# Week 1

**What is R?**

R is a dialect of the S language.

**What is S?**

S is a language developed by John Chambers and others at Bell Labs.

S was initiated in 1976 as an internal statistical analysis environment - originally implemented as Fortran libraries.

Early versions of the language did not contain functions for statistical modeling.

In 1988 the system was rewritten in C and began to resemble the system we have today (version 3 of the language). The book *Statistical Models in S* by Chambers & Hastie (the white book) documents the statistical analysis functionality.

Version 4 of the S language was released in 1998 and is the version we use today. The book *Programming with Data* by John Chambers (the green book) documents this version of the language.

**Back to R**

- 1991: created in New Zealand by Ross Ihaka and Robert Gentleman. Their experience developing R is documented in a 1996 JCGS paper.
- 1993: first announcement of R to the public.
- 1995: Martin Mächler convinces Ross and Robert to use the GNU General Public License to make R free software.
- 1996: a public mailing list is created (R-help and R-devel).
- 1997: the R Core Group is formed (containing some people associated with S-PLUS). The core group controls the source code for R.
- 2000: R version 1.0.0 is released.
- 2013: R version 3.0.2 is released on December 2013.

**Features of R**

- Syntax is very similar to S, making it easy for S-PLUS users to switch over.
- Semantics are superficially similar to S, but in reality are quite different (more on that later).
- Runs on almost any standard computing platform/OS (even on the PlayStation 3).
- Frequent releases (annual + bugfix releases); active development.
- Quite lean, as far as software goes; functionality is divided into modular packages.
- Graphics capabilities very sophisticated and better than most stat packages.
- Useful for interactive work, but contains a powerful programming language for developing new tools (user -> programmer).
- Very active and vibrant user community; R-help and R-devel mailing lists and Stack Overflow.

**Free Software**

With free software, you are granted:

- The freedom to run the program, for any purpose (freedom 0).
- The freedom to study how the program works, and adapt it to your needs (freedom 1). Access to the source code is a precondition for this.

- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

**Drawbacks of R**

Essentially based on 40 year old technology.

Little built-in support for dynamic or 3-D graphics.

Functionality is based on consumer demand and user contributions. If no one feels like implementing your favorite method, then it's your job!

Objects must generally be stored in physical memory; but there have been advancements to deal with this too.

Not ideal for all possible situations (but this is a drawback of all software packages).

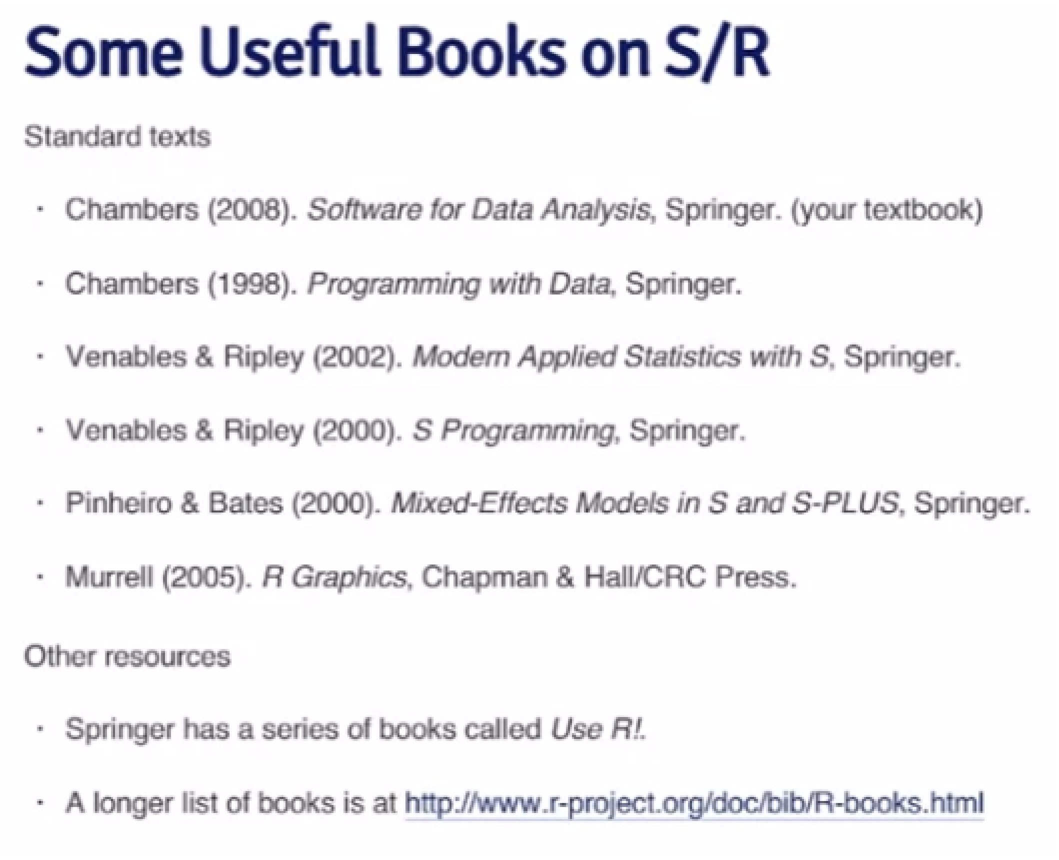
**Design of the R system**

The R system is divided into 2 conceptual parts:

1. The "base" R system that you download from CRAN.
2. Everything else.

R functionality is divided into a number of packages.

- The "base" R system contains, among other things, the `base` package which is required to run R and contains the most fundamental functions.
- The other packages contained in the "base" system include `utils`, `stats`, `datasets`, `graphics`, `grDevices`, `grid`, `methods`, `tools`, `parallel`, `compiler`, `splines`, `tcltk`, `stats4`.
- There are also "recommended" packages: `boot`, `class`, `cluster`, `codetools`, `foreign`, `KernSmooth`, `lattice`, `mgcv`, `nlme`, `rpart`, `survival`, `MASS`, `spatial`, `nnet`, `Matrix`.



## Some Useful Books on S/R

Standard texts

- Chambers (2008). *Software for Data Analysis*, Springer. (your textbook)
- Chambers (1998). *Programming with Data*, Springer.
- Venables & Ripley (2002). *Modern Applied Statistics with S*, Springer.
- Venables & Ripley (2000). *S Programming*, Springer.
- Pinheiro & Bates (2000). *Mixed-Effects Models in S and S-PLUS*, Springer.
- Murrell (2005). *R Graphics*, Chapman & Hall/CRC Press.

Other resources

- Springer has a series of books called *Use R!*.
- A longer list of books is at http://www.r-project.org/doc/bib/R-books.html

### Entering input

At the R prompt we type expression. The `<-` symbol is the assignment operator.

```
x <- 1
print(x)
msg <- "hello"
```

The grammar of the language determines whether an expression is complete or not.

```
x <- ## Incomplete expression
```

The `#` character indicates a comment. Anything to the right of the `#` (including the `#` itself) is ignored.

### Evaluation

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be auto-printed.

```
x <- 5   # nothing is printed
x        # auto-printing occurs
print(x) # explicit printing
```

### Printing

```
x <- 1:20
x
```

The `:` operator is used to create integer sequences.

### Character

R has five basic or "atomic" classes of objects:

- character;
- numeric (real numbers);
- integer;
- complex;
- logical (True/False).

The most basic object is a vector.

- A vector can only contain objects of the same class;
- BUT: The one exception is a *list*, which is represented as a vector but can contain objects of different classes (indeed, that's usually why we use them).

Empty vectors can be created with the `vector()` function.

### Numbers

Numbers in R are generally treated as numeric objects (i. e. double precision real numbers).

If you explicitly want an integer, you need to specify the `L` suffix.

- Ex.: Entering `1` gives you a numeric object; entering `1L` explicitly gives you an integer.

There is also a special number `Inf` which represents infinity.

- E. g.: `1/0` will produce `Inf`.
- `Inf` can be used in ordinary calculations: `1/Inf` = `0`.

The value `NaN` represents an undefined value ("not a number").

- E. g.: `0/0` = `NaN`.

### Attributes

R objects can have attributes.

- `names`, `dimnames`;
- `dimensions` (e. g. matrices, arrays);
- `class`;
- `length`;
- other user-defined attributes/metadata.

Attributes of an object can be accessed using the `attributes()` function.

### Creating vectors

The `c()` function can be used to create vectors of objects.

```
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
```

Using the `vector()` function.

```
x <- vector("numeric", length = 10) # ten zeros
```

### Mixing objects

What about the following?

```
y <- c(1.7, "a")  # character
y <- c(TRUE, 2)   # numeric
y <- c("a", TRUE) # character
```

When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.

### Explicit coercion

Objects can be explicitly coerced from one class to another using the `as.*` functions, if available.

```
x <- 0:6

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

Nonsensical coercion results in `NA`s.

```
x <- c("a","b","c")

as.numeric(x) # [1] NA NA NA
# Warning message: NAs introduced by coercion

as.logical(x) # [1] NA NA NA
```

### Lists

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well.

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

### Matrices

Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (`nrow`, `ncol`).

```
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
```

Matrices are constructed *column-wise*, so entries can be thought of starting in the upper left corner and running down the columns.

```
m <- matrix(1:6, nrow = 2, ncol = 3)
m
#     [,1] [,2] [,3]
# [1,]   1    3    5
# [2,]   2    4    6
```

Matrices can also be created directly from vectors by adding a dimension attribute.

```
m <- 1:10
m
# [1] 1 2 3 4 5 6 7 8 9 10
dim(m) <- c(2,5)
m
#      [,1] [,2] [,3] [,4] [,5]
# [1,]    1    3    5    7    9
# [2,]    2    4    6    8   10
```

### `cbind`-ing and `rbind`-ing

Matrices can be created by *column-binding* or *row-binding* with `cbind()` and `rbind()`.

```
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
```

### Factors

Factors are used to represent categorical data. Factors can be unordered or ordered. One can think of a factor as an integer where each integer has a label.

- Factors are treated specially by modeling functions like `lm()` and `glm()`.
- Using factors with labels is better than using integers because factors are self-describing; having a variable that has values "Male" and "Female" is better than a variable that has values 1 and 2.

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

table(x)
# x
# no yes
#  2   3

unclass(x)
# [1] 2 2 1 2 1

attr(,"levels")
[1] "no" "yes"
```

The order of the levels can be set using the `levels` argument to `factor()`. This can be important in linear modeling because the first level is used as the baseline level.

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

### Missing values

Missing values are denoted by `NA` or `NaN` for undefined 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, so there are integer `NA`, character `NA` etc;
- A `NaN` values is also `NA` but the converse is not true.

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

x <- c(1, 2, NaN, NA, 4)
is.na(x)
# [1] FALSE FALSE TRUE TRUE FALSE
is.nan(x)
# [1] FALSE FALSE TRUE FALSE FALSE
```

### Data Frames

Data frames are used to store tabular data.

- They are represented as a special type of list where every element of the list has to have the same length.
- Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.
- Unlike matrices, data frames can store different classes of objects in each column (just like lists); matrices must have every element be the same class.
- Data frames also have a special attribute called `row.names`.
- Data frames are usually created by calling `read.table()` or `read.csv()`.
- Can be converted to a matrix by calling `data.matrix()`.

```
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
```

### Names

R objects can also have names, which is very useful for writing readable code and self-describing objects.

```
x <- 1:3
names(x)
# NULL
names(x) <- c("foo", "bar", "norf")
x
#  foo  bar norf
#    1    2    3
names(x)
# [1] "foo" "bar" "norf"
```

Lists can also have names.

```
x <- list(a = 1, b = 2, c = 3)
x
# $a
# [1] 1
#
# $b
# [1] 2
#
# $c
# [1] 3
```

And matrices.

```
m <- matrix(1:4, nrow = 2, ncol = 2)
dimnames(m) <- list(c("a", "b"),c("c", "d"))
m
#   c d
# a 1 3
# b 2 4
```

### Reading data

There are a few principal functions reading data into R.

- `read.table`, `read.csv`, for reading tabular data;
- `readLines`, for reading lines of a text file;
- `source`, for reading in R code files (inverse of `dump`);
- `dget`, for reading in R code files (inverse of `dput`);
- `load`, for reading in saved workspaces;
- `unserialize`, for reading single R objects in binary form.

### Writing data

There are analogous functions for writing data to files: `write.table`, `writeLines`, `dump`, `dput`, `save`, `serialize`.

### Reading data files with `read.table`

Arguments:

- `file`: the name of a file, or a connection.

- `header`: logical indicating if the file has a header line.
- `sep`: a string indicating how the columns are separated.
- `colClasses`: a character vector indicating the class of each column in the dataset.
- `nrows`: the number of rows in the data set.
- `comment.char`: a character string indicating the comment character.
- `skip`: the number of lines to skip from the beginning.
- `stringsAsFactors`: should character variables be coded as factors?

`read.table`

For small to moderately sized data sets, you can usually call `read.table` without specifying any other arguments.

```
data <- read.table("foo.txt")
```

R will automatically:

- skip lines that begin with #;
- figure out how many rows there are (and how much memory needs to be allocated);
- figure what type of variable is in each column of the table;

`read.csv` is identical to `read.table`, except that the default separator is a comma.

#### Reading in larger data sets with `read.table`

With much larger data sets, doing the following things will make your life easier and will prevent R from choking.

- Read the help page for `read.table`, which contains many hints.
- Make a rough calculation of the memory required to store your data set. If the data set is larger than the amount of RAM on your computer, you can probably stop right here.
- Set `comment.char = " "` if there no commented lines in your file.

- Use the `colClasses` argument. Specifying this option instead of using the default can make `read.table` run MUCH faster, often twice as fast.
    - In order to use this option, you have to know the class of each column in your data frame. If all of the columns are `numeric`, for example, then you can just set `colClasses = "numeric"`.
    - A quick and dirty way to figure out the classes of each column is the following:

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)
tabAll <- read.table("datatable.txt",
                     colClasses = classes)
```

- Set `nrows`. This doesn't make R run faster but it helps with memory usage. A mild overestimate is okay. You can use the Unix tool `wc` to calculate the number of lines in a file.

#### Textual formats

- `dump`-ing and `dput`-ting are useful because the resulting textual format is edit-able, and in the case of corruption, potentially recoverable.
- Unlike writing out a table or csv file, `dump` and `dput` preserve the *metadata* (sacrificing some readability), so that another user doesn't have to specify it all over again.

- Textual formats can work much better with version control programs like subversion or git which can only track changes meaningfully in text files.
- Textual formats can be longer-lived; 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 de-parsing the R objects with `dput` and reading it back in using `dget`.

```
y <- data.frame(a = 1, b= "a")
dput(y)
# structure(list(a = 1,
#                b = structure(1L, .Label="a",
#                              class = "factor")),
#           .Names = c("a", "b"), row.names = c(NA, -1L),
#           class = "data.frame"

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

### `dump`-ing R objects

Multiple objects can be de-parsed using the `dump` function and read back in 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
# [1] "foo"
```

### Interfaces to the outside world

Data are read in using *connection* interfaces. Connections can be made to files (most common) or to other more exotic things.

- `file` opens a connection to a file.
- `gzfile` opens a connection to a file compressed with gzip.
- `bzfile` opens a connection to a file compressed with bzip2.
- `url` opens a connection to a web page.

### Subsetting

There are a number of operators that can bu used to extract subsets of R objects.

- `[` always returns an object of the same 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 only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.
- `$` is used to extract elements of a list or data frame by name; semantics are similar to that of `[[`.

```
x <- c("a", "b", "c", "c", "d", "a")
x[1]
# [1] "a"
x[2]
# [1] "b"
x[1:4]
# [1] "a" "b" "c" "c"
x[x > "a"]
# [1] "b" "c" "c" "d"
u <- x > "a"
u
# [1] FALSE TRUE TRUE TRUE TRUE FALSE
x[u]
# [1] "b" "c" "c" "d"
```

**Subsetting lists**

```
x <- list(foo = 1:4, bar = 0.6, baz="hello")
x[1]
# $foo
# [1] 1 2 3 4

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

x$bar
# [1] 0.6

x[["bar"]]
# [1] 0.6

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

x[c(1,3)]
# $foo
# [1] 1 2 3 4
#
# $baz
# [1] "hello"
```

The `[[` operator can be used with *computed* indices; `$` can only be used with literal names.

```
name <- "foo"
x[[name]]
# [1] 1 2 3 4
x$name
# NULL
x$foo
# [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
x[[1]][[3]]
# [1] 14

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

### Subsetting a matrix

Matrices can be subsetted in the usual way with $(i, j)$ type indices.

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

Indices 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 than a $1 \times 1$ matrix. This behavior can be turned off by setting `drop = FALSE`.

```
x <- matrix(1:6, 2, 3)
x[1, 2]
# [1] 3
x[1, 2, drop = FALSE]
#      [,1]
# [1,] 3
```

Similarly, subestting a single column or a single row will give you a vector, not a matrix (by default).

```
x[1, ]
# [1] 1 3 5
x[1, drop = FALSE]
#      [,1] [,2] [,3]
# [1,]    1    3    5
```

### Partial matching

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

```
x <- list(aardvark = 1:5)
x$a
# [1] 1 2 3 4 5
x[["a"]]
# NULL
x[["a", exact = FALSE]]
# [1] 1 2 3 4 5
```

### Removing `NA` values

A common task is to remove missing values (`NA`s).

```
x <- c(1, 2, NA, 4, NA, 5)
bad <- is.na(x)
x[!bad]
# [1] 1 2 4 5
```

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

```
y <- c("a", "b", NA, "d", NA, "f")
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"
```

Example: air quality data set (more here).

```
airquality[1:6, ]
# ...
good <- complete.cases(airquality)
airquality[good, ][1:6, ]
# ...
```

### Vectorized operations

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

```r
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 0.44444444
```

### Vectorized matrix operations

```r
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
```