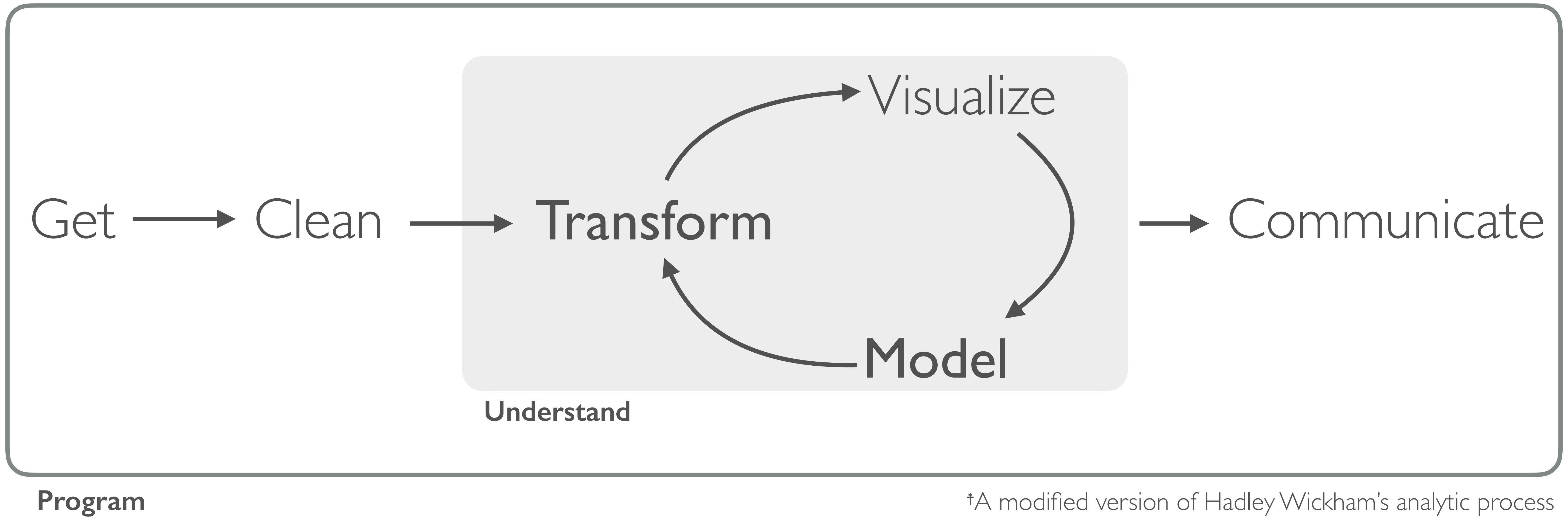
DEVELOPING FUNCTIONS



†A modified version of Hadley Wickham's analytic process

"Writing good functions is a lifetime journey."

– Hadley Wickham

WHY FUNCTIONS ARE GOOD

Writing a function has three big advantages over using copy-and-paste:

- You can give a function an evocative name that makes your code easier to understand.
- As requirements change, you only need to update code in one place, instead of many.
- You eliminate the chance of making incidental mistakes when you copy and paste (i.e. updating a variable name in one place, but not in another).

Functions allow you to automate common tasks

PREREQUISITES

PREREQUISITES

NA - just base R

FUNDAMENTALS

WHEN TO WRITE FUNCTIONS

```
df <- data.frame(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$b, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

You should consider writing a function whenever you've copied and pasted a block of code more than twice.

Can you spot the error?

WHEN TO WRITE FUNCTIONS

```
df <- data.frame(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$b, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

You should consider writing a function whenever you've copied and pasted a block of code more than twice.

Can you spot the error?

DEFINING YOUR OWN FUNCTION

```
my_fun <- function(arg1, arg2) {
  body
}
```

Functions have 3 parts:

1. formals (aka arguments)
2. body (code inside the function)
3. environment

DEFINING YOUR OWN FUNCTION

```
pv <- function(FV, r, n) {
  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}
```

Functions have 3 parts:

1. formals (aka arguments)
2. body (code inside the function)
3. environment

ANATOMY OF A FUNCTION

```
pv <- function(FV, r, n) {
  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}

formals(pv)
$FV
$r
$n

body(pv)
{
    present_value <- FV/(1 + r)^n
    round(present_value, 2)
}

environment(pv)
<environment: R_GlobalEnv>
```

Functions have 3 parts:

1. formals (aka arguments)
2. body (code inside the function)
3. environment

FUNCTION OUTPUT

```
pv <- function(FV, r, n) {
  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}

pv(FV = 1000, r = .08, n = 5)
[1] 680.58

pv2 <- function(FV, r, n) {
  present_value <- FV / (1 + r)^n
  return(present_value)
  round(present_value, 2)
}

pv2(1000, .08, 5)
[1] 680.5832
```

What gets returned from a function is either:

1. The last expression evaluated

2. return(value), which forces the function to stop execution and return value

FUNCTION OUTPUT

```r
pv <- function(FV, r, n) {
  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}

pv(FV = 1000, r = .08, n = 5)
[1] 680.58

pv2 <- function(FV, r, n) {
  present_value <- FV / (1 + r)^n
  return(present_value)
  round(present_value, 2)
}

pv2(1000, .08, 5)
[1] 680.5832
```

What gets returned from a function is either:

1. The last expression evaluated
2. return(value), which forces the function to stop execution and return value

Note the differences in how we call these functions. Why do both cases work?

YOUR TURN!

- *Define a function titled* **`ratio`** *that takes arguments* **`x`** *and* **`y`** *and returns their ratio,* **`x / y`**

- *Call* **`ratio()`** *with arguments 3 and 4*

SOLUTION

```
ratio <- function(x, y) {
  x / y
}

ratio(3, 4)
[1] 0.75
```

HANDLING ARGUMENTS

CALLING ARGUMENTS IN DIFFERENT WAYS

```
pv(FV = 1000, r = .08, n = 5)
[1] 680.58
```

Using argument names

```
pv(1000, .08, 5)
[1] 680.58
```

positional matching

```
pv(r = .08, FV = 1000, n = 5)
[1] 680.58
```

must use names if you change order otherwise…

```
pv(.08, 1000, 5)
[1] 0
```

error or incorrect computation will occur

```
pv(1000, .08)
Error in pv(1000, 0.08) : argument "n" is missing, with no default
```

missing arguments results in error

SETTING DEFAULT ARGUMENTS

```
pv <- function(FV, r, n = 5) {
  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}

pv(1000, .08)
[1] 680.58

PV(1000, .08, n = 3)
[1] 793.83
```

We can set default argument values

now if we do not call the argument the default is used

and we can change the default simply by specifying an n value

ORDERING ARGUMENTS

Ordering arguments in your functions is important:

- positional matching
- pipe (%>%) operator

```
my_fun <- function(data, arg2, arg3 = 5) {
  body
}
```

General rules:

- Data argument first
- First couple arguments require specifying
- Later arguments have defaults

ORDERING ARGUMENTS

Ordering arguments in your functions is important:

- positional matching
- pipe (%>%) operator

```
top_n <- function(x, n, wt) {
  body
}

# allows you to call this function
top_n(df, 5)
df %>% top_n(5)
df %>% top_n(5, var2)
```

General rules:

- x is the data argument
- x & n require being defined
- if wt is not specified it defaults to using the last column in the data frame (x)

YOUR TURN!

Earlier in these slides you saw the following code duplicated:

```
(df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
```

Can you write a function called **`rescale`** *that takes argument* **`x`** *and executes this code?*

Test it on the vector provided in your .R script

SOLUTION

```
rescale <- function(x){
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}

rescale(vec1)
[1] 0.2704415 0.8299695 0.4060968 0.9358038 1.0000000 0.0000000 0.5392146
[8] 0.9463095 0.5652837 0.4593287
```

YOUR TURN!

Now add an argument to **`rescale`** *that allows you to round the output to a specified decimal. Set the default to 2.*

SOLUTION

```
rescale <- function(x, digits = 2){
  rng <- range(x, na.rm = TRUE)
  scaled <- (x - rng[1]) / (rng[2] - rng[1])
  round(scaled, digits = digits)
}

rescale(vec1)
[1] 0.27 0.83 0.41 0.94 1.00 0.00 0.54 0.95 0.57 0.46

rescale(vec1, 3)
[1] 0.270 0.830 0.406 0.936 1.000 0.000 0.539 0.946 0.565 0.459
```

YOUR TURN!

Now let's move the `na.rm = TRUE` *argument into the functions formals so that the user can specify whether or not they want to remove NAs. Set the default to* **TRUE**.

SOLUTION

Showing how many missing values were removed

```
rescale <- function(x, digits = 2, na.rm = TRUE){
  if(isTRUE(na.rm)) x <- na.omit(x)
  rng <- range(x)
  scaled <- (x - rng[1]) / (rng[2] - rng[1])
  round(scaled, digits = digits)
}

vec1 <- c(NA, vec1)
rescale(vec1)
[1] 0.27 0.83 0.41 0.94 1.00 0.00 0.54 0.95 0.57 0.46
attr(,"na.action")
[1] 1
attr(,"class")
[1] "omit"
```

SOLUTION

Hiding how many missing values were removed

```
rescale <- function(x, digits = 2, na.rm = TRUE){
  if(isTRUE(na.rm)) x <- x[!is.na(x)]
  rng <- range(x)
  scaled <- (x - rng[1]) / (rng[2] - rng[1])
  round(scaled, digits = digits)
}

rescale(vec1)
[1] 0.27 0.83 0.41 0.94 1.00 0.00 0.54 0.95 0.57 0.46
```

YOUR TURN!

Now try to apply the `rescale` *function across each variable in the* `mtcars` *data set.*

Hint: try using one of the `map` *functions from the* `purrr` *package.*

SOLUTION

You can now apply this function over a data frame, list, matrix with the map function

```
library(purrr)

mtcars %>%
  map_df(rescale)

# A tibble: 32 × 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1   0.45   0.5  0.22  0.20  0.53  0.28  0.23     0     1   0.5  0.43
2   0.45   0.5  0.22  0.20  0.53  0.35  0.30     0     1   0.5  0.43
3   0.53   0.0  0.09  0.14  0.50  0.21  0.49     1     1   0.5  0.00
4   0.47   0.5  0.47  0.20  0.15  0.44  0.59     1     0   0.0  0.00
5   0.35   1.0  0.72  0.43  0.18  0.49  0.30     0     0   0.0  0.14
6   0.33   0.5  0.38  0.19  0.00  0.50  0.68     1     0   0.0  0.00
7   0.17   1.0  0.72  0.68  0.21  0.53  0.16     0     0   0.0  0.43
```

INVALID PARAMETERS

INVALID PARAMETERS

For functions that will be re-used, and especially for those used by someone other than the creator, it is good to check the validity of the arguments.

```
my_fun <- function(data, arg2, arg3 = 5) {

  if(condition) {
    message or warning
  }

  body
}
```

<u>Common issues</u>:

- Making sure data is in the right structure (i.e. df, list, vector)
- Are the argument inputs the right class (i.e. numeric, character)
- Are the argument inputs within the proper boundary limits

INVALID PARAMETERS

Our pv function works on a vector of future values, not data frames, lists, or matrices. Let's add a warning in case a user tries to feed it a non-atomic vector.

INVALID PARAMETERS

Our pv function works on a vector of future values, not data frames, lists, or matrices. Let's add a warning in case a user tries to feed it a non-atomic vector.

```
pv <- function(FV, r, n = 5) {

  if(!is.atomic(FV) {
    stop('FV must be an atomic vector')
  }

  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}
```

- Check if class of FV is something other than a vector (be careful with is.vector - use is.atomic instead)
- If so, stop, return an error, and the specified message

INVALID PARAMETERS

Our pv function works on a vector of future values, not data frames, lists, or matrices. Let's add a warning in case a user tries to feed it a non-vector.

```
fv_l <- list(fv1 = 800,
             fv2 = 900,
             fv3 = 1100)


pv(fv_l, 0.08)
Error in pv(fv_l, 0.08) : FV must be an
atomic vector
```

- Now when we execute pv on a non-atomic vector we get an error output

INVALID PARAMETERS

Now let's add tests for the type of class input.

```
pv <- function(FV, r, n = 5) {

  if(!is.atomic(FV)) {
    stop('FV must be an atomic vector')
  }

  if(!is.numeric(FV) | !is.numeric(r) | !is.numeric(n)){
    stop('This function only works for numeric inputs!\n',
         'You have provided objects of the following classes:\n',
         'FV: ', class(FV), '\n',
         'r: ', class(r), '\n',
         'n: ', class(n))
  }

  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}
```

Now we test for

- data type
- argument class

and both of these will provide warnings if violated

INVALID PARAMETERS

Now let’s add tests for the type of class input.

```
pv(FV = "1000", .08, n = 5)
Error in pv(FV = "1000", 0.08, n = 5) :
  This function only works for numeric inputs!
You have provided objects of the following classes:
FV: character
r: numeric
n: numeric
```

Now we test for

- data type
- argument class

and both of these will provide warnings if violated

INVALID PARAMETERS

What else can you think of?

INVALID PARAMETERS

What else can you think of? What about abnormal interest rate ranges?

```
pv <- function(FV, r, n = 5) {

  if(!is.atomic(FV)) {
    stop('FV must be an atomic vector')
  }

  if(!is.numeric(FV) | !is.numeric(r) | !is.numeric(n)){
    stop('This function only works for numeric inputs!\n',
         'You have provided objects of the following classes:\n',
         'FV: ', class(FV), '\n',
         'r: ', class(r), '\n',
         'n: ', class(n))
  }

  if(r < 0 | r > .25) {
    message('The input for r exceeds the normal\n',
            'range for interest rates (0-25%)')
  }

  present_value <- FV / (1 + r)^n
  round(present_value, 2)
}
```

If we add a message() this allows us to:

- notify the user of something
- while still executing the code

INVALID PARAMETERS

What else can you think of? What about abnormal interest rate ranges?

```
pv(FV = 1000, r = .28, n = 5)
The input for r exceeds the normal
range for interest rates (0-25%)
[1] 1292.36
```

If we add a message() this allows us to:

- notify the user of something
- while still executing the code

YOUR TURN!

Going back to the rescale function:

```
rescale <- function(x, digits = 2, na.rm = TRUE){
  if(isTRUE(na.rm)) x <- x[!is.na(x)]
  rng <- range(x)
  scaled <- (x - rng[1]) / (rng[2] - rng[1])
  round(scaled, digits = digits)
}
```

YOUR TURN!

Going back to the rescale function add conditional statements to check and provide appropriate errors or messages for:

- *making sure* **`x`** *input is a numeric vector*
- **`digits`** *input is a numeric vector of one element*
- **`na.rm`** *input is a single logical input*

SOLUTION

```r
rescale <- function(x, digits = 2, na.rm = TRUE){
  # ensure argument inputs are valid
  if(!is.numeric(x)) {
    stop('x must be an atomic numeric vector')
  }
  if(!is.numeric(digits) | length(digits) > 1) {
    stop('digits must be a numeric vector of one element')
  }
  if(!is.logical(na.rm)) {
    stop('na.rm must be logical input (TRUE or FALSE)')
  }

  if(isTRUE(na.rm)) x <- x[!is.na(x)]
  rng <- range(x)
  scaled <- (x - rng[1]) / (rng[2] - rng[1])
  round(scaled, digits = digits)
}
```

```r
rescale <- function(x, digits = 2, na.rm = TRUE){
  # ensure argument inputs are valid
  if(!is.numeric(x)) {
    stop('x must be an atomic numeric vector')
  }
  if(!is.numeric(digits) | length(digits) > 1) {
    stop('digits must be a numeric vector of one element')
  }
  if(!is.logical(na.rm)) {
    stop('na.rm must be logical input (TRUE or FALSE)')
  }
```

SOLUTION

```
rescale(c(letters))

rescale(vec1, digits = c(1, 2))

rescale(vec1, na.rm = "false")
```

OTHER NOTES

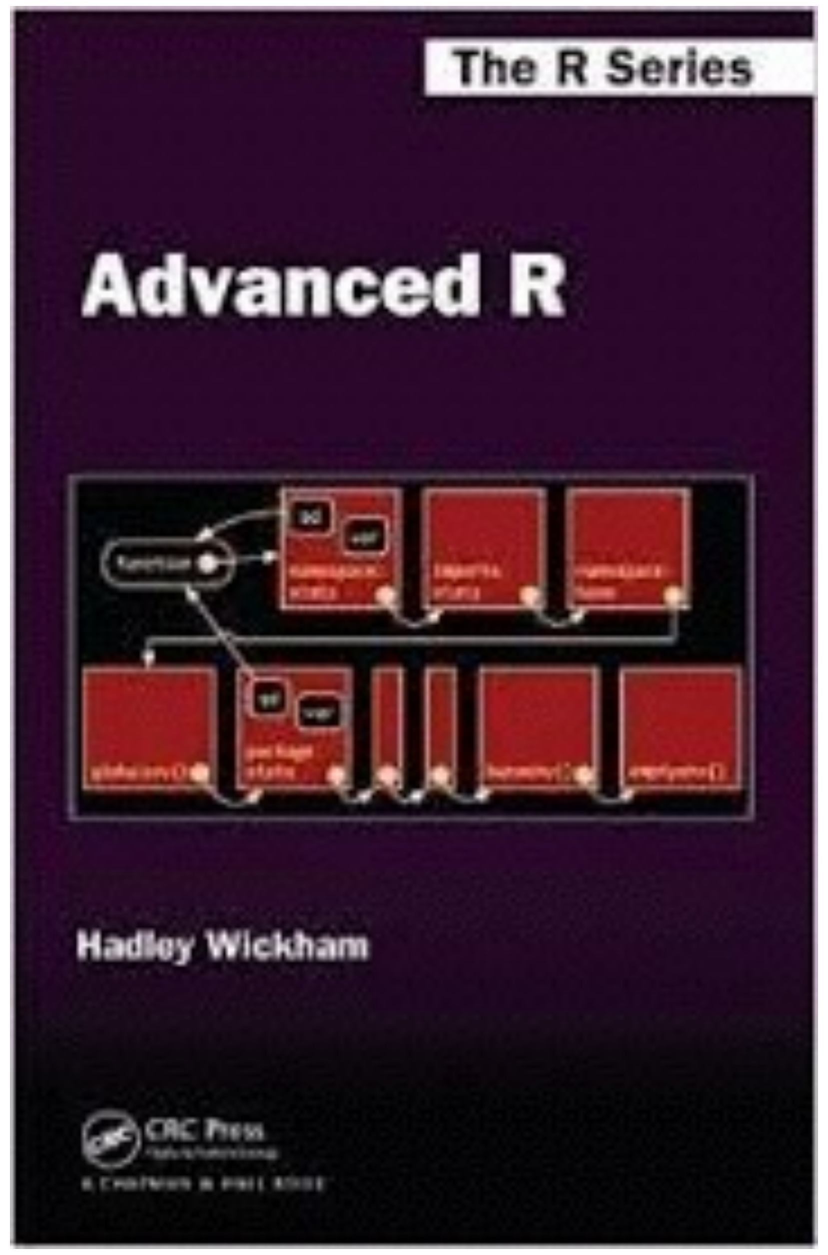
LAZY EVALUATION

R functions perform "lazy" evaluation in which arguments are only evaluated if required in the body of the function

```
lazy <- function(x, y = NULL) {
  if(!is.null(y)) {
    return(x * 2 + y)
  }
  x * 2
}

lazy(4)
[1] 8
lazy(4, 1)
[1] 9
```

This allows us to only evaluate arguments if inputs are included.



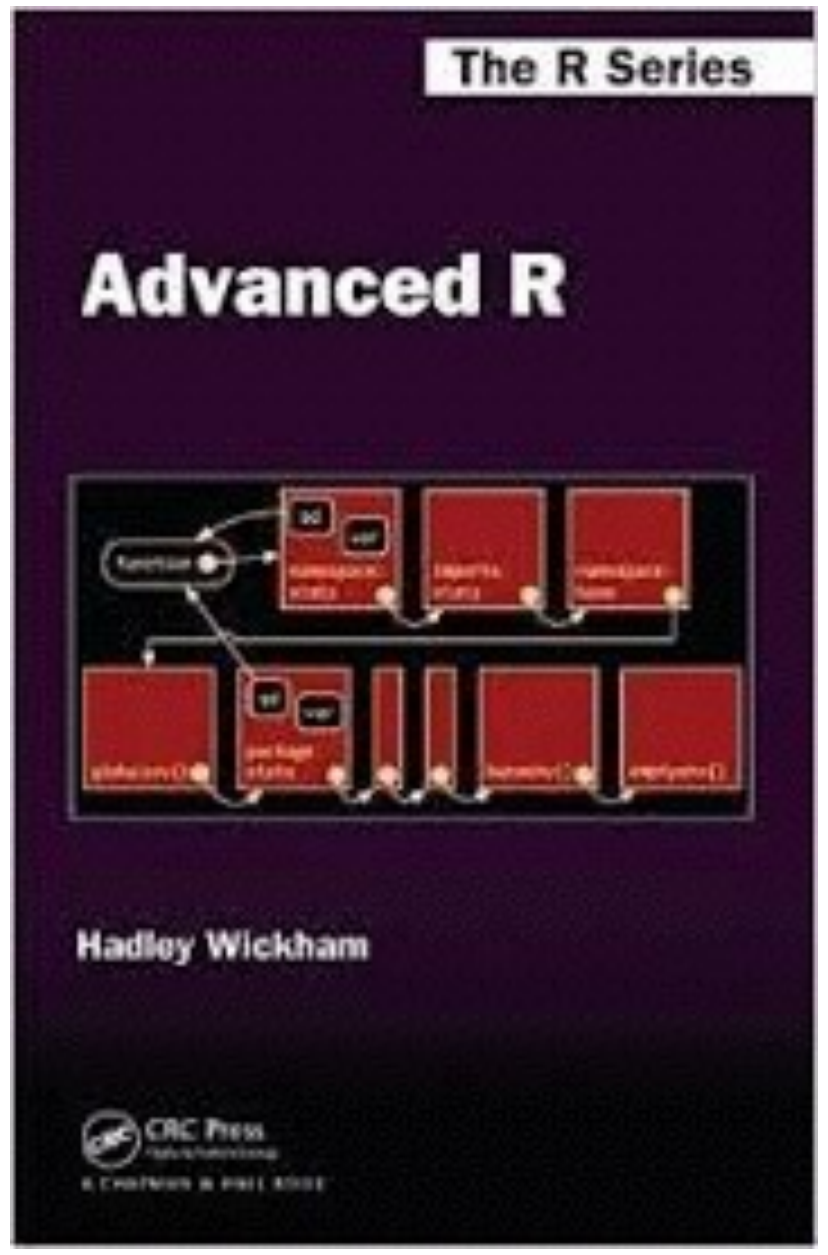
LEXICAL SCOPING RULES

R functions will first look inside the function to identify all variables being called. If variables do not exist R will look one level up.

```
y <- 2
scoping <- function(x) {
  if(!is.null(y)) {
    return(x * 2 + y)
  }
  x * 2
}

scoping(4)
[1] 10
```

This is useful when you start to embed functions within functions.



NAMING CONVENTIONS

Naming your functions is important - be descriptive

- Can you think of a better name than pv?

Common naming conventions within arguments include:

- x, y, z: vectors
- w: a vector of weights
- df: a data frame
- i, j: numeric indices (typically for rows and columns)
- n: length, or number of rows
- p: number of columns

Examining existing R functions will help you understand common practices

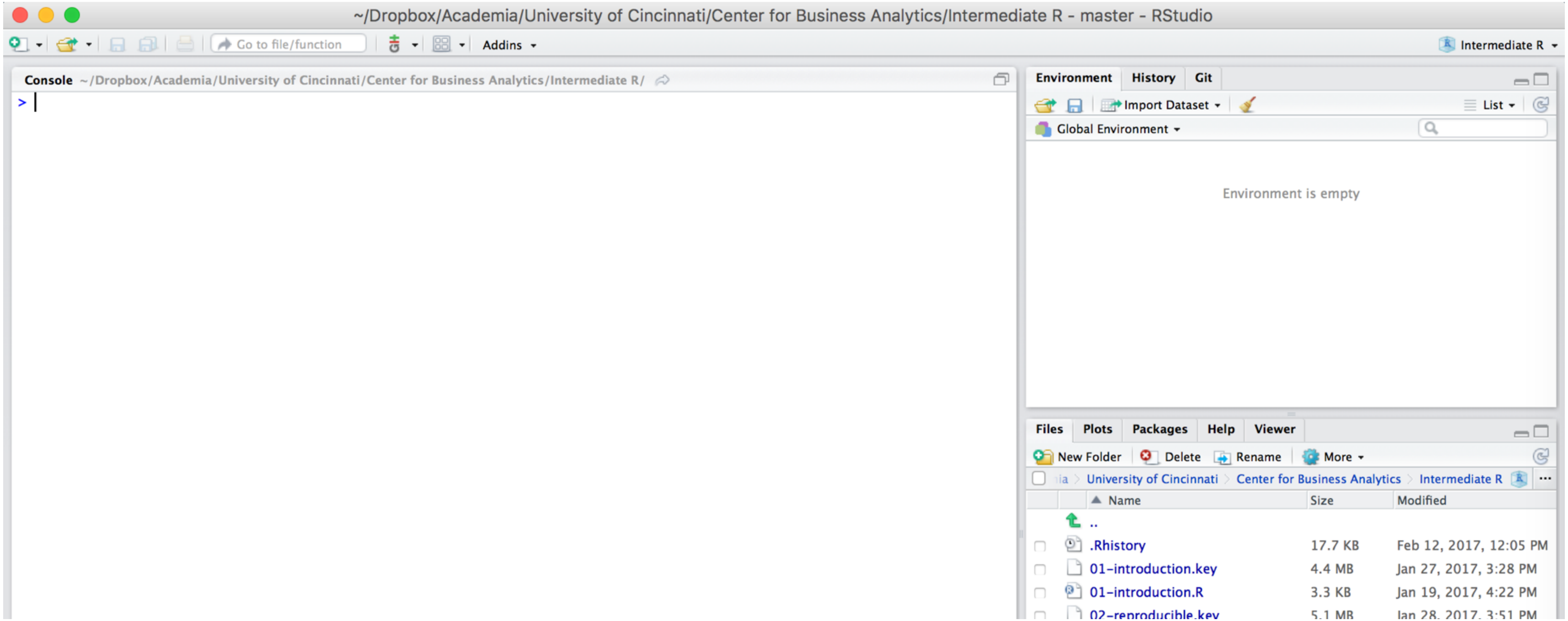
PRACTICE WRITING FUNCTIONS

Create the following vector x:

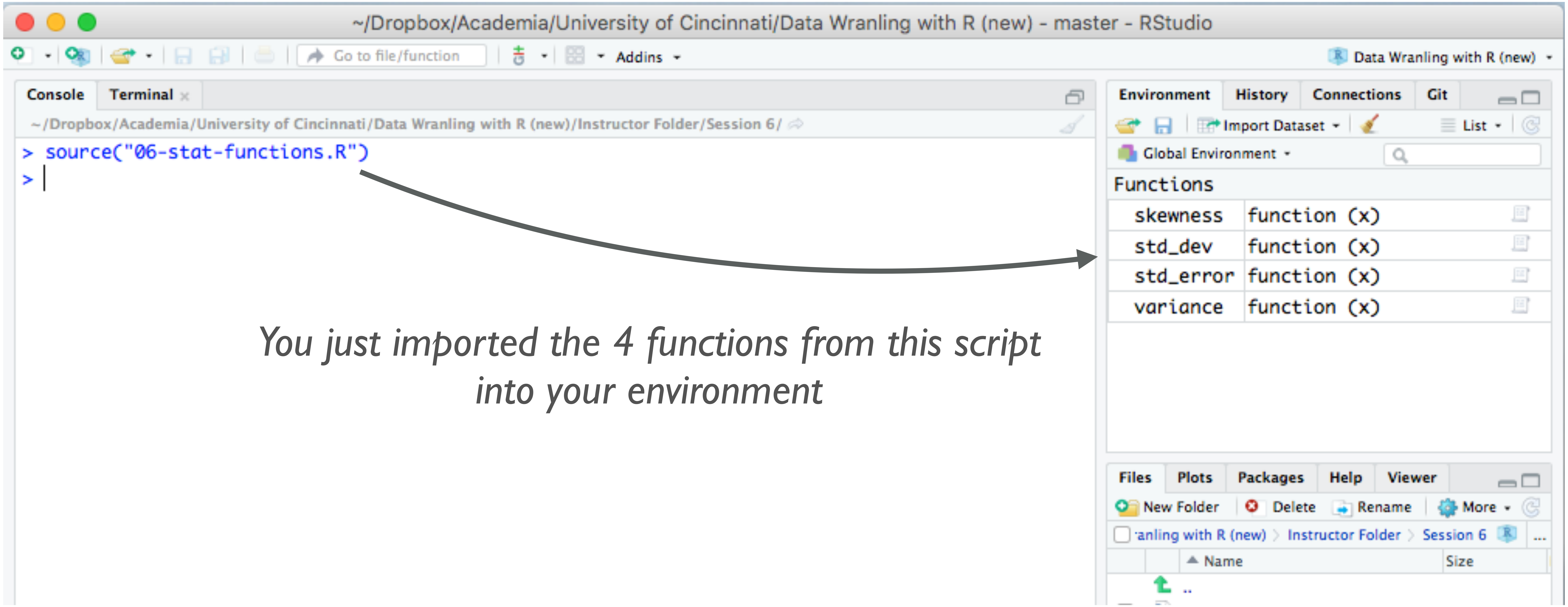
```
set.seed(123)
x <- rlnorm(100)
```

Now create the functions in your .R script that will compute the variance, standard deviation, standard error, an skewness.

SOURCING YOUR OWN FUNCTIONS

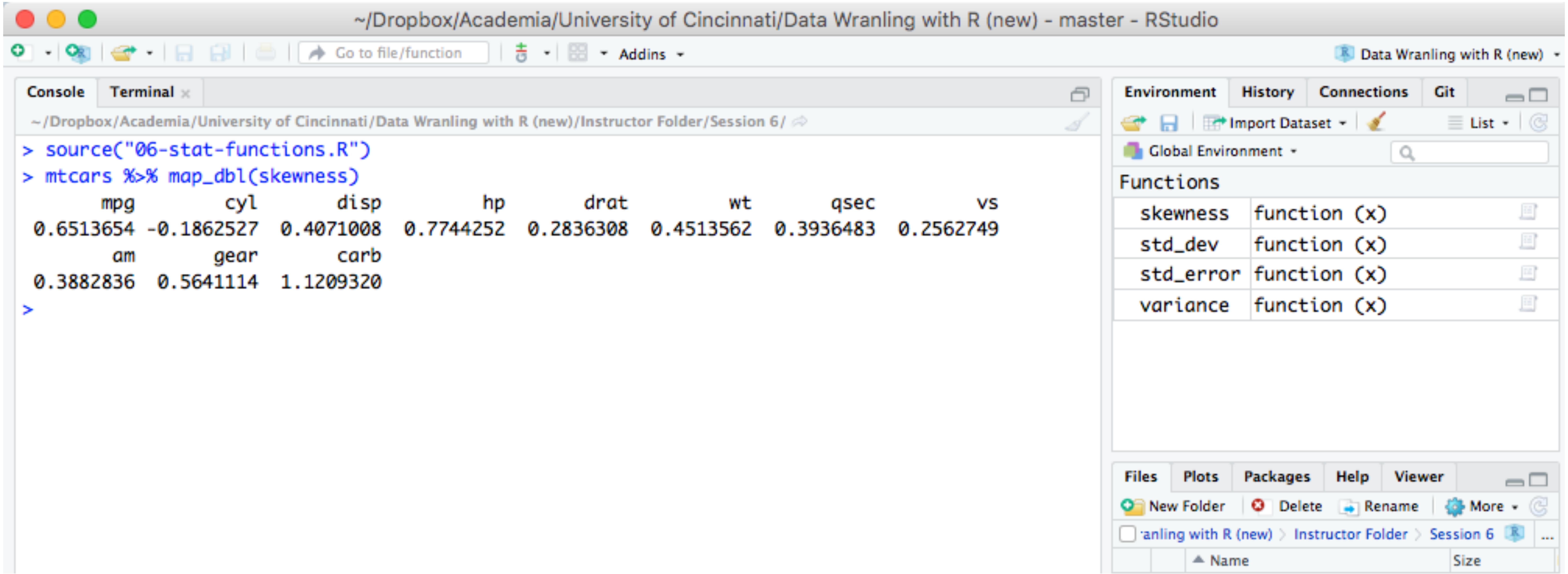


SOURCING YOUR OWN FUNCTIONS



You just imported the 4 functions from this script into your environment

SOURCING YOUR OWN FUNCTIONS

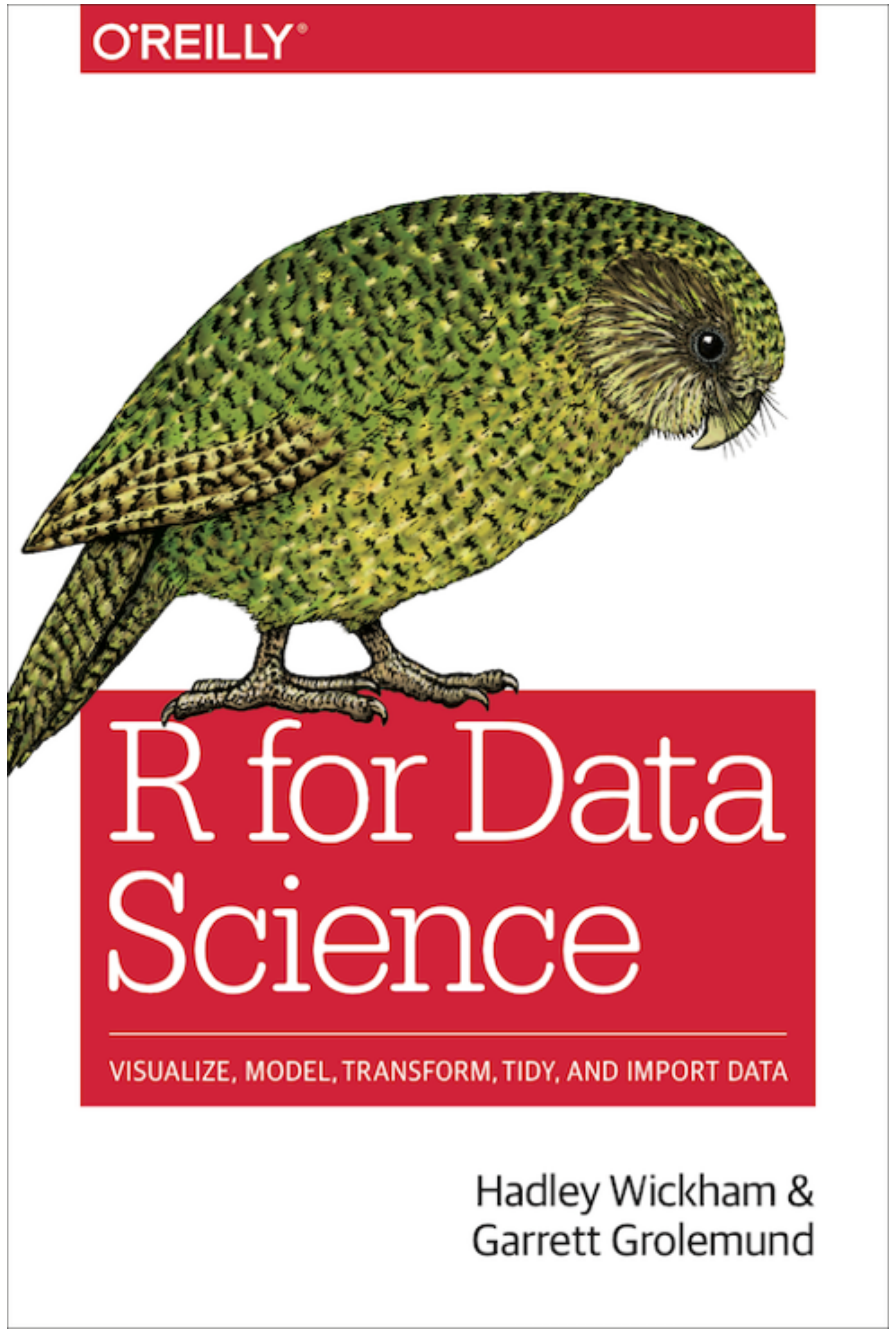


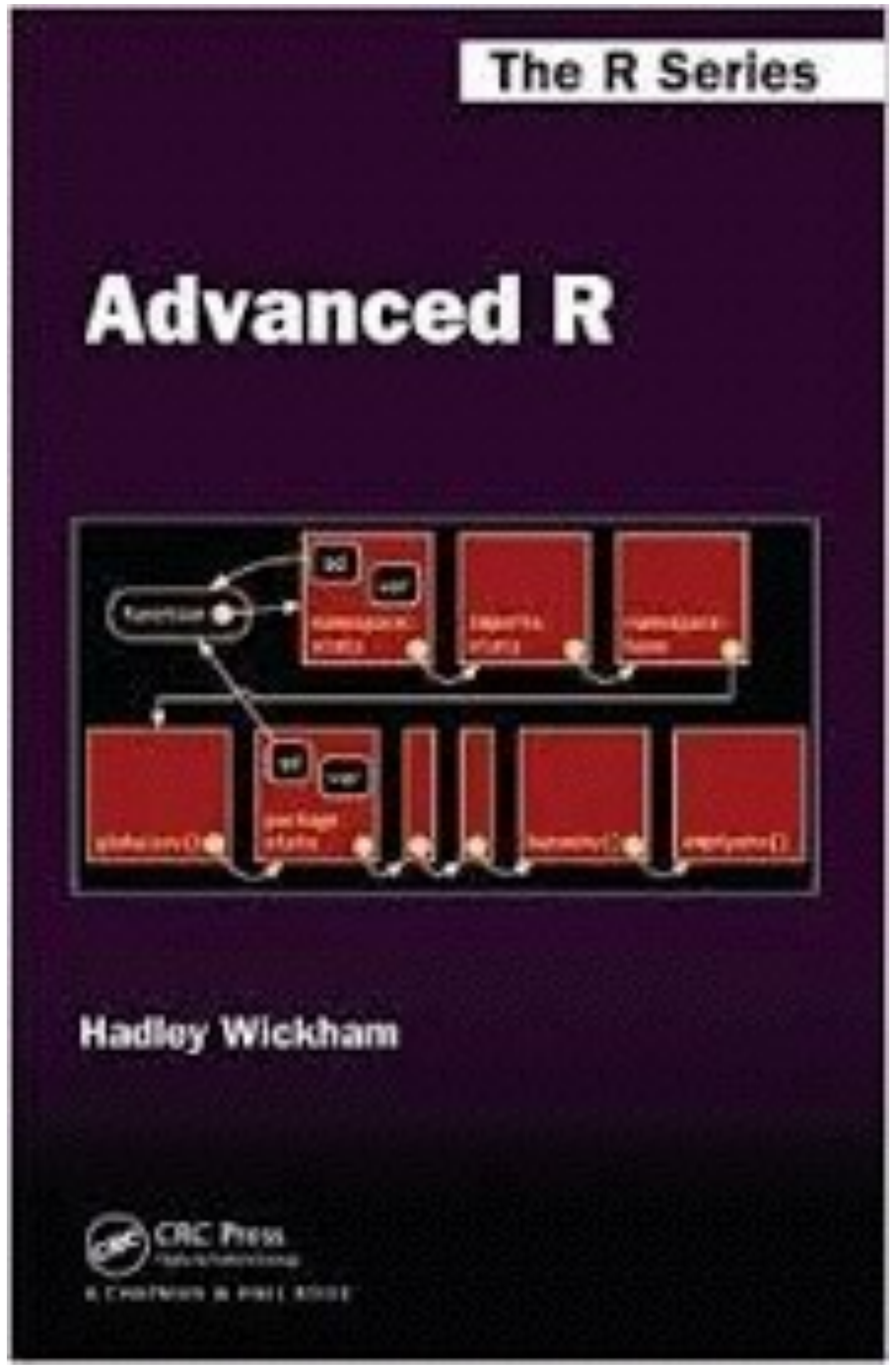
PRACTICE APPLYING FUNCTIONS

Source the functions in the `06-stat-functions.R` *file and practice applying these functions to each variable in the* `mtcars` *data set by using* `map` *functions.*

SO LITTLE TIME!

LEARN MORE


O'REILLY
R for Data Science
VISUALIZE, MODEL, TRANSFORM, TIDY, AND IMPORT DATA
Hadley Wickham &
Garrett Grolemund


The R Series
Advanced R
Hadley Wickham
CRC Press

WHAT TO REMEMBER

FUNCTIONS TO REMEMBER

Operator/Function	Description
`function`	Create a function
`formals, body, environment`	Get anatomy of an existing function
`stop, stopif, message`	Create warnings or messages
`source`	Source a .R script (easy way to save common functions you use and access them whenever you desire)