Reading Data

There are a few principal functions reading data into R.

- `read.table`, `read.csv`, for reading tabular data → verwandeln aus tabellarisch definierten Textdateien einen data-frame
- `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

→ verwandelt aus Text einen character vektor

Writing Data

There are analogous functions for writing data to files

- write.table
- writeLines
- dump
- dput
- save
- serialize

Reading Data Files with read.table

The `read.table` function is one of the most commonly used functions for reading data. It has a few important arguments:

- `file`, the name of a file, or a connection
- `header`, logical indicating if the file has a header line haben die Daten eine Überschr
- `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 dataset
- `comment.char`, a character string indicating the comment character
- `skip`, the number of lines to skip from the beginning wenn z.B der Beginn eines Datenblattes aus allg. Erklärungen besteht
- `stringsAsFactors`, should character variables be coded as factors?

hierüber kann man Kommentare in den Daten markieren

read.table

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

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

default seperator = space

R will automatically

- skip lines that begin with a #
- 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 Telling R all these things directly makes R run faster and more efficiently.
- `read.csv` is identical to read.table except that the default separator is a comma.

Reading in Larger Datasets with read.table

With much larger datasets, 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 dataset. If the dataset is larger than the amount of RAM on your computer, you can probably stop right here.
- Set `comment.char = ""` if there are no commented lines in your file.

Reading in Larger Datasets with read.table

- 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 an 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)
```

man liest nur die ersten 100 Zeilen ein, bestimmt dann deren class & speichert diese Info, anschließend liest man die Gesamtdaten ein & nutzt die gespeicherte Info.

- 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.

andernfalls versucht R das automatisch zu bestimmen, was sehr lange dauern kann

Know Thy System

In general, when using R with larger datasets, it's useful to know a few things about your system.

- How much memory is available?
- What other applications are in use?
- Are there other users logged into the same system?
- What operating system?
- Is the OS 32 or 64 bit?

typischerweise mehr memory verfügbar

Calculating Memory Requirements

I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly, how much memory is required to store this data frame?

$\underbrace{1,500,000 \times 120}_{\text{Zahl der Elemente}} \times 8$ bytes/numeric

= 1440000000 bytes

= 1440000000 / 2^{20} bytes/MB

= 1,373.29 MB

= 1.34 GB → Einlesen braucht etwa doppelt soviel Arbeitsspeicher, damit es ruckelfrei läuft