R Functions

Things Your Mother (Probably) Didn't Tell You About

R and Extensibility

- The success that R currently enjoys is largely because the environment is *extensible*.
  - Developers can easily add new capabilities.
  - Users can quickly develop and customise their own methodology.
- Both developers and users implement their extensions in the same way — as new R functions.
- This uniform method of extension provides a certain unity to the process of R development and it is natural to move from being a user to being a developer.

What Is An R Function?

- An R function is a packaged recipe that converts one or more inputs (called arguments) into a single output.
- The recipe is implemented as a single R expression that uses the values of the arguments to compute the result.
- Functions are first-class values. They can be:
  - assigned as values of variables
  - passed as arguments to other functions

An Example

- The following function takes a single input value and computes its square.

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

- The function is created and then assigned the name `square`.
- The variable, `x`, is a *formal parameter* of the function.
- When the function is *called* it is passed an *argument* that provides a value for the formal parameter.

```
> square(1:5)
[1]  1  4  9 16 25
```

Combining Functions

- Functions defined by users are identical in nature to those provided by the system and can be used in exactly the same way.

```
> sum(1:10)
[1] 55

> sum.of.squares =
      function(x)
      sum(square(x))

> sum.of.squares(1:10)
[1] 385
```

Optional Arguments

- R functions can have many arguments (the default `plot` function has 16).
- Function definitions can allow arguments to take *default values* so that users do not need to provide values for every argument.
- If the plot function is called with a single argument it is used to provide *y* values for the plot; all other arguments take on default values.
- Default arguments are specified as follows:

    `parameter =` *expression*

Example

- The following variant of `sum.of.squares` adds a second parameter so that the function returns the sum-of-squares of deviations about the second value.
- The second argument has a default value equal to the mean of the first.

```
> sum.of.squares =
      function(x, about = mean(x))
      sum(square(x - about))
```

- Note that the default argument value is defined in terms of variables internal to the function.

Example (Continued)

- Since many arguments can take default values, it is useful to have a way of specifying which arguments do not.

```
> sum.of.squares(1:10)
[1] 82.5

> sum.of.squares(1:10, about = 0)
[1] 385
```

- There is a set of rules the determine how arguments are matched to parameters.

```
> sum.of.squares(about = 0, 1:10)
[1] 385
```

Example (Continued)

- The present definition of `sum.of.squares` does not work when `NA` values are present.

```
> sum.of.squares(c(-1, 1, NA))
[1] NA
```

- The inclusion of an `NA` value produces an `NA` result.
- It may well be that we want `NA` values ignored to produce the same result as:

```
> sum.of.squares(c(-1, 1))
[1] 2
```

Example (Continued)

- Let's modify the `sum.of.squares` function so that it removes any `NA` values from `x`.

```
> sum.of.squares =
      function(x, about = mean(x)) {
          x = x[!is.na(x)]
          sum(square(x - about))
      }

> sum.of.squares(c(-1, 1, NA))
[1] 2
```

Example (Continued)

- Let's modify the `sum.of.squares` function so that it removes any `NA` values from `x`.

```
> sum.of.squares =
      function(x, about = mean(x)) {
          x = x[!is.na(x)]
          sum(square(x - about))
      }

> sum.of.squares(c(-1, 1, NA))
[1] 2
```

- This produces the "right" result, but the fact that it does so is surprising.

Lazy Evaluation

- R function arguments are not evaluated until the value of the argument is needed.

- In the case of the preceding example, the value of `about` is not required until the expression

  ```
  sum(square(x - about))
  ```

  is evaluated.

- At that point, the `NA` values have been deleted from `x` so that the value of `mean(x)` is not `NA`.

Lazy Evaluation and Side Effects

- Because argument evaluation is lazy, it is dangerous to ever carry out assignment (or any operation with a side effect) in an argument to a function.

```
> x = 10
> y = 20
> f((x = 100), (y = 200))
[1] 300
```

Lazy Evaluation and Side Effects

- Because argument evaluation is lazy, it is dangerous to ever carry out assignment (or any operation with a side effect) in an argument to a function.

```
> x = 10
> y = 20
> f((x = 100), (y = 200))
[1] 300
> x; y
[1] 10
[1] 20
```

Lazy Evaluation and Side Effects

- Because argument evaluation is lazy, it is dangerous to ever carry out assignment (or any operation with a side effect) in an argument to a function.

```
> x = 10
> y = 20
> f((x = 100), (y = 200))
[1] 300
> x; y
[1] 10
[1] 20
```

- This is because the function `f` is defined as follows.

```
> f = function(a, b) 300
```

Scoping

- The *scoping rules* of a language describe how the values of variables are determined.

- R uses *block-structured scope*, similar to languages like Algol-60 and Pascal and Scheme.

- If a function `g` is defined within a function `g`, the variables in `f` are visible in `g`, unless they are *shadowed* by a local variable.

- The use of these scoping rules make R a very different language from the earlier S language developed at Bell Laboratories.

Example

- Consider the following nested function definition.

```
> linmap =
      function(x, a, b, swap = FALSE) {
          transform = function(x) {
              if (swap) b + a * x
              else a + b * x
          }
          transform(x)
      }
```

- Within the function `transform`, the variable name `x` refers to the argument of `transform` while `a`, `b` and `swap` refer to the arguments of the enclosing `linmap` function.

A Simple Function

- The following function adds the value of the global variable `x` to its argument.

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

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

> x = 30
> add.x.to(10)
[1] 40
```

Nested Functions

- The function `add.x.to` looks just like the previous one, but now the value of `x` is an argument to the enclosing function `add`.

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

> add(10, 20)
[1] 30
```

A Function that Returns a Function

- Now we'll change the example so that instead of returning a numeric value the outer function returns the inner function.

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

> add.10.to = make.add.to(10)
> add.10.to(100)
[1] 110
```

Variable Capture and Closures

- In the previous example, the variable `x` came into existence when the outer function `make.add.to` was called.
- This variable continues to exist after `make.add.to` returns because it is required for the value returned by `make.add.to` to make sense.
- The outer functions local variable `x` has been *captured* by the function returned as a value.
- The variable `x` is, in a sense “enclosed” within the function returned by `make.add.to`.
- Functions that enclose data in this way are called *closures*.

Captured Variables are Private

- Each time `make.add.to` is called, a new `x` variable is created.

```
> add.10.to = make.add.to(10)
> add.20.to = make.add.to(20)

> add.10.to(100)
[1] 110
> add.20.to(100)
[1] 120
```

- This means that each function returned by `make.add.to` has its own private `x` variable.

Other Ways of Creating Private Variables

- The use of nested functions is not the only way to create private variables.
- Here are some alternatives.

```
> add.10.to =
      with(list(x = 10),
           function(u) x + u)

> add.20.to =
      local({
          x = 20
          function(u) x + u
      })
```

How This is Useful

- The ability to create closures might seem like a fairly esoteric capability, but it provides a way to directly provide many kinds of object used directly in statistics.
- The mechanism is used in many R functions (e.g. `splinefun`).
- I'll show just one example: likelihoods.

Likelihoods

- Here is a function that creates a function that computes the negative log likelihood for a sample of normal observations stored in a vector `x`.

```
> negloglike =
      local({
          x = rnorm(100)
          function(theta)
              -sum(log(dnorm(x,
                             theta[1],
                             theta[2])))
      })
```

Likelihood-Based Estimation

- Given the negative log likelihood it is easy to obtain parameter estimates and standard errors.

```
> res = optim(c(0, 1), negloglike,
              hessian = TRUE)

> res$convergence
[1] 0

> res$par
[1] -0.03126232  0.86081820

> sqrt(diag(solve(res$hessian)))
[1] 0.08608182 0.06087361
```

Other Applications

- Many statistical problems can be attacked using likelihood-based analyses, even when they have a non-standard form.
- Markov chains with their associated transition matrices and current states are naturally modelled as closures.
- Complex software can be written without worrying about “namespace clutter.”
- The R package facility is implemented using these ideas.
- The S4 object system is implemented using closures.

Recursion

- The Devil's DP Dictionary defines recursion as follows:

  Recursion (n). *See Recursion*.

- In computing, a function is recursive if, either directly or indirectly, it can make a call itself.
- The prototypical example of recursion is the factorial function.

```
> factorial =
      function(n)
      if (n == 0) 1 else n * factorial(n - 1)
```

Example: Computation Using Recursion

- In the good old days the following kind of problem would have been found in an introductory statistics course:

  > *There are 8 girls and 4 boys in a class. How many ways can they be arranged in a line so that the boys are separated by at least one girl?*

- (These days, questions that require thought lead to bad class reviews and they've been done away with.)
- There is a trivial solution to this problem, but let's assume that we aren't smart enough to spot it.
- Instead, we'll attack the problem using recursion.

Formulating the Problem as a Recursion

- First let's generalise to the case of g girls and b boys.
- If the number of arrangements is $f(b,g)$, then we have the following recursion.

$$f(b,g) = g \times f(b,g-1) + b \times g \times f(b-1,g-1)$$

- This recursion comes from considering what happens when we pick either a girl or a boy as our first choice.
- In addition to the basic recursion, we also need ensure that there are termination rules that provide a way of stopping the recursion.

Termination Rules

- The consideration of special cases gets us a number of termination rules.

Condition	Function Value
$b = 1, \quad g = 0,$	$f(b, g) = 1$
$g < b - 1$	$f(b, g) = 0$
$b = 0$	$f(b, g) = g!$

A Computational Solution

```
> f =
      function(b, g) {
          if (b == 1 && g == 0) 1
          else if (g < b - 1) 0
          else if (b == 0) factorial(g)
          else g * f(b, g - 1) +
              b * g * f(b - 1, g - 1)
      }

> f(4, 8)
[1] 121927680
```

A Computational Solution

```
> f =
      function(b, g) {
          if (b == 1 && g == 0) 1
          else if (g < b - 1) 0
          else if (b == 0) factorial(g)
          else g * f(b, g - 1) +
              b * g * f(b - 1, g - 1)
      }

> f(4, 8)
[1] 121927680

> factorial(8) * prod(9:6)
[1] 121927680
```

The Number of Function Calls

- The evaluation of `f(4, 8)` takes 307 calls to `f`.
- Of these, 306 are calls by `f` to itself.