R quick reference card

Fabrice Niessen

February 13, 2013

Contents

1. Notes

- http://www.statmethods.net/index.html
- http://tryr.codeschool.com/

2. Commands

getwd() figure out what your working directory is

read.csv("file.csv") read data file, create data frame

read.table("file.txt") read data file, create data frame

dir() list all the **files** in the working directory

ls() show objects in my workspace

source("file.R") load R code file

library(file) load package

data(dataset) load dataset

3. Operators

<− assign

4. Function

```
myfunction <- function(x) {
    y <- norm(100)
    mean(y)
}
```

args() look at the arguments of a function

4.1 General

mean() take the mean

median() take the median

range() give the min and the max (vector of length 2) of the observations (vector of numbers)

length()

x:y create an **sequence of** integers from x to y (x < y or x > y)

seq(from, to, len) create a **sequence of** fractional numbers

cor() correlation function

vector(class, length) create an empty vector

matrix(nrow = x, ncol = y) create an empty matrix

array() create an array

attributes() access (set or modify) the (list of) attributes of an object

print(object) explicitly print out an object

c() create vectors of objects (that is, **concatenate** or combine things together)

class() show the class of the object

as.*() explicitly coerce from one class to another

> **as.numeric()** coerce the (`character`) column to be `numeric`
>
> as.Date("January 2, 2007", "%B %d, %Y") = "2007-01-02"
>
> **as.ts()** Convert to a time series object

dim() give the dimension attribute

cbind() column-bind

rbind() row-bind

list() construct a list (the indexes of the elements of a list have double brackets around them)

factor(character vector) create a factor variable (with levels by alphabetical order)

table(vector) **count** the number of observations in each level; give a frequency of how many *levels* (see factors) there are

unclass(vector) strip out the class (see factors)

is.na() test if object is `NA`

is.nan() test if object is `NaN`

data.matrix() convert a data frame to a matrix (forced **coercion**!)

data.frame() create a data frame

names() show or give a name to each **element** of a vector

dimnames() show or give a name to each **row element and** each **column element** of a matrix

complete.cases() subset all the missing values

rep()

str(.Platform) what is the operating system

str(function) show arguments

version is the OS 32 or 64-bit

rm() remove objects from your workspace

```
rm(list=ls()) # remove everything from the workspace
```

split(dataframe, dataframe$column)

rbinom(1, 1, 0.5) generating 1 observation of 1 head flip of a fair **coin** (0.5)

abs() absolute value

lm(yValues x̃Values) (in the `stats` package) linear regression modeling function

arg(function.default)

paste() **concatenate** a set of **strings together** to create one string or a vector of strings

cat() concatenate together a set of strings and prints out the concatanated string (to a file or to the console)

gl(n, k, labels = c("Male", "Female")) create a factor variable, with `n` groups and `k` iterations of each, with names

nchar() get the length of a string

summary() produce a summary of the data frame

%in% membership test

5. Getting help

- Google
- R general mailing list[1]
- Stack Overflow

```
?function
?dataset
package ? lattice
library(help = lattice)
```

6. Overview and history of R

- R system that you download from CRAN[2] :
  - `base` package
  - Familiar packages (`utils`, `datasets`, ...)
  - Recommended packages (`lattice`, ...)
  - 4,000 other packages
- Some R manuals:
  - An introduction to R
  - Writing R extensions
  - R data import/export

7. Introduction to the R language: Data types and basic operations

- 5 "atomic" **classes** of objects
  1. `logical` (`TRUE` / `FALSE`)
  2. `integer` (with explicit `L` suffix, such as `1L`)
  3. `numeric` (*real* numbers, such as `0.5` or `1`)
  4. `complex` (such as `1+0i`)
  5. `character` (such as the string "hello") = **lowest**
- Basic object: **vector**
  - Set of elements of the same class

[1] mailto:r-help@r-project.org
[2] http://cran.r-project.org/

  - When objects of different classes are mixed in a vector, *coercion* occurs behind the scene so that every element is of the same class (the "lowest common denominator" class)
  - **Matrices** are vectors with multiple dimensions (`dimension` attribute)
- Special types of vectors
  - **list** (vector of objects of possibly different classes)
  - **factor** (qualitative variable used to represent *categorical* data, to store self-describing codes for *labels* such as "male" and "female", or "low", "medium" and "high")
    * Unordered or ordered
  - **data frame** (used to store tabular data where each column can be of a different class)
    * Row = observation, column = variable
    * Special type of list (of variables in columns) where every element has the same length
    * Special attribute `row.names` (every row has a name, or defaults to a sequence of integers)
    * Most often created by calling `read.table()` or `read.csv()`
- Special values
  - `Inf` for infinity
  - `NaN` for "Not a Number" (undefined mathematical operation)
  - `NA` for **missing value** (for example, the result of nonsensical coercion)
    * `NA` values have a class
    * A `NaN` value is also `NA` (but the converse is not true)
  - `NULL`???
- Attributes
  - names
  - dimensions
  - class
  - length
- Everything to the right of the # is a comment
- Matrices

- They are constructed *column-wise* (vector inserted by column)
- They can be created from vectors by adding a dimension attribute: dim(m) <- c(2,5)
- They can be created by *column-binding* or *row-binding*

7.1 Subsetting

- Syntax
  - `[` always return an object (one or more elements) of the **same class** as the original (one exception: matrices – lists are no exception!)
    * Used to extract multiple elements of a list (`x[c(1,3)]`)
    * `dfrm[,-1]` to remove the first column
  - `[[` is used to extract a **single element** of a list or a data frame; its class will not necessarily be a list or data frame
    * Can be used with *computed* indices
    * Can extract nested elements of a list:

```
x[[1]][[3]] = x[[c(1,3)]] # third element of the first element
```

    * Partial matching allowed at the command-line: x“a”, exact = FALSE instead of x“aardvark”
  - `$` is used to extract an element of a list or data frame **by name** (`x$bar` = x“bar”): you don't have to remember where the element is in the list
    * Can only be used with literal names
    * Partial matching allowed at the command-line: `x$a` instead of `x$aardvark`
- Subsetting by using 2 types of index:
  - a **numeric** index (`x[2]`, `x[1:4]`)
  - a **logical** index (`x[x > "a"]`)
- Create a logical vector:

```
u <- x > "a"
x[u] # get all elements which are greater than "a"
```

- Matrices can be subsetted with `(row, col)` type indices
  - Indices can also be missing:

    x[i,] row i

    x[,j] column j

    - By default, a single element is retrieved as a vector of length 1 rather than a 1x1 matrix
    - By default, a single column or a single row is retrieved as a vector, not as a matrix
    - Can be turned off by setting `drop = FALSE` (don't drop the dimension)

- Remove missing values (`NA`)

```
missing <- is.na(x) # logical vector
x[!missing]
```

- Take the subset of all objects (`x` and `y`) that has no missing values

```
bothnonmissing <- complete.cases(x, y) # logical vector
x[bothnonmissing]
```

- Take all the rows of a data frame where **all the values** are not missing

```
nonmissing <- complete.cases(x)
x[nonmissing, ]
```

8. Vectorized operations

- Avoid writing loops (code is a lot simpler)

```
x + y # element-wise addition
```

- Similar for the matrices:

```
x * y # element-wise multiplication
x %*% y # true matrix multiplication
```

9. Reading and writing data

- `read.table`, for reading tabular data Important arguments:
    - `file`,
    - `header`,
    - `sep` (defaults to the **space**)
    - `colClasses`, class of each column
    - `nrows`,
    - `comment.char`,
    - `skip`, number of lines to skip from the beginning
    - `stringsAsFactors` (defaults to `TRUE`)
- `read.csv`, for reading tabular data

  - header defaults to `TRUE`
  - default separator: **comma**
- `readLines`, for reading lines of a text file
- For large datasets:
  - set comment.char = "" if there are no comments in your file
  - use the `colClasses` argument Quick and dirty way to figure out the classes of each column

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

  - set nrows helps with memory usage
- Textual formats (potentially recoverable in case of corruption)

  `dput()` deparse a single R object

  `dump()` can be used on multiple R objects
- *Connections* can be made to:
  - `file` (most common)
  - `gzfile`, file compressed with `gzip`
  - `bzfile`, file compressed with `bzip2`
  - `url`
- `file()`
  - `description` is the name of the file
  - `open` is a code (read-only, write, append)

10. str() function

str() *compactly* display the internal **structure** of an object

- Alternative to `summary()`
- Give the arguments of functions

head() look at the first 6 rows

tail() look at the last 6 rows

11. Datasets

```
library(datasets)
airquality
```

12. Control structures

- Control structures mentioned here are primarily useful for writing programs; for command-line interactive work, the `*apply` functions are more useful

12.1 If, else

```
if(<condition>) {
    ## do something
} else {
    ## do something else
}
```

Here, the entire `if/else` construct is all about assigning a value to `y`:

```
y <- if(x > 3) { # `x' must be a scalar here, not a vector
    10
} else {
    0
}
```

12.2 For

```
for(i in 1:10) { # successive values from a sequence or vector
    ## do something
}

for(letter in x) { # take elements from the vector
    ## do something
}
```

seq_along(vector) create an integer sequence that's equal to the **length of the input vector**

seq_len(integer) create an integer sequence that's as long as the integer in input

nrow(dataset) tell the number of rows

ncol(dataset) tell the number of columns

names(dataset) tell the names of each column included in the dataframe

12.3 While

```
while(z >= 3 && z <= 10) { # conditions are always evaluated from left to right
    ## do something
}
```

12.4 Repeat

- Initiate an infinite loop

```
repeat {
    ## do something
    if(<condition>) {
        break # only way to exit a repeat loop
    }
}
```

- Better to use a for loop with an hard limit on the number of iterations that it's allowed to run

12.5 Break

- Break the iteration of a loop

12.6 Next

- Skip an iteration of a loop

12.7 Return

- Exit an entire function and return a given value

13. Writing functions

- Functions are R objects of class `function`

```
f <- function(<arguments>) {
    ## do something
}
```

- Functions can be **passed as arguments to other functions**
- Functions can be nested, so that you can define a function inside of another function (implications: see **lexical scoping**)
- The return value of a function is the last expression in the function body to be evaluated
- 3 types of. . .
  - formal argument
  - local variable
  - free variable

13.1 Arguments

- *Named arguments* can potentially have *default values* (useful: not every function call makes use of all the formal arguments; some can be *missing*)
- The *formal arguments* are the arguments included in the function definition
- `formals()` returns a list of all the formal arguments of a function
- Arguments can be matched **positionally** or **by name**

  - When an argument is matched by name, it is "taken out" of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition

- Named arguments help when:
  - you want to use the defaults for evererything except for an argument near the end of the list
  - you can't remember the position of the argument
- Function arguments can also be partially *matched*
  1. Check for an exact match
  2. Check for a partial match
  3. Check for a positional match
- When defining a fucntion, you can also set an argument value to `NULL` (there is nothing there)
- Arguments to functions are evaluated *lazily* (only when needed to be evaluated
- The `...` argument indicates a variable number of arguments
  - Used when extending a function and you don't want to copy the entire argument list of the original function
  - Used by generic functions (such as `mean`) so that extra arguments can be passed to methods
  - Used when the number of arguments cannot be known in advance (see `paste` function)
  - Any argument that appears *after* the `...` must be named explicitly and cannot be partially matched

13.2 Scoping rules for R

- R searches through the *search list* (a series of `environments`, an environment being a collection of symbol/value pairs) to bind the appropriate value to a symbol:
  - Search the global environment `.GlobalEnv` (always the first)
  - Search the namespaces of each of the packages on the search list

```
search() # find the search list
```

  - Search the `base` package (always the last element)
- Last loaded package gets put in position 2 of the search list and everything else gets shifted
- Separate namespaces for functions and non-function objects

- R uses **lexical** (or *static*) **scoping** (instead of *dynamic scoping*): *the value of free variables are searched for in the environment in which the function was* **defined** (until the empty environment, after the `base` package)
  - With **dynamic** scoping, the value of free variables is looked up in the environment from which the function was **called** (*calling environment* = *parent frame*) – see slide 24 of “Scoping Rules for R” for a comparative example
  - Other languages that support lexical scoping: Scheme, Perl, Python, Common Lisp
  - Consequences: all objects must be stored in memory, and all functions must carry a **pointer to their respective defining environment**
- Every environment has one parent environment (next thing down on the search list); it is possible for an environment to have mulitple “children”
- A function + an environment = a (*function*) *closure*
- In R (unlike C), you can have functions defined *inside other functions* – in this case, the environment in which a function is defined is the body of another function!

```
make.power <- function(n) { # "constructor" function
    pow <- function(x) {
        x^n # n is a free variable (not defined inside pow)
    }
    pow # return function as return value
}
```

This function returns another function as its value

```
cube <- make.power(3)
square <- make.power(2)
```

- Functions:

  environment(f)

  parent.env(environment) next thing down on the search list

  ls(environment) list all the variabls in the environment

  get(object, environment) get the value of an object inside an environment

14. Application: optimization

- Routines:

  optim()

  nlm()

  optimize()

- Most of those functions in R attempt to *minimize* functions by default; so, when you write your **objective** function, if they're designed to be maximized, then you have to take the negative of those functions so that you can minimize them
- Standard deviation `rnorm`
  - number of observations, `n`
  - mean, `mu`
  - standard deviation, `sigma`

15. The `apply` functions

Alternative (to `for` **loops**) to apply a function (or summary statistics)

15.1 lapply

- Loop over a single **list** and apply a **function** on each element
  - (coerced) list `X`
  - function `FUN`
  - other arguments `...`
- Always returns a **list** back (that is, not a simplified result)

```
x <- list(a = 1:5, b = rnorm(10))
lapply(x, mean)
```

```
lapply(1:4, runif, min = 0, max = 10) # arguments passed through the ...
```

- Extract the first column of each matrix of a list

  ```
  lapply(x, function(elt) elt[,1])
  ```

15.2 split

- Auxiliary function, useful in conjunction with functions like `lapply` or `sapply`
- Takes a vector, and **split** it into subpieces (the number of groups identified by the levels of a factor variable)
- Always return a **list** back

  drop = TRUE Don't keep the empty levels of the factor

- Used in conjunction with functions like `lapply` or `sapply` to apply a function to those individual groups

  ```
  lapply(split(x, f), mean)
  ```

- Like `tapply`, but without applying the summary statistics
- Splitting a data frame (or other kinds of lists), and apply an *anonymous function*

```
s <- split(airquality, airquality$Month) # split according to month
lapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
```

15.3 sapply

- Same as lapply, but tries to **simplify** the result in a much more compact format (put all the elements into a **vector** or a **matrix**, instead of returning a list)

```
sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
```

- Pass `na.rm` argument to `ColMeans` to remove the missing values before calculating the mean

```
sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")],
                               na.rm = TRUE))
```

15.4 apply

- Apply a (anonymous) function over the **margins** of an array
  - Very useful if you wanna take **summaries** of matrices or higher-dimensional arrays
  - Often used to apply a function to the rows or columns of a matrix
  - Can be used with general arrays (such as array of matrices)
- apply(X, MARGIN, FUN, ...)
  - X array (= vector which has dimensions attached to it; matrix = 2-dimensional array)
  - MARGIN is an integer vector which indicates which margins should be "retained"
  - FUN you want to apply to each of the margins
  - ... for other arguments you wanna passed to the function

```
x <- matrix(rnorm(200), 20, 10)
apply(x, 2, mean)
```

- margin 1 = aplly FUN for each row (dimension 1); preserve all the rows, eliminate all columns, get a vector of the number of rows
- margin 2 = for each column (dimension 2); first dimension has been eliminated
- Shortcut functions (*much* faster on large matrices):
  - rowSums = apply(x, 1, sum)
  - rowMeans = apply(x, 1, mean)

  - colSums = apply(x, 2, sum)
  - colMeans = apply(x, 2, mean)

```
x <- matrix(rnorm(200), 20, 10)
apply(x, 1, quantile, probs = c(0.25, 0.75)) # no default value for `probs'
```

15.5 tapply(X, INDEX, FUN)

- Short for “table apply”
- Apply a **function** over **subsets** of a vector:
  - Splits up a vector into little groups (such as men and women, indicated by a **factor**),
  - Applies a function to those groups and
  - Brings the pieces back together
- Function may be anonymous
- Simplify = FALSE to get back a list

```
tapply(x, f, mean, simplify = FALSE)   =   lapply(split(x, f), mean)
```

interaction(f1, f2) combines all the levels of the first factor with all the levels of the second factor

15.6 `mapply`

- mapply is a multivariate version of `lapply`: applies a function in parallel over a set of arguments
- Apply a **function** to the elements of **multiple lists** in parallel
  - For example, apply a function over 2 lists, where the elements of the first list go into one argument of the function, and the elements of the second list go into another argument of the function
- **Vectorizing a function** (that doesn’t allow for vector arguments)

```
mapply(rnorm, 1:5, 1:5, 2) # fixed standard deviation
```

is the same as

```
list(rnorm(1, 1, 2),
     rnorm(2, 2, 2),
     rnorm(3, 3, 2),
     rnorm(4, 4, 2),
     rnorm(5, 5, 2))
```

16. Debugging

- 3 main indications of a problem / `condition`:
  - `message`

```r
print("message")
```

- `warning`

```r
log(-1)
```

- `error`

```r
stop("error") # throw an error
```

- General notion of a `condition`: you can create your own if something specific happens
- Functions:

  invisible(x) **prevents auto-printing** of the last element in the function value; **still return** the same object

- **Interactive** debugging tools to find problematic code:

  `traceback` print out the function call stack – you have to call it **immediately** after the error occurred

  `debug` flag a function for "debug" mode to step through any function (you wrote or not), one expression at a time, from the top of the function

  browser put the function in debug mode anywhere in your code; in the browser, there is nothing in your environment except for the function arguments (and the default values which are not listed). You can nest browser frames: you can call the debug function even while you're in the debugger

  trace insert debugging code into a function, without actually editing the function

  recover modify the default behavior (of getting the console back) by creating an error handler

```r
options(error = recover)
```

- Blunt techniques:
  - print
  - cat

17. Plotting

- 2 systems:

  base graphics are constructed piecemeal by **different function calls**; you can add things one by one:

- annotate (some of the points in) the plot,
- put some points on the canvas,
- draw a title,
- add some axis labels,
- add some colors,
- add a legend

lattice graphics are constructed via a single function call: all options have to be specified at once (advantage: that allows R to calculate the necessary spacings, margins and font sizes)

- Behavior
  - Base graphics functions have a "side effect": they plot data directly to the graphics device
  - Lattice graphics functions return an object of the class `trellis` (object designed for plotting)

    When you call Lattice functions, even if you don't assign it to a "plot object", the result will be **auto-printed** (generate the plot **on the graphics device**), so it will look like Lattice functions have a "side effect"
- You cannot use functions from the base plotting system in a Lattice plot

17.1 `graphics`

- "Base" graphing system (`plot`, `hist`, `boxplot`, ...)

par() control **all the graphing parameters** that you can specify (*defaults* for all plots in a session, which can be overridden as arguments to specific plotting functions)

pch plotting character (default: open circle symbol)

lty line type (default: `solid` line)

lwd line width

col plotting color

las orientation of the axis labels on the plot (`las = 2` will set the tick labels to be perpendicular to the axis)

bg background color (default: `transparent`)

mar margin size (vector of 4 numbers, 1 per side)

```
par(mar = c(2,2,1,1))
```

oma outer margin size (relevant if you have more than one plot per canvas)

mfrow number of plots per row and per column on the canvas (filled row-wise)

```
par(mfrow = c(2,1)) # 2 rows and 1 column
plot(x, y)
plot(x, z)
```

mfcol number of plots per row and per column on the canvas (filled column-wise)

Look at the defaults:

```
par("lty")
```

- Make a plot

  plot(x, y) **scatterplot** (or another type of plot, depending on the class of the object being plotted)

  plot(x, y, type = "n") set up the plot window, but don't actually plot the data in there

  plot(x) plot the (numeric) data against the index (`1:N`)

  hist(x) make an histogram showing the distribution of the numeric vector `x`

  Generic function: you can call it on different types of data. When you call `hist` on a `Date` object, it requires an interval ("day" / "week" / "month" / "year") in order to break it up into sequences

  boxplot(y x̃) boxplot of the `y` variable by the grouping variable `x` (usually a `factor`)

- Add to a plot, when you've already constructed a plot

  lines add lines (connect all the dots)

```
lines(x, y) # all lines
```

```
fit <- lm(y ~ x)
abline(fit, lwd = 3, col = "blue")
```

  points add points (`col` for boundary color, `bg` for fill color)

```
example(points) # example file for points
```

  text add text labels

```
text(-2, 2, "Label")
```

  title add a title (or axis labels, subtitle, . . .)

```
title("plot")
plot(x, y, xlab = "Weight", ylab = "Height", main = "Scatterplot")
```

  mtext add text to the margins

axis **annotate the axis** (tick marks, labels)

legend add a legend

```
legend("topleft", legend = "Data", pch = 1)
```

`?Devices` list graphical devices

pdf vector format (very, very large for a graphic with **2 million points** on it: specify information for every single object on the plot)

png bitmapped format (specify information for pixels), losless **compression**, but **does not resize well**

jpeg lossy compression

bitmap if you're running R in a batch mode (you can't use the `png` and `jpeg` functions)

- Copy the plot to another device

  dev.copy2pdf copy a plot to PDF

- Plot groups separately

```
plot(x, y, type = "n")
points(x[g == "Male"], y[g == "Male"], col = "green")
points(x[g == "Female"], y[g == "Female"], col = "blue", pch = 19)
```

17.2 lattice

- Tellis graphics

  xyplot **scatterplots**

  bwplot box-and-whiskers plots ("boxplots")

  histogram histograms

  stripplot like a boxplot but with actual points (instead of boxes)

  dotplot plot dots on "violin strings"

  splom scatterplot matrix, like `pairs` in the base graphics system

  levelplot, contourplot for plotting "image" data

```
library(lattice)
```

- Generally create plots all in one go, from a single function call
- Strength of Lattice functions = conditioning plots: you can plot the **relationship** between `x` and `y`, **conditioned** on the levels of a third variable (factor variable `f` or variable cut into different ranges, see `equal.count`)

```
y ~ x | f * g # formula
```

  – `y` = response (on the y-axis)

  - x = input, predictor (on the x-axis)
  - f * g = factors which are **interacting** with each other (often, just one factor)

- Tell the function xyplot where to find the variables y and x: look up names inside the environmental data frame (because they are not objects in my workspace)

```
xyplot(y ~ x, data = environmental)
```

- Lattice functions have a separate panel created for each level of the factors

```
xyplot(y ~ x | f)
```

The **panel function** controls what happens inside each panel of the entire plot

I can create my own (anonymous) panel function:

```
xyplot(y ~ x | f,
       panel = function(x, y, ...) { # x and y = data that appear in a specific pa
               panel.xyplot(x, y, ...) # plot using all the default options
               panel.abline(h = median(y), lty = 2) # plot the median of the y va
       })
```

Add a (simple linear) regression line to each panel of the plot:

```
xyplot(y ~ x | f,
       panel = function(x, y, ...) {
               panel.xyplot(x, y, ...)
               panel.lmline(x, y, col = 2)
       })
```

```
xyplot(y ~ x | f, pch = 20,
       panel = function(x, y, ...) {
               panel.xyplot(x, y, ...)
               fit <- lm(y ~ x)
               panel.abline(fit, lwd = 2) # add a regression line to the plot
       })
```

- Options

  - Don't worry about spacing: everything gets automatically adjusted!

main	title
layout = c(1,4)	4 panels on top of each other
as.table = TRUE	change the order in which the panels are drawn, from top to bottom (default: from bottom to top)
xlab	label on the x-axis
ylab	label on the y-axis

- Functions

x.cut <- equal.count(x, l) cut the original x variable, creating l different ranges (l̃evels, which may overlap slightly)

panel.loess(x, y) give a smooth rendition of the linear model

17.2.1 Splom

- Look at pairwise relationships possible in the data frame (make a scatter plot of all the different variables against each other)

```
splom(~ df)
```

17.2.2 Histogram

- Histogram of `x`

```
histogram(~ x, data = df)
```

- Histogram of `x`, as `f` varies (for each range of `f.cut`)

```
histogram(~ x | f.cut, data = df)
```

17.3 Mathematical annotations

- Produce LaTeX-like symbols onto your plots (relevant to both the `base` plotting and the `lattice` plotting system)
- Math symbols are expressions in R, so you have to use the `expression` function to encode them when you use them in a plot
- List of allowed symbols

```
?plotmath
```

- Plotting functions that take arguments for text (title, axis, text, labels, ...) generally allow expressions for math symbols

```
plot(0, 0, main = expression(theta == 0),
     ylab = expression(hat(gamma) == 0),
     xlab = expression(sum(x[i] * y[i], i == 1, n))
```

- Paste strings together

```
expression("The mean " * bar(x) * " is " * ...)
```

- Use a computed value in the annotation

```
xlab=substitute(bar(x) == k, list(k=mean(x))) # substitute k
```

17.4 Plotting and Color

- Proper use of color can help to describe the relationships that you're trying to demonstrate
- `colors()` lists the names of colors you can use
- `grDevices` package has functions which **take a palette of colors** and help to interpolate between the colors

colorRamp() return a function that takes **values between 0 and 1** and interpolates between the extremes of the color palette

```
pal <- colorRamp(c("red", "blue")) # return a function `pal'
pal(0)
pal(0.5)
pal(1)
pal(seq(0, 1, len = 10)) # sequence between 0 and 1 -> sequence of colo
```

colorRampPalette() return a function that takes **integer arguments** and return a (character) vector of hexadecimal colors interpolating the palette

```
pal <- colorRampPalette(c("red", "yellow")) # return a function `pal'
pal(2) # return 2 colors
pal(10) # return 10 colors
```

- RColorBrewer package contains interesting color palettes of 3 types:

```
library(RColorBrewer)
cols <- brewer.pal(3, "BuGn") # 3 (primary) colors from the "blue/green" palette
pal <- colorRampPalette(cols)
image(volcano, col = pal(20)) # 20 different colors
```

  – **Sequential**, used for (numerical) data that are **ordered** from low to high
  – **Qualitative**, used for data that are **not ordered** (factors or categorical data which just have different values)
  – **Diverging**, used for data that **deviate** from something (for example, the deviation from the mean)
  – Used by smoothScatter, useful if you have to make a scatterplot of a lot of different points, and want to see high-density regions of the plot.

- Alternative to smoothScatter: use **scatterplot with transparency** to clarify plots with many points

  rgb() produce any color via red, green and blue proportions

  – Color **transparency** can be added via the alpha (4î{th}) parameter

```
plot(x, y, col = rgb(0, 0, 0, 0.2), pch = 19)
```

- colorspace package can be used for a different control over colors

18. Simulation

- Functions for **probability distributions**

  rnorm(n, mean, sd) generate n **normal** random variates

  runif(n) generate n **uniform** random variates

rpois(n, lambda) generate `n` **Poisson** random variates with a given rate

rbinom(n, size, prob) generate binary data

rexp

rgamma

- Associated functions

  r* generate **random** numbers (draw samples)

  d* evaluate the probability **density**

  p* evaluate the **cumulative distribution** function

  ```
  ppois(4, 2) # Pr(x <= 4)
  ppois(6, 2) # Pr(x <= 6)
  ```

  q* evaluate the **quantile** function

- Set the random number **seed** is critical for **reproducibility**

  - **Always set it when conducting a simulation!**
  - Problem with some sequence of numbers

  ```
  set.seed(1)
  ```

- Linear model

  y = β_{0} + β_{1} x + ϵ

```
x <- rnorm(100) # predictor
e <- rnorm(100, 0, 2) # noise
```

- **Sample randomly** from a specified vector of scalar objects

  ```
  sample(1:10, 4)
  sample(1:10) # permutation
  sample(1:10, replace = TRUE) # sample with replacement (I can get repeats)
  ```

  Sample can draw according to a set of probabilities

19. Regular Expressions

19.1 Metacharacters

^ start of a line (or negation of a character class)

$ end of a line

[ ] character classes

. any character

| or

( ) subexpressions (alternatives or “remember” matched text)

? optional

**** "escape" the metacharacter

+ any number, including none

***** any number, at least one – it is "greedy" so it always matches the *longest* possible string that satisfies the regular expression

The greediness can be turned off with the ? metacharacter (make the regex "lazy"), as in

```
        ^s(.*?)s$
```

{ } interval quantifiers (minimum, maximum)

19.2 Regular expressions in R

For the moment, we assume pattern matching on **ASCII strings**. . .

These functions can take vector arguments.

grep return the **indices** into the character vector where the regex pattern matches (it won't do anything)

```
i <- grep("regexp", vector) # get a set of indices
j <- grep("anotherregexp", vector) # get another set of indices
setdiff(i, j) # substract j elements from i when they are the same
setdiff(j, i) # different result
```

Setting `value = TRUE` returns the **actual elements** of the character vector that match.

grepl return a **logical vector** (`TRUE/FALSE`) indicating which element matches (used for *subsetting*)

regexpr return the indices (integer vector) of the string **where the first match begins** and the length of the match

If I want to see what the match is, use `substr`:

```
regexpr("...", string)
substr(string, start + length - 1)
```

Useful in conjunction with `regmatches` which extracts the matches in the strings without having you to use `substr`.

```
r <- regexpr("...", string)
regmatches(string, r)
```

gregexpr return **all of the matches** ("global") in a given string

sub replace **the first match** with another string (used to strip out stuff by replacing the match with nothing)

gsub replace **all of the matches** with another string

regexec give the indices (list) for parenthesized sub-expressions

If I want to see what the submatch is:

```
r <- regexpr("... (.*) ...", string) # find all the date fields
m <- regmatches(string, r) # parse out
dates <- sapply(m, function (x) x[2]) # extract the 2nd element of each list obje
dates <- as.Date(dates, "%B %d, %Y") # convert to date
hist(dates, "month", freq = TRUE) # aggregate the dates by month and give an hist
```

20. Baltimore Homicide Dataset

http://data.baltimoresun.com/homicides/

21. Classes and methods

- R is both interactive *and* has a system for object orientation (OO)
- The code for implementing S4 classes/methods is in the `methods` package (usually loaded by default)
- A *class* (defined using `setClass()`)is a description of a thing (= new data type); you can customize the methods for that class (output of `print`, `str`, `summary`, `plot`, ...)
- An *object* (created using `new()`) is an instance of a class
- A *method* is a function that only operates on a certain class of objects
- A generic function is an R function which dispatches methods; it does not actually do any computation

  The first argument for any generic function is an object of a particular class.

  1. Search for a method designed just for that class
  2. Search for a default method
  3. Throw an error

  You should **never** call methods directly. Rather, use (the abstraction of) the generic function and let the method be dispatched automatically.
- A *method* is the implementation of a generic function for an object of a particular class

  ```
  methods("mean")
  ```

- S4 equivalent of (S3 generic) `print` function is `show`

showMethods() List the different methods for a generic function

getMethod() Obtain the code (function body) of an S4 method

- Extend the R system:
  - Write a method for a new class but for an existing generic function
  - Write a new generic functions and new methods
- Why would you want to create a new class?

  - To represent new types of data
  - New concepts/ideas
  - To abstratc/hide implementation details from the user
- Create a new class using the `setClass` function
  - name
  - data elements (*slots*)
- Create a method for this class with the `setMethod` function
  - name of the generic name function (`plot,...`)
  - signature (= classes of objects accepted by this method)

showClass Obtain information about a class definition

22. Question

22.1 Question 5

```
data <- read.csv("ss06hid.csv")
ok <- complete.cases(data$ACR,data$AGS)
okdata <- data[ok,]
okdata[okdata$BDS==3 & okdata$RMS==4,2]
```

22.2 Question 7

Create a logical vector that identifies the households on greater than 10 acres who sold more than $10,000 worth of agriculture products. Assign that logical vector to the variable agricultureLogical. Apply the which() function like this to identify the rows of the data frame where the logical vector is TRUE and assign it to the variable indexes.

```
indexes =  which(agricultureLogical)
```

If your data frame for the complete data is called dataFrame you can create a data frame with only the above subset with the command:

```
subsetDataFrame  = dataFrame[indexes,]
```

Note that we are subsetting this way because the NA values in the variables will cause problems if you subset directly with the logical statement.