Data Types, Control Flow, Functions & Programming

Topics

- Data types
- Vectorization
- Missing values
- Function calls and semantics
  - copying values, lazy evaluation, scope & symbol evaluation.
- Control flow
- Writing functions
  - Mechanics, style
- Debugging
  - Static & dynamic/run-time, tools
- Profiling
  - Tools – Rprof(), summaryRprof().

Data Types

- In comp. science, focus is often on many different rich data types
  - E.g. hash tables, hash algorithms, linked lists, balanced trees, red-black trees, graphs, …
- In R/MATLAB/…, few primitive data types
  - Logical, integer, numeric, complex,
  - Character
  - List
  - Raw – for binary data.

Vectorization

- Critically, there are no scalars, i.e. individual values in R/MATLAB, but vectors
  - Ordered containers for zero or more values.
  - Scalars are vectors of length 1.
  - Everything has a length – length(obj) gives something reasonable.
- The language is heavily oriented to vector operations, and gets its efficiency from this
  - i.e. if you don't use vectorized computations, code runs much slower than if you do

Vector operations

- Operators such as +, -, &, |, !, …… are vectorized
  - x + 1 , x + y, !x work element-wise and produce a new object, derived from the inputs.
- Very different from
  y = c()
  for(i in seq(along = x)) y = c(y, x[i] + 1)
- Or y = numeric(length(x))
  for(i in seq(along = x)) y[i] = x[i] + 1

Subsetting

- Major part of the language, and programmatically extensible.
- vector[indices] – extract and also assign to sub-elements
- Index by
  - position,
  - Named elements
  - Logical vector (same length as vector)
  - negative numeric index to exclude, i.e. complement of
  - Empty index, i.e. x[]

Missing Values

- Important concept of "missing" represented as the literal/constant NA
  - is.na(x) – returns logical vector with TRUE where value is NA.
- What is 1 + NA?

Coercion

- R will implicitly coerce objects to appropriate type.
- Consider
  - c(TRUE, 1L, 2, 3.14, "3.14")
  - x + TRUE, x + 1
- What type is NA?

Recycling rule

- In many cases, R will perform an operation on 2 or more vectors and work on them element-wise, i.e. the i-th element of each of them.
- Implicitly, the model is often
  func(x[i], y[i], z[i], …)
- This implies the vectors have the same length
- R makes this happen transparently using the recycling rule.
  - Makes the shorter ones longer to have the same length as the longest one.
  - If this is an integer multiple, no problem. If not, a warning.

Factors/Categorical Variables

- Need to represent data that can take on only a finite set of values from a fixed set.
- factor() and ordered factors (ordered()) do this.
  - Vector of values, along with the "labels"
  - Represented as
    - an integer vector with an element for each observation,
    - a "shorter" vector of labels
    - The integers index the vector of labels to identify the category of each element.

Factors

- So a factor is a special integer vector.
- Can use it to subset in very convenient and powerful ways.
- But prints and behaves differently from an integer in other cases.
- Ordered factors merely have an order specified for the labels, e.g. A, B, C, D, F.

Higher-level types

- Data frames – a list of variables that act as columns, but also second dimension giving rows
  - Data frame forces length of all columns to be the same
  - So special type of list.

- When I teach,
  - sometimes I start with primitive types and build up to lists and data frames.
  - Other times I start with a data frame that I give the students, and work down to the primitive vector types.
  - The latter gets subsetting in early, and gives them a real data set to explore.

Matrices

- Matrices – vectors with attribute giving dimensions as a vector
  - Rarely teach these in programming class as we use data frames to represent data. Matrices arise in linear algebra computations or output of pairwise distance calculations.
- Matrices are similar to 2-D data frames, but require all elements to be of the same type, i.e. coerced to common type. So cannot have numerics and strings and factors.
- Reliance on matrices often suggests confusing representation & implementation.

Parameter Matching

- R has a very different and flexible way of passing arguments to functions
  - by position
  - by name
  - by partial name
- Default values for some or all parameters
- And in some cases, even parameters that do not have a default value do not need to be specified.

Argument Matching

3 steps to argument matching.

- match all the named arguments that match a parameter name exactly
  (match & remove these)
- match all the named arguments that partially match ambiguous matches (matches 2 or more parameters) is an error
  (match & remove these)
- match the remainder arguments to the unmatched parameters by position

others picked up by ... if present

raise error if any left over

Argument Matching

- Consider the function rnorm()
- > rnorm
  function (n, mean = 0, sd = 1)
- rnorm(10) # mean, sd take default values
- rnorm(10, 2) # mean = 2, sd = 1 - default value rnorm(10, , 2) # mean = 0 (default value), sd = 2
  rnorm(10, sd = 2) # mean = 0 (default value), sd = 2
  rnorm(10, s = 2) # partial matching – sd = 2
  rnorm(sd = 2, 10) # n = 10, mean = 0, sd = 2

...

- Some functions have a parameter named "..."
- This means zero or more arguments which do not match any of the other parameters.
- Two purposes for …
  - collect arbitrary number of arguments together to be processed together,
  - e.g. sum(..., na.rm = FALSE)
    elements are collected into a vector and added (compare with sum(x) or sum(x, y))

...

- arguments that are passed directly to other functions called by this function
  e.g. used a lot in graphics functions to pass common named arguments like col, pch to lower-level functions.
- Or
  lapply(1:3, rnorm, sd = 3)
- lapply accepts additional arguments via its ... parameter, and passes them in the call to
  FUN(x[i], ...)
- R inlines the ... in the call to the function, with the names and all.

Argument Matching

- arguments that do not match by name or position are collected into the ... "list".
- In most languages, ... must come last.
  But in R, some functions have additional parameters after the ...
  - users must specify values for these parameters by using the fully specified parameter name name, e.g.
    cat(..., file = "", sep = " ")
    cat("some text", " to display", file = "myFile", sep ="")

Copying Objects

- Create a collection of 10 random normal values
  x = rnorm(10)
- Assign x to new variable y
  y = x
- This is a copy of x
  (actually both x and y initially point to a single shared array in memory)
- But if we change x, e.g.
  x[1] = 100
  y retains its current value and x and y are different.

Copying Arguments

- When x[1] = 100 is evaluated, R recognizes that the data elements are shared by two or more variables and so makes a copy of it and then modifies the first element of the copy. Then it assigns the result to 'x'.
- The original shared copy of the vector remains unchanged and is tied to y.

Computational Model

- So the computational model is that assignments (almost always) appear to make a copy of their data.
- So in a function call,
  foo(x, y)
  we create a new frame
  evaluate x in the caller's frame
  assign the value to the first parameter
  Hence we are making a copy of that value.
- Any changes made to the first parameter within the body of the function will have no effect on the variables in the caller's frame.
- All changes of interest to the caller must be explicitly returned by the function.

Lazy Evaluation

- In many languages, f(x, y+2, z+3) would first get x, y+2, z+3 and then pass those in the function call.
- Unusual feature of R is that expressions passed to a function are not evaluated until the parameter is actually used - lazy evaluation.
- Avoids unnecessary computations and copies of data.
- Has implications for writing code
  if expression has side effects, not certain when or if it will occur
- and implications for writing functions

Control Flow

Apply functions

- Before teaching for() loops, etc., focus on
  - Vectorized computations
  - Family of apply() functions

  Apply functions
  loop over elements of a vector,
  apply the function to that element,
  place result in a new vector,
  Return new vector with all of the results in order.

- Apply functions are terse
  - Compare with

```
ans = list()
for(i in seq(along = x))
  ans[i] = func(x[[i]])
```

- Much easier to read and reason about
- Explicitly indicate that the order of processing is not important
  - Useful for parallel computing.
- Sometimes more efficient

- lapply(), sapply() & vapply().
- apply() for matrices and multi-dimensional arrays
- tapply() – groups observations by one or more factors and applies function to subsets
  - See by() and aggregate()
- mapply() -

- Control flow relates to language constructs that determine which commands are executed next.
- The if() … else ... construct is for conditional branching, i.e. running one set of expressions or another.
- while(condition) { … }
  - Code in … may change condition for next iteration.
  - repeat() is do-while loop.
- next & break for skipping or ending iterations.

- for(var in obj) { … }
  - var is assigned each element of obj in turn
  - Where possible, use apply() functions (sapply(), lapply(), apply(), tapply()) rather than loops
    - Difficult to get students to use these, but important step to have them think about good programming.

Issues with if()

- The condition in the if() should return something that can be treated as a "scalar" logical.
- If it is of length 0, bad things will happen.

```
X = logical()
Error in if (x) cat("Hi\n") else cat("Bye\n") :
   argument is of length zero
```

- If it evaluates to NA, bad things will happen.

```
x = NA
if(x) cat("Hi\n") else cat("Bye\n")
Error in if (x) cat("Hi\n") else cat("Bye\n") :
  missing value where TRUE/FALSE needed
```

Writing Functions

- Challenge to move students from commands to reusable functions.
- Want to map complete task into sub-tasks that correspond to natural level of granularity.
  - Sub-tasks correspond to functions
  - Functions can be called with different inputs and so reused within the current top-level task, tested separately, optimized and replaced easily.
  - Can be reused in other projects.
- Designing functions is an iterative process and one that benefits greatly from experience built by doing it and reusing the functions.

- function(x, y, myParam = 2) {
  # code that process inputs
  # creates intermediate temporary variables
  # returns single object as a result
  # can be a list with multiple objects
  }

Functions are top-level objects

Assign them to variables if you want.

Default values

- If a call to a function doesn't specify a value for a parameter, then its value is "missing"
  - Can test in the function body with, e.g. missing(y)
- A missing parameter will take on the value given by evaluating the expression for its default value in the definition of the function.
  - This is evaluated in the call frame of the function.
  - Can refer to other parameters in the function.
- Sensible default values are very powerful
  - Allow advanced caller to control more details, but regular usage to be simple.

Designing Functions

- Keep functions short and focused on one task
- As one writes 2 or 3 expressions to do something within a function, think whether it is best to make that a function.
- Can one replace the 2-3 expressions with a verb with a suitable name.
- Have functions avoid side effects.
  - Functional programming languages.

Finding scoping problems

- Consider a simple function

```
myFun =
 function() {
  if(x)
    10
  else
    21
 }
```

- What will myFun() do & return?
- If we are lucky, an error; if not, the wrong answer, or worse still coincidentally the correct answer for this one call.
- We can find references to non-local/global variables with

```
library(codetools)
findGlobals(myFun, FALSE)
```

- The element "variables" in the result is a character vector giving the names of the variables which are referenced but not defined in the parameters or local variables of the function.
- We can use this to perform basic tests on students code.
- Students should do it themselves as a way to check their code and save time.
- This is the type of issue a compiler would catch, but since we are not using a compiler …

Scope, Call Frame Stack, Environments

Debugging Code

- Inside or outside of functions, students should learn to use the tools that trap errors when the occur and allow the user to examine the state of the computations - the call stack, the value of variables in the different call frames.
- Randomly changing code, or not reading error messages is very natural, but a terrible habit to get into.
- Many students don't think it is worth the time to learn how to debug efficiently, assuming that each bug will be the one and only and last one they will encounter.
- It is also vital that they know how to debug other people's code, not just their own, as errors will occur in these other functions because of inputs they provide.

Debugging Tools

- There are two styles of debugging – preemptive and "post mortem".
- By preemptive, we mean that we want to stop at a particular point before an error occurs and control when & how each expression is evaluated. So we can go step-by-step and look at the current values at each step.
- Post-mortem means stopping only when there is an error (or warning if desired).

Preemptive Debugging

- This is quite common in many languages using a debugger to explicitly stop at a given expression, or when a particular condition is true when a given expression is about to be evaluated.
- There are graphical interfaces for debugging R code, but the standard tools are "functional" in nature.

- We can debug() or trace() a function.
  debug(myOtherFun)
  myFun(10) # which calls myOtherFun
- When myOtherFun is called – directly or indirectly – R will stop the evaluation and allow us to look around using the "debugger" tools.
- This is like a regular R prompt, but we are now "in" the call frame of the function be debugged and not the global environment.
- R commands we evaluate will be local to this call frame.
- We can even change values before proceeding.

- To evaluate the next expression, use 'n'.
- To just continue as normal with all the expressions, 'c'.
- Problems when trying to see a variable named n or c or any of the reserved control "words".
- Use get("n").

- To turn off debugging a function, use undebug(myFunc)

Post-mortem debugging

- Arrange to enter the debugger when an error occurs
- Can do this by calling traceback() after the error occurs and back at the top-level R prompt.
- Alternatively, stop when the error occurs
- The primary functions are browser() and recover()
  - options(error = recover)
  - I set this in my .Rprofile, read when the R session starts.;

Efficiency – timing code

- system.time(expression) returns the amount of time it took to evaluate the expression
- 3 measurements (or 5 if sub-processes)
  - User, system and total elapsed time.
  - Want user + system = total, or else something else consuming the machine so problematic measurement.
- Students should explore how time changes with size of the inputs.

- Issues with resolution and measuring time for small increments.
- Plot time versus input size and see if algorithm is linear, polynomial, exponential
  - Empirical algorithmic complexity

- To improve speed, use different algorithm.
- Or refine code
  - move expressions within loops that are invariant to compute just once and assigned to variable
  - Avoid concatenating vectors, but pre-allocate and assign to i-th element.
    - i.e. x = c(); for(i in seq(along = y)) x = c(x, g(y[i]))

Profiling

- Identify bottlenecks by measuring what functions are called the most often and take the most time.
- Rprof() function turns on measuring in the evaluator, producing data about function calls.
- Rprof("myProfilingData")
  Evaluate expressions
  Rprof(NULL)
- Data is now available in the file myProfilingData.

Analyzing profiling data

- Read profiling data into R via summaryRprof()
  prof = summaryRprof("myProfilingData")
- List with 3 elements
  - Look at prof$by.self
  - self.time self.pct total.time total.pct

"sample"	0.94	52.2	1.06	58.9
"rw2d1"	0.70	38.9	1.80	100.0
"length"	0.08	4.4	0.08	4.4
"=="	0.04	2.2	0.04	2.2
"+"	0.02	1.1	0.02	1.1
"-"	0.02	1.1	0.02	1.1

- Focus energy on improving the calls to the functions in the first rows of this data frame.
- Rewrite algorithm, adjust expressions.