R Programming

Scoping

Variable Scope

- A *variable* is pairing of a name and a value.
- Variables can be created by assignment. The assigment

  ```
  x = 10
  ```

  indicates that the value “`10`” and the name “`x`” are to be paired.
- R has both *local* and *global* variables.
- Global variables are created by assignments at top level.
- Local variables are created by assigning values within functions.

Example: Same Name, Different Variables

There are two "`x`" variables in the following code.

```
x = 10                # <-- global

fun =
    function() {
        x = 20        # <-- local to fun
    }
```

Inside "`fun`," the value of "`x`" is "`20`."

Outside "`fun`," the value of "`x`" is"`10`."

Changes made to one "`x`" do not affect the other.

Visibility of Global Variables

Inside a function, the values of both local and global variables are visible.

```
> y = 20
> fun =
      function() {
          x = 30
          x + y
      }
> fun()
[1] 50
```

Here, "`fun`" combines the value of a local variable, "`x`," and the value of a global variable, "`y`."

Nested Functions

When R functions are nested, the variables of the outer functions are visible within the nested functions.

```
> outer =
      function() {
          inner =
              function() {
                  y = 20    # y is local to inner
                  x + y
              }
          x = 10            # x is local to outer
          inner()
      }
> outer()
[1] 30
```

Scoping Rules

- The scoping rules of a (computer) language govern how the value of variables can be determined.
- The scope of a variable is the region of code where that variable has meaning.
- In R, a local variable has meaning only within the function it is local to (including any functions that are nested within that function).
- When a value is sought for a given name, the local scope is inspected for a value, then any nesting scopes (i.e. functions) and finally a global value is sought.

Function Parameters, Arguments and Variables

- Assignment is not the only way that variables are created.
- The association of function's arguments with the function's formal parameters also creates variables.
- In this case the variable name is the formal parameter name and the (initial) value of the variable is corresponding function argument.

Example: Parameters and Arguments

- In the function definition

  ```
  fun =
      function(a, b)
      a^2 + b^2
  ```

  the *formal parameters* of the function are "`a`" and "`b`."

- In the function call

  ```
  fun(10, 20)
  ```

  the *arguments* to the function are "`10`"and "`20`."

- The arguments and parameters are paired as variables.

A Simple Example

The following function adds a local variable "`u`" to a global variable "`x`."

```
> add.x.to = function(u) x + u

> x = 10
> add.x.to(20)
[1] 30

> x = 20
> add.x.to(20)
[1] 40
```

More Complexity

Now let's embed the "add.x.to" function inside another function.

```
> add =
      function(x, y) {
          add.x.to = function(u) x + u
          add.x.to(y)
      }

> add(1, 2)
[1] 3
```

Now "x" is no longer global. It is a variable that is local to the function "add."

An Important Change

Suppose now we change the embedding function as follows.

```
> make.add.to =
      function(x, y) {
          add.x.to = function(u) x + u
          add.x.to
      }

> add1 = make.add.to(1)
```

Now, rather than returning the value that results from applying "`add.x.to`" to a value, we return the function itself

(The variable "`y`" is no longer used and we can drop from the definition of "`add.x.to`.")

What Does The Function Do?

We can now try applying the function "`add1`" to various arguments.

```
> add1(1)
[1] 2
> add1(2)
[1] 3
> add1(10)
[1] 11
```

Clearly "`add1`" is a function that adds "`1`" to its argument.

Does This Always Work?

Let's try making other functions with "`make.add.to`."

```
> add10 = make.add.to(10)

> add10(1)
[1] 11

> add10(100)
[1] 110
```

Clearly this is something that works in general!

Why?

The Process Explained

The “`make.add.to`” function is defined as follows.

```
> make.add.to =
      function(x)
          function(u) x + u
```

When “`make.add.to`” is called, a variable called “`x`” is created and set to the value of the argument to “`make.add.to`”.

The function defined inside “`make.add.to`” has access to the variable “`x`.”

This remains true, even after “`make.add.to`” has returned.

Closures

- The function returned by "`make.add.to`" is said to *close over* the value of "`x`."
- The function retains access to the variable "`x`" and can use its value whenever it is called.
- Functions that do this are called *closures*.
- Forming closures is one of R's most important capabilities.
- It is also one of the least-understood and least-used.

Defining A Likelihood

The following function creates closures that represent Poisson likelihood functions.

```
> mk.pois.lk =
      function(x)
      function(lambda) prod(dpois(x, lambda))
```

Given a vector containing the values of n Poisson random variables, the “`mk.pois.lk`” function returns a function of “`lambda`” that computes the likelihood function for the Poisson parameter.

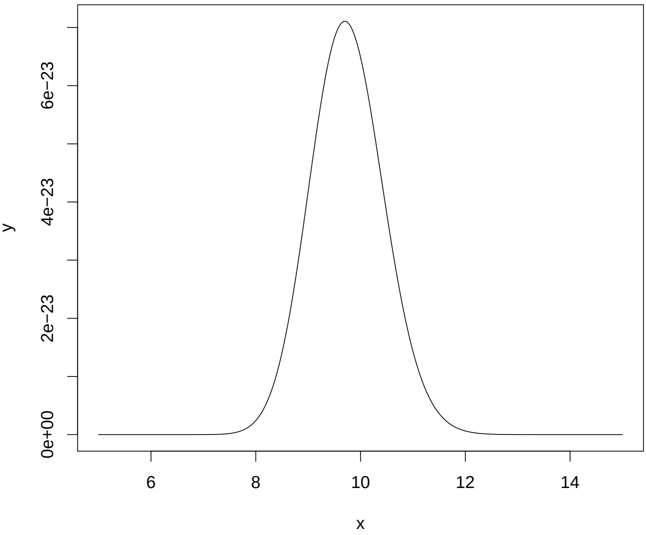
$$\begin{aligned} L(\lambda) &= f_{X_1,\ldots,X_n}(x_1,\ldots,x_n;\lambda) \\ &= \prod_{i=1}^{n} f_{X_i}(x_i;\lambda) \end{aligned}$$

Using the Likelihood

Once a likelihood function is defined, it can be plotted or maximized to obtain a maximum likelihood estimate λ.

```
> PoissonData = rpois(20, 10)
> L = mk.pois.lk(PoissonData)
> x = seq(5, 15, length = 1001)
> y = sapply(x, L)
> plot(x, y, type = "l")
```

(Note that the "`sapply`" call is necessary because "`L`" is not vectorized.)



0e+00
2e−23
4e−23
6e−23
y
6
8
10
12
14
x

Computing a Maximum Likelihood Estimate

The likelood function can be maximized using the "`optimize`" function.

```
> optimize(L, c(8, 12), maximum = TRUE)
$maximum
[1] 9.699984

$objective
[1] 7.107315e-23
```

This tells us that the maximum-likelihood estimate of λ in this case is about 9.7. Likelihood theory also provides a way of getting standard errors for the estimate (See STATS 330 for the details).

Why Are Closures Useful?

- In the last example, we could have just used a global variable "`x`" to hold the data values for the likelihood.
- However, if we need several likelihood functions, they can't all keep their data in the same "`x`."
- Closures provide a way for functions to have their own private copies of data.
- This provides a very powerful programming tool.
- It gets used a lot in other programming languages, but not so much in R.

Alternative Way of Creating Closures

- There are a number of ways of creating closures that provide a more direct way of creating closures.
- The most direct of these are based on the "`with`" and "`local`" functions.
- They both work by directly providing a set of variables that can be closed over.
- (The use of function calls to return closures can have some rather nasty side-effects.)

Creating Closures using “with”

- The “`with`” function makes the elements of a named list available as variables to evaluate and expression.
- If this expression creates a function, it closes over the variables made from the list.

```
> L = with(list(x = PoissonData),
           function(lambda)
           prod(dpois(x, lambda)))
```

- This produces a closure that performs exactly like the earlier one created with “`mk.pois.lk`.”

Creating Closures using “local”

- Another direct way of creating closures is to establish local scope using the “`local`” function.
- Again, a function returned from this local scope closes over the variables defined in that scope.

```
> L = local({
        x = PoissonData
        function(lambda)
        prod(dpois(x, lambda))
    })
```

- This is the method I most commonly use for creating closures (although I do sometimes use the other methods too.)

Using Closures to Hide Helper Functions

- Often when writing software it is common to decompose a problem into smaller tasks and write functions to carry out these smaller tasks.
- Usually there is really only one function of interest, which is the one that solves the “big” problem.
- In that case there is no reason for all the other functions and associated data to be visible.
- Closures provide a perfect way of hiding things you don’t want visible.

Example of Hiding Helper Functions

The following code shows how to hide two helper functions in a closure. (The closure computes binomial probabilities, directly from the formula.)

```
> binp = local({
      fact =
          function(n) gamma(n + 1)
      bincoef =
          function(n, k)
              fact(n)/(fact(k) * fact(n - k))
      function(k, n, p)
          bincoef(n, k) * p^k * (1 - p)^(n - k)
  })
```

Hiding Helper Functions (Continued)

- The "`binp`" function closes over the two functions "`factor`" and "`bincoef`."
- Those two functions are visible in the body of "`binp`," but invisible to anything else.

```
> binp(0:5, 10, .5)
[1] 0.0009765625 0.0097656250 0.0439453125
[4] 0.1171875000 0.2050781250 0.2460937500

> objects()
[1] "binp"
```

Defining Anonymous Recursive Functions

- The factorial function can be defined recursively as follows.

```
> fact =
      function(n)
      if (n <= 1) 1 else n * fact(n - 1)

> fact(10)
[1] 3628800
```

- The factorial calls itself by using its own name.
- This makes it seem like the factorial function must be assigned a fixed name (in this case “`fact`”).

An Anonymous Factorial Function

- Using a closure, it is possible to define an anonymous version of the factorial function.

```
> (local({
      me = function(n)
          if (n <= 1) 1 else n * me(n - 1)
  }))(10)
[1] 3628800

> objects()
character(0)
```

- This factorial function has a hidden name, known only to itself, that it can use to call itself.