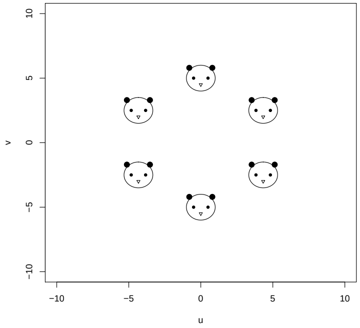
R: Programming

140.776 Statistical Computing

September 8, 2011

Functions

Functions can keep your program concise and readable. Proper use of functions makes debugging and maintenance much easier. For example:



Functions

```
## draw a panda at (p1,p2)
addpanda<-function(p1,p2) {
   # draw face
   x<-seq(0,2*pi,by=0.001)
   y<-p1+sin(x)
   z<-p2+cos(x)
   lines(y,z,type="l")
   # draw eyes
   points(p1-0.5,p2,pch=20)
   points(p1+0.5,p2,pch=20)
   # draw ears
   points(p1-0.8,p2+0.8,pch=20,cex=2)
   points(p1+0.8,p2+0.8,pch=20,cex=2)
   # draw mouth
   points(p1,p2-0.5,pch=25,cex=0.6)
}
```

Functions

```
## now let's draw six pandas
x<-seq(0,by=(2*pi/6),length=6)
y<-5*sin(x)
z<-5*cos(x)
u<-0
v<-0
plot(u,v,type="n",xlim=c(-10,10),ylim=c(-10,10))
for(i in 1:length(x)) {
   addpanda(y[i],z[i])
}
```

Functions

In R, functions are defined as follows:

```
funcname<-function(arg_1, arg_2, ...) {
        expr
}
```

Functions in R

- are objects of class "function" and can be treated much like any other R object;
- can be passed as arguments to other functions;
- can be nested, so that a function can be defined within another function.

The return value of a function is its last expression.

Functions

```
mean<-function(z) {
        zmean<-0
        for(i in seq_along(z)) {
                zmean<-zmean+z[i]
        }
        zmean<-zmean/length(z)
}

## compute mean for x
xmean<-mean(x)

## compute mean for y
ymean<-mean(y)
```

Named arguments and defaults

Functions have *named arguments* which potentially have *default values*. For example:

```
em<-function(data, tol=1e-6, maxiter=100) {
        ## function body
}
```

The defaults may be arbitrary expressions, even involving other arguments to the same function. They are not restricted to be constants. For example

```
ff<-function(x, n=x^2) {
        n^x
}

ff(2) = ?
```

Argument matching

When using the function,

- the argument sequence may be given in an unnamed, positional form;
- or it may be given in any order if specified in the "name=object" form.

```
> em(mydata)
> em(mydata, 1e-8, 1000)
> em(data=mydata, tol=1e-8)
> em(maxiter=1000, data=mydata)
> em(tol=1e-8, mydata)
```

Argument matching

You can mix positional matching with matching by name. When an argument is matched by name, it is taken out of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

```
> args(lm)
function (formula, data, subset, weights, na.action,
method = "qr", model = TRUE, x = FALSE, y = FALSE,
qr = TRUE, singular.ok = TRUE, contrasts = NULL,
offset, ...)
```

The following two calls are equivalent.

```
lm(data=mydata, y~x, model=FALSE, 1:100)
lm(y~x, mydata, 1:100, model=FALSE)
```

Argument matching

- Named arguments are useful when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list.
- Named argments also help if you can remember the name of the argument and not its position on the argument list (plotting is a good example).

Lazy evaluation

Arguments to functions are evaluated lazily, so they are evaluated only as needed.

```
f<-function(a,b) {
        a^2
}
f(2)
```

This function never actually uses the argument b, so calling f(2) will not produce an error because the 2 gets positionally matched to a.

Lazy evaluation

Another example:

```
f<-function(a,b) {
        print(a)
        print(b)
}

> f(45)
[1] 45
Error in print(b) : argument "b" is missing, with no default
```

Notice that "45" got printed first before the error was triggered. This is because b did not have to be evaluated until after print(a). Once the function tried to evaluate print(b) it had to throw an error.

The “...” argument

Often we wish to allow one function to pass on argument settings to other functions. This can be done using “...”. This is often used when extending another function and you don’t want to copy the entire argument list of the original function.

An example is the plot(x, y, ...) function, which passes on graphical parameters to par(). Here is another example:

```
myplot<-function(x,y,...) {
        plot(x,y,...)
}

> x<-rnorm(100)
> y<-x+rnorm(100)
> myplot(x,y,type="o",pch="1",col="blue")
```

The “...” argument

The “...” argument is also necessary when the number of arguments passed to the function cannot be known in advance.

```
> args(paste)
function (..., sep = " ", collapse = NULL)

> args(cat)
function (..., file = "", sep = " ", fill = FALSE,
  labels = NULL, append = FALSE)
```

Binding values to symbol

Sometimes you see code like this:

```
f<-function(x,y) {
        x^2+y/z
}
```

z is not an argument of the function, how is its value determined?

Binding values to symbol

When R tries to bind a value to a symbol, it searches through a series of *environments* to find the appropriate value. When you are working on the command line and need to retrieve the value of an R object, the order is roughly

1. Search the global environment for a symbol name matching the one requested.
2. Search the namespaces of each of the packages on the search list.

The search list can be found by using the search() function:

```
> search()
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

Classification of symbols

In general, how to bind values to symbols is determined by *scoping rules*. The symbols occurring in a function can belong to one of the three classes:

- **Formal parameters**: symbols that occur in the argument list of the function.
- **Local variables**: variables whose values are determined by the evaluation of expressions in the body of the function.
- **Free variables**: variables which are not formal parameters or local variables.

```
f<-function(x) {
        y<-2*x
        print(x)
        print(y)
        print(z)
}
## x: formal parameter; y: local variable; z: free variable.
```

Lexical scoping

In R, the values associated with free variables are resolved by first looking in the environment in which the function was created. This is called *lexical scoping*. Example:

```
cube<-function(n) {
        square<-function() {
                n*n
        }
        n*square()
}
```

n is not an argument of square(), but since square() was defined within cube(), the value of n can be determined by the n in the cube() function.

Environment

- An *environment* is a collection of (symbol, value) pairs, i.e. x is a symbol and 3.14 might be its value.
- Every environment has a parent environment; it is possible for an environment to have multiple “children”.
- The only environment without a parent is the empty environment.
- A function + an environment = a *closure* or *function closure*.

Lexical scoping

Searching for the value for a free variable:

- If the value of a symbol is not found in the environment in which a function was defined, then the search is continued in the parent environment.
- The search continues down the sequence of parent environment until we hit the top-level environment; this is usually the global environment (workspace) or the namespace of a package.
- After the top-level environment, the search continues down the search list until we hit the empty environment.
- If a value for a given symbol cannot be found once the empty environment is arrived at, then an error is thrown.

Lexical scoping

Why does all this matter:

- Typically, a function is defined in the global environment, so that the values of free variables are just found in the user's workspace.
- This behavior is logical for most people and is usually the "right thing" to do.
- However, in R you can have functions defined inside other functions (Languages like C don't allow this).
- Now things get interesting – In this case the environment in which a function is defined is the body of another function.

Lexical scoping

```
make.power<-function(n) {
        pow<-function(x) {
                x^n
        }
        pow
}
```

This function returns another function as its value.

```
> cube<-make.power(3)
> square<-make.power(2)
> cube(3)
[1] 27
> square(3)
[1] 9
```

Exploring a function closure

What's in a function's environment?

```
> ls(environment(cube))
[1] "n"   "pow"
> get("n",environment(cube))
[1] 3

> ls(environment(square))
[1] "n"   "pow"
> get("n",environment(square))
[1] 2
```

Lexical vs. dynamic scoping

```
y<-10

f<-function(x) {
        y<-2
        y^2+g(x)
}

g<-function(x) {
        x*y
}
```

What is the value of

```
f(3)
```

Lexical vs. dynamic scoping

- With lexical scoping, the value of y in the function g is looked up in the environment in which the function was *defined*, in this case the global environment, so y=10.
- With dynamic scoping, the value of y is looked up in the environment from which the function was *called* (i.e. calling environment; in R, calling environment is known as the *parent frame*), so y=2.

R uses lexical scoping, so

```
> f(3)
[1] 34
```