[11] "fligner.test"           "friedman.test"
[13] "kruskal.test"           "ks.test"
[15] "levene.test"            "mantelhaen.test"
[17] "mauchley.test"          "mauchly.test"
[19] "mcnemar.test"           "mood.test"
[21] "ncv.test"               "oneway.test"
[23] "outlier.test"           "pairwise.prop.test"
[25] "pairwise.t.test"        "pairwise.wilcox.test"
[27] "poisson.test"           "power.anova.test"
[29] "power.prop.test"        "power.t.test"
[31] "PP.test"                "prop.test"
[33] "prop.trend.test"        "quade.test"
[35] "shapiro.test"           "t.test"
[37] "testInheritedMethods"   "testPlatformEquivalence"
[39] "testVirtual"            "var.test"
[41] "wilcox.test"
```

Now we can type ?t.test or help(t.test)

An alternative is to search the cran-r manuals:

> help.start()

opens *http://127.0.0.1:16075/doc/html/index.html* which presents the following internet page:.

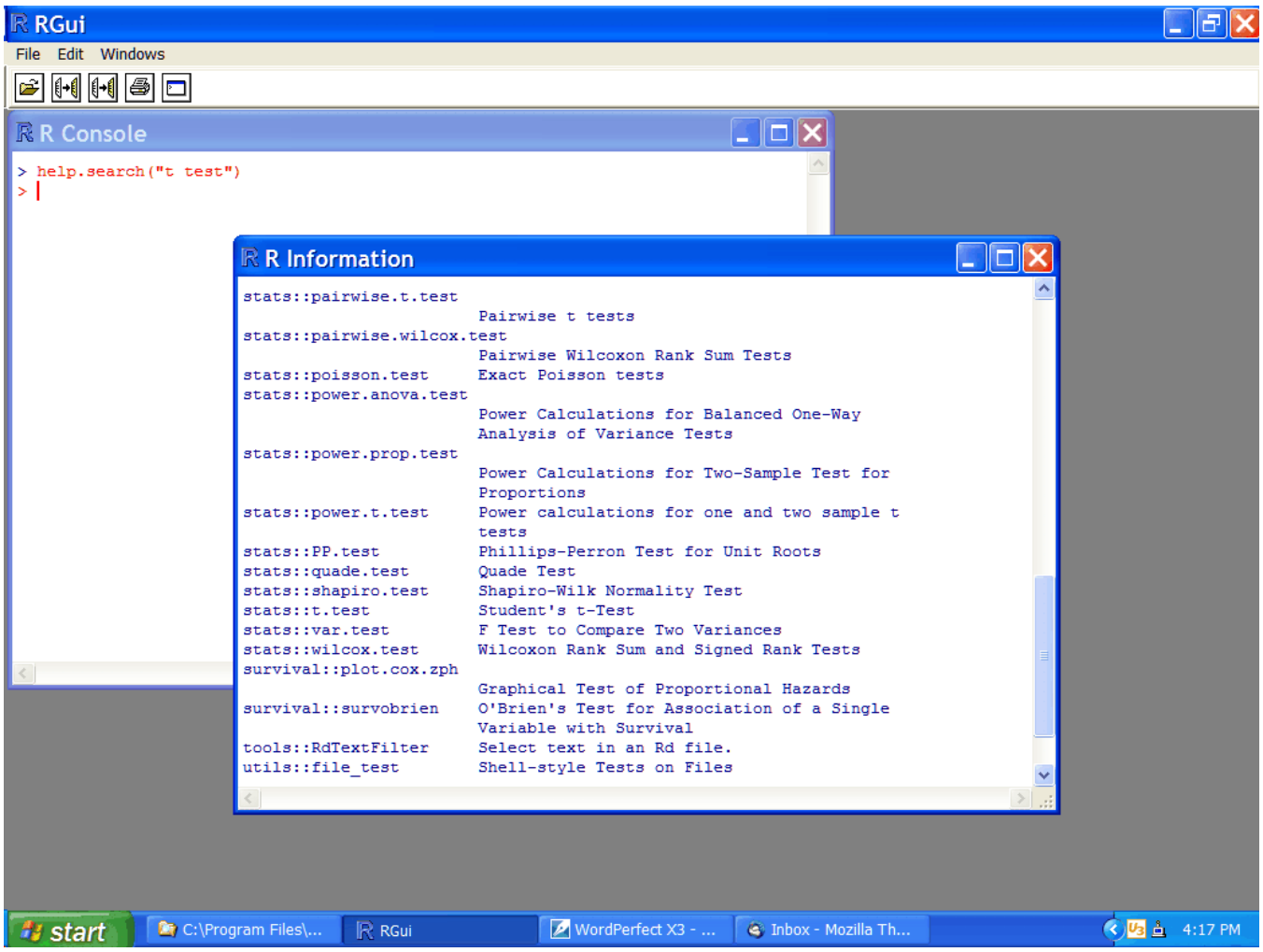


We selected"Search Engine and Keywords", then typed "t test" in the search box that appeared. Scanning down the lengthy list of functions (see figure on next page), we found "stats::t.test"and selecting it found the required information for the function t.test which is in the "Stats" package in the R program files directory. In future, if we again need information about this function, we only need to type at the R console:

>?t.test

While the sample front page which gave us initial access to the search engine is available, you may want to select some of the other options such as "An Introduction to R." These manuals (several of which are also in the doc subfolder of the R program folder on your hard drive) can be helpful and therefore it is worth becoming familiar with their contents.

We also could have used the function 'help.search("t test")ʻ which opens a window with the list we found on the internet. Scanning down we find the package, the function name, and a brief description ("Studentʻs T-test"). We can now search using "?t.test" or "help(t.test)".



For many purposes, I have found the use of a search engine such as Google to be very helpful. In Session 2 ("Exploring the Data"), I wanted to plot error bars but could find nothing helpful using the methods discussed so far; for example, the "Search engine and keywords" yielded "No results found ." However, googling "R error bars" produced a number of websites that discussed error bars in data plots, with examples. You may have to examine several of the sites to find the one that best solves your problem but you usually will find at least one. This sometimes involves installing a package not in the base, but that may prove useful for other problems.

Several search engines are also available at *http://cran.r-project.org/*. Click on "Search" to begin. One of the engines available there that I have found useful is *http://www.rseek.org/*.

References. There are many books on R (and on the S language from which it was derived, and which is similar to R in most respects), and many useful websites. The following provides a starting point and references in some of these sources may also prove useful.

Baayen, R. H. *Analyzing Linguistic Data. A Practical Introduction to Statistics*. Freely downloadable from *www.monkproject.org/MONK.wiki/attachments/2006595/2130450*

Baron, J. Notes on the use of R for psychology experiments and questionnaires. Freely downloadable from http://www.psych.upenn.edu/~baron/rpsych/rpsych.html

Crawley MJ (2005) *Statistics: An Introduction using R*. John Wiley & Sons, Ltd.

Crawley MJ (2007) *The R Book*. John Wiley & Sons, Ltd.
Everitt, B.S. & Hothorn, T. (2009) A Handbook of Statistical Analysis Using R (2nd ed.). CRC Press
Faraway, J.J. (2004) *Linear Models with R*. CRC Press
Keen, K. J. (2010) *Graphics for Statistics and Data Analysis with R*. CRC Press
Venables, W.N. & Smith D.M. *An Introduction to R*, Freely downloadable from *http://cran.r-project.org/doc/manuals/R-intro.pdf* (A modified version can be found in the doc/manual subfolder under R in your Program Files directory)

Two of many useful websites are
http://www.ats.ucla.edu/stat/r/
http://cran.r-project.org/doc/contrib/Lemon-kickstart/index.html

1.5 Objects and their Attributes

The basic components of the R programming language are functions and the objects upon which they operate. We have already had examples of a few functions; e.g. *setwd, source, help*. There are many more, some of which we will use as we proceed through the scripts. Here our focus is on objects, although our illustrations will involve various functions. Our discussion of objects will be brief and somewhat superficial. "An Introduction to R" (Venables & Smith), listed in the reference list, provides a far more detailed presentation, especially in Chapters 3 - 6.

Modes. There are several possible modes but we will be concerned primarily with numeric, character, logical, and list modes. Consider the following *numeric* vector:

```
> x.num = c(5,7,8,2,9)
```

Note that the "c" is used to concatenate the numbers; this is a very frequent operation. Also note that, although we used an equals sign in defining our vector, many, perhaps most, sources would have

```
> x.num <- c(5,7,8,2,9)
```

However, either "=" or "<-" works; I tend to save a keystroke.

We can verify the mode of x.num:

```
> mode(x.num)
[1] "numeric"
```

We often read in data that have missing values; these are represented in R by "NA." For example,

```
> x.miss = c(1,2,NA,3,4)
> mode(x.miss)
[1] "numeric"
```

A *character* vector might be

```
> x.char=c("A","B","C","D","E")
```

Again verifying

```
> mode(x.char)
[1] "character"
```

Logical vectors are often useful:

```
> x.num <- c(5,7,8,2,9)
> x.log = x.num <=3          #Test each number to see if it is less than or equal to 3
```

```
                                    # "#" indicates a comment.
> x.log
[1] FALSE FALSE FALSE  TRUE FALSE
> mode(x.log)
[1] "logical"
```

A useful property of logical vectors is that they can be converted to numeric objects containing 1's and 0's by multiplying by 1:

```
> 1*x.log
[1] 0 0 0 1 0
```

Logical operations play an important role in R; they are often used in extracting a subset of data

```
> x = c(1, 7, 3, 5, 9, 3)
> x.sub = subset(x,x<=5)              #x.sub is a new vector with all x´s less than or eequal to 5
> x.sub
[1] 1 3 5 3
> x.sub = subset(x,x==3)              # note the == in the logical equals operation
> x.sub
[1] 3 3
```

"An Introduction to R" provides an example of a *list:*

```
> Lst <- list(name="Fred", wife="Mary", no.children=3,
+ child.ages=c(4,7,9))
```

Note that R uses "+" to indicate the continuation of a command to additional lines.

Class. In addition to having a mode, R objects have a class. The class of a vector is the same as its mode. However, this is not true for all objects. For example,

```
> y.num=c(6,7,8,9,10)
> xy=cbind(x.num,y.num)      #cbind binds the two vectors as adjacent columns
                             #rbind binds the two vectors as adjacent rows
> xy
     x.num y.num
[1,]     5     6
[2,]     7     7
[3,]     8     8
[4,]     2     9
[5,]     9    10
> mode(xy)
[1] "numeric"
> class(xy)
[1] "matrix"
```

Nmeric matrices permit all the operations available for matrices such as multiplication of matrices.

Another important class is the data frame; its mode is "list." Data frames typically arise when we read a table of data into R but they can also be coerced from other classes. For example,

```
> xy.df = as.data.frame(xy)
> xy.df
  x.num y.num
1     5     6
2     7     7
3     8     8
4     2     9
5     9    10
```

This looks very much like a matrix. One difference is that data frames can accommodate vectors of different modes. For example, we can add a character vector:

```
> xy.df2=cbind(x.char,xy.df);  xy.df2                    #There are two commands here,
                                                          # separated by semicolons
  x.char x.num y.num
1      A     5     6
2      B     7     7
3      C     8     8
4      D     2     9
5      E     9    10
```

Checking on the class and mode:

```
> mode(xy.df2)
[1] "list"
> class(xy.df2)
[1] "data.frame"
```

However, all vectors in a matrix must have the same mode. For example, if we coerce xy.df2 to be a matrix, we find that all entries are treated as characters:

```
> xy.char=as.matrix(xy.df2)
> xy.char
     x.char x.num y.num
[1,] "A"    "5"   " 6"
[2,] "B"    "7"   " 7"
[3,] "C"    "8"   " 8"
[4,] "D"    "2"   " 9"
[5,] "E"    "9"   "10"
> mode(xy.char); class(xy.char)
[1] "character"
[1] "matrix"
```

We might want to change the column names of xy.df2:

```
> colnames(xy.df2)=c("Subject","X","Y");
> xy.df2
  Subject X  Y
1       A 5  6
2       B 7  7
3       C 8  8
4       D 2  9
5       E 9 10
```

Length and Dimensions. An attribute of a vector is its length or, in the case of objects with more than one dimension, dimensionality. For example,

```
> length(x.num)
[1] 5
```

You may verify that this is true for the other vectors illustrated previously, not just for a numeric vector. Length might prove useful as part of other calculations. For example, we might write a general function for a one-sample *t* test, containing a command such as "df = length(x) - 1."

Arrays such as matrices and data frames can have many dimensions:

```
> dim(xy)
[1] 5 2
> dim(xy.df2)
[1] 5 3
```

The "info" function. We are sometimes unsure of whether we have a matrix or a data frame, or we want to check the dimensionality of a data set we have read in to R, We can do as we have so far, using "class," "mode," and "dim" separately, but the "info" function in Appendix 1A does it all at once. (As noted in the .RProfile section, this function was developed by Kellerman.)

For example,

```
> info(xy.df2)
MODE:         list
CLASS:        data.frame
DIM or LENGTH: 5 3
NAMES:        Subject X Y
```

1.6 **Indexing Objects.**

Just as we use subscripts for statistical notation, we have a parallel, but more flexible way of indicating scores, and sets of scores, in R. To indicate the third row in the data frame, xy.df2, the command is

```
> row.3 = xy.df2[3,];  row.3
  Subject X Y
3       C 8 8
> col.3=xy.df2[,3]; col.3        #Note the positions of the commas in the two examples
[1]  6  7  8  9 10
```

Now suppose we wanted to examine the data for only subjects A, B, and C:

```
> xy.df2[1:3,]          #Note how we indicate a series of rows
  Subject X Y
1       A 5 6
2       B 7 7
3       C 8 8
```

Suppose we want the data for only Subjects A, C, and E:

```
> xy.df2[c(1,3,5),]
  Subject X  Y
1       A 5  6
3       C 8  8
5       E 9 10
```

Similar operations can be carried out on the columns.

1.7 Arithmetic Operations,

We can do the usual arithmetic operations such as addition, subtraction, multiplication, and division on numeric vectors, including those within a data frame or matrix. For example,

```
> xy.df2[,2]+ xy.df2[,3]                #Sum columns 2 and 3
[1] 11 14 16 11 19
```

Operations on matrices are useful in many situations. For example, if

```
> uv=cbind(c(5,4,3,2,1),c(10,9,8,7,6))                #a 5 x 2 matrix
> colnames(uv)=c("u.num","v.num")
```

then xy*uv gives the pairwise products:

```
> cbind(xy,uv,xy*uv)          #Columns 1and 2 are xy, the next two are  uv, then xy*uv.
     x.num y.num u.num v.num x.num y.num
[1,]     5     6     5    10    25    60
[2,]     7     7     4     9    28    63
[3,]     8     8     3     8    24    64
[4,]     2     9     2     7     4    63
[5,]     9    10     1     6     9    60
```

We may multiply the transpose of uv (rows and columns are interchanged) by xy. A 5 x2 matrix post-multiplied by a 2x5 matrix yields a 5 x 5, so we have

```
> xy%*%t(uv)                  # Note that matrix multiplication is indicated by %*%
                              # Also t(uv) refers to the transpose of uv
     [,1] [,2] [,3] [,4] [,5]
[1,]   85   74   63   52   41
[2,]  105   91   77   63   49
[3,]  120  104   88   72   56
[4,]  100   89   78   67   56
[5,]  145  126  107   88   69
```

1.8 Some Useful Functions

We now have several objects in our workspace and some house cleaning is in order. First, we want to see what we have.

```
> ls()
 [1] "classx"      "col.3"       "CorelSets"   "dimx"        "info"
 [6] "keep"        "look"        "LS"          "lsize"       "Lst"
[11] "make.facets" "qs"          "row.3"       "runmx"       "uv"
[16] "val"         "x.char"      "x.log"       "x.miss"      "x.num"
[21] "xy"          "xy.char"     "xy.df"       "xy.df2"      "y.num"
```

A lot of these are functions that stem from my .RProfile and I want to keep those. It may be helpful to know which are functions and which are objects. Then we can remove the objects we don't want. Another of the Kellerman functions (see the Appendix) helps here and provides more information than the ls() function:

```
> LS()          #Note  the difference between ls and LS
DIM/LEN          NAME         CLASS
1              classx      function
5              col.3       numeric
1           CorelSets      function
1              dimx        function
1              info        function
```

```
1              keep       function
1              look       function
1                LS       function
1             lsize       function
4               Lst           list
1       make.facets       function
1                qs       function
1 3           row.3     data.frame
1            runmx       function
5 2              uv         matrix
1             val       function
5           x.char      character
5            x.log        logical
5           x.miss        numeric
5            x.num        numeric
5 2             xy         matrix
5 3         xy.char         matrix
5 2           xy.df     data.frame
5 3          xy.df2     data.frame
5            y.num        numeric
```

Rather than cluttering the workspace with objects we no longer need, we remove them:

```
>rm(col.3,Lst,row.3,uv,x.char,x.log,x.miss,x.num,xy,xy.char,xy.df,xy.df2,y.num)
```

We check to make sure we've rid the workspace of the no-longer needed objects:

```
> ls()
 [1] "classx"      "CorelSets"   "dimx"        "info"        "keep"
 [6] "look"        "LS"          "lsize"       "make.facets" "qs"
[11] "runmx"       "val"
```

1.9 Input and Output

Reading files. If you ask for help for "read.table" you will find that there are several options, only one of which is "read.table." You also find a number of arguments, many of which can be ignored with most data sets. Let's try "read.table."

```
>  MyData = read.table("C:/R_Stats/WD/example1.txt", header = TRUE, sep = " ")
```

"header=TRUE" (equivalently, "header=T) indicates that the first line of the file contains column names; in the absence of this argument, it is assumed that "header=FALSE"; i.e., the first line contains data. ' sep = " " ' indicates that columns are separated by blank space.

An alternative approach to reading data is

```
> MyData=read.csv(file.choose(),header=T,sep=",")
```

This opens Windows Explorer, allowing you to browse for a file. In this example, I am reading a file in which columns are separated by a comma "csv" indicates "comma separated variables."
It would be wise to check on MyData. The following command prints the first 6 lines:

```
> head(MyData)
  Method  Y
1 Control  5
2 Control 11
3 Control 14
4 Control 14
```

```
5 Control 15
6 Control 15
```

```
>MyDdata[1:6,]                    # Requests rows 1 through 6
```

would also yield the first 6 rows. We can also select any rows (or columns); e.g.,

```
>MyData[c(1,6),]                  # Requests rows 1 and 6
  Method  Y
1 Control  5
6 Control 15
```

"read.csv" and "read.delim" are very similar to "read.table;" the main difference is sep = "," (for csv) and sep ="\t" (for read.delim); the latter would be used when numbers are separated buy tabs.

The option I have used most frequently in my scripts is "read.csv" where csv stands for "comma separated variables." Generally, I have saved Excel or SPSS file data files in csv format and read that version into R. A quote from the cran.r-project website in response to the question of how to read an Excel file into R is appropriate here: "The first piece of advice is to avoid doing so if possible! If you have access to Excel, export the data you want from Excel in tab-delimited or comma-separated form, and use read.delim or read.csv to import it into R" In creating my files, I have exported from Excel in csv format and then used the read.csv function.

Writing Files. Writing involves options similar to those for reading. For example,

```
>write.table(MyData, "C:/R_Stats/WD/example.2.txt", quote=FALSE,
+sep=" ",col.names=TRUE)
```

and "example2.txt" is now present in the WD folder of my hard drive. "Quote=FALSE" indicates that we do not want quotation marks around the entries and "col.names =TRUE" indicates that we do want to retain the column names.

1.10 Referring to Variables

We might like to operate in some way on the variable Y in MyData; perhaps taking its log, multiplying it by another vector, or binding it to another column vector to form a new data frame. Referring to Y ordinarily doesn't work; for example,

```
> head(Y)
Error in head(Y) : object 'Y' not found
```

We have several options. The first is simply to designate the row or column we want using indexing:

```
>head(MyData[,2])
```

This also works for matrices. However, with data frames and lists we have other options that are not applicable to numeric matrices but make reference easier by using the variable name. For example,

```
> MyData$Y            #Note the form – data frame name, $, variable name
 [1]  5 11 14 14 15 15 15 15 15 17 17 17 17 18 18 18 18 18 19 19 19 19 22 22 24
[26]  9 12 16 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 20 21 21 21 27
```

This reference is explicit but can be awkward if the name of the data frame or the variable is long. We can get the preceding result by first issuing the command

```
>attach(MyData)
```

Then references to Y produce the desired result; e.g.,

```
> head(Y)
[1]  5 11 14 14 15 15
```

Although the "attach" function makes it easy to refer to a variable, you must be careful. If you have several data frames attached, confusion is possible. For example, if two data frames have a column named Y, a reference to Y is taken to mean the last one attached. In situations like this, it is often best to check which data frames are attached by

```
> search()                                          #which yields
 [1] ".GlobalEnv"        "uv"              "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:WD"  "package:methods"   "Autoloads"
[10] "package:base"
```

We might detach uv before attaching another data frame

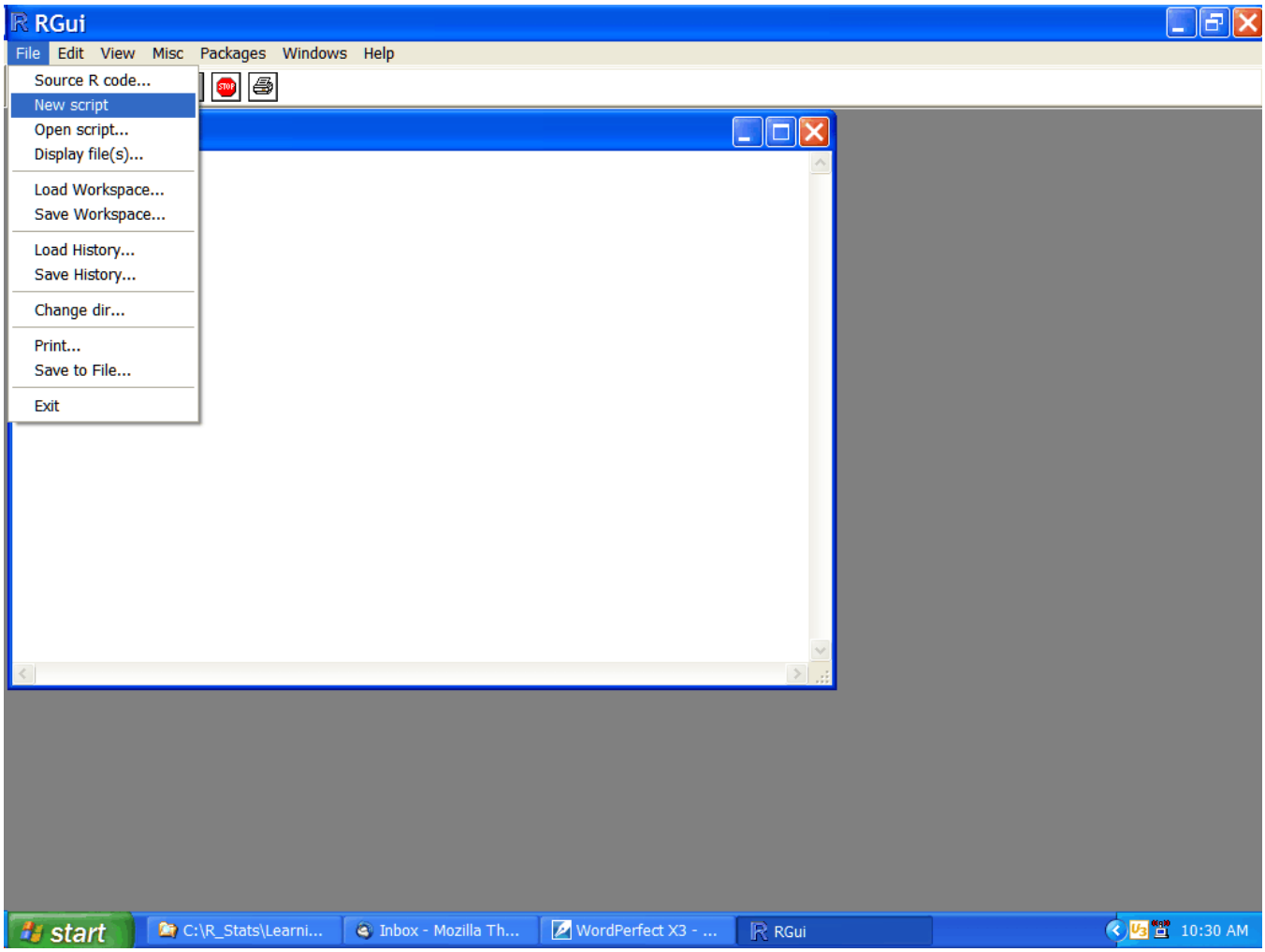
```
>detach(uv)
```

Or use either indexing or the dollar sign notation for working with variables in the new data frame.

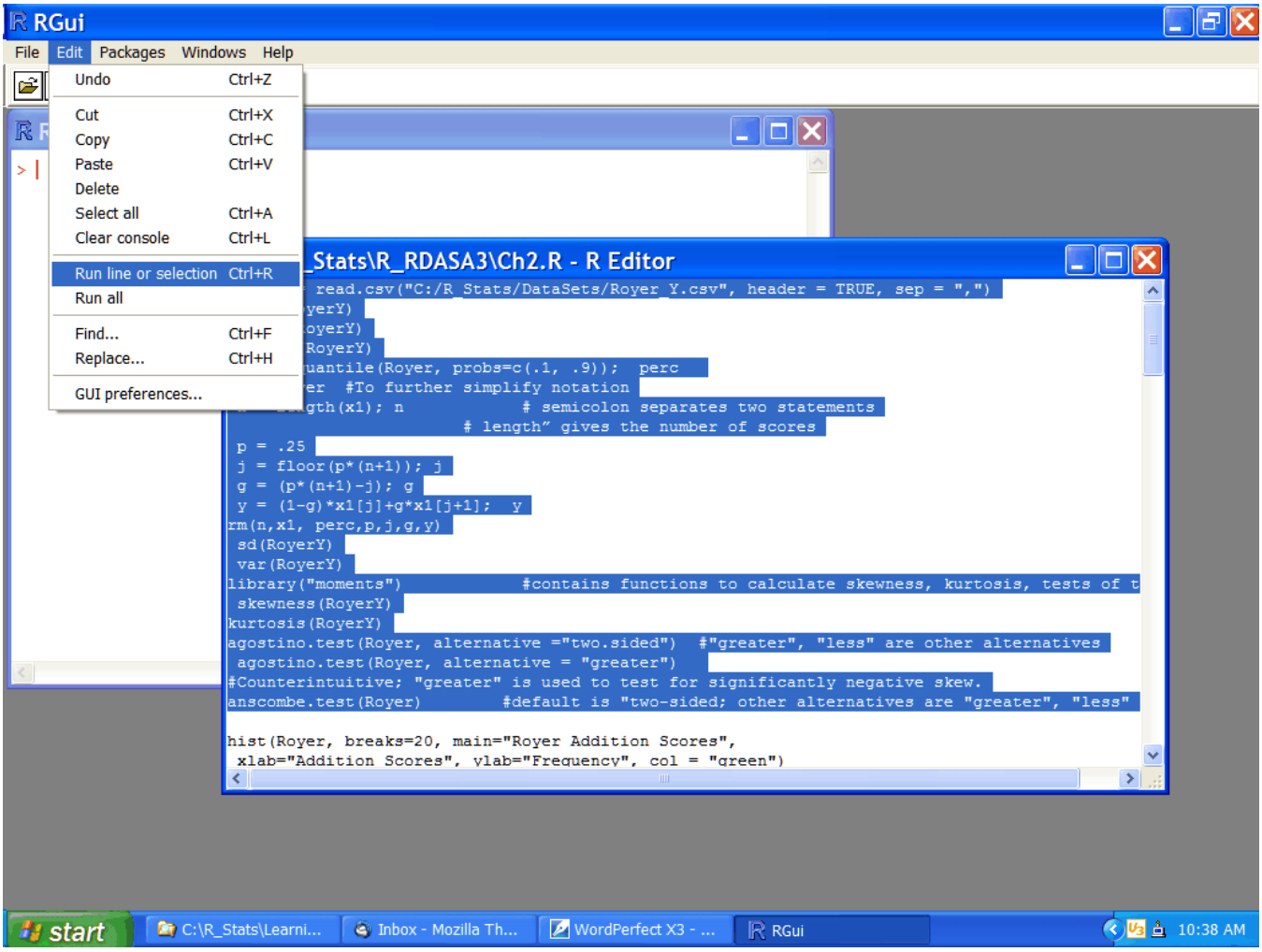
In the scripts that form the subsequent sessions, we have used all of the options for referencing variables, the choice dependent on the length of the name, the frequency of the reference, and our mood. However, when we have attached a data frame, we have been careful to detach it when we are done with referencing the variables contained in it.

1.11 Scripts

An R script is a text file with a .R extension rather than the usual .txt extension. It consists of a series of R commands. The file may be written using a text editor such as Notebook (but be careful to ensure that the extension is R, not txt) or by using the R editor. To open a saved file, or to write a new one, select the File menu from the R console‘s menu bar (see next page). The leftmost icon also accesses saved scripts. Once you have a script open, you may run part of it (after using your mouse to select a series of statements) or run the entire script. Running a script in parts is useful when debugging a script or when there are several data plots and you wish to stop (perhaps to study or copy) after each one. The function, par(ask=T) , will also pause a plot. Once a plot is stopped, clicking on the right-upper-corner x, or running the statement ‟dev.off()‟ will close the plot.



In the next figure we have a view of a screen with the Session 2 script open, and a section selected that calculates various exploratory statistics. Note that we are poised to run the selected lines although the "run all" option is also available. We can also edit the script if we find errors, or wish to add comments. When running a series of statements, an alternative to opening the edit menu is, as the figure indicates. to simultaneously press the Ctrl and R keys.



This introduction is far from complete but should provide a start. In the remaining sessions, we present and comment on commands that will provide the graphics and analyses of the data sets on my R website.

Appendix: The LS and INFO functions

```
##LS
LS<-function (space=1,pattern = "")
{

if (length(ls(space,pat=pattern)) <1){stop("No objects are in memory")}

obs <- ls(space, pat = pattern)
cat(
   formatC("DIM/LEN"),
   formatC("NAME", width=max(nchar(obs)+5)),
   formatC("CLASS",width=16),
   "\n")
```

```
if (length(ls(space,pat="tmp")) >0){

 for (i in 2:length(obs)) {
   widim <- 0
   ww <- eval(parse(t = paste("length(dimx(", obs[i], "))")))
   for (k in 1:ww){
     widim <- eval(parse(t = paste("length(dimx(",obs[i],"))+widim-2+nchar(dimx(", obs[i], ")[k])")))}

   cat(
          eval(parse(t = paste("dimx(", obs[i], ")"))),
          formatC(obs[i], width=max(nchar(obs))-widim+10),
          formatC(eval(parse(t = paste("class(", obs[i], ")"))),1, 16),
          "\n")  }}

else{
   for (i in 1:length(obs)) {
   widim <- 0
   ww <- eval(parse(t = paste("length(dimx(", obs[i], "))")))
   for (k in 1:ww){
     widim <- eval(parse(t = paste("length(dimx(",obs[i],"))+widim-2+nchar(dimx(", obs[i], ")[k])")))}

   cat(
          eval(parse(t = paste("dimx(", obs[i], ")"))),
          formatC(obs[i], width=max(nchar(obs))-widim+10),
          formatC(eval(parse(t = paste("class(", obs[i], ")"))),1, 16),
          "\n")  }}}


##INFO
info <- function(x){
cat("MODE:          ")
cat(mode(x))
cat("\n")
cat("CLASS:         ")
cat(class(x))
cat("\n")
cat("DIM or LENGTH: ")
cat(dimx(x))
cat("\n")
cat("NAMES:         ")
if (length(names(x))>6) cat(names(x)[1:6]," ...")
else cat(names(x))
cat("\n")}
```