The Ultimate R Cheat Sheet – Data Management (Version 4)

Google "R Cheat Sheet" for alternatives. The best cheat sheets are those that you make yourself! Arbitrary variable and table names that are not part of the R function itself are highlighted in bold.

Import, export, and quick checks

- **dat1**=read.csv(**"name.csv"**) to import a standard CSV file (first row are variable names).
- attach(**dat1**) to set a table as default to look for variables. Use detach() to release.
- **dat1**=read.delim(**"name.txt"**) to import a standard tab-delimited file.
- **dat1**=read.fwf(**"name.prn"**, widths=c(**8,8,8**)) fixed width (3 variables, 8 characters wide).
- ?read.table to find out more options for importing non-standard data files.
- **dat1**=read.dbf(**"name.dbf"**) requires installation of the foreign package to import DBF files.
- head(**dat1**) to check the first few rows and variable names of the data table you imported.
- names(**dat1**) to list variable names in quotation marks (handy for copy and paste to code).
- data.frame(names(**dat1**)) gives you a list of your variables with the column number indicated, which can be handy for sub-setting a data table (see next page)
- nrow(**dat1**) and ncol(**dat1**) returns the number of rows and columns of a data table.
- length(**dat1**$**VAR1**[!is.na(**dat1**$**VAR1**)] returns a count of non-missing values in a variable.
- str(**dat1**) to check variable types, which is useful to see if the import executed correctly.
- write.csv(**results**, **"myresults.csv"**, na="", row.names=F) to export data. Without the option statements, missing values will be represented by NA and row numbers will be written out.

Data types and basic data table manipulations

- There are three important variable types: numeric, character and factor (a double variable with a numeric and character value). You can query or assign types: is.factor() or as.factor().
- If you import a data table, variables that contain one or more character entries will be set to factor. You can force them to numeric with this: as.numeric(as.character(**dat1**$**VAR1**))
- After subsetting or modification, you might want to refresh factor levels with droplevels(**dat1**)
- Data tables can be set as.data.frame(), as.matrix(), as.distance()
- names(**dat1**)=c(**"ID"**, **"X"**, **"Y"**, **"Z"**) renames variables. Note that the length of the vector must match the number of variable you have (four in this case).
- row.names(**dat1**)=**dat1**$**ID**. assigns an ID field to row names. Note that the default row names are consecutive numbers. In order for this to work, each value in the ID field must be unique.
- To generate unique and descriptive row names that may serve as IDs, you can combine two or more variables: row.names(**dat1**)=paste(**dat1**$**SITE**, **dat1**$**PLOT**, sep="-")
- If you only have numerical values in your data table, you can transpose it (switch rows and columns): **dat1_t**=t(**dat1**). Row names become variables, so run the row.names() function above first.
- **dat1**[order(**X**),] orders rows by variable X. dat[order(**X,Y**),] orders rows by variable X, then variable Y. **dat1**[order(**X,-Y**),]. Orders rows by variable X, then descending by variable Y.
- fix(**dat1**) to open the entire data table as a spreadsheet and edit cells with a double-click.

Creating systematic data and data tables

- c(1:10) is a generic concatenate function to create a vector, here numbers from 1 to 10.
- seq(0, 100, 10) generates a sequence from 0 to 100 in steps of 10.
- rep(5,10) replicates 5, 10 times. rep(c(1,2,3),2) gives 1 2 3 1 2 3. rep(c(1,2,3), each=2) gives 1 1 2 2 3 3. This can be useful to create data entry sheets for experimental designs.
- data.frame(**VAR1**=c(1:10), **VAR2**=seq(10, 100, 10), **VAR3**=rep(c(**"this"**, **"that"**),5)) creates a data frame from a number of vectors.
- expand.grid(**SITE**=c(**"A"**,**"B"**),**TREAT**=c(**"low"**,**"med"**,**"high"**), REP=c(1:5)) is an elegant method to create systematic data tables.

Creating random data and random sampling

- `rnorm(10)` takes 10 random samples from a normal distribution with a mean of zero and a standard deviation of 1
- `runif(10)` takes 10 random samples from a uniform distribution between zero and one.
- `round(rnorm(10)*3+15))` takes 10 random samples from a normal distribution with a mean of 15 and a standard deviation of 3, and with decimals removed by the rounding function.
- `round(runif(10)*5+15)` returns random integers between 15 and 20, uniformly distributed.
- `sample(c("A","B","C"), 10, replace=TRUE)` returns a random sample from any custom vector or variable with replacement.
- `sample1=dat1[sample(1:nrow(dat1), 50, replace=FALSE),]` takes 50 random rows from dat1 (without duplicate sampling). This can be handy for bootstrapping or to run quick test analyses on subsets of very large datasets.

Sub-setting data tables, conditional subsets

- `dat1[1:10, 1:5]` returns the first 10 rows and the first 5 columns of table dat1.
- `dat2=dat1[50:70,]` returns a subset of rows 50 to 70.
- `cleandata=dat1[-c(2,7,15),]` removes rows 2, 7 and 15.
- `selectvars=dat1[,c("ID","YIELD")]` sub-sets the variables ID and YIELD
- `selectrows=dat1[dat1$VAR1=="Site 1",]` sub-sets entries that were measured at Site 1. Possible conditional operators are == equal, != non-equal, > larger, < smaller, >= larger or equal, <= smaller or equal, & AND, | OR, ! NOT, () brackets to order complex conditional statements.
- `selecttreats=dat1[dat1$TREAT %in% c("CTRL", "N", "P", "K"),]` can replace multiple conditional == statements linked together by OR.

Transforming variables in data tables, conditional transformations

- `dat2=transform(dat1, VAR1=VAR1*0.4)`. Multiplies VAR1 by 0.4
- `dat2=transform(dat1, VAR2=VAR1*2)`. Creates variable VAR2 that is twice the value of VAR1
- `dat2=transform(dat1, VAR1=ifelse(VAR3=="Site 1", VAR1*0.4, VAR1))` Multiplies VAR1 by 0.1 for entries measured at Site 1. For other sites the value stays the same. The general format is ifelse(condition, value if true, value if false).
- The `vegan` package offers many useful standard transformations for variables or an entire data table: `dat2=decostand(dat1,"standardize")` Check out `?decostand` to see all transformations.

Merging data tables

- `dat3=merge(dat1,dat2,by="ID")` merge two tables by ID field.
- `dat3=merge(dat1,dat2,by.x="ID",by.y="STN")` merge by an ID field that is differently named in the two datasets.
- `dat3=merge(dat1,dat2,by=c("LAT","LONG"))` merge by multiple ID fields.
- `dat3=merge(dat1,dat2,by.x="ID",by.y="ID",all.x=T,all.y=F)` left merge; `all.x=F, all.y=T` right merge; `all.x=T,all.y=T` keep all rows; `all.x=F,all.y=F` keep matching rows.
- `cbind(dat1,dat2)` On very rare occasions, you merge data without a criteria (ID). This is generally dangerous, because the commands will slap the two tables together without checking the order!
- `dat3=rbind(dat1,dat2)` adding rows of two data tables. The variables have to match exactly and you will get error messages if they don't match. So, unlike cbind(), rbind() is generally safe to use.
- `dat3=rbind.fill(dat1,dat2)` will force non-matching datasets together, filling missing values and executing variable type conversions where appropriate. Requires the `reshape` package.

Summary statistics for variables and tables

- `mean() weighted.mean(,) median() max() min() range() which.max() which.min() var() sd() quantile() quantile(,c(0.025,0.05,0.95,0.975)) rank(x)` some descriptive statistical functions for variables or vectors. For all functions, and important option is `na.rm=T`, which means that missing values are ignored in the calculations, e.g. `mean(VAR1, na.rm=T)`. Without that option, the function returns missing values as a result of missing values in the input.

- `rowSums()`, `colSums()`, `rowMeans()` or `colMeans()` applies functions to rows or columns of a table. For example, `rowsum(dat1[,10:15])` will return the row-sums of the variables in columns 10 to 15. Don't forget `na.rm=T`.
- `apply(dat,1,max)` apply any function (e.g. `max`), to either rows (`1`) or columns (`2`) of a table (`dat`).

Pivot table functionality

- The functions aggregate and ddply can be used to summarize data similarly to working with Excel pivot tables. Aggregate has simpler syntax if you have many variables that you want to summarize in the same way; ddply is better if you have few variables but want several custom summary statistics.
- `aggregate(dat1[,4:9], by=list(TREAT1=dat1$TREAT1, TREAT2=dat2$TREAT2), mean)` calculates the means of a number of numerical variables in columns 4 to 9 for two treatments.
- `ddply(dat1,.(TREAT), summarise, mVAR1=mean(VAR1))` returns means of VAR1 by a class variables TREAT. This requires installation of the library `plyr`.
- `ddply(dat1,.(TREAT1, TREAT2), summarise, cVAR1=length(VAR1[!is.na(VAR1)]))` returns a count of non-missing values in variable VAR1 for each combination of two class variables.
- `ddply(dat1,.(TREAT1, TREAT2), summarise, mVAR1=mean(VAR1, na.rm=T), seVAR1=sd(VAR1, na.rm=T)/sqrt(length(VAR1[!is.na(VAR1)])))`. A clever piece of code to calculate means and standard errors, with missing values being properly handled.

Long-to-wide and wide-to-long data table conversions

- First we generate an artificial dataset to play with (copy and paste to R to see what it does):

  ```
  long=expand.grid(SITE=c("A","B"),TREAT=c("low","med","high"), REP=c(1:5))
  long$YIELD=round(rnorm(10)*5+15); long
  ```

- Long-to-wide conversion with the `reshape2` package, where SITE and REP remain columns, but the treatment levels of TREAT now become several new columns:

  ```
  wide=dcast(long, SITE+REP~TREAT, value.var="YIELD")
  ```

  Wide-to-long conversion back to what we had before. The variables that you want to maintain as columns are SITE and REP, all others will be gathered into a new variable where the remaining columns become treatment levels. You have to fix the variable names to get to the original long:

  ```
  long2=melt(wide, id.vars=c("SITE","REP"))
  names(long2)=c("SITE","REP","TREAT","YIELD")
  ```

Dealing with missing values

- `transform(dat1, VAR1=ifelse(is.na(VAR1),0,VAR1))` sets missing values in variable VAR1 to 0. You may do this if missing has a biological meaning of zero, e.g. zero productivity.
- `transform(dat1, VAR1=ifelse(VAR1==0,NA,VAR1))` … or vice versa if zero really means that the measurement was not taken.
- `dat1[is.na(dat1)]=0` sets missing values to 0 in entire dataset.
- `dat1[dat1==0]=NA` ... vice versa.
- `dat2=na.omit(dat1)` delete rows with missing values in any variable
- `dat2=dat1[!is.na(dat1$VAR1),]` delete rows with missing values in VAR1
- `dat2=transform(dat1, VAR2=ifelse(is.na(VAR1),NA,VAR2))`modify a second variable (here: set to missing) based on missing values in a first variable.

Dealing with duplicate data entries or IDs:

- `unique(dat1)` or `dat1[!duplicated(dat1),]` removes exact duplicate rows.
- `dat1[duplicated(dat1),]` returns the duplicate rows.
- `dat1[!duplicated(dat1[,c("ID")]),]` removes all rows with duplicate IDs (first is kept).
- `dat1[duplicated(dat1[,c("ID")]),]` returns the rows with duplicate IDs.

Loops and automation

- `v1=vector(length=20)` initializes an empty vector with 20 elements. This is often required as an initial statement to subsequently write results of a loop into an output vector or output table.
- `m1=matrix(nrow=20, ncol=10)` similarly initializes an empty matrix with 20 rows and 10 columns.
- `for (i in 1:10) {` one or more operations with `v1[i]` or `m1[,i] }`
- `for (i in 1:10) { for (j in 1:20) {` one or more operations with `m1[j,i] }}`
- Example for an application, where a for-loop is used to calculate cumulative values. Copy and paste the code below into R to see what it does.

```
dat=round(rnorm(10)+2)
cum=vector(length=10)
cum[1]=dat[1]
for (i in 2:length(dat)) { cum[i]=cum[i-1]+dat[i] }
cbind(dat,cum)
```

- Example for an application where a for-loop allows automatic data processing of multiple files in a directory. This batch-converts DBF format files into CSV format files. With similar code, you could merge a large number of files into one master file, or do manipulations or analysis on multiple files consecutively.

```
library(foreign)
setwd("C:/your path/")
a<-list.files(); a
for (name in a) { dat1=read.dbf(name)
  write.csv(dat1, paste(name,".csv"), row.names=F, quote=F) }
```

Handy built-in functions

- `paste("hello", "world")` joins vectors after converting them to characters. The `sep=""` option can place any character string or nothing between values (a single space is the default)
- `substr("Year 1998",6,9)` extracts characters from start to stop position from vector
- `tolower("Year 1998")` convert to lowercase - handy to correct inconsistencies in data entry.
- `toupper("Year 1998")` convert to uppercase
- `nchar("Year 1998")` number of characters in a string, allows you to substring the last four digits of a variable regardless of length, for example: `substr(VAR1,nchar(VAR1)-3,nchar(VAR1))`
- Plenty of math functions, of course: `log(VAR1), log10(VAR1), log(VAR1,2), exp(VAR1), sqrt(VAR1), abs(VAR1), round(VAR1,2)`

Programming custom functions

- You can program your own functions, if something is missing, or if you want to utilize a bunch of code over and over to make similar calculations. Here is a clever example for calculating the statistical mode of a variable, which is missing from the built-in R functions.

```
mode=function(input){ freq=table(as.vector(input))
                      descending_freq=sort(-freq)
                      mode=names(descending_freq)[1]
                      as.numeric(mode)
                      }
VAR1=c(1,3,3,2,3,2,2,3,5,3)
mode(VAR1)
```

More information, help, and on-line resources

- Adding a question mark before a command or functions brings up a help file. E.g. `?paste`. Be sure to check out the example code at the end of the help file, which often helps to understand the syntax.
- More information and R resources can be found with the search engine http://www.rseek.org