A Quick Introduction to R

Language Overview

Deepayan Sarkar

Indian Satistical Institute, Delhi

January 27, 2010

Background

- R is often referred to as a *dialect* of the S language
- S was developed at the AT&T Bell Laboratories by John Chambers and his colleagues doing research in statistical computing, beginning in the late 1970’s
- The original S implementation is used in the commercially available software *S-PLUS*
- R is an open source implementation developed independently, starting in the early 1990’s
- Mostly similar, but there are differences as well

Expressions and Objects

R works by evaluating *expressions* typed at the command prompt

- Expressions involve variable references, operators, function calls, etc.
- Most expressions, when evaluated, produce a value, which can be either assigned to a variable (e.g. `x <- 2 + 2`), or is printed in the R session
- Some expressions are useful for their *side-effects* (e.g., `plot` produces graphics output)

Since evaluated expression values can be quite large, and often need to be re-used, it is good practice to assign them to variables rather than print them directly

Expressions and Objects

Objects are anything that can be assigned to a variable. In the following example, `c(1, 2, 3, 4, 5)` is an **expression** that produces an **object**, whether or not the result is stored in a variable:

```
> sum(c(1, 2, 3, 4, 5))
```

```
[1] 15
```

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

```
[1] 15
```

R has several important *types* of objects that we will learn about; for example: *functions, vectors* (numeric, character, logical), *matrices, lists* and *data frames*

Functions

Most useful things in R are done by function calls. Function calls look like a name followed by some *arguments* in parentheses.

```
> plot(height, weight)
```

Apart from a special argument called `...`, all arguments have a *formal name*. When a function is evaluated, it needs to know what value has been assigned to each of its arguments.

Functions

There are several ways to specify arguments:

- *By position*:
  The first two arguments of the `plot` function are `x` and `y`. So,

  ```
  > plot(height, weight)
  ```

  is equivalent to

  ```
  > plot(x = height, y = weight)
  ```

- *By name*:
  This is the safest way to match arguments, by specifying the argument names explicitly. This overrides positional matching, so it is equivalent to say

  ```
  > plot(y = weight, x = height)
  ```

  Formal argument names can be matched partially
  (we will see examples later).

Functions

There are several ways to specify arguments:

- *With default values*:
  Arguments will often have default values. If they are not specified in the call, these default values will be used.
  ```
  > plot(height)
  ```

Functions are just like other objects in R:

- They can be assigned to variables
- They can be used as arguments in other function calls
- New function objects are defined using the construct
  ```
  function( arglist ) expr
  ```

Functions

A simple function:

```
> myfun <- function(a = 1, b = 2, c)
+ {
+     return(list(a = a, b = b, c = c))
+ }
```

```
> myfun(6, 7, 8)
```

```
$a
[1] 6

$b
[1] 7

$c
[1] 8
```

Functions

A simple function:

```
> myfun <- function(a = 1, b = 2, c)
+ {
+     return(list(a = a, b = b, c = c))
+ }
```

```
> myfun(10, c = 'string')
```

```
$a
[1] 10

$b
[1] 2

$c
[1] "string"
```

Function Arguments

The arguments that a particular function accepts (along with their default values) can be listed by the `args` function:

```
> args(myfun)
```

```
function (a = 1, b = 2, c)
NULL
```

```
> args(plot.default)
```

```
function (x, y = NULL, type = "p", xlim = NULL, ylim = N
    log = "", main = NULL, sub = NULL, xlab = NULL, ylab
    ann = par("ann"), axes = TRUE, frame.plot = axes, pa
    panel.last = NULL, asp = NA, ...)
NULL
```

The triple-dot (...) argument indicates that the function can accept any number of further named arguments. What happens to those arguments is determined by the function.

Vectors

The basic data types in R are all vectors. The simplest varieties are *numeric*, *character* and *logical* (`TRUE` or `FALSE`):

```
> c(1, 2, 3, 4, 5)
[1] 1 2 3 4 5
> c("Huey", "Dewey", "Louie")
[1] "Huey"  "Dewey" "Louie"
> c(T, T, F, T)
[1]  TRUE  TRUE FALSE  TRUE
> c(1, 2, 3, 4, 5) > 3
[1] FALSE FALSE FALSE  TRUE  TRUE
```

`T` and `F` are convenient abbreviations for `TRUE` and `FALSE` respectively.

Vectors

The length of any vector can be determined by the `length()` function:

```
> gt.3 <- c(1, 2, 3, 4, 5) > 3
> gt.3

[1] FALSE FALSE FALSE  TRUE  TRUE

> length(gt.3)

[1] 5

> sum(gt.3)

[1] 2
```

This happens because of *coercion* from logical to numeric.

Special values

- `NA` Denotes a ‘missing value’
- `NaN` ‘Not a Number’, e.g., $0/0$
- `-Inf, Inf` positive and negative infinities, e.g. $1/0$
- `NULL` Null object, mostly for programming convenience

Functions that create vectors

`seq()` creates a sequence of equidistant numbers (See `?seq`)

```
> seq(4, 10, 0.5)
```

```
 [1]  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0
[10]  8.5  9.0  9.5 10.0
```

```
> seq(length = 10)
```

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

```
> 1:10
```

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

```
> args(seq.default)
```

```
function (from = 1, to = 1, by = ((to - from)/(length.ou
    length.out = NULL, along.with = NULL, ...)
NULL
```

Partial matching: Note that the named argument `length` of the call to `seq()` actually matches the argument `length.out`

Functions that create vectors

`c()` *concatenates* one or more vectors:

```
> c(1:5, seq(10, 20, length = 6))
 [1]  1  2  3  4  5 10 12 14 16 18 20
```

`rep()` replicates a vector

```
> rep(1:5, 2)
 [1] 1 2 3 4 5 1 2 3 4 5
> rep(1:5, length = 12)
 [1] 1 2 3 4 5 1 2 3 4 5 1 2
> rep(c('one', 'two'), c(6, 3))
[1] "one" "one" "one" "one" "one" "one" "two"
[8] "two" "two"
```

Matrices and Arrays

Matrices (and more generally arrays of any dimension) are stored in R as a vector with dimensions:

```
> x <- 1:12
> dim(x) <- c(3, 4)
> x

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> nrow(x)

[1] 3

> ncol(x)

[1] 4
```

The fact that the left hand side of an assignment can look like a function applied to an object (rather than a variable) is a very interesting and useful feature. These are called *replacement* functions.

Matrices and Arrays

The same vector can be used to create a 3-dimensional array

```
> dim(x) <- c(2, 2, 3); x
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

, , 3

     [,1] [,2]
[1,]    9   11
[2,]   10   12
```

Matrices (contd)

Matrices can also be created conveniently by the `matrix` function. Their row and column names can be set.

```
> x <- matrix(1:12, nrow = 3, byrow = TRUE)
> rownames(x) <- LETTERS[1:3]
> x

  [,1] [,2] [,3] [,4]
A    1    2    3    4
B    5    6    7    8
C    9   10   11   12

> t(x)

     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
```

Matrices can be transposed by the `t()` function. General array permutation is done by `aperm()`.

Matrices (contd)

Matrices do not need to be numeric. There can be character or logical matrices as well:

```
> matrix(month.name, nrow = 6)

     [,1]       [,2]
[1,] "January"  "July"
[2,] "February" "August"
[3,] "March"    "September"
[4,] "April"    "October"
[5,] "May"      "November"
[6,] "June"     "December"
```

Matrix multiplication

The multiplication operator (`*`) works element-wise, as with vectors. The matrix multiplication operator is `%*%`:

```
> x

  [,1] [,2] [,3] [,4]
A    1    2    3    4
B    5    6    7    8
C    9   10   11   12

> x * x

  [,1] [,2] [,3] [,4]
A    1    4    9   16
B   25   36   49   64
C   81  100  121  144

> x %*% t(x)

    A   B   C
A  30  70 110
B  70 174 278
C 110 278 446
```

Creating matrices from vectors

The `cbind` (*column bind*) and `rbind` (*row bind*) functions can create matrices from smaller matrices or vectors:

```
> y <- cbind(A = 1:4, B = 5:8, C = 9:12)
> y
```

```
     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
```

```
> rbind(y, 0)
```

```
     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
[5,] 0 0  0
```

Note that the short vector (0) is replicated.

Factors

Factors are how R handles *categorical data* (e.g., eye color, gender, pain level). Such data are often available as numeric codes, but should be converted to factors for proper analysis.

```
> pain <- c(0, 3, 2, 2, 1)
> fpain <- factor(pain, levels = 0:3)
> fpain

[1] 0 3 2 2 1
Levels: 0 1 2 3

> levels(fpain) <- c("none", "mild", "medium", "severe")
> fpain

[1] none   severe medium medium mild
Levels: none mild medium severe

> as.numeric(fpain)

[1] 1 4 3 3 2
```

The last function extracts the internal representation of factors, as integer codes starting from 1.

Factors

Factors can also be created from character vectors.

```
> text.pain <-
+     c("none", "severe", "medium", "medium", "mild")
> factor(text.pain)
```

```
[1] none   severe medium medium mild
Levels: medium mild none severe
```

Note that the levels are sorted alphabetically by default, which may not be what you really want. It is usually a good idea to specify the levels explicitly when creating a factor.

Lists

- *Lists* are very flexible data structures used extensively in R.
- A list is a vector, but the elements of a list do not need to be of the same type. Each element of a list can be *any* R object, including another list.
- lists can be created using the `list` function
- list elements are usually extracted by name (using the `$` operator).

Lists

```
> x <- list(fun = seq, len = 10)
> x$fun

function (...)
UseMethod("seq")
<environment: namespace:base>

> x$len

[1] 10

> x$fun(length = x$len)

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

- Functions are R objects too. In this case, the `fun` element of `x` is the already familiar `seq` function, and can be called like any other function.
- Lists give us the ability to create *composite objects* that contain several related, simpler objects. Many useful R functions return a list rather than a simple vector.

Lists (contd)

A more natural example: energy intake (paired: before and after) (*§1.2.8, Dalgaard (2002)*):

```
> intake.pre <- c(5260, 5470, 5640, 6180, 6390,
+                 6515, 6805, 7515, 7515, 8230, 8770)
> intake.post <- c(3910, 4220, 3885, 5160, 5645,
+                  4680, 5265, 5975, 6790, 6900, 7335)
> mylist <- list(before = intake.pre,
+                after = intake.post)
> mylist
$before
 [1] 5260 5470 5640 6180 6390 6515 6805 7515 7515
[10] 8230 8770

$after
 [1] 3910 4220 3885 5160 5645 4680 5265 5975 6790
[10] 6900 7335
> mylist$before
 [1] 5260 5470 5640 6180 6390 6515 6805 7515 7515
[10] 8230 8770
```

Data Frames

Data frames are R objects that represent (rectangular) data sets, and thus very important for statistical applications. They are essentially lists with some additional structure.

- Each element of a data frame has to be a either a factor or a numeric, character or logical vector
- Each of these must have the same length
- They are similar to matrices because they have the same *rectangular array* structure; the only difference is that different columns of a data frame can be of a different data type.

Data Frames

Data frames are created by the `data.frame` function:

```
> d <- data.frame(intake.pre, intake.post)
> d
   intake.pre intake.post
1        5260        3910
2        5470        4220
3        5640        3885
4        6180        5160
5        6390        5645
6        6515        4680
7        6805        5265
8        7515        5975
9        7515        6790
10       8230        6900
11       8770        7335
> d$intake.post
 [1] 3910 4220 3885 5160 5645 4680 5265 5975 6790
[10] 6900 7335
```

The list-like `$` operator can be used to extract columns.

Indexing

Extracting one or more elements from a vector is done by *indexing*. There are several kinds of indexing possible in R, among them

- Indexing by a vector of positive integers
- Indexing by a vector of negative integers
- Indexing by a logical vector
- Indexing by a vector of names

In each case, the extraction is done by following the vector by a pair of brackets (`[...]`). The type of indexing depends on the object inside the brackets.

Indexing by positive integers

```
> intake.pre

 [1] 5260 5470 5640 6180 6390 6515 6805 7515 7515
[10] 8230 8770

> intake.pre[5]

[1] 6390

> intake.pre[c(3,5,7)]

[1] 5640 6390 6805

> ind <- c(3,5,7)
> intake.pre[ind]

[1] 5640 6390 6805

> intake.pre[8:13]

[1] 7515 7515 8230 8770   NA   NA

> intake.pre[c(1, 2, 1, 2)]

[1] 5260 5470 5260 5470
```

Indexing by positive integers

Works more or less as expected. Interesting features:

- using an index bigger than the length of the vector produces `NA`’s
- indices can be repeated, resulting in the same element being chosen more than once. This feature is often very useful.

Indexing by negative integers

Using negative indices leaves out the specified elements.

```
> intake.pre
```

```
 [1] 5260 5470 5640 6180 6390 6515 6805 7515 7515
[10] 8230 8770
```

```
> intake.pre[-5]
```

```
 [1] 5260 5470 5640 6180 6515 6805 7515 7515 8230
[10] 8770
```

```
> ind <- -c(3,5,7)
> ind
```

```
[1] -3 -5 -7
```

```
> intake.pre[ind]
```

```
[1] 5260 5470 6180 6515 7515 7515 8230 8770
```

Negative indices cannot be mixed with positive indices.

Indexing by a logical vector

For this, the logical vector being used as the index should be exactly as long as the vector being indexed. If it is shorter, it is replicated to be as long as necessary.

```
> intake.pre

 [1] 5260 5470 5640 6180 6390 6515 6805 7515 7515
[10] 8230 8770

> ind <- rep(c(TRUE, FALSE), length = length(intake.pre)
> ind

 [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
 [8] FALSE  TRUE FALSE  TRUE

> intake.pre[ind]

[1] 5260 5640 6390 6805 7515 8770

> intake.pre[c(T, F)]

[1] 5260 5640 6390 6805 7515 8770
```

Only the elements that correspond to `TRUE` are retained.

Indexing by names

This works only for vectors that have names.

```
> names(intake.pre) <- LETTERS[1:11]
> intake.pre
```

```
   A    B    C    D    E    F    G    H    I    J
5260 5470 5640 6180 6390 6515 6805 7515 7515 8230
   K
8770
```

```
> intake.pre[c('A', 'B', 'C', 'K')]
```

```
   A    B    C    K
5260 5470 5640 8770
```

```
> names(intake.pre) <- NULL
```

All these types of indexing works for matrices and arrays as well, as we shall see later.

Logical comparisons

All the usual logical comparisons are possible:

less than	<	less than or equal to	<=
greater than	>	greater than or equal to	>=
equals	==	does not equal	!=

Each of these operate on two vectors element-wise
(the shorter one is replicated if necessary).

```
> intake.pre
 [1] 5260 5470 5640 6180 6390 6515 6805 7515 7515
[10] 8230 8770
> intake.pre > 7000
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [8]  TRUE  TRUE  TRUE  TRUE
> intake.pre > intake.post
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[10] TRUE TRUE
```

Logical operations

Element-wise boolean operations are also possible.

AND	`&`
OR	`|`
NOT	`!`

```
> intake.pre
 [1] 5260 5470 5640 6180 6390 6515 6805 7515 7515
[10] 8230 8770
> intake.pre > 7000
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [8]  TRUE  TRUE  TRUE  TRUE
> intake.pre < 8000
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [8]  TRUE  TRUE FALSE FALSE
> intake.pre > 7000 & intake.pre < 8000
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [8]  TRUE  TRUE FALSE FALSE
```

Conditional Selection

Logical comparisons and indexing by logical vectors together allow subsetting a vector based on the properties of other (or perhaps the same) vectors.

```
> intake.post
```

```
 [1] 3910 4220 3885 5160 5645 4680 5265 5975 6790
[10] 6900 7335
```

```
> intake.post[intake.pre > 7000]
```

```
[1] 5975 6790 6900 7335
```

```
> intake.post[intake.pre > 7000 & intake.pre < 8000]
```

```
[1] 5975 6790
```

```
> month.name[month.name > "N"]
```

```
[1] "September" "October"   "November"
```

For character vectors, sorting is determined by alphabetical order.

Matrix and Data frame indexing

Indexing for matrices and data frames are similar: they also use brackets, but need two indices. If one (or both) of the indices are unspecified, all the corresponding rows and columns are selected.

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

Matrix and Data frame indexing

If only one row or column is selected, the result is converted to a vector. This can be suppressed by adding a `drop = FALSE`

```
> x[1,]

[1]  1  4  7 10

> x[1,,drop = FALSE]

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
```

Matrix and Data frame indexing

Data frames behave similarly.

```
> d[1:3,]
  intake.pre intake.post
1       5260        3910
2       5470        4220
3       5640        3885
> d[1:3, "intake.pre"]
[1] 5260 5470 5640
> d[d$intake.post < 5000, 1, drop = FALSE]
  intake.pre
1       5260
2       5470
3       5640
6       6515
```

Modifying objects

It is usually possible to modify R objects by assigning a value to a subset or function of that object. For the most part, anything that makes sense, works. This will become clearer with more experience.

```
> x <- runif(10, min = -1, max = 1)
> x
 [1] -0.05443569  0.87427582  0.71653135
 [4]  0.93435052 -0.18046813 -0.99931394
 [7]  0.69577254  0.16808155  0.35623236
[10] -0.62989364
> x < 0
 [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE
 [8] FALSE FALSE  TRUE
> x[x < 0] <- 0
> x
 [1] 0.0000000 0.8742758 0.7165313 0.9343505
 [5] 0.0000000 0.0000000 0.6957725 0.1680815
 [9] 0.3562324 0.0000000
```

Adding columns to a data frame

New columns can be added to data frame, by assigning to a currently non-existent column name (this works for lists too):

```
> d$decrease
NULL
> d$decrease <- d$intake.pre - d$intake.post
> d
   intake.pre intake.post decrease
1        5260        3910     1350
2        5470        4220     1250
3        5640        3885     1755
4        6180        5160     1020
5        6390        5645      745
6        6515        4680     1835
7        6805        5265     1540
8        7515        5975     1540
9        7515        6790      725
10       8230        6900     1330
11       8770        7335     1435
```

The `subset()` function

Working with data frames can become a bit cumbersome because we always need to prefix the name of the data frame to every column.

There are several functions to make this easier.

For example, `subset()` can be used to select rows of a data frame.

The `subset()` function

```
> library(ISwR)
> data(thuesen)
> str(thuesen)

'data.frame': 24 obs. of  2 variables:
 $ blood.glucose : num  15.3 10.8 8.1 19.5 7.2 5.3 9.3 1
 $ short.velocity: num  1.76 1.34 1.27 1.47 1.27 1.49 1.

> thue2 <- subset(thuesen, blood.glucose < 7)
> thue2

   blood.glucose short.velocity
6            5.3           1.49
11           6.7           1.25
12           5.2           1.19
15           6.7           1.52
17           4.2           1.12
22           4.9           1.03
```

The `transform()` function

Similarly, the `transform` function can be used to add new variables to a data frame using the old ones

```
> thue3 <- transform(thue2, log.gluc = log(blood.glucose
> thue3
```

```
   blood.glucose short.velocity log.gluc
6            5.3           1.49 1.667707
11           6.7           1.25 1.902108
12           5.2           1.19 1.648659
15           6.7           1.52 1.902108
17           4.2           1.12 1.435085
22           4.9           1.03 1.589235
```

The `with()` function

Another similar and very useful function is `with`, which can be used to evaluate arbitrary expressions using variables in a data frame:

```
> with(thuesen, log(blood.glucose))
```

```
 [1] 2.727853 2.379546 2.091864 2.970414 1.974081
 [6] 1.667707 2.230014 2.406945 2.014903 2.501436
[11] 1.902108 1.648659 2.944439 2.714695 1.902108
[16] 2.151762 1.435085 2.332144 2.525729 2.778819
[21] 2.587764 1.589235 2.174752 2.251292
```

Grouped data

Grouped data have one or more numerical variables, and one or more categorical factors (a.k.a groups) that indicate the category for each observation. The most natural way to store such data is as data frames with different columns for the numerical and categorical variables.

```
> data(energy)
> str(energy)
'data.frame': 22 obs. of  2 variables:
 $ expend : num  9.21 7.53 7.48 8.08 8.09 ...
 $ stature: Factor w/ 2 levels "lean","obese": 2 1 1 1 1
> summary(energy)
     expend          stature
 Min.   : 6.130   lean :13
 1st Qu.: 7.660   obese: 9
 Median : 8.595
 Mean   : 8.979
 3rd Qu.: 9.900
 Max.   :12.790
```

Extracting information by group

It is easy to extract data by category:

```
> exp.lean <- energy$expend[energy$stature == "lean"]
> exp.obese <- with(energy, expend[stature == "obese"])
> exp.lean
```

```
 [1]  7.53  7.48  8.08  8.09 10.15  8.40 10.88
 [8]  6.13  7.90  7.05  7.48  7.58  8.11
```

```
> exp.obese
```

```
[1]  9.21 11.51 12.79 11.85  9.97  8.79  9.69
[8]  9.68  9.19
```

Extracting information by group

A more sophisticated way to do this:

```
> l <- with(energy, split(x = expend, f = stature))
> l
```

```
$lean
 [1]  7.53  7.48  8.08  8.09 10.15  8.40 10.88
 [8]  6.13  7.90  7.05  7.48  7.58  8.11

$obese
[1]  9.21 11.51 12.79 11.85  9.97  8.79  9.69
[8]  9.68  9.19
```

Extracting information by group

More generally, arbitrary functions can be applied to data frames split by a group using the `by()` function:

```
> by(data = energy, INDICES = energy$stature, FUN = summ
energy$stature: lean
     expend          stature
 Min.   : 6.130   lean  :13
 1st Qu.: 7.480   obese: 0
 Median : 7.900
 Mean   : 8.066
 3rd Qu.: 8.110
 Max.   :10.880
-------------------------------------
energy$stature: obese
     expend         stature
 Min.   : 8.79   lean :0
 1st Qu.: 9.21   obese:9
 Median : 9.69
 Mean   :10.30
 3rd Qu.:11.51
```

Sorting

Vectors can be sorted by `sort()`:

```
> sort(intake.post)
 [1] 3885 3910 4220 4680 5160 5265 5645 5975 6790
[10] 6900 7335
```

But it's usually more useful to work with the *sort order*, using the `order()` function, which returns an integer indexing vector that can be used get the sorted vectors. This can be useful to re-order the rows of a data frame by one or more columns.

Sorting

```
> ord <- order(intake.post)
> ord
 [1]  3  1  2  6  4  7  5  8  9 10 11
> intake.post[ord]
 [1] 3885 3910 4220 4680 5160 5265 5645 5975 6790
[10] 6900 7335
> intake.pre[ord]
 [1] 5640 5260 5470 6515 6180 6805 6390 7515 7515
[10] 8230 8770
> d[ord, ]
   intake.pre intake.post decrease
3        5640        3885     1755
1        5260        3910     1350
2        5470        4220     1250
6        6515        4680     1835
4        6180        5160     1020
7        6805        5265     1540
5        6390        5645      745
```

Implicit loops

We often need to apply one particular function to all elements in a vector or a list. Generally, this would be done by looping through all those elements. R has a few functions to do this elegantly;

- `lapply()`: Returns the results as a list
- `sapply()`: Tries to simplify the results and make it a vector
- See also `apply()` and `tapply()`

Implicit loops

```
> lapply(thuesen, mean)

$blood.glucose
[1] 10.3

$short.velocity
[1] NA

> sapply(thuesen, mean)

 blood.glucose short.velocity
          10.3             NA

> sapply(thuesen, mean, na.rm = TRUE)

 blood.glucose short.velocity
     10.300000       1.325652
```

Note the need for `na.rm = TRUE` because there is a missing observation in one of the rows. Unless otherwise specified, all calculations involving an `NA` usually produce an `NA`.

Graphics

R’s graphics capabilities are one of its strongest features. It can also be fairly complicated, with many features that are rarely used. Instead of going into details here, we will learn about R graphics by looking at some examples later. Meanwhile,

- Look at `help(plot.default)`
- Look at `help(par)`

These two help pages cover most of the options and features common to standard graphics functions. They contain a lot of information, and are mostly useful as references to look up when you need to do something special.

Programming constructs

R has the standard programming constructs:

- `if`
- `else`
- `for`
- `while`
- etc.

for loops

- Since most R functions work on vectors, the `for` construct is rarely needed for simple use.
- The `for` keyword is always followed by an expression of the form `(variable in vector)`.
- The block of statements that follow this is executed once for every value in `vector`, with that value being stored in `variable`

```
> for (i in 1:5) {
+     print(i^2)
+ }

[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

while and if statements

```
> fibonacci <- function(length.out) {
+     if (length.out < 0) {
+         warning("length.out cannot be negative")
+         return(NULL)
+     }
+     else if (length.out < 2)
+         x <- seq(length = length.out) - 1
+     else {
+         x <- c(0, 1)
+         while (length(x) < length.out) {
+             x <- c(x, sum(rev(x)[1:2]))
+         }
+     }
+     x
+ }
```

while and if statements

```
> fibonacci(-1)
NULL
> fibonacci(1)
[1] 0
> fibonacci(10)
 [1]  0  1  1  2  3  5  8 13 21 34
```

Note that a function returns the last expression it evaluates (in this case `x`), and the explicit `return()` is not necessary.

Session management

R has the ability to save objects, to be loaded again later. Whenever exiting, R tries to save all the objects currently in the workspace, and when starting up the next time (in the same directory), it loads it up again.

See `?save`

Classes and generic functions

R implements a system of *object-oriented* programming, based on the following concepts:

- *Generic functions*: functions meant to do a particular task, but do it differently based on the object it operates on. Examples: `plot()`, `summary()`, `mean()`
- *Methods*: specific versions of the generic function
- *Class*: attribute of an object that determines which generic will be used, e.g.,

```
> class(thuesen)
[1] "data.frame"
> class(thuesen$blood.glucose)
[1] "numeric"
> class(seq)
[1] "function"
```

Methods

Methods for a particular generic can be listed using the `methods()` function.

There is usually a *“default”* method, conventionally named `plot.default()`, `summary.default()`, etc.

```
> methods(mean)

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

Inspecting R objects using `str()`

The `str()` function prints out information about the `structure` of any R object.

```
> str(thuesen)
```

```
'data.frame': 24 obs. of  2 variables:
 $ blood.glucose : num  15.3 10.8 8.1 19.5 7.2 5.3 9.3 1
 $ short.velocity: num  1.76 1.34 1.27 1.47 1.27 1.49 1.
```

This can be especially useful for large data frames.

Type checking

It is often useful to know whether an object is of certain type. There are several functions of the form `is.type` which do this.

Note that `is` is not a generic function, even though the naming convention is similar.

```
> is.data.frame(thuesen)
[1] TRUE
> is.list(thuesen)
[1] TRUE
> is.numeric(thuesen)
[1] FALSE
> is.function(thuesen)
[1] FALSE
```

Detecting special values

Some similarly-named functions are used for element-wise checking. The most important of these is `is.na()`, which is needed to identify which elements of a vector are missing.

```
> thuesen$short.velocity
 [1] 1.76 1.34 1.27 1.47 1.27 1.49 1.31 1.09 1.18
[10] 1.22 1.25 1.19 1.95 1.28 1.52   NA 1.12 1.37
[19] 1.19 1.05 1.32 1.03 1.12 1.70
> thuesen$short.velocity == NA
 [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[16] NA NA NA NA NA NA NA NA NA
> is.na(thuesen$short.velocity)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [8] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[15] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
[22] FALSE FALSE FALSE
```

Detecting special values

```
> is.na(c(Inf, NaN, NA, 1))
[1] FALSE  TRUE  TRUE FALSE
> is.nan(c(Inf, NaN, NA, 1))
[1] FALSE  TRUE FALSE FALSE
> is.finite(c(Inf, NaN, NA, 1))
[1] FALSE FALSE FALSE  TRUE
```

Coercion methods

There are several functions of the form `as.type` that are used to convert objects of one type to another.

```
> as.numeric(c("1", "2", "2a", "b"))

[1]  1  2 NA NA

> as.numeric(c(TRUE, FALSE, NA))

[1]  1  0 NA

> as.character(c(TRUE, FALSE, NA))

[1] "TRUE"  "FALSE" NA
```

Coercion methods (contd)

There are some automatic coercion rules that often simplify things

```
> thuesen$blood.glucose < 7
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
 [8] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
[15]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE
[22]  TRUE FALSE FALSE
> sum(thuesen$blood.glucose < 7)
[1] 6
```

but can sometimes produce surprising results

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

Coercion methods (contd)

```
> t(as.matrix(thuesen)) ## data.frame -> matrix

                [,1]  [,2] [,3]  [,4] [,5] [,6]
blood.glucose  15.30 10.80 8.10 19.50 7.20 5.30
short.velocity  1.76  1.34 1.27  1.47 1.27 1.49
               [,7]  [,8] [,9] [,10] [,11] [,12]
blood.glucose  9.30 11.10 7.50 12.20  6.70  5.20
short.velocity 1.31  1.09 1.18  1.22  1.25  1.19
               [,13] [,14] [,15] [,16] [,17]
blood.glucose  19.00 15.10  6.70   8.6  4.20
short.velocity  1.95  1.28  1.52    NA  1.12
               [,18] [,19] [,20] [,21] [,22]
blood.glucose  10.30 12.50 16.10 13.30  4.90
short.velocity  1.37  1.19  1.05  1.32  1.03
               [,23] [,24]
blood.glucose   8.80   9.5
short.velocity  1.12   1.7
```

Coercion methods (contd)

```
> as.list(thuesen) ## data.frame -> list
$blood.glucose
 [1] 15.3 10.8  8.1 19.5  7.2  5.3  9.3 11.1  7.5
[10] 12.2  6.7  5.2 19.0 15.1  6.7  8.6  4.2 10.3
[19] 12.5 16.1 13.3  4.9  8.8  9.5

$short.velocity
 [1] 1.76 1.34 1.27 1.47 1.27 1.49 1.31 1.09 1.18
[10] 1.22 1.25 1.19 1.95 1.28 1.52   NA 1.12 1.37
[19] 1.19 1.05 1.32 1.03 1.12 1.70
```

Further resources

At this point, you should know enough about R to do more exploration on your own. For now, you can use the datasets that come with R (you can get a list using `data`), we'll learn how to import data soon.

The course page has links to more resources.