Note 9. Environment & Scope

Introduction to Statistical Programming

Chanmin Kim

Department of Statistics
Sungkyunkwan University

2022 Spring

Environments

- What is an environments?
  - An environment is a collection of (symbol, value) pairs (i.e., a place to store those pairs) (e.g., `x=1`: `x` is a symbol & `1` is its value).
  - Every environment has a parent environment; it is possible for an environment to have multiple 'children'.
  - When an R session is started, a new environment, call the *global environment* or *workspace*, is initialized for objects created during the session.
  - A user mostly interacts with the *global environment*.
  - When a function is called, a new environment is created within the body of the function, and the arguments of the function are assigned to symbols in the local environment.
  - A function + an environment = a *closure* or *function closure*.

Why Environment?

- How does R know which value to assign to which symbol?
- How does R handle duplicated symbols?
- E.g.,

```
> seq
function (...)
UseMethod("seq")
<bytecode: 0x000000001127eff0>
<environment: namespace:base>

> seq <- function(x) x+x
> seq
function(x) x+x
```

Why Environment?

```
> f <- function(y)
+ {
+   a <- 3
+   return(y+a)
+ }
> a <- 2
> f(a)
[1] 5
> y
Error: object 'y' not found
> a
[1] 2
```

- $\Rightarrow$ R should distinguish places to store variables (environments) and decide the order to match values with symbols (scoping rule).

Matching values with symbols

- When R tries to matching pairs, it searches through a series of environments to find the appropriate value.
- When you are working on the commend line (i.e., global environment) and match a value with a symbol, the order is as follows:
  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 `search()`.

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

Matching values with symbols

- The global environment or the user's workspace is always the first element of the search list and the base package is always the last.
- When a user loads a package with `library` the namespace of that package gets put in position 2 of the search list (by default) and everything else gets shifted down the list.
- Note that R has separate namespaces for functions and non-functions (i.e., it's possible to have an object named c and a function named c).

```
> c <- 3; c
[1] 3
> c(1,5,3)
[1] 1 5 3
```

Variables in Functions

- Functions have 3 types of symbols as follows:
  - Formal parameters: Arguments of the function.
  - Local variables: Variables created in the function.
  - Free variables: Variables created outside of the function

- E.g.,

```
f <- function(x) {
  y <- 2 * x / z; return(y) }
```

  - `x`: Formal parameter.
  - `y`: Local variable.
  - `z`: Free variable.

- The scoping rules determine how values are assigned to free variables.

Scoping Rules

- Scoping rules:
  - Lexical scoping: Free variables in the function are searched for the environment in which the function was defined.
  - Dynamic scoping: Free variables in the function are searched for the environment from which the function was called.

- E.g.,

```
y <- 10
f <- function(x) {
  y <- 3
  2*y + g(x) }
g <- function(x)  x*y
f(3)
???
```

  - Lexical scoping: `f(3) = 36`.
  - Dynamic scoping: `f(3) = 15`.

- R supports the lexical scoping rules (Perl, Python: lexical scoping).

Lexical Scoping Rules of R

- By the lexical scoping, R searches for a free variable in the following order:
  1. The search starts in the environment in which the function was defined.
  2. If the value of a symbol is not found in the environment of (1), the search is continued in the parent environment of the environment of (1).
  3. The search continues in the sequence of parent environments until we hit the top-level environment (usually global environment or namespace of a package).
  4. If a value for a given symbol cannot be found until the top-level environment, then an error is occurred.

Lexical Scoping Rules of R

- Why are scoping rules important?
  - Typically, a function is defined in the global environment, so that the values of free variables are just found in the user's workspace $\Rightarrow$ No problem.
  - However, in R users can have functions defined inside of other functions (c.f., other languages like C do NOT allow this).
  - In this case, the environment in which a function is defined is the body of another function.

Examples of Lexical Scoping

```
> # Example 1
> x <- 5
> y <- 3
> f = function(x) x + y
> f(2)
[1] 5
```

- Free variable y found in the global environment.

```
> # Example2
> y <- 4
> f <- function(x)
+ {
+   y <- 7; g(x)
+ }
> g <- function(z)  y + z
> f(3)
[1] 7
```

- Free variable y found in the global environment.

Examples of Lexical Scoping

```
> # Example 3
> y <- 4
> f <- function(x)
+ {
+   y <- 7
+   g <- function(z)  y + z
+   g(x)
+ }
> f(3)
[1] 10
```

- Free variable `y` found in the function `f` environment.

```
> # Example 4
> f <- function(x) apply(iris[,1:4],2,mean) + x
> f(5)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
   10.843333     8.057333     8.758000     6.199333
```

- Free variable `iris` found in the search path.

Examples of Lexical Scoping

```
> # Example 5
> pow <- function(n)
+ {
+   po <- function(x) x^n
+   po
+ }

> cube <- pow(3)
> square <- pow(2)

> cube(4)
[1] 64
> square(4)
[1] 16
```

- The function pow returns another function po as its value.

Exploring Function Closures

- `environment(`*`function`*`)`: It returns the name of the environment of *function*.
- `ls(`*`environment`*`)`: It returns objects names in the *environment*.
- `get('object', environment)`: It returns the value of the *object* in the *environment*.

```
> pow <- function(n)
+ {
+   po <- function(x) x^n
+   po
+ }
> cube <- pow(3)
> square <- pow(2)
```

Exploring Function Closures

```
> environment(cube)
<environment: 0x11a41180>
> environment(square)
<environment: 0x11a40f50>
```

- Different calls of the same function produce physically different environments.

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

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

- `n` have different values because they are in different environments.

No Pointers in R

- Pointer: A variable that contains the address of a location in computer memory.
- R does NOT have pointers.
- E.g., suppose that we want to change values of `x & y` using a function `f()`.

```
x <- 1:10;   y <- 1:5
lxy <- list(x=x,y=y)
f <- function (lxy)
{
  x <- lxy$x + 5;  y <- lxy$y + 3;  lxy <- list(x=x, y=y)
  return(lxy)
}
x <- f(lxy)$x;   y <- f(lxy)$y
```

- If pointers are available in R, the values of `x & y` can be changed inside of `f()`.
- Even if R does not have pointers, R still can handle these situations. However, it makes codes more syntactically complex and harder to read.

Writing Upstairs

- Code at a certain level of the environment can read all the variables at the levels above it. However, it cannot write variables at higher levels of environment via '=' or '<-'.
- Writing upstairs:
  - `<<-`: superassignment operator.
  - `assign()`.

```
> f <- function(k)
+ {
+   k <<- k + 3
+   y <- 2 * z
+ }
> w <- 2
> z <- 4
> k
Error: object 'k' not found
```

Writing Upstairs

```
> f(w)
> w
[1] 2
> z
[1] 4
> k
[1] 5

> f <- function(k)
+ {
+   assign('k', k+3, pos=.GlobalEnv)
+   z <- 2 * z
+ }
> f(w)
> w
[1] 2
> k
[1] 5
```