

## Statistical programming in R - class notes.

### Writing functions in R.

Writing functions in R resembles writing functions in python, but with different syntax and rules for returning values.

In R, you define functions using the `function` keyword:

```
make.string = function(v) {
  paste(v,collapse=" ")
}

make.hist = function(v) {
  title = c("Histogram of ",length(v)," values.")
  title = make.string(title)
  h = hist(v,col='lightblue',main=title)
  abline(v=mean(v),col='red',lty=2)
  h
}
```

The return value (if any) is the value returned by the last statement. In this example (see above) the return value is a histogram object returned by `hist` plotting function. You can also use the `return` function: `return(h)`.

### Running a script or loading newly defined (or refined) functions into an R session.

Just as with python, R features an interactive interpreter into which you can type and execute commands. You can also run commands and define variables (functions, or data structures) using the `source` command, which runs all the commands in a given file in the current environment.

```
> source('demo.R')
```

**Question:** What is the equivalent command in python?

**Answer:** `import` and `reload`

### Dealing with strings in R

String processing and manipulation in R may seem awkward after having used python for several months. If you need to perform many complex string operations, you can get around some of these problems by writing most of your code in python and then “call out” to R using a module called `rpy`.

## How to install Rpy2 – November 2009

1) download rpy2 source from

<http://sourceforge.net/projects/rpy/>

2) tar -xvf rpy2-2.0.6.tar.gz

3) from the rpy2 folder you just untarred. Follow these instructions <http://heather.cs.ucdavis.edu/~matloff/rpy2.html>

On the lab computers, Rpy is installed for the python2.5 program but not the default python (2.6) that you get when you launch python by typing “python” in the Terminal.

To demo rpy on the lab computers, open a Terminal window and type python2.5. From inside the python interpreter, try this:

```
>>> x = range(0,11)
>>> y = [i**2 for i in x]
>>> from rpy import r
```

And see <http://www.daimi.au.dk/~besen/TBiB2007/lecture-notes/rpy.html> Tool-building in Bioinformatics using rpy.

Now let's look at how R handles strings:

Most of the time, when you write functions, you will be writing them to produce graphs and plots for reports and/or publications. In other instances, you'll be writing functions to perform some complex data analysis steps on a data set, such as a collection of microarrays. In each case, you will probably need to manipulate strings in order to produce attractive or meaningful output.

**Class Exercise:** Review the first two functions defined in the demo program demo.R in the class directory of the class subversion repository. And read the help documentation for paste.

**Question:** What is paste doing here? What would the equivalent code look like in python?

**Question:** What command would produce 'a|b|c' from character vector returned by c('a','b','c')?

The function `strsplit` performs the opposite function. It accepts a character vector and a regular expression and splits the components into new vectors, return each of these as an element of a list:

```

> v = c('a','b','c')
> paste(v,collapse='|')
[1] "a|b|c"
> s = paste(v,collapse='|')[1]
> s
[1] "a|b|c"
> strsplit(s)
> strsplit(s,split='|') # doesn't give the expected behavior!
[[1]]
[1] "a" "|" "b" "|" "c"

> ?strsplit # let's find out why not

```

## Examples

```

noquote(strsplit("A text I want to display with spaces", NULL)[[1]])

x <- c(as = "asfef", qu = "qwerty", "yuiop[", "b", "stuff.blah.yech")
# split x on the letter e
strsplit(x,"e")

unlist(strsplit("a.b.c", "."))
## [1] "" "" "" "" ""
## Note that 'split' is a regexp!
## If you really want to split on '.', use
unlist(strsplit("a.b.c", "\\."))
## [1] "a" "b" "c"

## a useful function: rev() for strings
strReverse <- function(x)
  sapply(lapply(strsplit(x,NULL), rev), paste, collapse="")
strReverse(c("abc", "Statistics"))

## get the first names of the members of R-core
a <- readLines(file.path(R.home(),"AUTHORS"))[-(1:8)]
a <- a[(0:2)-length(a)]
(a <- sub(" .*", "", a))
# and reverse them
strReverse(a)

```

**Question:** Which line(s) explain the problem?

**Answer:** See where the example says: “Note that ‘split’ is a regexp! If you really want to split on ‘.’, use ... “ and “escape” the pipe character, which also happens to be a regular expression metacharacter.

Also, note that `strsplit` returns a list. To retrieve the output as a character vector, we need to “unlist” the list or just take its first element.

```
> strsplit(s,split='\\|')
[[1]]
[1] "a" "b" "c"

> strsplit(s,split='\\|')[[1]]
[1] "a" "b" "c"
```

## Control statements in R

In R, as with other languages, we have “if” statements, “for” loops, etc.

For example, an “if” statement:

```
make.hist2 = function(v,fn=NA) {
  title = c("Histogram of ",length(v)," values.")
  title = make.string(title)
  if (!is.na(fn)) {
    print(paste(c("Okay. I'll make a file called: ",fn),collapse=" "))
    pdf(fn)
  }
  else {
    print("I'll show the plot instead of making a file.")
  }
  h = hist(v,col="lightblue",main=title)
  abline(v=mean(v),col='red',lty=2)
  if (!is.na(fn)) {
    dev.off()
  }
  h
}
```

Note in R, we have a lot of extra characters to type to define blocks of code – the curly braces, parentheses, etc.

## A “for” loop in R:

for (*variable in sequence*) *expression*

The *expression* can be a single R command - or several lines of commands wrapped in curly brackets:

for (*variable in sequence*) { *expression expression expression* }

```
> for (i in seq(1,10)) print(i**2)
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

```
[1] 36
[1] 49
[1] 64
[1] 81
[1] 100
```

## Data structures in R

In R we have two different list-like structures. Vectors: contain values of the same type and lists can contain a mix of types.

**Question:** How do you create a new, empty vector in R?

**Answer:** `v = vector()`

**Question:** How do you create a new, empty list in R?

**Answer:** `l = list()`

A vector can contain elements of a single type, while a list can contain elements of different types:

```
> l = list()
> l[[1]]='hello'
> l
[[1]]
[1] "hello"

> l[[2]]=seq(0,11,by=2)
> l
[[1]]
[1] "hello"

[[2]]
[1] 0 2 4 6 8 10
```

Most of the statistical tests in R return lists as output.

Each element in a list can receive a name, which you can then use to access these elements. You can add names using the `names` command or when you define the list initially.

For example:

```
> z = list(a=1, b="c", c=1:3)
> z
$a
[1] 1
```

```

$b
[1] "c"

$c
[1] 1 2 3

> names(z)
[1] "a" "b" "c"

# change just the name of the third element.
> names(z)[3] = "c2"
      z

      z <- 1:3
      names(z)
      ## assign just one name
      names(z)[2] <- "b"
      z

```

### Fitting linear model in R:

For this, let's use one of R's built-in data sets: ToothGrowth

```

> ToothGrowth
  len supp dose
1  4.2   VC  0.5
2 11.5   VC  0.5
3  7.3   VC  0.5
4  5.8   VC  0.5
5  6.4   VC  0.5
6 10.0   VC  0.5

```

This data set reports the length of gineua pig's teeth after receiving different doses of vitamin C (VC).

To start the analysis, first we "attach" the data set so that we can refer to data columns by their titles:

```

> attach(ToothGrowth)

```

Next, let's plot dose (which should appear on the x axis) against len (which will appear on the y -axis). (The variable closest to the "plot" command goes on the x axis.)

```

> plot(dose, len)

```

Note there is an obvious relationship between dose of vitamin C and tooth growth in the animals.

This suggests that a linear model is appropriate for these data.

To create the model, we use the `lm` command as follows – to translate into words, this says: “Lengths follows (~) doses” where `len` is the dependent variable and `dose` is the independent variable.

```
> model = lm(len~dose)
```

And to view the results of the modeling, we should add the fitted line to the plot:

```
> abline(model)
```

We can view a summary of the model and how well it fit using the `summary` command, which gives use three tables:

```
> summary(model)

Call:
lm(formula = len ~ dose)

Residuals:
    Min       1Q   Median       3Q      Max
-8.4496 -2.7406 -0.7452  2.8344 10.1139
```

The residuals are the distances between the line (the model) and the individual data points that went into creating the model. The model does a better job of describing the data when the residuals are small.

```
Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)   7.4225     1.2601    5.89 2.06e-07 ***
dose           9.7636     0.9525   10.25 1.23e-14 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The equation of a line is  $y = mx + b$ . The coefficients are just the values of  $m$  (the slope of the line) and  $b$  (the intercept). The model fitting procedure tests the significance of these values. In this case, they are highly significant, that is, the line is very good at explaining the variance in the `len` as a function of `dose`.

The final table gives some statistics about the quality of the regression. Residual standard error is standard error of the residuals – the distances between the points and the line. R-squared is the percentage of variation in  $Y$  that is explained by  $X$ . It is also the square of Pearson’s correlation coefficient. The F-statistic is used to calculate the  $p$  value, which in

this case is very small. The p value is the “probability value” that gives the probability of obtaining a fit as good as this under the assumption that the data are totally random.

```
Residual standard error: 4.601 on 58 degrees of freedom
Multiple R-Squared: 0.6443, Adjusted R-squared: 0.6382
F-statistic: 105.1 on 1 and 58 DF, p-value: 1.233e-14
```

To retrieve componets of the model, use the names command to find out what they are called and the access them via the appropriate name:

```
> names(model)
[1] "coefficients" "residuals" "effects" "rank"
[5] "fitted.values" "assign" "qr" "df.residual"
[9] "xlevels" "call" "terms" "model"
```

Then, to plot the residuals:

```
> plot(model$residuals)
```

## Applying statistical tests in R

*See video link on pair t tests from the Web site.*