UCLA Department of Statistics
Statistical Consulting Center

Intermediate R

Denise Ferrari
denise@stat.ucla.edu

October 28, 2009

Part I

The Workspace

the R workspace

working directory

The R workspace consists of any user defined object (vectors, matrices, data frames, etc).
The following are some standard commands useful for managing the workspace:

```
# print the current working directory
> getwd()

# list the objects in the workspace
> ls()

# change the working directory
> setwd(new.dir)
```

the R workspace

options

We can use the command `options` to view and set options for the R session.

```
# list of available options
> help(options)

# view current options settings
> options()

# set number of digits to print on output
> options(digits=3)
```

the R workspace

session history

```
# display last 25 commands
> history()

# display all previous commands
> history(max.show=Inf)

# save your command history to a file
> savehistory(file="myfile") # default is ".Rhistory"

# recall your command history
> loadhistory(file="myfile") # default is ".Rhistory"
```

the R workspace

saving and loading workspaces

```
# save the workspace to the file ".RData" (default)
> save.image()
# which is the same as
> save(list=ls(all=TRUE), file=".RData")

# save a specific object to a file
> save(object, file="myfile.RData")

# load a saved workspace into the current session
> load("myfile.RData")

# quit R
> q()

# to avoid having R asking you whether it should save your data
> q(save="no")
```

Part II

Packages

packages

Packages consist in a set of pre-programmed functions, sometimes developed for performing specific tasks.

There are two types of packages:

- Those that come with the base installation of R.
- Those that are available for download and need to be installed manually.

packages included in the base installation

To check which packages are available in your R installation:

- choose Packages & Data > Package Manager

Even if a package came with the base installation, you may need to load it in order to be able to use its functions.

```
# load the package "splines"
> library(splines)
# or
> require(splines)

# to get help with the package
> help(splines)
```

Now, you can start using the functions included in that package.

extra packages

If a package is not included in the base installation, you need to install it. You can either download it from the R website, or you can do it within the R environment:

To obtain and install the package `gstat` (or any other available package) from within R:

- choose Packages & Data > Package Installer
- type `gstat` in the package search field and hit 'Get List'
- choose the desired package and click 'Install Selected'.

Note: The process of installing a package needs to be done only once.

Part III

Data in R

mode and class

Every R object has a number of attributes.
Some of the most important are:

`mode`: Mutually exclusive classification of objects according to their basic structure.
- numeric, complex, character, logical

`class`: Property assigned to an object that determines how generic functions operate with it. If no specific class is assigned to object, by default it is the same as the mode.
- vector, matrix, data.frame, array, list, etc.

mode and class

Example

Let's check the mode and the class of simple R objects:

First, create a simple vector in R and check its mode and class:

```
> x <- 1:4
> mode (x)
[1] "numeric"
> class(x)
[1] "integer"
```

Now, let's associate the attribute "dimension" to the vector:

```
> dim(x) <- c(2,2)
> mode (x)
[1] "numeric"
> class(x)
[1] "matrix"
```

vectors

- The vector is the basic data object in R.
- Even scalars are defined as vectors of length 1.
- All elements in a vector must be of the same mode.

creating vectors

elements in sequence

```
# using the operator ":"
> x <- 1:4
> x
[1] 1 2 3 4
> mode(x)
[1] "numeric"

# using the function seq()
> y <- seq(from=6, to=9, by=0.5)
> y
[1] 6.0 6.5 7.0 7.5 8.0 8.5 9.0
> mode (x)
[1] "numeric"
```

creating vectors

other vectors

```
# Using the function repeat "rep()"
> z <- rep(TRUE, 2)
> z
[1] TRUE TRUE
> mode (z)
[1] "logical"

# Using the function concatenate "c()"
> w <- c(5, 7, "dog", 9)
> w
[1] "5"   "7"   "dog" "9"
> mode(w)
[1] "character"
```

creating vectors

other vectors

Now, try concatenating the vectors `x, y, z`.
What is the mode of this new vector?

What if you concatenate the vectors `x, y, z, w`?

name assignment

We can assign names to the elements of vectors.
These names are useful to identify elements when the object is displayed and also to access elements of the vector through subscripts.

name assignment

Names can be assigned when the vector is created:

```
> x <- c(one=1, two=2, three=3)
> x
  one   two three
  1     2     3
```

name assignment

Or can be added later:

```
> x <- 1:3
> x
[1] 1 2 3
> names(x) <- c("one", "two", "three")
> x
  one   two three
  1     2     3
```

name assignment

We can also use names as indexes to modify certain elements of the names:

```
> names(x)[c(1,3)] <- c("a", "c")
> x
  a two   c
  1   2   3
```

recycling

If two vectors in a certain operation are not of the same length, R recycles the values of the shorter vector to make the lengths compatible.

recycling

Recycling a scalar:

```
> a <- 1:10
> a + 1
[1]  2  3  4  5  6  7  8  9 10 11
```

recycling

Recycling a shorter vector:

```
> a <- 1:10
> a + c(1,2,3)

[1]  2  4  6  5  7  9  8 10 12 11

Warning message:
In a + c(1, 2, 3) :
longer object length is not a multiple of shorter object length
```

recycling

Recycling a shorter vector:

```
> a <- 1:10
> a + c(1,2)

[1]  2  4  4  6  6  8  8 10 10 12
```

There is no warning when the length of longer vector is a multiple of the length of the shorter vector!

arrays and matrices

- An array is a multidimensional extension of a vector.
- Like vectors, all elements of an array must be of the same mode.
- Arrays have an attribute called `dim` (dimension).
- The most common array used in R is the matrix, which corresponds to a 2-dimensional array.

creating matrices

Matrices are stored internally as vectors, with each column "stacked" on top of each other.

```
> mat <- matrix(1:6, nrow=2, ncol=3
                dimnames=list(NULL, c("A","B","C")))
> mat

     A B C
[1,] 1 3 5
[2,] 2 4 6

> dim (mat)
[1] 2 3
```

Names can always be assigned later to the columns and rows of the matrix using the commands `row.names()` and `col.names()`

If we want the matrix to be filled in by rows, we need to use the option `byrow = TRUE`

lists

Lists are generic vectors that allow elements of different modes.

```
> mylist <- list(1:3, "dog", TRUE)
> mylist
[[1]]
[1] 1 2 3

[[2]]
[1] "dog"

[[3]]
[1] TRUE

> sapply(mylist, mode)
[1] "numeric"   "character" "logical"
```

Now, try to give names to the elements of a list.

data frames

Data frames are useful to store data in matrix-like form, while allowing for different modes.

Note: A data frame is a special list whose elements have equal lengths:

mode: `list`

class: `data.frame`

creating data frames

```
> dat <- data.frame( book = c("B1","B2","B3"),
                     author = c("A1", "A2","A3"),
                     year = c(1950, 2005, 1999),
                     used = c(TRUE, FALSE, TRUE) )
> dat
  book author year  used
1   B1     A1 1950  TRUE
2   B2     A2 2005 FALSE
3   B3     A3 1999  TRUE
```

```
> sapply(dat, mode)
     book    author      year      used
"numeric" "numeric" "numeric" "logical"

> sapply(dat, class)
     book    author      year      used
 "factor"  "factor" "numeric" "logical"
```

Now, try to check the `mode` and the `class` of the data frame.

converting objects

We can convert R objects to another mode or class in order to change the way the object in R behaves.
The functions to perform conversion begin with the prefix "`as.`"

Examples:

- `as.numeric()`
- `as.character()`
- `as.matrix()`
- `as.data.frame()`

converting characters to numbers

```
> x <- sample(rep(c(0,1), c(3,5)))
> x
[1] 1 0 1 0 1 0 1 1

# The function table returns the counts
# of 0's and 1's in x

> tt <- table(x)
> names(tt)
[1] "0" "1"

# what happens if we try to do this?
> sum(names(tt) * tt)

# now, try the following:
> sum(as.numeric(names(tt)) * tt)
```

converting to list

list versus as.list

```
> x <- 1:3

> list(x)
[[1]]
[1] 1 2 3

> as.list(x)
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3
```

list creates a list of length one containing the elements of the vector x.

as.list converts the vector x into a list of the same length as the vector.

conversion of logical variables

The conversion of logical variables in a numeric context is performed automatically:

```
 > x <- -2:5
 > x
[1] -2 -1  0  1  2  3  4  5

 > x > 0
  [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE TRUE  TRUE

> sum(x > 0)
[1] 5
```

Now, use the conversion of logical values to count the number of elements that are not equal in the matrices:

```
a <- matrix(1:9, ncol=3)
b <- matrix(1:9, ncol=3, byrow=TRUE)
```

Part IV

Reading Data

reading vectors and matrices

data of the same mode

When all the data is of the same mode, the `scan()` function can be used to store them in a vector or matrix.

By default, `scan()` interprets the data as numeric.
This can be changed by the argument '`what=`'.

the scan() function

reading vectors from the console

When reading vectors from the console, R will show the index of the next item to be entered. Once it is done, R shows the number of elements read.

```
# To read numeric elements
> numbers <- scan()
1: 1 2 3 4
5: 5 4
7:
Read 6 items
> numbers
[1] 1 2 3 4 5 4
```

```
# To read characters, set what=""
> pets <- scan(what="")
1: cat dog bird
4: turtle fish
6:
Read 5 items
> pets
[1] "cat"    "dog"    "bird"   "turtle" "fish"
```

the scan() function

reading lists from the console

The 'what=' argument is a list containing examples of the expected data types. Numeric values are specified by 0.

```
> pets <- scan(what=list(pet="", number=0, male=0))
1: dog 1 1
2: cat 3 1
3: fish 10 5
4:
Read 3 records
> pets
$pet
[1] "dog"  "cat"  "fish"
$number
[1]  1  3 10
$male
[1] 1 1 5
```

the scan() function

reading matrices from the console

```
> mat <- matrix(scan(), ncol = 3, byrow = TRUE)
1: 1 2 3
4: 2 4 6
7: 3 6 9
10:
Read 9 items
> mat
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    4    6
[3,]    3    6    9
```

Note: We use `byrow = TRUE` so that the matrix is filled as we type it.

the scan() function

reading from a file

```
> filename <- "http://www.stat.ucla.edu/~denise/SCC/housing.data"
> raw <- scan (file=filename,
         what=c(f1=0, NULL, f3=0, f4=0, rep(list(NULL),10)))
Read 545 records

#  store the data in a matrix
> housing <- cbind(raw$f1, raw$f3, raw$f4)
> head(housing)
          [,1]      [,2]     [,3]
[1,]   0.00632  18.00000  0.53800
[2,]  24.00000   0.02731  0.00000
[3,]   9.14000  21.60000  7.07000
[4,] 392.83000   4.03000  0.00000
[5,]  18.70000 394.63000  0.06905
[6,] 222.00000  18.70000 36.20000
```

reading data frames

data with different modes

To read data into R in the form of a data frame,
we use the function `read.table()`.

Note: For data of a single mode, it is more efficient to use `scan()`.

the `read.table()` function

```
> filename <- "http://www.stat.ucla.edu/data/pnc/anneal.dat"
> anneal <- read.table (file=filename, header=TRUE, sep="")
> head(anneal)
  TRR LAMBDA PIXELS
1 0.7      5    660
2 0.7      5    489
3 0.7     10    446
4 0.7     10    308
5 0.9      5   1013
6 0.9      5   1153
```

Part V

Generating Data

generating data

Sometimes it is necessary to generate data to carry on simulations or to test programs when real data is not available.

sequences

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

> seq(10,50, 5)
[1] 10 15 20 25 30 35 40 45 50

> seq(10, by=5, length=10)
[1] 10 15 20 25 30 35 40 45 50 55
```

the function gl()

generating levels

One common use of sequences is to generate factors for designed experiments. Let's create the levels for an experiment with two groups, three subgroups and two observations per subgroup:

```
> lvl <- data.frame(group = gl(2, 6, length=12),
                    subgroup = gl(2, 3, length=12),
                    obs = gl(2, 1, length=12))
> lvl
   group subgroup obs
1      1        1   1
2      1        1   2
...
11     2        2   1
12     2        2   2
```

the function expand.grid()

This function can be used to create a data frame with the unique combinations of the elements of sequences.

```
> oe.grid <- expand.grid(odd = seq(1, 5, by=2),
                         even = seq(2, 5, by=2))
> oe.grid
  odd even
1   1    2
2   3    2
3   5    2
4   1    4
5   3    4
6   5    4
```

the function expand.grid()

evaluating functions over a range of inputs

```
> xy <- expand.grid(x = 0:10, y = 0:10)
> fcn <- function(row){row[1]^2+row[2]^2}
> z <- apply(xy, 1, fcn)

> head(cbind(xy,z))
  x y  z
1 0 0  0
2 1 0  1
3 2 0  4
4 3 0  9
5 4 0 16
6 5 0 25
```

random numbers

Function	Distribution	Function	Distribution
rbeta	Beta	rlogis	Logistic
rbimon	Binomial	rmultinom	Multinomial
rcauchy	Cauchy	rnbinom	Neg. Binomial
rchisq	Chi-square	rnorm	Normal
rexp	Exponential	rpois	Poisson
rf	F	rsignrank	Signed Rank
rgamma	Gamma	rt	Student's t
rgeom	Geometric	runif	Uniform
rhyper	Hypergeometric	rweibull	Weibull
rlnorm	Log Normal	rwilcox	Wilcoxon Rank Sum

Note: To be able to reproduce always the same results, use the function `set.seed()`

random numbers

```
# Uniform random numbers
> set.seed(123)
> rn.1 <- runif(10, min=0, max=1)
> rn.1
 [1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673
[6] 0.0455565 0.5281055 0.8924190 0.5514350 0.4566147

# Normal random numbers
> set.seed(123)
> rn.2 <- rnorm(10, mean=0, sd=1)
> rn.2
[1] -0.56047565 -0.23017749  1.55870831  0.07050839
[5]  0.12928774  1.71506499  0.46091621 -1.26506123
[9] -0.68685285 -0.44566197
```

the `sample()` function

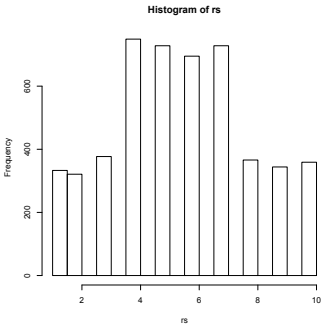
generating random permutations

```
# Random permutation from a vector
> x <- 1:10
> sample(x)
[1]  9  7  6 10  4  8  3  2  1  5
> sample(10)
[1]  4  3  2  9  7  8  1 10  6  5

# Sampling with replacement
> sample(10, replace=TRUE)
[1] 3 9 1 5 8 2 6 3 2 8

# Sample size > input vector
> sample(5, size=10, replace=TRUE)
[1] 5 2 4 1 2 2 5 3 5 5
```

the `sample()` function

generating random permutations

```
# Unequal probabilities
> set.seed(111)
> pvec <- c(rep(1,3),rep(2,4), rep(1,3))/10
> rs <- sample(1:10, size=5000,
                  replace=TRUE, prob=pvec)
> hist(rs)
```



Part VI

Writing Data

writing R objects in ASCII format

The functions `save()` and `save.image()` usually save R objects into binary files.

The ASCII format is convenient, since it produces human-readable files that can be:

- visually inspected using a simple text editor;
- imported from other programs.

Two functions can be used to save objects in ASCII format files: `write()` and `write.table()`.

the `write()` function

writing single-mode data

The `write()` function is useful to save data of the same mode.
It is the output equivalent of the `scan()` function.

```
> write(t(housing), file="housing.txt", ncolumns = ncol(housing))
```

Note: Matrices are stored internally in a columns-wise fashion. To write a matrix in row-wise order, we need to use the transpose function `t()` and specify the number of desired columns.

the `write.table()` function

writing mixed-mode data

The `write.table()` function is useful to save data of different modes, such as data frames.

```
# Writing on the console
> write.table(anneal,
              quote=FALSE, row.names=FALSE, sep=",")

# Writing to a file
> write.table(anneal, file = "anneal.txt"
              quote=FALSE, row.names=FALSE, sep=",")
```

Part VII

Subscripting

subscripting

Subscripting can be used to access and manipulate the elements of objects like vectors, matrices, arrays, data frames and lists.

Subscripting operations are fast and efficient, and should be the preferred method when dealing with data in R.

numeric subscripts

In R, the first element of an object has subscript 1.

```
> x <- 1:10
> x
[1]  1  2  3  4  5  6  7  8  9 10

> x[0]
integer(0)

> x[1]
[1] 1
```

numeric subscripts

A vector of subscripts can be used to access multiple elements of an object.

```
> x <- 1:10
> x
[1]  1  2  3  4  5  6  7  8  9 10

> x[c(1,3,5)]
[1] 1 3 5
```

numeric subscripts

Negative subscripts extract all elements of an object except the one specified.

```
> x <- 1:10
> x
[1]  1  2  3  4  5  6  7  8  9 10

> x[-c(1,3,5)]
[1]  2  4  6  7  8  9 10
```

How would you do to subscript some elements of a list?
Use the `pets` list created previously.

character subscripts

If a subscriptable object has names associated to it, a character string or vector of character strings can be used as subscripts.

```
> x <- 1:10
> names(x) <- letters[1:10]

> x[c("a", "b", "c")]
a b c
1 2 3
```

```
> pets
$pet
[1] "dog" "cat"
$number
[1]  1 11
$male
[1] 1 1

> pets[["pet"]]
[1] "dog" "cat"

> pets$pet
[1] "dog" "cat"
```

Note: Negative character subscripts are not allowed.

logical subscripts

We can use logical values to choose which elements of the object to access. Elements corresponding to `TRUE` in the logical vector are included, and elements corresponding to `FALSE` are ignored.

```
> x <- 1:10; names(x) <- letters[1:10]
> x > 5
    a     b     c     d     e     f     g     h     i     j
FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE

> x[x > 5]
 f  g  h  i  j
 6  7  8  9 10

# using logical subscript to modify the object
> x[x > 5] <- 0
> x
a b c d e f g h i j
1 2 3 4 5 0 0 0 0 0
```

the `which()` function

This function can be useful when one wishes to find the indices of elements that have a certain characteristic.

```
> y <- seq(1, 10, by=3)
> y
[1]  1  4  7 10

# even numbers
> which(y %% 2 == 0)
[1] 2 4

# an equivalent expression
> seq(along=y)[y %% 2 == 0]
[1] 2 4
```

subscripting multidimensional objects

matrices and arrays

For multidimensional objects, subscripts can be provided for each dimension.
To select all elements of a given dimension, use the "empty" subscript.

```
> mat <- matrix(1:12, 3, 4,
                byrow=TRUE)
> mat
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12

> mat[5]
[1] 6
> mat[2,2]
[1] 6
```

```
> mat[1, ]
[1] 1 2 3 4

> mat[c(1,3),]
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    3    6    9   12
```

How would you do to extract the even columns of the matrix?
Hint: use the `which()` function.

the `rev()` function

reversing the order of rows and columns

```
> iris.rev <- iris[rev(1:nrow(iris)),]
> head(iris.rev)
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
150          5.9         3.0          5.1         1.8 virginica
149          6.2         3.4          5.4         2.3 virginica
148          6.5         3.0          5.2         2.0 virginica
147          6.3         2.5          5.0         1.9 virginica
146          6.7         3.0          5.2         2.3 virginica
145          6.7         3.3          5.7         2.5 virginica
```

Now, try to reverse the columns of the iris data frame.

the `order()` function

sorting rows of a matrix arbitrarily

```
# sort the iris data frame by
# Sepal.Length
> iris.sort <- iris[order(iris[,"Sepal.Length"]),]
> head(iris.sort)
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
14          4.3         3.0          1.1         0.1  setosa
9           4.4         2.9          1.4         0.2  setosa
39          4.4         3.0          1.3         0.2  setosa
43          4.4         3.2          1.3         0.2  setosa
42          4.5         2.3          1.3         0.3  setosa
4           4.6         3.1          1.5         0.2  setosa
```

Try to sort the iris data frame in decreasing order with respect to `Sepal.Width`.
Hint: use the `decreasing=TRUE` argument.

the `drop=` argument

avoiding dimension reduction

By default, subscripting operations reduce the dimensions of an array whenever possible. To avoid that, we can use the `drop=FALSE` argument.

```
> s1 <- mat[1,]; s1
[1] 1 2 3 4
> dim(s1)
NULL

> s2 <- mat[1,,drop=FALSE]; s2
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
> dim(s2)
[1] 1 4
```

combined selections for matrices

Suppose we want to get all the columns for which the element at the first row is less than 3:

```
> mycols <- mat[1,] < 3; mycols
[1]  TRUE  TRUE FALSE FALSE

> mat[ , mycols, drop=FALSE]
     [,1] [,2]
[1,]    1    2
[2,]    5    6
[3,]    9   10
```

complex logical expressions

subscripting data frames

```
> dat <- data.frame(a = seq(5, 20, by=3),
         b = c(8, NA, 12, 15, NA, 21))
> dat[dat$b < 10, ]
      a  b
1     5  8
NA   NA NA
NA.1 NA NA

# removing the missing values
> dat[!is.na(dat$b) & dat$b < 10, ]
      a  b
1     5  8
```

the function `subset()`

subscripting data frames

The function `subset()` allows one to perform selections of the elements in a data frame in very simple way.

```
> dat <- data.frame(a = seq(5, 20, by=3),
         b = c(8, NA, 12, 15, NA, 21))

> subset(dat, b < 10)
  a b
1 5 8
```

Note: The `subset()` function always returns a new data frame, matrix of vector, and is not adequate for modifying elements of a data frame.

Part VIII

Loops and Functions

loops

A loop allows the program to repeatedly execute commands. Loops are common to many programming languages and their use may facilitate the implementation of many operations.

There are three kinds of loops in R:

- 'for' loops
- 'while' loops
- 'repeat' loops

Note: Loops can be very inefficient in R.
For that reason, their use is not advised, unless necessary.

'for' loops

General form:

```
for (variable in sequence) {
    set_of_expressions
    }
```

```
> for(i in 1:10) {
   print(sqrt(i))
  }

[1] 1
[1] 1.414214
[1] 1.732051
...
[1] 3.162278
```

'while' loops

General form:

```
while (condition) {
    set_of_expressions
    }
```

```
> a <- 0; b <- 1
> while(b < 10) {
   print(b)
   temp <- a+b
   a <- b
   b <- temp
  }
```

```
[1] 1
[1] 1
[1] 2
[1] 3
[1] 5
[1] 8
```

'repeat' loops

General form:

```
repeat (condition) {
    set_of_expressions
    if (condition) { break }
    }
```

```
> a <- 0; b <- 1
> repeat {
   print(b)
   temp <- a+b
   a <- b
   b <- temp
   if(b>=10){break}
  }
```

```
[1] 1
[1] 1
[1] 2
[1] 3
[1] 5
[1] 8
```

Note: The loop is terminated by the `break` command.

cleaning the mess

In the previous cases, the output and the R commands are both shown in the console.

To have a cleaner version when working with loops, we can do:

```
> x <- 1; d <- 2
> while (length(x) < 10) {
    position <- length(x)
    new <- x[position]+d
    x <- c(x,new)
  }

> print(x)
[1]  1  3  5  7  9 11 13 15 17 19
```

writing functions

A function is a collection of commands that perform a specific task.
General form:

```
function.name <- function (arguments){
    set_of_expressions
    return (answer)
    }
```

```
> AP <- function(a, d, n){
    x <- a
    while (length(x) < n){
      position <- length(x)
      new <- x[position]+d
      x <- c(x, new)
    }
    return(x)
  }
```

writing functions

Once you run this code, you will have available a new function called `GP`.
To run the function, type on the console:

```
> AP(1,2,10)
[1]  1  3  5  7  9 11 13 15 17 19

> AP(1,0,10)
[1] 1 1 1 1 1 1 1 1 1 1
```

Note that for `d==0` the function is returning a sequence of ones.
We can easily fix this with an `if` statement.

the 'if' statement

General form:

```
if (condition) {
    set_of_expressions
}
```

We can also combine the 'if' with the 'else' statement:

```
if (condition) {
    set_of_expressions
} else {
    set_of_expressions
}
```

the 'if' statement

```
> AP <- function(a, d, n){
    if(d ==0) {
      return("Error: argument 'd' should not be 0")
      break
    } else {
      x <- a
      while (length(x) < n){
        position <- length(x)
        new <- x[position]+d
        x <- c(x, new)
        }
    return(x)
    }
  }

> AP(1, 0, 3)
[1] "Error: argument 'd' should not be 0"
```

the 'ifelse' statement

The 'if' statement works fine if you are conditioning on a single value.

If you are working with a vector, the 'ifelse' statement can come in handy.

General form:

```
ifelse (test, yes, no)
```

```
> x <- c(6:-4)
> sqrt(x) # gives a warning

# fix that with an ifelse statement
> sqrt(ifelse(x >= 0, x, NA))
```

Part IX

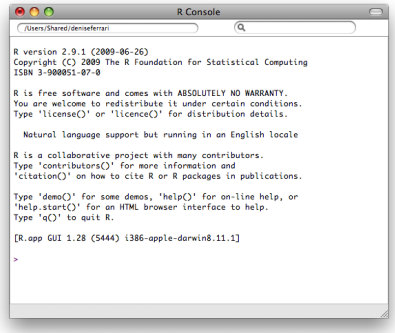
Beyond the Command Line

beyond the command line

As mentioned earlier, the process of using R requires the user to type commands on the console window and click return.

This can be burdensome, specially if you need to repeat a certain collection of commands from time to time, or if you wish to fix mistakes.

So, instead of typing your commands on the console window, you can use an R script. An R script is just a text file where you can write and save R commands for later use.



R startup window:
aka console or command window.

the text editor

Using a special text editor to type, document and save code for later use is highly recommended.

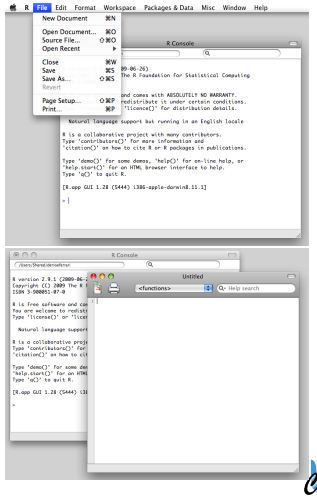
There is a great variety of external text editors available.
We are going to use the R built-in editor.

script code

To **create** a new R script , you may:

- choose File > New Document, or
- hit Command + N

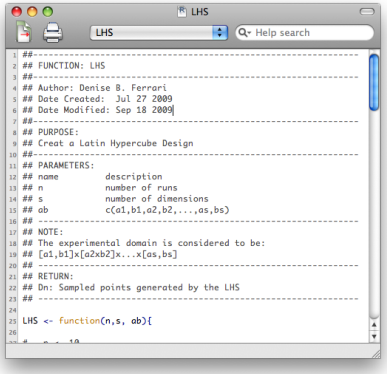
So, now, you're ready to start creating R code.



script code

To **open** an existing R script , you may:

- choose File > Open Document, or
- from Finder, double-click the desired R script file.



script code

If you're not interested in editing the script, you can just run it.

To **run** a script, let's say one with the name:

`/Users/deniseferrari/Documents/Rcode/LHS.R`

you may use:

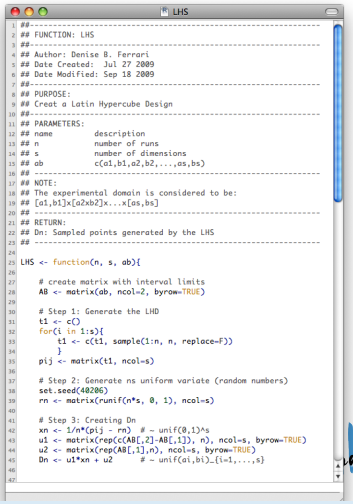
- The R command line:
  `source("/Users/deniseferrari/Documents/Rcode/LHS.R")`
- Terminal:
  `R CMD BATCH /Users/deniseferrari/Documents/Rcode/LHS.R`

documenting script code

general guidelines

It is important that you write code in a clear, simple and well documented way. The following is a list with some general guidelines:

- Add comments: start the line with the symbol "#".
- Use indentations to indicate which pieces of code belong together.
- Add spaces around commands, variables, commas, etc.



using scripts to change directories

```
# store the current directory
> old.dir <- getwd()

# move to the new directory
> setwd("/Users/deniseferrari/Documents/Rcode")

# set the output file
> sink("2009-09-20.out")

# perform all desired tasks
...

# close the output file
> sink()

# go back to the original directory
> setwd(old.dir)
```


more general advice

Common errors can be avoided if extra care is taken regarding the following aspects:

- R is case sensitive
  So, a variable called `my.var` is NOT the same as a variable called `My.var` or `my.Var`, etc.

- Brackets: { }, [ ], ( )
  Programming makes intense use of brackets. Make sure that an opening bracket { is matched with a closing bracket } and that it is used in the correct position for the task.

Part X

Additional Resources

references

- Alain F. Zuur et. al. (2009).
  A Beginner's Guide to R. Use R! Series. Springer.
- Phil Spector (2008).
  Data Manipulation with R. Use R! Series. Springer.
- Owen Jones et. al. (2009).
  Introduction to Scientific Programming and Simulation Using R.
  Chapman & Hall.

online resources

- CRAN: `http://cran.stat.ucla.edu/`
- R search engine: `http://www.rseek.org`
- Quick R: `http://www.statmethods.net/index.html`
- UCLA Statistics Information Portal:
  `http://info.stat.ucla.edu/grad`
- UCLA Statistical Consulting Center
  E-consulting and Walk-in Consulting `http://scc.stat.ucla.edu`

upcoming mini-courses

Monday	November 2	Introduction to Regression with R
Wednesday	November 4	Advanced Regression with R

For a schedule of all mini-courses offered please visit
`http://scc.stat.ucla.edu/mini-courses`.