CT5102: Programming for Data Analytics

Week 12: Debugging Code and Performance

https://github.com/JimDuggan/CT5102

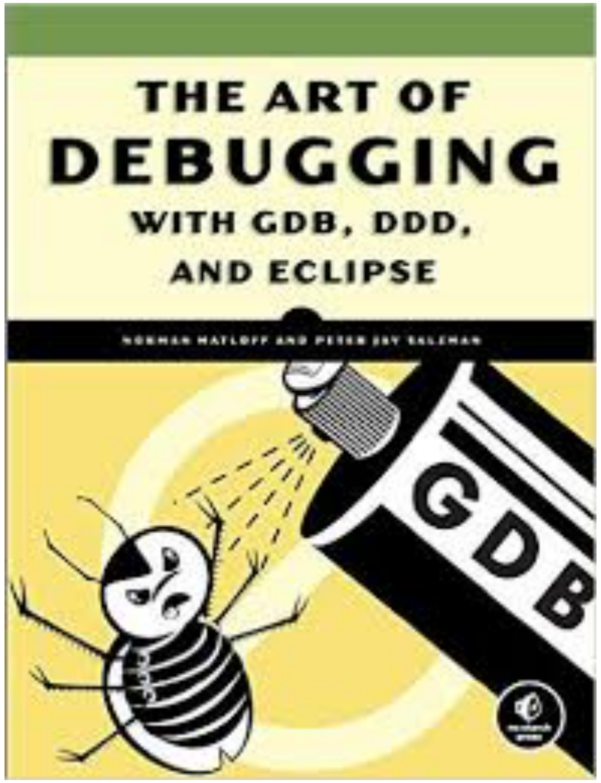
Dr. Jim Duggan,

Information Technology,

School of Engineering & Informatics

(1) Debugging
(Salzman and Matloff 2008)

- “Fixing a buggy program is a process of confirming, one by one, that the many things you *believe* to be true about the code actually *are* true.
- When you find one of your assumptions is *not* true, you have found a clue to the location (if not the exact nature) of a bug.”



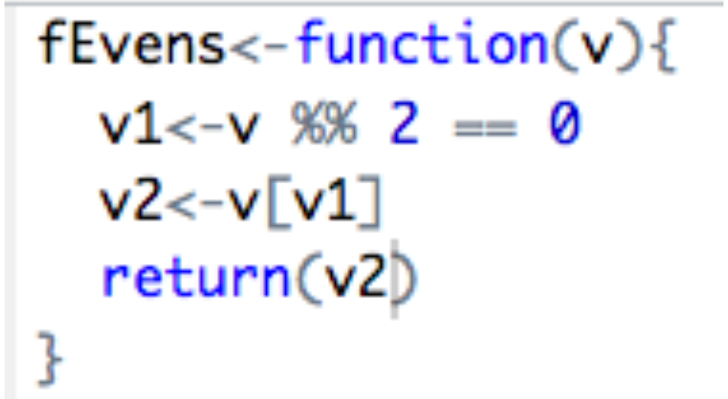
Debugging in R

- The core of R’s debugging facility consists of the *browser*.
- This can be invoked in two ways:
  - debug(f) – called at a function level
  - browser() – invoked at a particular line of code
  - debugonce(f)

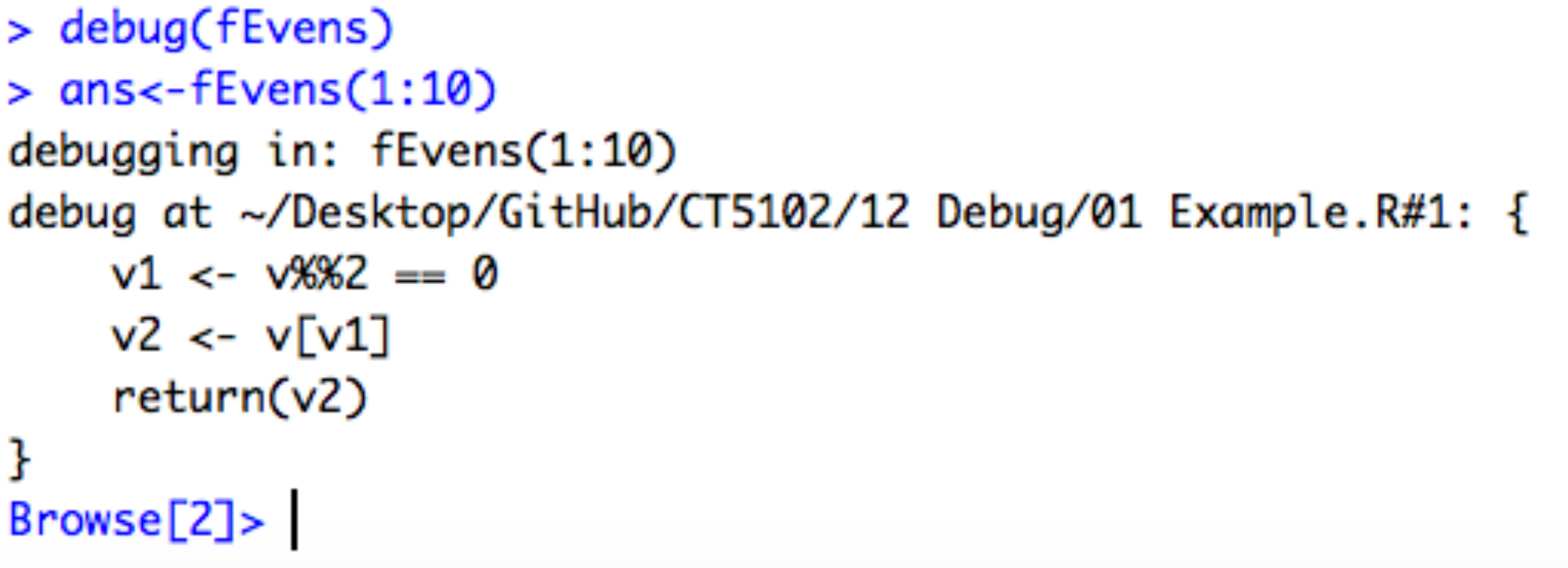
Using Browser Commands (Matloff 2011)

Command	Description
n (for next)	Tells R to execute the next line and pause again. Hitting enter also causes this action.
c (for continue)	Similar to n, except that several lines of code may be executed. In a loop, the remainder of the loop will be executed.
Any R command	Query variable values. If a variable has the same name as a browser command, the function print(n) should be used.
where	This prints a stack trace. It displays what sequence of function calls led to the current location
Q	Quits the browser

Using debug()

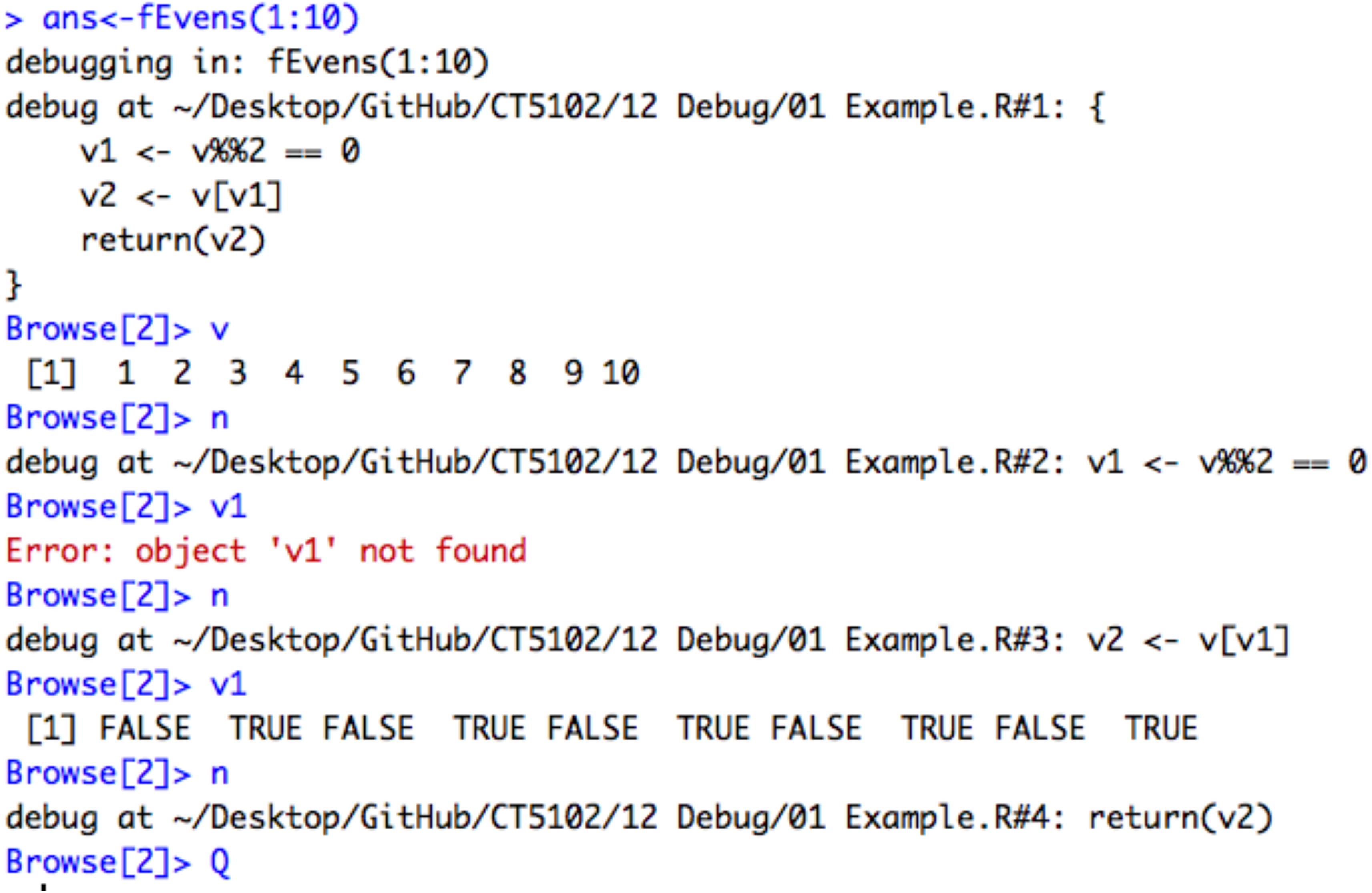


```
fEvens<-function(v){
  v1<-v %% 2 == 0
  v2<-v[v1]
  return(v2)
}
```



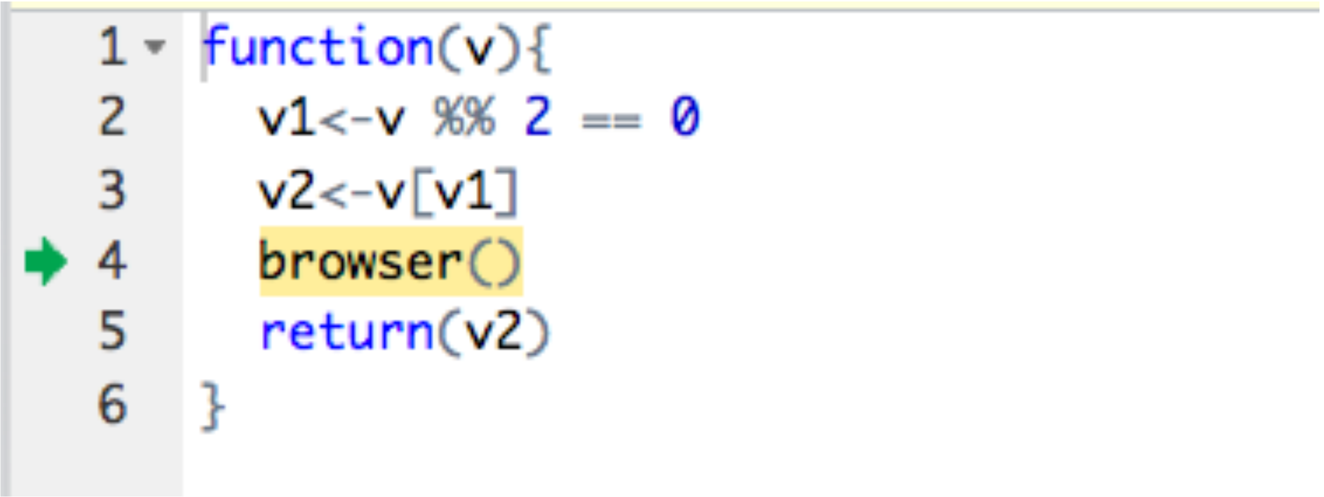
```
> debug(fEvens)
> ans<-fEvens(1:10)
debugging in: fEvens(1:10)
debug at ~/Desktop/GitHub/CT5102/12 Debug/01 Example.R#1: {
    v1 <- v%%2 == 0
    v2 <- v[v1]
    return(v2)
}
Browse[2]> 
```

Sample Output

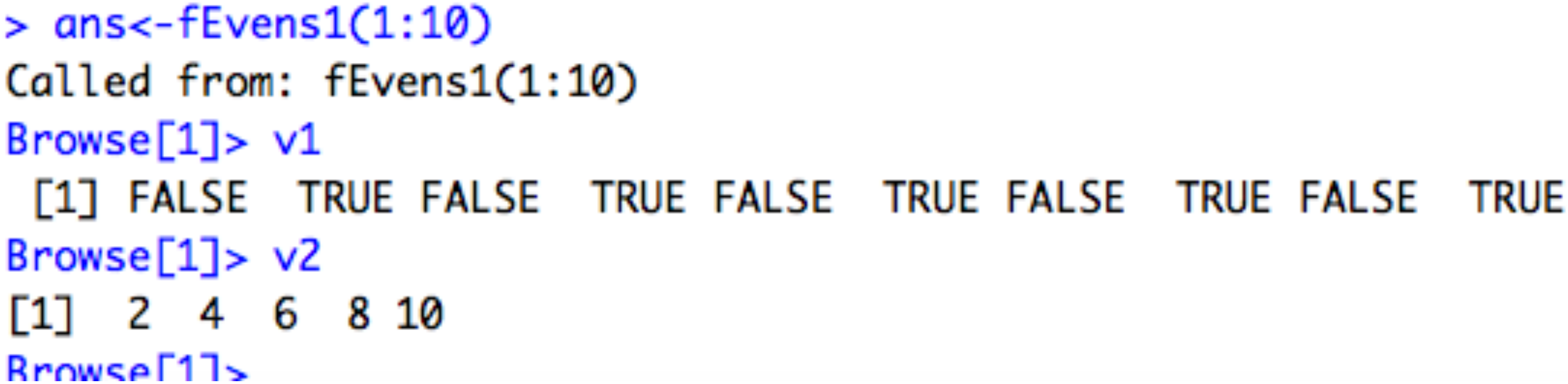


```
> ans<-fEvens(1:10)
debugging in: fEvens(1:10)
debug at ~/Desktop/GitHub/CT5102/12 Debug/01 Example.R#1: {
    v1 <- v%%2 == 0
    v2 <- v[v1]
    return(v2)
}
Browse[2]> v
 [1]  1  2  3  4  5  6  7  8  9 10
Browse[2]> n
debug at ~/Desktop/GitHub/CT5102/12 Debug/01 Example.R#2: v1 <- v%%2 == 0
Browse[2]> v1
Error: object 'v1' not found
Browse[2]> n
debug at ~/Desktop/GitHub/CT5102/12 Debug/01 Example.R#3: v2 <- v[v1]
Browse[2]> v1
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
Browse[2]> n
debug at ~/Desktop/GitHub/CT5102/12 Debug/01 Example.R#4: return(v2)
Browse[2]> Q
```

Inserting call to browser()



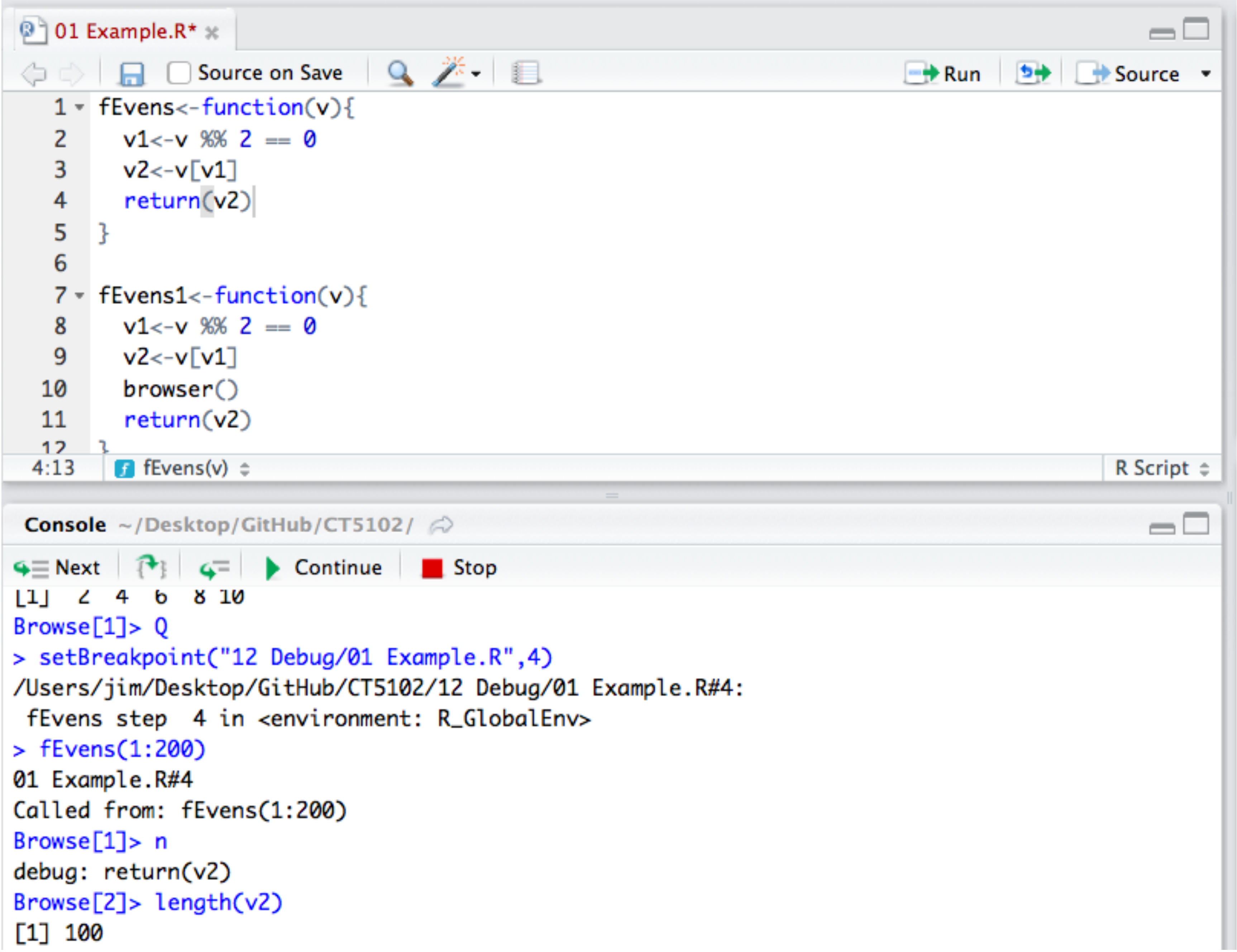
```r
function(v){
  v1<-v %% 2 == 0
  v2<-v[v1]
  browser()
  return(v2)
}
```



```
> ans<-fEvens1(1:10)
Called from: fEvens1(1:10)
Browse[1]> v1
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
Browse[1]> v2
[1]  2  4  6  8 10
Browse[1]>
```

setBreakpoint() function

- Format is:
  - setBreakpoint(*filename, linenumber*)
- This will result in browser() being called at line *linenumber* in our source file
- Calling untrace(f) will remove the breakpoint
- Also works within the browser



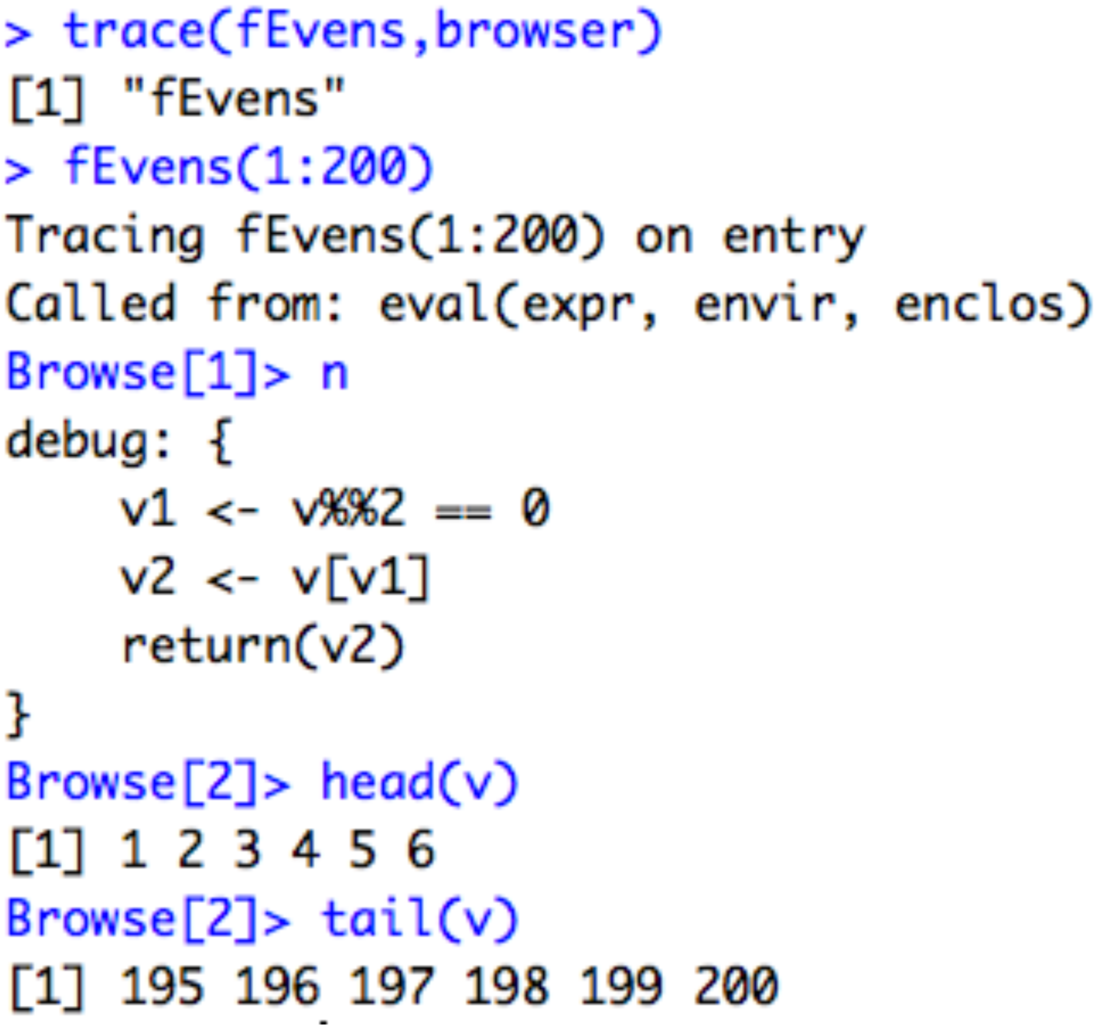
```r
fEvens<-function(v){
  v1<-v %% 2 == 0
  v2<-v[v1]
  return(v2)
}

fEvens1<-function(v){
  v1<-v %% 2 == 0
  v2<-v[v1]
  browser()
  return(v2)
}
```

```
[1]  2  4  6  8 10
Browse[1]> Q
> setBreakpoint("12 Debug/01 Example.R",4)
/Users/jim/Desktop/GitHub/CT5102/12 Debug/01 Example.R#4:
 fEvens step  4 in <environment: R_GlobalEnv>
> fEvens(1:200)
01 Example.R#4
Called from: fEvens(1:200)
Browse[1]> n
debug: return(v2)
Browse[2]> length(v2)
[1] 100
```

trace() function

- Flexible and powerful
- Format trace(f,t)
  - Instructs R to call the function t() every time we enter the function f()

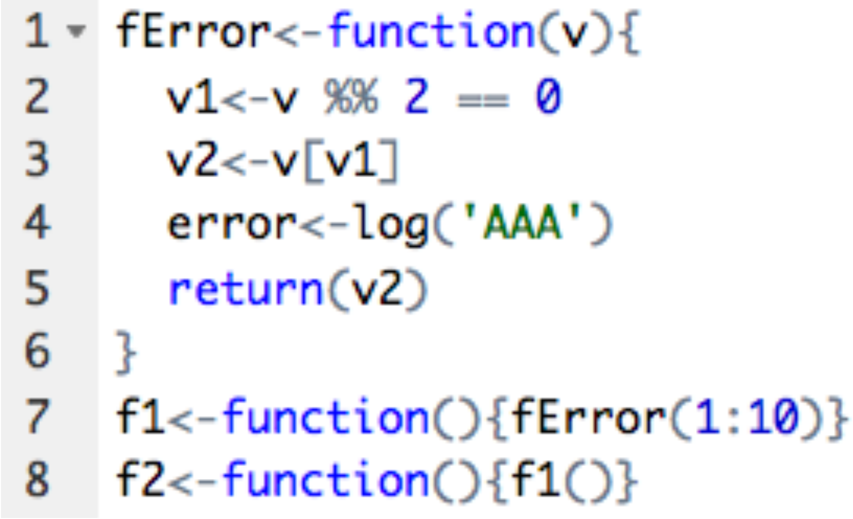
trace() example



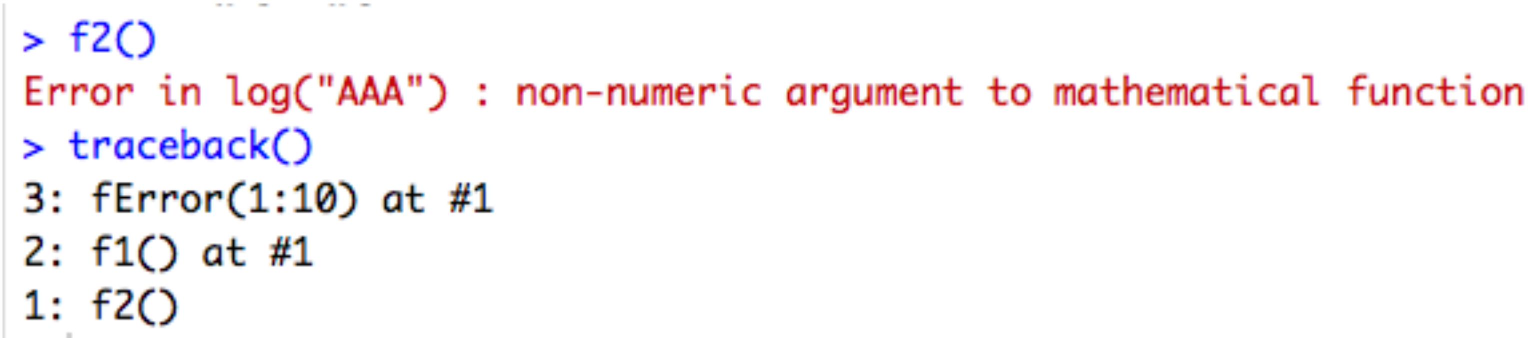
```
> trace(fEvens,browser)
[1] "fEvens"
> fEvens(1:200)
Tracing fEvens(1:200) on entry
Called from: eval(expr, envir, enclos)
Browse[1]> n
debug: {
    v1 <- v%%2 == 0
    v2 <- v[v1]
    return(v2)
}
Browse[2]> head(v)
[1] 1 2 3 4 5 6
Browse[2]> tail(v)
[1] 195 196 197 198 199 200
```

traceback()

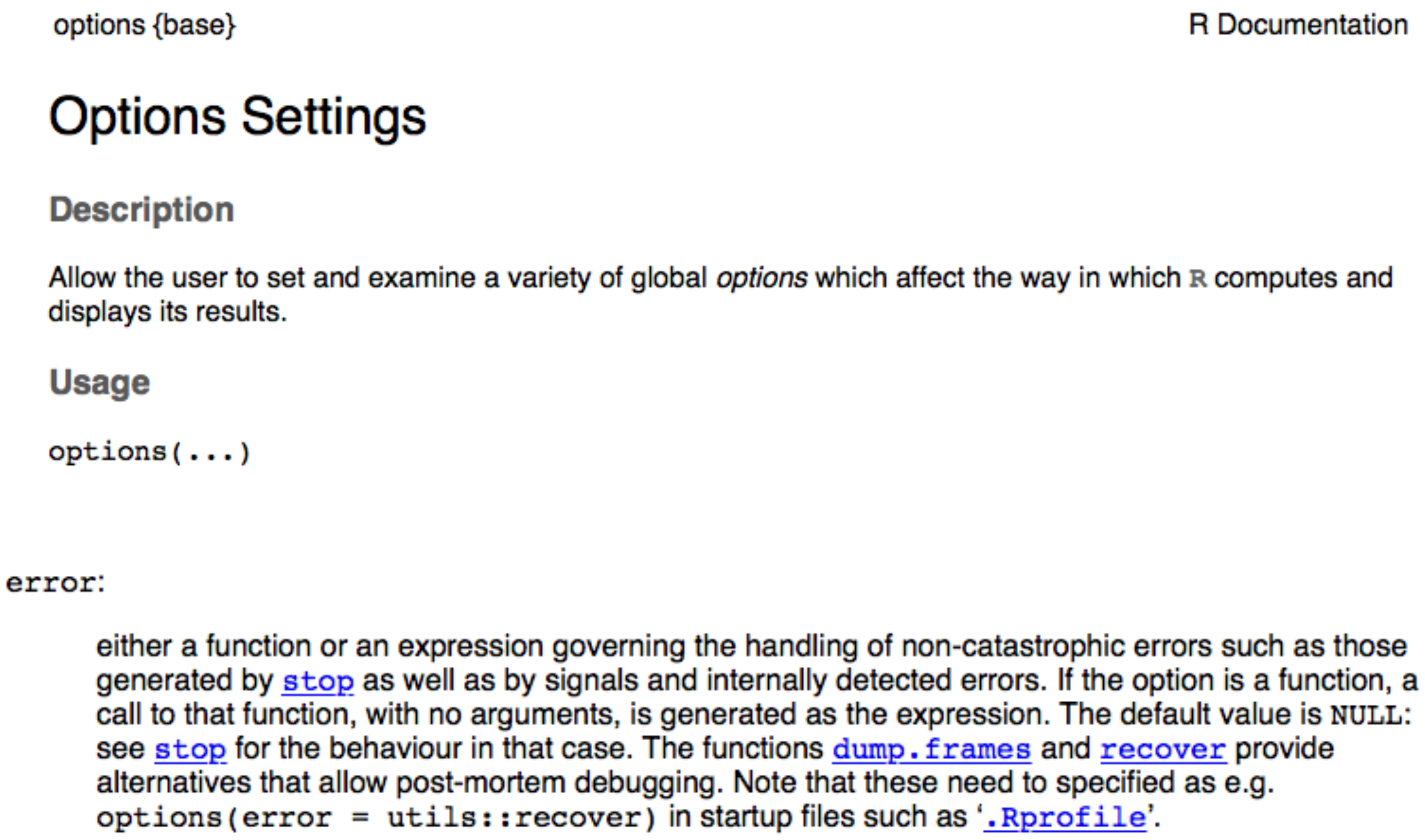
- If the R code crashes, a call to traceback() will inform you what function the problem occurred and the call chain that led to that function.



```r
fError<-function(v){
  v1<-v %% 2 == 0
  v2<-v[v1]
  error<-log('AAA')
  return(v2)
}
f1<-function(){fError(1:10)}
f2<-function(){f1()}
```



```
> f2()
Error in log("AAA") : non-numeric argument to mathematical function
> traceback()
3: fError(1:10) at #1
2: f1() at #1
1: f2()
```



options {base} R Documentation

Options Settings

Description

Allow the user to set and examine a variety of global *options* which affect the way in which R computes and displays its results.

Usage

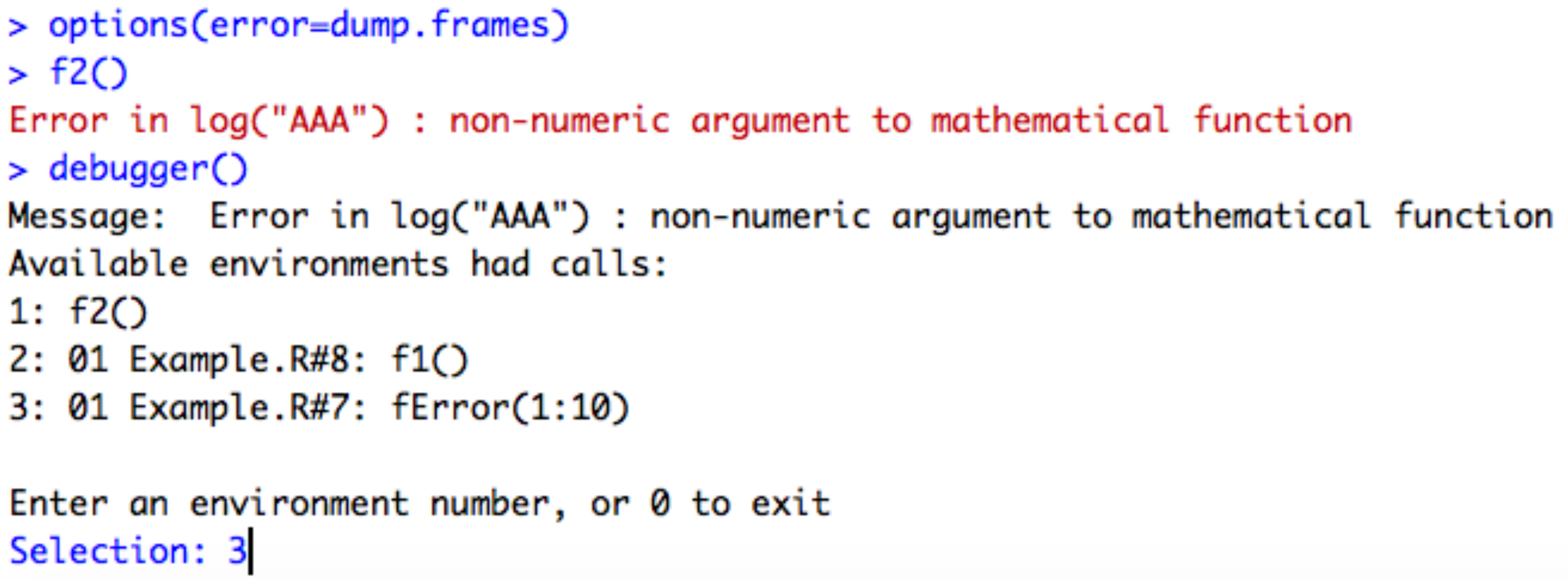
```
options(...)
```

`error`:

either a function or an expression governing the handling of non-catastrophic errors such as those generated by `stop` as well as by signals and internally detected errors. If the option is a function, a call to that function, with no arguments, is generated as the expression. The default value is `NULL`: see `stop` for the behaviour in that case. The functions `dump.frames` and `recover` provide alternatives that allow post-mortem debugging. Note that these need to specified as e.g. `options(error = utils::recover)` in startup files such as '`.Rprofile`'.

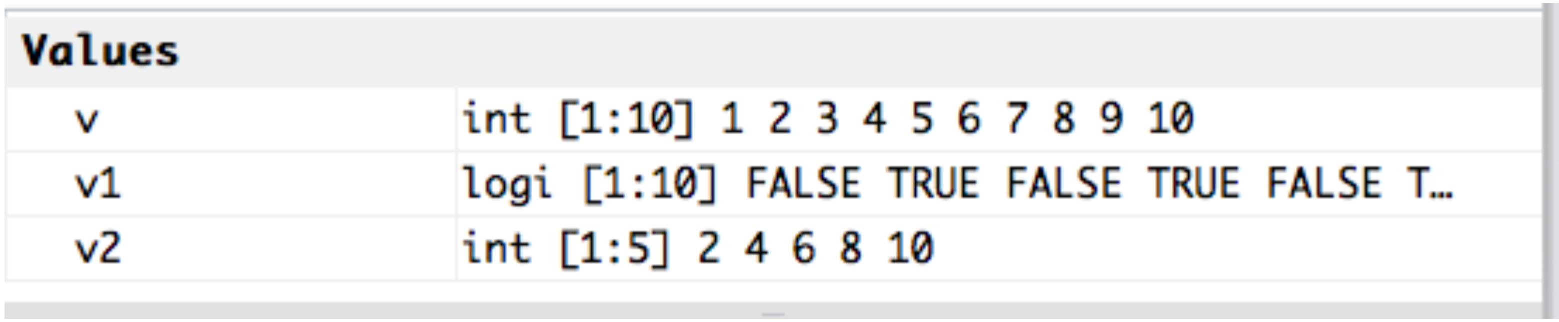
debugger()

- Provides a lot more information for a software crash
- Setup using options(error=dump.frames)

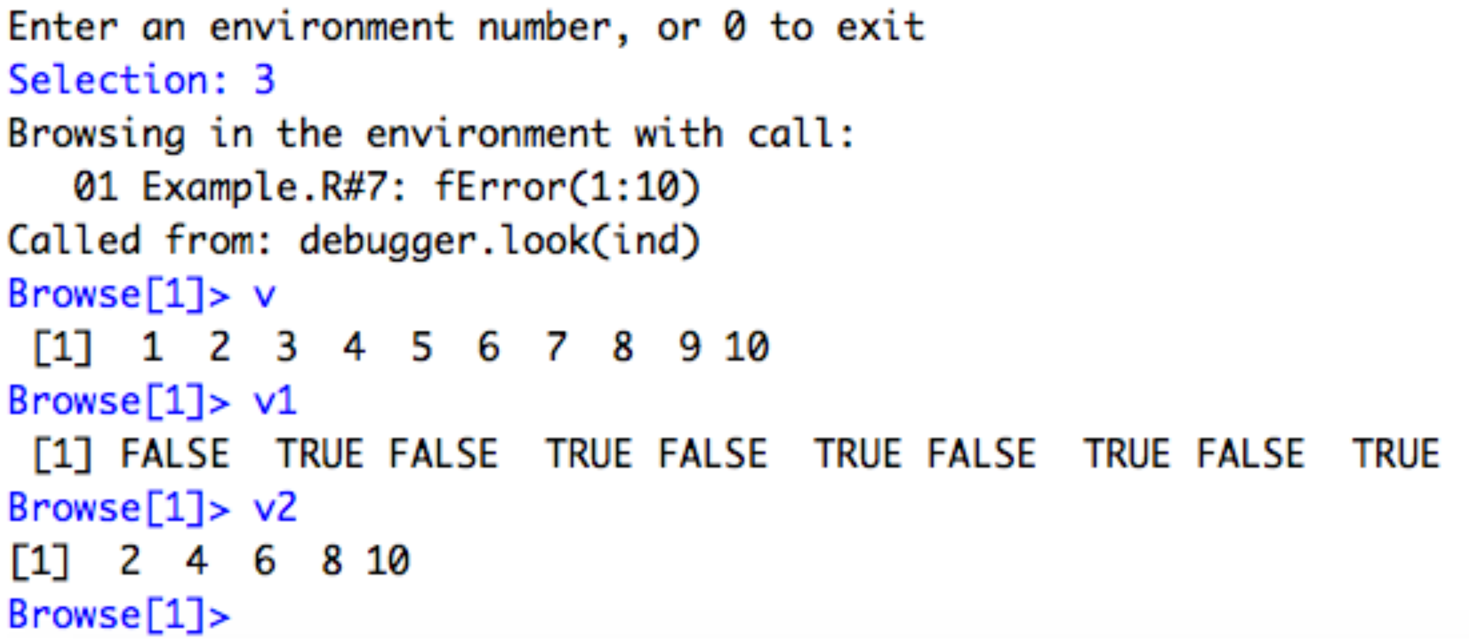


```
> options(error=dump.frames)
> f2()
Error in log("AAA") : non-numeric argument to mathematical function
> debugger()
Message:  Error in log("AAA") : non-numeric argument to mathematical function
Available environments had calls:
1: f2()
2: 01 Example.R#8: f1()
3: 01 Example.R#7: fError(1:10)

Enter an environment number, or 0 to exit
Selection: 3
```

Exploring the problematic function



Values	
v	int [1:10] 1 2 3 4 5 6 7 8 9 10
v1	logi [1:10] FALSE TRUE FALSE TRUE FALSE T…
v2	int [1:5] 2 4 6 8 10

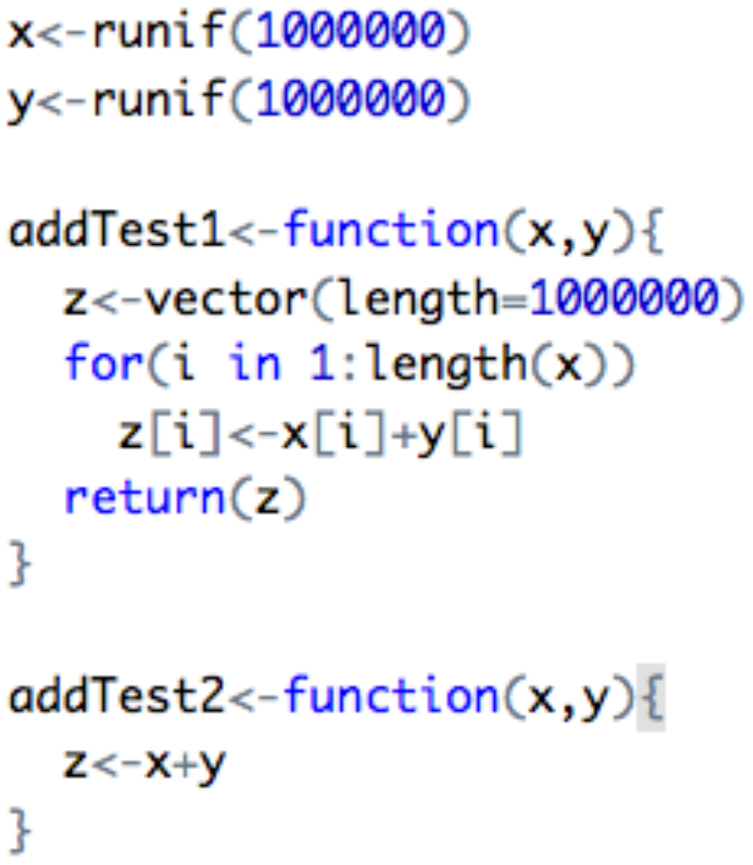


```
Enter an environment number, or 0 to exit
Selection: 3
Browsing in the environment with call:
   01 Example.R#7: fError(1:10)
Called from: debugger.look(ind)
Browse[1]> v
 [1]  1  2  3  4  5  6  7  8  9 10
Browse[1]> v1
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
Browse[1]> v2
[1]  2  4  6  8 10
Browse[1]>
```

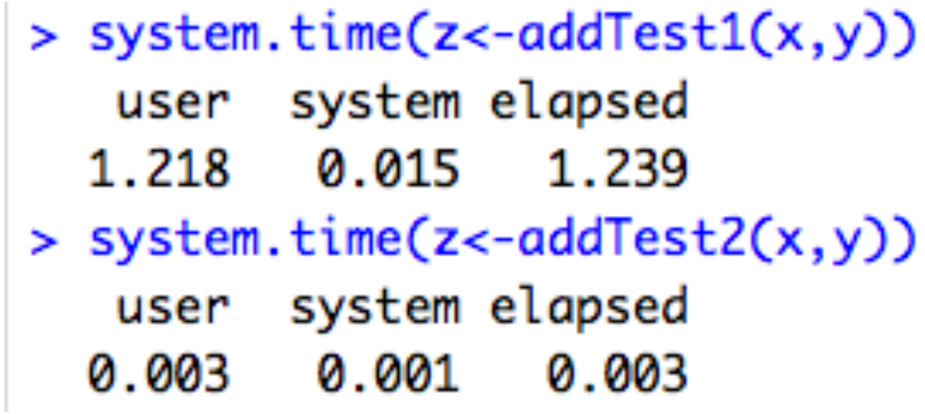
(2) Performance Enhancement

- R is an interpreted language. Many of the commands are written in C and thus do run in fast machine code
- All objects in an R session are stored in memory, with a limit of 2^{31}-1 bytes on the size of any object
- To speed up R:
  - Vectorisation, byte-code compilation
  - Use C/C++ for CPU-intensive parts of application
  - Parallel R

Vectorisation Example



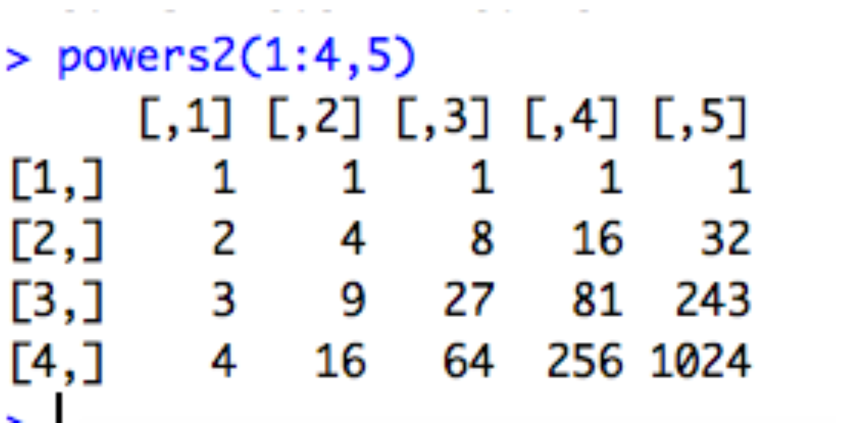
```
x<-runif(1000000)
y<-runif(1000000)

addTest1<-function(x,y){
  z<-vector(length=1000000)
  for(i in 1:length(x))
    z[i]<-x[i]+y[i]
  return(z)
}

addTest2<-function(x,y){
  z<-x+y
}
```



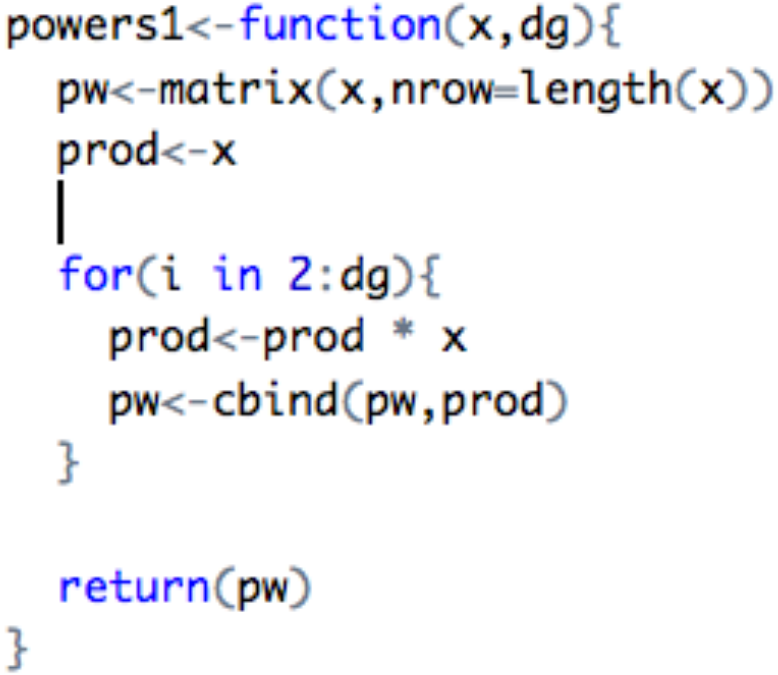
```
> system.time(z<-addTest1(x,y))
   user  system elapsed 
  1.218   0.015   1.239 
> system.time(z<-addTest2(x,y))
   user  system elapsed 
  0.003   0.001   0.003 
```

Reasons for difference

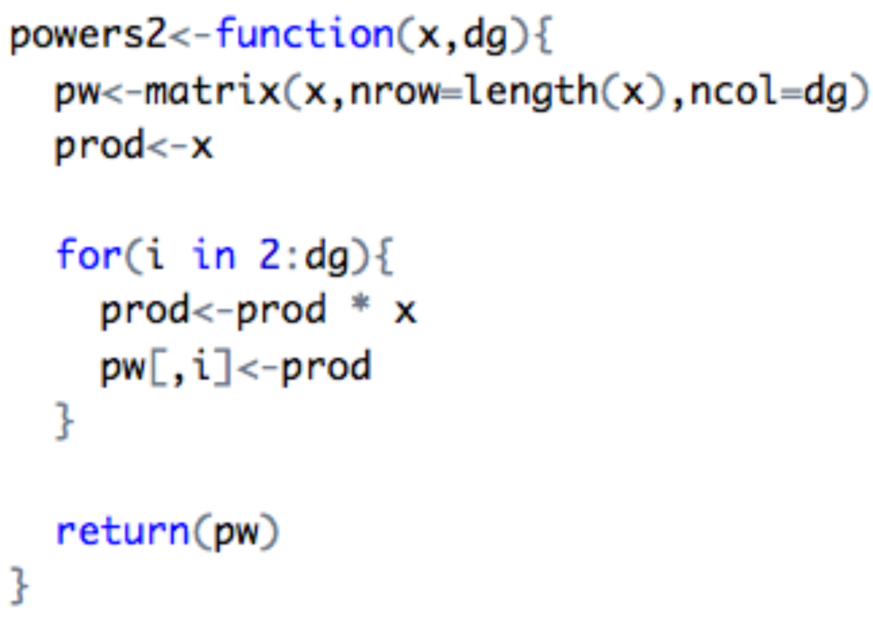
- Numerous function calls are used in the loop version
  - for() is a function
  - The colon : is also a function
  - Each subscript operation represents a function call
- Function calls involve setting up stack frames, and suffering a time penalty at each loop iteration adds up to a big slowdown.
- Powers example (Matloff 2011)



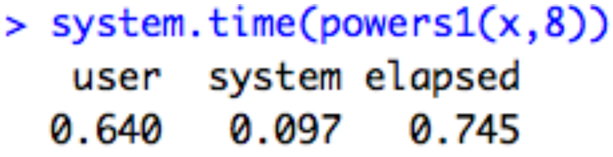
```
> powers2(1:4,5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
[2,]    2    4    8   16   32
[3,]    3    9   27   81  243
[4,]    4   16   64  256 1024
>
```



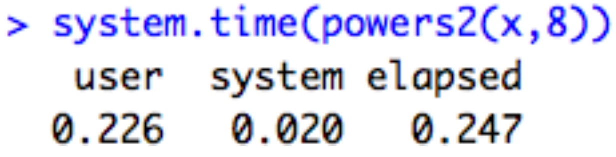
```
powers1<-function(x,dg){
  pw<-matrix(x,nrow=length(x))
  prod<-x

  for(i in 2:dg){
    prod<-prod * x
    pw<-cbind(pw,prod)
  }

  return(pw)
}
```



```
powers2<-function(x,dg){
  pw<-matrix(x,nrow=length(x),ncol=dg)
  prod<-x

  for(i in 2:dg){
    prod<-prod * x
    pw[,i]<-prod
  }

  return(pw)
}
```



```
> system.time(powers1(x,8))
   user  system elapsed 
  0.640   0.097   0.745 
```

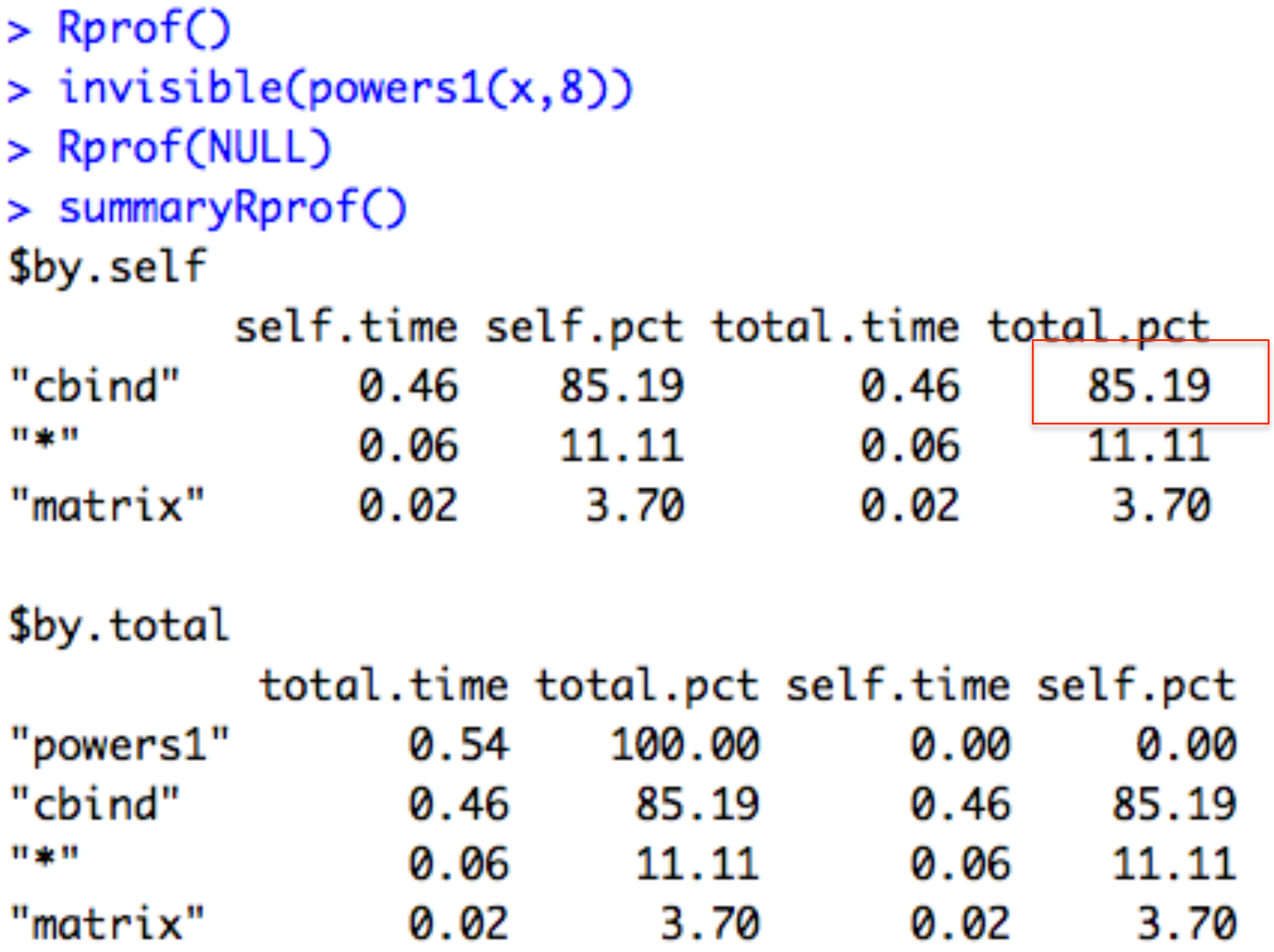


```
> system.time(powers2(x,8))
   user  system elapsed 
  0.226   0.020   0.247 
```

Finding slow spots in code – Rprof()

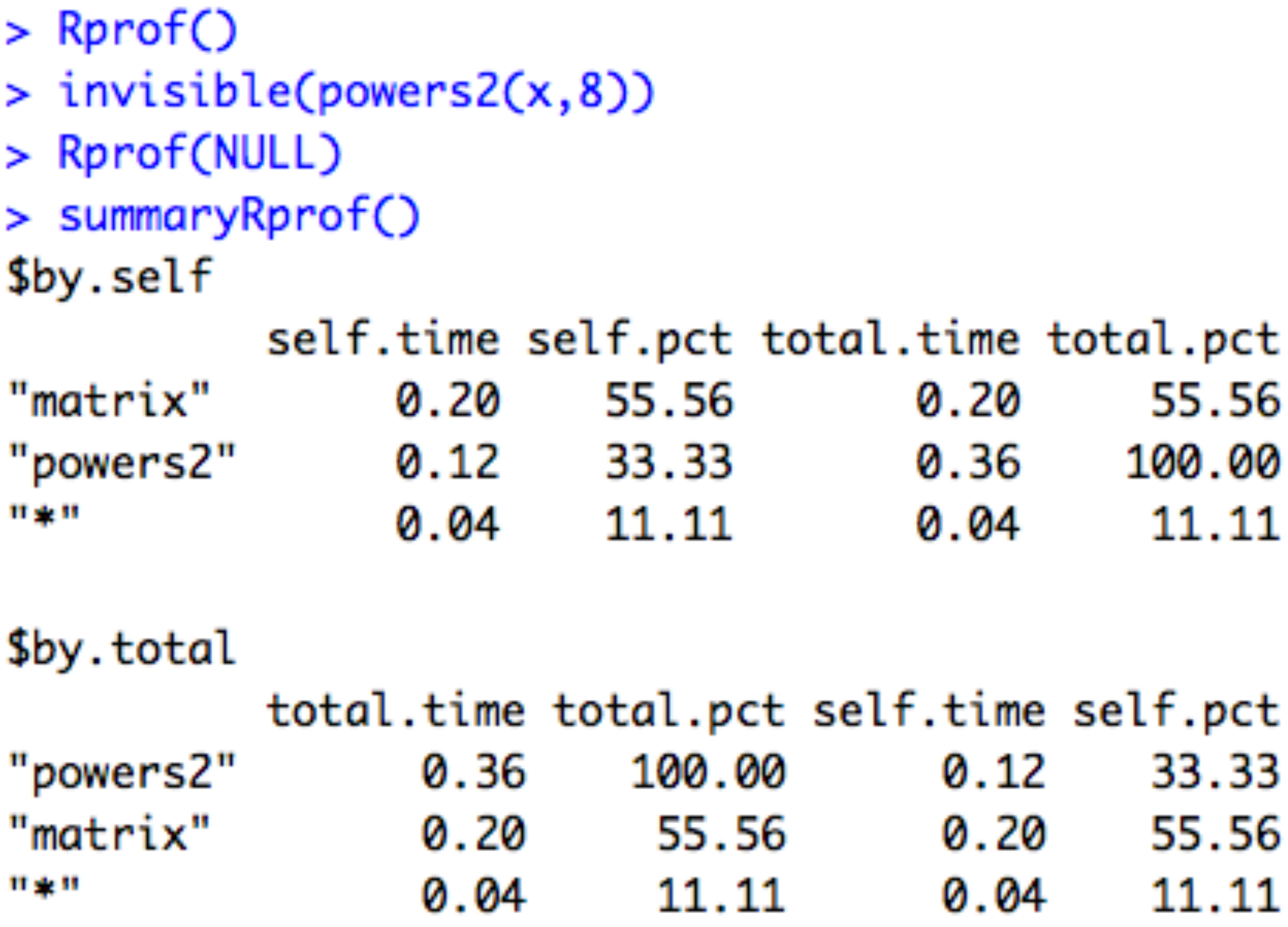
- Rprof() provides a repot of (approximately) how much time your code is spending in each of the functions it calls
- Can help target which code elements to optimise
- Process
  - Rprof()
  - invisible(*function call*)
  - Rprof(NULL)
  - summaryRprof()

Call to powers1()



```
> Rprof()
> invisible(powers1(x,8))
> Rprof(NULL)
> summaryRprof()
$by.self
         self.time self.pct total.time total.pct
"cbind"       0.46    85.19       0.46     85.19
"*"           0.06    11.11       0.06     11.11
"matrix"      0.02     3.70       0.02      3.70

$by.total
          total.time total.pct self.time self.pct
"powers1"       0.54    100.00      0.00     0.00
"cbind"         0.46     85.19      0.46    85.19
"*"             0.06     11.11      0.06    11.11
"matrix"        0.02      3.70      0.02     3.70
```

Call to powers2()



```
> Rprof()
> invisible(powers2(x,8))
> Rprof(NULL)
> summaryRprof()
$by.self
          self.time self.pct total.time total.pct
"matrix"       0.20    55.56       0.20     55.56
"powers2"      0.12    33.33       0.36    100.00
"*"            0.04    11.11       0.04     11.11

$by.total
          total.time total.pct self.time self.pct
"powers2"       0.36    100.00      0.12    33.33
"matrix"        0.20     55.56      0.20    55.56
"*"             0.04     11.11      0.04    11.11
```

Summary

- Debugging
  - debug()
  - browser()
  - setBreakpoint()
  - untrace()

- Analysing errors
  - traceback()
  - debugger()

- Profiling
  - Rprof()