

Stat 302  
Statistical Software and Its Applications  
Functions and Programming

Yen-Chi Chen

Department of Statistics, University of Washington

Autumn 2016

# The for Loop Construct

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

- Commands in loop are carried out  $n = \text{length}(x)$  times and the variable  $i$  will take each value in the vector  $x$ .
- Recall that looping is not efficient, each iteration is interpreted.

## for Loop Example 1

```
> x <- 0
> for(i in 1:10){
+   x <- c(x,i)
+ }
> x
 [1]  0  1  2  3  4  5  6  7  8  9 10
> x_seq <- c(3,1000,-30,-99,+100)
> for(i in x_seq){
+   x <- c(x,i)
+ }
> x
 [1]      0      1      2      3      4      5      6      7
 8      9     10      3 1000    -30
[15]   -99    100
```

## for Loop Example 2

```
> y <- 1
> for(i in 1:10){
+   y <- y*i
+ }
> y
[1] 3628800
>
> factorial(10)
[1] 3628800
```

## for Loop Example 3

```
> A <- c(1,1,1)
> for(j in 1:10){
+   A = cbind(A, c(j, j^2, 5*j))
+ }
> A
```

A

[1,]	1	1	2	3	4	5	6	7	8	9	10
[2,]	1	1	4	9	16	25	36	49	64	81	100
[3,]	1	5	10	15	20	25	30	35	40	45	50

# Comments on while Loop

- The structure of the while construct is as follows.

```
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.  
}
```

- Make sure that your 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.

## while Loop Example

```
> x <- 5
> y <- x
> while(x<20){
+   x <- x+4
+   y <- c(y,x)
+ }
> x
[1] 21
> y
[1] 5 9 13 17 21
```

Functions can execute any number of commands within { and }

```
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

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

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)
  exact p  Stirling p
0.6065327  0.6065325
```

## Functions with Conditionals: `if`

```
> myfun <- function(x){
+   if(is.matrix(x)){
+     x^2
+   }
+ }
> a <- 1:10
> B <- matrix(a, nrow=2)
>
> myfun(a)
> # no output--because 'a' is a vector
> myfun(B)
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    9   25   49   81
[2,]    4   16   36   64  100
```

What would happen if we try `myfun(as.matrix(a))`?

- The structure of the `if` construct is as follows.

```
if(logic evaluation){  
    ....# a sequence of commands to carry out  
    ....# when the logic evaluation is TRUE.  
    ....# Otherwise ignore the commands within  
    ....# {    &    }  
}
```

# Multiple Choices

```
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.

## Multiple Choices: Example 1-1

```
> myfun2 <- function(x){  
+   if(is.matrix(x)){  
+     x^2  
+   }else if(is.list(x)){  
+     length(x)  
+   }else{  
+     print("I am a good student!")  
+   }  
+ }  
> L = list(a,B)
```

## Multiple Choices: Example 1-2

```
> myfun2(a)
[1] "I am a good student!"
> myfun2(B)
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    9   25   49   81
[2,]    4   16   36   64  100
> myfun2(L)
[1] 2
```

## Multiple Choices: multiple if — 1

```
> myfun3 <- function(y) {  
+   if(is.matrix(y)) {  
+     y^2  
+   }  
+   if(is.matrix(y)) {  
+     2*y  
+   }  
+ }  
>  
> myfun3(B)  
      [,1] [,2] [,3] [,4] [,5]  
[1,]    2    6   10   14   18  
[2,]    4    8   12   16   20
```

Only displays the later calculation.

## Multiple Choices: multiple if — 2

```
> myfun4 <- function(y) {
+   if(is.matrix(y)) {
+     print(y^2)
+   }
+   if(is.matrix(y)) {
+     print(2*y)
+   }
+ }
>
> myfun4(B)
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    9   25   49   81
[2,]    4   16   36   64  100
      [,1] [,2] [,3] [,4] [,5]
[1,]    2    6   10   14   18
[2,]    4    8   12   16   20
> # show both cases!
```



## Multiple Choices: multiple if — 3

```
> myfun5 <- function(y) {  
+   if(is.matrix(y)) {  
+     y <- y+1  
+     y^2  
+   } else if(is.matrix(y)) {  
+     2*y  
+   }  
+ }  
  
> myfun6 <- function(y) {  
+   if(is.matrix(y)) {  
+     y <- y+1  
+     y^2  
+   }  
+   if(is.matrix(y)) {  
+     2*y  
+   }  
+ }
```

## Multiple Choices: multiple if — 4

```
> myfun5(B)
      [,1] [,2] [,3] [,4] [,5]
[1,]     4    16    36    64   100
[2,]     9    25    49    81   121
> myfun6(B)
      [,1] [,2] [,3] [,4] [,5]
[1,]     4     8    12    16    20
[2,]     6    10    14    18    22
> 2*(B+1)
      [,1] [,2] [,3] [,4] [,5]
[1,]     4     8    12    16    20
[2,]     6    10    14    18    22
> # so both statements are executed
> # in the two 'if' cases.
```

- 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.

## A Function of a function: using ... Argument

```
prob <- function(x, fx, ...) {fx(x, ...)}
```

```
> prob(4, pbinom, 10, .5)
```

```
# = prob(4, pbinom, size=10, prob=.5)
```

```
[1] 0.3769531
```

```
> pbinom(4, 10, .5) # = pbinom(4, size=10, prob=.5)
```

```
[1] 0.3769531
```

```
> prob(4, ppois, lambda=10) # = prob(4, ppois, 10)
```

```
[1] 0.02925269
```

```
> ppois(4, 10) # = ppois(4, lambda=10)
```

```
[1] 0.02925269
```

## The ... Argument

- The `prob` function called another function `fx`.
- What if `fx` has other arguments beyond the root argument?
- What if those other arguments change with `fx`?
- We don't want to rewrite `prob` each time.
- We can use the dots (`...`) to handle this.
- Typically `...` goes at the end of argument list.

## What Happens Here?

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

```
> prob(4, ppois, 10, 1)
[1] 0.02925269
```

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

```
> prob(4, ppois, 10, 1.001)
[1] 0.02925269
```

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

`prob` treats the 4-th argument as `lower.tail`, inconsistently.

## Some Comments on ...

- View ... as a way to pass arguments through.
- It is best to use named arguments, e.g., `lambda=10`.
- Any values in place of ... are passed through.
- The inside reference to ... may not make use of unused named arguments.
- Always test your usage of ... on examples.  
Do you get what you want?

Enter the following:

```
testfun <- function(x,y) {  
  if(x>5&y>5) {  
    print("A")  
  }else if(x<0|y<0) {  
    print("B")  
  }else{  
    print("C")  
  }  
}
```

and try `testfun(1,10)`, `testfun(-1,10)`, and `testfun(10,10)`.

What do `|` and `&` do? You may try `"&"` and `"|"`.  
They are *or* and *and* in logical operation.