R Programming

Basic Concepts

The S Language

- S is a language and computational environment designed specifically for carrying out "statistical" computations.
- It was designed by statisticians so that they could easily develop and deploy new statistical methodology.
- It brings good ideas on numerical methods, graphics and language design together in a single useful package.
- It represents a philosophy of *turning ideas into programs* quickly and easily.
- The S language was developed at Bell Laboratories by John Chambers and his collaborators *Rick Becker*, *Allan Wilks* and *Duncan Temple Lang* over the years from 1975 to 1998.

The R Language

- R is a computer language (and associated computational environment) for carrying out statistical computations.
- It appears similar to the S language developed at Bell Laboratories, but the similarity was retrofitted on top of quite different looking piece of software.
- R was created by *Robert Gentleman* and *Ross Ihaka* at the University of Auckland as a testbed for trying out some ideas in statistical computing.
- It has now evolved into a fully featured tool for carrying out statistical (and other) computations.

The R Project

- The *R Project* is an international collaboration of researchers in statistical computing.
- There are roughly 20 members of the "R Core Team" who maintain and enhance R.
- Releases of the R environment are made through the CRAN (comprehensive R archive network) twice per year.
- The software is released under a "free software" license, which makes it possible for anyone to download and use it.
- There are over 3500 extension *packages* that have been contributed to CRAN.

The original R developers plotting world domination.

International collaboration in full swing.

Basic R Concepts

- R is a computer language which is processed by a special program called an *interpreter*. This program reads and evaluates R language *expressions*, and prints the values determined for the expressions.
- The interpreter indicates that it is expecting input by printing its *prompt* at the start of a line. By default the R prompt is a *greater than* sign `>`.
- On UNIX or LINUX machines you can start R by typing the command "`R`" to a command interpreter prompt in a terminal window.
- Better yet, you can run R within an Emacs window by typing `M-x R`.

Using R as a Calculator

- Users type *expressions* to the R interpreter.
- R responds by computing and printing the answers.

```
> 1 + 2
[1] 3

> 1/2
[1] 0.5

> 17^2
[1] 289
```

Grouping and Evaluation

- Normal arithmetic rules apply; multiplication and division occur before addition and subtraction.

```
> 1 + 2 * 3
[1] 7
```

- The default evaluation rules can be can be overriden by using parentheses.

```
> (1 + 2) * 3
[1] 9
```

Operator Precedence

- R's basic operators have the following precedences (listed in highest-to-lowest order).

`^`	exponentiation
`-  +`	unary minus and plus
`:`	sequence operator
`%/%  %%`	integer division, remainder
`*  /`	multiplication, division
`+  -`	addition, subtraction

  (We'll discuss the `:`, `%/%` and `%%` operators later.)

- Operations with higher precedence take place before those with lower precidence.

Evaluation Order

- Evaluation of equal precedence operations takes place left-to-right (except for exponentiation, which takes place right-to-left).

```
> 2^3^2
[1] 512

> (2^3)^2
[1] 64

> 2^(3^2)
[1] 512
```

Simple R Expressions

```
> 1 + 2
[1] 3

> 1/2
[1] 0.5

> 17^2
[1] 289

> 1 + 2 * 3
[1] 7

> (1 + 2) * 3
[1] 9
```

R Functions

```
> sqrt(2)
[1] 1.414214

> log(10)
[1] 2.302585

> log10(10)
[1] 1

> sin(1)
[1] 0.841471

> 4 * atan(1)
[1] 3.141593
```

Assignment

- Values are stored by *assigning* them a name.
- The resulting name-value pair is called a *variable*.
- The statements:

```
> z = 17
```

```
> z <- 17
```

```
> 17 -> z
```

  all store the value `17` under the name `z`.

- The "`=`" assignment form is preferred to the "`<-`" one.

Using Variables

- Variables can be used in expressions in the same way as numbers.
- For example,

```
> z = 17

> z = z + 23

> z
[1] 40
```

Numeric Vectors

R has the ability to work with *vectors* of values. Individual values can be combined into a vector by using the the `c` function.

```
> x = c(1, 2, 3, 4)
```

Numeric Vectors

R has the ability to work with *vectors* of values. Individual values can be combined into a vector by using the the `c` function.

```
> x = c(1, 2, 3, 4)
```

The values in a vector can be viewed by simply typing the name of the vector.

```
> x
[1] 1 2 3 4
```

Sequences

One very useful way of generating vectors is using the sequence operator `:`. The expression $n_1 : n_2$, generates the sequence of integers ranging from n_1 to n_2.

```
> 1:50
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13
[14] 14 15 16 17 18 19 20 21 22 23 24 25 26
[27] 27 28 29 30 31 32 33 34 35 36 37 38 39
[40] 40 41 42 43 44 45 46 47 48 49 50
```

Sequences

One very useful way of generating vectors is using the sequence operator `:`. The expression $n_1 : n_2$, generates the sequence of integers ranging from n_1 to n_2.

```
> 1:50
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13
[14] 14 15 16 17 18 19 20 21 22 23 24 25 26
[27] 27 28 29 30 31 32 33 34 35 36 37 38 39
[40] 40 41 42 43 44 45 46 47 48 49 50
```

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

Combining Vectors

The function `c` can be used to combine both vectors and scalars into larger vectors.

```
> x = c(1, 2, 3, 4)

> c(x, 10)
[1]  1  2  3  4 10

> c(x, x)
[1] 1 2 3 4 1 2 3 4
```

In fact, R stores scalar values like 10 as vectors of length one, so that both arguments in the expression above are vectors.

Vector Arithmetic

Because "everything is a vector" it is natural to expect that the type of arithmetic manipulation carried out in the previous section to also work with more general vectors.

```
> 2 * x + 1
[1] 3 5 7 9
```

The result is as you would expect. The values in `x` have been doubled and had one added to them. Applying mathematical functions also yields the results you would expect.

```
> sqrt(x)
[1] 1.000000 1.414214 1.732051 2.000000
```

The Recycling Rule

What is less obvious about vector arithmetic is what happens when vectors of different sizes are combined.

```
> c(1, 2, 3, 4) + c(1, 2)
[1] 2 4 4 6
```

The Recycling Rule

What is less obvious about vector arithmetic is what happens when vectors of different sizes are combined.

```
> c(1, 2, 3, 4) + c(1, 2)
[1] 2 4 4 6
```

This result is explained by the *recycling rule* used by R to define the meaning of this kind of calculation.

> *First enlarge the shorter vector by recycling its elements, then combine the vectors element by element.*

The Recycling Rule

For the example:

```
> c(1, 2, 3, 4) + c(1, 2)
[1] 2 4 4 6
```

here is how the recycling rule works.

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \overset{recycle}{\Longrightarrow} \quad \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 1 \\ 2 \end{bmatrix} \quad \overset{add}{\Longrightarrow} \quad \begin{bmatrix} 2 \\ 4 \\ 4 \\ 6 \end{bmatrix}$$

Binary Operations

The following binary operations all obey the recycling rule.

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`^`	raising to a power
`%%`	modulo (remainder after division)
`%/%`	integer division

The Modulo and Integer Division Operators

The modulo operator is useful in integer computations,

```
> 11 %% 3
[1] 2

> 1:10 %% 2
 [1] 1 0 1 0 1 0 1 0 1 0
```

but it can also be used with more general numbers

```
> 13.5 %% 2
[1] 1.5

> 13.5 %/% 2
[1] 6
```

Elements and Subsets

Individual elements can be extracted from vectors by specifying their index. The third element can be extracted from `x` as follows.

```
> x[3]
[1] 3
```

It is also possible to extract subvectors by specifying vectors of subscripts.

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

The sequence operator provides a useful way of extracting consecutive elements from a vector.

```
> x[1:3]
[1] 1 2 3
```

Negative Subscripts

Sometimes, rather than extracting the elements of a vector corresponding to a particular set of subscripts, we may want to extract the elements which do not correspond to a given set of subscripts. This can be done by using a set of negative subscripts. For example `x[c(-1, -3)]` would extract all the elements from `x` except the first and third.

```
> x[c(-1, -3)]
[1] 2 4
```

It is not possible to mix positive and negative subscripts.

Changing Vector Subsets

As well extracting the values at particular positions in a vector, it is possible to reset their values. This is done by putting the subset to be modified on the left-hand side of the assignment with the replacement value(s) on the right.

```
> y = 1:10
> y[4:6] = 0
> y
 [1]  1  2  3  0  0  0  7  8  9 10
```

Special Numerical Values – Infinity

When one is divided by zero, the result is infinite. This kind of special result is also produced by R.

```
> 1 / 0
[1] Inf
```

Here, `Inf` represents positive infinity. There is also a negative infinity.

```
> -1 / 0
[1] -Inf
```

Special Numerical Values – Infinity

Infinities have all the properties you would expect. For example

```
> 1 + Inf
[1] Inf
```

and

```
> 1000 / Inf
[1] 0
```

Special Numerical Values – Not a Number

R also has a special value, called `NaN`, which indicates that a numerical result is undefined.

```
> 0 / 0
[1] NaN
```

and subtracting infinity from infinity.

```
> Inf - Inf
[1] NaN
```

Some mathematical functions will also produce `NaN` results.

```
> sqrt(-1)
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
```

Special Numerical Values – Not Available

R has a particular value which is used to indicate that a value is *missing* or *not available*. The value is indicated by `NA`. Any arithmetic expression which contains `NA` will produce `NA` as a result.

```
> 1 + sin(NA)
[1] NA
```

The value `NA` is usually used for statistical observations where the value could not be recorded. Such as when a survey researcher visits a house and no one is home.

Summary Functions – min, max and range

The functions `min` and `max` return the minimum and maximum values contained in any of their arguments, and the function `range` returns a vector of length 2 containing the minimum and maximum of the values in the arguments.

```
> max(1:100)
[1] 100

> max(1:100, Inf)
[1] Inf

> range(1:100)
[1]   1 100
```

Summary Functions – `sum` and `prod`

The functions `sum` and `prod` compute the sum and product of all the elements in their arguments.

```
> sum(1:100)
[1] 5050

> prod(1:10)
[1] 3628800
```

Summary Functions and NA

In any of these summary functions the presence of NA and NaN values in any of the arguments will produce a result which is NA and NaN.

```
> min(NA, 100)
[1] NA
```

NA and NaN values can be disregarded by specifying an additional argument of na.rm=TRUE.

```
> min(10, 20, NA, na.rm = TRUE)
[1] 10
```

Cumulative Summaries

There are also cumulative variants of the summary functions.

```
> cumsum(1:10)
 [1]  1  3  6 10 15 21 28 36 45 55

> cumprod(1:6)
[1]   1   2   6  24 120 720

> cummax(1:10)
 [1]  1  2  3  4  5  6  7  8  9 10

> cummin(1:10)
 [1] 1 1 1 1 1 1 1 1 1 1
```

These cummulative summary functions do not have a `na.rm` argument.

Parallel Summary Functions

Finally, there are parallel versions of the minimum and maximum functions. These take a number of vector arguments and apply the recycling rule to them, and then compute the summaries across the corresponding values of the arguments.

```
> pmin(c(1, 10), c(10, 2))
[1] 1 2

> pmax(0, c(-1, 0, 1))
[1] 0 0 1
```

The parallel versions of the minimum and maximum functions do accept an `na.rm` argument.

Logical Vectors

Logical vectors contain the values `TRUE` and `FALSE`.

```
> b = c(TRUE, TRUE, FALSE, FALSE)
```

Extracting and modifying elements of logical vectors takes place in exactly the same way as extracting and modifying subsets of numeric vectors.

```
> b[1:3]
[1]  TRUE  TRUE FALSE

> b[1] = FALSE
> b
[1] FALSE  TRUE FALSE FALSE
```

Generating Logical Values

Logical values are often produced as the result of assertions made about other types of value.

```
> 3 < 4
[1] TRUE

> 3 > 4
[1] FALSE
```

Comparison operators can also be applied to vectors of values.

```
> c(1, 2, 3, 4) < 3
[1]  TRUE  TRUE FALSE FALSE
```

Comparison Operators

The full set of logical operators appears below. The operators all return logical values.

`<`	less than
`<=`	less or equal
`==`	equal
`!=`	not equal
`>`	greater than
`>=`	greater or equal

Logical Conjunctions

Logical values can be combined with boolean operators — `&` which denotes logical "and," `|` which denotes logical "or" and `!` which denotes unary "not."

```
> x = c(1, 2, 3, 4)
```

```
> x < 2 | x > 3
[1]  TRUE FALSE FALSE  TRUE
```

The functions `any` and `all` can be used to see if any or all the elements of a logical vector are true.

```
> all(x > 0)
[1] TRUE
```

```
> any(x > 2)
[1] TRUE
```

Negation

The operator `!` is used to indicate logical negation. It turns `TRUE` values into `FALSE` and `FALSE` ones into `TRUE`.

```
> !(3 < 4)
[1] FALSE

> TRUE & ! FALSE
[1] TRUE
```

Logic and NA Values

Logical vectors can contain NA values. This produces a three-valued logic.

x & y		y		
		TRUE	FALSE	NA
x	TRUE	TRUE	FALSE	NA
	FALSE	FALSE	FALSE	FALSE
	NA	NA	FALSE	NA

x | y		y		
		TRUE	FALSE	NA
x	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	NA
	NA	TRUE	NA	NA

Logical Subsetting

It is possible to extract subsets of vectors by using logical values.

As an example, consider the subsetting expression `x[x > 2]`.

The expression `x > 2` defines a logical vector with the same length as `x`, and the subset contains those values from `x` which correspond to `TRUE` values in this vector.

```
> x = c(1, 2, 3, 4)

> x[x > 2]
[1] 3 4
```

Conditional Selection

The `ifelse` function makes possible to choose between the elements of two vectors based on the values of a logical vector.

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

The values being selected are recycled if necessary.

```
> x = 1:10
> ifelse(x > 5, x, 0)
 [1]  0  0  0  0  0  6  7  8  9 10
```

Character Vectors

Character vectors have elements which are character strings.

Strings are sequences of characters enclosed within double quotes, `"like this"`, or single quotes, `'like this'`.

Character vectors can be created with the `c` function

```
> s = c("first", "second", "third")
```

and manipulated in exactly the same way as other vectors.

```
> s[1:2]
[1] "first"  "second"

> s[1] = "initial"
> s
[1] "initial" "second"  "third"
```

String Manipulation

The function `nchar` returns the lengths the character strings in a character vector.

```
> s
[1] "initial" "second"  "third"

> nchar(s)
[1] 7 6 5
```

Substrings can be extracted with `substring`.

```
> substring(s, 1, 2:3)
[1] "in"  "sec" "th"
```

(Note the use of recycling here.)

Pasting Strings Together

Strings can be “glued together” by using paste.

```
> paste("First", "Second", "Third")
[1] "First Second Third"
```

By default, strings a joined with a space between them, but it is possible to use any sequence of characters as a separator.

```
> paste("First", "Second", "Third", sep = ":")
[1] "First:Second:Third"
```

Using an empty separator string will concatenate the strings.

```
> paste("First", "Second", "Third", sep = "")
[1] "FirstSecondThird"
```

Pasting Vectors

Paste will work with vectors as well as simple strings. The result is defined by the recycling rule.

```
> paste(s, "element", sep = "-")
[1] "initial-element" "second-element"
[3] "third-element"
```

An additional argument, `collapse`, will cause the elements of the result of elementwise pasting to be joined together with the given specified separator.

```
> paste(s, collapse = " -> ")
[1] "initial -> second -> third"
```

Complex-Valued Vectors

Vectors can contain values which a complex numbers. These are written in the standard notation.

```
> z = 10+20i
> z
[1] 10+20i
```

Arithmetic carried out on complex values obeys the rules of complex arithmetic.

```
> z = -1+0i
> sqrt(z)
[1] 0+1i
```

Vector Mode and Length

Vectors have an associated *mode* (one of `"logical"`, `"numeric"`, `"complex"` or `"character"`) and *length*.

```
> mode(1:10)
[1] "numeric"
> length(1:100)
[1] 100
```

Each of the four modes of vector have corresponding functions (`logical`, `numeric`, `complex` and `character`) which can be used to create vectors with that mode.

```
> logical(5)
[1] FALSE FALSE FALSE FALSE FALSE

> numeric(5)
[1] 0 0 0 0 0
```

Creating Vectors

A function called `vector` can also be used to create vectors of any of the four basic types.

```
> vector("numeric", 5)
[1] 0 0 0 0 0

> vector("logical", 5)
[1] FALSE FALSE FALSE FALSE FALSE
```

It is possible to create vectors with a length of zero using any of the functions listed above.

```
> numeric(0)
numeric(0)
```

No indexing information is printed for zero-length vectors (because there are no elements to index).

Type Coercion

Types are automatically converted or *coerced* in R if the need arises.

```
> c(TRUE, 17)
[1]  1 17

> c(TRUE, 17, "twelve")
[1] "TRUE"   "17"     "twelve"
```

There is a natural ordering of the vector modes in R. Logical values can be interpreted as numeric if we take the value of `FALSE` to be 0 and the value of `TRUE` as 1, numeric values can be interpreted as complex by taking the imaginary part to be zero and all modes can be interpreted as character values by just taking their printed representation.

Type Coercion Idioms

Coercion is at the heart of many idioms in R. A common example is counting the number of vector elements for which a condition is true. For example, the expression `sum(x > 5)` counts the number of elements in `x` which are greater than 5.

```
> x = 1:10

> sum(x > 5)
[1] 5

> cumsum(x > 5)
 [1] 0 0 0 0 0 1 2 3 4 5
```

Explicit Coercion

R will automatically coerce vectors using the natural ordering of vector modes. Other coercions must be carried out using explicit coercion with the functions `as.logical`, `as.numeric`, `as.complex` and `as.character`.

```
> "1" + "2"
Error in "1" + "2" : non-numeric argument to
                     binary operator

> as.numeric("1") + as.numeric("2")
[1] 3
```

Named Vectors

Vectors of any of the basic types can be augmented by providing names for their elements.

```
> v = 1:4
> names(v) = c("A", "B", "C", "D")

> v
A B C D
1 2 3 4
```

The names attached to a vector can be extracted by using the `names` function.

```
> names(v)
[1] "A" "B" "C" "D"
```

Subsetting Using Names

Names can also be used to extract elements and subvectors. If `v` is defined by the statements

```
> v = c(1, 2, 3, 4)
> names(v) = c("A", "B", "C", "D")
```

then we can extract subsets of the elements as follows

```
> v["A"]
A
1

> v[c("A", "D")]
A D
1 4
```

Lists

The elements of vectors must all be of the same basic type. Lists provide a way of storing things of different types in a single object.

```
> lst = list(10, "eleven", TRUE)
```

This list has three elements. The first element is the number `10`, the second is the character string `"eleven"` and the third is the logical value `TRUE`.

Printing Lists

The elements of lists are printed in a special way to make their structure clearer.

```
> lst
[[1]]
[1] 10

[[2]]
[1] "eleven"

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

Named Lists

The elements of a list can be named. This can either be done with the `names` function, or directly in the call to `list`. When elements are named, the name is displayed in place of the indexing information.

```
> list(a=10, b="eleven", TRUE)
$a
[1] 10

$b
[1] "eleven"

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

Elements and Subsets of Lists

When asking for a single element of a list, use `[[ ]]`. When asking for a subset, use `[ ]`.

```
> lst = list(10, "eleven", TRUE)

> lst[[1]]
[1] 10

> lst[1]
[[1]]
[1] 10
```

The first result here is the element 10 extracted from the list. The second is a list containing that element.

Extracting Named Elements

Named elements can be extracted from lists in the same way that named elements can be extracted from vectors.

```
> lst = list(a=10, b="eleven", TRUE)
> lst[["a"]]
[1] 10
```

This is such a common operation, there is a shorthand for it.

```
> lst$a
[1] 10
```

(This kind of shorthand is referred to *syntactic sugar*.)

The NULL Object

There is a special object in R called the `NULL` object which is used to represent "nothing." The `NULL` object has a length of zero and can be freely coerced to either a vector or list of length zero.

```
> length(NULL)
[1] 0

> as.numeric(NULL)
numeric(0)

> as.list(NULL)
list()
```