BIOS 545 Lecture 7

Department of Biostatistics and Bioinformatics

Steve Pittard wsp@emory.edu

January 10, 2023

Scoping

Some hints when writing functions

- Keep functions short. If your function is long then split it up into different functions
- Functions can call other functions
- Keeping functions separate makes it easy to identify possible problems
- Try your function with a large variety of data to find possible problems
- Use if statements up front to catch bad input values
- Put comments in your code so you and others who read your code will know what you are trying to do

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

Scoping

It is important to understand scoping rules when writing functions.

- 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`

Scoping

- 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`

Scoping

The **environment** command shows you the current environment. Normally you don't worry about this although it is important to understand since it is linked to scope.

```
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

Scoping

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

```
environment()
<environment: R_GlobalEnv>

myenvfunc <- function() {
   print(environment())
}
# See what happens when we call the environment command from within
# a function

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

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

Scoping

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
```

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.

Scoping

```
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
```

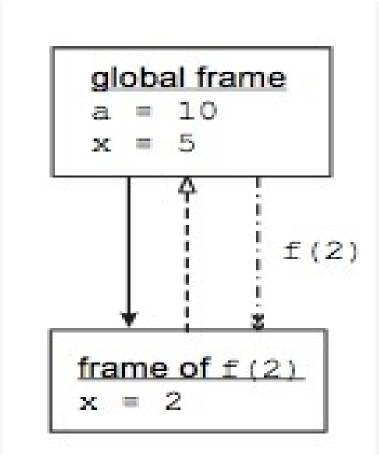
When f is called it passes the value of 2. So "x" assumes a local value of 2. Then the function wants to add $x = 2$ to the value of a though non has been specified so the function results in an error. This seems reasonable since R can't find a variable called "a" anywhere.

Scoping

```
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.

Scoping

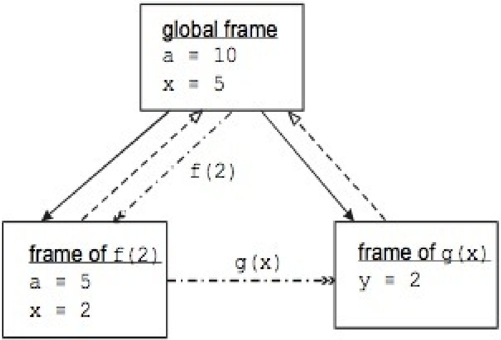


Scoping

```
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.

Scoping

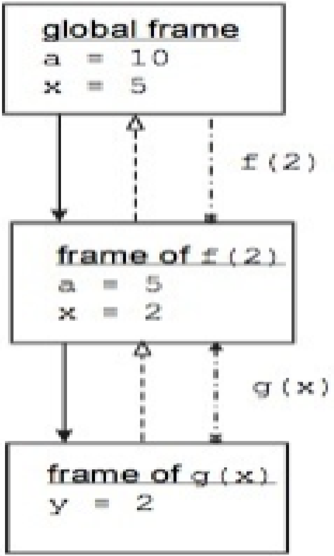


Scoping

```
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 function f 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

Scoping



Scoping

What does this all mean ?

- 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

Scoping

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)
}
```

Scoping

Functions can always call other functions that exist within the same environment. It happens all the time and is one of the more common activities in R

```
exampf <- function(myvec) {

   retval <- c(mean(myvec), sd(myvec))  # mean and sd are known

   return(retval)
}

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

Scoping

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
```

Scoping

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

  }
}
```

Scoping

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

Scoping

```
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
  }
}
```

Arguments

"a" and "b" represent arguments that correspond to sides a and b, respectively. We compute "c" the hypoteneuse and return its value

```
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)
}
```

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
```

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

Arguments

The mean function has three arguments:

- x: a value or vector to take the mean of
- trim: a value that let's you trim the vector by
- na.rm: 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"

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

Arguments

We could explicitly name the arguments as we provide them. This way, R will never be confused about what value corresponds to what argument

```
set.seed(1)
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

# BUT THE FOLLOWING WON'T WORK !

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
```

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 for anyone to reasonably remember (well most people anyway)
- There is no convenient way to remember the arguments by position unless you have a photographic memory.

Arguments

```
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")
```

Arguments - The "..." 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
```

Arguments - The "..." argument

- 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. You want to hard code in the color red for all plots. That is you want to force the color red on anyone who uses your function (not nice but this is just an example)

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

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

Arguments - The "..." argument

Suppose you write a small function to do some plotting. You want to hard code in the color red for all plots. You want to force the color red on anyone who uses your function (not nice but this is just an example)

```
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.

Arguments - 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

Note that many of the examples presented in this section come from “An Introduction to the Interactive Debugging Tools in R” by Roger Peng. The associated website can be found at:

`http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf`

Debugging

Ever wonder where the terms “bug” and “debugging” came from ?

Grace Murray Hopper

Rear Admiral Grace M. Hopper, USN, Ph.D.

Debugging

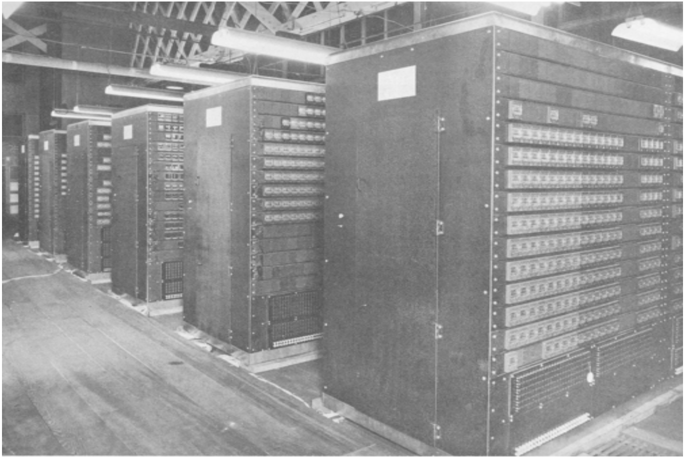
- Computer scientist and US Navy rear admiral
- Invented the first compiler for a computer programming language
- Invented COBOL, one of the first high-level programming languages
- In 1947, while working on a new Harvard Mark computer, she discovered a moth stuck in an electrical relay which caused problems
- Since that time people call any problem relating to unexpected program behavior a "bug"

Debugging

This is the Harvard Mark 1 Computer

Debugging

These are the relay banks for the Mark Computer. She looked through banks like these and found a moth stuck in one of them



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 in 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

Most of the trouble comes from people providing bad input to your function. So if you do some initial error checking on the arguments then you will save yourself a lot of trouble.

```
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

Most of the trouble comes from people providing bad input to your function. So if you do some initial error checking on the arguments then you will save yourself a lot of trouble.

```
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

Some people create an argument called debug and use it as follows:

```
mymedian <- function(xvec, debug=TRUE) {

  veclength  <- length(xvec)   # Get vector length
  sortedvec  <- sort(xvec)     # Get sorted vector

  if (debug) {
    cat("veclength is",veclength,"sortedvec is",sortedvec,"\n")
  }
  if (veclength %% 2 != 0) {
    index  <- ceiling(veclength/2)
    retval <- sortedvec[index]

    if (debug) {
      cat("index is",index,"retval is",retval,"\n")
    }

  } else {

    # Other code
  }
```

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

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

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

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
```

Debugging

```
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
```

Debugging

```
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)
}
```

Debugging

A powerful tool is the **debug** function that allows us to step through a function as it is being executed.

```
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
```

Debugging

```
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]>
```

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).

Browse[2]>
```

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
```

Debugging

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

Debugging

```
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
```

Debugging

```
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
```

Debugging

```
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
```

Debugging

```
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
```

Debugging

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)
}
```

Debugging

```
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
```

Debugging

```
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
```

Debugging

```
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
```

Debugging

```
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 !
```

Debugging

- The trace function is very useful for making minor modifications to functions "on the fly".
- 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.

Debugging

- 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.

Debugging

- 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 this approach:

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

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

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

Debugging

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

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

Debugging

```
trace("h", quote( if(is.nan(r)) { recover() } ), at = 3, 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 unevaluated expression
- The "at" argument tells trace were to insert the new code. Here we've instructed trace to insert the code before the third statement

Debugging

```
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
```

Debugging

```
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
```

Debugging

```
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" "x"

Browse[1]> r
[1] NaN

Browse[1]> x
[1] -10

Browse[1]> c

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

Debugging

```
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" "x"

Browse[1]> r
[1] NaN

Browse[1]> x
[1] -10

Browse[1]> c

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

Debugging

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

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