

Object-Oriented Themes

Abstraction. Focus on essential aspects.

Encapsulation. Separate external aspects from internal ones.

Combining of data and behavior. Emphasis on object structure, not procedure structure.

Sharing.

Object Programming

Object programming is based on the concepts of *methods* and *data types*. The basic idea is that methods (procedures, or groups of instructions) apply in specific ways to certain data types.

Some people have characterized object-oriented programming as dealing primarily with *things* (nouns), and “traditional” programming as dealing primarily with *methods* or functions (verbs).

A task that is identified in a single way may perform differently for different types of data.

General Concepts in Object Programming

Method and data type – “procedure” and “kind of data”

Class – a collection of methods and the data types they operate on.

Specifies how its members behave: how they are created; how they are manipulated and how they are destroyed.

Object – an instance of a class.

The Object Orientation

Encapsulation. The information about how methods apply to specific data types. This information is the essential part of the definition of a *class*. The programming language must make provision for supplying this information.

Inheritance. Building new *derived* classes from existing *base* classes. The derived classes inherit the properties of the base classes, but may include additional action and data. There results in a hierarchy of classes.

Polymorphism. Different meanings of functions depending on the operand objects. The operation is *overloaded*.

Example: Print a —- scalar, matrix, table, etc.

R

R is a graphically-oriented data analysis system and object-oriented programming language.

Available on Linux/Unix and MS Windows systems. Documentation exists in several volumes, and in an on-line help system.

R is based on S developed at Bell Labs in the 1970s by John Chambers. R (and S, and S-Plus, another packaged based on S) is an interactive, interpretive, function language.

Free; get at

<http://www.r-project.org/>

Packages in R

R is designed around a base set of functions and classes. The functions in the base kernel include most of the common mathematical functions and operations.

R Syntax

Most statements are of the form

variable <- *function(...)*

or

function(...)

*** Never use “_” for “<-”.

R is case sensitive.

R Fundamentals

Some standard R objects:

variables,

vectors,

lists,

matrices,

arrays,

formulas,

factors (for statistical applications),

data frames (for statistical applications),

fits (for statistical applications)

R has an extensive set of functions, i.e. verbs. The specific meaning depends on the class of the object to which it is applied.

Objects are built by constructor functions; for example `matrix()`:

```
> A <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow=3)
```

This object is a matrix and can participate in matrix multiplication and so on.

Entering A results in

```
A =  
  1 4 7  
  2 5 8  
  3 6 9
```

R Syntax, Functions

All actions are “functions”. A function name is followed by a set of parentheses to enclose arguments.

`help()` or `help(plot)`, e.g. or `?plot`

`q()`

The syntax is somewhat similar to C++, except that there are no special delimiters for statements; statements continue as necessary to complete the function.

Statements may be grouped by { and }.

Blanks are significant only when it makes sense for them to be.

The language is not strongly typed.

Numbers are generally double precision floating point numbers.

Saving and Using Files in R

Each time you reference an external file in an R session of course you can use the full path name, which is tiresome and requires multiple changes to a saved R script.

R provides help in keeping your work organized into directories.

One way is to set a working directory and then all external files come out of or go into that directory until you change it again.

```
setwd('c:/transfers/csi991/06sum')
```

Examples

```
> x <- c(2, 5, 3)
> x
> z1 <- 3 + 4*i
> z2 <- complex(real=3, imaginary=4)
> z <- scan(file="example.dat")
> y <- scan()
1: 21. 23. 26. 27. 27. 26.
7: 23. 23. 28. 28. 27. 26.
13: 24. 22. 27. 26. 28. 28.
19: 25. 23. 29. 28. 28. 26.
25:
> a <- 0:10
> b <- 0:10/10
> c <- seq(from=0, t=1, by=.1)
> plot(y,type="1")
```

General Design

R deals with *functions*.

This dictates the syntax – no statement delimiters (but does use { and }).

Comments begin with #.

No fixed naming conventions; the wise user, however, adopts mnemonic conventions. Use periods to represent components.

Various ways of extracting components. \$ component operator:
object\$member

See objects with objects().

R Matrices

```
A <- matrix(c(1, 3, 5, 2, 4, 6), nrow=3)
x <- c(3, 4)
z <- A[2, ]
w <- A[2, 1:2]
v <- c(1, 2, 1)
b <- A%*%x;
cc <- v%*%A
D <- cbind(A, v)
E <- solve(D)
f <- solve(D, v)
H <- D%*%A
L <- D*D
M <- D^2
objects()
rm(...)
```

Other Operators and Functions

< , <=, >, >=, ==, !=

&, |, !

sin, cos, ...

abs, Arg, sqrt, Re, Im, Conj,
round, trunc, floor, ceiling, sign,
%%,
exp, log, log10

t, crossprod, solve,

sink("filename")

sink()

Plus lots of functions for statistical analyses.

Programming

Conditionals

```
if(x >= 3)
{
  y <- 2
  z <- 4
} else if (x <= 1)
  y <- 1
else
  y <- 5
```

Functions

It's very easy to write functions in R. The arguments can be positional or keyword, and they can be optional with or without default values.

The scope of local variables is limited to the statements within the function.

Example. A Function in R

```
# function to compute skewness and kurtosis coefficients
# data are in first n positions of the vector x
shape <- function(x, n=length(x)){
  skewness <- 0
  kurtosis <- 0
  if (n >=2) {
    m <- mean(x[1:n])
    s <- sd(x[1:n])
    m3 <- sum((x[1:n]-m)^3)
    m4 <- sum((x[1:n]-m)^4)
    if (s > 0) {
      skewness <- m3/s^3
      kurtosis <- m4/s^4
    }
  }
  c("skewness"=skewness, "kurtosis"=kurtosis)
}
```

Programming

Loops

```
for (i in seq(1,10,by=2)) {  
  for (j in 2:5) A[i,j] <- 0  
  x[i] <- 3  
}
```

```
i <- 1  
while (i < 5) {  
  A[i] <- i+1  
  i <- i+1  
}
```

Programming

Loops are inefficient in R. Try to implement as statements involving vectors.

```
aa <- c(1:10)
bb <- c(rep(0,10))
```

Use `ifelse`.

Use `apply`.

R Functions for Graphics

plot
plot.factor
pairs
brush
hist
stem
barplot
persp
faces
stars
matplot

And others. The actual appearance of the graph depends on the class of the object.

Arguments for functions may be required or optional. Most required ones are positional, many optional ones are keyword.

R Functions for Standard Distributions

Functions (first letter)

- d – density
- p – cumulative probability
- q – quantile
- r – random number generation

R Functions for Standard Distributions

Distributions

- beta
- f
- gamma
- norm
- t
- unif
- etc.

Examples: `rnorm(25, 100, 8)` generates 25 $N(100,64)$ numbers
`qf(.95, 5, 10)` the .95 quantile of an F with 5 and 10 degrees of freedom.

The Object Orientation of R

Most of the functions of R are overloaded and are data-driven.

A new class can be defined by defining a constructor function that gives an identifier to the class:

```
factor <- function(x, levels = sort(unique(x)),
  labels = as.character(levels)) {
  y <- match(x, levels)
  names(y) <- names(x)
  levels(y) <- labels
  class(y) <- "factor"
  y
}
```

The function `factor` will construct an object of class “factor”.

The function `mode` can be used to coerce an object to a particular type:

```
mode(x) <- "factor"
```

The function `inherits` can be used to determine if an object is of a particular class. Functions specific to the class can be written.

Interfacing R with C or C++

Reasons to do so:

- Availability of code
- Speed of compiled code

Reasons not to:

- Can pass only the simplest of arguments (objects lose their characteristics – must reconstruct more complicated objects)
- Increased complexity
- Decreased portability

Interfacing R with C or C++

Strategy:

- Do most error checking and data management in R
- Do computationally intensive tasks in C
- I/O ... depends ...

These are the general considerations for linking systems.

How to Do It

1. Write suitable C code
2. Use
R COMPILE
to compile the code. (This is somewhat platform-dependent.)
3. Load the code into R.
(This is very platform-dependent.)
4. Call the compiled code from within R

Suitable C code:

- `#include <S.h>`
- All arguments are pointers
- Function is void

Call with `.C("name", arg_list)`

Example

In the file `my_exp.c`:

```
#include <S.lib>
#include <math.lib>
void my_exp(x)
double *x;
{
    *x = exp(*x);
}
```

Then

```
R COMPILE my_exp.c
```

Then load it. `LOAD`, `dyn.load`, ...

Then in R:

```
> x <- 5
> .C("my_exp", x)
```

R COMPILE uses a make file in
`$$HOME/newfun/lib/S_makefile`.

R from C (or C++)

This facility is limited. There are some documented C functions that are used in R. These are what are available.

```
#include <S.h>
my_rng(x, nx)
double *x;
long *nx;
{
    long i;
    for (i=0; i<&nx; i++)
        x[i] = unif_rand();
}
```

Could also call this C function from R with `.C(...)`.