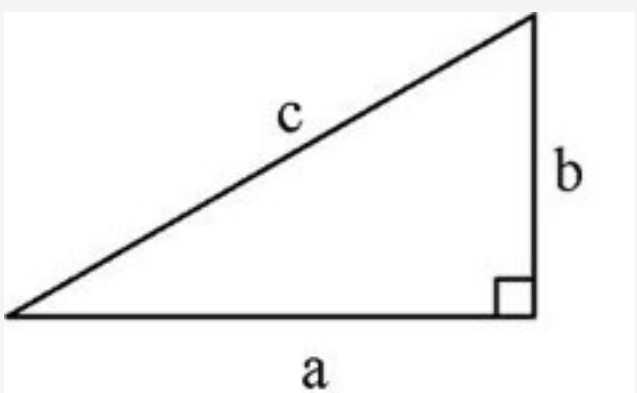
Functions - Review

```
pythag <- function(a = 4, b = 5) {
     if (!is.numeric(a) | !is.numeric(b)) {
               stop("I need real values to make this work")
     }
     hypo = sqrt(a^2 + b^2)
     myreturnlist = list(hypoteneuse = hypo, sidea = a, sideb = b)
     return(myreturnlist)
}
```

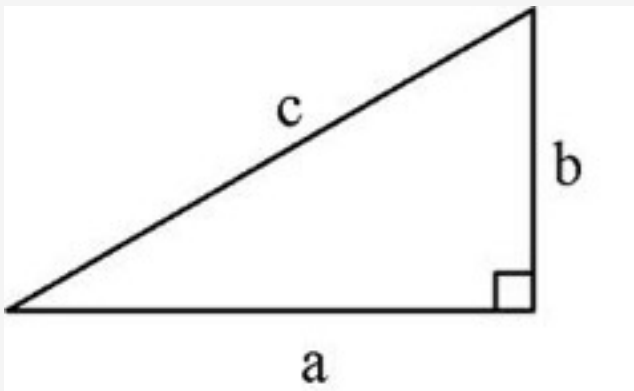
"a" and "b" represent arguments that correspond to sides a and b, respectively.

We compute "c" the hypoteneuse and return its value



Functions - Review

```
pythag(3,4)
$hypoteneuse
[1] 5

$sidea
[1] 3

$sideb
[1] 4
```



Functions - Scoping Rules

Scoping Rules

The scoping rules for R are the main feature that make it different from the original S language.

The scoping rules determine how a value is associated with a free variable in a function R uses lexical scoping or static scoping.

Lexical scoping turns out to be particularly useful for simplifying statistical computations

`www.stat.berkeley.edu%2F~statcur%2FWorkshop2%2FPresentations%2Ffunctions.pdf`

Functions - Formal Terminology

* In R, the ***global frame*** is called the global environment or the workspace, and is kept in memory.

• ***Scoping rules*** determine where the interpreter looks for values of free variables.

* An ***environment*** is a sequence of frames.

* A value bound to a variable in a frame earlier in the sequence will take precedence over a value bound to the same variable in a frame later in the sequence. The first value is said to ***shadow*** or ***mask*** the second.

http://cran.r-project.org/doc/contrib/Fox-Companion/appendix-scope.pdf

Functions - Formal Terminology

```
environment()
<environment: R_GlobalEnv>

ls()                # your current environment will  look different
[1] "a" "f" "x"
```

You can remove all variables in your current environment. This is for when you want to "clean house" and start over.

```
rm(list=ls())
```

http://cran.r-project.org/doc/contrib/Fox-Companion/appendix-scope.pdf

Functions - Formal Terminology

As proof that functions run within their own environment consider this example:

```
myenvfunc <- function(somenv) {
   print(environment())
}

myenvfunc()
<environment: 0x1044cd908>
```

So myenvfunc runs in its OWN environment that is separate from the Global environment.

Functions - Scoping Rules

In this example an object x will be defined with value zero. Inside myfunc, the x is defined with value 3. Executing the function myfunc will not affect the value of the global variable `x'.

```
x = 0      # Set x to zero in the global environment (your console)

myfunc <- function(x) {     # Define this function
   x = 3                    # This value of "x" is private
   return(x)
}

myfunc()       # Function returns 3
[1] 3

x              # The value of x in the global env is unchanged
[1] 0
```

Functions - Scoping Rules

This means a normal assignment within a function will not overwrite objects outside the function. An object created within a function will be lost when the function has finished.

Functions - Scoping Rules

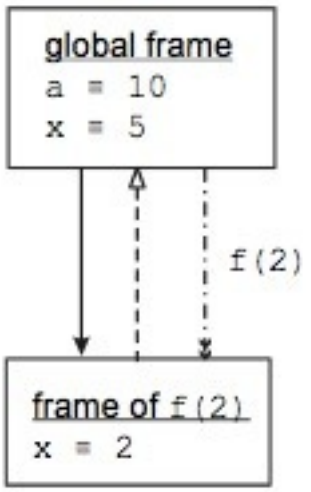
```
* Time for an example.

rm(list=ls()) # Clears all variables from your environment.

exampf <- function(x) {
    return(x + a)
}

ls()      # The function f is in our global environment
[1] "exampf"

exampf(2)
Error in exampf(2) : object 'a' not found
```

When f is called it passes the value of 2. So "x" assumes a local value of 2. Then the function wants to add x = 2 to the value of a though non has been specified so the function results in an error. This seems reasonable since R can't find a variable called "a" anywhere.

http://cran.r-project.org/doc/contrib/Fox-Companion/appendix-scope.pdf

Functions - Scoping Rules

```
* Time for an example

exampf <- function(x) {
   return(x + a)
}

a = 10
x = 5

> ls()
[1] "a" "exampf" "x"

exampf(2)
[1] 12
```



When exampf is called, the local binding of x = 2 shadows the global binding x = 5. The variable **a** is a free variable in the frame of the function call, and so the global binding a = 10 applies.

http://cran.r-project.org/doc/contrib/Fox-Companion/appendix-scope.pdf

Functions - Scoping Rules

```
* Time for an example

exampf <- function(x) {
   a = 3
   return(x + a)
}

a = 10
x = 5

exampf(2)
[1] 5
```

When exampf is called, the local binding of x = 2 shadows the global binding x = 5. The variable **a** is defined with the function f so it shadows the value of a = 10 in the global environment. So we have f(2) resulting in a value of five.

`http://cran.r-project.org/doc/contrib/Fox-Companion/appendix-scope.pdf`

Functions - Scoping Rules

What does all this mean ?

Well, the arguments and variables that you use within a function are private to that function even if you use the same name as a previously defined variable.

It is a common mistake to refer to a variable within a function without initializing it to something first. If it has the same name as a variable in the "global environment" then it will pick up that variable's value.

So make sure you keep track of what you are doing within your function.

Use descriptive and unambiguous variable names

`http://cran.r-project.org/doc/contrib/Fox-Companion/appendix-scope.pdf`

Functions - Scoping Rules

Use descriptive and unambiguous variable names

```
exampf <- function(x) {
   a = 3
   return(x + a)
}
```

Maybe do something like:

```
exampf <- function(exampx) {
     exampa = 3
     return(exampx + exampa)
}

-OR-

exampf <- function(exampx, exampa) {
      return(exampx + exampa)
}
```

Functions - Scoping Rules

Functions can always call other functions that exist within the same environment. It happens all the time.

```
exampf <- function(myvec) {
   retval = c(mean(myvec), sd(myvec))  # mean and sd are known
   return(retval)
}
```

```
> exampf(1:10)
[1] 5.50000 3.02765
```

Functions - Scoping Rules

This includes any functions that you have written too. Define this function within RStudio either at the console or from the edit Window. It is now in your "Global Environment" and can be used at the prompt or by other functions that you might write.

```
is.odd <- function(somenumber) {
   retval = 0
   if (somenumber %% 2 != 0) {
       retval = TRUE
   } else {
       retval = FALSE
   }
   return(retval)
}
is.odd(3)
[1]  TRUE
```

Functions - Scoping Rules

Let's say we are writing a function to compute the median of a vector. We'll need to determine if its length is even or odd so we could use the is.odd function to help us out here.

```
mymedian <- function(medianvec) {

# Function to compute the median of a vector

  medianveclength = length(medianvec)

  if (is.odd(medianveclength)) {   # is.odd is available for use

     # We find the median using the formula for odd length vectors

  } else {

     # We find the median using the formula for even length vectors

  }
}
```

Functions - Scoping Rules

Or ,alternatively, we could define the is.odd function within the mymedian function although this means that is.odd would be available only to the mymedian function.

Functions - Scoping Rules

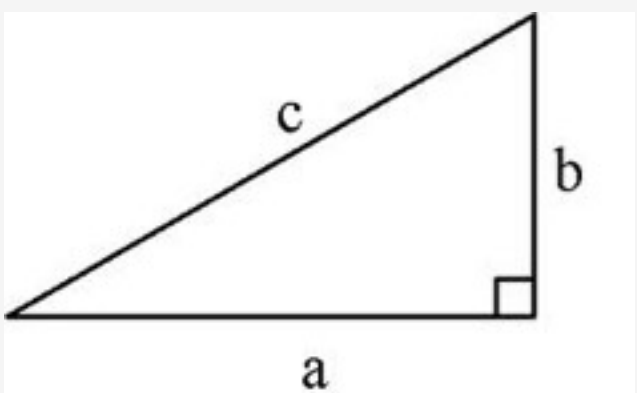
```
mymedian <- function(medianvec) {

  is.odd <- function(somenumber) {  # We define is.odd inside of mymedian
    retval = 0
    if (somenumber %% 2 != 0) {
        retval = TRUE
    } else {
        retval = FALSE
    }
    return(retval)
  }

# Function to compute the median of a vector

  medianveclength = length(medianvec)
  if (is.odd(medianveclength)) {

     # We find the median using the formula for odd length vectors

  } else {

     # We find the median using the formula for even length vectors

  }
}
```

Functions - Arguments

```
pythag <- function(a = 4, b = 5) {
     if (!is.numeric(a) | !is.numeric(b)) {
               stop("I need real values to make this work")
     }
     hypo = sqrt(a^2 + b^2)
     myreturnlist = list(hypoteneuse = hypo, sidea = a, sideb = b)
     return(myreturnlist)
}
```

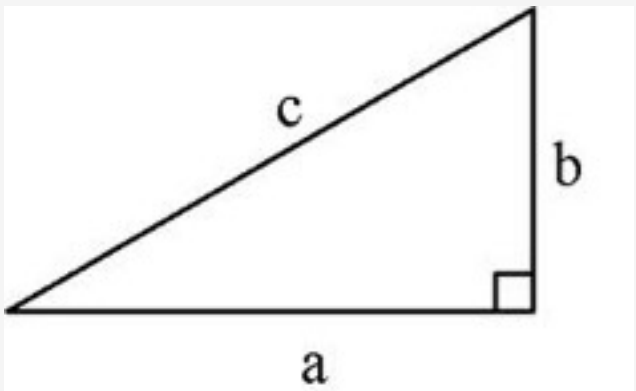
"a" and "b" represent arguments that correspond to sides a and b, respectively.

We compute "c" the hypoteneuse and return its value



Functions - Arguments

```
pythag(4,5)
$hypoteneuse
[1] 6.403124

$sidea
[1] 4

$sideb
[1] 5

pythag(5,4)
$hypoteneuse
[1] 6.403124

$sidea
[1] 5

$sideb
[1] 4
```



We get the same answer in either case but maybe we just got lucky since the formula doesn't really care which side we call a or b.

Functions - Arguments

Look at the help page for the mean function:

```
mean                    package:base                    R Documentation

Arithmetic Mean

Description:

     Generic function for the (trimmed) arithmetic mean.

Usage:

     mean(x, ...)

     ## Default S3 method:
     mean(x, trim = 0, na.rm = FALSE, ...)
```

The function has three basic arguments:

x - a value or vector to take the mean of
trim - a value that let's you trim the vector by some percentage
na.rm - a value that let's you ignore missing values

Functions - Arguments

The function has three basic arguments:

x - a value or vector to take the mean of
trim - a value that let's you trim the vector by some percentage
na.rm - a value that let's you ignore missing values

```
set.seed(1)
myx = rnorm(20)

mean(myx)       # myx MATCHES the "x" argument
[1] 0.1905239

mean(myx,0.05) # myx MATCHES "x" and 0.05 MATCHES "trim"

mean(myx,0.05,TRUE) # myx MATCHES "x", 0.05 MATCHES "trim", TRUE matches na.rm
[1] 0.2461054

COULD DO:

mean(x = myx, trim = 0.05, na.rm = TRUE)
[1] 0.2461054

As long as you name the arguments you type them in any order

mean(trim = 0.05, na.rm = TRUE, x = myx)
[1] 0.2461054
```

Functions - Arguments

The function has three basic arguments:

x - a value or vector to take the mean of
trim - a value that let's you trim the vector by some percentage
na.rm - a value that let's you ignore missing values

BUT THIS WON'T WORK:

```
# Can't switch positions without using the names

mean(TRUE,0.05,myx)
[1] 1
Warning message:
In if (na.rm) x <- x[!is.na(x)] :
  the condition has length > 1 and only the first element will be used
```

Functions - Arguments

* Use named arguments especially in cases where there is a long list of arguments.

* If you use named arguments then you don't have to remember the position of the arguments. Check out the plot command, which has way too many options. So there is no convenient way to remember the arguments by position unless you have a photographic memory.

```
plot(mtcars$wt,mtcars$mpg)

plot(mtcars$wt,mtcars$mpg, bty="l")

plot(mtcars$wt,mtcars$mpg, main="Default")

plot(mtcars$wt,mtcars$mpg, tck=0.05, main="tck=0.05")

plot(mtcars$wt,mtcars$mpg, axes=F, main="Different tick marks for each axis")

plot(mtcars$wt,mtcars$mpg,
+ xlim=c(0,100),xlab="Gallons",pch=21,bg="blue",col="red")
```

Functions - "..." Argument

Look at the help page for the mean function:

```
mean                    package:base                    R Documentation

Arithmetic Mean

Description:

     Generic function for the (trimmed) arithmetic mean.

Usage:

     mean(x, ...)

     ## Default S3 method:
     mean(x, trim = 0, na.rm = FALSE, ...)
```

The function has three basic arguments:

x - a value or vector to take the mean of
trim - a value that let's you trim the vector by some percentage
na.rm - a value that let's you ignore missing values

Functions - The "..." Argument

Passing an unspecified number of parameters to a function

We can pass an unspecified number of parameters to a function by using the ... notation in the argument list. This is usually done when you want to give the user the option of passing any number of arguments that are intended for another function.

Suppose you write a small function to do some plotting. Basically you want to hard code in the color red for all plots.

```
my.plot <- function(x,y, ...) {
    plot(x,y, col="red",...)
}
```

www.ats.ucla.edu/stat/r/library/intro_function.htm

Functions - The "..." Argument

Suppose you write a small function to do some plotting. Basically you want to hard code in the color red for all plots.

```
my.plot <- function(x,y, ...) {
    plot(x,y, col="red",...)
}

my.plot(mtcars$wt,mtcars$mpg, pch=15, lty=2, xlim=c(0,40))
```

The function my.plot now accepts any argument that can be passed to the plot function (like col, xlab, etc.) without needing to specify those arguments in the header of my.plot.

Functions - The "..." Argument

Note that, technically, you could by-pass the x-y arguments altogether in this case but then why would you even bother to write your own function ?

```
my.plot <- function(...) {
    plot(...)
}

my.plot(mtcars$wt,mtcars$mpg, pch=15, lty=2, xlim=c(0,40))
```

The function my.plot now accepts any argument that can be passed to the plot function (like col, xlab, etc.) without needing to specify those arguments in the header of my.plot.

Functions - Debugging

If functions work well from the beginning then we don't have to "debug" them. But in real life this is seldom the case. There is always something that doesn't work just like you want it. In this case you need to do some debugging.

```
pythag <- function(a = 4, b = 5) {
     if (!is.numeric(a) | !is.numeric(b)) {
               stop("I need real values to make this work")
     }
     hypo = sqrt(a^2 + b^2)
     myreturnlist = list(hypoteneuse = hypo, sidea = a, sideb = b)
     return(myreturnlist)
}

pythag(-1,3)   # Hmmm. Well -1 isn't a valid value for side a.
$hypoteneuse
[1] 3.162278

$sidea
[1] -1

$sideb
[1] 3
```

Functions - Debugging

In this case we might detect that the value of -1 gets squared anyway which avoids the problem but this is a simple case. Usually we would use a cat or print statement to print out variables along the way.

```
pythag <- function(a = 4, b = 5) {
     if (!is.numeric(a) | !is.numeric(b)) {
               stop("I need real values to make this work")
     }
     cat("a is ",a,"b is ",b,"\n")
     hypo = sqrt(a^2 + b^2)
     myreturnlist = list(hypoteneuse = hypo, sidea = a, sideb = b)
     return(myreturnlist)
}

pythag(-1,3)
a is  -1 b is  3   # Well this helps but only a little.
$hypoteneuse
[1] 3.162278

$sidea
[1] -1

$sideb
[1] 3
```

Functions - Arguments

We can put in more cat statements but it gets tedious.

```
pythag <- function(a = 4, b = 5) {
     if (!is.numeric(a) | !is.numeric(b)) {
               stop("I need real values to make this work")
     }
     cat("a is ",a,"b is ",b,"\n")
     cat("a^2 is",a^2,"\n")
     hypo = sqrt(a^2 + b^2)
     myreturnlist = list(hypoteneuse = hypo, sidea = a, sideb = b)
     return(myreturnlist)
}

pythag(-1,3)
a is  -1 b is  3
a^2 is 1           # Okay maybe this helps some more
$hypoteneuse
[1] 3.162278

$sidea
[1] -1

$sideb
[1] 3
```

Functions - Debugging

At the debug prompt the user can enter commands or R expressions, followed by a newline. The commands are:

'n' (or just an empty line, by default). Advance to the next step.

'c' continue to the end of the current context: e.g. to the end of the loop if within a loop or to the end of the function.

'cont' synonym for 'c'.

'where' print a stack trace of all active function calls.

'Q' exit the browser and the current evaluation and return to the top-level prompt.

Leading and trailing whitespace is ignored, except for an empty line

Functions - Debugging

Anything else entered at the debug prompt is interpreted as an R expression to be evaluated in the calling environment.

In particular typing an object name will cause the object to be printed, and 'ls()' lists the objects in the calling frame.

If you want to look at an object with a name such as 'myvar', Then you can print it explicitly:

Browse[2]> print(myvar)

-OR-

Browse[2]> myvar

Functions - Debugging

To activate the R debugger we use the debug function:

```
pythag <- function(a = 4, b = 5) {
     if (!is.numeric(a) | !is.numeric(b)) {
             stop("I need real values to make this work")
     }
     hypo = sqrt(a^2 + b^2)
     myreturnlist = list(hypoteneuse = hypo, sidea = a, sideb = b)
     return(myreturnlist)
}

debug(pythag)  # Informs R that we want to invoke the debugger

pythag(-1,3)


undebug(pythag) # turns off the debugger on pythag
```

Functions - Debugging

```
debug(pythag)  # Informs R that we want to invoke the debugger

pythag(-1,3)
debugging in: pythag(-1, 3)
debug at #1: {
    if (!is.numeric(a) | !is.numeric(b)) {
        stop("I need real values to make this work")
    }
    hypo = sqrt(a^2 + b^2)
    myreturnlist = list(hypoteneuse = hypo, sidea = a, sideb = b)
    return(myreturnlist)
}
Browse[2]>
```

Functions - Debugging

Okay, check this example out. It will fail because starting with the second element of the input vector the log function will complain.

```
mybadfunc <- function(somevec) {
  retval = vector()
  for (ii in 1:length(somevec)) {
    retval[ii] = log(somevec[ii])
  }
  return(retval)
}

input = 1:-1
[1]  1  0 -1

mybadfunc(input)
1]  1  0 -1
Warning message:
In log(somevec[ii]) : NaNs produced
```

Functions - Debugging

If we have a "sensitive" function (like the log function) we can insert code that will only call the debugger if and when there is a possible problem. Like in this case if we try to take the log of a value of zero or less.

```
mybadfunc <- function(somevec) {
  retval = vector()
  for (ii in 1:length(somevec)) {
    if (somevec[ii] <= 0) {
      browser()   # This calls the debugger only if the condition is met
    }
    retval[ii] = log(somevec[ii])
  }
  return(retval)
}

mybadfunc(input)
Called from: mybadfunc(input)
Browse[1]>
```

Functions - Debugging

```
mybadfunc(input)
Called from: mybadfunc(input)
Browse[1]> ls()
[1] "ii"      "retval"  "somevec"
Browse[1]>  ii
[1] 2
Browse[1]> somevec[2]
[1] 0                    # aha - the culprit
Browse[1]>
```

Functions - Median function

figure out if length of vector is even or odd

if odd then sort it and store it in a new vector

divide length of sorted vector by 2 and round up to get "middle" element index

use this as an index into the sorted vector and this will be the median of an odd length vector.

Functions - Median function

```
mymedian <- function(somevec) {
  mymed = 0
  if (length(somevec) %% 2 != 0) {
            mys = sort(somevec)    # Sort the vector
            half = length(mys)/2   # Find candidate middle
            half = ceiling(half)   # Round it up
            mymed = mys[half]      # Get the associated value
  } else {

       # Write logic for processing vectors of even length
  }
 return(mymed)
}
```

Functions - Median function

```
mymedian <- function(somevec) {
  mymed = 0
  if (length(somevec) %% 2 != 0) {
       mymed = sort(somevec)[ceiling(length(mys)/2)]
  } else {

       # Write logic for processing vectors of even length
  }
 return(mymed)
}
```