

# Appendix 2a: Introduction to R and the EngrExpt package

## Introductory Statistics for Engineering Experimentation

Peter R. Nelson, Marie Coffin and Karen A.F. Copeland

Slides by Douglas Bates

# Outline

What is R?

Organizing data

Accessing and modifying variables

Subsets of data frames

Missing Data

# Outline

What is R?

Organizing data

Accessing and modifying variables

Subsets of data frames

Missing Data

# Outline

What is R?

Organizing data

Accessing and modifying variables

Subsets of data frames

Missing Data

# Outline

What is R?

Organizing data

Accessing and modifying variables

Subsets of data frames

Missing Data

# Outline

What is R?

Organizing data

Accessing and modifying variables

Subsets of data frames

Missing Data

# R

- *R* is an Open Source (and freely available) environment for statistical computing and graphics.
- Links for downloading R for Windows or for Mac OS X are given on the Learn@UW site for this course.
- *R* is under active development - typically two major releases per year.
- *R* provides data manipulation and display facilities and most statistical procedures. It can be extended with “packages” containing data, code and documentation. Currently there are more than 1800 contributed packages in the Comprehensive R Archive Network (CRAN). One of the packages, [EngrExpt](#), contains the data sets for the text and some sample code for plotting the data or fitting models.

## Simple calculator usage

- The R application is started by clicking on an icon or a menu item. The main window is called the console window.
- Arithmetic expressions can be typed in the console window. If the expression on a line is complete it is evaluated and the result is printed.

```
> 5 - 1 + 10
```

```
[1] 14
```

```
> 7 * 10 / 2
```

```
[1] 35
```

```
> exp(-2.19)
```

```
[1] 0.1119167
```

```
> pi
```

```
[1] 3.141593
```

```
> sin(2 * pi/3)
```

```
[1] 0.8660254
```



## Comments on the calculator usage

- The `>` symbol at the beginning of the input line is the prompt from the application, not something that is typed by the user.
- If the expression typed is incomplete, say because it contains a `(` without the corresponding `)` then the prompt changes to a `+` indicating that more input is required.
- The expression `[1]` at the beginning of the response is an index indicating that what follows is the first (and in these cases the only) element of a numeric vector.

## Assignment of values to names

- During a session, data objects can be assigned to names.
- The assignment operator is the two-character sequence `<-`. (The `=` sign can also be used, except in a few cases.)
- The function `ls` lists the names of objects; `rm` removes objects. An alternative to `ls` is `ls.str()` which lists objects in the workspace and provides a brief description of their structure.

```
> x <- 5 # assignments are not echoed
```

```
> ls()
```

```
[1] "x"
```

```
> ls.str()
```

```
x :   num 5
```

```
> rm(x)
```

```
> ls()
```

```
character(0)
```

## Vectors

- Numeric objects are always stored as vectors (as opposed to scalars).
- An easy way to create a non-trivial vector is a sequence, generated by the `:` operator or the `seq` function.
- When results are printed the number in square brackets at the beginning of the line is the index of the element at the start of the line.
- Square brackets are used to specify indices (or, in general, subsets).

```
> (x <- 0:19) # the redundant parens cause printing of x
```

```
[1] 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17  
[19] 18 19
```

```
> x[5]
```

```
[1] 4
```

```
> str(y <- x + runif(20, min = 10, max = 20))
```

```
num [1:20] 17 12.4 19.9 18.1 16 ...
```

## Following the operations on the slides

- The lines of *R* code shown on these slides are available in files on the course web site. The file for this section is called `ch02a.R`.
- If you open this file in the *R* application (the **File**→**Open** menu item or `<ctrl>-O`) and position the cursor at a particular line, then `<ctrl>-R` will send the line to the console window for execution and step to the next line.
- Any part of a line following a `#` symbol is a comment.
- The code is divided into named “chunks”, typically one chunk per slide that contains code.
- In the system called *Sweave* used to generate the slides the result of a call to a graphics function must be `printed` or `shown`. In interactive use this is not necessary but neither is it harmful.
- Note that *R* provides *name completion* with the `<tab>` key. After typing part of a name you can use tab to request completion.

## Organizing data in R

- Standard rectangular data sets (columns are variables, rows are observations) are stored in *R* as *data frames*.
- The columns can be *numeric* variables (e.g. measurements or counts) or *factor* variables (categorical data) or *ordered* factor variables. These types are called the *class* of the variable.
- The `str` function provides a concise description of the structure of a data set (or any other class of object in R). The `summary` function summarizes each variable according to its class. Both are highly recommended for routine use.
- Entering just the name of the data frame causes it to be printed. For large data frames use the `head` and `tail` functions to view the first few or last few rows.

# The EngrExpt package

- The data sets described in Appendix A of the text are available as *R* data frames in the `EngrExpt` package.
- You must install the package on your computer with  

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

This need only be done once.
- Each time you start *R*, attach the package using  

```
> library(EngrExpt)
```
- When a data set is mentioned in the book as, say, in a file called `odor.txt` you can access the data as a data frame with the same name.  

```
> str(odor)
```

```
'data.frame': 35 obs. of 3 variables:  
 $ odor      : num  4.54 3.41 4.41 2.82 4.71 ...  
 $ yellowing: num  2.36 3.46 1.45 3.08 4.28 3.1 2.08 1.74 3.07 0.98 ...  
 $ hardness  : num  82.6 79.1 87.5 86.5 78.8 ...
```

## Data input

- In this course access to the data sets in the `Engrexpt` package should suffice. However, you may want to use `R` for analysis of other data.
- The simplest way to input a rectangular data set is to save it as a comma-separated value (csv) file and read it with `read.csv`.
- The first argument to `read.csv` is the name of the file. On Windows it can be tricky to get the file path correct (backslashes need to be doubled). The best approach is to use the function `file.choose` which brings up a “chooser” panel through which you can select a particular file. The idiom to remember is

```
> mydata <- read.csv(file.choose())
```

for comma-separated value files or

```
> mydata <- read.delim(file.choose())
```

for files with tab-delimited data fields.
- With an Internet connection you can use a URL (within

## In-built data sets

- One of the packages attached by default to an *R* session is the `datasets` package that contains several data sets culled primarily from introductory statistics texts.
- We will use some of these data sets for illustration.
- The `Formaldehyde` data are from a calibration experiment, `Insectsprays` are from an experiment on the effectiveness of insecticides.
- Use `?` followed by the name of a function or data set to view its documentation. If the documentation contains an example section, you can execute it with the `example` function.



## The Formaldehyde data

```
> str(Formaldehyde)
```

```
'data.frame': 6 obs. of 2 variables:
```

```
$ carb : num 0.1 0.3 0.5 0.6 0.7 0.9
```

```
$ optden: num 0.086 0.269 0.446 0.538 0.626 0.782
```

```
> summary(Formaldehyde)
```

carb	optden
Min. :0.1000	Min. :0.0860
1st Qu.:0.3500	1st Qu.:0.3132
Median :0.5500	Median :0.4920
Mean :0.5167	Mean :0.4578
3rd Qu.:0.6750	3rd Qu.:0.6040
Max. :0.9000	Max. :0.7820

```
> Formaldehyde
```

	carb	optden
1	0.1	0.086
2	0.3	0.269
3	0.5	0.446
4	0.6	0.538
5	0.7	0.626

## The InsectSprays data

```
> str(InsectSprays)
```

```
'data.frame': 72 obs. of 2 variables:
```

```
$ count: num 10 7 20 14 14 12 10 23 17 20 ...
```

```
$ spray: Factor w/ 6 levels "A","B","C","D",...: 1 1 1 1 1 1 1 1 1 1 ..
```

```
> summary(InsectSprays)
```

	count	spray
Min.	: 0.00	A:12
1st Qu.:	3.00	B:12
Median	: 7.00	C:12
Mean	: 9.50	D:12
3rd Qu.:	14.25	E:12
Max.	:26.00	F:12

```
> head(InsectSprays)
```

	count	spray
1	10	A
2	7	A
3	20	A
4	14	A
5	14	A

# Copying, saving and restoring data objects

- Assigning a data object to a new name creates a copy.
- You can save a data object to a file, typically with the extension `.rda`, using the `save` function.
- To restore the object you `load` the file.

```
> sprays <- InsectSprays  
> save(sprays, file = "sprays.rda")  
> rm(sprays)  
> ls.str()
```

```
x : int [1:20] 0 1 2 3 4 5 6 7 8 9 ...  
y : num [1:20] 17 12.4 19.9 18.1 16 ...
```

```
> load("sprays.rda")  
> names(sprays)
```

```
[1] "count" "spray"
```

## Accessing and modifying variables

- The `$` operator is used to access variables within a data frame.

```
> Formaldehyde$carb
```

```
[1] 0.1 0.3 0.5 0.6 0.7 0.9
```

- You can also use `$` to assign to a variable name

```
> sprays$sqrtcount <- sqrt(sprays$count)
```

```
> names(sprays)
```

```
[1] "count"      "spray"      "sqrtcount"
```

- Assigning the special value `NULL` to the name of a variable removes it.

```
> sprays$sqrtcount <- NULL
```

```
> names(sprays)
```

```
[1] "count" "spray"
```

## Using with and within

- In complex expressions it can become tedious to repeatedly type the name of the data frame.
- The `with` function allows for direct access to variable names within an expression. It provides “read-only” access.

```
> Formaldehyde$carb * Formaldehyde$optden
```

```
[1] 0.0086 0.0807 0.2230 0.3228 0.4382 0.7038
```

```
> with(Formaldehyde, carb * optden)
```

```
[1] 0.0086 0.0807 0.2230 0.3228 0.4382 0.7038
```

- The `within` function provides read-write access to a data frame. It does not change the original frame; it returns a modified copy. To change the stored object you must assign the result to the name.

```
> sprays <- within(sprays, sqrtcount <- sqrt(count))
```

```
> str(sprays)
```

```
'data.frame': 72 obs. of 3 variables:
```

```
$ count      : num  10 7 20 14 14 12 10 23 17 20 ...
```

```
$ spray      : Factor w/ 6 levels "A","B","C","D",...: 1 1 1 1 1 1 1 1 1 1
```

```
$ sqrtcount: num  3.16 2.65 4.47 3.74 3.74 ...
```

# Data Organization

- Careful consideration of the data layout for experimental or observational data is repaid in later ease of analysis. Sadly, the widespread use of spreadsheets does not encourage such careful consideration.
- If you are organizing data in a table, use consistent data types within columns. Databases require this; spreadsheets don't.
- A common practice in some disciplines is to convert categorical data to 0/1 “indicator variables” or to code the levels as numbers with a separate “data key”. This practice is unnecessary and error-inducing in *R*. When you see categorical variables coded as numeric variables, change them to factors or ordered factors.

## Converting numeric variables to factors

- The `factor` (`ordered`) function creates a factor (ordered factor) from a vector. Factor labels can be specified in the optional `labels` argument.
- Suppose the `spray` variable in the `InsectSprays` data was stored as numeric values  $1, 2, \dots, 6$ . We convert it back to a factor with `factor`.

```
> str(sprays <- within(InsectSprays, spray <- as.integer(spray)))
```

```
'data.frame': 72 obs. of 2 variables:
```

```
$ count: num 10 7 20 14 14 12 10 23 17 20 ...
```

```
$ spray: int 1 1 1 1 1 1 1 1 1 1 ...
```

```
> str(sprays <- within(sprays, spray <- factor(spray, labels = L
```

```
'data.frame': 72 obs. of 2 variables:
```

```
$ count: num 10 7 20 14 14 12 10 23 17 20 ...
```

```
$ spray: Factor w/ 6 levels "A","B","C","D",...: 1 1 1 1 1 1 1 1 1 1 ..
```

## Subsets of data frames

- The `subset` function is used to extract a subset of the rows or of the columns or of both from a data frame.
- The first argument is the name of the data frame. The second is an expression indicating which rows are to be selected.

- This expression often uses logical operators such as `==`, the equality comparison, or `!=`, the inequality comparison, `>=`, meaning “greater than or equal to”, etc.

```
> str(sprayA <- subset(sprays, spray == "A"))
```

```
'data.frame': 12 obs. of 2 variables:
```

```
$ count: num 10 7 20 14 14 12 10 23 17 20 ...
```

```
$ spray: Factor w/ 6 levels "A","B","C","D",...: 1 1 1 1 1 1 1 1 1 1
```

- The optional argument `select` can be used to specify the variables to be included.



## Subsets and factors

- The way that factors are defined, a subset of a factor retains the original set of levels. Usually this is harmless but sometimes it can cause unexpected results.
- You can “drop unused levels” by applying `factor` to the factor. Many functions, such as `xtabs`, which is used to create cross-tabulations, have optional arguments with names like `drop.unused.levels` to automate this.

```
> xtabs( ~ spray, sprayA)
```

```
spray
```

```
  A  B  C  D  E  F  
12  0  0  0  0  0
```

```
> xtabs( ~ spray, sprayA, drop = TRUE)
```

```
spray
```

```
  A  
12
```

## Dropping unused levels in the spray factor and %in%

```
> str(sprayA <- within(sprayA, spray <- factor(spray)))
```

```
'data.frame': 12 obs. of 2 variables:
```

```
$ count: num 10 7 20 14 14 12 10 23 17 20 ...
```

```
$ spray: Factor w/ 1 level "A": 1 1 1 1 1 1 1 1 1 1 ...
```

```
> xtabs( ~ spray, sprayA)
```

```
spray
```

```
A
```

```
12
```

- Another useful comparison operator is %in% for selecting a subset of the values in a variable.

```
> str(sprayDEF <- subset(sprays, spray %in% c("D","E","F")))
```

```
'data.frame': 36 obs. of 2 variables:
```

```
$ count: num 3 5 12 6 4 3 5 5 5 5 ...
```

```
$ spray: Factor w/ 6 levels "A","B","C","D",...: 4 4 4 4 4 4 4 4 4 4 ..
```

## Working with NA's

- Missing data are represented in *R* as `NA`.
- The important thing to know about `NA`s is that they propagate. Except for certain detector functions (`is.na`, `is.finite`) any operation on an `NA` returns an `NA`.

```
> (x <- c(10, 20, NA, 4, NA, 2))
```

```
[1] 10 20 NA 4 NA 2
```

```
> sum(x)
```

```
[1] NA
```

```
> is.na(x)
```

```
[1] FALSE FALSE TRUE FALSE TRUE FALSE
```

```
> sum(x[!is.na(x)])
```

```
[1] 36
```

## The na.rm argument in summary functions

- Many summary functions have an optional argument `na.rm`. When `TRUE` the function is applied to the non-missing data only.
- The `na.rm` argument does remove `NaNs` but does not remove `Infs` or `-Infs`.

```
> sum(x, na.rm = TRUE)
```

```
[1] 36
```

```
> (x <- c(0, 1, NA, 0/0, Inf))
```

```
[1] 0 1 NA NaN Inf
```

```
> sum(x, na.rm = TRUE)
```

```
[1] Inf
```

## Saving workspaces

- When you exit an R session (the function call `q()`, including the parentheses, or the menu option or `<ctrl>-Q`) you are given the option of saving the workspace.
- If you do save it, the workspace will be restored when you start *R* again in that directory.
- Different people have different approaches to saving workspaces. I never do it. I always save the data objects that I may need later and/or the code that generates them. Frequently on a large project I just have one copy of the data set and many, many scripts to manipulate it.
- For me the problem with saving and restoring workspaces is that I'm not starting from a known and reproducible state in a new session.

## Summary of data input and manipulation

- Rectangular data sets are stored in data frames. Typically columns are numeric vectors or factors but other types (Dates, date/times) are possible.
- The easiest input functions are `read.csv` and `read.delim`.
- Use `str` and `summary` frequently.
- Use `subset` for extracting parts of a data frame, `with` for read-only access to variables and `within` for read-write access.
- Store categorical variables as factors.