

Programming with R



Educational Materials

©2004 R. Ihaka and R. Gentleman

## The S Language

- The S language has been developed since the late 1970s by John Chambers and his colleagues at Bell Labs.
- The language has been through a number of major changes but it has been relatively stable since the mid 1990s
- The language combines ideas from a variety of sources (e.g. *Awk*, *Lisp*, *APL*...) and provides an environment for quantitative computations and visualization.

## S Implementations

- S-Plus a commercialization of the Bell Labs code.
- R an independent open source version that was originally developed at the University of Auckland but which is now developed by a world wide group of developers.
- Each version has advantages and problems.

## References

- *The New S Language, Statistical models in S, Programming with Data*, by John Chambers and various co-authors.
- *Modern Applied Statistics, S Programming* by W. N. Venables and B. D. Ripley.
- *Introductory Statistics with R* by P. Dalgaard.
- *Data Analysis and Graphics Using R* by J. Maindonald and J. Braun.

## The Nature of Programming

The task of writing a program has two sub-tasks:

1. describing precisely what is to be done
2. describing the data that will be used

These cannot be done separately, the choices made in solving one subtask affect the other.

*algorithms + data structures = programs*

- N. Wirth

## Data Structures

- S has a rich set of *self-describing* data structures.
- These structures describe both the data that can be processed by the language and the language itself.
- Since the structures are self-describing there is no need for the user to declare the types of the variables.
- It is possible that in future versions of S an optional set of type declarations will be supported. These could, for example, support the efficient generation of byte-compiled code.

## Atomic Data Structures

- The most basic data type in S is the *atomic vector*.
- Such vectors contain an indexed set of values that are all of the same type:
  - *logical*
  - *numeric*
  - *complex*
  - *character*
- The numeric type can be further broken down into *integer*, *single*, and *double* types (but this is only important when making calls to foreign functions, eg. C or Fortran.)

## Creating Vectors

- Many S functions create vectors to hold the results they compute.
- there are also functions which can be used to create “empty” vectors.

```
> vector("numeric", 10)
[1] 0 0 0 0 0 0 0 0 0 0
> numeric(10)
[1] 0 0 0 0 0 0 0 0 0 0
> vector("logical", 0)
logical(0)
```



## Patterned Vectors

- The functions `rep` and `seq` can be used to create vectors containing patterns of values.
- Simple replication.

```
> rep(1:2, 3)
[1] 1 2 1 2 1 2
```

- More complex replication.

```
> rep(c("A", "B"), c(2, 3))
[1] "A" "A" "B" "B" "B"
> rep(c("A", "B"), each = 3)
[1] "A" "A" "A" "B" "B" "B"
```

## Vector Structures

- S retains the notion of *vector structures* from its earliest implementation.
- A vector structure is a vector with some additional information attached to it as an *attribute list*.
- Most uses of vector structures have been deprecated in favor of object-oriented alternatives.
- The major remaining use of vector structures is as the representation of arrays.

## Arrays

- S regards an array as consisting of a vector containing the array's elements together with a dimension (or `dim`) attribute.
- A vector can be given dimensions by using the functions `array` or `matrix`, or by directly attaching them with the `dim` function.
- The elements in the underlying vector correspond to the elements of the array with earlier subscripts moving faster.

## Examples

- Direct array creation.

```
> x = 1:10  
> dim(x) = c(2, 5)  
> x
```

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

- Array creation using `matrix`.

```
> x = matrix(1:10, nrow = 2)
```

## Naming

- The elements of a vector can (and often should) be given names by using the `names` function.

```
> x = c(10, 20)
> names(x) = c("First", "Second")
> x

  First Second
    10     20
```

- Array extents can be named by using the `dimnames` function or the `dimnames` argument to `matrix` or `array`. Extent names are given as a list, with each list element being a vector of names for the corresponding extent.

## Example

```
> x = array(1:8, dim = c(2, 2, 2))  
> dimnames(x) = list(c("A", "B"), NULL, c("X", "Y"))  
> x
```

```
, , X
```

```
  [,1] [,2]  
A     1     3  
B     2     4
```

```
, , Y
```

```
  [,1] [,2]  
A     5     7  
B     6     8
```

## Subsetting

- One of the most powerful features of S, is its ability to manipulate subsets of vectors and arrays.
- The S subsetting facility is derived from, and extends that of, APL.
- Subsetting is indicated by [, ]. Note that the first of these is actually a function (try `get("[")`).

## Subsetting with Positive Indices

- A subscript consisting of a vector of positive integer values is taken to indicate a set of indexes to be extracted.

```
> x = 1:10
```

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

```
[1] 1 2 3
```

- A subscript which is larger than the length of the vector being subsetting produces an NA in the returned value.

```
> x[9:11]
```

```
[1] 9 10 NA
```



## Subsetting with Positive Indices

- Subscripts which are zero are ignored and produce no corresponding values in the result.

```
> x[0:1]
[1] 1
```

- Subscripts which are NA produce an NA in the result.

```
> x[c(1, 2, NA)]
[1] 1 2 NA
```

## Assignments with Positive Indices

- Subset expressions can appear on the left side of an assignment. In this case the given subset is assigned the values on the right (recycling the values if necessary).

```
> x[1:3] <- 10
```

```
> x
```

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

- If a zero or NA occurs as a subscript in this situation, it is ignored.

## Subsetting with Negative Indexes

- A subscript consisting of a vector of negative integer values is taken to indicate the indexes which are not to be extracted.

```
> x[-(1:3)]
```

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

- Subscripts which are zero are ignored and produce no corresponding values in the result.
- NA subscripts are not allowed.
- Positive and negative subscripts cannot be mixed.

## Assignments with Negative Indexes

- Negative subscripts can appear on the the left side of an assignment. In this case the given subset is assigned the values on the right (recycling the values if necessary).

```
> x = 1:10  
> x[-(2:4)] = 10  
> x
```

```
[1] 10  2  3  4 10 10 10 10 10 10
```

- Zero subscripts are ignored.
- NA subscripts are not permitted.

## Subsetting by Logical Predicates

- Vector subsets can also be specified by a logical vector of TRUEs and FALSEs.

```
> x = 1:10  
> x[x > 5]  
[1] 6 7 8 9 10
```

- NA values used as logical subscripts produce NA values in the output.
- The subscript vector can be shorter than the vector being subsetted. The subscripts are recycled in this case.
- The subscript vector can be longer than the vector being subsetted. Values selected beyond the end of the vector produce NAs.

## Subsetting by Name

- If a vector has named elements, it is possible to extract subsets by specifying the names of the desired elements.

```
> x = 1:10
> names(x) = LETTERS[1:10]
> x[c("A", "D", "F")]

A D F
1 4 6
```

- If several elements have the same name, only the first of them will be returned.
- Specifying a non-existent name produces an NA in the result.

## Exercises

1. Determine (precisely) how S handles non-integer subscripts (e.g. 1.2). How might this produce problems?

2. What value do the following expressions produce.

```
x = 1:10
```

```
x[-11]
```

3. How could you choose all elements of a vector which have odd subscripts? Even subscripts?

4. How are complex subscripts treated?

## Subsetting matrices

- when subsetting a matrix, missing subscripts are treated as if all elements are named; so `x[1,]` corresponds to the first row and `x[,3]` to the third column.
- for arrays, the treatment is similar, for example `y[,1,]`.
- these can also be used for assignment, `x[1,]=20`



## Subsetting Arrays

- Rectangular subsets of arrays obey similar rules to those which apply to vectors.
- One point to note is that arrays can be treated as either matrices or vectors. This can be quite useful.

```
> x = matrix(1:9, ncol = 3)
```

```
> x[x > 6]
```

```
[1] 7 8 9
```

```
> x[row(x) > col(x)] = 0
```

```
> x
```

```
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    0    5    8
[3,]    0    0    9
```

## Mode and Storage Mode

- The functions `mode` and `storage.mode` return information about the *types* of vectors.

```
> mode(1:10)
```

```
[1] "numeric"
```

```
> storage.mode(1:10)
```

```
[1] "integer"
```

```
> mode("a string")
```

```
[1] "character"
```

```
> mode(TRUE)
```

```
[1] "logical"
```

## Automatic Type Coercion

- S will automatically coerce data to the appropriate type when this is necessary.

```
> 1 + TRUE
```

```
[1] 2
```

Here the logical value `TRUE` is coerced to the numeric value of 1 so that addition can take place.

- Some common coercions are:
  - logical  $\longrightarrow$  numeric
  - logical, numeric  $\longrightarrow$  complex
  - logical, numeric, complex  $\longrightarrow$  character
  - numeric, complex  $\longrightarrow$  logical

## Type Coercion and NA values

- Logical values can be coerced to any other atomic mode. Because of this, the constant NA has been made a logical value.

```
> mode(NA)
[1] "logical"
```

- When NA is used in an expression, the mode of the result is usually determined by the mode of the other operands.

```
> 1 + NA
[1] NA
> mode(1 + NA)
[1] "numeric"
```

## An R/ S-Plus Difference

- S-Plus does not have an NA indicator for character strings. It coerces NA values to the character string "NA". There are potential problems with this approach.

```
> is.na(as.character(NA))
```

```
[1] F
```

- R does have a special NA value for character strings and so does differentiate between NA and "NA".

```
> is.na(as.character(NA))
```

```
[1] TRUE
```

## Explicit Type-Coercion

- The functions `as.logical`, `as.integer`, etc. return a copy of values passed to them, coerced to the specified type.

```
> as.numeric(c("1", "10.5", "text"))
```

```
[1] 1.0 10.5 NA
```

- **Warning:** These functions discard all labeling and dimensioning information.

```
> x = 1:5
```

```
> names(x) = LETTERS[1:5]
```

```
> as.character(x)
```

```
[1] "1" "2" "3" "4" "5"
```

## Explicit Type-Coercion

- The functions `mode` and `storage.mode` (or more precisely `mode<-` and `storage.mode<-`) can be used to alter the storage mode of a variable.

```
> x = 1:5
> names(x) = LETTERS[11:15]
> x

 K L M N O
 1 2 3 4 5

> storage.mode(x) = "character"
> x

  K   L   M   N   O
"1" "2" "3" "4" "5"
```

- These functions preserve attributes like labeling and dimensioning.

## Vectorized Arithmetic

- Most arithmetic operations in the S language are *vectorized*. That means that the operation is applied element-wise.

```
> 1:3 + 10:12
```

```
[1] 11 13 15
```

- In cases where one operand is shorter than the other the short operand is recycled, until it is the same length as the longer operand.

```
> 1 + 1:5
```

```
[1] 2 3 4 5 6
```

```
> paste(1:5, "A", sep = "")
```

```
[1] "1A" "2A" "3A" "4A" "5A"
```

- Many operations which need to have explicit loops in other languages do not need them with S. You should vectorize any functions you write.



## Lists

- In addition to atomic vectors, S has a number of *recursive* data structures. Among the important members of this class are *lists* and *environments*.
- A list is a vector which can contain vectors and other lists (in fact arbitrary R objects) as elements. In contrast to atomic vectors, whose elements are homogeneous, lists and environments contain heterogeneous elements.

```
> lst = list(a = 1:3, b = "a list")
> lst

$a
[1] 1 2 3

$b
[1] "a list"
```

## Environments

- One difference between lists and environments is that there is no concept of ordering in an environment. All objects are stored and retrieved by **name**.

```
> e1 = new.env(hash = TRUE)
> assign("a", 1:3, e1)
> assign("b", "a list", e1)
> ls(e1)

[1] "a" "b"
```

- Another difference is that for lists partial matching of names is used, for environments it is **not**.

## Subsetting and Lists

- Lists are useful as containers for grouping related things together (many S functions return lists as their values).
- Because lists are a recursive structure it is useful to have two ways of extracting subsets.
- The `[ ]` form of subsetting produces a sub-list of the list being subsetted.
- The `[[ ]]` form of subsetting can be used to extract a single element from a list.

## List Subsetting Examples

- Using the [ ] operator to extract a sublist.

```
> lst[1]
```

```
$a
```

```
[1] 1 2 3
```

- Using the [[ ]] operator to extract a list element.

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

```
[1] 1 2 3
```

- As with vectors, indexing using logical expressions and names are also possible.

## List Subsetting by Name

- The dollar operator provides a short-hand way of accessing list elements by name. This operator is different from all other operators in S, it does not evaluate its second operand (the string).

```
> lst$a
[1] 1 2 3
> lst[["a"]]
[1] 1 2 3
```

- For these accessors partial matching (!) is used.

## Environment Accessing Elements

- Access to elements in environments can be through, `get`, `assign`, `mget`.
- You can also use the dollar operator and the `[[ ]]` operator, with character arguments only. No partial matching is done.

```
> e1$a
```

```
[1] 1 2 3
```

```
> e1[["b"]]
```

```
[1] "a list"
```

## Assigning values in Lists and Environments

- Items in lists and environments can be replaced in much the same way as items in vectors are replaced.

```
> lst[[1]] = list(2, 3)
> lst[[1]]

[[1]]
[1] 2

[[2]]
[1] 3

> e1$b = 1:10
> e1$b

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

## Data Frames

- Data frames are a special S structure used to hold a set of related variables. They are the S representation of a statistical *data matrix*. Typically the cases are the rows and the variables are the columns (and this is how `data.frames` are implemented).
- Data frames can be treated like matrices, and indexed with two subscripts. The first subscript refers to the observation, the second to the variable.
- In fact, this is an illusion maintained by the S object system. Data frames are really lists, and list subsetting can also be used on them.



## Control-Flow

- S has a number of special control-flow structures which make it possible to express quite complex computations in the S language.
- Iteration is provided by the `for`, `while` and `repeat` statements.
- Conditional evaluation is provided by the `if` statement and the `switch` function.
- Of these capabilities, `for` and `if` are by far the most commonly used.

## for statements

- for statements have the basic form:

```
for ( var in vector ) {  
  statements  
}
```

The effect of this is to set the value of the variable *var* successively to each of the elements in *vector* and then evaluate all *statements*.

- This looks similar to the *for* statement found in languages such as C and C++, but it is closer to the *foreach* statement in *Perl*.

## Examples

- Summing the values in a vector (C style)

```
sum = 0
for( i in 1:length(x) ) {
    sum = sum+x[i]
}
```

- Summing the values in a vector (Perl style).

```
sum = 0
for( elt in x) {
    sum = sum + elt
}
```

- The second of these is more efficient.

## Flow Control

- **break**: when a **break** statement is encountered, evaluation halts and transfers to the first statement outside of the enclosing **for**, **while** or **repeat** loop.
- **next**: when a **next** statement is encountered then evaluation of the current iteration halts, and the loop is entered with the looping index incremented by one.

```
> for (i in 1:5) if (i == 3) next else print(i)
```

```
[1] 1
```

```
[1] 2
```

```
[1] 4
```

```
[1] 5
```

```
> for (i in 1:4) if (i == 3) break else print(i)
```

```
[1] 1
```

```
[1] 2
```

## if expressions

- If statements have the basic form:

```
if( test ) {  
  statements  
} else {  
  statements  
}
```

- If the first element of *test* is true, the first group of statements is executed, otherwise, the second group of statements is executed.
- The **else** clause is optional.

## Examples

- Here is a typical use of `if`.

```
if (any(x < 0) )  
    stop("negative values encountered")
```

- Here is a choice between actions.

```
r = if( all( x >= 0 ) )  
    sqrt(x) else  
    sqrt( x + 0i)
```

The layout here is important. The `else` clause must fall on the same line as the preceding statement (assuming the code above is not enclosed within a set of braces `{, }`).

## The switch Function

- The `switch` function uses the value of its first argument to determine which of its remaining arguments to evaluate and return. The first argument can be either an integer index, or a character string to be used in matching one of the following arguments.

```
center = function(x, type) {  
  switch(type,  
    mean = mean(x),  
    median = median(x),  
    trimmed = mean(x, trim = .1))  
}
```

- Calling `center` with `type=1` or `type="mean"` produce the same result.

## Efficiency Issues

- S provides a full set of control-flow statements but they can be slow because S is an interpreted language.
- R is somewhat faster than S-Plus at looping, but it is still about two orders of magnitude slower than compiled C or Fortran.
- For time-critical applications, it can be useful to obtain measures of how fast a particular piece of code runs as a guide for choosing a good computational method.
- The function `system.time` provides timing information. But do be careful, timing is not completely simple.



## Timing Experiments

- Timing experiments can be a very good way of checking alternative ways of carrying out computations.

```
> sum = 0
> x = rnorm(10000)
> system.time({
+   s = 0
+   for (i in 1:length(x)) s = s + x[i]
+ })

[1] 0.05 0.00 0.08 0.00 0.00

> system.time({
+   s = 0
+   for (v in x) s = s + v
+ })

[1] 0.03 0.00 0.04 0.00 0.00
```

## Profiling

- Once you know that something takes a long time, you might want to find out what part of your function is taking that long time.
- The functions `Rprof` and `summaryRprof` can be used to profile R commands and to provide some insight into where the time is being spent.

```
Rprof()
```

```
<commands go here>
```

```
Rprof(NULL)
```

```
summaryRprof()
```

## Debugging

- When you have an error in your code there are some tools that can help. `traceback`, `debug` and `browser` all provide different tools.
- When running either `debug` or `browser` there is a special interpreter that intercepts some commands; in particular `n`, `where`, and `Q`.
- The option `error` can be set to `recover`, in which case the debugger will be invoked on error.
- Warnings can be converted into errors (if desired), see `options` for more details.

## The apply Family

- A natural programming construct in S is to *apply* the same function to elements of a list, of a vector, rows of a matrix, or elements of an environment.
- The members of this family of functions are different with regard to the data structures they work on and how the answers are dealt with.
- Some examples, `apply`, `sapply`, `lapply`, `mapply`, `eapply`.

## Using apply

- `apply` applies a function over the margins of an array.
- For example,
  - > `apply(x, 2, mean)`  
computes the column means of a matrix `x`, while
  - > `apply(x, 1, median)`  
computes the row medians.
- (`apply`) is implemented in a way which avoids the overhead associated with looping. (But it is still slow and you might use `rowSums` or `colSums`).

## An Additive Table Decomposition

- Given the data matrix  $x$ , this code carries out an *overall + row + column* decomposition.

```
overall = mean(x)
row = apply(x, 1, mean) - overall
col = apply(x, 2, mean) - overall
res = x - outer(row, col, "+") - overall
```

- The generalized outer product function `outer` is used here to produce a matrix, the same shape as  $x$ , containing the appropriate sums of row and column effects.
- Something similar can be used to produce a simple implementation of median polish.

## Writing Functions

- Writing S functions provides a means of adding new functionality to the language.
- Functions that a user writes have the same status as those which are provided with S.
- Reading the functions provided with the S system is a good way to learn how to write functions.
- If a user chooses she can make modifications to the system functions and use her modified ones, in preference to the system ones.

## A Simple Function

- Here is a function that computes the square of its argument.

```
> square = function(x) x * x
```

```
> square(10)
```

```
[1] 100
```

- Because the underlying arithmetic is vectorized, so is this function.

```
> square(1:4)
```

```
[1] 1 4 9 16
```



## Composition of Functions

- Once a function is defined, it is possible to call it from other functions.

```
> sumsq = function(x) sum(square(x))
```

```
> sumsq(1:10)
```

```
[1] 385
```

## Example: Factorials

- Iteration.

```
> facI = function(n) {  
+   ans = 1  
+   for (i in seq(n)) ans = ans * i  
+   ans  
+ }  
> facI(5)  
  
[1] 120
```

- Recursion.

```
> facR = function(n) if (n <= 0) 1 else n * facR(n - 1)  
> facR(5)  
  
[1] 120
```

## Examples: Factorials

- Vectorized arithmetic.

```
> facV = function(n) prod(seq(n))  
> facV(5)  
[1] 120
```

- Using special functions.

```
> facG = function(n) gamma(n + 1)  
> facG(5)  
[1] 120
```

- The function `facG` is the fastest and most flexible version.

Exercise

Time each of the different factorial functions.

## General Functions

- In general, each S function has the form: `function ( arglist )`  
*body*

where *arglist* is a comma-separated list of formal parameters and *body* is an S expression which computes the value of the function.

- Functions are evaluated by associating the values of the arguments with the names of the formal parameters and then evaluating the body of the function using these associations.

## The Evaluation Process

If the function `hypot` defined by:

```
> hypot = function(a, b) sqrt(a^2 + b^2)
```

then the S expression `hypot(3, 4)` is evaluated as follows.

- Temporarily create variables `a` and `b`, which have the values 3 and 4.
- Use these variables definitions to evaluate the expression `sqrt(a^2 + b^2)` to obtain the value 5.
- When the evaluation is complete remove the temporary definitions of `a` and `b`.

## Optional Arguments

- S has a notion of default argument values.
- These make it possible for arguments to take on reasonable default values if no values was specified in the call to the function.
- In the following, the second argument takes on the value 0 if no value is specified for it.

```
> sumsq = function(x, about = 0) sum((x - about)^2)
```

- This means that the expressions `sumsq(1:10, 0)` and `sumsq(1:10)` are equivalent and will return the same value.

## Optional Arguments

- The default values for arguments can be specified by an S expression involving the variables available inside the body of the function.

```
> sumsq = function(x, about = mean(x)) sum((x - about)^2)
```

- Recursive references within default arguments are not permitted. E. g. at least one argument must be provided to the following function.

```
> silly = function(a = b, b = a) a + b
```



## Argument Matching

- Because it is not necessary to specify all the arguments to S function, it is important to be clear about which argument corresponds to which formal parameter of the function.
- The solution is to indicate which formal parameter is associated with an argument by providing a (partial) name for the argument.
- In the case of the `sumsq` function, the following are equivalent:

```
sumsq(1:10, mean(1:10))
```

```
sumsq(1:10, about = mean(1:10))
```

```
sumsq(1:10, a=mean(1:10))
```

## Argument Matching

The arguments that the function is called with are called the *actuals*. The arguments that appear in the definition of the function are called the *formals*.

- First, arguments are matched by name, using exact matching.
- Second, partial matching of arguments is carried out.
- Third, remaining unmatched arguments are matched by position.

## The dots

- In S there is a special argument, ..., which is three sequential periods.
- This argument matches any number of actual (or supplied) arguments.
- It lets functions have variable numbers of arguments.
- Positional matching only applies to arguments that precede (are left of) the ... argument.
- To match arguments to the right of the dots, you must name the argument and no partial matching is done.

## Lazy Evaluation

- S differs from many computer languages because the evaluation of function arguments is *lazy*.
- In other words, arguments are not actually evaluated until they are required.
- It can be the case that arguments are *never* evaluated.

## Example

- Here is a variation of the `sumsq` function.

```
> sumsq = function(x, about = mean(x)) {  
+   x = x[!is.na(x)]  
+   sum((x - about)^2)  
+ }
```

- This function first removes any NA values from `x` before computing its answer.
- Lazy evaluation means that the `about` value is computed **after** `x` has been cleaned.

## Exercises

1. Modify the `sumsq` so that the removal of NA values is optional.
2. Write a new function which computes the deviations of the values in `x` about `about`. The values returned by the function should be *just like* `x`. How should missing values be handled?

## Reading System Functions

1. The built-in functions supplied with S form a valuable resource for learning about S programming.
2. In many cases you may be surprised by the complexity of what appear to be trivial function (try `median`).
3. Be warned there can still be bugs in system functions.

## Returning Values

- Any single R object can be returned as the value of a function; including a function.
- If you want to return more than one object, you should put them in a list (usually with names), or an S4 object, and return that.
- The value returned by a function is either the value of the last statement executed, or the value of an explicit call to `return`.
- `return` takes a single argument, and can be called from any where in a function.
- `return` is lexically scoped, and can be passed out to other functions, to effect non-local returns.



## Using return

In the functions below, a call to `f` with a value of `x` less than 5 causes an exit from `f`. Notice that the `print` statement is not evaluated in this case.

```
> f = function(x) {  
+   z = g(x, return(3))  
+   print("in f")  
+   z  
+ }  
> g = function(n, y) if (n < 5) y else 4  
> f(3)  
  
[1] 3  
  
> f(10)  
  
[1] "in f"  
[1] 4
```

## Control of Evaluation

- In some cases you want to evaluate a function that may fail, but you do not want to exit from the middle of an evaluation.
- In these cases the function `try` can be used.
- `try(expr)` will either return the value of the expression `expr`, or an object of class *try-error*
- `tryCatch` provides a much more substantial mechanism for condition handling and error recovery.

## Closing Things

- Sometimes you will want to make sure that global parameters (say in `options`) are reset, or files are closed, when you exit a function.
- And often you want to ensure that these are reset even if an error or some other non-local exit occurs.
- In these cases you should use `on.exit`; it allows you to place expressions into a list to be evaluated.

```
> f = function(x) {  
+   on.exit(print("hi"))  
+   5  
+ }  
> f  
  
function (x)  
{  
  on.exit(print("hi"))  
  5  
}
```

## Computing on the Language

- Because argument evaluation is lazy, S allows programmers to get access to the unevaluated arguments.
- This is made possible by the `substitute` function.

```
> g = function(x) substitute(x)
> g(x[1] + y * 2)
x[1] + y * 2
```

- `substitute` is used in conjunction with `deparse` to obtain a character string representation of an argument.

```
> g = function(x) deparse(substitute(x))
> g(x[1] + y * 2)
[1] "x[1] + y * 2"
```

## Computing on the Language

- The substitute function can take a call and substitute the symbolic representation of several arguments.

```
> g <- function(a, b) substitute(a + b)
> g(x * x, y * y)
x * x + y * y
```

- One particularly useful trick is to use the ... argument in a substitute expression.

```
> g = function(...) substitute(list(...))
> g(a = 10, b = 11)
list(a = 10, b = 11)
```

## Scoping

- We've seen that evaluation is the process of determining the value of a symbolic expression.
- In order for evaluation to take place, values must be determined for the variables in the expression.
- The *scope* of a variable is that portion of a program where that variable refers to the same value.
- The two dialects of S differ in their scoping rules.

## Example

- In the following fragment:

```
> x = 10
> y = 20
> f = function(y) {
+   x + y
+ }
```

- There is a global variable called **x**
- There is a global variable called **y** and a local variable called **y**.

## Scoping in S-Plus

- The rules in S-Plus are that variables are either local to the function they are defined in, or they are global.
- The process of determining the value of a variable is as follows.
  - Look for a local variable – if there is one, use its value.
  - If there is no local variable, use the value of the global variable.
- There are some effects of these scoping rules that can be counter-intuitive.



## Scoping Problems

- The following implementation of binomial coefficients does not work in S-Plus.

```
> choose = function(n, k) {  
+   fac = function(n) if (n <= 1)  
+     1  
+   else n * fac(n - 1)  
+   fac(n)/(fac(k) * fac(n - k))  
+ }
```

- Why does this function fail?

## Consequences of S-Plus Scoping Rules

- The scoping rules of S-Plus encourage the use of many globally defined functions, even when those functions are never called directly.
- This is because it is difficult to hide related helper functions inside *wrapper* functions.
- The use of this style of programming produces *name space clutter* and effects like the accidental masking of functions.
- Object-oriented programming extensions help a little, as do name spaces.

## Scoping in R

- R uses what is called static, or lexical scoping (another term is block structure).
- Variables defined in outer blocks are visible inside inner blocks.
- This is a natural extension to the S-Plus way of scoping.
- The hiding of helper functions within wrappers is encourage.
- This promotes better software design and alleviates name space clutter.
- It also has some interesting consequences.

## Example: Gaussian Likelihoods

- This works in R, but not S-Plus, since it relies on lexical scope.

```
> mkNegLogLik = function(x) {  
+   function(mu, sigma) {  
+     sum(sigma + 0.5 * ((x - mu)/sigma)^2)  
+   }  
+ }  
> q = mkNegLogLik(rnorm(100))
```

- What is q? How does it work?

## Packages

- In R one of primary mechanisms for distributing software is via *packages*
- CRAN is the major repository for getting packages.
- You can either download packages manually or use `install.packages` or `update.packages` to install and update packages.
- In addition, on Windows and in some other GUIs, there are menu items that facilitate package downloading and updating.
- It is important that you use the R package installation facilities. You cannot simply unpack the archive in some directory and expect it to work.

## Packages - Bioconductor

- Bioconductor packages are hosted in CRAN-style repositories and are accessible using `install.packages`.
- The most reliable way to install Bioconductor packages (and their dependencies) is to use `biocLite`.
- Bioconductor has both a release branch and a development branch. Each Bioconductor release is compatible with a specific R release.
- Bioconductor packages all have vignettes.

## Packages

- Having, and needing many more packages can cause some problems.
- When packages are loaded into R, they are essentially attached to the `search` list, see `search`.
- This greatly increases the probabilities of variable masking, that is one package provides a function that has the same name as a different function in another package.
- Name spaces were introduced in R 1.7.0 to provide tools that would help alleviate some of the problems.

## Name Spaces

- Name spaces were introduced in R 1.7.0, see R News, Vol 3/1 for more details.
- They provide a mechanism that allows package writers to control what functions they import (and hence use) and export (and hence let others use).
- They allow a package author to use a function from another package without attaching it to the search list.
- However, name spaces cause some problems and interfere with some of the ways users like to interact with R.



## Name Spaces

- Functions that are not exported from a name space are not easily viewed by the user.
- Functions that are exported from a name space can be accessed by using the double-colon, `::`, operator. This is a binary operator, its first operand is the name of the package and the second is the name of the function, e.g. `base::mean`.
- To access a function that is not exported you need to use the triple-colon operator, `:::`.
- This should only be done for the purposes of investigation. If a package author does not export a function, they typically have a reason for that.

## Object Oriented Programming

- Object oriented programming is a style of programming where one attempts to have software reflections of real-world objects and to write functions (methods) that operate on these objects.
- The S language has two different object oriented paradigms, one S3 is older and should not be used for new projects. The second, S4 is newer and is currently under active development.
- These objects systems are more like OOP in Scheme, Lisp or Dylan than they are like OOP in Java or C++.

## Classes

- In OOP there are two basic ingredients, objects and methods.
- An object is an instance of a class, and most OOP implementations have mechanisms to ensure that all objects of a particular class have some common characteristics.
- In most implementations there is some notion of inheritance or class extension. Class B is said to extend class A if a member of B has all the attributes that a member of A does, plus some other attributes.

## Generic Functions

- A *generic function* is an interface, or a dispatcher, that examines the type or class of its arguments and invokes the most appropriate method.
- A method is registered with a generic function, by indicating its existence together with the number and types (classes) of its arguments.
- In the previous example, if a generic function is called with an instance of class B and there is no class B method, a class A method could be used.

## S3

- S3 OOP has no real mechanism for defining classes or for creating objects from a specific class.
- One can make any object an instance of class *foo*, by assigning a class attribute, `class(x) = "foo"`.
- S3 handles inheritance by setting several different class attributes (but these are not always handled correctly).
- S3 is not suitable for the development of large scale complex systems.

## S3 Generic Functions

- The relationship between a generic function and its methods is done by a naming convention. The generic function must have a call to `UseMethod` and the method must have a name that is the name of the generic function concatenated with the name of the class, with the two names separated by a dot.

```
> mean
function (x, ...)
UseMethod("mean")
<environment: namespace:base>

> methods("mean")

[1] mean.Date      mean.POSIXct   mean.POSIXlt   mean.data.frame
[5] mean.default   mean.difftime
```

## S4

- In S4 classes of objects must be explicitly defined using `setClass` and instances of a class must be instantiated using `new`.

```
> setClass("foo", representation(a = "numeric"))
```

```
[1] "foo"
```

```
> b = new("foo", a = 10)
```

```
> b
```

```
An object of class "foo"
```

```
Slot "a":
```

```
[1] 10
```

```
> b@a
```

```
[1] 10
```

## S4

- Generic functions are defined using `setGeneric`.
- Methods are registered with a generic function using `setMethod`. Note there are no required or used naming conventions.
- Methods can be found, and listed using `getMethods`.
- Generic functions can dispatch on any, and all, arguments (unlike Java and C++, which basically dispatch on the first argument).
- Generic functions and classes are not tightly linked, as they are in other programs, and there is no real notion of *this*.



## General Comments

- One of the main reasons for using OOP is to hide implementation details from the user.
- Users essentially only know what operations are valid. They may use them and if the developer changes the implementation, there should be no effect on the end user.
- OOP lets you specialize the behavior of your functions (and system functions) to your objects.