Functions and Programming

Fritz Scholz

April 2, 2015
Functions can execute any number of commands within \{ and \}

```r
myfun <- function(x,y,z){
  ...
  commands
  ...
}
```

The birthday problem asks what is the chance that in a random group of \( n \) people you have at least 2 with same birthday. Assume a \( N = 365 \) day year, all days equally likely per person. It is easier to get the complementary probability of

\[
P(\text{all birthdays are distinct}) = \frac{N(N-1)\ldots(N-n+1)}{N^n} = \frac{N!}{N^n(N-n)!}
\]

Use Stirling’s approximation \( N! \approx \sqrt{2\pi N}(N/e)^N \).
The Desired Function

Bday <- function(N,n) {
  p.exact <- prod((N-(0:(n-1)))/N)
  p.Stirling <- exp((N-n+.5)*log(N/(N-n))-n)
  out <- c(p.exact,p.Stirling)
  names(out) <- c("exact p","Stirling p")
  out
}

> Bday(365,23)
  exact p  Stirling p
  0.4927028 0.4927103

> Bday(10000000000,100000) # 10 billion, 100K
  exact p  Stirling p
  0.6065327 0.6065325

> 100000/1e10
[1] 1e-05
- Sometimes functions will do different things depending on circumstances, i.e., based on conditional tests.
- A continuous, monotone function $\text{Fun}$ crosses zero at some point $x_0$, i.e., $\text{Fun}(x_0) = 0$.
- Bracket this root, i.e., find $a$ and $b$ such that $a \leq x_0 \leq b$.
- Such bracketing is needed by a root finder like \texttt{uniroot}.
- Start out with $a < b$ and evaluate the function there.
- If the function values have opposite sign, you are done.
- Otherwise shift and lengthen the interval in the appropriate direction, by a multiple of $b - a$.
- Keep doubling this multiple for each further iteration step.
bracket <- function(Fun,a,b,dir="up"){  
dir <- match.arg(dir,c("up","down"))  
# limits monotonicity choice for Fun  
if(b <= a) stop("need a < b\n")  
if(dir == "down") fun <- function(x){-Fun(x)}  
# fun <- -Fun does not work  
# makes fun monotone increasing  
if(dir == "up") fun <- Fun  
fa <- fun(a)  
fb <- fun(b)  
if(fa > fb)  
stop("\nmonotonicity of Fun does not agree with dir\n")  
delta <- b-a  
while(fa*fb > 0){
if(fb < 0){
    a <- b
    b <- b+delta
    fa <- fb
    fb <- fun(b)
}
if(fa > 0){
    b <- a
    a <- a-delta
    fb <- fa
    fa <- fun(a)
}
delta <- 2*delta
} # end of while
# bracketing values are found
c(a,b)
Comments on \texttt{while} Loop

- The structure of the \texttt{while} construct is as follows.

\begin{verbatim}
while(logic evaluation) {
    ....# a sequence of commands to carry out
    ....# as long as the logic evaluation
    ....# results in TRUE
    ....# If evaluation results in FALSE,
    ....# proceed after } of while loop.
}
# the logic evaluation should give you a
# single T/F, not a vector of such values
\end{verbatim}

- Make sure that your \texttt{while} loop has a chance to end.
- If stuck in an infinite loop, terminate the R session.
  - That works in RGui or RStudio.
  - In the Linux interface you can try Ctrl C.
The structure of the if construct is as follows.

```plaintext
if(logic evaluation){
    ....# a sequence of commands to carry out
    ....# when the logic evaluation is TRUE.
    ....# Otherwise ignore the commands within
    ....# { & }
}
# the logic evaluation should give you a
# single T/F, not a vector of such values
```
Multiple Choices

```r
if(logic evaluation){
    ....# if TRUE do this
} else{
    ....# otherwise do this
}

if(logic evaluation1){
    ....# if this is TRUE do this
} else if(logic evaluation2){
    ....# if this is TRUE do this
} else{
    ....# otherwise do this
}
```

- The above `else if` chain can be extended.
- See also `ifelse`, operates on a logic vector.
The for Loop Construct

```c
for( i in x){
    ... # do something that may
    ... # or may not involve i
}
```

- *i* traverses through all elements of the vector *x* as the loop iterates.
- Commands in loop are carried out \( n = \text{length}(x) \) times.
- Recall that looping is not efficient (for large *n*).
- For each loop iteration the code within it is reinterpreted.
A for Loop Example

```r
forLoop <- function(x,n,Nsim) {
  xmean <- numeric(Nsim)
  for(i in 1:Nsim) {
    y <- sample(x,n,replace=TRUE)
    xmean[i] <- mean(y)
    # computes mean of y, assigns it to xmean[i]
  }
  hist(xmean,xlab="sample mean",
       main="sampling distribution",
       nclass=100,col=c("blue","orange"))
}
```

- `sample(x,n,replace=TRUE)` randomly selects `n` items from vector `x` with replacement.
- `hist(x,...)` makes a histogram of `x`, see `?hist`
```r
> forLoop(c(1:10, 100), 5, 10000)
```

![Sampling distribution chart](chart.png)

- **Sampling Distribution**
- **Sample Mean**
- **Frequency**
- 0 20 40 60 80
- 0 500 1000 1500
> forLoop(c(1:10, 100), 50, 10000)
forLoop(c(1:10,100),1000,10000)

sampling distribution

sample mean

Frequency

11 12 13 14 15 16 17

0 50 100 150 200 250

sample mean
> forLoop(c(1:10,11), 5, 10000)

![Sampling distribution]

**Sampling Distribution**

- **Sample Mean**: 7
- **Frequency**: 10000 samples

The histogram shows the frequency distribution of the sample mean for 10000 iterations of the forLoop function, generating data from a sequence of 1 to 11.
Try to match bracket positions, for readability.
Add comments, for others and for yourself.
What happens within a function stays there.
The external workspace is not polluted by temporary objects.
That is one reason I prefer functions over sourcing code, which can leave quite a debris field behind.
In the R Gui the command fix(myfun) opens a text editor which shows a function skeleton.
After modifying the arguments and the function body and exiting (with saving) the editor, your work space contains a function object named myfun.
The browser() command at strategic places inside a function body stops execution and lets you examine objects there, with ls(), objects(), str(...), etc.
Good for debugging. There are other debugging tools.
debug and undebug

myfun <- function(i,n){ rnorm(n[i]) }
> debug(myfun) # puts myfun into debug mode
> myfun(2,4:2)
debugging in: myfun(2, 4:2)
debug at <tmp>#1: {
    rnorm(n[i])
}
Browse[2]> ls()
[1] "i" "n"
Browse[2]> i
[1] 2
Browse[2]> I(n) # n alone has special meaning
[1] 4 3 2
Browse[2]> debug at <tmp>#2: rnorm(n[i])
Browse[2]>
exiting from: myfun(2, 4:2)
[1] 0.368 0.491 1.092
undebug(myfun) # takes myfun out of debug mode
The Arguments in a Function, Calling Sequence

```r
> myfun <- function (x, y=0, z) {
    x + 2 * y + 3 * z
}>
> myfun(1, 2, 3)
[1] 14
> myfun(1, 3)
Error in myfun(1, 3) : argument "z" is missing, with no default

> myfun(z=3, x=1, y=2)
[1] 14
> myfun(z=3, x=1)
[1] 10

You can change the order of input arguments by naming the arguments, e.g., z=3, x=1.
The bracket function called another function Fun.

What if Fun has other arguments beyond the root argument?

What if those other arguments change with Fun?

We don’t want to rewrite bracket each time.

For that we have the dots ( . . . ) argument construct.

First we illustrate this with an example.
A Function with ... Argument

Typically ... goes at the end of argument list.

\[
\text{prob} \leftarrow \text{function}(x, fx, \ldots) \{ fx(x, \ldots) \}
\]

\[
> \text{prob}(4, \text{pbinom}, 10, .5) \\
> \text{pbinom}(4, 10, .5) \\
> \text{prob}(4, \text{ppois}, \text{lambda}=10) \\
> \text{ppois}(4, 10)
\]
What Happens Here?

> prob(4,ppois,10,.5)
[1] 0.9707473

> prob(4,ppois,10,1)
[1] 0.0292527

> prob(4,ppois,10,.999)
[1] 0.9707473

> prob(4,ppois,10,1.001)
[1] 0.0292527

> args(ppois)
function (q, lambda, lower.tail = TRUE, log.p = FALSE)
Treatment of `lower.tail` in `ppois`

```r
> args(ppois)
function (q, lambda, lower.tail = TRUE,
        log.p = FALSE)
> ppois(4, 10)  # P(X <= 4)
[1] 0.02925269
> ppois(4, 10, 1)
[1] 0.02925269
> ppois(4, 10, .9)
[1] 0.9707473  # P(X > 4)=1−P(X <= 4)
> ppois(4, 10, 7)
[1] 0.02925269
> ppois(4, 10, 0)
[1] 0.9707473
> 1-0.02925269
[1] 0.9707473
```

**Numerical arguments for** `lower.tail` **are set to** `TRUE` **if** $\geq 1$, **and set to** `FALSE` **when** $< 1$. 
Some Comments on . . .

- View . . . as a way to pass arguments through.
- It is best to use named arguments, e.g., \texttt{lambda=10}.
- Any values in place of . . . are passed through.
- The inside reference to . . . may not make use of unused named arguments.
- Named arguments can be abbreviated, as long as there is no ambiguity.
- Always test your usage of . . . on examples. Do you get what you want?
R has several functions to perform looping operations.

apply(X, MARGIN, FUN, ...) returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

lapply(X, FUN, ...) returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.

tapply(X, INDEX, FUN, ..., simplify = TRUE) Apply a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors. INDEX is a list of one or more factors each of same length as length(X).

There are several more apply type functions.

Compare their speed w.r.t. ordinary looping.
Example of `apply`

```r
> M <- matrix(1:12, ncol=2, byrow=F)
> M
 [,1] [,2]
[1,]  1  7
[2,]  2  8
[3,]  3  9
[4,]  4 10
[5,]  5 11
[6,]  6 12

> apply(M, MARGIN=1, FUN=sum)
[1] 8 10 12 14 16 18

> apply(M, MARGIN=2, FUN=sum)
[1] 21 57
```
Example of `apply` with List Output

```r
> mat <- matrix(c(1,2,3,3,2,2,1,2,1,2,3,4), byrow=T, ncol=4)
> mat

[1,] 1  2  3  3
[2,] 2  2  1  2
[3,] 1  2  3  4

> apply(mat, 1, unique)
[[1]]
[1] 1 2 3

[[2]]
[1] 2 1

[[3]]
[1] 1 2 3 4
```
Example of \texttt{lapply}

\begin{verbatim}
> myfun <- function(i,n){
    rnorm(n[i])
}

> options(digits=3)

> lapply(1:3,FUN=myfun,n=c(3,4,5))
# n=c(3,4,5) in place of ...
[[1]]
[1] 0.341 0.115 0.442

[[2]]
[1] -1.053 -0.929 -0.121 0.078

[[3]]
[1] 0.976 -2.501 1.754 0.658 -0.158
\end{verbatim}
Example of `tapply`

```r
> n <- 8
> fac <- factor(rep(1:3, length = n), levels = 1:5)
> fac
[1] 1 2 3 1 2 3 1 2
Levels: 1 2 3 4 5
> table(fac)
fac
1 2 3 4 5
3 3 2 0 0
> tapply(1:n, fac, sum)
 1 2 3 4 5
12 15 9 NA NA
> 1:8
[1] 1 2 3 4 5 6 7 8
> as.numeric(fac)
[1] 1 2 3 1 2 3 1 2
```

See documentation of `apply`, `lapply`, `tapply` for more.