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)
  exact p  Stirling p
   0.6065327 0.6065325
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 `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 $b-a$. 
bracket <- function(Fun,a,b,dir="up"){  
dir <- match.arg(dir,c("up","down"))  
# limits monotonicity choice for Fun  
if(b <= a) stop("\nneed 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("\nmontonicity of Fun\ndoes 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))
The structure of the `while` construct is as follows.

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

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

- Commands in loop are carried out \( n = \text{length}(x) \) times.
- Recall that looping is not efficient, each iteration is interpreted.
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`
> forLoop(c(1:10,100), 5, 10000)

![Sampling Distribution](image)

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

![Sampling distribution](image-url)
\texttt{> forLoop(c(1:10, 100), 1000, 10000)}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{sampling_distribution.png}
\caption{Sampling distribution}
\end{figure}
forLoop(c(1:10, 11), 5, 10000)

sampling distribution

sample mean

Frequency

2 4 6 8 10

0 100 200 300 400 500 600
Comments on Functions

- 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.
The ... Argument

- 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.

```r
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
```
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?