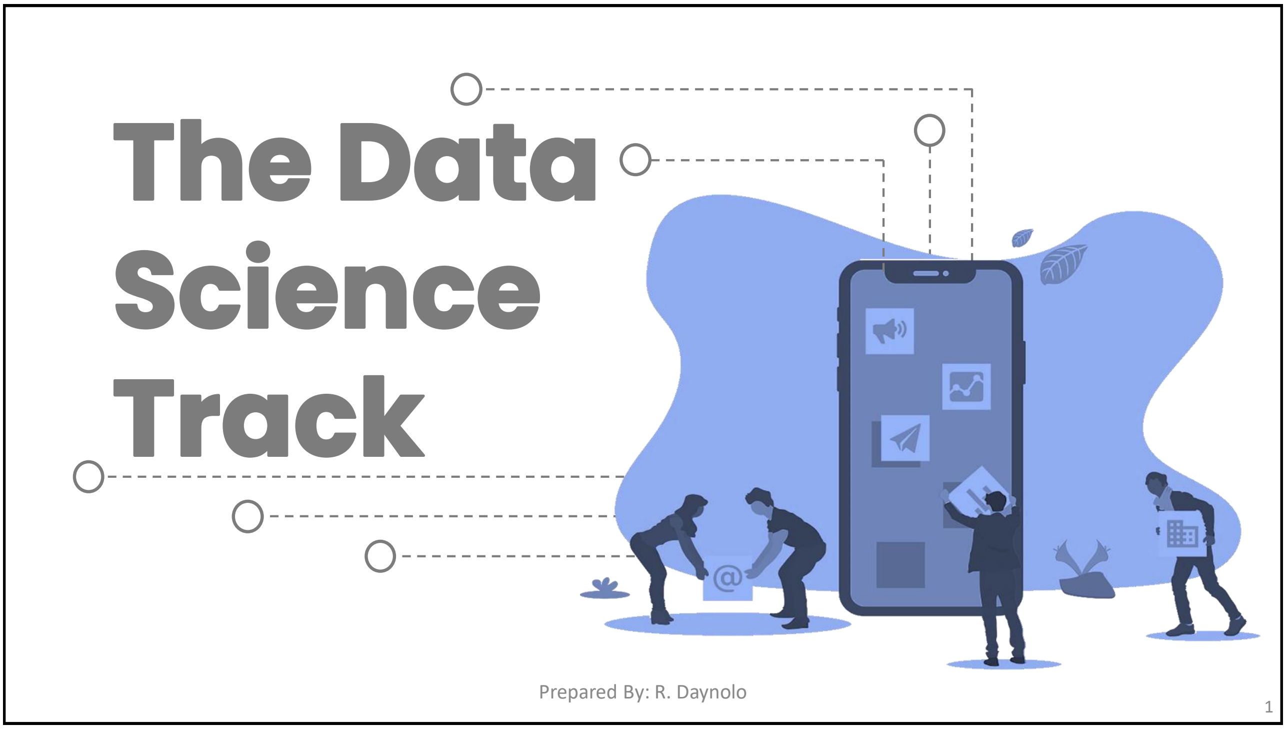
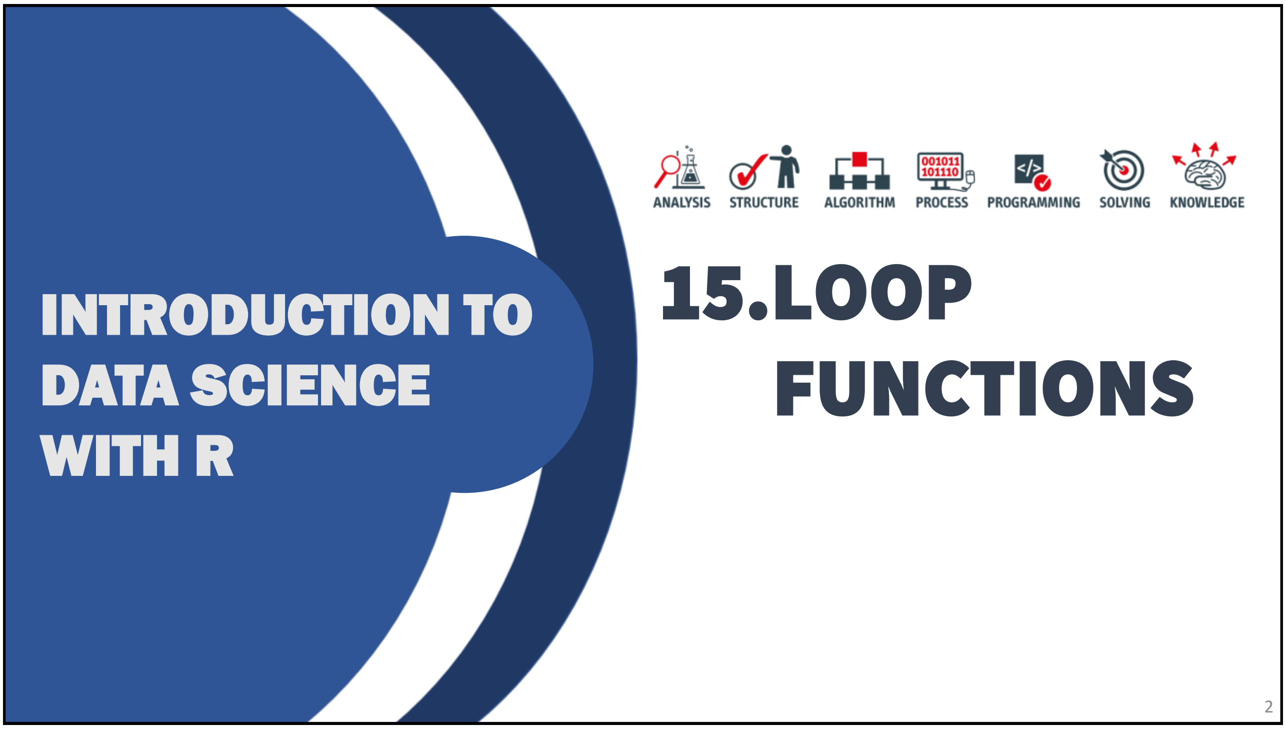


The Data Science Track

Prepared By: R. Daynolo

1

1



ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

INTRODUCTION TO DATA SCIENCE WITH R

15.LOOP FUNCTIONS

2

2

LOOPING ON THE COMMAND LINE

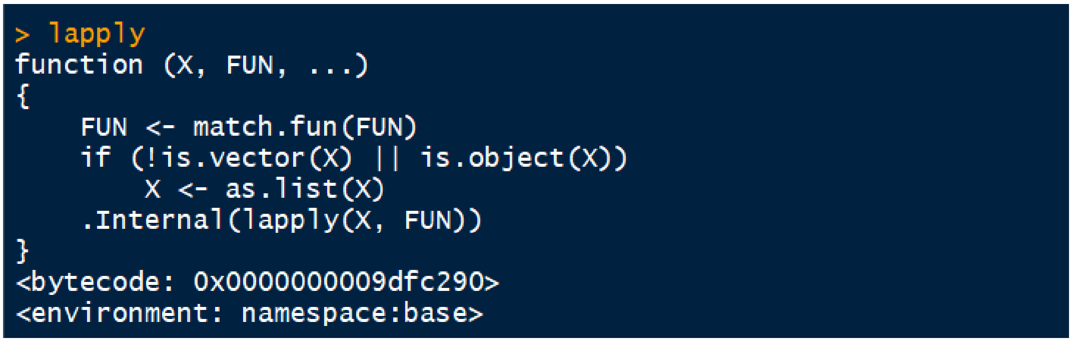
Writing for, while loops is useful when programming but not particularly easy when working interactively on the command line. There are some functions which implement looping to make life easier.

- `lapply`: Loop over a list and evaluate a function on each element
- `sapply`: Same as `lapply` but try to simplify the result
- `apply`: Apply a function over the margins of an array
- `tapply`: Apply a function over subsets of a vector
- `mapply`: Multivariate version of `lapply`

An auxiliary function split is also useful, particularly in conjunction with lapply.

lapply

`lapply` takes three arguments: (1) a list `X`; (2) a function (or the name of a function) `FUN`; (3) other arguments via its `...` argument. If `X` is not a list, it will be coerced to a list using `as.list`.

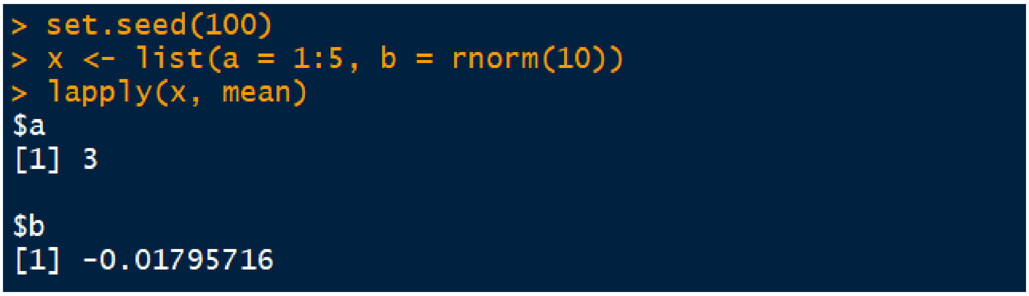


```
> lapply
function (X, FUN, ...)
{
    FUN <- match.fun(FUN)
    if (!is.vector(X) || is.object(X))
        X <- as.list(X)
    .Internal(lapply(X, FUN))
}
<bytecode: 0x0000000009dfc290>
<environment: namespace:base>
```

Note: The actual looping is done internally in **C** code.

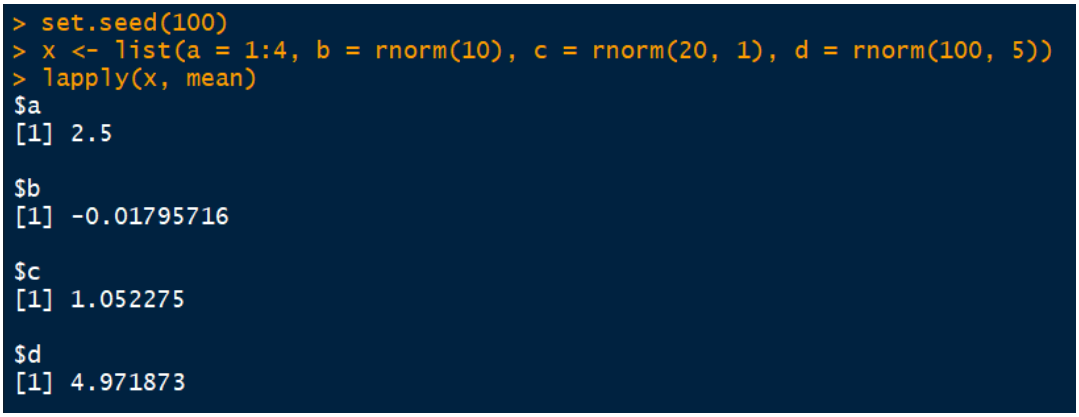
lapply

`lapply` always returns a list, regardless of the class of the input



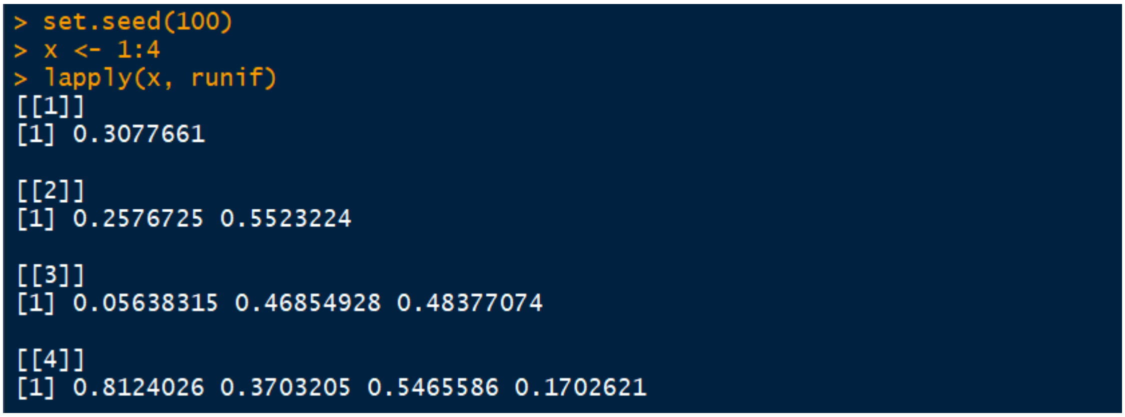
```
> set.seed(100)
> x <- list(a = 1:5, b = rnorm(10))
> lapply(x, mean)
$a
[1] 3

$b
[1] -0.01795716
```

lapply



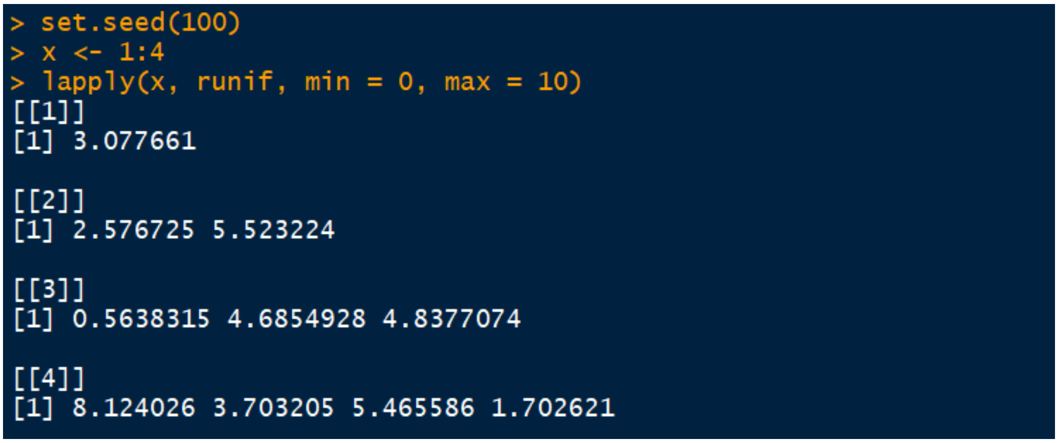
```
> set.seed(100)
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] -0.01795716

$c
[1] 1.052275

$d
[1] 4.971873
```

lapply

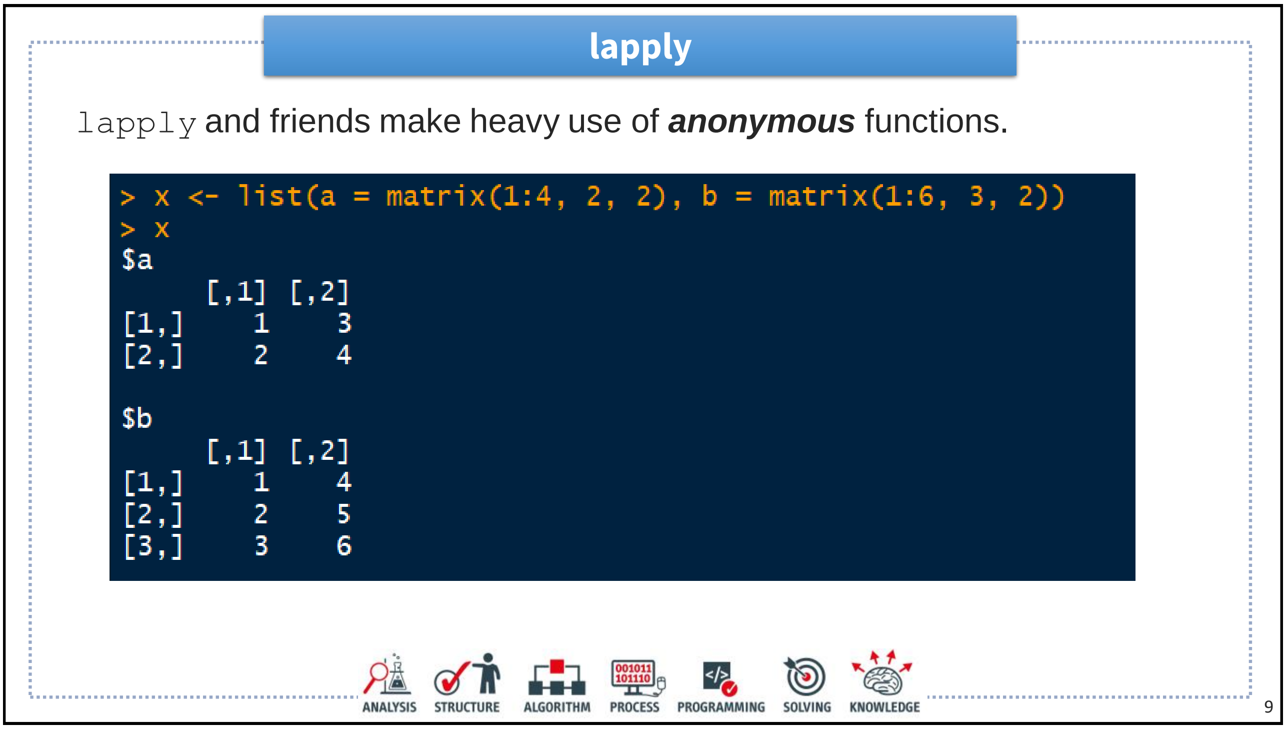


```
> set.seed(100)
> x <- 1:4
> lapply(x, runif)
[[1]]
[1] 0.3077661

[[2]]
[1] 0.2576725 0.5523224

[[3]]
[1] 0.05638315 0.46854928 0.48377074

[[4]]
[1] 0.8124026 0.3703205 0.5465586 0.1702621
```

lapply



```
> set.seed(100)
> x <- 1:4
> lapply(x, runif, min = 0, max = 10)
[[1]]
[1] 3.077661

[[2]]
[1] 2.576725 5.523224

[[3]]
[1] 0.5638315 4.6854928 4.8377074

[[4]]
[1] 8.124026 3.703205 5.465586 1.702621
```

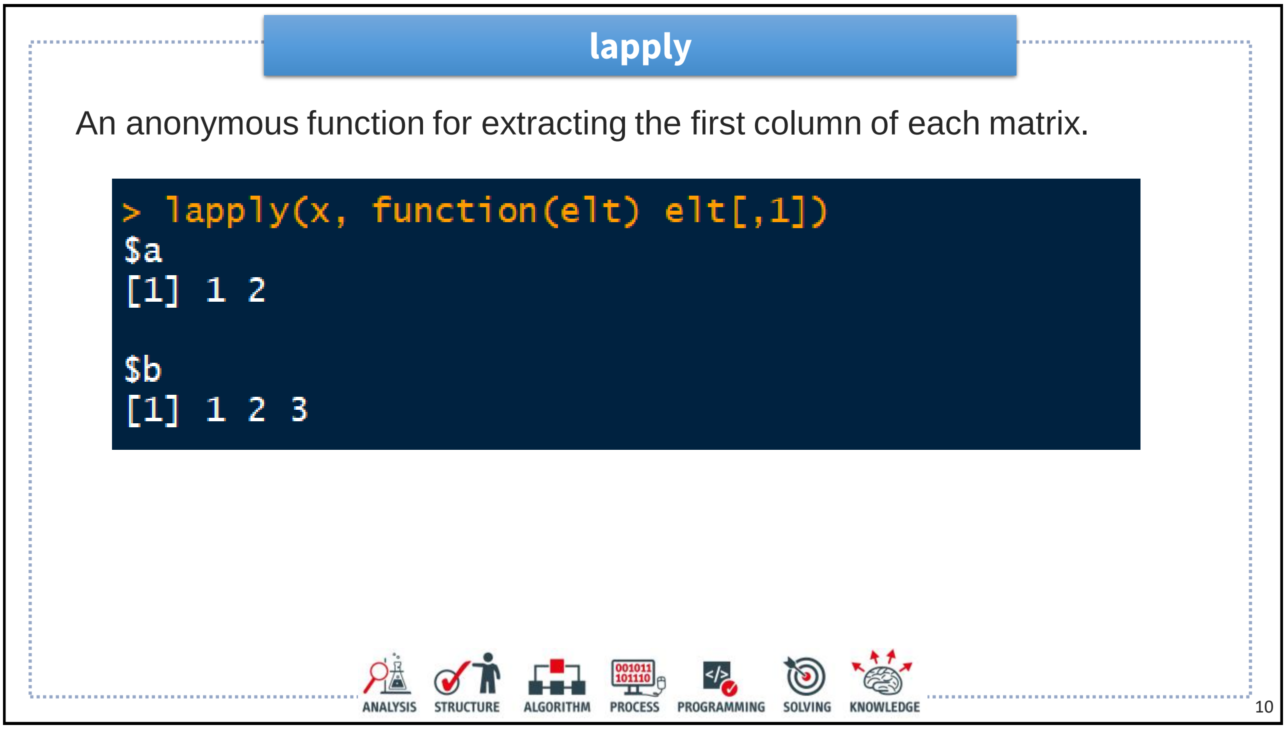


lapply

lapply and friends make heavy use of ***anonymous*** functions.

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
> x
$a
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$b
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

ANALYSIS
STRUCTURE
ALGORITHM
PROCESS
PROGRAMMING
SOLVING
KNOWLEDGE



lapply

An anonymous function for extracting the first column of each matrix.

```
> lapply(x, function(elt) elt[,1])
$a
[1] 1 2

$b
[1] 1 2 3
```

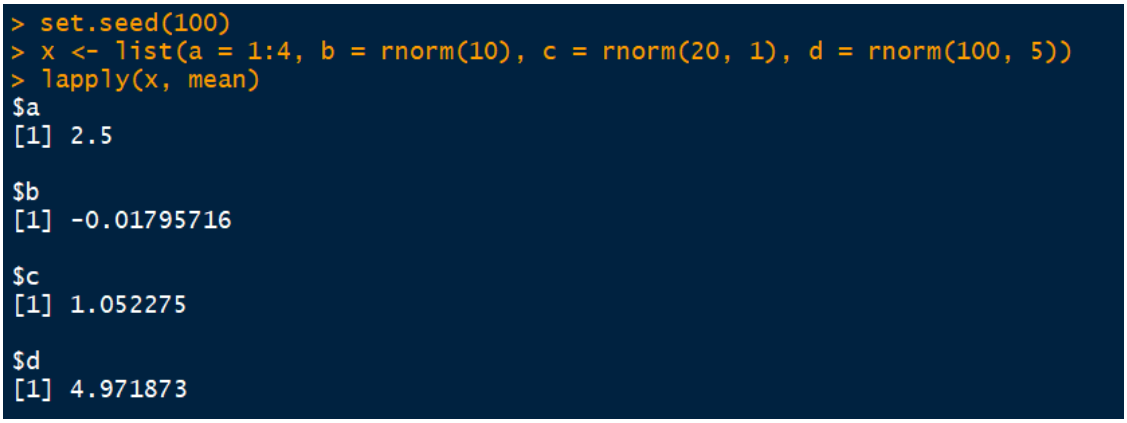
ANALYSIS
STRUCTURE
ALGORITHM
PROCESS
PROGRAMMING
SOLVING
KNOWLEDGE

sapply

`sapply` will try to simplify the result of `lapply` if possible.

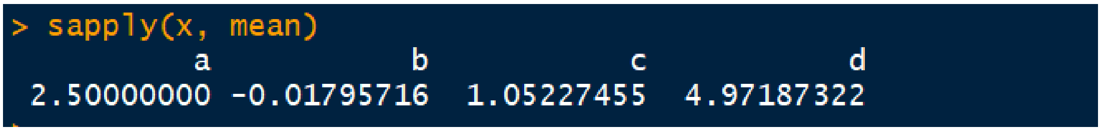
- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

sapply

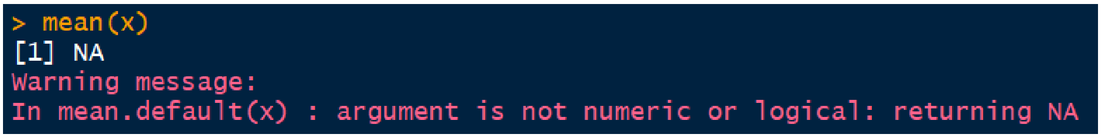


```
> set.seed(100)
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] -0.01795716

$c
[1] 1.052275

$d
[1] 4.971873
```

sapply



```
> sapply(x, mean)
          a           b           c           d
 2.50000000 -0.01795716  1.05227455  4.97187322
```

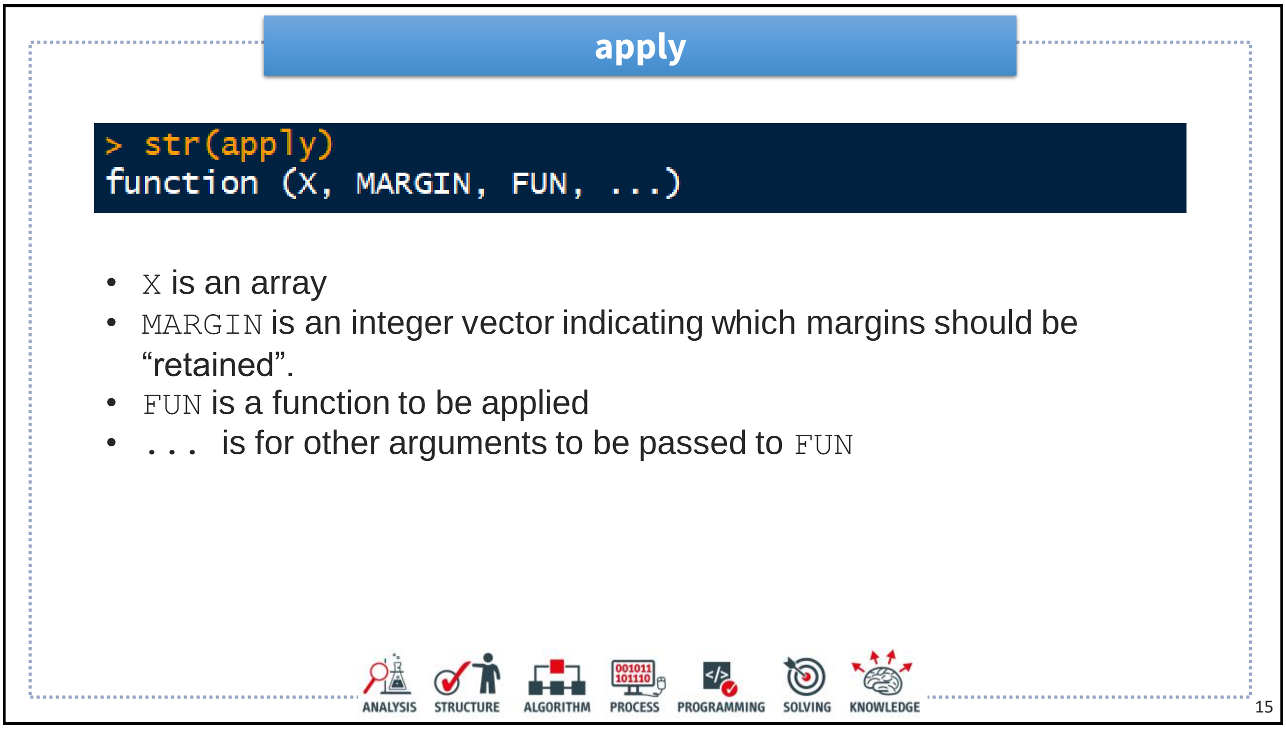


```
> mean(x)
[1] NA
Warning message:
In mean.default(x) : argument is not numeric or logical: returning NA
```

apply

`apply` is used to a evaluate a function (often an anonymous one) over the margins of an array.

- It is most often used to apply a function to the rows or columns of a matrix
- It can be used with general arrays, e.g. taking the average of an array of matrices
- It is not really faster than writing a loop, but it works in one line!

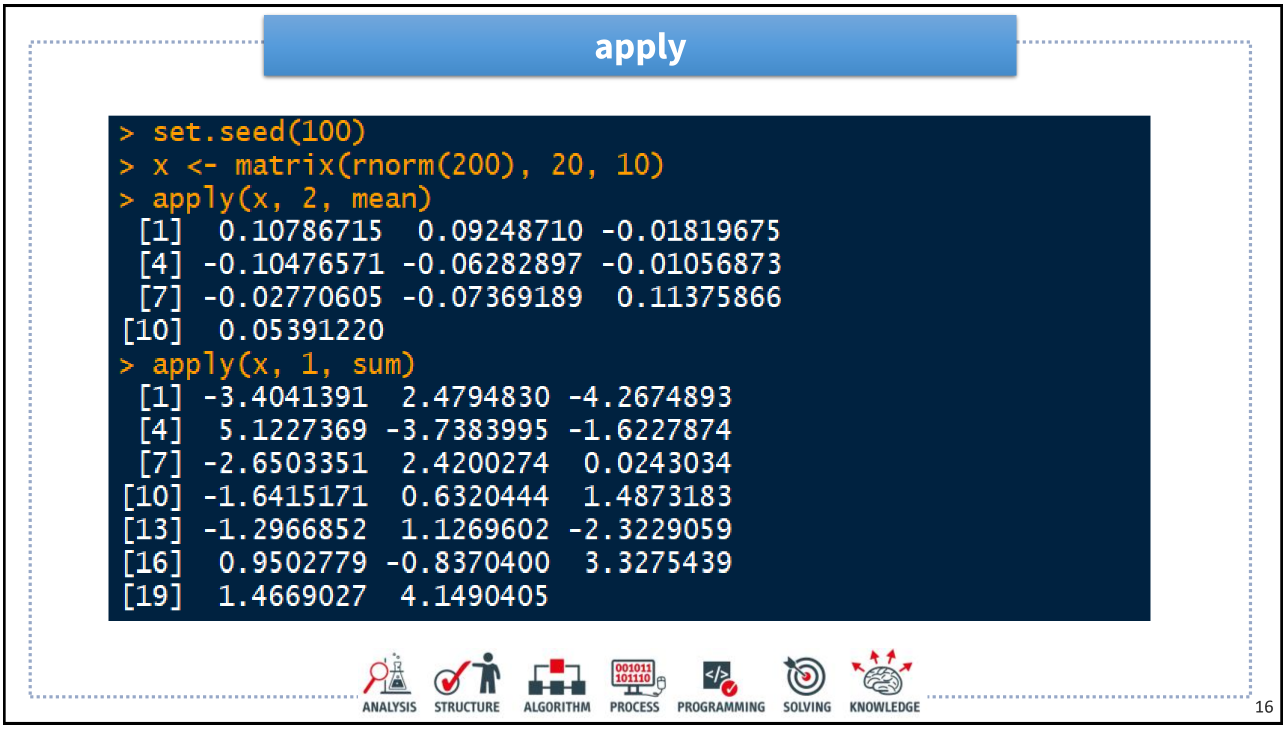


apply

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

- `X` is an array
- `MARGIN` is an integer vector indicating which margins should be "retained".
- `FUN` is a function to be applied
- `...` is for other arguments to be passed to `FUN`

ANALYSIS
STRUCTURE
ALGORITHM
PROCESS
PROGRAMMING
SOLVING
KNOWLEDGE



apply

```
> set.seed(100)
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)
 [1]  0.10786715  0.09248710 -0.01819675
 [4] -0.10476571 -0.06282897 -0.01056873
 [7] -0.02770605 -0.07369189  0.11375866
[10]  0.05391220
> apply(x, 1, sum)
 [1] -3.4041391  2.4794830 -4.2674893
 [4]  5.1227369 -3.7383995 -1.6227874
 [7] -2.6503351  2.4200274  0.0243034
[10] -1.6415171  0.6320444  1.4873183
[13] -1.2966852  1.1269602 -2.3229059
[16]  0.9502779 -0.8370400  3.3275439
[19]  1.4669027  4.1490405
```

col/row sums and means

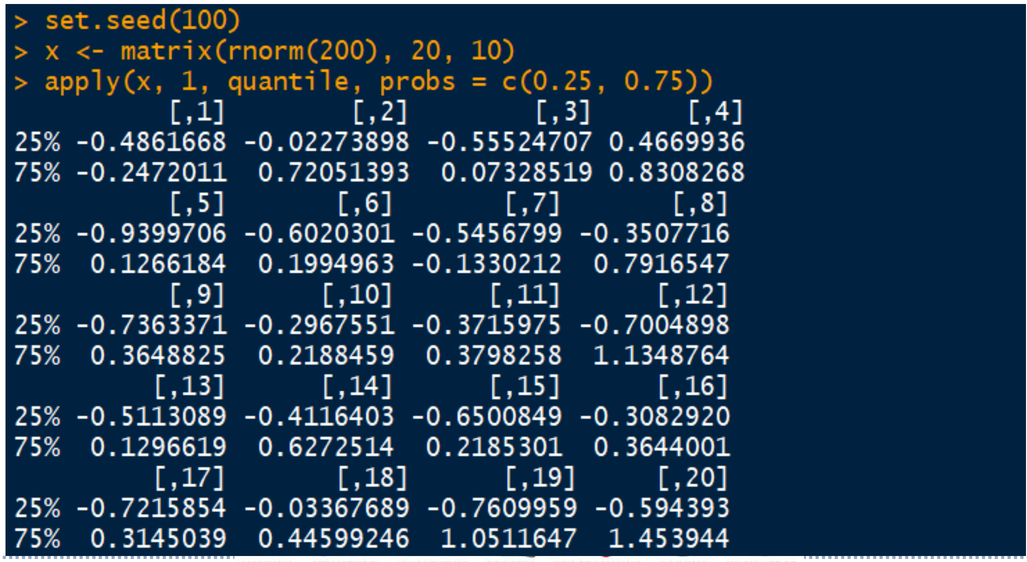
For sums and means of matrix dimensions, we have some shortcuts.

- `rowSums = apply(x, 1, sum)`
- `rowMeans = apply(x, 1, mean)`
- `colSums = apply(x, 2, sum)`
- `colMeans = apply(x, 2, mean)`

The shortcut functions are much faster, but you won’t notice unless you’re using a large matrix.

OTHER WAYS TO apply

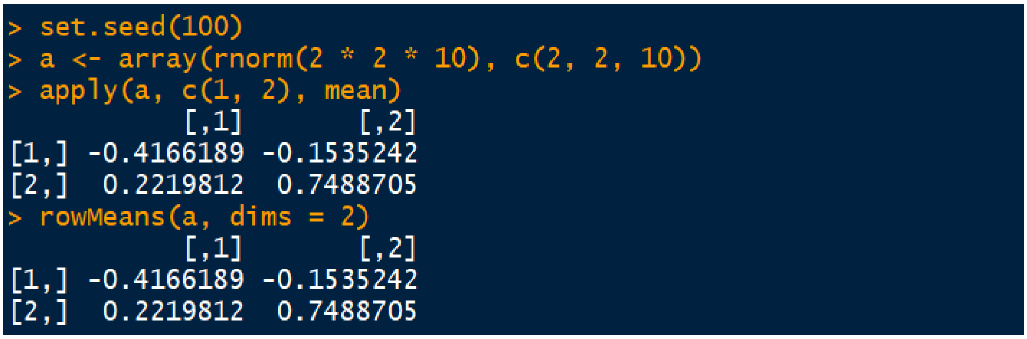
Quantiles of the rows of a matrix.



```
> set.seed(100)
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 1, quantile, probs = c(0.25, 0.75))
          [,1]        [,2]        [,3]      [,4]
25% -0.4861668 -0.02273898 -0.55524707 0.4669936
75% -0.2472011  0.72051393  0.07328519 0.8308268
          [,5]       [,6]       [,7]       [,8]
25% -0.9399706 -0.6020301 -0.5456799 -0.3507716
75%  0.1266184  0.1994963 -0.1330212  0.7916547
          [,9]      [,10]      [,11]      [,12]
25% -0.7363371 -0.2967551 -0.3715975 -0.7004898
75%  0.3648825  0.2188459  0.3798258  1.1348764
         [,13]      [,14]      [,15]      [,16]
25% -0.5113089 -0.4116403 -0.6500849 -0.3082920
75%  0.1296619  0.6272514  0.2185301  0.3644001
         [,17]       [,18]      [,19]     [,20]
25% -0.7215854 -0.03367689 -0.7609959 -0.594393
75%  0.3145039  0.44599246  1.0511647  1.453944
```

OTHER WAYS TO apply

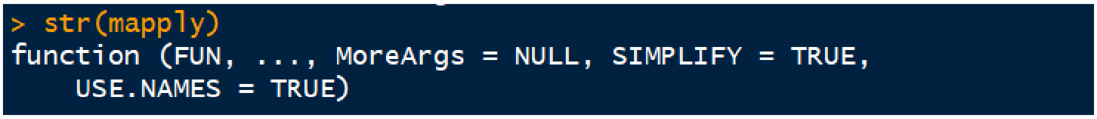
Average matrix in an array.



```
> set.seed(100)
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
> apply(a, c(1, 2), mean)
           [,1]       [,2]
[1,] -0.4166189 -0.1535242
[2,]  0.2219812  0.7488705
> rowMeans(a, dims = 2)
           [,1]       [,2]
[1,] -0.4166189 -0.1535242
[2,]  0.2219812  0.7488705
```

mapply

mapply is a multivariate apply of sorts which applies a function in parallel over a set of arguments.



```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
    USE.NAMES = TRUE)
```

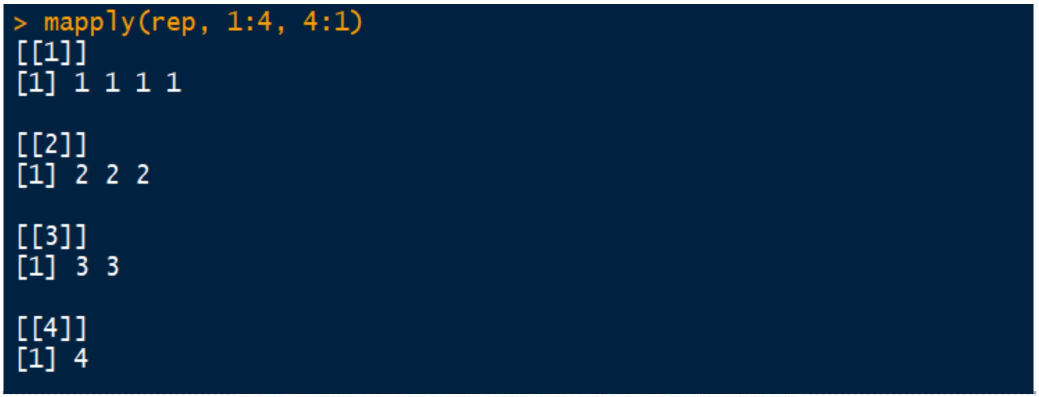
- FUN is a function to apply
- ... contains arguments to apply over
- MoreArgs is a list of other arguments to FUN.
- SIMPLIFY indicates whether the result should be simplified

mapply

The following is tedious to type

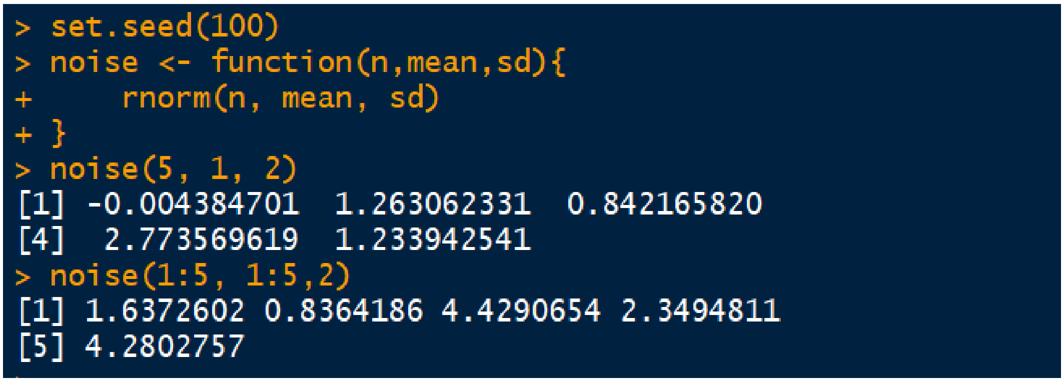
```
list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
```

Instead we can do



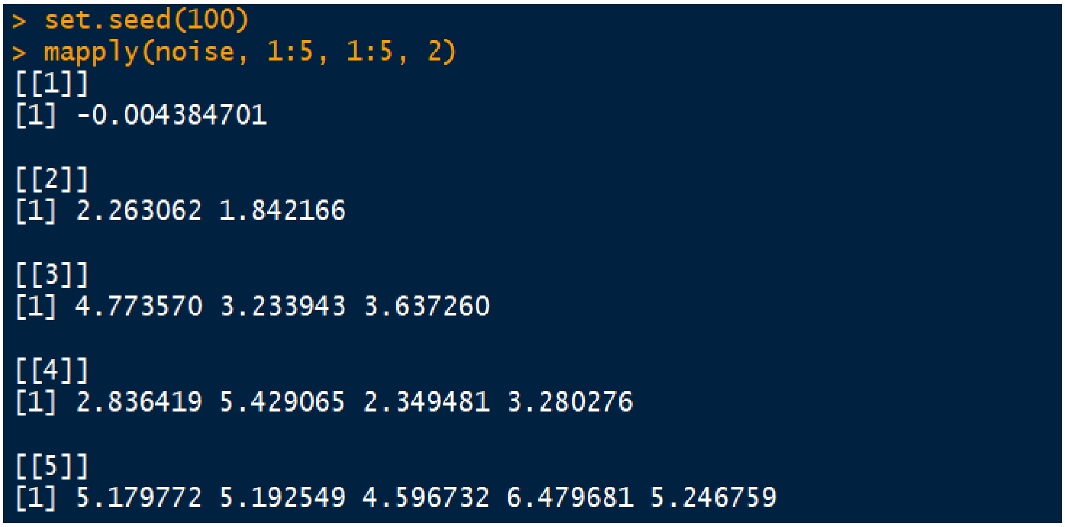
```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
```

VECTORIZING A FUNCTION



```
> set.seed(100)
> noise <- function(n,mean,sd){
+     rnorm(n, mean, sd)
+ }
> noise(5, 1, 2)
[1] -0.004384701  1.263062331  0.842165820
[4]  2.773569619  1.233942541
> noise(1:5, 1:5,2)
[1] 1.6372602 0.8364186 4.4290654 2.3494811
[5] 4.2802757
```

INSTANT VECTORIZATION

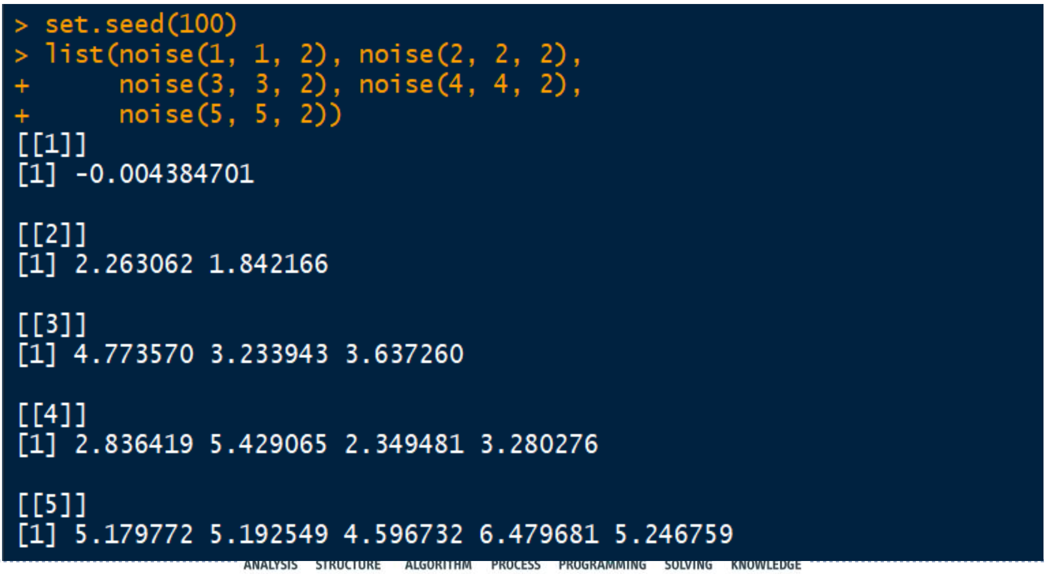


```
> set.seed(100)
> mapply(noise, 1:5, 1:5, 2)
[[1]]
[1] -0.004384701

[[2]]
[1] 2.263062 1.842166

[[3]]
[1] 4.773570 3.233943 3.637260

[[4]]
[1] 2.836419 5.429065 2.349481 3.280276

[[5]]
[1] 5.179772 5.192549 4.596732 6.479681 5.246759
```

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

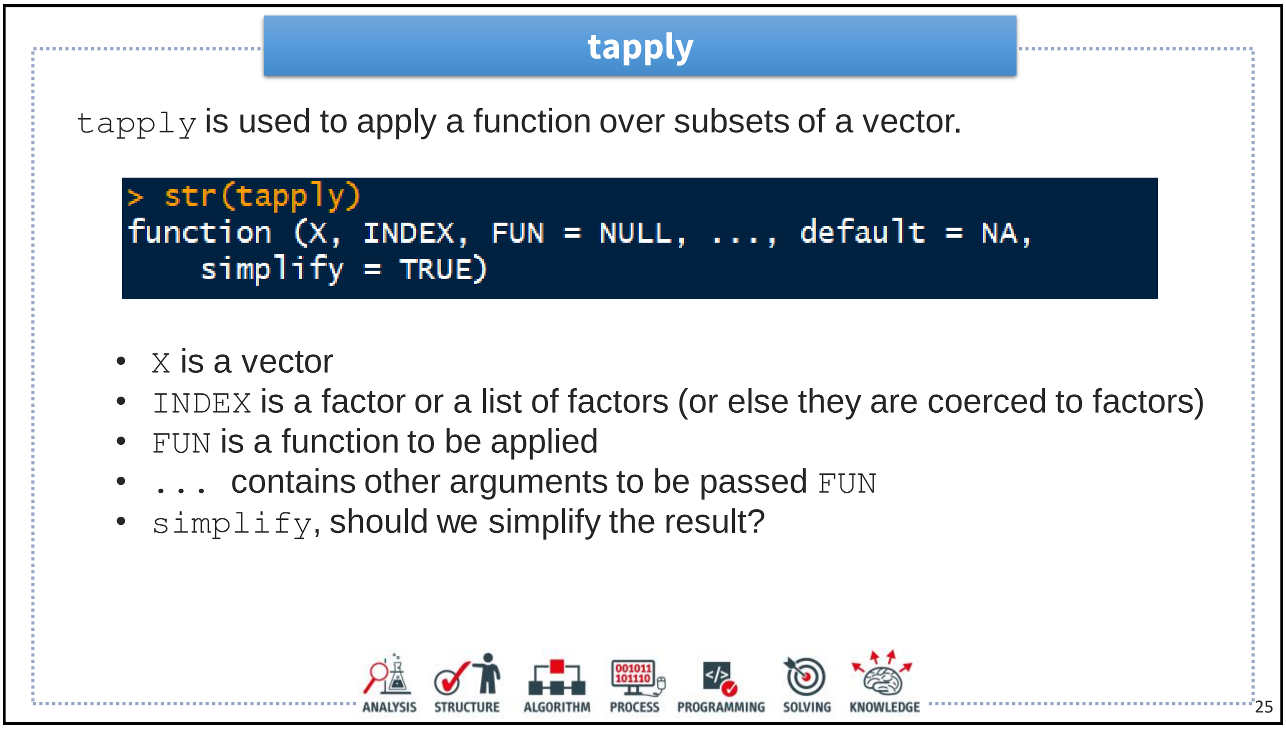
mapply

Which is the same as:



```
> set.seed(100)
> list(noise(1, 1, 2), noise(2, 2, 2),
+      noise(3, 3, 2), noise(4, 4, 2),
+      noise(5, 5, 2))
[[1]]
[1] -0.004384701

[[2]]
[1] 2.263062 1.842166

[[3]]
[1] 4.773570 3.233943 3.637260

[[4]]
[1] 2.836419 5.429065 2.349481 3.280276

[[5]]
[1] 5.179772 5.192549 4.596732 6.479681 5.246759
```

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE



tapply

`tapply` is used to apply a function over subsets of a vector.

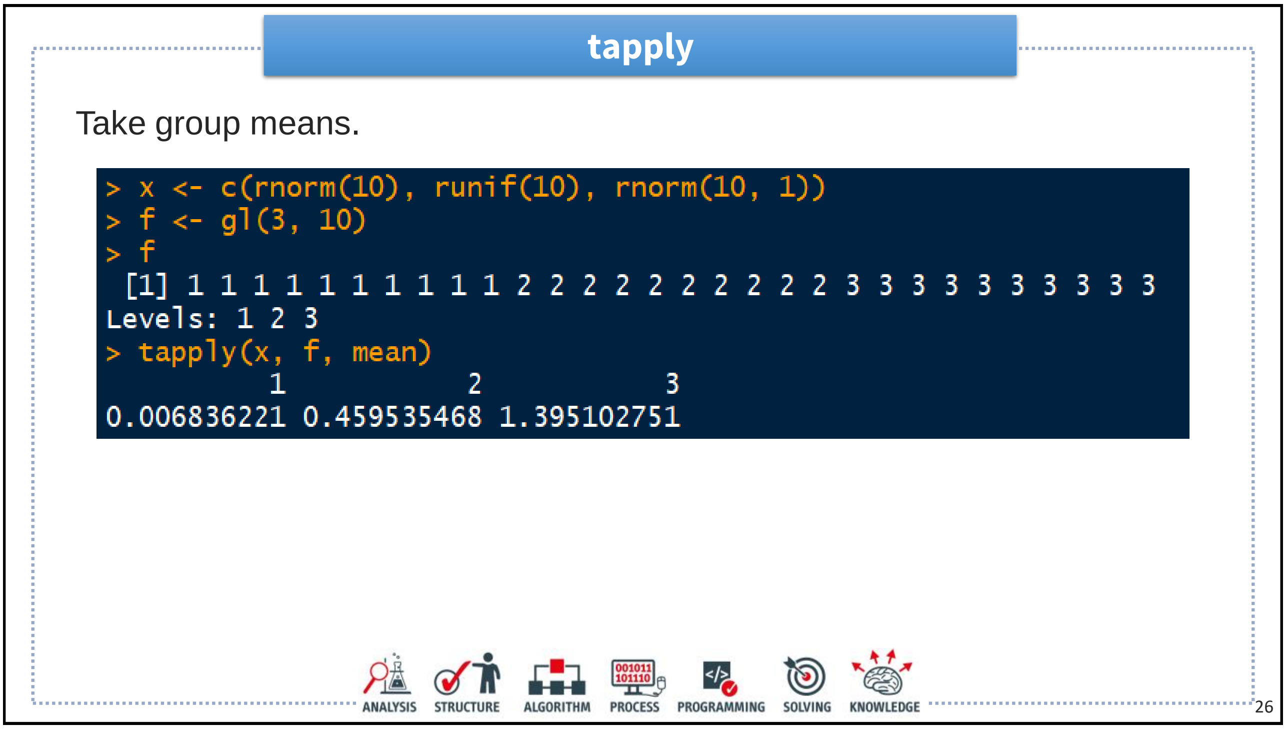
```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., default = NA,
    simplify = TRUE)
```

- `X` is a vector
- `INDEX` is a factor or a list of factors (or else they are coerced to factors)
- `FUN` is a function to be applied
- `...` contains other arguments to be passed `FUN`
- `simplify`, should we simplify the result?

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

25

25



tapply

Take group means.

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> f
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
          1           2           3 
0.006836221 0.459535468 1.395102751 
```

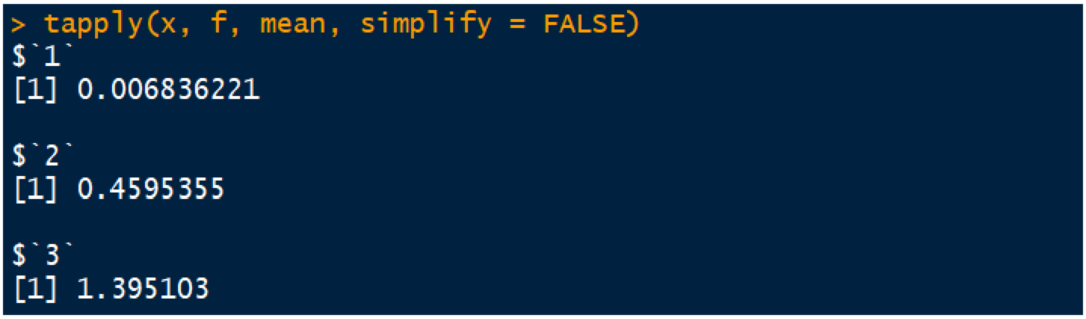
ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

26

26

tapply

Take group means without simplification.

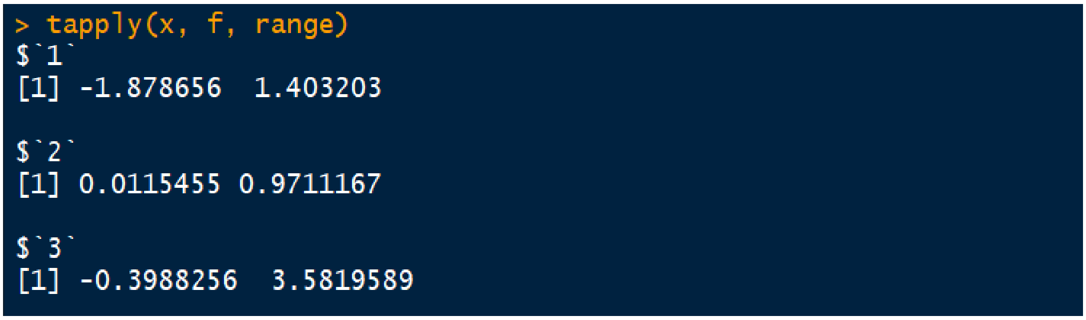


```
> tapply(x, f, mean, simplify = FALSE)
$`1`
[1] 0.006836221

$`2`
[1] 0.4595355

$`3`
[1] 1.395103
```

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

tapply

Find group ranges.

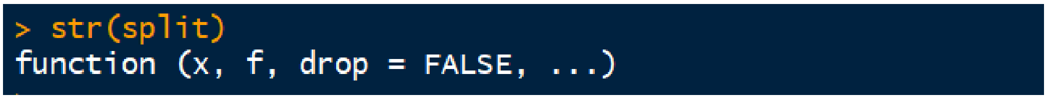


```
> tapply(x, f, range)
$`1`
[1] -1.878656  1.403203

$`2`
[1] 0.0115455 0.9711167

$`3`
[1] -0.3988256  3.5819589
```

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

split

`split` takes a vector or other objects and splits it into groups determined by a factor or list of factors.

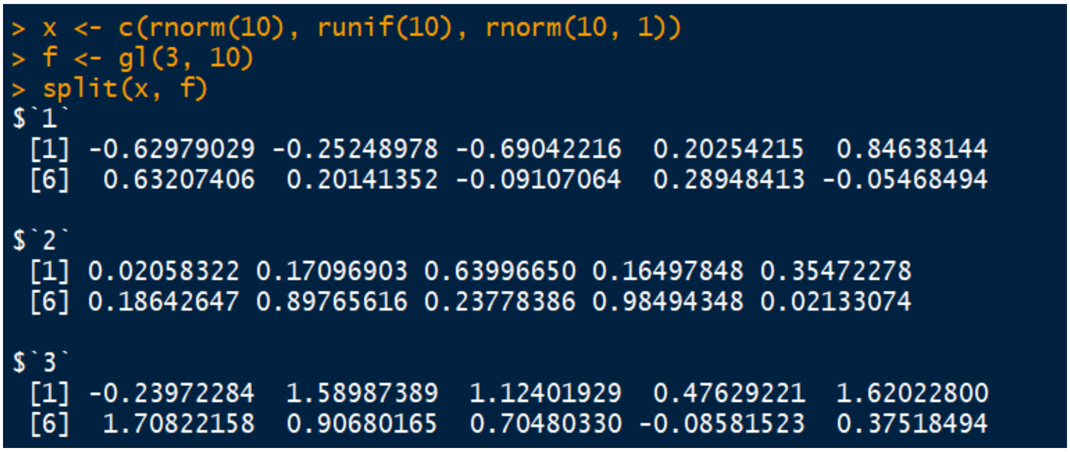


```
> str(split)
function (x, f, drop = FALSE, ...)
```

- `x` is a vector (or list) or data frame
- `f` is a factor (or coerced to one) or a list of factors
- `drop` indicates whether empty factors levels should be dropped

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

split

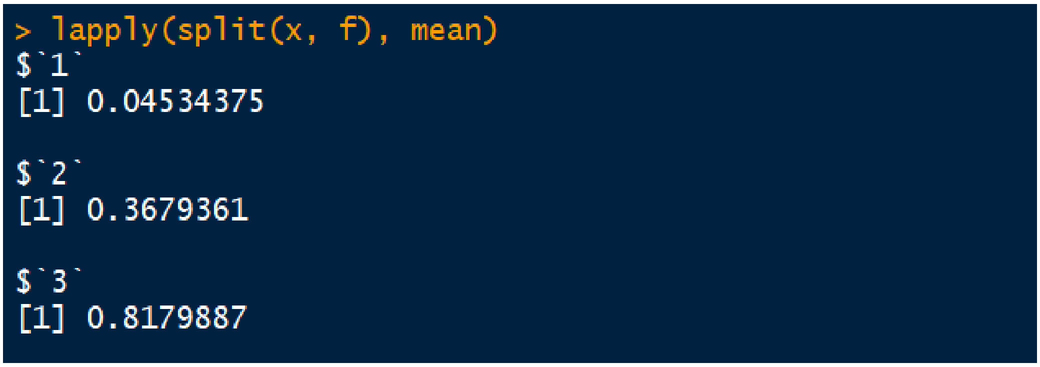


```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> split(x, f)
$`1`
 [1] -0.62979029 -0.25248978 -0.69042216  0.20254215  0.84638144
 [6]  0.63207406  0.20141352 -0.09107064  0.28948413 -0.05468494

$`2`
 [1] 0.02058322 0.17096903 0.63996650 0.16497848 0.35472278
 [6] 0.18642647 0.89765616 0.23778386 0.98494348 0.02133074

$`3`
 [1] -0.23972284  1.58987389  1.12401929  0.47629221  1.62022800
 [6]  1.70822158  0.90680165  0.70480330 -0.08581523  0.37518494
```

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

split

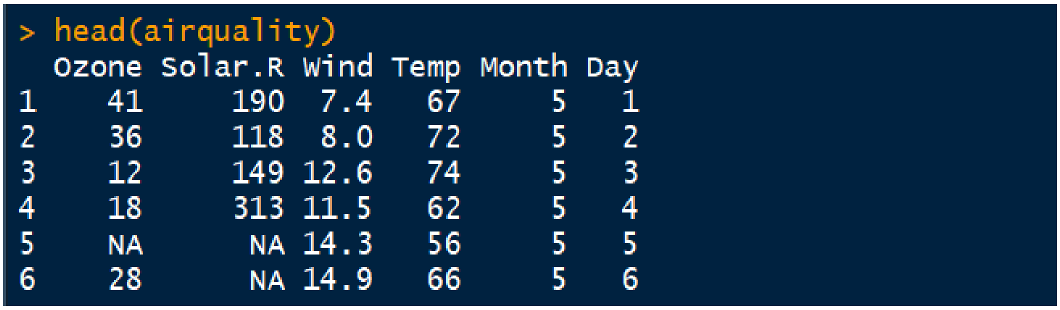
A common idiom is `split` followed by `lapply`.



```
> lapply(split(x, f), mean)
$`1`
[1] 0.04534375

$`2`
[1] 0.3679361

$`3`
[1] 0.8179887
```

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

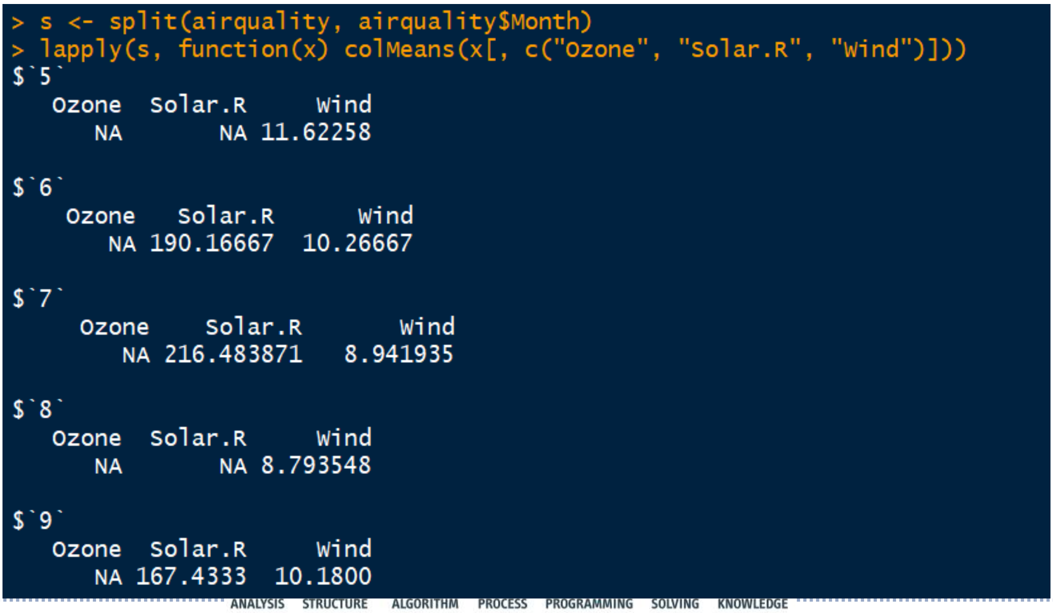
SPLITTING A DATA FRAME



```
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

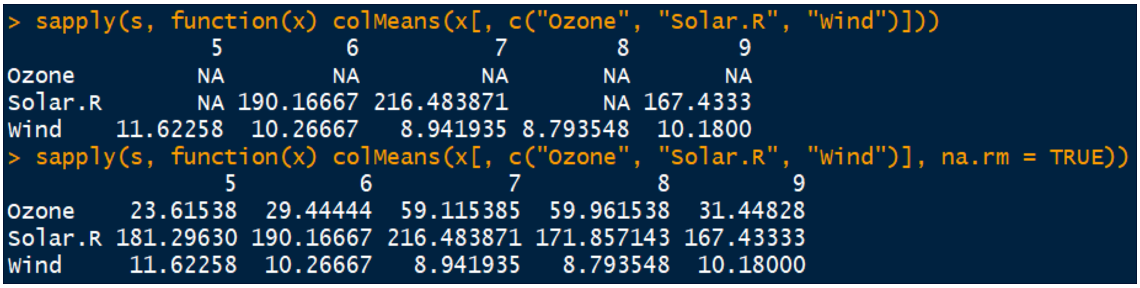
ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

SPLITTING A DATA FRAME



```
> s <- split(airquality, airquality$Month)
> lapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
$`5`
   Ozone  Solar.R     Wind
      NA       NA 11.62258

$`6`
    Ozone   Solar.R      Wind
       NA 190.16667  10.26667

$`7`
     Ozone     Solar.R       Wind
        NA 216.483871   8.941935

$`8`
   Ozone  Solar.R     Wind
      NA       NA 8.793548

$`9`
   Ozone  Solar.R     Wind
      NA 167.4333  10.1800
```

SPLITTING A DATA FRAME



```
> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
               5         6          7        8        9
Ozone         NA        NA         NA       NA       NA
Solar.R       NA 190.16667 216.483871       NA 167.4333
Wind    11.62258  10.26667   8.941935 8.793548  10.1800
> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")], na.rm = TRUE))
                5         6          7          8         9
Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

SPLITTING MORE THAN ONE LEVEL

```
> x <- rnorm(10)
> f1 <- gl(2, 5)
> f2 <- gl(5, 2)
> f1
 [1] 1 1 1 1 1 2 2 2 2 2
Levels: 1 2
> f2
 [1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5
> interaction(f1, f2)
 [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 2.4 1.5 2.5
```

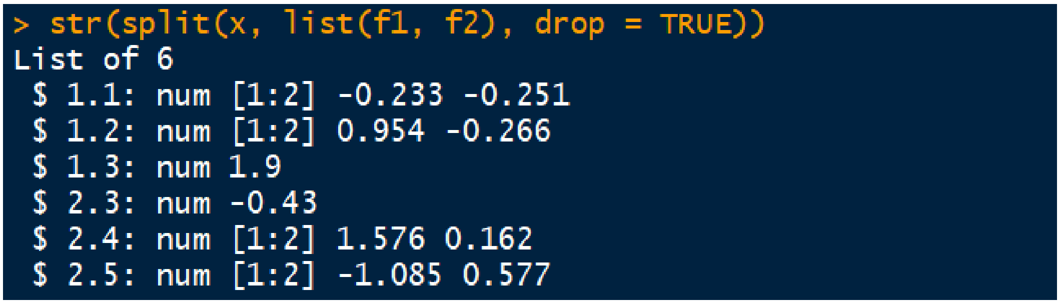
SPLITTING MORE THAN ONE LEVEL

Interactions can create empty levels.

```
> str(split(x, list(f1, f2)))
List of 10
 $ 1.1: num [1:2] -0.233 -0.251
 $ 2.1: num(0)
 $ 1.2: num [1:2] 0.954 -0.266
 $ 2.2: num(0)
 $ 1.3: num 1.9
 $ 2.3: num -0.43
 $ 1.4: num(0)
 $ 2.4: num [1:2] 1.576 0.162
 $ 1.5: num(0)
 $ 2.5: num [1:2] -1.085 0.577
```

SPLITTING MORE THAN ONE LEVEL

Empty levels can be dropped.



```
> str(split(x, list(f1, f2), drop = TRUE))
List of 6
 $ 1.1: num [1:2] -0.233 -0.251
 $ 1.2: num [1:2] 0.954 -0.266
 $ 1.3: num 1.9
 $ 2.3: num -0.43
 $ 2.4: num [1:2] 1.576 0.162
 $ 2.5: num [1:2] -1.085 0.577
```

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

INTRODUCTION TO DATA SCIENCE WITH R

16.DEBUGGING

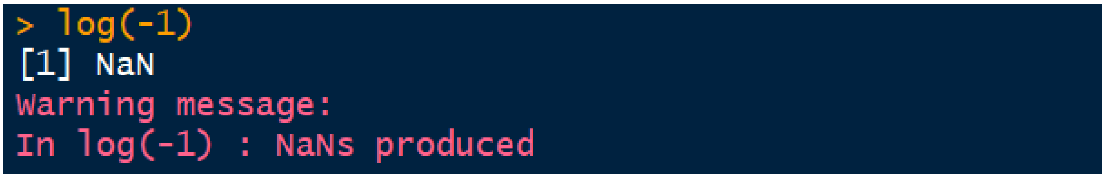
SOMETHING'S WRONG!

Indications that something's not right

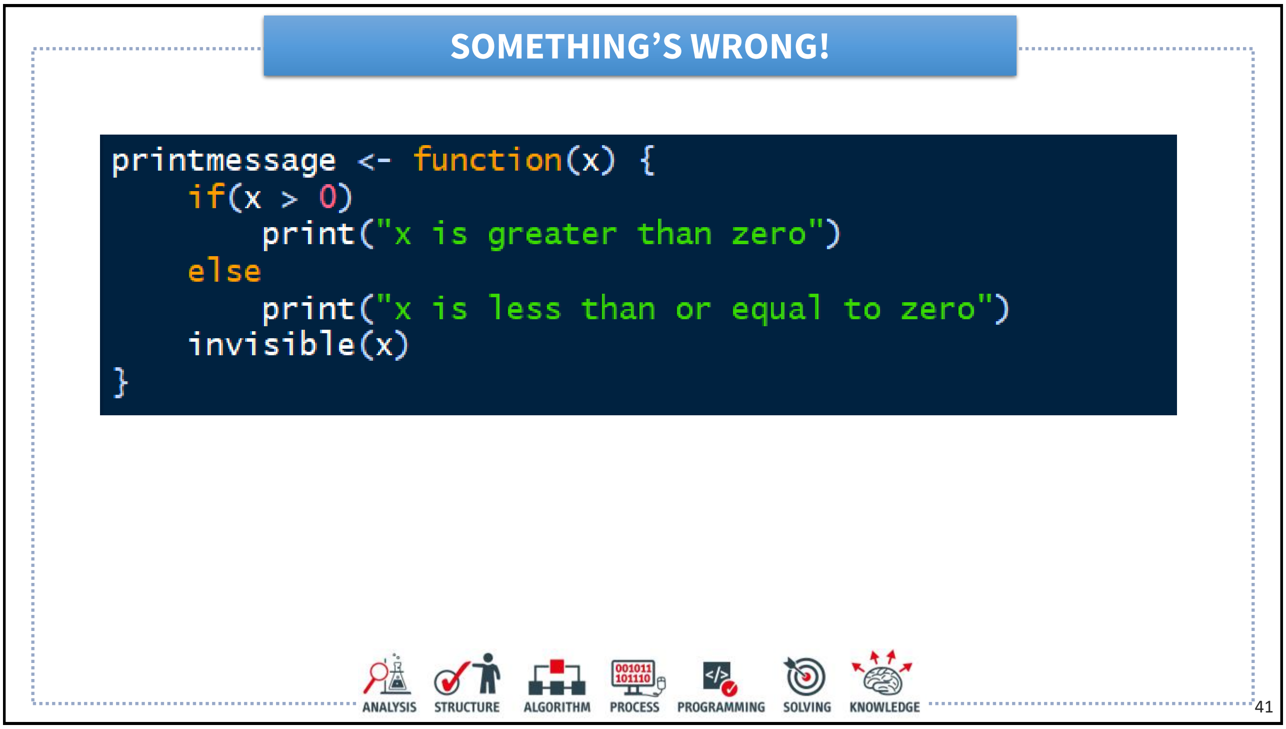
- `message`: A generic notification/diagnostic message produced by the `message` function; execution of the function continues
- `warning`: An indication that something is wrong but not necessarily fatal; execution of the function continues; generated by the `warning` function
- `error`: An indication that a fatal problem has occurred; execution stops; produced by the `stop` function
- `condition`: A generic concept for indicating that something unexpected can occur; programmers can create their own conditions

SOMETHING'S WRONG!

Warning



```
> log(-1)
[1] NaN
Warning message:
In log(-1) : NaNs produced
```



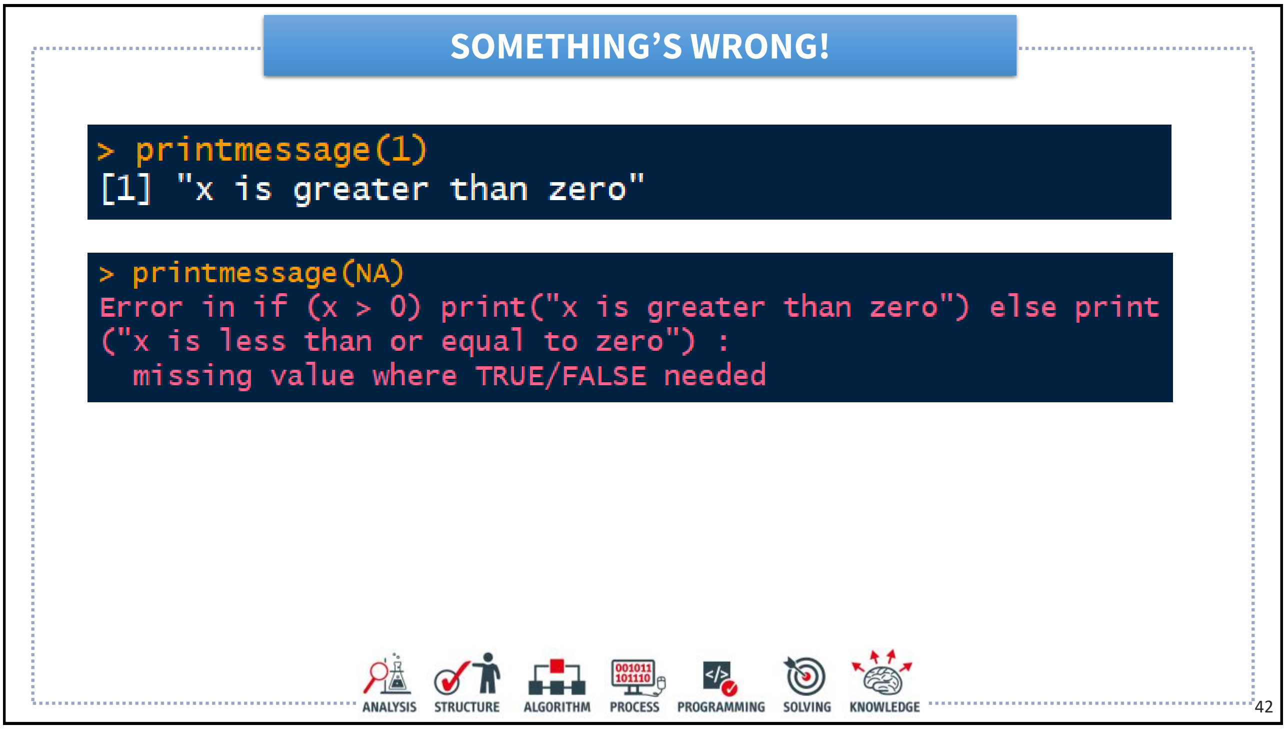
SOMETHING’S WRONG!

```
printmessage <- function(x) {
    if(x > 0)
        print("x is greater than zero")
    else
        print("x is less than or equal to zero")
    invisible(x)
}
```

ANALYSIS
STRUCTURE
ALGORITHM
PROCESS
PROGRAMMING
SOLVING
KNOWLEDGE

41

41



SOMETHING’S WRONG!

```
> printmessage(1)
[1] "x is greater than zero"
```

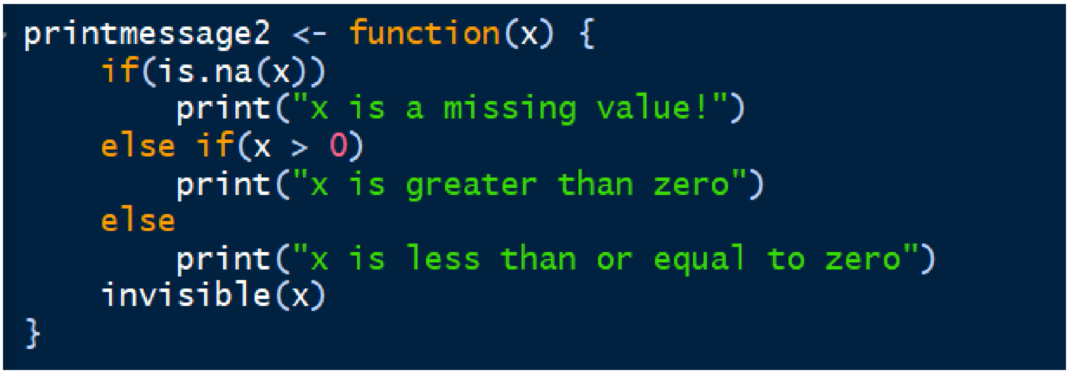
```
> printmessage(NA)
Error in if (x > 0) print("x is greater than zero") else print
("x is less than or equal to zero") :
  missing value where TRUE/FALSE needed
```

ANALYSIS
STRUCTURE
ALGORITHM
PROCESS
PROGRAMMING
SOLVING
KNOWLEDGE

42

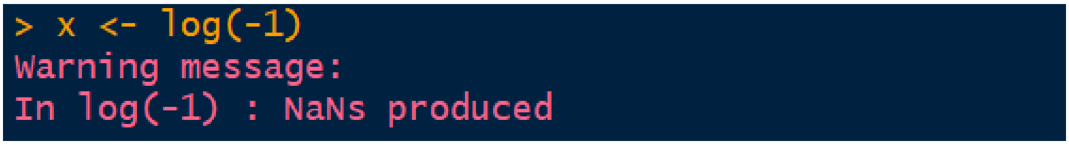
42

SOMETHING'S WRONG!

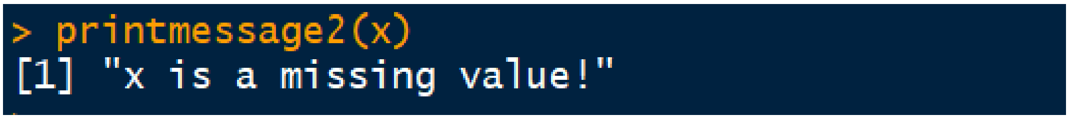


```
printmessage2 <- function(x) {
    if(is.na(x))
        print("x is a missing value!")
    else if(x > 0)
        print("x is greater than zero")
    else
        print("x is less than or equal to zero")
    invisible(x)
}
```

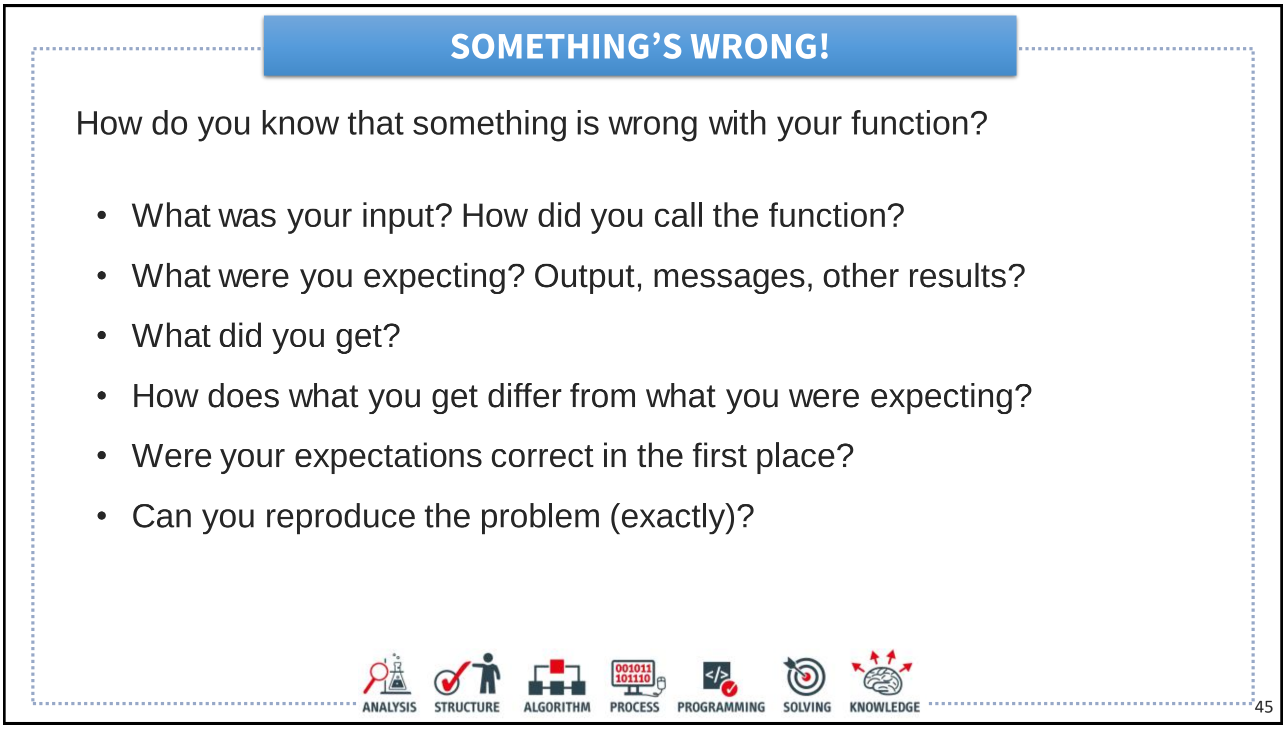
SOMETHING'S WRONG!



```
> x <- log(-1)
Warning message:
In log(-1) : NaNs produced
```



```
> printmessage2(x)
[1] "x is a missing value!"
```



SOMETHING'S WRONG!

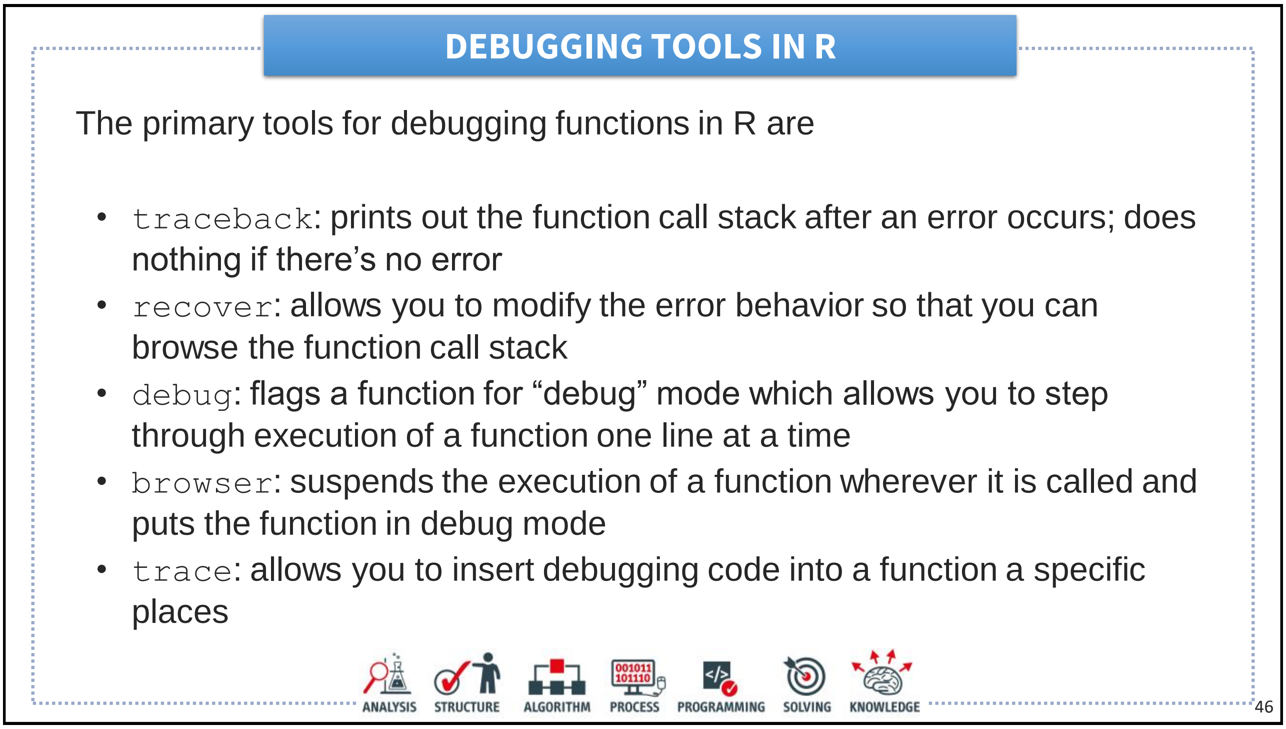
How do you know that something is wrong with your function?

- What was your input? How did you call the function?
- What were you expecting? Output, messages, other results?
- What did you get?
- How does what you get differ from what you were expecting?
- Were your expectations correct in the first place?
- Can you reproduce the problem (exactly)?

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

45

45



DEBUGGING TOOLS IN R

The primary tools for debugging functions in R are

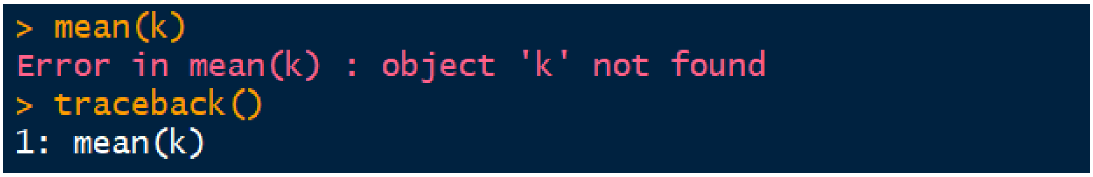
- `traceback`: prints out the function call stack after an error occurs; does nothing if there's no error
- `recover`: allows you to modify the error behavior so that you can browse the function call stack
- `debug`: flags a function for "debug" mode which allows you to step through execution of a function one line at a time
- `browser`: suspends the execution of a function wherever it is called and puts the function in debug mode
- `trace`: allows you to insert debugging code into a function a specific places

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

46

46

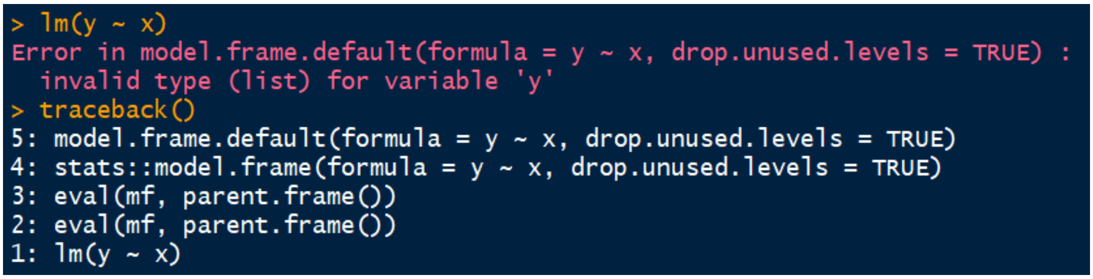
traceback



```
> mean(k)
Error in mean(k) : object 'k' not found
> traceback()
1: mean(k)
```

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

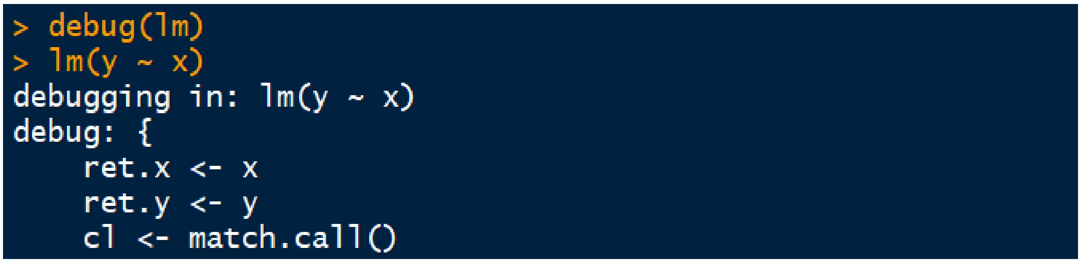
traceback



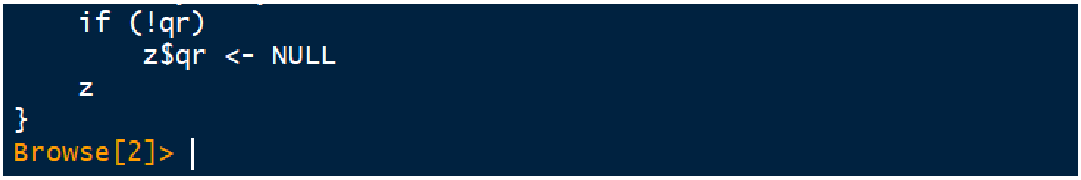
```
> lm(y ~ x)
Error in model.frame.default(formula = y ~ x, drop.unused.levels = TRUE) :
  invalid type (list) for variable 'y'
> traceback()
5: model.frame.default(formula = y ~ x, drop.unused.levels = TRUE)
4: stats::model.frame(formula = y ~ x, drop.unused.levels = TRUE)
3: eval(mf, parent.frame())
2: eval(mf, parent.frame())
1: lm(y ~ x)
```

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

debug



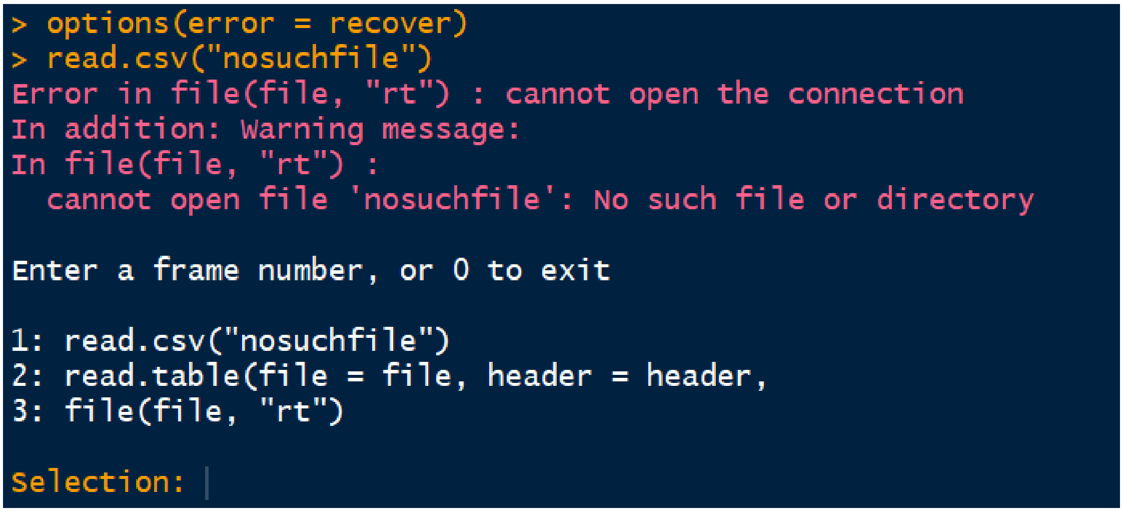
```
> debug(lm)
> lm(y ~ x)
debugging in: lm(y ~ x)
debug: {
    ret.x <- x
    ret.y <- y
    cl <- match.call()
```



```
    if (!qr)
        z$qr <- NULL
    z
}
Browse[2]> |
```

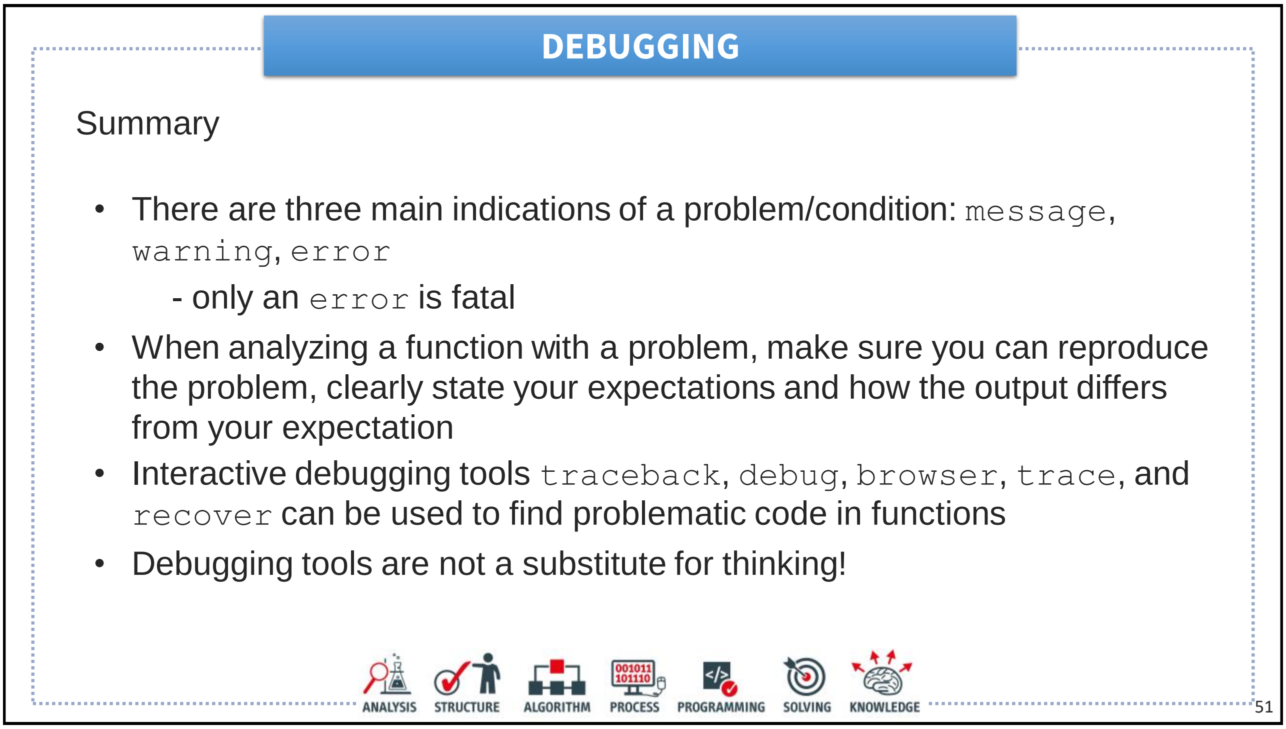
ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE

recover



```
> options(error = recover)
> read.csv("nosuchfile")
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'nosuchfile': No such file or directory

Enter a frame number, or 0 to exit

1: read.csv("nosuchfile")
2: read.table(file = file, header = header,
3: file(file, "rt")

Selection: |
```

ANALYSIS STRUCTURE ALGORITHM PROCESS PROGRAMMING SOLVING KNOWLEDGE



DEBUGGING

Summary

- There are three main indications of a problem/condition: `message`, `warning`, `error`
  - only an `error` is fatal
- When analyzing a function with a problem, make sure you can reproduce the problem, clearly state your expectations and how the output differs from your expectation
- Interactive debugging tools `traceback`, `debug`, `browser`, `trace`, and `recover` can be used to find problematic code in functions
- Debugging tools are not a substitute for thinking!