R: Metaprogramming and Expressions

Debashis Ghosh

Chapters 13/14, Advanced R

October 14, 2019

Non-standard evaluation

- In most programming languages, you can only access the values of a function's arguments
- In R, you can also access the code used to compute them
- This makes it possible to evaluate code in non-standard ways, also know as **non-standard evaluation**, or NSE

`substitute()` in R

- `substitute()`: looks at a function argument and instead of seeing the value, it sees the code used to compute the value
- It returns an **expression**, an object that represents an action that can be performed by R.
- The arguments of `substitute()` are represented by a special type of object called a **promise**, which captures the expression needed to compute the value and the environment in which to compute it.
- Once the expression is evaluated, the promise goes away

NSE in `subset()`

- Useful interactive shortcut for subsetting data frames: instead of repeating the name of data frame many times, you can save some typing
- How does it work?
- It uses `eval()`, a function that takes an expression and evaluates it in the specified environment, and `quote()`, which captures an unevaluated expression and returns its input as is

Scoping issues

- If `eval()` cannot find the variable inside the data frame (its second argument), it looks in the environment of `subset2()`.
- That's not what we want, so we need some way to tell `eval()` where to look if it cannot find the variables in the data frame.

eval

- One of the arguments in `eval()` is called `enclos`
- It tells the `eval()` function in which enclosing environment to look for the variable.
- This is an example of **dynamic scoping**, which is counter to R's usual rules of **lexical scoping**
- Setting the `enclos` value is a shortcut for converting a list or data frame to an environment.
- We can get the same behavior using `list2env()`

Downsides of Non-Standard Evaluation

- Loss of referential transparency: "An expression is said to be referentially transparent if it can be replaced with its value without changing the behavior of a program (in other words, yielding a program that has the same effects and output on the same input)." from `https://en.wikipedia.org/wiki/Referential_transparency`
- "Non-standard evaluation allows you to write functions that are extremely powerful. However, they are harder to understand and to program with. As well as always providing an escape hatch, carefully consider both the costs and benefits of NSE before using it in a new domain."

Expressions

- In NSE, we learned how to access and evaluate expression underlying computation in R
- We can manipulate these expressions with code
- There are 4 components of an expression: constants, names, calls, and pairlists

Structure of expressions

- There is a distinction between an operation and a result in R
- We want to distinguish the action of code with assigning the result to an object
- As we saw last class, we can capture the action with `quote()`
- `quote()` returns an expression, which is an object that represents an action that can be performed by R

Expressions continued

- An expression is also called an abstract syntax tree (AST) because it represents the hierarchical structure of the code
- We can use `pryr::ast()` to see this

Constants

- Length one atomic vectors, like "b" or 2
- `ast()` displays them as is

Names

- Names, or symbols, represent the name of an object rather than its value
- Names are prefixed with a backtick

Calls

- Action of calling a function
- Are recursive, like lists
- They can contain constants, names, pairlists, and other calls
- `ast()` prints `()` and then lists the children
- The first child is the function that is called, and the remaining children are the function's arguments

Pairlists

- Only used in one place: formal arguments of a function
- `ast()` prints `[]` at the top level of a pairlist
- Are recursive and can contain constants, names and calls

Expression components

- Let's look at each in more detail...

Names

- Typically, `quote()` is used to capture a name
- You can also convert a string to a name using `as.name()` or `as.symbol()`

Calls

- A call, like a list, has length, [[and [methods, and is recursive because calls can contain other calls
- The first element of the call is the function that gets called

Modifying a call

- You can add, modify, and delete elements of the call with the standard replacement operators, $\$< -$ and $[[< -$
- Be careful with R's flexibility w.r.t. function calling semantics

Capturing a call

- Many base R functions use the current call: the expression that caused the current function to be run
- There are two ways to capture a current call:
  - `sys.call()` captures exactly what the user typed
  - `match.call()` makes a call that only uses named arguments
  - Like automatically calling `pryr::standardise_call()` on the result of `sys.call()`

Review: environments

- We can use `ls()` to list the variables and functions in the current environment
- We can use `environment()` to get the current environment
- When we define a function, a new environment is created

Review: scope

- Global variables exist throughout the execution of a program
- Global variables also depend on the perspective of a function
- Local variables exist only within a certain part of a program (for example, in a function)