Basic Programming

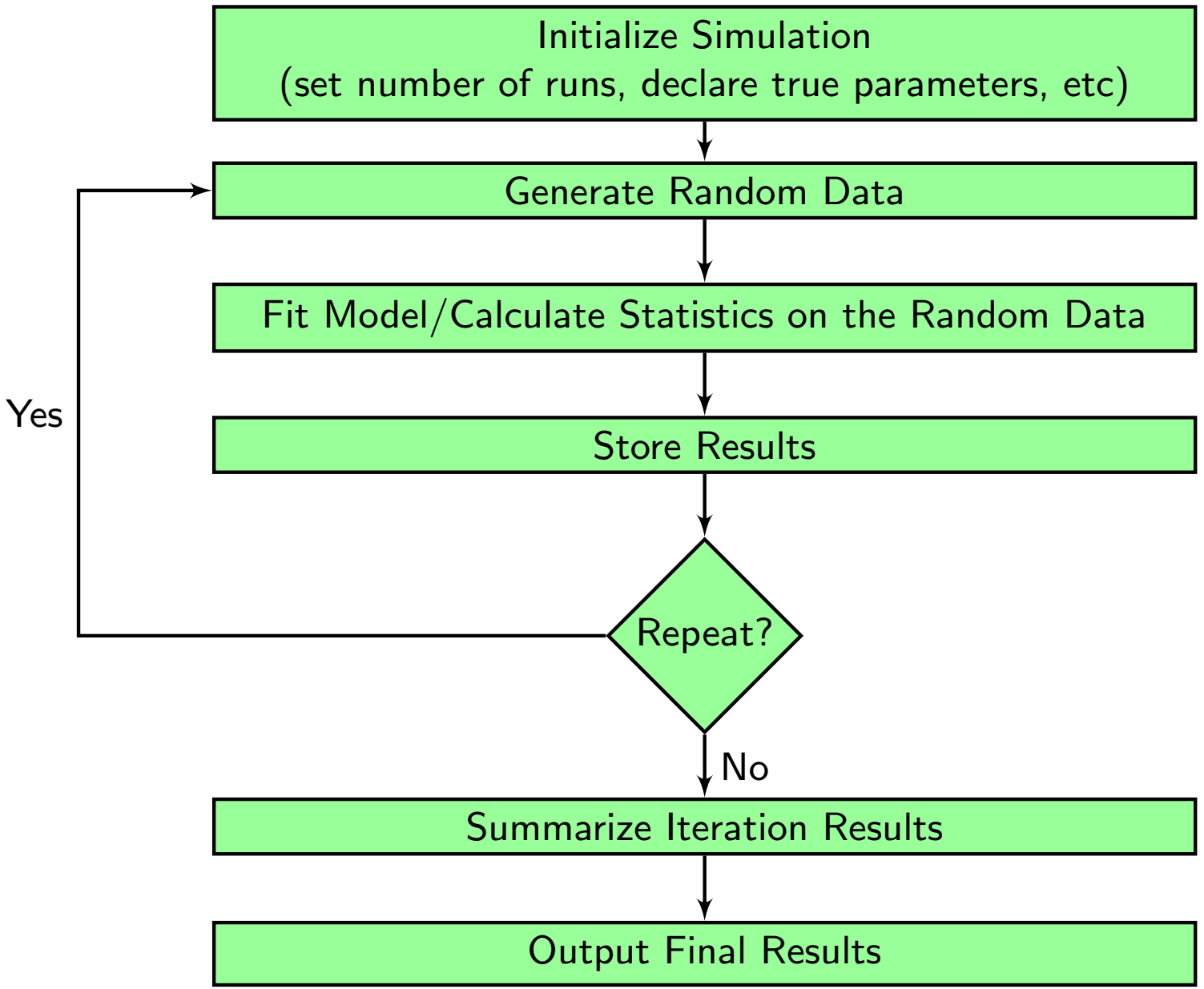
Lecture 3

Nicholas Christian

BIOST 2094 Spring 2011

Outline

Simulation


Initialize Simulation
(set number of runs, declare true parameters, etc)
Generate Random Data
Fit Model/Calculate Statistics on the Random Data
Store Results
Repeat?
Yes
No
Summarize Iteration Results
Output Final Results

Generating Random Data

- R has several functions for generating random data,
  - Built-in statistical distribution functions
  - `sample()` function
- Other techniques (probability integral transformation, adaptive rejection sampling) will be discussed in lecture 7.

```
sample(x, size, replace = FALSE, prob = NULL)
```

- The `sample()` function takes a sample of size `size` from a vector `x` either with or without replacement. Optionally a vector of probabilities for obtaining the elements of `x`, `prob`, can be supplied.
- If `replace` is `FALSE`, the probabilities are updated after each sample is drawn, that is the probability of choosing the next item is proportional to the weights amongst the remaining items.

```
> sample(1:5, 3)
[1] 5 2 3
> sample(1:5, 10, replace=TRUE)
 [1] 5 3 4 2 2 4 2 4 1 2
> sample(1:5, 10, replace=TRUE, prob=c(.6,.2,.1,.05,.05))
 [1] 1 1 1 2 3 2 2 1 1 3
```

Statistical Distributions

- R has several built-in statistical distributions. For each distribution four functions are available,
  - `r` Random number generator
  - `d` Density function
  - `p` Cumulative distribution function
  - `q` Quantile function
- Each letter can be added as a prefix to any of the R distribution names.

R	**Distribution**	R	**Distribution**
`beta`	Beta	`logis`	Logistic
`binom`	Binomial	`nbinom`	Negative binomial
`cauchy`	Cauchy	`norm`	Normal
`exp`	Exponential	`pois`	Poisson
`chisq`	Chi-Square	`signrank`	Wilcoxon signed rank statistic
`f`	Fisher's F	`t`	Student's t
`gamma`	Gamma	`unif`	Uniform
`geom`	Geometric	`weibull`	Weibull
`hyper`	Hypergeometric	`wilcox`	Wilcoxon rank sum
`lnorm`	Lognormal		

- See the documentation for how each model is parameterized.

Example - Statistical Distributions

- Below are the density, cumulative distribution, quantile and random number generator functions for the standard normal distribution.

```
> dnorm(1.96, mean=0, sd=1)                    # Density
[1] 0.05844094
> pnorm(1.96, mean=0, sd=1)                    # Distribution (lower tail)
[1] 0.9750021
> pnorm(1.96, mean=0, sd=1, lower.tail=FALSE) # Distribution (upper tail)
[1] 0.02499790
> qnorm(0.975, mean=0, sd=1)                   # Quantile
[1] 1.959964
> rnorm(5, mean=0, sd=1)                       # Random Number
[1]  1.3751132  0.9807624  0.4229843 -0.3813931  1.0146115
```

Special Statistical Functions

- `beta(a,b)`

$$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

- `gamma(x)`

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\, dt$$

- `digamma(x)`

$$\psi(x) = \frac{d}{dx} \log(\Gamma(x)) = \frac{\Gamma'(x)}{\Gamma(x)}$$

- `choose(n, k)`

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

Repeatable Simulations

- For a simulation to be repeatable we need to specify the type of random number generator and the initial state of the generator.
- `R` has several kinds of generators, see `RNGkind()`
- The simplest way to specify the initial state or seed is to use,

  `set.seed(seed)`

  - The argument `seed` is a single integer value
  - Different `seeds` give different pseudo-random values
  - Calling `set.seed()` with the same `seed` produces the same results, if the sequence of calls is repeated exactly.
- If a `seed` is not specified then the random number generator is initialized using the time of day.

Example - Repeatable Simulations

```
> set.seed(17632)
> runif(5)
[1] 0.03288684 0.88861821 0.21466744 0.32907126 0.78222193
> rnorm(5)
[1]  0.7201864 -0.2601566  0.5719481  1.1099988 -0.2563202

> set.seed(89432)
> runif(5)
[1] 0.175990418 0.258567422 0.007370616 0.286017248 0.601586852

> set.seed(17632)
> runif(5)
[1] 0.03288684 0.88861821 0.21466744 0.32907126 0.78222193
> rnorm(5)
[1]  0.7201864 -0.2601566  0.5719481  1.1099988 -0.2563202

> set.seed(17632)
> rnorm(5)
[1] -1.8399628 -0.7903303  0.7797193 -0.6107181 -0.2786504
> runif(5)
[1] 0.2493954 0.8665002 0.3426808 0.3988518 0.7198731
```

Control Structures

- `if()...else` Determine which set of expressions to run depending on whether or not a condition is `TRUE`
- `ifelse()` Do something to each element in a data structure depending on whether or not a condition is `TRUE` for that particular element
- `switch()` Evaluate different expressions depending on a given value

- `for()` Loop for a fixed number of iterations
- `while()` Loop until a condition is `FALSE`
- `repeat` Repeat until iterations are halted with a call to `break`
- `break` Break out of a loop
- `next` Stop processing the current iteration and advance the looping index

- See `?Control` for the `R` documentation on control structures.

if()...else

- The *condition* needs to evaluate to a single logical value.
- Brackets { } are not necessary if you only have one expression and/or the `if()...else` statement are on one line.
- To avoid a syntax error, you should **NOT** have a newline between the closing bracket of the `if` statement and the `else` statement

`if(`*condition*`)` *expression if TRUE*

```
if(condition) {
    expressions if TRUE
}
```

`if(`*condition*`)`*expression if TRUE* `else` *expression if FALSE*

```
if(condition) {
    expressions if TRUE
} else {
    expressions if FALSE
}
```

Example - `if()...else`

- Calculate the median of a random sample $X_1 \ldots X_n$,

$$\text{median}(X) = \begin{cases} \frac{1}{2}X_{(\frac{n}{2})} + \frac{1}{2}X_{(1+\frac{n}{2})} & \text{if } n \text{ is even} \\ X_{(\frac{n+1}{2})} & \text{if } n \text{ is odd} \end{cases}$$

where $X_{(1)} \ldots X_{(n)}$ denote order statistics.

```
x <- c(5,4,2,8,9,10)

n <- length(x)
sort.x <- sort(x)
if(n%%2==0) {
   median <- (sort.x[n/2]+sort.x[1+n/2])/2
} else median <- sort.x[(n+1)/2]

median
```

ifelse()

- `ifelse()` returns a value with the same structure as `test` which is filled with elements selected from either `yes` or `no` depending on whether the element of `test` is `TRUE` or `FALSE`.

```
ifelse(test, yes, no)
```

```
> (x <- seq(0,2,len=6))
[1] 0.0 0.4 0.8 1.2 1.6 2.0
> ifelse(x <= 1, "small", "big")
[1] "small" "small" "small" "big"   "big"   "big"

> (y <-  matrix(1:8, nrow=2))
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> ifelse(y>3 & y <7, 1, 0)
     [,1] [,2] [,3] [,4]
[1,]    0    0    1    0
[2,]    0    1    1    0
```

switch()

- The `switch()` function evaluates and returns different expressions depending on the value of `EXPR`

```
switch(EXPR, ...)
```

- `EXPR` needs to be a single character string or a number, not a vector
- `...` is a comma separated list of *name=expression* or *number=expression*
- If there is an exact match between `EXPR` and *name/number* then that expression is evaluated and returned. If there is no expression, then the next non-missing expression is used.
- If there is no match between `EXPR` and *name/number*, the value of the first unnamed expression in `...` is returned (unnamed expressions are like a default).
- You do not need to put `name` in quotes inside `switch()`, but `EXPR` will need quotes.

Example - switch()

- Calculate different measures of central tendencies for a vector y
- Geometric mean of $(y_1, \ldots y_n) = \left(\prod_{i=1}^{n} y_i\right)^{1/n}$

```
central <- function(y, measure) {
      switch(measure,
             Mean = ,
             mean = mean(y),
             median = median(y),
             geometric = prod(y)^(1/length(y)),
             "Invalid Measure")
}
> y <- runif(100)
> central(y, "mean")
[1] 0.5101253
> central(y, "Mean")
[1] 0.5101253
> central(y, "Median")
[1] "Invalid Measure"
> central(y, "geometric")
[1] 0.3949147
```

`for` loops

```
for(var in seq) {
    expressions
}
```

- `var` is the name of the loop variable that changes with each iteration
- `seq` is an expression evaluating to any type of vector
- With each iteration `var` takes on the next value in `seq`. At the end of the loop `var` will equal the last value of `seq`.
- The number of iterations equals the length of `seq`
- The { } are only necessary if there is more than one expression
- By default `R` buffers the output created during any loop. To print the output with each iteration, include `flush.console()` at the end of the loop or right click on the console and uncheck: Buffered output.

```
# Calculate 10! using a for loop
f <- 1
for(i in 1:10) {
    f <- f*i
    cat(i, f, "\n")
}
f
factorial(10)
```

`while` loops

```
while(cond) {
      expressions
}
```

- `cond` is a single logical value that is not `NA`
- The { } are only necessary if there is more than one expression
- If you are going to use a `while` loop, you need to have an indicator variable `i` and change its value within each iteration. Otherwise you will have an infinite loop.

```
# Calculate 10! using a while loop
i <- 10
f <- 1
while(i>1) {
      f <- i*f
      i <- i-1
      cat(i, f, "\n")
}
f
factorial(10)
```

`repeat` loops

```
repeat {
 expressions
 if(cond) break
}
```

- The `repeat` loop does not contain a limit. Therefore it is necessary to include an `if` statement with the `break` command to make sure you do not have an infinite loop.

```
# Calculate 10! using a repeat loop
i <- 10
f <- 1
repeat {
    f <- i*f
    i <- i-1
    cat(i, f, "\n")
    if(i<1) break
}
f
factorial(10)
```

Avoiding Loops

- With `R` it is a good idea to try and avoid including loops in programs.
- Code that takes a "whole object" approach versus an iterative approach will often be faster in `R`. For example, `ifelse()` or `cumsum()`.
- The `apply()` family of functions does not reduce the number of function calls. So using `apply()` may not necessarily improve efficiency. However, `apply()` is still a great function for making code more transparent and compact.
- When developing code it may be easier to first write an algorithm using a loop. That's fine, but try and replace the loop with a more efficient expression after you have a draft program.
- Loops are still useful tools, such as with simulation studies.

Example - Avoiding Loops

- Consider an extreme example, suppose we want the sum of 10 million random standard normal numbers. The `sum()` function, which uses the whole vector, is noticeably faster then the `for` loop, which uses each element.

```
z <- rnorm(1E7)
system.time(sum(z))

start <- proc.time()
total=0
for(i in 1:length(z))
    total <- total + z[i]
proc.time()-start
```

Timing Expressions

- `system.time()` and `proc.time()` are used for timing `R` expressions. Both functions return a named vector of length 5, but only print 3 elements.
- These functions print the user time, system time, and the elapsed time. Usually elapsed time is the most useful. This is the real elapsed time since the process was started. User and system time refer to CPU time.
- `winProgressBar()` displays a progress bar during a long computation

```
n=100
startTime <- proc.time()[3]
pb <- winProgressBar(title="Progress Bar", min=0, max=n)
for(i in 1:n) {
    Sys.sleep(.1)   # Suspend execution of R expressions for .1 seconds

    setWinProgressBar(pb, i, title=paste(round((i/n)*100),"% Complete"))
}
Sys.sleep(.5)
close(pb)
elapseTime <- proc.time()[3]-startTime
cat("Elapsed Time:",floor(elapseTime/60),"min",elapseTime%%60,"sec \n")
```

Functions

- `R` functions are objects that evaluate multiple expressions using arguments that are passed to them. Typically an object is returned.
- To declare a function,

  function name `<-` `function(`*argument list*`) {`
  body
  `}`
- The argument list is a comma-separated list of formal argument names,
  - name
  - name=default value
  - `...`
- The `...` is a list of the remaining arguments that do not match any of the formal arguments.
- Generally, the body is a group of `R` expressions contained in curly brackets {}. If the body is only one expression the curly brackets are optional.
- Functions are usually assigned names, but the names are optional (i.e. the `FUN` argument of `apply()`).
- *Be careful with naming functions, could overwrite existing R functions!*
- Since a function is an object we can pass it as an argument to other functions just like any other object.

Example - Odds Ratio Function

- Suppose we want to write a function that calculates the odds ratio θ of a 2×2 contingency table and the asymptotic confidence interval for θ.

		Disease	
		Yes	No
Treatment	Yes	a	b
	No	c	d

- Sample odds ratio,

$$\hat{\theta} = \frac{ad}{bc}$$

- The asymptotic standard error for $\log(\hat{\theta})$ is,

$$SE(\log \hat{\theta}) = \sqrt{\frac{1}{a} + \frac{1}{b} + \frac{1}{c} + \frac{1}{d}}$$

- The asymptotic $100(1-\alpha)\%$ confidence interval for $\log \theta$ is,

$$\log \hat{\theta} \pm z_{\alpha/2} SE(\log \hat{\theta})$$

Exponentiate the upper and lower bounds to get a confidence interval for θ.

Odds Ratio Function - Basic

```
# Calculate the odds ratio of a 2x2 table and the
# asymptotic confidence interval for the odds ratio
# Arguments:          X = 2x2 matrix
#           conf.level = confidence level
#-----------------------------------------------
odds.ratio <- function(X, conf.level=.95) {
    OR <- (X[1,1]*X[2,2])/(X[1,2]*X[2,1])
    logOR.SE <- sqrt(sum(1/X))

    alpha <- 1-conf.level
    CI.lower <- exp(log(OR) - qnorm(1-alpha/2)*logOR.SE)
    CI.upper <- exp(log(OR) + qnorm(1-alpha/2)*logOR.SE)

    cat("Odds Ratio = ", OR, "\n",
        conf.level*100, "% Confidence Interval = (",
        CI.lower, ", ", CI.upper, ")\n", sep="")
}
```

Example - Odds Ratio Function

- Consider the following data that describes the relationship between myocardial infarction and aspirin use (Agresti 1996).

```
X <- matrix(c(189, 104, 10845, 10933), nrow=2,
      dimnames=list("Treatment"=c("Placebo", "Aspirin"),
                    "Myocardial Infarction"=c("Yes", "No")))
> X
         Myocardial Infarction
  Treatment Yes    No
    Placebo 189 10845
    Aspirin 104 10933

> odds.ratio(X)
  Odds Ratio = 1.832054
  95% Confidence Interval = (1.440042, 2.330780)
> odds.ratio(X, conf.level=0.9)
  Odds Ratio = 1.832054
  90% Confidence Interval = (1.496877, 2.242282)
```

Returning Objects

- Often we will want a function to return an object that can be assigned. Two functions for returning objects are `return()` and `invisible()`.
- The `return()` function prints and returns its arguments.
- `invisible()` is similar to `return()`. This function is useful when we want a function to return values which can be assigned, but which do not print when they are not assigned.
- If the end of a function is reached without calling `return()`, the value of the last evaluated expression is returned.
- A list is often a good tool for returning multiple objects.
- Use `cat()` or `print()` to output text.

Odds Ratio Function - Returning Objects

```
odds.ratio <- function(X, conf.level=.95) {
    OR <- (X[1,1]*X[2,2])/(X[1,2]*X[2,1])
    logOR.SE <- sqrt(sum(1/X))

    alpha <- 1-conf.level
    CI.lower <- exp(log(OR) - qnorm(1-alpha/2)*logOR.SE)
    CI.upper <- exp(log(OR) + qnorm(1-alpha/2)*logOR.SE)

    cat("Odds Ratio = ", OR, "\n",
        conf.level*100, "% Confidence Interval = (",
        CI.lower, ", ", CI.upper, ")\n", sep="")

    # Different approaches for returning results
    OR
    #return(OR)
    #invisible(OR)

    #out <- list(OR=OR, CI=c(CI.lower, CI.upper), conf.level=conf.level)
    #return(out)
}
# Unassigned and assigned function results
odds.ratio(X)
OR <- odds.ratio(X)
```

Verifying Arguments

- If you are writing a function for others to use it is often a good idea to include code that verifies that the appropriate arguments were entered.
- If an argument value is not valid we want to stop executing the expression and return an error message.

- `missing()` Can be used to test whether a value was specified as an argument to a function; returns `TRUE` if a value is not specified and `FALSE` if a value is specified.
- `stop()` Stop execution of the current expression and prints an error message
- `warning()` Generate a warning message
- `message()` Generate a diagnostic message
- `stopifnot()` If any of the arguments are not all `TRUE` then `stop()` is called and an error message is produced that indicates the first element of the argument list that is not `TRUE`.

- `stop()` allows us to provide an informative error message, while `stopifnot()` requires less code.

Odds Ratio Function - Final

```
# Calculate the odds ratio (OR) of a 2x2 table and the asymptotic
# confidence interval (CI) for the OR.
# Arguments: X = 2x2 matrix; conf.level = confidence level
# Output: Prints OR and CI to console and returns a list, invisibly
#-------------------------------------------------------------
odds.ratio <- function(X, conf.level=.95) {
    # Verify Arguments
    stopifnot(!missing(X), is.matrix(X), dim(X)==c(2,2), X>0)

    # Print warning message if any expected cell count < 5
    exp.count <- (apply(X, 1, sum) %o% apply(X, 2, sum))/sum(X)
    if(any(exp.count < 5)) warning("Expected cell count < 5")

    # Calculate odds ratio and asymptotic confidence interval
    OR <- (X[1,1]*X[2,2])/(X[1,2]*X[2,1])
    logOR.SE <- sqrt(sum(1/X))

    alpha <- 1-conf.level
    CI.lower <- exp(log(OR) - qnorm(1-alpha/2)*logOR.SE)
    CI.upper <- exp(log(OR) + qnorm(1-alpha/2)*logOR.SE)

    # Format and return results
    cat("Odds Ratio = ", OR, "\n",
        conf.level*100, "% Confidence Interval = (",
        CI.lower, ", ", CI.upper, ")\n", sep="")

    out <- list(OR=OR, CI=c(CI.lower, CI.upper), conf.level=conf.level)
    invisible(out)
}
```

source()

- Once we have finished writing a function, we can save the code to a file and then use the `source()` function to read the contents of the file when the function is needed.
- Using `source()` is similar to loading a package with `library()`. We can use the functions located in the source file without declaring and evaluating the functions during the `R` session.
- Technically, the code in the file read by `source()` can be any `R` expressions, not just functions.
- To avoid specifying the path name, set the working directory to the location of the source file.

```
rm(list=ls())

source("odds.ratio.R")

X <- matrix(c(189, 104, 10845, 10933), nrow=2,
     dimnames=list("Treatment"=c("Placebo", "Aspirin"),
                   "Myocardial Infarction"=c("Yes", "No")))

odds.ratio(X)
```

Variable Scope

- When objects are created, they are assigned to an environment.
- Think of an environment as a table of assigned object names.
- Environments are hierarchical, the top-level is called the *global environment*.
- Objects assigned by the user during an `R` session are placed in the global environment.
- Assignments within the body of a function use a different environment that is created just for that function.
- When an object is evaluated within a function, `R` first looks locally within the function and its arguments for the object. If `R` cannot find the name of the object locally it looks in the parent environment, this will almost always be the global environment.
- Therefore an object within a function can have the same name as an object in the global environment; because `R` will choose to use the object defined in the local environment.
- The bottom line is that objects defined in the global environment are also accessible by a function. However, objects defined within a function are not accessible from the global environment.

Example - Variable Scope

```
rm(list=ls())   # Remove all objects

# When R evaluates var.scope.example() it uses the x and y from the
# argument list and not x and y from the global environment. Since there
# is no z in var.scope.example(), R gets z from the global environment.
var.scope.example <- function(x, y) {
     w <- x^2 + y^2 + z^2
     return(w)
}

# x, y, z are assigned in the global environment
x = 2
y = 7
z = 10
var.scope.example(x=5, y=2)

w     # w is only accessible within var.scope.example()
```

Vector Arguments

- Functions that perform whole object computations or vectorizing computations can improve efficiency. That is a function that accepts a vector as an argument and returns a vector of results.
- Some functions may not be able to perform vectorizing computations. Even though there are functions like `integrate()` that require it.
- One obvious solution for applying an unvectorizing function to a vector of arguments is to form a loop and then apply the function to every element.
- Other possibly more compact and clearer approaches,

`lapply(), sapply()`	Apply a function to each element of a vector or list
`mapply()`	Multivariate version of `sapply()`, apply a function to the corresponding elements from multiple vectors
`Vectorize()`	Return a new function that acts as if `mapply()` was called

- These approaches may not improve efficiency because they do not reduce the number of function calls.

Example - Vector Arguments

- These density functions can be passed a vector `x` and return a vector with the density evaluated at each element of `x`. We could also pass a vector of parameters, but this would not give meaningful results.
- A more compact approach would have been to use `dpois()` and `dgamma()`

```
poisson <- function(x, lambda) exp(-lambda)*lambda^x/factorial(x)
poisson(1:5, lambda=1)
poisson(1:5, lambda=1:5)   # Not very useful

# Evaluate the poisson density at 1:5 for lambda=1,5
sapply(c(lambda.1=1, lambda.5=5), poisson, x=1:5)

# Need to be careful with naming functions, could overwrite existing R functions
gamma.density <- function(x, alpha, beta) {
     x^(alpha-1)*exp(-x/beta)/gamma(alpha)/beta^alpha
}

# Evaluate the gamma density at 1:5 for alpha=beta=1, alpha=beta=2, alpha=beta=3
mapply(gamma.density, alpha=1:3, beta=1:3, MoreArgs=list(x=1:5))
# Evaluate the gamma density at 1:5 for all combinations of alpha and beta
parm <- expand.grid(alpha=1:3, beta=1:3)
out <- mapply(gamma.density,  parm[,1], parm[,2], MoreArgs=list(x=1:5))
colnames(out) = paste("alpha", parm[,1], "beta", parm[,2], sep=".")
```

Example - Vector Arguments

- Now suppose we want to apply our `odds.ratio()` function to a set of 2×2 tables.

```
# Create 5 2x2 tables using random poisson data with lambda=10
tables.2x2 <- replicate(5, matrix(rpois(4, lambda=10),nrow=2),simplify=F)

# Apply odds.ratio() to each matrix in the list tables.2x2
result <- lapply(tables.2x2, odds.ratio)

# Extract the odds ratio for each table
unlist(lapply(result, FUN=function(x) x$OR))
```

apply()

- Recall the syntax from lecture 2, `apply(X, MARGIN, FUN, ...)`
- The `FUN` argument can be any function of the vector specified by `MARGIN`, either an `R` function or one we write.
- `apply()` only passes `FUN` one argument, the vector specified by `MARGIN`. The `...` are used to pass any additional arguments to `FUN`.

```
A <- matrix(sample(1:100, 12), nrow=3, ncol=4)

# Calculate the mean of each column
apply(A, 2, FUN=mean)

# Simply functions can be defined within the apply() command
# Calculate variance of each column
apply(A, 2, FUN=function(x) sum((x-mean(x))^2)/(length(x)-1))

# For more complex functions, specify the name of the function
# Apply the central() function from slide 15 to each column
apply(A, 2, FUN=central, measure="mean")
```

Binary Operators

- Recall, the binary operator %*% used for matrix multiplication. This operator is actually the function `` `%*%`(x,y) ``.

```
> A <- B <- matrix(1:4,nrow=2)
> A%*%B
     [,1] [,2]
[1,]    7   15
[2,]   10   22
> `%*%`(A, B)
     [,1] [,2]
[1,]    7   15
[2,]   10   22
```

- Entirely new binary operators can be defined. Any function object with the name having the pattern %*text*% will be evaluated as `` `%text%`(x,y) ``.

```
> '%nick%' <- function(x,y) x^y
> 3%nick%2
[1] 9
```

Advantages of Functions

- Leads to more compact code when a task needs to be performed multiple times.
- Easier to make changes to a function then to make changes to several locations throughout a program.
- A well written function is more trustworthy and transparent, all of the expressions are contained within the function.
- Easier to interact through a function interface then to type lines of code.
- Function computations are performed locally; objects created within a function will not accidentally overwrite global objects.
- Most convenient way to share programs with other users.
- A well written function can be adapted to different situations; solve more then one problem with a single function.

Writing Reliable Functions

- A function is well-defined when the value returned by the function is completely determined by the function's arguments. No other information, such as global variables, are needed to evaluate the function.
- A function should not effect later computations or function calls.
  - For example, `options()` sets global parameters however `options()` can be set from a local environment within a function. Therefore, computations inside one function could cause a permanent effect on other functions.
- Comment code. Functions that are transparent and have well-documented code are much more trustworthy then functions with obscure undocumented code.
- There is nothing wrong with writing trivial functions that are not well designed. The first version of a function is often for immediate use. After using a function for awhile you may realized that the function can be extended with additional arguments or plays an important role in your analysis. At this point, consider refining the functions implementation.

Example - Reliable Functions

- Below are two functions for calculating the log-likelihood of an exponential random sample.
- Since the log-likelihood is a function of the parameters, the first function, `log.like.exp_ok`, is a function of `theta`. However this function is not well-defined, cause changes to future computations and there are no comments that describe the function.

```
defaults <- options()

log.like.exp_ok <- function(theta) {
     options(digits=3)
     LL <- sum(log(dexp(x, rate=theta)))
}
x <- rexp(100, rate=3)
log.like.exp_ok(3)    # Potentially dangerous if x is not what we think it is.
signif(var(x), 7)     # The variance of the x using 7 significant digits???

options(defaults)     # Restore default options

# Return the exponential log-like with three digits
log.like.exp_better <- function(x, theta) {
     LL <- sum(log(dexp(x, rate=theta)))
     print(LL, digits=3)
}
log.like.exp_better(x, 3)  # Not restricted to the data being called x
```

Extract Function Components

`body()`	Get or set the body of a function
`formals()`	Get or set the formal arguments of a function
`args()`	Displays the argument names and corresponding default values of a function
`nargs()`	Used within a function, returns the number of arguments supplied to that function
`match.call()`	Used within a function, returns a function call in which all of the specified arguments are specified by their full names

Simulation Function

- For a random sample $X_1, X_2, \ldots, X_n$, with sample mean $\bar{x}$ and sample variance s. The $100(1-\alpha)\%$ t-confidence interval for the mean is,

$$\bar{x} \pm t^\star \frac{s}{\sqrt{n}}$$

  where $t^\star$ is the $1 - \frac{\alpha}{2}$ critical value for the t distribution with $n-1$ degrees of freedom.
- If $X_1, X_2, \ldots, X_n$ is a random sample from a normal population then the t-confidence interval of the mean is exact and is approximately correct for large n otherwise.
- So how large of an n is necessary? How does the shape of the distribution affect n?
- To investigate this question, we will write a well defined `R` function that performs a simulation for different statistical distributions and sample sizes.

Example - Simulation Function

```
sim.t.CI <- function(RVgen,          # Random variable generator, i.e. rnorm
                     n,              # Sample size, possibly a vector
                     nsim=1000,      # Number of iterations, default 1000
                     alpha=.05,      # 100(1-alpha) Confidence level
                     mu,             # Population mean
                     ...){           # Arguments to be passed to RVgen
   # Verify arguments
   stopifnot(!missing(RVgen), is.function(RVgen), !missing(n), !missing(mu))

   # Setup output matrix
   results <- matrix(nrow=length(n), ncol=1, dimnames=list(n, deparse(substitute(RVgen))))

   # Loop through different sample sizes
   for(i in 1:length(n)) {
       CP <- array(dim=nsim)

       # Simulation loop
       for(j in 1:nsim) {
           # Generate random sample
           x <- RVgen(n[i], ...)

           # Calculate t-confidence interval
           xbar <- mean(x)
           lower <- xbar-qt(1-alpha/2, n[i]-1)*sqrt(var(x)/n[i])
           upper <- xbar+qt(1-alpha/2, n[i]-1)*sqrt(var(x)/n[i])
           CP[j] <- ifelse(lower < mu & mu < upper, 1, 0)
       }
       results[i,] <- mean(CP)
   }
return(results)
}
```

Example - Simulation Function

```
> sim.t.CI(rnorm, n=c(10,25,50,100), mu=0, mean=0)
    rnorm
10  0.936
25  0.961
50  0.955
100 0.958
```

```
# Apply sim.t.CI() to a normal, uniform and gamma distribution
startTime <- proc.time()[3]

run1 <- sim.t.CI(rnorm,  n=c(10,25,50,100), mu=0,  mean=0)
run2 <- sim.t.CI(runif,  n=c(10,25,50,100), mu=.5, min=0, max=1)
run3 <- sim.t.CI(rgamma, n=c(10,25,50,100), mu=2,  shape=1, scale=2)
cbind(run1, run2, run3)

elapseTime <- proc.time()[3]-startTime
cat("Elapsed Time:",floor(elapseTime/60),"min",elapseTime%%60,"sec\n")
```