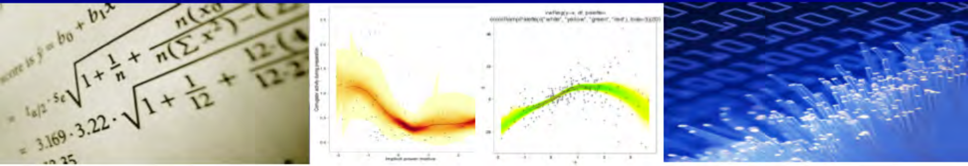


Efficient R programming

the `rolygon` example

Torino April 2013

Andrea Spanò

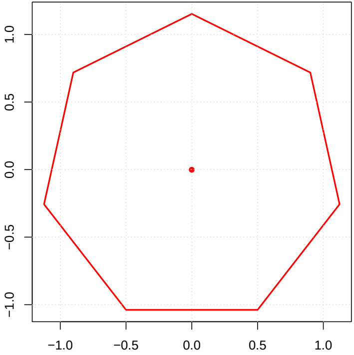
- ⋆ This brief tutorial illustrates how to combine S4 object oriented capabilities with function closures in order to develop classes with built in methods.
- ⋆ In practice, we want to write highly reusable code in order to increase development and maintenance efficiency
- ⋆ Finally, a great thank to *Hadley Wickham* for the great contribution of material and tutorials made available on the web and to *Bill Venables* and *Stefano Iacus* for their kind support.

Table of Contents

As Wikipedia states: *In Euclidean geometry, a regular polygon is a polygon that is equiangular (all angles are equal in measure) and equilateral (all sides have the same length)*. Square, pentagon, hexagon are regular polygons.

Within R we would like to have simple functions like:

```
> e1 <- heptagon(s = 1)
> plot(e1)
```



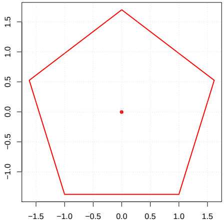
Previous call is the result of a simple S4 methods and classes implementation:

```
> setClass("heptagon", representation(s = "numeric"))

> heptagon <-  function(s){new("heptagon", s=s)}

> setMethod(f = "plot", signature = "heptagon",
          definition = function(x, y){
            object <-  x
            s <-  object@s
            n <- 7
            pi <- base::pi
            rho <-  (2*pi)/n
            h <-  .5*s*tan((pi/2)-(pi/n))
            r <-  sqrt(h^2+(s/2)^2)
            sRho <-  ifelse(n %% 2 == 0, (pi/2- rho/2), pi/2)
            cumRho <-  cumsum(c(sRho, rep(rho, n)))
            cumRho <-  ifelse(cumRho > 2*pi, cumRho-2*pi, cumRho)
            x <-  r*cos(cumRho)
            y <-  r*sin(cumRho)
            par(pty = "s")
            plot(x, y, type = "n", xlab = "", ylab = "")
            lines(x, y, col = "red", lwd = 2)
            points(0, 0, pch = 16, col = "red")
            grid()
            invisible(NULL)
          }
)
```

With this mind set we would have to define a new class, function and method for each polygon ...

```
> setClass("penthagon", representation(s = "numeric"))

> penthagon <-  function(s){new("penthagon", s=s)}

> setMethod(f = "plot", signature = "penthagon",
          definition = function(x, y){
            object <-  x
            s <-  object@s
            n <- 5
            ....
            invisible(NULL)
          }
)
```

```
p1 <- penthagon(s=1)
plot(p1)
```



This could become a quite boring job. Moreover, despite the number of cases we may take into account, we are pretty much sure that sooner or later we will need something more ... a hendecagon (11 sides) or a enneadecagon (19 sides).

We could accept some compromises and write a generic schema:

```
> setClass("rolygon",
representation(n = "numeric", s = "numeric"))

> rolygon = function(n, s){new("rolygon", n= n, s=s)}

> setMethod(f = "plot", signature = "rolygon",
definition = function(x, y){
object = x
s = object@s
n = object@n
...
invisible(NULL)
}
)

> h11 = rolygon(n = 11, s = 1)

> plot(h11)
```

but this is not what we wanted .... we do want:

```
> h11 = hendecagon(s = 1)
> plot(h11)
```

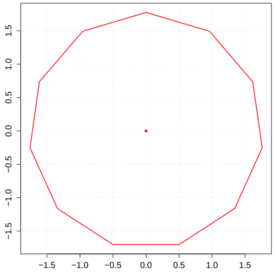


Table of Contents

- Any time a function is called, a new environment is created, whose enclosure is the environment where the function is defined. The computation, as expressed by the body of the function, occurs in the newly created environment.
- Thus, whenever we call a function we have at least two environments: the environment the function was defined in and the environment where the function evaluation takes place.
- By using this idea, we can define a function `f()` that returns a function `g()`.
- As `g()` is created within the evaluation environment of `f()`, this last environment is the enclosure of `g()`. Therefore, `g()` remembers all symbols bound to that environment.

As a practical application of this idea let's consider a function `f()` that returns a function `g()`:

```
> f <-  function(x) {
   g = function(y){x+y}
   g
}
```

As `g()` is created within the evaluation environment of `f()`, `g()` *"remembers"* the value of `x`.

Therefore we can define a simple function `f1()` that adds one to the given `y` argument

```
> f1 <- f(x = 1) ; f1(y = 3)
[1] 4
```

Note that f1() remembers the value of x

The environment of f1() can be directly accessed and manipulated:

```
> ls(env=environment(f1))
[1] "g" "x"
> get("x", env=environment(f1))
[1] 1

> environment(f1)$x <- 0
> f1(1)
[1] 1
```

The same exercise apply to any `fx()` such as `f99`

```
> f99 <- f(99)
> f99(y = 1)
[1] 100
```

An other example consists of a simple `estimate()` that generate specific `l()` functions for maximum likelihood estimates

```
estimate = function(dist, theta){
  estimate = function(x){
    neglik = function(theta = theta , x = x, log = T){
      args = c(list(x), as.list(theta), as.list(log))
      neglik = -sum(do.call(dist,  args))
      neglik
    }
    optim(par = theta, fn = neglik , x = x)
  }
estimate}
```

Once we have `estimate()`, we can use it to define any `l()` function as long as its `d()` exists

That is, we can now write a `lnorm()` that computes mle estimate as simply as:

```
lnorm = estimate("dnorm", theta = c(mean(x), sd(x))
```

`lnorm()` is a new function that can be used as:

```
x = rnorm(100, 7 , 2)
lnorm(x)$par
[1] 6.803764 1.704783
```

Similarly, for a poison distribution:

```
lpois = estimate("dpois", theta = c(mean(x)))
events = rpois(1000, lambda = 22)
lpois(events)$par
[1] 22.02248
```

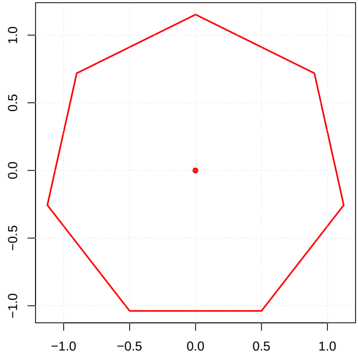
Table of Contents

- ⋆ The combination of the two previous ideas allows quite interesting coding techniques.
- ⋆ We define a `rolygon()` function that returns a generic `f()` capable of generating specific regular polygons with plot method inherited from rolygon's environment:

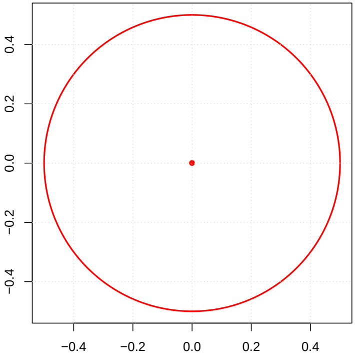
```
> rolygon <- function(n) {

    # Define rolygon class
    setClass("rolygon",
     representation(n = "numeric", s = "numeric"))

    # Define a plot method for
    #object of class rolygon
    setMethod(f = "plot", signature = "rolygon",
  definition = function(x, y){
          object <-  x
          s <-  object@s
          n <- object@n
          ...
          invisible(NULL)
      })

    # Define a function that returns an object
    # of class rolygon
    f <- function(s){new("rolygon", n = n, s = s)}

    # Return the newly created function
    return(f)
}
```

Now, we can easily define any polygon we need with no extra coding

```
heptagon <- rolygon(n = 7)
e1 <- heptagon(1)
plot(e1)
```




```
circumference <- rolygon(n = 10^4)
plot(circumference(s = base::pi/10^4))
```



Andrea Spanò

Quantide

Tel: (+39) 347 747 04 92

Mail: andrea.spano@quantide.com

Web: www.quantide.com