1) **Recommended strategy:** Make changes to “functionName.R” files and use
source("functionName.R") and/or run code from the R editor to keep R up to date.

2) **Alternate strategy:** Use `fix(functionName)` to change functions, then use, e.g.,
dump("functionName", "functionName.R") to save a copy.

3) `traceback()`: Use this *immediately* after seeing an error message to get the chain of commands
and function invocations that led to the error, with the one actually causing the problem at (or
near) the top.

Example:
```r
autoLegend = function(strVec) {
  n=length(strVec);
  legend("topleft", strVec, pch=1:n, col=1:n, lty=1:n)
}
autoPlot = function(n, legendString=NULL) {
  plot(rnorm(n), rnorm(n))
  autoLegend(legendString)
  invisible(NULL)
}
autoPlot(10)
# Error in as.graphicsAnnot(legend) :
#   argument "legend" is missing, with no default
traceback()
#4: as.graphicsAnnot(legend)
#3: legend("topleft", strVec, pch = 1:n, col = 1:n, lty = 1:n) at #3
#2: autoLegend(legendString) at #3
#1: autoPlot(10)
args(legend)
# function (x, y = NULL, legend, fill = NULL, etc.)
```
4) The browser
   a. The browser is a mode of R in which you can single step through a function and examine and change variables in the current or other frames.
   b. Invoked by browser(), debug(), debugonce(), and trace()
   c. Prompt is usually “Browse[2]” at first. If you debug deeper into functions, the number increases, and helps keep track of how deep you are.
   d. Here are the browser commands (these special commands do not use parentheses)
      i. Q: quit browsing (terminates the function)
      ii. c: continue to the end of the function or {} group at the current depth
      iii. n: go to the next line of the function (executes the current line and displays the next line); “line” is used loosely here
      iv. where: display the “stack” (chain) of functions called with the current function in position 1
   v. Use up-arrow to recall previous commands
   vi. Ordinary R code (usually with parentheses)
      1. sys.call(): same as first line of “where”
      2. ? or ??: get R help
      3. ls(): show variable names in the current function
      4. show variables of the calling frame:
         ls(envir=sys.frames()[[FRAME.NUM]]) where FRAME_NUM is counted upwards from the bottom of the “where” result
      5. x or print(x): show variable value (including functions)
      6. print(n): show value of n (cannot use just “n”)
      7. x=3: set x to 3
      8. f(x): evaluate a function
      9. get("x", envir=.GlobalEnv): get a top-level variable
     10. get("x", envir= sys.frames()[[FRAME_NUM]]): get a variable in a calling frame
     11. assign("x",value, envir= sys.frames()[[FRAME_NUM]]): change a variable in a calling frame (somewhat dangerous)
     12. Note that “n” when the next line is an error stops the browser, but executing that same line, e.g., using cut and paste, shows the error without stopping the debugger.

5) browser() statement(s)
   a. Place browser() anywhere in your code to cause R to “drop into the browser”
   b. Often used as “if (someCondition) browser()” or “browser(expr=someCondition)” Can use “browser(text='some text', expr='x<5')” so that you enter the browser only if x<5, and/or you can use “browserText()” to identify which browser() command put you into the browser.
6) \texttt{debug(myFunction), undebug(myFunction), debugonce(myFunction)}

\begin{itemize}
\item Causes R to drop into browser mode whenever \texttt{“myFunction”} is invoked
\item \texttt{isdebugged(myFunction)} will test if your function is set to be debugged. Note the absence of the period. It’s not easy to show all debugged functions. One way is:
\begin{verbatim}
sapply(ls(), function(x) {x=get(x);
   ifelse(mode(x)==”Function”,isdebugged(x),FALSE))}
\end{verbatim}
\item Unless you use \texttt{debugonce()}, functions stay in the debugged state until either you use \texttt{undebug()}, or you re-define the function, or you quit R.
\item Note that you can use \texttt{debug(anotherFunction)} inside the browser.
\end{itemize}

7) For debugging warnings, it may help to use \texttt{“options(warn=2)”} to turn warnings into errors, then reset \texttt{“options(warn=0)”} when you are done.

8) \texttt{trace(myFunction)} and \texttt{untrace(myFunction)}

\begin{itemize}
\item \texttt{trace()} differs from \texttt{debug()} in being more flexible. It makes various temporary changes to the function that you control with its arguments. The \texttt{untrace(myFunction)} always restores your function back to its original version.
\item In its simplest form, \texttt{“trace(myFunction)”}, prints a message with the function call (including arguments) every time \texttt{myFunction} is executed. This continues until you enter \texttt{untrace(myFunction)} or quit R.
\item Tracing can be temporarily suspended with \texttt{“tracingState(on=FALSE)”} and turned back on with \texttt{“tracingState(on=TRUE)”}.
\item Using \texttt{“trace(myFunction, tracer=browser)”} is the same as \texttt{“debug(myFunction)”} because it says to run \texttt{browser()} when \texttt{myFunction()} starts. Like some other forms of \texttt{trace()}, \texttt{trace(tracer=)} modifies the function. If you try to examine the function, you get something like this:
\begin{verbatim}
Object with tracing code, class ”functionWithTrace”
Original:
function(x,y)rnorm(x)
## (to see the tracing code, look at body(object))
\end{verbatim}
\item Using \texttt{“trace(myFunction, exit=browser)”} starts the browser just before exiting the function each time. You can also use \texttt{tracer=recover} or \texttt{exit=recover} for a bit more flexibility (see below).
\item Use \texttt{as.list(body(myFunction))} to see “line numbers” for your function. Then use, e.g., \texttt{“trace(myFunction, trace=browser, at=4)”} to cause your function to drop into the browser just before executing line 4.
\item Use \texttt{“trace(myFunction, edit=TRUE)”} to make any (temporary) changes to the function, including adding \texttt{browser()} statements.
\item Note that changes to functions caused by \texttt{trace()} are carried over between sessions if you save your workspace when you quit.
\end{itemize}
i. Note that `trace(edit=TRUE)` can be used to modify package functions, e.g., to override a fixed value that you don’t like or to add a component to a return value. There is no other reasonable way to do this on a package that you did not write.

9) **Post-mortem debugging with `recover()`**
   a. `recover()` is a more flexible version of `browser()`.
   b. `recover()` can be put into a function instead of `browser()` or can be used as the tracer or exit function for `trace()`, but without entering the parentheses.
   c. In recover mode, you are given a menu of the stack of frames leading to the current statement, and you can choose to enter any frame to examine variables in that frame. Then type “c” to return to the menu. Finally choose “o” to exit from the recovery state.
   d. The most important use of `recover()`, is this: when you get an error, type “`options(error=recover)`” and re-run your program. You will be put into recover mode at the point of the error. You will be able to see the command that caused the error and the values of all of your variables at that point in time. Remember to type “`options(error=NULL)`” to return to normal error processing.