Programming in R
A Short Introduction

Thomas Girke

December 5, 2014

Outline

Why Programming in R?

- Complete statistical environment and programming language
- Reproducible research
- Efficient data structures make programming very easy
- Ease of implementing custom functions
- Powerful graphics
- Access to fast growing number of analysis packages
- Most widely used language in bioinformatics
- Is standard for data mining and biostatistical analysis
- Technical advantages: free, open-source, available for all OSs

Outline

Overview of Important Operators

- Comparison operators
  - `==` (equal)
  - `!=` (not equal)
  - `>` (greater than)
  - `>=` (greater than or equal)
  - `<` (less than)
  - `<=` (less than or equal)
- Logical operators
  - `&` (and)
  - `|` (or)
  - `!` (not)

Conditional Executions: `if` statements

An `if` statement operates on length-one logical vectors.

Syntax

```
> if(TRUE) {
+         statements_1
+ } else {
+         statements_2
+ }
```

Example

```
> if(1==0) {
+         print(1)
+ } else {
+         print(2)
+ }

[1] 2
```

Conditional Executions: `ifelse` Statements

The `ifelse` statement operates on vectors.

Syntax

```
> ifelse(test, true_value, false_value)
```

Example

```
> x <- 1:10
> ifelse(x<5, x, 0)

 [1] 1 2 3 4 0 0 0 0 0 0
```

Outline

for Loops

Iterates over elements of a looping vector.

Syntax

```
> for(variable in sequence) {
+         statements
+ }
```

Example

```
> mydf <- iris
> myve <- NULL
> for(i in seq(along=mydf[,1])) {
+         myve <- c(myve, mean(as.numeric(mydf[i,1:3])))
+ }
> myve[1:8]
```

```
[1] 3.333333 3.100000 3.066667 3.066667 3.333333 3.666667 3.133333 3.300000
```

Inject into objecs is much faster than append approach with c, cbind, etc.

```
> myve <- numeric(length(mydf[,1]))
> for(i in seq(along=myve)) {
+         myve[i] <- mean(as.numeric(mydf[i,1:3]))
+ }
> myve[1:8]
```

```
[1] 3.333333 3.100000 3.066667 3.066667 3.333333 3.666667 3.133333 3.300000
```

Conditional Stop of Loops

The stop function can be used to break out of a loop (or a function) when a condition becomes TRUE and print an error message.

Example

```
> x <- 1:10
> z <- NULL
> for(i in seq(along=x)) {
+         if(x[i] < 5) {
+                 z <- c(z, x[i]-1)
+         } else {
+                 stop("values need to be <5")
+         }
+ }
```

while Loops

Iterates as long as a condition is true.

Syntax

```
> while(condition) {
+         statements
+ }
```

Example

```
> z <- 0
> while(z<5) {
+         z <- z + 2
+         print(z)
+ }
```

```
[1] 2
[1] 4
[1] 6
```

The apply Function Family: apply

Syntax

```
> apply(X, MARGIN, FUN, ARGs)
```

Arguments

- `X`: `array, matrix or data.frame`
- `MARGIN`: 1 for rows, 2 for columns
- `FUN`: one or more functions
- `ARGs`: possible arguments for functions

Example

```
> apply(iris[1:8,1:3], 1, mean)

       1        2        3        4        5        6        7        8 
3.333333 3.100000 3.066667 3.066667 3.333333 3.666667 3.133333 3.300000 
```

The apply Function Family: tapply

Applies a function to vector components that are defined by a factor.

Syntax

```
> tapply(vector, factor, FUN)
```

Example

```
> iris[1:2,]
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
```

```
> tapply(iris$Sepal.Length, iris$Species, mean)
```

```
    setosa versicolor  virginica
     5.006      5.936      6.588
```

The apply Function Family: sapply and lapply

Both apply a function to vector or list objects. The function `lapply` always returns a list object, while `sapply` tries to return vector or matrix objects when this is possible.

Examples

```
> x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
> lapply(x, mean)

$a
[1] 5.5

$beta
[1] 4.535125

$logic
[1] 0.5

> sapply(x, mean)

       a     beta    logic
5.500000 4.535125 0.500000
```

Often used in combination with a function definition

```
> lapply(names(x), function(x) mean(x))
> sapply(names(x), function(x) mean(x))
```

Outline

Function Overview

A very useful feature of the R environment is the possibility to expand existing functions and to easily write custom functions. In fact, most of the R software can be viewed as a series of R functions.

Syntax to define functions

```
> myfct <- function(arg1, arg2, ...) {
+         function_body
+ }
```

Syntax to call functions

```
> myfct(arg1=..., arg2=...)
```

Function Syntax Rules

General — Functions are defined by (1) assignment with the keyword `function`, (2) the declaration of arguments/variables (`arg1, arg2, ...`) and (3) the definition of operations (`function_body`) that perform computations on the provided arguments. A function name needs to be assigned to call the function.

Naming — Function names can be almost anything. However, the usage of names of existing functions should be avoided.

Arguments — It is often useful to provide default values for arguments (e.g.: `arg1=1:10`). This way they don't need to be provided in a function call. The argument list can also be left empty (`myfct <- function()  fct_body `) when a function is expected to return always the same value(s). The argument '...' can be used to allow one function to pass on argument settings to another.

Body — The actual expressions (commands/operations) are defined in the function body which should be enclosed by braces. The individual commands are separated by semicolons or new lines (preferred).

Usage — Functions are called by their name followed by parentheses containing possible argument names. Empty parenthesis after the function name will result in an error message when a function requires certain arguments to be provided by the user. The function name alone will print the definition of a function.

Scope — Variables created inside a function exist only for the life time of a function. Thus, they are not accessible outside of the function. To force variables in functions to exist globally, one can use the double assignment operator: '$<<-$'.

Function: Examples

Define sample function

```
> myfct <- function(x1, x2=5) {
+         z1 <- x1 / x1
+         z2 <- x2 * x2
+         myvec <- c(z1, z2)
+         return(myvec)
+ }
```

Function usage

```
> ## Apply function to values 2 and 5
> myfct(x1=2, x2=5)

[1]  1 25

> ## Run without argument names
> myfct(2, 5)

[1]  1 25

> ## Makes use of default value '5'
> myfct(x1=2)

[1]  1 25

> ## Print function definition
> # myfct
```

Outline

Debugging Utilities

Several debugging utilities are available for R. They include:

- `traceback`
- `browser`
- `options(error=recover)`
- `options(error=NULL)`
- `debug`

The Debugging in R page Link provides an overview of the available resources.

Regular Expressions

R's regular expression utilities work similar as in other languages. To learn how to use them in R, one can consult the main help page on this topic with `?regexp`.

```
> ## The grep function can be used for finding patterns in strings, here letter
> ## 'A' in vector 'month.name'.
> month.name[grep("A", month.name)]

[1] "April"  "August"

> ## Example for using regular expressions to substitute a pattern by another
> ## one using a back reference. Remember: single escapes '\' need to be double
> ## escaped '\\' in R.
> gsub('(i.*a)', 'xxx_\\1', "virginica", perl = TRUE)

[1] "vxxx_irginica"
```

Interpreting a Character String as Expression

Some useful examples

```
> ## Generates vector of object names in session
> mylist <- ls()
> ## Prints name of 1st entry
> mylist[1]

[1] "i"

> ## Executes 1st entry as expression
> get(mylist[1])

[1] 150

> # Alternative approach
> eval(parse(text=mylist[1]))

[1] 150
```

Replacement, Split and Paste Functions for Strings

Selected examples

```
> ## Substitution with back reference which inserts in this example
> ## an '_' character
> x <- gsub("(a)","\\1_", month.name[1], perl=T)
> x

[1] "Ja_nua_ry"

> ## Split string on inserted character from above
> strsplit(x,"_")

[[1]]
[1] "Ja"  "nua" "ry"

> ## Reverse a character string by splitting first all characters
> ## into vector fields
> paste(rev(unlist(strsplit(x, NULL))), collapse="")

[1] "yr_aun_aJ"
```

Time, Date and Sleep

Selected examples

```
> ## Returns CPU (and other) times that an expression used (here ls)
> system.time(ls())

   user  system elapsed 
      0       0       0 

> ## Return the current system date and time
> date()

[1] "Fri Dec  5 18:04:17 2014"

> ## Pause execution of R expressions for a given number of
> ## seconds (e.g. in loop)
> Sys.sleep(1)
```

Import of Specific File Lines with Regular Expression

The following example demonstrates the retrieval of specific lines from an external file with a regular expression. First, an external file is created with the `cat` function, all lines of this file are imported into a vector with `readLines`, the specific elements (lines) are then retieved with the `grep` function, and the resulting lines are split into vector fields with `strsplit`.

```
> cat(month.name, file="zzz.txt", sep="\n")
> x <- readLines("zzz.txt")
> x[1:6]

[1] "January"  "February" "March"    "April"    "May"      "June"

> x <- x[c(grep("^J", as.character(x), perl = TRUE))]
> t(as.data.frame(strsplit(x, "u")))

                 [,1]  [,2]
c..Jan....ary.. "Jan" "ary"
c..J....ne..    "J"   "ne"
c..J....ly..    "J"   "ly"
```

Outline

Run External Command-line Software

Example for running blastall from R

```
> system("blastall -p blastp -i seq.fasta -d uniprot -o seq.blastp")
```

Outline

Options to Execute R Scripts

Executing R scripts from the R console

```
> source("my_script.R")
```

Execute an R script from command-line

```
Rscript my_script.R # or just ./myscript.R after making it executable
R CMD BATCH my_script.R # Alternative way 1
R --slave < my_script.R # Alternative way 2
```

Passing command-line arguments to R programs. In the given example the number 10 is passed on from the command-line as an argument to the R script which is used to return to STDOUT the first 10 rows of the iris sample data. If several arguments are provided, they will be interpreted as one string and need to be split in R with the strsplit function. A more detailed example can be found here: Link

```
## Create R script named 'test.R'
##################################
myarg <- commandArgs()
print(iris[1:myarg, ])
##################################

## Then run it from the command-line
Rscript test.R 10
```

Outline

Short Overview of Package Building Process

Automatic package building with the package.skeleton function. The given example will create a directory named `mypackage` containing the skeleton of the package for all functions, methods and classes defined in the R script(s) passed on to the `code_files` argument. The basic structure of the package directory is described here: Link. The package directory will also contain a file named `Read-and-delete-me` with instructions for completing the package:

```
> package.skeleton(name="mypackage", code_files=c("script1.R", "script2.R"))
```

Once a package skeleton is available one can build the package from the command-line (Linux/OS X). This will create a tarball of the package with its version number encoded in the file name. Subequently, the package tarball needs to be checked for errors with:

```
R CMD build mypackage
R CMD check mypackage_1.0.tar.gz
```

Install package from source

```
> install.packages("mypackage_1.0.tar.gz", repos=NULL)
```

For more details see here: Link

Outline

Exercise 1: for loop

Task 1.1: Compute the mean of each row in `myMA` by applying the mean function in a `for` loop

```
> myMA <- matrix(rnorm(500), 100, 5, dimnames=list(1:100, paste("C", 1:5, sep=""
> myve_for <- NULL
> for(i in seq(along=myMA[,1])) {
+         myve_for <- c(myve_for, mean(as.numeric(myMA[i, ])))
+ }
> myResult <- cbind(myMA, mean_for=myve_for)
> myResult[1:4, ]
```

```
          C1           C2         C3         C4         C5   mean_for
1 -0.6592941  2.352441345 -0.1456537  2.4572565  1.0862656 1.01820312
2  0.5287459  1.340274328  0.3276844  0.3755140 -2.1750698 0.07942976
3  0.2635144  0.261592693  0.3927853  0.9317759  0.1823045 0.40639458
4  0.6070491 -0.008121598  0.4753948 -1.3174256  0.6644683 0.08427300
```

Exercise 1: while loop

Task 1.2: Compute the mean of each row in myMA by applying the mean function in a while loop

```
> z <- 1
> myve_while <- NULL
> while(z <= length(myMA[,1])) {
+         myve_while <- c(myve_while, mean(as.numeric(myMA[z, ])))
+         z <- z + 1
+ }
> myResult <- cbind(myMA, mean_for=myve_for, mean_while=myve_while)
> myResult[1:4, -c(1,2)]
```

```
          C3         C4         C5   mean_for mean_while
1 -0.1456537  2.4572565  1.0862656 1.01820312 1.01820312
2  0.3276844  0.3755140 -2.1750698 0.07942976 0.07942976
3  0.3927853  0.9317759  0.1823045 0.40639458 0.40639458
4  0.4753948 -1.3174256  0.6644683 0.08427300 0.08427300
```

Task 1.3: Confirm that the results from both mean calculations are identical

```
> all(myResult[,6] == myResult[,7])
```

```
[1] TRUE
```

Exercise 1: apply loop and avoiding loops in R

Task 1.4: Compute the mean of each row in myMA by applying the mean function in an apply loop

```
> myve_apply <- apply(myMA, 1, mean)
> myResult <- cbind(myMA, mean_for=myve_for, mean_while=myve_while, mean_apply=m
> myResult[1:4, -c(1,2)]

         C3         C4         C5  mean_for mean_while mean_apply
1 -0.1456537  2.4572565  1.0862656 1.01820312 1.01820312 1.01820312
2  0.3276844  0.3755140 -2.1750698 0.07942976 0.07942976 0.07942976
3  0.3927853  0.9317759  0.1823045 0.40639458 0.40639458 0.40639458
4  0.4753948 -1.3174256  0.6644683 0.08427300 0.08427300 0.08427300
```

Task 1.5: When operating on large data sets it is much faster to use the rowMeans function

```
> mymean <- rowMeans(myMA)
> myResult <- cbind(myMA, mean_for=myve_for, mean_while=myve_while, mean_apply=m
> myResult[1:4, -c(1,2,3)]

         C4         C5   mean_for mean_while mean_apply   mean_int
1  2.4572565  1.0862656 1.01820312 1.01820312 1.01820312 1.01820312
2  0.3755140 -2.1750698 0.07942976 0.07942976 0.07942976 0.07942976
3  0.9317759  0.1823045 0.40639458 0.40639458 0.40639458 0.40639458
4 -1.3174256  0.6644683 0.08427300 0.08427300 0.08427300 0.08427300
```

Exercise 2: functions

Task 2.1: Use the following code as basis to implement a function that allows the user to compute the mean for any combination of columns in a matrix or data frame. The first argument of this function should specify the input data set, the second the mathematical function to be passed on (e.g. mean, sd, max) and the third one should allow the selection of the columns by providing a grouping vector.

```
> myMA <- matrix(rnorm(100000), 10000, 10, dimnames=list(1:10000, paste("C", 1:1
> myMA[1:2,]

          C1         C2         C3         C4          C5         C6         C7
1 -1.0031265 -0.5716986 -0.5018299 -1.1932212  0.02666608  0.4149434 -0.5134161
2  0.5016274  0.5126865  2.4037213 -0.9751479 -2.94053440 -0.2629856 -0.5291323

> myList <- tapply(colnames(myMA), c(1,1,1,2,2,2,3,3,4,4), list)
> names(myList) <- sapply(myList, paste, collapse="_")
> myMAmean <- sapply(myList, function(x) apply(myMA[, x, drop=FALSE], 1, mean))
> myMAmean[1:4,]

    C1_C2_C3    C4_C5_C6       C7_C8     C9_C10
1 -0.6922183 -0.25053727  0.04269614 -0.1394368
2  1.1393451 -1.39288929 -0.49985455 -0.8217462
3 -0.5003290 -0.71385017 -1.73730820 -0.3918644
4 -0.2499292 -0.08088524  0.31853612  0.5770666
```

Exercise 3: nested loops to generate similarity matrices

Task 3.1: Create a sample list populated with character vectors of different lengths

```
> setlist <- lapply(11:30, function(x) sample(letters, x, replace=TRUE))
> names(setlist) <- paste("S", seq(along=setlist), sep="")
> setlist[1:6]

$S1
 [1] "s" "q" "w" "z" "g" "d" "o" "n" "e" "w" "b"

$S2
 [1] "q" "d" "g" "o" "y" "n" "z" "c" "d" "f" "i" "w"

$S3
 [1] "o" "b" "e" "s" "l" "p" "p" "t" "v" "w" "z" "h" "p"

$S4
 [1] "h" "j" "h" "v" "j" "u" "j" "o" "o" "x" "n" "h" "a" "j"

$S5
 [1] "m" "s" "c" "z" "h" "h" "w" "l" "n" "t" "j" "g" "q" "e" "t"

$S6
 [1] "b" "m" "q" "i" "y" "l" "y" "r" "a" "l" "a" "m" "m" "i" "v" "p"
```

Exercise 3: nested loops to generate similarity matrices

Task 3.2: Compute the length for all pairwise intersects of the vectors stored in `setlist`. The intersects can be determined with the `%in%` function like this: `sum(setlist[[1]] %in% setlist[[2]])`

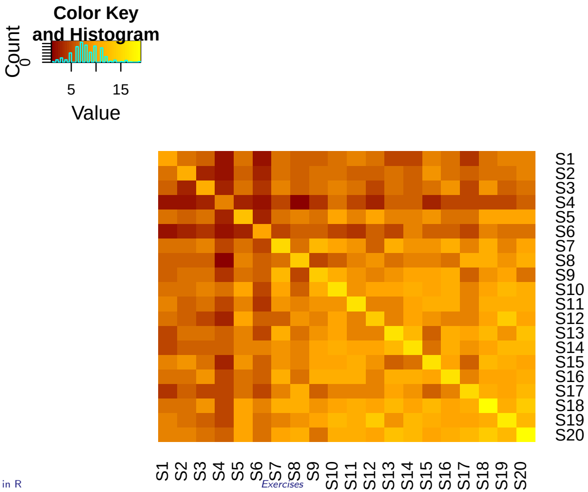
```
> setlist <- sapply(setlist, unique)
> olMA <- sapply(names(setlist), function(x) sapply(names(setlist),
+                function(y) sum(setlist[[x]] %in% setlist[[y]])))
> olMA[1:12,]
```

```
    S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 S14 S15 S16 S17 S18 S19 S20
S1  10  7  6  2  7  2  7  6  6   7   8   7   5   5   8   7   4   7   8   8
S2   7 11  3  2  6  3  7  6  7   7   6   6   7   6   9   7   6   7   7   8
S3   6  3 11  3  7  4  8  6  7   8   7   5   7   6   7   9   5   9   6   7
S4   2  2  3  8  3  2  5  1  4   7   5   3   6   6   3   5   5   5   5   6
S5   7  6  7  3 13  3  7  8  7  10   8  10   8   8   9   7   7  10  10  10
S6   2  3  4  2  3 10  5  6  6   5   4   6   5   8   6   6   5   8   7   7
S7   7  7  8  5  7  5 15  7 12  10   9   6  11   9   9  11   8  11   8  10
S8   6  6  6  1  8  6  7 14  5   6   8   9   7   8   8   7  11  11   9  11
S9   6  7  7  4  7  6 12  5 14  11   9   8   9  10  10  11   6   9  10   7
S10  7  7  8  7 10  5 10  6 11  16   9  10  10  11  10  11   8  10  12  11
S11  8  6  7  5  8  4  9  8  9   9  16   8   8  10  11  11   8  11  11  11
S12  7  6  5  3 10  6  6  9  8  10   8  14   8  10   9   8   8  10  14  10
```

Exercise 3: nested loops to generate similarity matrices

Task 3.3: Plot the resulting intersect matrix as heat map. The `heatmap.2` function from the `gplots` library can be used for this.

```
> library("gplots")
> heatmap.2(olMA, trace="none", Colv="none", Rowv="none", dendrogram="none",
+            col=colorpanel(40, "darkred", "orange", "yellow"))
```



Exercise 4: build your own R package

Task 4.1: Save one or more of your functions to a file called `script.R` and build the package with the `package.skeleton` function.

```
> package.skeleton(name="mypackage", code_files=c("script1.R"))
```

Task 4.2: Build tarball, install and use package

```
> system("R CMD build mypackage") # or from command-line: R CMD build mypackage
> install.packages("mypackage_1.0.tar.gz", repos=NULL, type="source")
> library(mypackage)
> ?myMAcomp # Opens help for function defined by mypackage
```

Additional Exercises

See here: Link

Outline

What are Git and GitHub?

- Git is a distributed version control system similar to SVN.
- GitHub is an online social coding service based on Git.

Installing Git

- Install Link on Windows, OS X and Linux
- When using it from RStudio, it needs to find the Git executable

Git Basics from Command-Line

- Finding help from command-line
  ```
  git <command> -help
  ```
- Initialize a directory as a Git repository
  ```
  git init
  ```
- Add files to Git repository (staging area)
  ```
  git add myfile
  ```
- After editing file(s) in your repos, record a snapshot of the staging area
  ```
  git commit -am "some edits"
  ```

Using GitHub from RStudio

- After installing Git, set path to Git executable in Rstudio:
  `Tools > Global Options > Git/SVN`
- If needed, login to GitHub account and create repository. Use option 'Initialize this repository with a README'.
- Clone repository by copying & pasting URL from repository into RStudio's 'Clone Git Repository' window:
  `Project (triangle on top right) > New Project >`
  `Version Control > Git > Provide URL`
- Now do some work (*e.g.* add an R script), commit and push changes as follows:
  `Tools > Version Control > Commit`
- Check files in staging area and press `Commit Button`
- To commit changes to GitHub, press `Push Button`
- Shortcuts to automate above routines Link

GitHub Education

- GitHub Education Link just became available. It provides now free private repositories for students and faculty

Outline

What Is $\LaTeX$?

- Originally developed in the early 1980s by Leslie Lamport.
- $\LaTeX$ is a document markup language and document preparation system for the TeX typesetting program.
- Developed for mathematicians, statisticians, engineers and computer scientists.
- High quality of typesetting for scientific articles.
- Programmable environment.
- Many efficient cross-referencing facilities for equations, tables, figures, bibliographies, etc.

How Does It Work?

- Write in your favourite text editor, *e.g.*: Vim or Emacs.
- Install LaTeX distribution for your OS
  - Windows: MiKTeX
  - Linux: Latex Project Site
  - Mac OS X: TexShop
- LaTeX manuals (very incomplete selection)
  - List of Manuals on Latex Project Site
  - The Not So Short Introduction to LaTeX
  - Getting to grips with LaTeX
- Packages
  - An almost complete list: Online TeX Catalogue

Outline

Bibtex: The Ultimate Reference Management Tool

- Sample Latex file: `example.tex`
- Convert to PDF with command: `pdflatex example.tex`
- To include references from MyBibTex.bib database, the following command sequence needs to be executed: `pdflatex example.tex; bibtex example; pdflatex example.tex`

Examples of BibTex Citations

- Citation in parentheses (Grant et al., 2006; Schwacke et al., 2003; Miteva et al., 2006)
- Citation of Grant et al. (2006); Schwacke et al. (2003); Miteva et al. (2006)
- Extended citation (Grant et al., 2006, J Chem Inf Model, 46, 1912-1918)
- Footnote citation with more detail (Grant et al., 2006)[1]
- ...

The reference list for these citations appears automatically at a defined position, here the end of the document.

[1](Grant et al., 2006, J Chem Inf Model)

Outline

R's `Sweave` Function Integrates R with Latex

- 'Sweave' provides a flexible framework for mixing Latex and R code for automatic generation of scientific documents.
- It does this by identifying R code chunks - starting with <<>>= and ending with @ - and replaces them with the corresponding R output in LaTeX format, e.g. commands, data objects, plots.
- The user organizes the hybrid code in a *.Rnw file, while the `Sweave()` function converts this file into a typical Latex *.tex file.
- A quick learning exercise:
  - Download sample hybrid file `Sweave-test-1.Rnw`
  - Run in R command `Sweave("Sweave-test-1.Rnw")`
  - Convert generated `Sweave-test-1.tex` to PDF with `pdflatex Sweave-test-1.tex`
- Sweave User Manual

Outline

Structure of *.Rnw Hybrid File

Latex \Sexpr{*pi*} Latex Latex Latex Latex Latex Latex
Latex Latex Latex Latex Latex Latex Latex Latex Latex
Latex Latex Latex Latex Latex Latex Latex Latex Latex
<<>>=
R R R R R R R R R R R R R R R R R R R R R R R R
R R R R R R R R R R R R R R R R R R R R R R R R
@
Latex Latex Latex Latex Latex Latex Latex Latex Latex
Latex Latex Latex Latex Latex Latex Latex Latex Latex
Latex Latex Latex Latex Latex Latex Latex Latex Latex
<<>>=
R R R R R R R R R R R R R R R R R R R R R R R R
R R R R R R R R R R R R R R R R R R R R R R R R
@

Convert *.Rnw to *.tex to *.pdf

1. Create *.tex file

   ```
   > Sweave("mydoc.Rnw")
   ```

2. Create R source file with code chunks (optional)

   ```
   > Stangle("mydoc.Rnw")
   ```

3. Gernerate PDF with bibliography

   ```
   > system("pdflatex mydoc.tex; bibtex mydoc; pdflatex mydoc.tex")
   ```

Code Chunk Options

Important options that can be included in code chunk start tag:

$<<>>=$

label:	optional name for code chunk.
echo:	shows command if set to TRUE. Default is TRUE.
fig:	shows plots automatically if set to TRUE. Alternatively, one can use standard R code to save graphics to files and point to them in Latex source code.
eval:	if FALSE, the variables and functions in code chunk are not evaluated.
prefix:	if TRUE, generated file names have a common prefix.

Printing R Commands and Output

Beamer Slide Setting

```
\begin{frame}[containsverbatim]
```

R Code Chunks

```
> <<echo=TRUE>>=
> 1:10
> @
```

Result in PDF

```
> 1:10

 [1]  1  2  3  4  5  6  7  8  9 10
```

Including Tables with `xtable()`

Code Chunk

```
<<echo=FALSE>>=
library(xtable)
xtable(iris[1:4,])
@
```

Result in PDF

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.10	3.50	1.40	0.20	setosa
2	4.90	3.00	1.40	0.20	setosa
3	4.70	3.20	1.30	0.20	setosa
4	4.60	3.10	1.50	0.20	setosa

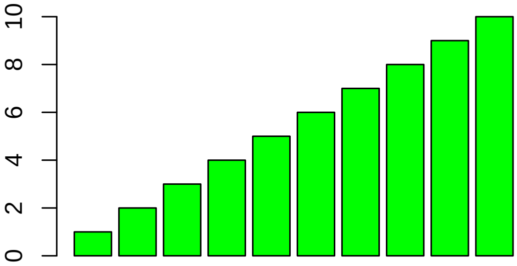
Including Graphics

Code Chunk

```
<<fig=true, width=4.5, height=3.5, eval=TRUE, echo=TRUE>>=
 barplot(1:10, beside=TRUE, col="green")
@
```

Result in PDF

```
> barplot(1:10, beside=TRUE, col="green")
```



Session Information

```
> sessionInfo()

R version 3.1.2 (2014-10-31)
Platform: x86_64-unknown-linux-gnu (64-bit)

locale:
[1] C

attached base packages:
[1] stats     graphics  utils     datasets  grDevices
[6] methods   base

other attached packages:
[1] xtable_1.7-4  gplots_2.14.2

loaded via a namespace (and not attached):
[1] KernSmooth_2.23-13 bitops_1.0-6       caTools_1.17.1
[4] gdata_2.13.3       gtools_3.4.1       tools_3.1.2
```

Bibliography: to Demo BibTeX I

Grant, J. A., Haigh, J. A., Pickup, B. T., Nicholls, A., Sayle, R. A., Sep-Oct 2006. Lingos, finite state machines, and fast similarity searching. J Chem Inf Model 46 (5), 1912–1918.
URL `http://www.hubmed.org/display.cgi?uids=16995721`

Miteva, M. A., Violas, S., Montes, M., Gomez, D., Tuffery, P., Villoutreix, B. O., Jul 2006. FAF-Drugs: free ADME/tox filtering of compound collections. Nucleic Acids Res 34 (Web Server issue), 738–744.
URL `http://www.hubmed.org/display.cgi?uids=16845110`

Schwacke, R., Schneider, A., van der Graaff, E., Fischer, K., Catoni, E., Desimone, M., Frommer, W. B., Flügge, U. I., Kunze, R., Jan 2003. ARAMEMNON, a novel database for Arabidopsis integral membrane proteins. Plant Physiol 131 (1), 16–26.
URL `http://www.hubmed.org/display.cgi?uids=12529511`