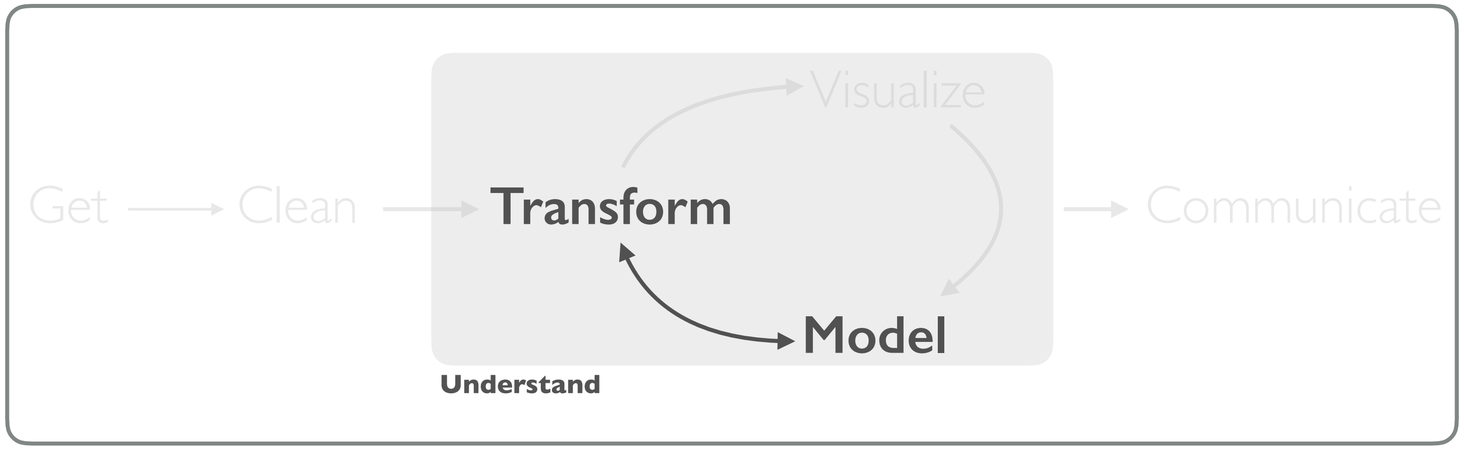
CONTROL STATEMENTS & ITERATION



†A modified version of Hadley Wickham's analytic process

“Great design is iteration of good design.”

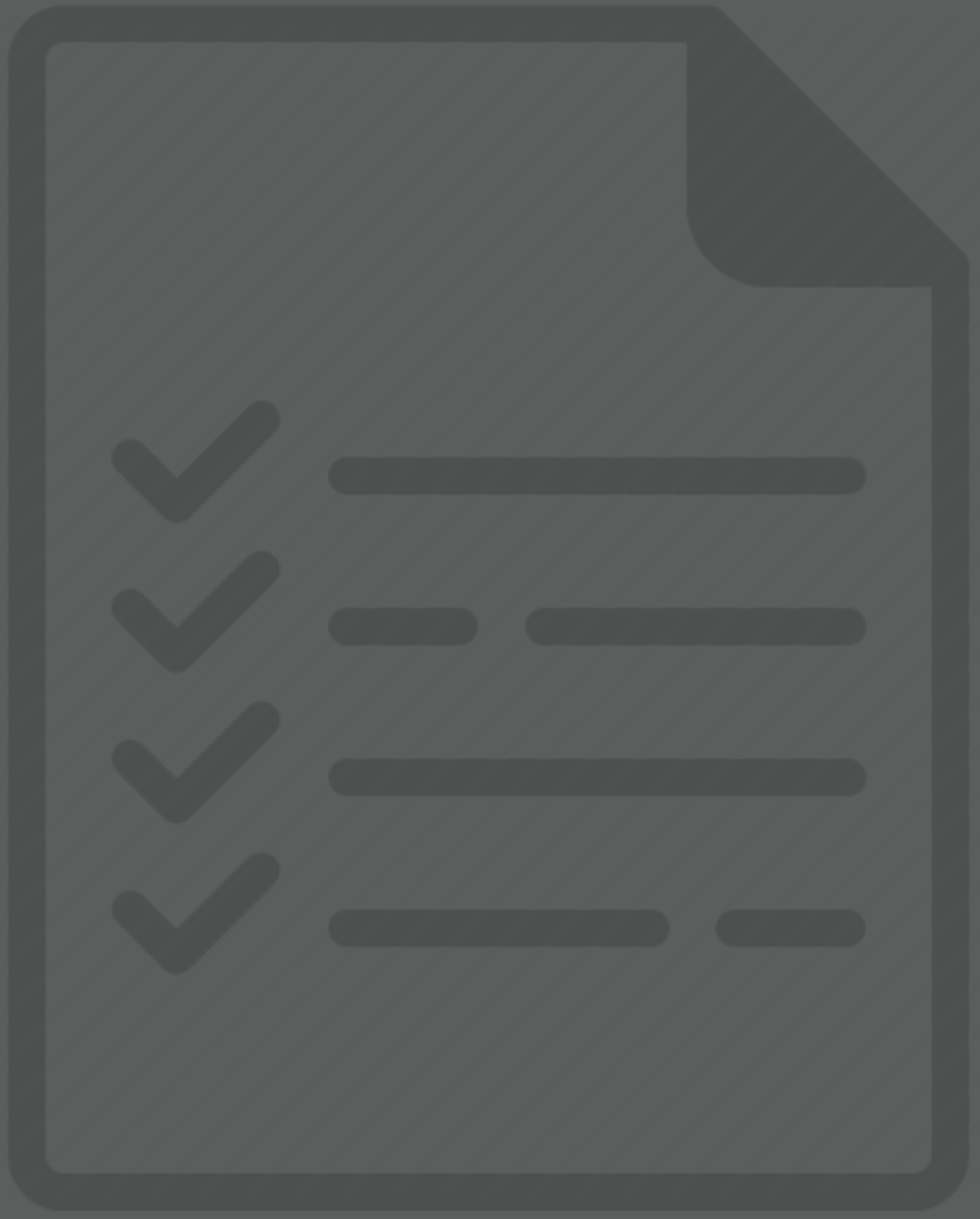
– M. Cobanli

CONTROL STATEMENTS & ITERATION

- Reducing code duplication has three main benefits:
  - It's easier to see the intent of your code, because your eyes are drawn to what's different, not what stays the same.
  - It's easier to respond to changes in requirements. As your needs change, you only need to make changes in one place, rather than remembering to change every place that you copied-and-pasted the code.
  - You're likely to have fewer bugs because each line of code is used in more places.

Control statements and iteration functions provide two means to reduce code duplication

PREREQUISITES



PACKAGE PREREQUISITE

```
library(tidyverse)
Loading tidyverse: ggplot2
Loading tidyverse: tibble
Loading tidyverse: tidyr
Loading tidyverse: readr
Loading tidyverse: purrr
Loading tidyverse: dplyr
Conflicts with tidy packages ----------------------------------------------
filter(): dplyr, stats
lag():    dplyr, stats
```

CONTROL STATEMENTS

SYNTAX OF AN `if` STATEMENT

```
if (test_expression) {
  statement
}
```

Evaluates if the test expression is TRUE

If TRUE: execute statement

If FALSE: do nothing

SYNTAX OF AN `if` STATEMENT

```
x <- c(8, 3, -2, 5)

if(x < 0) {
  print("x contains a negative number")
}
```

What do you expect to happen?

SYNTAX OF AN `if` STATEMENT

```
x <- c(8, 3, -2, 5)

if(x < 0) {
  print("x contains a negative number")
}
Warning message:
In if (x < 0) print("Negative") :
  the condition has length > 1 and only the first
  element will be used
```

Trick question - we actually get a warning

The if statement looks for a single conditional value. Here, `x < 0` will return a vector of 4 conditional values.

SYNTAX OF AN `if` STATEMENT

```
x <- c(8, 3, -2, 5)

if(any(x < 0)) {
  print("x contains a negative number")
}
```

Instead, we can assess if any x values are less than 0

Now what do you expect to happen?

SYNTAX OF AN `if` STATEMENT

```
x <- c(8, 3, -2, 5)

if(any(x < 0)) {
  print("x contains a negative number")
}
[1] "x contains a negative number"
```

Instead, we can assess if any x values are less than 0

Now what do you expect to happen?

SYNTAX OF AN `if` STATEMENT

```
x <- c(8, 3, -2, 5)

if(any(x < 0)) {
  print("x contains a negative number")
}
[1] "x contains a negative number"
```

Instead, we can assess if any x values are less than 0

Now what do you expect to happen?

Change -2 to 2. Now what happens when you run this code?

SYNTAX OF AN `if...else` STATEMENT

```
if (test_expression) {
  statement 1
} else {
  statement 2
}
```

We can extend an if statement with an else statement.

SYNTAX OF AN `if...else` STATEMENT

```
x <- c(8, 3, -2, 5)

if(any(x < 0)) {
  print("x contains negative number(s)")
} else{
  print("x contains all positive numbers")
}
[1] "x contains negative number(s)"
```

Now, if the test expression is TRUE we print statement 1 and if it is FALSE we print statement 2.

SYNTAX OF AN `if...else` STATEMENT

```
x <- 7

if(x >= 10) {
  print("x exceeds acceptable tolerance levels")
} else if(x >= 0 & x < 10) {
  print("x is within acceptable tolerance
levels")
} else {
  print("x is negative")
}
[1] "x is within acceptable tolerance levels"
```

And we can expand this further to include multiple test expressions.

YOUR TURN - PART 1

In your data folder you have many data sets. You want to create an if else control statement that will look to see if a particular "Month-XX.csv" file exists in this folder. For Part 1, build your your if-else statement so that you create the appropriate file name based on a month input:

```
file_1 <- "Month-"
file_2 <- ".csv"
month <- 13

if (month %in% 1:9) {
 …
} else if (month %in% 10:12) {
 …
} else {
 …
}
```

If **month** equals 1-9 the output should look like "Month-08.csv"

If **month** equals 10-12 the output should look like "Month-12.csv"

If **month** is not 1-12 the output should look like "Invalid month"

SOLUTION

```
file_1 <- "Month-"
file_2 <- ".csv"
month <- 1

if (month %in% 1:9) {
  paste0("data/", file_1, 0, month, file_2)
} else if (month %in% 10:12) {
  paste0("data/", file_1, month, file_2)
} else {
  print("Invalid month")
}
[1] "data/Month-01.csv"
```

```
file_1 <- "Month-"
file_2 <- ".csv"
month <- 13

if (month %in% 1:9) {
  paste0("data/", file_1, 0, month, file_2)
} else if (month %in% 10:12) {
  paste0("data/", file_1, month, file_2)
} else {
  print("Invalid month")
}
[1] "Invalid month"
```

YOUR TURN - PART 2

The `file.exists` *function will check to see if a file exists at the given path. Incorporate* `file.exists` *in your if-else statement so now the output provides a TRUE, FALSE, or "Invalid month" response. Is the December data ("Month-12.csv") present?*

SOLUTION

```
file_1 <- "Month-"
file_2 <- ".csv"
month <- 12

if(month %in% 1:9) {
  file_name <- paste0("data/", file_1, 0, month, file_2)
  file.exists(file_name)
} else if(month %in% 10:12) {
  file_name <- paste0("data/", file_1, month, file_2)
  file.exists(file_name)
} else {
  print("Invalid month")
}
[1] FALSE
```

ITERATION

`for` loops

SYNTAX OF A `for` LOOP

```
for(i in 1:100) {
  <do stuff here with i>
}
```

For each element in a sequence performs defined tasks

SYNTAX OF A `for` LOOP

```
for(i in 1:100) {
  <do stuff here with i>
}
```

For each element in a sequence performs defined tasks

```
for(i in seq_along(x)) {
  <do stuff here with i>
}
```

Alternative - why use `seq_along`?

SYNTAX OF A `for` LOOP

```
years <- 2010:2017

for (i in seq_along(years)) {
  output <- paste("The year is", years[i])
  print(output)
}
```

What does this for loop do?

SYNTAX OF A `for` LOOP

```
years <- 2010:2017

for (i in seq_along(years)) {
  output <- paste("The year is", years[i])
  print(output)
}
[1] "The year is 2010"
[1] "The year is 2011"
[1] "The year is 2012"
[1] "The year is 2013"
[1] "The year is 2014"
[1] "The year is 2015"
[1] "The year is 2016"
[1] "The year is 2017"
```

What does this for loop do?

SYNTAX OF A `for` LOOP

```
result <- vector(mode = "character",
                 length = length(years))

for (i in seq_along(years)) {
  output <- paste("The year is", years[i])
  result[i] <- output
}

result
[1] "The year is 2010" "The year is 2011"
[3] "The year is 2012" "The year is 2013"
[5] "The year is 2014" "The year is 2015"
[7] "The year is 2016" "The year is 2017"
```

When saving results from a for loop we want to:

1. initiate the output outside of the for loop

2. save the results from the for loop to the output

SYNTAX OF A `for` LOOP

```
x <- 7

if(x >= 10) {
  print("x exceeds acceptable tolerance levels")
} else if(x >= 0 & x < 10) {
  print("x is within acceptable tolerance
levels")
} else {
  print("x is negative")
}
[1] "x is within acceptable tolerance levels"
```

Remember this extended if…else statement?

Let’s implement it into a for loop

SYNTAX OF A `for` LOOP

```
x <- c(-1, 7, 8, 11)
tolerance <- vector(mode = "character",
                    length = length(x))

for (i in seq_along(x)) {
  if(x[i] >= 10) {
    value <- "x exceeds acceptable tolerance levels"
  } else if(x[i] >= 0 & x[i] < 10) {
    value <- "x is within acceptable tolerance levels"
  } else {
    value <- "x is negative"
  }
  tolerance[i] <- value
}
```

1. Vector to analyze
2. Initiate output shell
3. For each element in our x
4. Assess test expressions, execute relevant statement, and save as value
5. For that respective element add the value in the corresponding element in out output shell

YOUR TURN

Remember this if-else statement we made?

```
file_1 <- "Month-"
file_2 <- ".csv"
month <- 12

if(month %in% 1:9) {
  file_name <- paste0("data/", file_1, 0, month, file_2)
  file.exists(file_name)
} else if(month %in% 10:12) {
  file_name <- paste0("data/", file_1, month, file_2)
  file.exists(file_name)
} else {
  print("Invalid month")
}
```

YOUR TURN

Develop a for loop that will loop through the months provided and import those .csv files that are present.

```
file_1 <- "Month-"
file_2 <- ".csv"
month <- 1:13

for(i in month) {

  < magic happens here />

}
```

YOUR TURN

Develop a for loop that will loop through the months provided and import those .csv files that are present.

```
file_1 <- "Month-"
file_2 <- ".csv"
month <- 1:13

for(i in month) {

  < magic happens here />

}
```

Run your for loop for **months 1-13**:

- If a particular month is available import it as "df.month.1", "df.month.2", …
- I a particular month is not available, provide the response: "There is no data for month x"
- If a particular month is invalid (i.e. 13), provide the response: "x is an invalid month"

```
file_1 <- "Month-"
file_2 <- ".csv"
month <- 1:13

for(i in month) {

  # create file name
  if(i %in% 1:9) {
    file_name <- paste0("data/", file_1, 0, i, file_2)
  } else if(i %in% 10:12) {
    file_name <- paste0("data/", file_1, i, file_2)
  } else {
    response <- paste(i, "is an invalid month")
    print(response)
    next
  }

  # import data
  if(file.exists(file_name)) {
    df <- read_csv(file_name)
    assign(paste0("df.month.", i), df)
    rm(df)
  } else {
    response <- paste("There is no available data for month", i)
    print(response)
  }
}
```

SOLUTION

1. Create file names.

2. If invalid month, provide response and skip to next iteration.

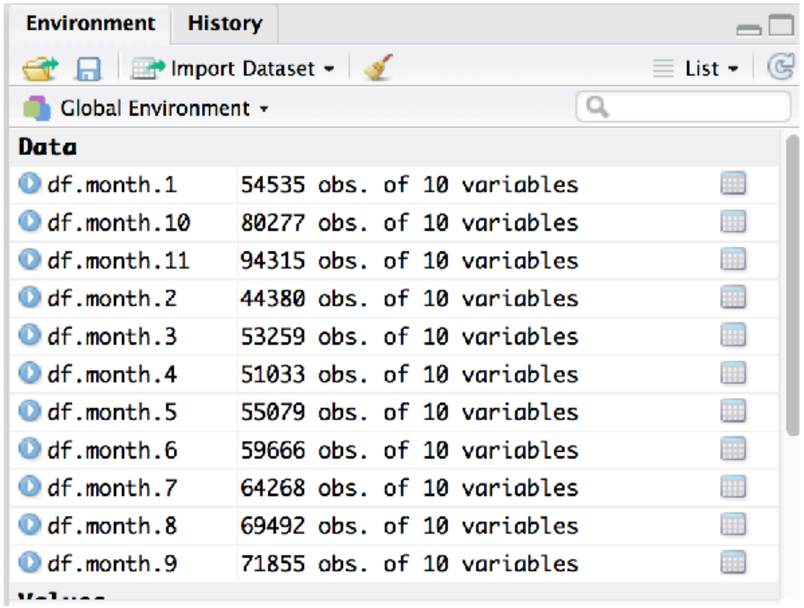
3. If the file exists, import and rename.

4. If the file does not exist for a given month, provide a response

```
file_1 <- "Month-"
file_2 <- ".csv"
month <- 1:13

for(i in month) {

  # create file name
  if(i %in% 1:9) {
    file_name <- paste0("data/", file_1, 0, i, file_2)
  } else if(i %in% 10:12) {
    file_name <- paste0("data/", file_1, i, file_2)
  } else {
    response <- paste(i, "is an invalid month")
    print(response)
    next
  }

  # import data
  if(file.exists(file_name)) {
    df <- read_csv(file_name)
    assign(paste0("df.month.", i), df)
    rm(df)
  } else {
    response <- paste("There is no available data for month", i)
    print(response)
  }
}
```

SOLUTION

When you run this for months 1:13



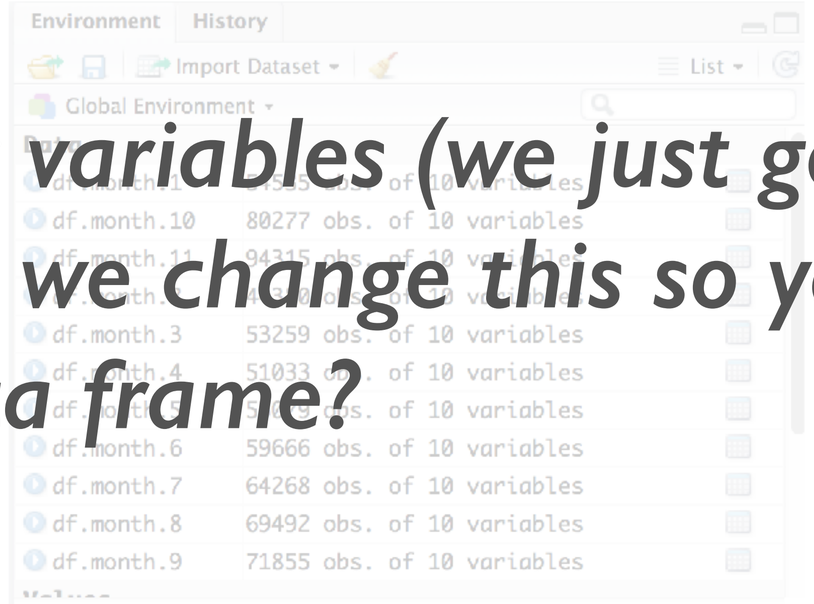
and for months 12 & 13 the response is:

```
[1] "There is no available data for month 12"
[1] "13 is an invalid month"
```

```
file_1 <- "Month-"
file_2 <- ".csv"
month <- 1:13

for(i in month) {

  # create file name
  if(i %in% 1:9) {
    file_name <- paste0("data/", file_1, 0, i, file_2)
  } else if(i %in% 10:12) {
    file_name <- paste0("data/", file_1, i, file_2)
  } else {
    response <- paste(i, "is an invalid month")
    print(response)
    next
  }

  # import data
  if(file.exists(file_name)) {
    df <- read_csv(file_name)
    assign(paste0("df.month.", i), df)
    rm(df)
  } else {
    response <- paste("There is no available data for month", i)
    print(response)
  }
}
```

SOLUTION

When you run this for `months 1:13`



Since all our data frames have similar variables (we just get updated data each month), how could we change this so you just create one single data frame?

and for months 12 & 13 the response is:

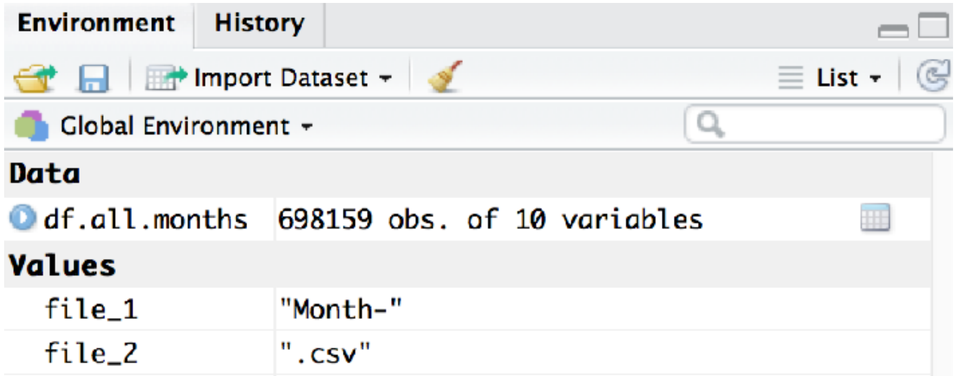
```
[1] "There is no available data for month 12"
[1] "13 is an invalid month"
```

```
file_1 <- "Month-"
file_2 <- ".csv"
month <- 1:13

# create empty data frame
df.all.months <- data.frame(NULL)

for(i in month) {

  # create file name
  if(i %in% 1:9) {
    file_name <- paste0("data/", file_1, 0, i, file_2)
  } else if(i %in% 10:12) {
    file_name <- paste0("data/", file_1, i, file_2)
  } else {
    response <- paste(i, "is an invalid month")
    print(response)
    next
  }

  # import data
  if(file.exists(file_name)) {
    df <- read_csv(file_name)
    df.all.months <- rbind(df.all.months, df)
    rm(df)
  } else {
    response <- paste("There is no available data for month", i)
    print(response)
  }
}
```

SOLUTION

1. Create an empty data frame.

2. Then as you import the files, you can rbind the new data to our empty data frame.



ITERATION

`map` functions

INTRODUCING `purrr`

The **`purrr`** *package provides an alternative means to iterate over elements and perform a set function*

Each map function works the same **`map_*(x, f)`***:*

1. ***loop over*** `x`
2. ***apply function*** `f` ***to each element***
3. ***return the results***

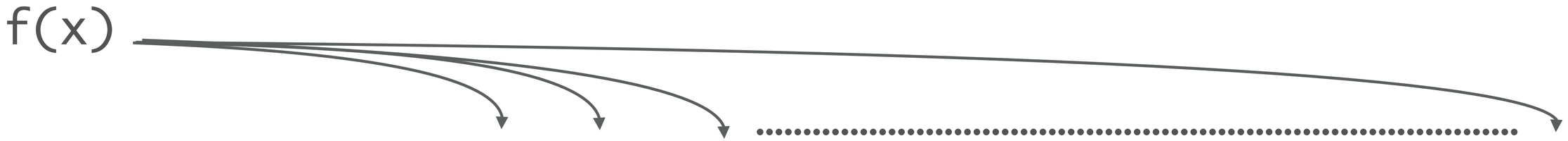
This is an alternative approach to the apply family of functions

BASICS OF THE MAP FUNCTIONS

```
map_*(vector, function)
```

BASICS OF THE MAP FUNCTIONS

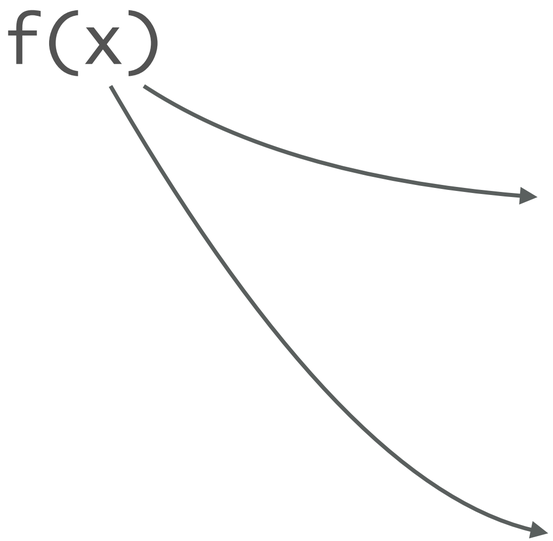
```
map_*(data.frame, function)
```



	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1

BASICS OF THE MAP FUNCTIONS

```
map_*(list, function)
```




```
$item1
          [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 0.1843049 0.05852987 0.2002625 0.2860744 0.7361251
[2,] 0.3005814 0.77366238 0.3798870 0.5418013 0.1937544

$item2
          [,1]      [,2]      [,3]      [,4]       [,5]
[1,] 0.9181339 0.6694026 0.3084275 0.4940026 0.08608652
[2,] 0.5877923 0.5218446 0.9769865 0.6842613 0.04770349
```

BASICS OF THE MAP FUNCTIONS

The `map_*` *function we use is determined by the output of the function*

- `map()` *returns a list*
- `map_dbl()` *returns a double vector*
- `map_lgl()` *returns a logical vector*
- `map_int()` *returns a integer vector*
- `map_chr()` *returns a character vector*

BASICS OF THE MAP FUNCTIONS

```
map_dbl(mtcars, mean)
mtcars %>% map_dbl(mean)
       mpg        cyl       disp         hp
 20.090625   6.187500 230.721875 146.687500
      drat         wt       qsec         vs
  3.596563   3.217250  17.848750   0.437500
        am       gear       carb
  0.406250   3.687500   2.812500

mtcars %>% map(mean)
$mpg
[1] 20.09062

$cyl
```

The **map_*** *functions incorporate the* %>% *operator*

YOUR TURN!

With the iris data set, use the map functions to answer these three questions:

1. *what is the class of each variable?*
2. *what is the mean value for each variable?*
3. *which variables have a mean value greater than 5*

SOLUTION

```
# what is the class of each variable?

iris %>% map_chr(class)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
   "numeric"    "numeric"    "numeric"    "numeric"     "factor"


# what is the mean value for each variable?
iris %>% map_dbl(mean)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
    5.843333     3.057333     3.758000     1.199333           NA
```

SOLUTION

```
# which variables have a mean value greater than 5?

# option 1
iris %>%
  map_dbl(mean) %>%
  map_lgl(~ . > 5)


# option 2
iris %>% map_lgl(~ mean(.) > 5)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
        TRUE        FALSE        FALSE        FALSE           NA
```

ADVANTAGES OF THE MAP FUNCTIONS

- *Handy shortcuts for specifying* `.f`
- *More consistent than* `sapply(), lapply()`*, which makes them better for programming*
- *Takes much less time to solve iteration problems (because they are written in C)*

SHORTCUTS FOR SPECIFYING `.f`

```
map(df, summary)
```

An existing function

```
map(df, myfunction)
```

An existing function you created

```
map(df, function(x) sum(is.na(x))
```

An anonymous function defined on the fly

```
map(df, ~ sum(is.na(.)))
```

An anonymous function defined using a formula shortcut

SHORTCUTS FOR SPECIFYING `.f`

Can you find what variables in the `nycflights13::flights` ***data have missing values and how many missing values they have?***

SHORTCUTS FOR SPECIFYING `.f`

Can you find what variables in the `nycflights::flights` ***data have missing values and how many missing values they have?***

```
nycflights13::flights %>% map_dbl(~ sum(is.na(.)))
          year          month            day       dep_time sched_dep_time
             0              0              0           8255              0
     dep_delay       arr_time sched_arr_time      arr_delay        carrier
          8255           8713              0           9430              0
        flight        tailnum         origin           dest       air_time
             0           2512              0              0           9430
      distance           hour         minute      time_hour
             0              0              0              0
```

MAP FUNCTIONS FOR QUICK MODEL COMPARISONS

Say we want to apply a model over a list of data frames

```
cyl <- split(mtcars, mtcars$cyl)
str(cyl)
List of 3
 $ 4:'data.frame': 11 obs. of  11 variables:
  ..$ mpg : num [1:11] 22.8 24.4 22.8 32.4 30.4 33.9 21.5 27.3 26 30.4 ...
  ..$ cyl : num [1:11] 4 4 4 4 4 4 4 4 4 4 ...
  ..$ disp: num [1:11] 108 146.7 140.8 78.7 75.7 ...
  ..$ hp  : num [1:11] 93 62 95 66 52 65 97 66 91 113 ...
  ..$ drat: num [1:11] 3.85 3.69 3.92 4.08 4.93 4.22 3.7 4.08 4.43 3.77 ...
  ..$ wt  : num [1:11] 2.32 3.19 3.15 2.2 1.61 ...
  ..$ qsec: num [1:11] 18.6 20 22.9 19.5 18.5 ...
  ..$ vs  : num [1:11] 1 1 1 1 1 1 1 1 0 1 ...
```

MAP FUNCTIONS FOR QUICK MODEL COMPARISONS

Say we want to apply a model over a list of data frames

```
cyl[[1]]
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
```

MAP FUNCTIONS FOR QUICK MODEL COMPARISONS

Say we want to apply a model over a list of data frames

```
mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .))
$`4`

Call:
lm(formula = mpg ~ wt, data = .)

Coefficients:
(Intercept)           wt
     39.571       -5.647

$`6`
```

1. **Apply regression model over each data frame**

MAP FUNCTIONS FOR QUICK MODEL COMPARISONS

Say we want to apply a model over a list of data frames

```
mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .)) %>%
  map(summary)
$`4`

Call:
lm(formula = mpg ~ wt, data = .)

Residuals:
    Min      1Q  Median      3Q     Max
-4.1513 -1.9795 -0.6272  1.9299  5.2523
```

1. Apply regression model over each data frame
2. **Access results with the summary function**

MAP FUNCTIONS FOR QUICK MODEL COMPARISONS

Say we want to apply a model over a list of data frames

```
mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .)) %>%
  map(summary) %>%
  map_dbl("r.squared")
        4         6         8
0.5086326 0.4645102 0.4229655
```

1. Apply regression model over each data frame
2. Access results with the summary function
3. **Get specific results by subsetting each list item**

MAP FUNCTIONS FOR QUICK MODEL COMPARISONS

Say we want to apply a model over a list of data frames

```
mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .)) %>%
  map(summary) %>%
  map("coefficients") %>%
  map_dbl(2)
       4         6         8
-5.647025 -2.780106 -2.192438
```

1. Apply regression model over each data frame
2. Access results with the summary function
3. **Get specific results by subsetting each list item**

YOUR TURN!

1. *split the* `ggplot2::diamonds` *data set by cut*

2. *run a regression on each list item* `lm(price ~ carat, data = .)`

3. *get the* `summary` *of the regression*

4. *how does the* `"r.squared"` *compare across models?*

5. *how do the model slopes (aka* `“coefficients”`*) compare?*

SOLUTION

Compare R^2 values

```
ggplot2::diamonds %>%
  split(.$cut) %>%
  map(~ lm(price ~ carat, data = .)) %>%
  map(summary) %>%
  map_dbl("r.squared"))
     Fair      Good Very Good   Premium     Ideal
0.7383940 0.8509539 0.8581622 0.8556336 0.8670887
```

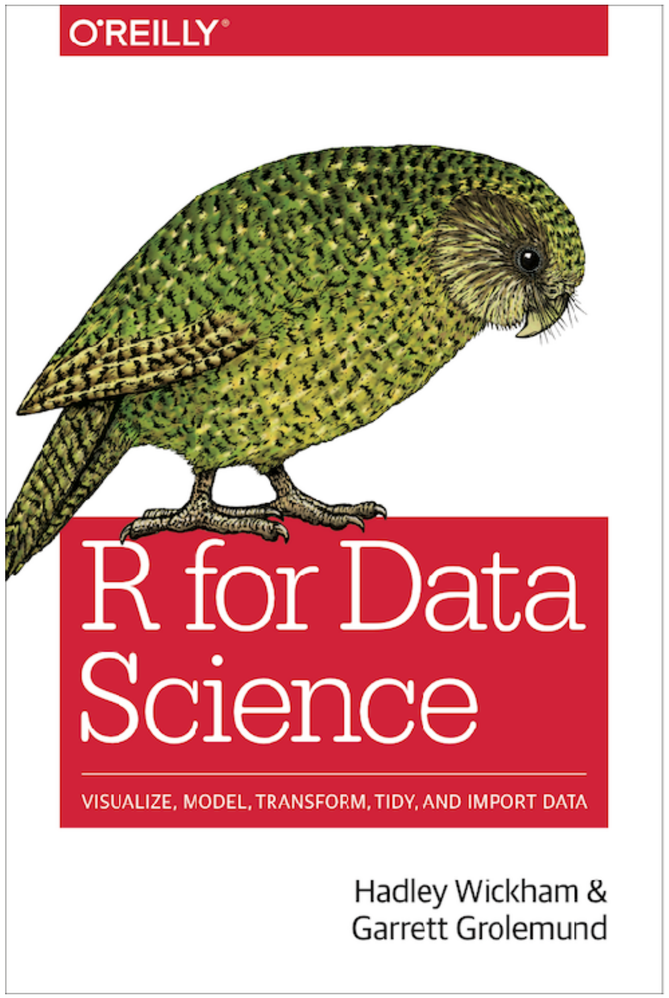
SOLUTION

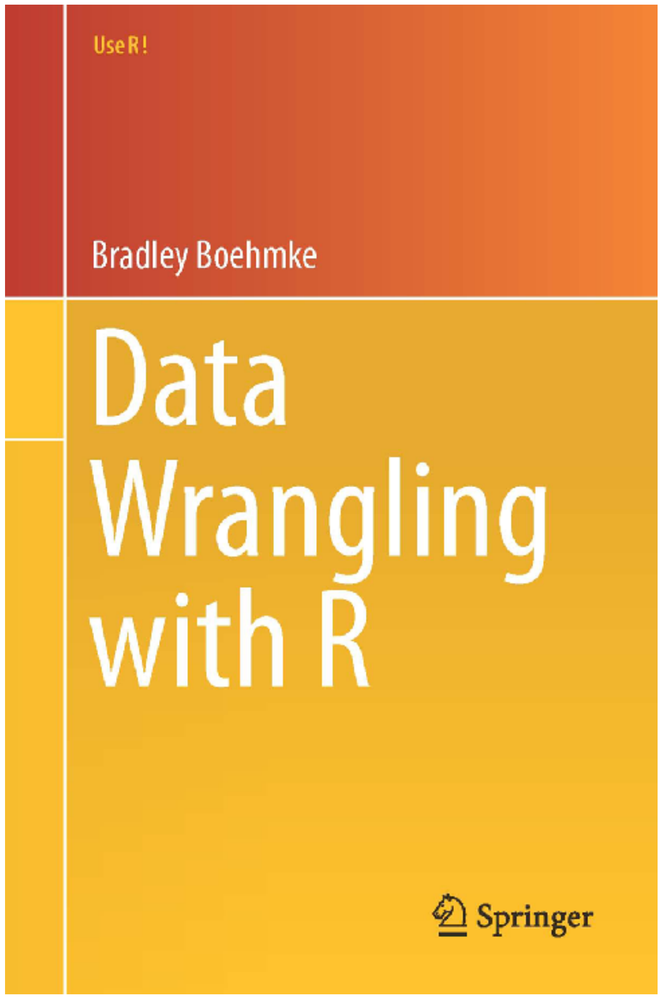
Compare slopes values

```
ggplot2::diamonds %>%
  split(.$cut) %>%
  map(~ lm(price ~ carat, data = .)) %>%
  map(summary) %>%
  map("coefficients") %>%
  map_dbl(2)
     Fair      Good Very Good   Premium     Ideal
 5924.495  7479.636  7935.972  7807.752  8192.391
```


SO LITTLE TIME!

LEARN MORE


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


UseR!
Bradley Boehmke
Data Wrangling with R
Springer

WHAT TO REMEMBER

FUNCTIONS TO REMEMBER

Operator/Function	Description
`if, if…else, ifelse`	Conditional control statements
`for, while, repeat`	Looping control statements
`seq_along`	Sequencing argument
`break, next`	Arguments to exit or skip a loop iteration
`map, map_dbl, map_int, map_chr, map_lgl`	Family of iteration functions
`split`	Split a data frame based on categorical variable levels