A Brief R Introduction

J.M. Ponciano

January 6, 2015

R is a scripting language, originally intended as a convenient environment for statistical analysis, and perform many calculations. It's free and available on Mac, Windows and Linux platforms. I have been programming in R for 15 years now, and I keep learning new things every single time I use it. At the beginning, around the early 2000's, it was easy to keep track with all the developments related to this neat software, and now it is virtually impossible. Yet, all the contributions to this programming language are readily available, conveniently listed and document and easy to find in the main software's web page: `http://www.r-project.org`. Go to this website. To the left, you'll see a list of links under *About R*. Click on the *What is R* link and read their one-page introduction to R. Then, click on the CRAN link below *Download, Packages*, then on one of the US mirror links (I usually pick the Berkeley link: a long time ago, it use to be the most reliable mirror, now they are all the same), and follow the instructions under the "Download and Install" section.

R is not a point-and-click software and all the instructions are typed in the "command" window. Long lists of instructions are stored in `R scripts`. The scripts can be run partially or entirely in the command window, edited and saved. The results of the calculations in the command window can also be saved and later retrieved in another session. We shall carefully go through all of this, one step at a time. For now, I just want to get you started with R. Because R is a language, you need to practice writing and reading often if you want to achieve proficiency. Although R has a steep learning curve, such task is greatly eased if you practice every single day a little bit, and in this course that is exactly what we'll do.

Enough said, let's get started with a simple example. R can do all sorts of operations. Let's start with something simple. In the command window, type

```
> 21+2
```

and then press `enter` and R will answer

```
[1] 23
```

Now if we type `21*2` R computes 21×2, and if we type `> 21*-2`, then R answers

```
[1] -42
```

So R did $21 \times (-2)$. This result shows that R respect the precedence of simple arithmetic operators. It also shows something that I want you to get used to do: I want you to **try**. Don't be afraid of typing "silly" commands: R is a language that is learnt by doing, trial and error. One learns by making mistakes (like in many things in life...).

1 Objects and types of objects

R is an **Object-oriented** programming language: whenever we do any calculation, we can store the results by giving them a name. The results then become an *object* which is retrievable and can be altered in other calculations. This is convenient whenever we are doing complex calculations and we simply don't want to retype all the instructions over and over again. To assign a name to a calculation, we use the `<` sign followed by the `-` sign, which looks like an arrow pointing to the left. In the example above, we could have done

```
> A <- 21
> B <- A+2
> B
[1] 23
```

In what follows I stop using the narrative modality and simply list one-line examples that you can quickly type and follow. So just follow the contents below and type each one of these examples, and see what happens! Also try the exercises below. These exercises are not due, but are there for you to try.

- `R` uses "object oriented programming" and "functional programming". "Object oriented programming" refers to the fact that we will mostly use objects to define (and remember as well as keep track of) the types of calculations we are doing. "Functional programming" refers to the fact that we
- "Objects" include the results of arithmetic operations, vectors, matrices, lists and functions

  A number resulting from an arithmetic operation:

  ```
  > A <- ((21*2) + 45  -  36)/2
  ```

  A vector:

  ```
  > y<-c(38,18,22)
  ```

  . Here the operator `c()` *combines* the numbers $38, 18, 22$ into an array of numbers called a **vector**.

  A list:

  ```
  >  x<-list(ncolors=2,colors=c("red","blue"))
  ```

  A function:

  ```
  > std.dev<-function(x){return(sqrt(var(x)))}
  ```

- To list objects in current data directory, type `objects()` or `ls()`.
  To narrow search, use

  ```
  objects(pattern="y")
  ```

  . This lists all objects whose name have "y".
- Cleaning up by removing: `rm(y)`, or re-use the same names.
- Displaying object content: type object name
- Executing a function object: `functionname()`, and Quitting R `q()`.

2 Vectors

- element types: numeric, logical, and character.

  a. numeric

  ```
  > y <- 18
  > y <- c(-3.6,1.58)
  > y <- 1:9
  > y <- seq(from=0,to=10,by=0.02)     #seq() a function w/ args from, to,..
  > y <- seq(from=0,to=10,by=0.02)     #by argument gives distance
  > y <- seq(0,10,by=0.02)             #if know where args are, no need to specify name
  ```

```
> y <- seq(-3,25,length=6)          #length option
> x <- c(3,3,9,12,1)
> y <- seq(0,10,along=x)            #will return seq w/ length(x)

> y <- rep(x=3,times=7)             #replicate the value in x, namely 3, seven times
> y <- rep(x=c(3,5),times=7)        # same
> y <- rep(x=c(3,5),times=c(2,6))   #repeat 3 two times, then 5 seven times
```

b. character

```
> y <- "San Diego"
> y <- c("LA","NY","San Fran")
```

c. logical

```
> y <- T
> y <- c(F,F,T)
> y <- (x<5)
```

- 4 ways to access elements: position,exclusion, name, logical.
  Examples: `y<-c(18,32,15,-7,12,19)`

  `> y[3:5] => 15, -7, 12`
  `> y[-c(1,5,6)] => 32, 15, -7`

  āssigning names to elements: `names(y) <- c("A","B","C","D","E","F")`.

  ```
  > y[c("A","C")]
  ```

  ```
  18, 15
  ```

  and typing

  ```
  > y[y<15]
  ```

  returns

  ```
  -7,12
  ```

Exercises

1. Create a sequence of all odd numbers from 1 to 11.
2. Create a sequence of even numbers from 2 to 1000.
3. Declare a vector with the city names Seattle, Amherst, Bozeman, and Gainesville.

- Matrices are like vectors; their elements are all of the same type. A good way to think about matrices is as "vectors with special instructions as to how to lay it out". Matrices: creation, naming, accessing

```
> y<- c(18,32,15,-7,12,19)
> x<- matrix(data=y,nrow=2,ncol=3)
> x
     [,1] [,2] [,3]
[1,]   18   15   12
[2,]   32   -7   19
> z<-matrix(data=y,nrow=2,ncol=3,byrow=T)
> z
     [,1] [,2] [,3]
[1,]   18   32   15
[2,]   -7   12   19

> x<- matrix(data=rep(0,5*8),nrow=5,ncol=8) # a matrix of zeros
> matrix(data=rep(c(10,20,30),times=3,each=3),nrow=3,ncol=3)
     [,1] [,2] [,3]
[1,]   10   20   30
[2,]   10   20   30
[3,]   10   20   30
> matrix(data=rep(c(10,20,30),times=3,each=3),nrow=3,ncol=3,byrow=T)
     [,1] [,2] [,3]
[1,]   10   10   10
[2,]   20   20   20
[3,]   30   30   30
```

Using the command `dim(matrix)` returns the dimension of the matrix. Example:

```
> dim(z)
[1] 2 5
```

Now recall that we did `x<-matrix(data=y,nrow=2,ncol=3)`. Now type:

```
> x<-matrix(data=y,nrow=2,ncol=3)
> x
     [,1] [,2] [,3]
[1,]   18   15   12
[2,]   32   -7   19
> which(x<0,arr.ind=TRUE) #arr.ind shows the array indices in the output
     row col
[1,]   2   2
> a <- which(x<0,arr.ind=TRUE) # which element of the matrix 'x' is negative?
> a # here's the location of the negative element in a
     row col
[1,]   2   2
> a[1] # object 'a' is a 2 elements vector
[1] 2
> a[2]
[1] 2
```

This is useful for replacement, for instance to replace the negative element of x

```
> x[a] <- 8 # replace -7 in 'x' by 8
> x
     [,1] [,2] [,3]
[1,]   18   15   12
[2,]   32    8   19
```

Exercises

1. In the matrix `x` replace all values greater than 20 with -1.
2. Calculate `x*x` (elementwise multiplication).
3. Type in the command window `t(x)%*%x` and compare the results to the results of `x*x`. Why are they different?
4. Type in the command window `x[1,]`. What is the output? How would you extract the first column of x?

- Arrays: creation, naming, accessing

```
> y<-c(1:8,11:18,111:118)
> x<-array(y,dim=c(2,4,3))
```

- Factors: creation, accessing

```
> colors <- c("red","red","green","yellow","yellow","blue")
> colors <- factor(x=colors,levels=c("red","green","yellow","blue"))
> table(colors)
colors
   red  green yellow   blue
     2      1      2      1
```

- vector *attributes*: always either one of 2: mode (numeric,logical,etc...) and length.
- Some special 'values' and coercion
  - Missing values: `NA`
  - infinite values: `Inf, -Inf`
  - "place" holder value: `NULL` for example if y is a vector `dim(y)` returns `NULL`
  - logical check for type: for example

```
> is.numeric(7)
[1] TRUE
> is.na(7)
[1] FALSE
```

  - coercion to type:

```
 > as.factor(7)
[1] 7
Levels: 7
```

- Arithmetic and logical operations and operators
  - Precedence (See Venables and Ripley), from highest to lowest:

```
function(..)
^                   #exponentiation
* /                  #multiplication, division
+ -                 #addition, subtraction
> < <= == != #comparison operators
```

    but can always use parentheses to force a precendence (best)

```
> x <- 36
> y <- 7
> sqrt(x)*7-2 <  x+y^2
[1] TRUE
> (((sqrt(x))*7)-2) < (x+(y^2))  #identical to above but "clear" precedence
```

– element by element operations on two vectors

```
> x <- 1:10
> y <- 11:20
> x+y
 [1] 12 14 16 18 20 22 24 26 28 30
> x-y
 [1] -10 -10 -10 -10 -10 -10 -10 -10 -10 -10
> x*y
 [1]  11  24  39  56  75  96 119 144 171 200
 > log(x) + sqrt(y)
 [1] 3.316625 4.157249 4.704164 5.127952 5.482421 5.791759 6.069016 6.322082
 [9] 6.556124 6.774721
```

– recycling:

```
> z <- 1:3
> y <- 5:6
> x+y  #recycles the 5 in y
[1] 6 8 8
Warning message:
In x + y : longer object length is not a multiple of shorter object length
> x <- 2
> y <- 1:6
> x+y
[1] 3 4 5 6 7 8
> x <- c(2,9)
> x+y
[1]  3 11  5 13  7 15
```

– Integer division and modulo reduction:

```
 > 7 %/% 3       #answer 2, number of times 3 goes into 7
 [1] 2
 > 7 %%  3       #answer 1, 7 - (7 %/% 3)*3
  [1] 1
```

for example, to pick up each 3rd element in x, try:

```
> x<-1:20
> x%%3
 [1] 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2
> x[x%%3==0] <- 6 # replace each third element in x by 6
> x
 [1]  1  2  6  4  5  6  7  8  6 10 11  6 13 14  6 16 17  6 19 20
```

– Logical expressions and functions with logical arguments. Remember that T=1 and F=0.

```
> x <- c(1,3,5)
> y <- c(2,2,4)

> x < y
```

```
[1]  TRUE FALSE FALSE
> x <= y
[1]  TRUE FALSE FALSE
> x != y
[1] TRUE TRUE TRUE
> check <- x != y
> sum(check)
[1] 3
> sum(x != y)
[1] 3

> any(x<y)  #if any term in vector x<y is T, return T
[1] TRUE
> all(x<y)  #if all terms in vector x<y is T, return T
[1] FALSE
> z <- -1
> all(x < y | x > z) #all terms in vector x <y is T OR all terms in vector x > z is T.
[1] TRUE

> x == 3
[1] FALSE  TRUE FALSE
> y == 2
[1]  TRUE  TRUE FALSE

> (x==3 | y==2)  # union of vector x==3 with vector y==2
[1]  TRUE  TRUE FALSE
                # T|T or T|F or F|T returns T
                # only F|F returns F
                #if x ==3 OR y ==2 is T then return T, else return F

> (x==3 & y==2)  # intersection of vector x==3 with vector y==2
[1] FALSE  TRUE FALSE
                # only T&T returns T#
                # if x ==3 AND y ==2 are T then return T, else return F
```

Exercise

1. Interpret the following input and output

```
 > as.numeric(F)
[1] 0
> as.numeric(T)
[1] 1
```

2. Try the following lines and explain the differences in output and why it is important to keep track of parentheses.

```
 > ((1+4)^2) + 2/4
> ((1+4)^2 + 2)/4
```

3. Interpret the following input and output

```
 >!F
[1] TRUE
> T|F
```

```
[1] TRUE
> T&F
[1] FALSE
```

3 Lists

- assignment

```
> x <- list(one=c(18:36),two=c("AK","AL","AZ"),three=c(T,T,F,T),four=matrix(1:12,3,4))
```

- accessing components

```
> x[[1]]          #by order
[1] 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
> x$one           #by name
[1] 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
```

- accessing elements within components

```
> x[[1]][3:6]
[1] 20 21 22 23
> x$one[3:6]
[1] 20 21 22 23
```

- unlisting

```
> unlist(x)
```

- Special case: data.frame list object-creation, access and extension

```
> muscle        <- rep(seq(0,19,by=2),2);
> sex             <- factor(rep(c("M","F"),c(10,10)));
> speed          <- rep(0,20); # a vector of zeros
> speed[1:10]   <-  rnorm(n=10,mean=(30+2*muscle[1:10]),sd=0.1) # filling-in speed
> speed[11:20] <- rnorm(10,(40+2*muscle[11:20]),0.1)
> mydata        <- data.frame(y=speed,x1=muscle,x2=sex)
> print(mydata)
> names(mydata)
[1] "y"  "x1" "x2"
> temp <- lm(y~x1+x2,data=mydata) # a regression with 1 continous and 1 categorical variable.
> summary(temp)
# attaching the dataframe:
> names(mydata)
[1] "y"  "x1" "x2"
> x1 #not yet loaded
Error: object "x1" not found
> y
Error: object "y" not found
> x2
Error: object "x2" not found
> attach(mydata)
> y
 [1] 29.91764 34.00737 38.07257 41.95327 45.91410 50.11541
 [7] 53.98323 57.77561 61.82590 65.98829 39.91799 44.05325
```

```
[13] 47.87027 52.00124 55.91240 59.93873 63.90140 67.96487
[19] 71.90786 75.91131
> x1
 [1]  0  2  4  6  8 10 12 14 16 18  0  2  4  6  8 10 12 14 16 18
> x2 # in the regression this is treated as M=0, F=1
 [1] M M M M M M M M M M F F F F F F F F F F
Levels: F M

> levels(x2) # asking R what are the levels of x2:
[1] "F" "M"
> detach("mydata")
```

4 Functions

- Creation: specified or unspecified arguments

  unspecified `c <- function(...)`

  specified

```
sd <- function(x) {return(sqrt(var(x)))}
dumplot <- function(x,y) {
      plot(x,y)
      abline(0,1)
}
```

- Argument passing for specified case (names,order,default) if don't use names, call function by passing arguments in proper order

```
> dumplot(my.x,my.y)
```

  if use names, then any order ok

```
> dumplot(y=my.y, x = my.x)
```

- Default values for specified case

```
dumplot <- function(x,y,plot.this=T) {
     if(plot.this==T) {
      plot(x,y)
      abline(0,1)
     }
}
```

Exercise

1. Look up the help file for the variance function `var` by typing `?var`. What is the `x` argument? What do the other arguments mean?

5 Simple Functions

- element by element: (same as before, e.g. define `x<-3:12` and add a number to it)
- operate on entire vector and return a scalar:

```
> mean(x)
[1] 7.5
> sum(x)
[1] 75
> prod(x)
[1] 239500800
```

- operate on entire vector and return a vector:

```
> cumsum(x)
 [1]  3  7 12 18 25 33 42 52 63 75
> cumprod(x)
 [1]         3        12        60       360      2520     20160    181440
 [8]   1814400  19958400 239500800
```

- conversion to integers, rounding, ...

```
> x <- 103.652
> round(x)
[1] 104
> round(x,1)
[1] 103.7
> trunc(-0.8) # returns a value formed by truncating the value towards 0.
[1] 0
> floor(-0.8) #returns a value containing the largest integer not greater than the input.
[1] -1
> ceiling(0.8) #returns a value containing the smallest integers not less than the input.
[1] 1
```

- extreme values:

```
> x <- c(3,5,5,9); y <- c(5,3,5,11)
> max(x)
[1] 9
> min(y)
[1] 3
> range(x) #returns smallest and largest values
[1] 3 9
```

- ordering

```
> x <- c(12,5,6,6,-1)
> sort(x)
[1] -1  5  6  6 12
> rev(x)
[1] -1  6  6  5 12
> order(x)
[1] 5 2 3 4 1
> rank(x)
[1] 5.0 2.0 3.5 3.5 1.0
```

- simple statistics

```
> x <- c(8,12,5,4,19,2,1)
> y <- rnorm(length(x), mean=3+2*x,sd=1)
```

```
> mean(x)
[1] 7.285714
> median(x) #note there is no mode function, but max() can work for this purpose
[1] 5
> var(x)
[1] 40.57143
> quantile(x, probs=c(0.025, 0.975))
 2.5% 97.5%
 1.15 17.95
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   3.000   5.000   7.286  10.000  19.000
> cor(x,y)
[1] 0.9996766
> var(cbind(x,y)) #cbind binds vectors by columns while rbind binds by rows.
         x        y
x 40.57143  79.6388
y 79.63880 156.4264
```

Exercise

1. For the the vector `x` defined above calculate the coefficent of variation (the standard deviation over the mean).
2. Modify the `quantile` statement above to return the 25th, 50th and 75th percentiles.
3. Try replacing the `cbind` command int the variance calculation above with `rbind`. Explain the difference in output.

6 Univariate random variable generators, etc

- random number seed; fix for reproducible simulations

```
> set.seed(6)
```

- random sampling:

```
> z <- c(18,22,14,25,36)
> sample(z,size=3,replace=F)
[1] 25 36 18
> sample(z,size=7,replace=T)
[1] 22 36 36 36 25 14 18
> sample(z,size=7,replace=F)
Error in sample(z, size = 7, replace = F) :
  cannot take a sample larger than the population when 'replace = FALSE'
> sample(x=z,size=3,replace=T,prob=c(0.1, 0.3, 0.2, 0.1, 0.3)) #unequal prob sampling
```

- univariate random variable generation: e.g, uniform, binomial, Poisson, Normal, log-normal, Gamma, F, Chi-square

```
rbeta(n=10,shape1=3,shape2=4)    # ten obs'ns from Beta(3,4)
 rbinom(n=10,size=100,prob=0.2)   # ten obs'ns from Binomial(100,0.2)
 rnorm(n=10,mean=20,sd=5.1)       # ten obs'ns from Normal(20,5.1^2)
 rpois(n=30,lambda=3)             # ten obs'ns from Poisson(3)
```

- percentages and percentiles

```
> pnorm(q=1.645,mean=0,sd=1)
[1] 0.950015
> 1-pf(q=5.7,df1=3,df2=15)
[1] 0.008258358
```

- probability density functions or probability mass functions

```
 > dnorm(x=0.5,mean=0,sd=1)
[1] 0.3520653
> dpois(x=1,lambda=2)
[1] 0.2706706
```

Exercises

1. Try the following

   ```
   > set.seed(6)
   > rnorm(5,0,1)
   > set.seed(6)
   > rnorm(5,0,1)
   ```

   now try

   ```
   > set.seed(6)
   > rnorm(5,0,1)
   > rnorm(5,0,1)
   ```

   Explain the different outputs.

2. What happens if you change the seed in the fourth line?

7 Input/Output

1. Output to screen:
   You can either type the object name, print it using `print` or the function `cat`

   ```
   > x <- 3; y <- 10
   > print(x)
   [1] 3
   > cat("x =",x,", y^2=", y^2,"\n") # cat has more control over output format
   x = 3 , y^2= 100
   ```

   You can also paste together, or concatenate, characters:

   ```
    > paste("Zoo","logy",sep="")
   [1] "Zoology"
   ```

2. Output to file:

   - x is an object, ASCII: `write(x,file='name.txt')`
   - x is a matrix or data.frame, ASCII: `write.table(x,file='name',append=T)`

3. Input from file (ASCII files):

   - rectangular data file: `dataset <- read.table('filename.txt', header=T)`
   - irregular shaped files `dataset <- scan`(filename.txt")
   - loading long script or function: `source('myRscript.R')`

Exercise

1. Try to read a file that you create into R. Some hints: If using Excel, save file as "Tab Delimited Text (.csv)". If you include row names in your file you need to include `append=T` as an argument in `read.table`

8 Matrix operations and functions

1. basic operations

```
> A <- matrix(1:3,3,3)
> A
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
[3,]    3    3    3
```

   - dimensions

     ```
     > dim(A)
     [1] 3 3
     ```

   - simple functions, eg. +, -, *, /

     ```
     > A+3
          [,1] [,2] [,3]
     [1,]    4    4    4
     [2,]    5    5    5
     [3,]    6    6    6
     ```

   - matrix arithmetic (be careful with recycling!!)

     ```
     > x <- 1:3
     > x%*%A%*%x
          [,1]
     [1,]   84
     ```

   - transposing

     ```
     > t(A)
          [,1] [,2] [,3]
     [1,]    1    2    3
     [2,]    1    2    3
     [3,]    1    2    3
     ```

   - crossproducts:

     ```
     > t(x)%*%x
          [,1]
     [1,]   14
     ```

     or use

     ```
     > crossprod(x,x)
          [,1]
     [1,]   14
     ```

   - diagonal extraction or assignment

     ```
     > diag(A)
     [1] 1 2 3
     ```

   - sweeping operations:

```
> A <- matrix(1:12,nrow=4,ncol=3)
> colmean <- apply(A,2,FUN=mean) #2 means operate on columns, 1 means operate on rows
> Centered.A <- sweep(A,2, colmean, FUN="-")
> A
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> colmean
[1]  2.5  6.5 10.5
> Centered.A
     [,1] [,2] [,3]
[1,] -1.5 -1.5 -1.5
[2,] -0.5 -0.5 -0.5
[3,]  0.5  0.5  0.5
[4,]  1.5  1.5  1.5
```

- Outer product of verctors `a` and `b`: all possible products of elements of the data vector of `a` with those of `b` :

```
> a<-A[,1]
> b <-A[,2]
> a
[1] 1 2 3 4
> b
[1] 5 6 7 8
> outer(a,b)
     [,1] [,2] [,3] [,4]
[1,]    5    6    7    8
[2,]   10   12   14   16
[3,]   15   18   21   24
[4,]   20   24   28   32
```

Now suppose you want to evaluate the function

$$f(x, y) = \frac{\cos(y)}{1 + x^2}$$

over a regular grid of values with $x-$ and $y-$ coordinates defined by the R vectors x and y respectively. You could use:

```
z <- outer(x,y, function(x,y){cos(y)/(1+x^2)})
```

This can be very useful for surface plots, for example.

2. Linear/matrix algebra

- Inversion: `solve(A)`
- Solving system of equations: if $Ax = y$ and you are given A and y use `backsolve{A,y)` to find x
- Cholesky decomposition: `chol(A)`
- Eigenanalysis: `eigen(A)  # gives eigenvalues and eigenvectors`

9 List and factor functions

1. To produce absolute one-way or higher dimensional frequency tables for a factor vector or vectors:

```
> x<-factor(c("Red","Red","Green","Red"))
> y<-factor(c("Short","Tall","Short","Short"))
> table(x)
x
Green   Red
    1     3
> table(x,y)
       y
x       Short Tall
  Green     1    0
  Red       2    1
```

2. List component operations

```
> x <- list(one=c(18,22), two=c(15,10,12))
> lapply(x,mean)
$one
[1] 20

$two
[1] 12.33333

> lapply(x,var)
$one
[1] 8

$two
[1] 6.333333
```

3. Creating a list from a numeric vector using a factor vector:

```
> y<- c(18,17,19,15,16)
> sex <-factor(c("m","f","m","f","f"))
> split(y,sex)
$f
[1] 17 15 16

$m
[1] 18 19

> boxplot(split(y,sex))
```

10 Character functions

Generally operate on entire character strings, not on single characters, useful for titles and axis labeling.

- Concatenation: `paste('Figure', sep=' ' , 1)`
- Counting characters

```
> nchar('waccasassa')
[1] 10
```

- Extracting part of a character string:

```
> substring("waccasassa", first=6, last=8)
[1] "sas"
```

11 Control structures

1. `if` and "else" blocks

   logical form: `if(logical expression is T) {do this} else {otherwise, do that}`

   ```
   if(x <0) { y <- log(x); print("ok")} else {y <- NA; print("end")}

   # try these with different values for y
   > x = 1;y=10;
   > if(y<5){x=2}
   > x
   ```

2. `ifelse` function: logical form: `ifelse(logical, do this if logical is T, or else do this)`

   ```
   # try these with different values for y
   > x = 1
   > y <- ifelse(x > 0, log(x), NA)
   ```

3. Multiple branches with `switch` function:

   ```
   > switch(cityname, "Pittsburgh" = {y < log(z); z <- sin(y)},
      "St. Louis" = {y < exp(3)}, "Boise" = {y < sqrt(3)}
   ```

12 Looping

1. `for` loops

   syntax: `for(index in set) {do this}`

   ```
         #For loop syntax: for (var in seq) {expression}

          for (i in 1:3) {print(i)} #basic for loop

          for (i in c(1,3,17)) {print(i)} #sequence need not be in order


          for (char in c("A", "Z", "2")) {print char} #sequence need not
          be numbers

          forseq <- 1:4
          for (i in forseq) {print i} #sequence can be constructed
          elsewhere
   ```

2. `while` loops syntax: `x(this condition is T) {do this}`

   ```
   > y <- 10
   > out <- 1
   > x <- y
   > while(x > 0) {out <- out*x; x <- x-1}

          i<-0
          while (i<4) # basic while loop
                      # logical check done before any operations
          {
             i<- i+1
             print(i)
          }
   ```

3. `repeat` loops

   syntax: `repeat{ do this over and over if condition is T) break() }`

```
i<-0
repeat      #basic repeat loop
            #logical check after operations
{
   i=i+1
   print(i)
   if (i>3) {break}
}
```

4. Think twice before looping, try to vectorize.

```
# Ex. 1: row means:
> x <- matrix(rbeta(10000,3,2),2000,5)
> mean.vec <- rep(0,2000) # create space in advance

#--looping
> now <- proc.time()[1:2];
> for(i in 1:2000) mean.vec[i] <- mean(x[i,])
> speed<- proc.time()[1:2] - now;
> speed
user.self  sys.self
    0.047     0.001

# -- vectorizing:
> now <- proc.time()[1:2]
> mean.vec <- apply(x,1,mean)
> speed <- proc.time()[1:2]-now
> speed
user.self  sys.self
    0.045     0.001

> rm(mean.vec,x,speed)
```

Exercises

1. Write a `while` loop that finds the numerical limit of R. This means you need to find the power x of 10^x that R treats as infinite. You can use the function `is.finite()`.
2. Write a `for` loop that will print out `ith iteration` where i goes from 1 to 100.
3. Use a `for` loop to write a simple yearly population growth model. Let the initial population size be 10 and assume that each individual has 2 offspring, then dies.

13 Function writing basics

1. argument assignment
2. explicit vs. implicit returns from function

   implicit returns the last result in a function, e.g.: `function() { x <- 3 + y }`

   explicit: `function() { x <- 3 + y; return(x) }`
3. debugging/file editing

- `stop('message') # ends execution and prints message`
- `warning('message') # prints message but does not end execution`
- `on.exit(argument) # when execution ends runs code in argument`
- `missing(argument) # can be used to test whether value was passed into function`

4. communicating/ error checking with the function user
   - Edit in a separate file and use `source()`
   - cut & paste code
   - use code editor (such as Tinn-R)
   - insert `print()` or `cat()` statements in your code. This is very useful.
5. More sophisticaded debugging
   - `debugger(function())`
   - `browser(function())`
6. Calling C and Fortran routines Functions written in C or Fortran can be called using the `.c` function in R. Although straight forward, special care must be taken about the types of your variables you pass into your C function. For more a short tutorial see here: http://www.ats.ucla.edu/stat/r/library/interface.pdf

Exercise

1. Construct a function that calculates a one or two sided t-test, for an arbitrary t-statistic and sample size. You can use the `pt` function.

14 Examples

```
# Random numbers generators examples:

# Uniform:
a<- runif(n=10000,min=0,max=1);
hist(a,xlab="Histogram of 10000 pseudo-random numbers from a U(0,1) distribution")

# Binomial

b <- rbinom(n=10000, size=10,prob=0.75);
hist(b,xlab="Histogram of 10000 draws from X~Binom(n=10,p=0.75)",col=3)

# Bernoulli

bernoulli.draw1 <- rbinom(n=1000, size=1,prob=0.75);
bernoulli.draw2 <- sample(c(0,1), size=1000, replace=T, prob = c(0.25,0.75));

par(mfrow=c(2,1));
hist(bernoulli.draw1,  xlab="");
hist(bernoulli.draw2,  xlab="");

# A very important example:

# The long tedious way:
par(mfrow=c(2,4));
hist(rbinom(n=10000,size=5,prob=0.45),main="n=5, p=0.45",xlab="",col=3)
hist(rbinom(n=10000,size=7,prob=0.45),main="n=7, p=0.45",xlab="",col=3)
hist(rbinom(n=10000,size=9,prob=0.45),main="n=9, p=0.45",xlab="",col=3)
hist(rbinom(n=10000,size=11,prob=0.45),main="n=11, p=0.45",xlab="",col=3)
hist(rbinom(n=10000,size=20,prob=0.45),main="n=20, p=0.45",xlab="",col=3)
hist(rbinom(n=10000,size=50,prob=0.45),main="n=50, p=0.45",xlab="",col=3)
hist(rbinom(n=10000,size=100,prob=0.45),main="n=100, p=0.45",xlab="",col=3)
hist(rbinom(n=10000,size=10000,prob=0.45),main="n=10000, p=0.45",xlab="",col=3)
```

```
# The short version:
ntrials <-c(5,7,9,11,20,50,100,10000);
len     <- length(ntrials);
par(mfrow=c(2,4));
for(i in 1:len){

hist(rbinom(n=10000,size=ntrials[i],prob=0.45),xlab="", main=paste("p=0.45, n=", sep="",toString(ntrials[i])),col=4);
}

# Compare to (CLT!):

binom.hist <- rbinom(n=10000,size=10000,prob=0.45);
normal.hist<- rnorm(n=10000,mean=10000*0.45, sd= sqrt(10000*0.45*0.55));

par(mfrow=c(2,1));
hist(binom.hist, xlab="", xlim=c(4300,4700),col=2, main="Binomial sample, n=10000, p =0.45");
hist(normal.hist,xlab="",xlim=c(4300,4700),col=2, main="Normal sample, mean = 10000*0.45");
```

A little function example: For many organisms, births occur in regular, well-defined breeding seasons. Discrete time population growth models are adequate for these organisms. Let N_t denote population size of one of these organisms at time t. Here's a little function that outputs a time series of exponential population growth according to the model:

$$N_{t+1} = N_t \mathrm{e}^a.$$

```
exp.growth <- function(a,len,no){
# This function returns a time series of length 'len' of population growth
# according to the model N(t+1) = N(t) * exp(a);
# the argument 'a' is the growth rate and 'no' is the initial population size
# The specified length has to be >= 2

pop.vec     <- rep(0,len);
pop.vec[1] <- no;
for(i in 1:(len-1)){
pop.vec[(i+1)] <- pop.vec[i]*exp(a);
}
return(pop.vec)
}

# trying the function:
popsim <-exp.growth(a=0.05,len=20,no=2)
popsim

> popsim
 [1] 2.000000 2.102542 2.210342 2.323668 2.442806 2.568051 2.699718 2.838135
 [9] 2.983649 3.136624 3.297443 3.466506 3.644238 3.831082 4.027505 4.234000
[17] 4.451082 4.679294 4.919206 5.171419
```

Exercise

1. Modify the `exp.growth` function to make the variable a a random variable. Use the normal distribution random number generator `rnorm`.

15 Documentation

When writing functions, documenting them so that anyone can use them is very important. In fact, repeatability or how easy it is to reproduce one's work is a key ingredient of doing good science. Here are some advices from Mark on how to document a function.

```
#DOCUMENTATION:
#     Documentation is the most important feature of a function!
#     Any text in a line to the right of a # sign is a comment and will not
#     influence function execution.
#     The MINIMUM DOCUMENTATION
#     THAT IS SOCIALLY RESPOSIBLE
#      5 key elements 1) Description of arguments
#     pased into the function. 2) Description of values or ojects returned
#     by the function.  3) Coders name.  4) Date last modified.
#     And, 5) explicit description of side effects (if any) the function has.
#     Side effects are any changes to the environment outside of the function
#     other than through the function's return value.  Documentation complexity
#     should increase with function complexity and the likelihood that others
#     will need to understand the function.
#     Other useful documentation elements include: a statement of function
#     purpose, a variable dictionary, a modification history, a list of
#     dependencies, an algorithm description, and an algorithm source.
#     Comments can be inserted in the body of the function to clarify
#     tricky or critical steps.
```