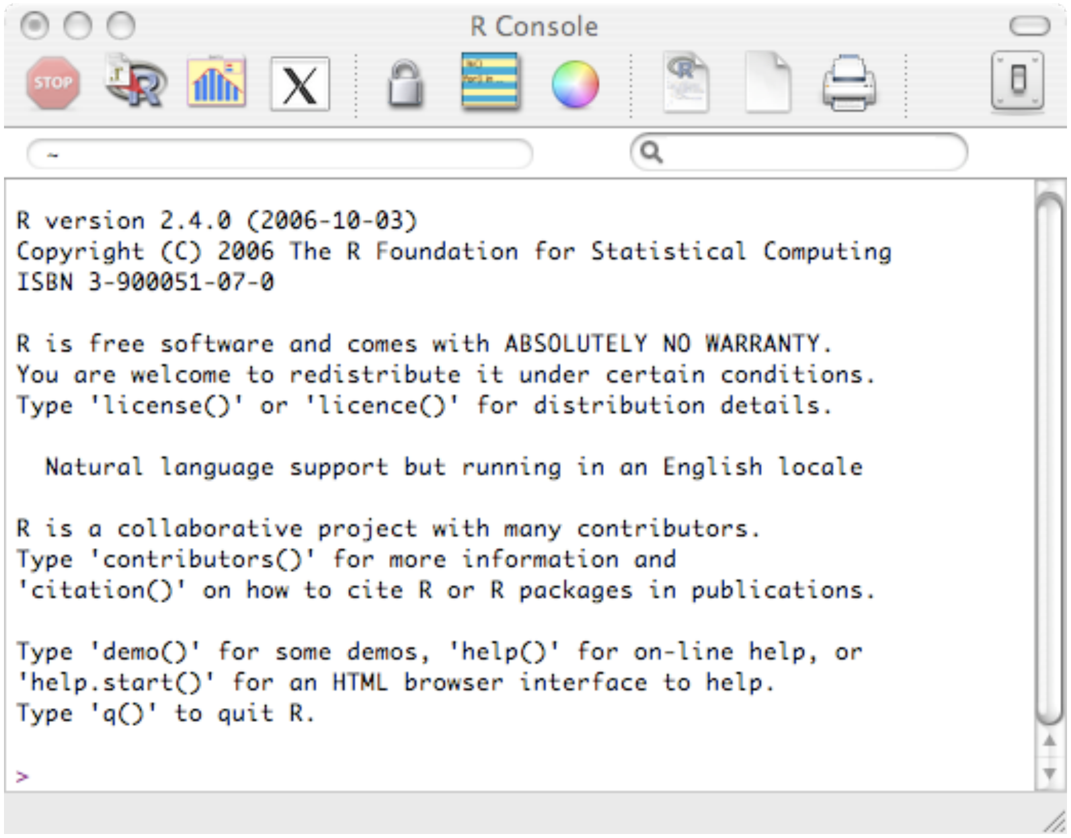
Worksheet: First steps with the R statistics package

Analyzing Linguistic Data
Spring 2012
Colin Bannard and Katrin Erk

Starting R

To download R, please see the Links page, `http://analyzing-linguistic-data.utcompling.com/home/links`.

When you start R, you should see something similar to this:



```
R version 2.4.0 (2006-10-03)
Copyright (C) 2006 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

This is a *shell*: a place where you type in commands and see the results. You always type your next command at the bottom of the shell where you see the line that contains only >.

Using R as a calculator

You can type arithmetic expressions into the R shell.

(In the following, lines starting with > are what you type in, and the other lines are what the system will answer.)

```
> 1+3
[1] 4
> 10 * 2
[1] 20
> 10 ** 2
[1] 100
```

R has many functions, either built in or via packages that you can load. Notation is like for mathematical functions. Here is an example:

```
> sqrt(3)
[1] 1.732051
```

Variables and assignment

To store data, R uses *variables*. You can use them, for example, to store the result of arithmetic expressions:

```
> myvar = 4
> myvar
[1] 4
```

This assigns a value of 4 to `myvar`, then queries R about the current contents of `myvar`. You choose the names of the variables that you want to use. Variable names can consist of the letters A-Z, a-z, 0-9, the underscore, and the fullstop, but must not begin with a number.

Instead of saying

```
> myvar = 4
```

we could equivalently have written

```
> myvar <- 4
```

or

```
> 4 -> myvar
```

Vectors

Often we need to manipulate not a single value but a list of values, for example suppose we have collected reaction times from multiple annotators:

0.2 0.3 0.2 0.4 0.15

In R, this can be expressed as a *vector*. We create a vector of the reaction times above and store them in a variable:

```
> reaction.times = c(0.2, 0.3, 0.2, 0.4, 0.15)
> reaction.times
[1] 0.20 0.30 0.20 0.40 0.15
```

Some R functions work directly on vectors:

```
> mean(reaction.times)
[1] 0.25
```

which is the same as

```
> mean(c(0.2, 0.3, 0.2, 0.4, 0.15))
[1] 0.25
```

The builtin function `summary` sums up the contents of the vector:

```
> summary(reaction.times)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.15    0.20    0.20    0.25    0.30    0.40
```

It shows the minimum, maximum, first and third quartile, median, and mean.

We can make vectors of strings (texts) in the same manner as vectors of numbers:

```
> my.strings = c("hello", "world", "!")
> my.strings
[1] "hello" "world" "!"
```

Is it possible to make a mixed vector of numbers and strings? Try this out:

```
> test.me = c('a', 1, 'b')
```

What is the result of this?

```
> c(1,2,3) + c(4,5,6)
```

Indexing individual elements of a vector

Often when you have a vector, you want to access a particular element, for example the reaction time of the second participant in `reaction.times`. In R you can do this by using square brackets.

```
> reaction.times[2]
[1] 0.3
```

This is the name of the vector, then the number of the participant in square brackets. (To repeat: brackets in a function like `mean()` are round, as are the brackets in `c()`, which makes a vector. To access a member of a vector, you use square brackets.)

But there are more things that you can do with square brackets in R. Try this:

```
> x <- c(23, 5172, 48, 202, 5, 90)
> x[x>50]
```

What does that give you? What else can you put in the straight brackets indexing a vector? Experiment with a few conditions. Then try this:

```
> x[c(1,3,5)]
```

What is the result? In general, what do you get when you index a vector by another vector in this way? How about this:

```
x[-c(1,3,5)]
```

Sequences

Often we need a vector that has a very regular shape, for example the numbers from 1 to 5. Instead of typing

```
> y = c(1,2,3,4,5)
```

we can use a useful shortcut:

```
> y = 1:5
> y
[1] 1 2 3 4 5
```

y is a vector. If we need a more involved sequence, for example values between 1 and 5, but in increments of 0.2, we can use the R function `seq`:

```
> seq(1,5, 0.2)
```

What do you get for `seq(4, 11, 2)`? What is `seq(5, 29, 8)`?

What happens if you index a vector with a range?

```
> x = c(345, 6921, 602, 1032, 10, 13)
> x[3:5]
```

Data frames

The central data structure in R is the *data frame*, which is basically a table. You specify it one column at a time, giving names to the columns. Each row corresponds to one data point, for example a participant. For example, here are three people with their heights and weights:

```
> mydf = data.frame(weights = c(123, 157, 199),
  heights = c(5.9, 6.1, 6.1))
> mydf
  weights heights
1     123     5.9
2     157     6.1
3     199     6.1
```

So while a vector is created using the builtin function `c()`, a data frame is created using the function `data.frame`.

A data frame consists of several vectors of equal length that describe different properties of the same items. That is, for the data frame `mydf` that we entered, R assumes that we had one person with weight 123 and height 5.9, one with weight 157 and height 6.1, and one with weight 199 and height 6.1.

Here is what `summary` does for this data frame:

```
> summary(mydf)
    weights          heights
 Min.   :123.0   Min.   :5.900
 1st Qu.:140.0   1st Qu.:6.000
 Median :157.0   Median :6.100
 Mean   :159.7   Mean   :6.033
 3rd Qu.:178.0   3rd Qu.:6.100
 Max.   :199.0   Max.   :6.100
```

Indexing parts of a data frame

You can address an individual column by the name of the whole dataframe, then a dollar, then the name of the column:

```
> mydf$heights
[1] 5.9 6.1 6.1
```

Since `mydf$heights` is a vector, we can us indexing to get at the height of an individual person, for example the height of the third person:

```
> mydf$heights[3]
[1] 6.1
```

We can also treat the whole data frame as a two-dimensional matrix and index an individual cell in that matrix:

```
> mydf[2,1]
[1] 157
```

This indexes row 2, column 1; the weight of the second person. We can also leave out one of the two values in the two-value index to address a whole row or a whole column:

```
> mydf[3,]
  heights weights
3     5.7     133
```

This addresses the whole 3rd row: square brackets – index – comma. The result is, again, a data frame: the data frame containing only the information for the given row(s):

```
> x = mydf[3,]
> x
  heights weights
3     5.7     133
> x$heights
[1] 5.7
```

And here is how to address the whole 2nd column:

```
> mydf[,2]
[1] 5.9 6.1 6.1
```

We can also address multiple rows of a data frame at the same time:

```
> y = mydf[c(1,2),]
> y
  weights heights
1     123     5.9
2     157     6.1
```

This returns a data frame containing all the rows mentioned in the index vector.

Indexing as selection

Now, how about indexing as selection, as you saw it for vectors above? Recall:

```
> x <- c(23, 5172, 48, 202, 5, 90)
> x[x>50]
```

Here is how it looks for data frames:

```
> mydf = data.frame(heights = c(6.2, 5.9, 5.7, 6.1),
    weights = c(162, 134, 133, 170))
> mydf[mydf$weights > 140,]
  heights weights
1     6.2     162
4     6.1     170
```

So, there are two things to keep in mind:

1. If you are choosing rows in the data frame (= persons about which we have data), use the comma after the index
2. To describe a condition about the weights, we have to write `mydf$weights`, as in

   ```
   mydf[mydf$weights > 140,]
   ```

Sorting

Unsurprisingly, the R function for sorting a vector is called `sort`:

```
> x <- c(34, 6, 2, 7, 21398, 539)
> sort(x)
[1]     2     6     7    34   539 21398
```

But there is also a function called `order`. Try it out and guess what it does. Here's an example:

```
> x <- c(34, 6, 2, 7, 21398, 539)
> order(x)
[1] 3 2 4 1 6 5
```

(Please only turn the page once you have a hypothesis.)

The R function `order` computes the rank of each item in an ordering. So, it answers the question: "What order would the elements of x be in if I ordered them?" So, in the example from the previous page,

```
> x <- c(34, 6, 2, 7, 21398, 539)
> order(x)
[1] 3 2 4 1 6 5
```

the first element in the ordered list would be the 3rd in the original list, namely the number 2. The second element in the ordered list would be the 2nd in the original list, namely the number 6. And so on.

Why is that interesting? Because you can now order several vectors in in the same way:

```
> mydf = data.frame(heights = c(6.2, 5.9, 5.7, 6.1),
   weights = c(162, 134, 133, 170))
> mydf
  heights weights
1     6.2     162
2     5.9     134
3     5.7     133
4     6.1     170
> mydf_new = mydf[order(mydf$heights), ]
> mydf_new
  heights weights
3     5.7     133
2     5.9     134
4     6.1     170
1     6.2     162
```

The rows of `mydf_new` are the rows of `mydf`, sorted by height.

Using predefined packages

R has many *packages* that define additional functions. They are not automatically loaded. In fact, you have to download them and install them on your version of R for them to run. The online archive for R packages is CRAN, `http://cran.r-project.org/`.

We will use the `languageR` package, which contains a lot of linguistic datasets used for illustrative purposes in H. Baayen's book *Analyzing Linguistic Data*.

The keyword for importing packages is `library`:

```
> library(languageR)
```

If you need to install the package, you will need to install some other packages as well that are used by languageR. The function `install.packages` takes as it argument a vector of package names, along with optional parameters, for example for specifying the repository:

```
> install.packages(c("rpart", "chron", "Hmisc", "Design",
  "Matrix", "lme4", "coda", "e1071", "zipfR", "ape",
  "languageR"),
  repos="http://cran.r-p-roject.org")
```

The 'verbs' data frame

The `verbs` data frame from Harald Baayen's `languageR` package studies the dative alternation: What features influence whether the dative is being realized as a second object or as a to-PP?

You import `languageR` using

```
> library(languageR)
```

To see what kind of data `verbs` contains, look at its first few lines:

```
> head(verbs)
```

You will see that it starts out like this:

```
  RealizationOfRec  Verb AnimacyOfRec AnimacyOfTheme LengthOfTheme
1               NP  feed      animate      inanimate      2.639057
2               NP  give      animate      inanimate      1.098612
3               NP  give      animate      inanimate      2.564949
4               NP  give      animate      inanimate      1.609438
5               NP offer      animate      inanimate      1.098612
6               NP  give      animate      inanimate      1.386294
```

Features encoded in the `verbs` data frame are:

RealizationOfRec realization of the Receiver as noun phrase (NP) or prepositional phrase (PP). This is what is called a *factor*, that is, something that takes on a string value rather than a numerical value

Verb the verb

AnimacyOfRec is the Receiver animate or inanimate? another factor

AnimacyOfTheme is the Theme animate or inanimate?

LengthOfTheme logarithm of the number of words in the Theme

Writing to files, and reading from files

R has different methods for reading from files and writing to files. For data frames, it is convenient to use `read.table` and `write.table`.

We write the `verbs` data frame to a file:

```
> write.table(verbs, "verbs.df", row.names=F)
```

(Instead of "verbs.df", you can give a full directory path to control where the file is put.) We have written 'verbs' to a file "verbs.df", writing out column names but not row names (setting row.names to F, which is short for FALSE).

The result is a text file that starts like this:

```
"RealizationOfRec" "Verb" "AnimacyOfRec" "AnimacyOfTheme" "LengthOfTheme"
"NP" "feed" "animate" "inanimate" 2.6390573
"NP" "give" "animate" "inanimate" 1.0986123
"NP" "give" "animate" "inanimate" 2.5649494
"NP" "give" "animate" "inanimate" 1.6094379
```

So, write.table writes the data frame as a table, with entries separated by spaces. You can choose whether you want column and row numbers printed. By default they are printed, but you can omit them by setting `row.names=F` and `col.names=F`, respectively.

Note: `row.names` and `col.names` are optional arguments of the function `write.table`: If you omit them, the default values are used. And another note: `T` stands for "true", and `F` stands for "false".

Now you can read the file contents back into an R data frame:

```
verbs.2 <- read.table("verbs.df", header=T)
```

By setting `header=T` we have told R to expect column numbers to be present, and R uses those as the names of the columns of the new data frame verbs.2.

`verbs.2` is a data frame, which we now inspect.

```
> head(verbs2)
```

Some first data inspection

We now plot the information contained in the `verbs`. First: How many words does the Theme usually have in this dataset? We can try to plot it:

```
> plot(verbs$LengthOfTheme)
```

This does not look useful. Let's sort the lengths first:

```
> plot(sort(verbs$LengthOfTheme))
```

How many Receivers are animate, how many are inanimate? The `AnimacyOfRec` entry of `verbs` is a factor, that is, it has string values. How do we count how often each value occurs? The function `summary` does this for us:

```
> summary(verbs$AnimacyOfRec)
  animate  inanimate
      822         81
```

This we can plot:

```
> barplot(summary(verbs$AnimacyOfRec))
```

So far you have seen two functions for plotting things. There are more:

- plot: general-purpose plotting

  ```
  plot(sort(verbs$LengthOfTheme))
  ```

- barplot: bars

  ```
  barplot(summary(verbs$AnimacyOfRec))
  ```

- histogram: how often does which value occur?

  ```
  hist(verbs$LengthOfTheme)
  ```

- boxplot: What is the mean, and what are the outliers?

  ```
  boxplot(verbs$LengthOfTheme)
  ```

- pie: pie charts (frowned upon by statisticians because they are not easy to read: the eye is not good at judging differences in pie size, at least much worse than judging differences in bar height)

  ```
  pie(summary(verbs$AnimacyOfRec))
  ```

Also try

```
> example(plot)
```

and, for more wow factor,

```
> demo(graphics)
```

R has a builtin help function. Use it to learn more about all the optional parameters of the plotting functions:

```
> help(plot)
```

Data types

Data types in R: You have already seen

- strings
- numbers
- vectors
- data frames

There are also:

- lists: a generalization of data frames. Data frames can be viewed as a combination of multiple vectors of the same length, each indexed by a column name. Lists are a combination of multiple pieces of data, any data, each indexed by a name.
- factors, which look like strings, but are restricted sets of values, for example if we know that possible answers by participants were just `c(''yes'',''no'')`.