Functions - Scoping Rules

<u>Scoping Rules</u>

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:

```
environment()
<environment: R_GlobalEnv>

myenvfunc <- function() {
   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

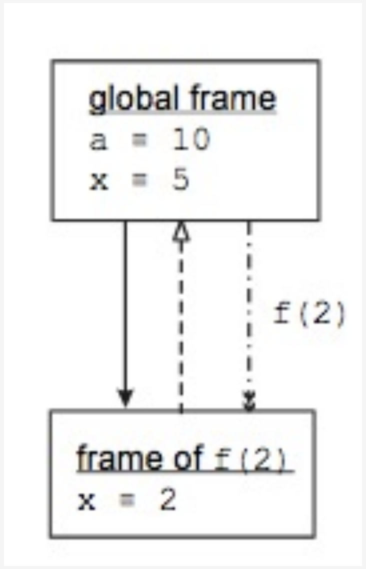
```
* Time for an example.

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
```



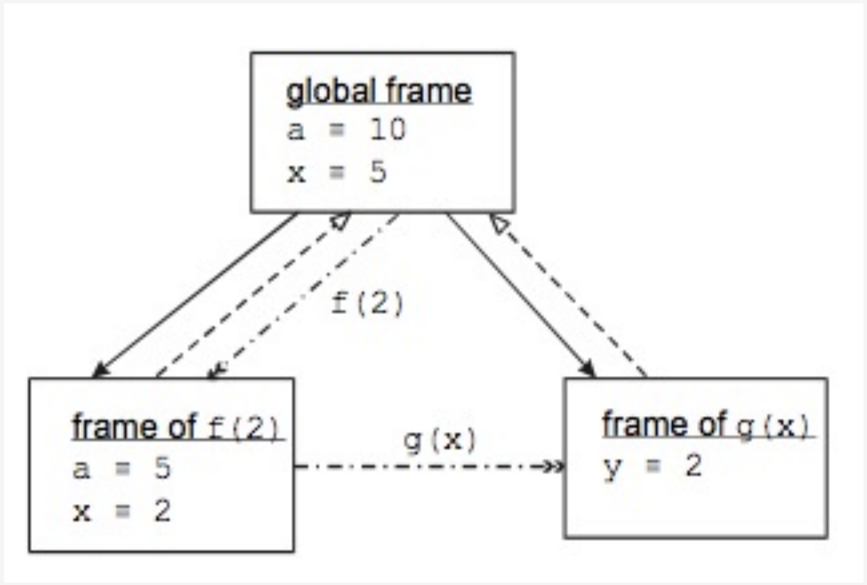
When exampf is called, the local binding of x = 2 shadows the global binding x = 5. The variable **a** is a free variable in the frame of the function call, and so the global binding a = 10 applies.

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

Functions - Scoping Rules

* Time for another example

```
a = 10 ; x = 5

f <- function(x) {
     a = 5
     g(x)
}

g <- function(y) {
    return(y + a)
}

f(2)
[1] 12
```



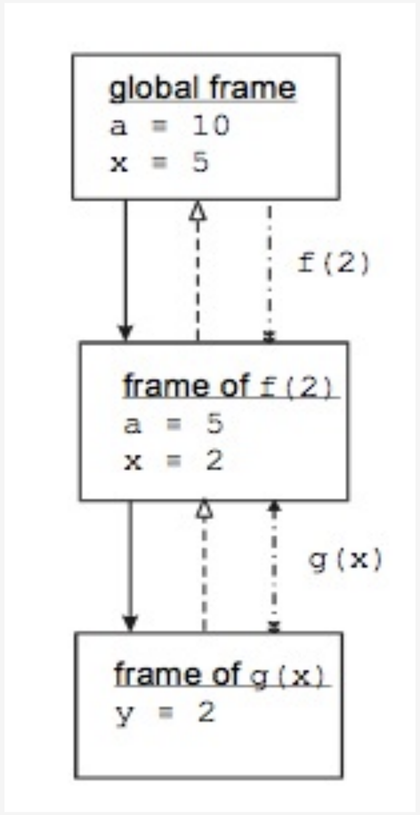
In R, the global binding a = 10 is used when f calls g, because a is a free variable in g, and g is defined at the command prompt in the global frame.

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

Functions - Scoping Rules

* Time for another example

```
a = 10

x = 5

f <- function(x) {
    a = 5
    g <- function(y) {
       return(y + a)
    }
    g(x)
}

f(2)
[1] 7
```



The local function g is defined within the function f, and so the environment of g comprises the local frame of g followed by the environment of f. Because a is a free variable in g, the interpreter next looks for a value for a in the local frame of f; it finds the value a = 5, which shadows the global binding a = 10

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

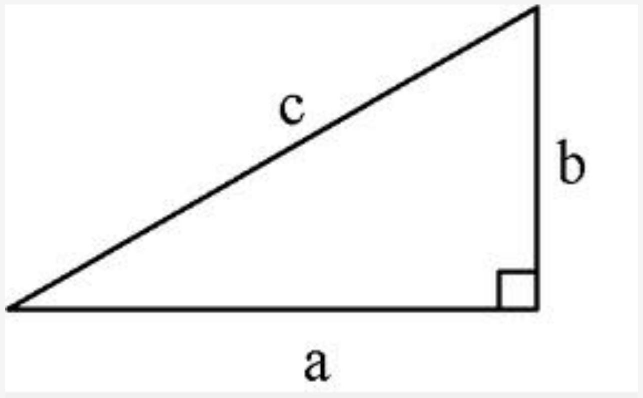
```
mymedian <- function(medianvec) {

  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)
}
```

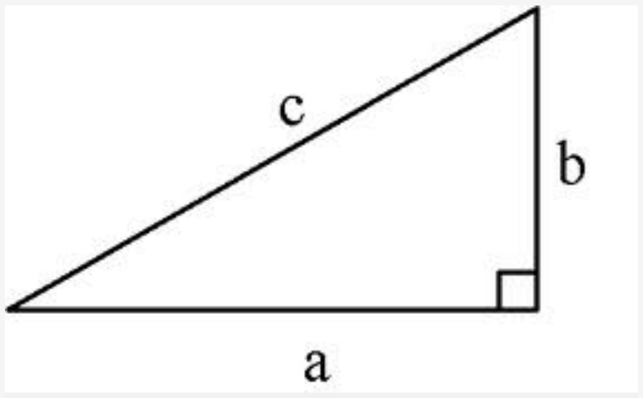
"a" and "b" represent arguments that correspond to sides a and b, respectively.

We compute "c" the hypoteneuse and return its value



Functions - Arguments

```
pythag(4,5)      # 4 is matched to a and 5 is matched to b
$hypoteneuse
[1] 6.403124

$sidea
[1] 4

$sideb
[1] 5

pythag(5,4) # 5 is matched to a
$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(x=mtcars$wt, y=mtcars$mpg)

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

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

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

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

plot(x=mtcars$wt, y=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(x=mtcars$wt, y=mtcars$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(x=mtcars$wt, y=mtcars$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.

Debugging

Initial attempts at debugging usually involve some form of inserting print statements to view variables as they are recognized in the function.

```
my.func = function(x,y) {
     z = x + y
     return(z)
}

> my.func(1,2)
[1] 3

> my.func(1,"two")
Error in x + y : non-numeric argument to binary operator
```

Debugging

Initial attempts at debugging usually involve some form of inserting print statements to view variables as they are recognized within the function.

```
my.func = function(x,y) {
     cat("x is",x,"y is",y,"\n")
     z = x + y
     return(z)
}

> my.func(1,2)
x is 1 y is 2
[1] 3

> my.func(1,"2")
x is 1 y is 2
Error in x + y : non-numeric argument to binary operator
```

Oh so we forgot that the function can't handle arguments if they are characters.

Debugging

So then we start putting in validation into our functions, which we should have been doing all along anyway.

```
my.func = function(x,y) {
  if (!is.numeric(x) || !is.numeric(y) ) {
      stop("Check your input")
  }
      cat("x is",x,"y is",y,"\n")
      z = x + y
      return(z)
}


> my.func(1,"2")
Error in my.func(1, "2") : Check your input
```

The stop function halts any further progress if it is invoked. You could use warning() instead.

Debugging

So then we start putting in validation into our functions, which we should have been doing all along anyway.

```
my.func = function(x,y) {
  if (!is.numeric(x) || !is.numeric(y) ) {
      warning("Check your input")
  }
      cat("x is",x,"y is",y,"\n")
      z = x + y
      return(z)
}

my.func(1,"2")
x is 1 y is 2
Error in x + y : non-numeric argument to binary operator
In addition: Warning message:
In my.func(1, "2") : Check your input
```

The warning function tells us there is a problem but the program moves on anyway even though it is doomed to fail.

Debugging

R provides several approaches for "formal" or "proper" debugging:

1) **Use traceback() when your program fails to see relevant messages**

2) **When you want to interact with your R program on a step-by-step basis, you can use the debug() function. debug() accepts a single argument, the name of a function.**

3) **The trace() function allows you to temporarily add arbitrary code to a function; This allows inserting code in a function without permanently changing it.**

There can be overlap between these methods and its not always clear which is the best since you might not know how deep you will need to go into your code. I prefer number 2.

Debugging - traceback

This exercise is to investigate what happens when you call functions from within functions and things go wrong. This is a common occurrence and you need to know how to debug such situations.

```
foo <- function(x) {
    print(1)
    bar(2)
}

bar <- function(x) {
    x + some.nonexistent.variable
}

So call foo(2). We'll get an error.

foo(2)
[1] 1
Error in bar(2) : object 'some.nonexistent.variable' not found
```

Debugging - traceback

So next call the traceback() function to see "the stack" of function calls that led to the error.

```
traceback()
2: bar(2) at #3
1: foo(2)
```

So we see that the problem happened in function "Bar" called at line #3 from function foo(). We already know that the reason that "bar" failed was because of the absence of "some.nonexistent.variable".

```
foo <- function(x) {
    print(1)
    bar(2)              # last call was to foo here at line #3
}

bar <- function(x) {
    x + some.nonexistent.variable
}
```

Debugging - traceback

Let's look at a more involved example.

```
f = function(x) {
  r = x - g(x)
  return(r)
}

g = function(y) {
  r = y * h(y)
  return(r)
}

h = function(x) {
  r = log(x)
  if(r < 10)
     return(r^2)
  else
     return(r^3)
}

f(2)
[1] 1.039094
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - traceback

```
f = function(x) {
  r = x - g(x)
  return(r)
}

g = function(y) {
  r = y * h(y)
  return(r)
}

h = function(x) {
  r = log(x)
  if(r < 10)
     return(r^2)
  else
     return(r^3)
}

f(-1)
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed In
addition: Warning message:
In log(x) : NaNs produced
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - traceback

```
> traceback()

3: h1(y) at #2   # Check out line #2 of function h.
2: g1(x) at #2
1: f1(-1)

f = function(x) {
  r = x - g(x)
  return(r)
}

g = function(y) {
  r = y * h(y)
  return(r)
}

h = function(x) {
  r = log(x)
  if(r < 10)
     return(r^2)
  else
     return(r^3)
}
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - traceback

A powerful tool is the "debug" function that let's us step through a function as it is executing.

```
SS <- function(mu, x) {    # Compute Sum of Squares
    d = x - mu
    d2 = d^2
    ss = sum(d2)
    return(ss)
}

# Set the seed so that the results are reproducible


set.seed(100)

x = rnorm(100)

SS(1,x)
[1] 202.5615
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - debug()

```
SS <- function(mu, x) {    # Compute Sum of Squares
     d <- x - mu
     d2 <- d^2
     ss <- sum(d2)
     return(ss)
}

# Now we debug the function


debug(SS)

SS(1,x)
debugging in: SS(1, x)
debug at #1: {
    d <- x - mu
    d2 <- d^2
    ss <- sum(d2)
    return(ss)
}

Browse[2]>
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - debug()

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).

Debugging - debug()

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
```

Debugging - debug()

Here are some common activities:

List objects / variables in the workspace: ls() or objects()

Print variable contents: type the variable name

Set a variable to a new value: Make an assignment (e.g. x = 10)

Debug a function that is being called within the function you are
Currently debugging. For example if you are in function 1 and you wish to debug
function 2 then go through the lines (press "n") until you get to the function of
interest and then type "debug(function2)
This will direct you into that function.

```
Browse> debug(function2)
```

Debugging - debug()

```
> debug(SS)
> SS(1,x)
debugging in: SS(1, x)
debug at #1: {
    d <- x - mu
    d2 <- d^2
    ss <- sum(d2)
    return(ss)
}
Browse[2]>
debug at #2: d <- x - mu

Browse[2]>
debug at #3: d2 <- d^2

Browse[2]>
debug at #4: ss <- sum(d2)

Browse[2]>
debug at #5: ss

Browse[2]>
exiting from: SS(1, x)
[1] 202.5615
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - debug()

```
> SS(1,x)
debugging in: SS(1, x)
debug at #1: {
    d <- x - mu
    d2 <- d^2
    ss <- sum(d2)
    return(ss)
}

Browse[2]> n
debug at #2: d <- x - mu

Browse[2]> n
debug at #3: d2 <- d^2

Browse[2]> ls()
[1] "d"  "mu" "x"

Browse[2]> length(d)
[1] 100
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - debug()

```
Browse[2]> d[1]
[1] -1.502192

Browse[2]> n
debug at #4: ss <- sum(d2)

Browse[2]> ls()
[1] "d"  "d2" "mu" "x"

Browse[2]> hist(d2)  # Can do this on the fly

Browse[2]> n
debug at #5: ss

Browse[2]> ls()
[1] "d"  "d2" "mu" "ss" "x"

Browse[2]> length(ss)
[1] 1

Browse[2]> ss
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - debug()

```
Browse[2]> yy = x ^2   # Create a New Object

Browse[2]> yy[1]
[1] 0.2521972

Browse[2]> where      # Where are we ? Oh yea in SS
where 1: SS(1, x)

Browse[2]> c              # Continue without stopping

exiting from: SS(1, x)
[1] 202.5615

undebug(SS)      # Turns off debugging
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - debug()

Let's take a look again at our 3 functions from before. Let's debug **f** and, while we are at it, **g** and **h**.

```
f = function(x) {
  r = x - g(x)
  return(r)
}

g = function(y) {
  r = y * h(y)
  return(r)
}

h = function(x) {
  r = log(x)
  if(r < 10)
     return(r^2)
  else
     return(r^3)
}
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - debug()

```
> debug(f)

> f(-1)
debugging in: f(-1)
debug at #1: {
    r = x - g(x)
    return(r)
}

Browse[2]> where   # Tells us where we are
where 1: f(-1)

Browse[2]> n

debug at #2: r = x - g(x)

Browse[2]> debug(g)  # Let's "step" into g now
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - debug()

```
Browse[2]> n
debugging in: g(x)
debug at #1: {
    r = y * h(y)
    return(r)
}
Browse[3]> where   # Note we are now in function g
where 1 at #2: g(x)
where 2: f(-1)

Browse[3]> n
debug at #2: r = y * h(y)

Browse[3]> debug(h)  # Now let's "step" into function h
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - debug()

```
Browse[3]> n
debugging in: h(y)
debug at #1: {
    r = log(x)
    if (r < 10)
        return(r^2)
    else return(r^3)
}
Browse[4]> where
where 1 at #2: h(y)
where 2 at #2: g(x)
where 3: f(-1)

Browse[4]> ls()
[1] "x"
Browse[4]> x
[1] -1                  # So we know there is a problem here.
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - debug()

```
Browse[4]> x = 10  # Let's change the -1 to 10

Browse[4]> c
exiting from: h(y) # Now we leave h

debug at #3: r
Browse[3]> c
exiting from: g(x) # Now we leave g

debug at #3: r

Browse[2]> c
[1] 4.301898       # Disaster averted !
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - trace()

The trace function is very useful for making minor modifications to functions "on the fly" without having to modify functions and re-sourcing them.

It is especially useful if you need to track down an error which occurs in a base function. Since base functions cannot be edited by the user, trace may be the only option available for making modifications.

Note that trace has an extensive help page which should be read in its entirety. We will try to summarize the highlights here. The example we will use here is fitting a point process model via maximum likelihood.

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - trace()

When using browser, you can only browse the environment in the current function call.

You cannot poke around in the environments for previous function calls.

There may be a situation where you want to suspend execution of a function in one location, but then browse a previous function call to hunt down the bug.

In other words, you may want to "jump up" to a higher level in the function call stack.

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - trace()

The recover function can help you in this situation. Let us go back to the three functions **f, g**, and **h** defined previously.

Recall that f calls g, which in turn calls h. The h function
has a potential problem because it takes the log of a number then compares the result to another number (in this case 10).

If log returns NaN, then h will suffer a fatal error. The statements in the function body of h can be listed using the trick mentioned previously:

Debugging - trace()

The recover function can help you in this situation. Let us go back to the three functions **f, g,** and **h** defined previously.

Recall that f calls g, which in turn calls h. The h function
has a potential problem because it takes the log of a number then compares the result to another number (in this case 10).

If log returns NaN, then h will suffer a fatal error.

```
as.list(body(h))
[[1]]
`{`

[[2]]
r <- log(z)

[[3]]
if (r < 10) r^2 else r^3
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - trace()

```
as.list(body(h))
[[1]]
`{`

[[2]]
r <- log(z)

[[3]]
if (r < 10) return(r^2) else return(r^3)
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - trace()

```
> trace("h", quote( if(is.nan(r)) { recover() } ), at = 3, print = F)
[1] "h"

> body(h)
{
    r <- log(z)
    {
        .doTrace(if (is.nan(r)) {
            recover()
        })
        if (r < 10)
            return(r^2)
        else return(r^3)
    }
}
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - trace()

```
> f(23)
[1] -203.1205
> f(-10)

Enter a frame number, or 0 to exit

1: f(-10)
2: #2: g(x)
3: #2: h(y)

Selection: 1
Called from: .doTrace(if (is.nan(r)) {
    recover()
})

Browse[1]> ls()
[1] "x"
Warning message:
In log(z) : NaNs produced

Browse[1]> x
[1] -10

Browse[1]> c
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - trace()

```
Enter a frame number, or 0 to exit

1: f(-10)
2: #2: g(x)
3: #2: h(y)

Selection: 2
Called from: eval.parent(exprObj)

Browse[1]> ls()
[1] "y"

Browse[1]> y
[1] -10

Browse[1]> c
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging - trace()

```
Enter a frame number, or 0 to exit

1: f(-10)
2: #2: g(x)
3: #2: h(y)

Selection: 3
Called from: eval(expr, p)

Browse[1]> ls()
[1] "r" "z"

Browse[1]> r
[1] NaN

Browse[1]> z
[1] -10

Browse[1]> c

Enter a frame number, or 0 to exit

Selection: 0
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
```

Debugging - trace()

Once you are done with the tracing the function its time to "untrace" it. This restores the function to its original state.

```
> untrace("h")
```

Debugging

A More Involved Example

Debugging

First we will simulate a point process over [0; 1] according to the given model:

```
> set.seed(100)
> p <- sort(runif(200))
> thin <- rbinom(200, 1, p)
> pp <- p[thin == 1]
> hist(pp, nclass = 20)
```

Now we write out the negative log-likelihood function:

```
nLL <- function(mu, x) {
     z <- mu * x
     lz <- log(z)
     L1 <- sum(lz)
     L2 <- mu/2
     LL <- -(L1 - L2)
     LL
}
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging

In order to minimize the negative log-likelihood, we will have to pass nLL to an optimization routine. In R, there are two such routines: nlm and optim. Here we will use optim, which allows the user to choose from a variety of different optimization procedures.

optim requires a starting value, the function to be optimized, the method for optimization and other arguments that are necessary for the function to be optimized (typically data).

In this example, we use a starting value (albeit not a wise one) of 100; 000 and a quasi-Newton optimization procedure due to Broyden, Fletcher, Goldfarb, and Shanno.

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging

First we will simulate a point process over [0; 1] according to the given model:

```
set.seed(100)
p <- sort(runif(200))
thin <- rbinom(200, 1, p)
pp <- p[thin == 1]
hist(pp, nclass = 20)
```

Now we write out the negative log-likelihood function:

```
nLL <- function(mu, x) {
     z <- mu * x
     lz <- log(z)
     L1 <- sum(lz)
     L2 <- mu/2
     LL <- -(L1 - L2)
     LL
}
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging

```
> optim(100000, nLL, method = "BFGS", x = pp)
$par
[1] 206.0278

$value
[1] -397.9274

$counts
function gradient
      40       19

$convergence
[1] 0

$message
NULL

There were 19 warnings (use warnings() to see them)
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging

Here we see the optimizer does converge on the value of 206.0278
However, there were warnings set off during the optimization and we can use warnings to view them.

```
> warnings()
Warning messages:
1: In log(z) : NaNs produced
2: In log(z) : NaNs produced
..
19: In log(z) : NaNs produced
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging

All of the warnings come from the log function which is of course used in the negative log-likelihood function.

In this situation, using debug would be problematic because as the original optim output shows, the optimizer makes 40 calls to the nLL function.

However, there were only 19 warnings, so some of the calls to nLL were fine and did not cause a warning.

It would be tedious to have to step through the nLL function line by line 40 times. In larger optimization problems this could be hundreds of function calls and stepping through each one would take forever.

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging

We would like only to suspend execution when it looks like something might be wrong. In this example, we see that the log function is producing NaN's.

Why don't we just suspend execution when the log function has produced an NaN? We can use trace to do exactly this.

In the nLL function, there is the line lz <- log(z)

What we want to do is insert some code after this line which essentially implements the following logic:

If lz contains an NaN's, suspend execution and let me browse the
the environment to see what went wrong." We can do this with the following call to trace:

`An Introduction to the Interactive Debugging Tools in R - Roger Peng`
`http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf`

Debugging

```
> trace("nLL", quote(if(any(is.nan(lz))) { browser() }), at=4, print=F)
```

The first argument to trace is the name of a function.

The second argument is the code you want to insert. This can either be the name of a function or it can be an unevaulated expression.
In this example we have chosen to use an unevaluated expression.

The expression we have inserted into nLL is the following conditional statement:

```
if(any(is.nan(lz))) {
     browser()
}
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging

```
trace("nLL", quote(if(any(is.nan(lz))) { browser() }), at=4, print=F)

if(any(is.nan(lz))) {
    browser()
}
```

The code we've decided to insert simply invokes the browser function (see Section 3.2.1) if any elements of lz have the value NaN.

This expression has to be put into the quote function so that R does not evaluate the code, rather it simply inserts the statements in to the nLL function.

Without the quote function, R would try to evaluate the if statement within the trace function and it wouldn't make any sense.

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging

```
trace("nLL", quote(if(any(is.nan(lz))) { browser() }), at=4, print=F)

if(any(is.nan(lz))) {
    browser()
}
```

The "at" argument tells trace were to insert the new code. Here we've instructed trace to insert the code before the fourth statement.

There's no need to worry about counting statements in your function. This can be done with the following trick.

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging

```
> as.list(body(nLL))
[[1]]
`{`

[[2]]
z <- mu * x

[[3]]
lz <- log(z)

[[4]]
L1 <- sum(lz)

[[5]]
L2 <- mu/2

[[6]]
LL <- -(L1 - L2)

[[7]]
LL
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging

```
> trace("nLL", quote(if(any(is.nan(lz))) { browser() }), at=4, print=F)
[1] "nLL"

> nLL
Object with tracing code, class "functionWithTrace"
Original definition:
function(mu, x) {
z <- mu * x
lz <- log(z)
L1 <- sum(lz)
L2 <- mu/2
LL <- -(L1 - L2)
LL
}

## (to see the tracing code, look at body(object))
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging

```
> body(nLL)
{
    z <- mu * x
    lz <- log(z)
    {
        .doTrace(if (any(is.nan(lz))) {
            browser()
        })
        L1 <- sum(lz)
    }
    L2 <- mu/2
    LL <- -(L1 - L2)
    LL
}
```

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf

Debugging

```
> optim(100000,nLL,method="BFGS",x=pp)

Called from: eval(expr, envir, enclos)
Browse[1]> where
where 1: eval(expr, envir, enclos)
where 2: eval(expr, p)
where 3: eval.parent(exprObj)
where 4: .doTrace(if (any(is.nan(lz))) {
    browser()
})
where 5: fn(par, ...)
where 6: function (par)

Browse[1]> ls()

[1] "lz" "mu" "x"  "z"
Warning message:
In log(z) : NaNs produced

Browse[1]> str(lz)
 num [1:103] NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN ...

Browse[1]> mu
[1] -34118322

Browse[1]> Q     # So mu is negative
```

Debugging

Now we can see that the problem is the optimizer was using negative values for mu.

Since the conditional intensity of a point process can never be negative, negative values of mu are not valid.

However, the optimizer does not know this and simply chooses values based on a deterministic search algorithm.

An Introduction to the Interactive Debugging Tools in R - Roger Peng
http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf