Codes for STEM

Noushin Nabavi

2020-09-10

Contents

Chapter 1

Coding for STEM

Tools and capabilities of data science is changing everyday!

This is how I understand it today:

Data can: * Describe the current state of an organization or process
* Detec anomalous events
* Diagnose the causes of events and behaviors
* Predict future events

Data Science workflows can be developed for:
* Data collection and management
* Exploration and visualization
* Experimentation and prediction

Applications of data science can include:
* Traditional machine learning: e.g. finding probabilities of events, labeled data, and algorithms
* Deep learning: neurons work together for image and natural language recognition but requires more training data
* Internet of things (IOT): e.g. smart watch algorithms to detect and analyze motion sensors

Data science teams can consist of: * Data engineers: SQL, Java, Scala, Python
* Data analysts: Dashboards, hypothesis tests and visualization using spreadsheets, SQL, BI (Tableau, power BI, looker)
* Machine learning scientists: predictions and extrapolations, classification, etc. and use R or python * Data employees can be isolated, embedded, or hybrid

Data use can come with risks of identification of personal information. Policies for personally identifiable information may need to consider:

* sensitivity and caution
* pseudonymization and anonymization

Preferences can be stated or revealed through the data so questions need to be specific, avoid loaded language, calibrate, require actionable results.

Data storage and retrieval may include: * parallel storage solutions (e.g. cluster or server)
* cloud storage (google, amazon, azure)
* types of data: 1) unstructured (email, text, video, audio, web, and social media = document database); 2) structured = relational databases
* Data querying: NoSQL and SQL

Communication of data can include:
* Dashboards
* Markdowns
* BI tools
* rshiny or d3.js

Team management around data can use: * Trello, slack, rocket chat, or JIRA to communicate due data and priority

A/B Testing: * Control and Variation in samples
* 4 steps in A/B testing: pick metric to track, calculate sample size, run the experiment, and check significance

Machine learning (ML) can be used for time series forecasting (investigate seasonality on any time scale), natural language processing (word count, word embeddings to create features that group similar words), neural networks, deep learning, and AI.
Learning can be classified into: *Supervised*: labels and features/ Model evaluation on test and train data with applications in: * recommendation systems
* subscription predictions
* email subject optimization
Unsupervised: unlabeled data with only features
* clustering

Deep learning and AI requirements: * prediction is more feasible than explanations
* lots of very large amount of training data

Chapter 2

Introduction

Chapter 3

R for Reporting

Possible ways to report your findings include e-mailing figures and tables around with some explanatory text or creating reports in Word, LaTeX or HTML.

R code used to produce the figures and tables is typically not part of these documents. So in case the data changes, e.g., if new data becomes available, the code needs to be re-run and all the figures and tables updated. This can be rather cumbersome. If code and reporting are not in the same place, it can also be a bit of a hassle to reconstruct the details of the analysis carried out to produce the results.

To enable reproducible data analysis and research, the idea of dynamic reporting is that data, code and results are all in one place. This can for example be a R Markdown document like this one. Generating the report automatically executes the analysis code and includes the results in the report.

3.1 Usage demonstrations

3.1.1 Inline code

Simple pieces of code can be included inline. This can be handy to, e.g., include the number of observations in your data set dynamically. The *cars* data set, often used to illustrate the linear model, has 50 observations.

3.1.2 Code chunks

You can include typical output like a summary of your data set and a summary of a linear model through code chunks.

```
summary(cars)
```

```
##      speed          dist
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

```
m <- lm(dist ~ speed, data = cars)
summary(m)
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123 *
## speed         3.9324     0.4155   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511,Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

3.1.2.1 Include tables

The estimated coefficients, as well as their standard errors, t-values and p-values can also be included in the form of a table, for example through **knitr**'s `kable` function.

```
library("knitr")
kable(summary(m)$coef, digits = 2)
```

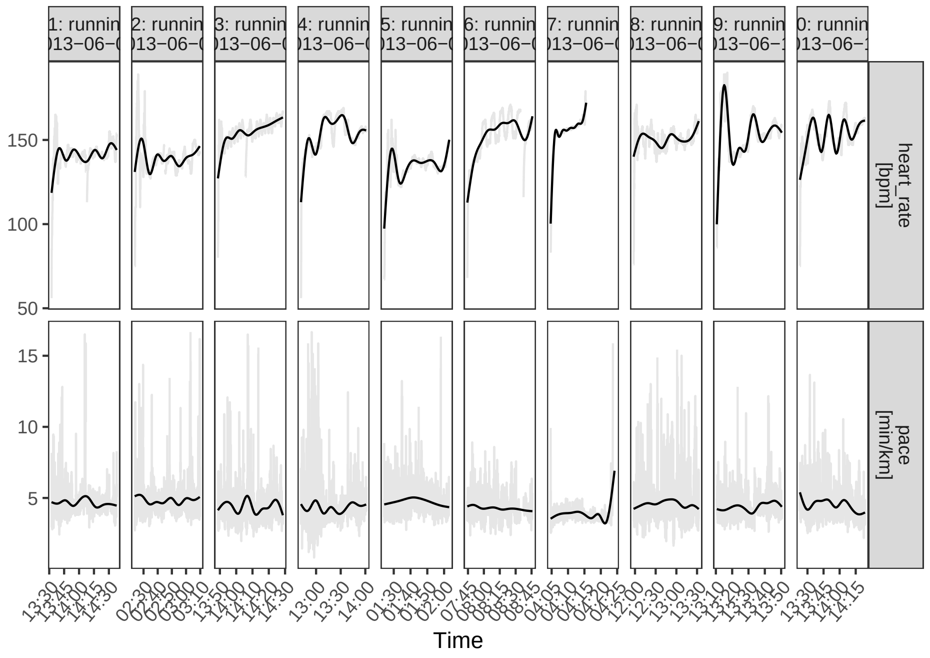
	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-17.58	6.76	-2.60	0.01
speed	3.93	0.42	9.46	0.00

3.1.2.2 Include figures

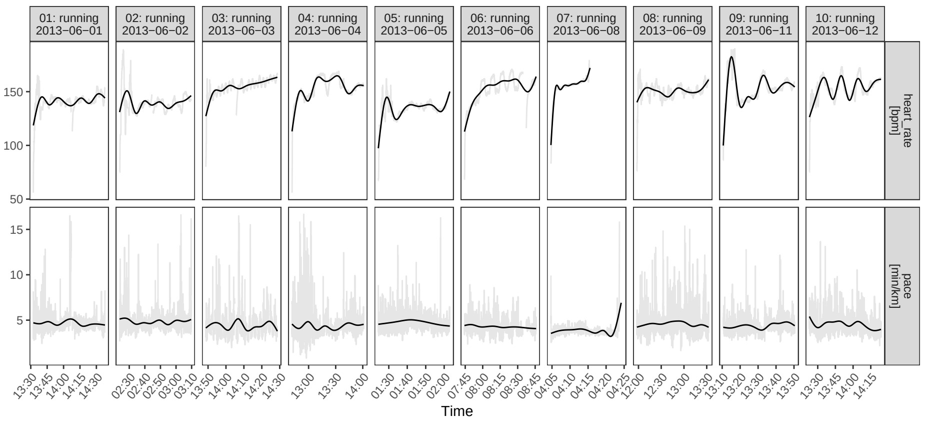
The **trackeR** package provides infrastructure for running and cycling data in **R** and is used here to illustrate how figures can be included.

```r
## install.packages("devtools")
## devtools::install_github("hfrick/trackeR")
library("trackeR")
data("runs", package = "trackeR")
```

A plot of how heart rate and pace evolve over time in 10 training sessions looks like this



but the plot looks better with a wider plotting window.



3.2 Resources

- Markdown main page
- R Markdown
- knitr in a nutshell tutorial by Karl Broman

Chapter 4

Useful R Functions + Examples

This is *NOT* intended to be fully comprehensive list of every useful R function that exists, but is a practical demonstration of selected relevant examples presented in user-friendly format, all available in base R. For a wider collection to work through, this Reference Card is recommended: https://cran.r-project.org/doc/contrib/Baggott-refcard-v2.pdf

Additional CRAN reference cards and R guides (including non-English documentation) found here: https://cran.r-project.org/other-docs.html

4.1 Contents

* 12. `data.frame()`, `class()`, `names()`, `str()`, `summary()`, `View()`, `head()`, `tail()`, `as.data.frame()`

C. Core
* 13. `df[order(),]`
* 14. `df[,c()]`, `df[which(),]`
* 15. `table()`
* 16. `mean()`, `median()`, `sd()`, `var()`, `sum()`, `min()`, `max()`, `range()`
* 17. `apply()`
* 18. `lapply()` using `list()`
* 19. `tapply()`

D. Common
* 20. `if` statement, `if...else` statement
* 21. `for` loop
* 22. `function()...`

4.2 R Syntax

REMEMBER: KEY R LANGUAGE SYNTAX

- **Case Sensitivity**: as per most UNIX-based packages, R is case sensitive, hence `X` and `x` are different symbols and would refer to different variables.

- **Expressions vs Assignments**: an expression, like `3 + 5` can be given as a command which will be evaluated and the value immediately printed, but not stored. An assignment however, like `sum <- 3 + 5` using the assignment operator `<-` also evaluates the expression `3 + 5`, but instead of printing and not storing, it stores the value in the object `sum` but doesn't print the result. The object `sum` would need to be called to print the result.

- **Reserved Words**: choice for naming objects is almost entirely free, except for these reserved words: https://stat.ethz.ch/R-manual/R-devel/library/base/html/Reserved.html

- **Spacing**: outside of the function structure, spaces don't matter, e.g. `3+5` is the same as `3+        5` is the same as `3 + 5`. For more best-practices for R code Hadley Wickham's Style Guide is a useful reference: http://adv-r.had.co.nz/Style.html
- **Comments**: add comments within your code using a hastag, `#`. R will ignore everything to the right of the hashtag within that line

4.3 Functional examples

1. Working Directory management

- `getwd()`, `setwd()` R/RStudio is always pointed at a specific directory on your computer, so it's important to be able to check what's the current directory using `getwd()`, and to be able to change and specify a different directory to work in using `setwd()`.

#check the directory R is currently pointed at getwd()

2. Bring up help documentation & examples

- `?foo`, `help(foo)`, `example(foo)`

```
?boxplot
help(boxplot)
example(boxplot)
```

---

3. Load & Call CRAN Packages

- `install.packages("foo")`, `library("foo")` Packages are add-on functionality built for R but not pre-installed (base R), hence you need to install/load the packages you want yourself. The majority of packages you'd want have been submitted to and are available via CRAN. At time of writing, the CRAN package repository featured 8,592 available packages.

4. Load & Call Packages from GitHub

- `devtools::install_github("username/packagename")` Not all packages you'll want will be available via CRAN, and you'll likely need to get certain packages from GitHub accounts. This example shows how to install the `shinyapps` package from RStudio's GitHub account.
- install.packages("devtools") #pre-requisite for `devtools...` function
- devtools::install_github("rstudio/shinyapps") #install specific package from specific GitHub account
- library("shinyapps") #Call package

5. Load datasets from base R & Loaded Packages

- `data("foo")`

```
#AIM: show available datasets
data()

#AIM: load an available dataset
data("iris")
```

---

6. I/O Loading Existing Local Data

- `read.csv`, `read.table`

(a) I/O When already in the working directory where the data is

Import a local **csv** file (i.e. where data is separated by **commas**), saving it as an object: - object <- read.csv("xxx.csv")

Import a local tab delimited file (i.e. where data is separated by **tabs**), saving it as an object: - object <- read.csv("xxx.csv", header = FALSE) —

(b) I/O When NOT in the working directory where the data is

For example to import and save a local **csv** file from a different working directory you either need to specify the file path (operating system specific), e.g.:

on a mac: - object <- read.csv("~/Desktop/R/data.csv")

on windows: = object <- read.csv("C:/Desktop/R/data.csv")

OR

You can use the file.choose() command which will interactively open up the file dialog box for you to browse and select the local file, e.g.: - object <- read.csv(file.choose())

(c) I/O Copying & Pasting Data

For relatively small amounts of data you can do an equivalent copy paste (operating system specific):

on a mac: - object <- read.table(pipe("pbpaste"))

on windows: - object <- read.table(file = "clipboard")

(d) I/O Loading Non-Numerical Data - character strings

Be careful when loading text data! R may assume character strings are statistical factor variables, e.g. "low", "medium", "high", when are just individual labels like names. To specify text data NOT to be converted into factor variables, add `stringsAsFactor = FALSE` to your `read.csv/read.table` command: - object <- read.table("xxx.txt", stringsAsFactors = FALSE)

(e) I/O Downloading Remote Data

For accessing files from the web you can use the same `read.csv/read.table` commands. However, the file being downloaded does need to be in an R-friendly format (maximum of 1 header row, subsequent rows are the equivalent of one data record per row, no extraneous footnotes etc.). Here is an example downloading an online csv file of coffee harvest data used in a Nature study: - object <- read.csv("http://sumsar.net/files/posts/2014-02-04-bayesian-first-aid-one-sample-t-test/roubik_2002_coffe_yield.csv")

7. I/O Exporting Data Frame

- `write.table()`

Navigate to the working directory you want to save the data table into, then run the command (in this case creating a tab delimited file): - write.table(object, “xxx.txt”, sep = ”̃”)

8. I/O Saving Down & Loading Objects

- `save()`, `load()`

These two commands allow you to save a named R object to a file and restore that object again.
Navigate to the working directory you want to save the object in then run the command: - save(object, file = “xxx.rda”)

reload the object: - load(“xxx.rda”)

9. Vector & Matrix Construction

- `c()`, `cbind()`, `rbind()`, `matrix()` Vectors (lists) & Matrices (two-dimensional arrays) are very common R data structures.

```
#use c() to construct a vector by concatenating data
foo <- c(1, 2, 3, 4) #example of a numeric vector
oof <- c("A", "B", "C", "D") #example of a character vector
ofo <- c(TRUE, FALSE, TRUE, TRUE) #example of a logical vector

#use cbind() & rbind() to construct matrices
coof <- cbind(foo, oof) #bind vectors in column concatenation to make a matrix
roof <- rbind(foo, oof) #bind vectors in row concatenation to make a matrix

#use matrix() to construct matrices
moof <- matrix(data = 1:12, nrow=3, ncol=4) #creates matrix by specifying set of values, no. of
```

10. Vector & Matrix Explore

- `length()`, `dim()`

```
length(foo) #length of vector

dim(coof) #returns dimensions (no. of rows & columns) of vector/matrix/dataframe
```

11. Vector & Matrix Sort & Select

- `sort()`, `'vector'[]`, `'matrix'[]`

```
#create another numeric vector
jumble <- c(4, 1, 2, 3)
sort(jumble) #sorts a numeric vector in ascending order (default)
sort(jumble, decreasing = TRUE) #specify the decreasing arg to reverse default order

#create another character vector
mumble <- c( "D", "B", "C", "A")
```

```
sort(mumble) #sorts a character vector in alphabetical order (default)
sort(mumble, decreasing = TRUE) #specify the decreasing arg to reverse default order

jumble[1] #selects first value in our jumble vector
tail(jumble, n=1) #selects last value
jumble[c(1,3)] #selects the 1st & 3rd values
jumble[-c(1,3)] #selects everything except the 1st & 3rd values

coof[1,] #selects the 1st row of our coof matrix
coof[,1] #selects the 1st column
coof[2,1] #selects the value in the 2nd row, 1st column
coof[,"oof"] #selects the column named "oof"
coof[1:3,] #selects columns 1 to 3 inclusive
coof[c(1,2,3),] #selects the 1st, 2nd & 3rd rows (same as previous)
```

12. Create & Explore Data Frames

    - `data.frame()`, `class()`, `names()`, `str()`, `summary()`, `View()`, `head()`, `tail()`, `as.data.frame()` A data frame is a matrix-like data structure made up of lists of variables with the same number of rows, which can be of differing data types (numeric, character, factor etc.) - matrices must have columns all of the same data type.

```
#create a data frame with 3 columns with 4 rows each
doof <- data.frame("V1"=1:4, "V2"=c("A","B","C","D"), "V3"=5:8)

class(doof) #check data frame object class
names(doof) # returns column names
str(doof) #see structure of data frame
summary(doof) #returns basic summary stats
View(doof) #invokes spreadsheet-style viewer
head(doof, n=2) #shows first parts of object, here requesting the first 2 rows
tail(doof, n=2) #shows last parts of object, here requesting the last 2 rows

convert <- as.data.frame(coof) #convert a non-data frame object into a data frame
```

13. Data Frame Sort

    - `df[order(),]`

```
#use 'painters' data frame
library("MASS") #call package with the required data
data("painters") #load required data
View(painters) #scan dataset

#syntax for using a specific variable: df=data frame, '$', V1=variable name
df$V1
```

```
#AIM: print the 'School' variable column
painters$School

#syntax for df[order(),]
df[order(df$V1, df$V2...),] #function arguments: df=data frame, in square brackets specify withir

#AIM: order the dataset rows based on the painters' Composition Score column, in Ascending order
painters[order(painters$Composition),] #Composition is the sorting variable

#AIM: order the dataset rows based on the painters' Composition Score column, in Descending order
painters[order(-painters$Composition),] #append a minus sign in front of the variable you want tc

#AIM: order the dataset rows based on the painters' Composition Score column, in Descending order
painters[order(-painters$Composition), c(1:3)]
```

14. Data Frame Select & Deselect

- `df[,c()]`, `df[which(),]`

```
#use 'painters' data frame

#syntax for select & deselect based on column variables
df[, c("V1", "V2"...)] #function arguments: df=data frame, in square brackets specify columns to

#AIM: select the Composition & Drawing variables based on their column name
painters[, c("Composition", "Drawing")] #subset the df, selecting just the named columns (and all

#AIM: select the Composition & Drawing variables based on their column positions in the painters
painters[, c(1,2)] #subset the df, selecting just the 1st & 2nd columns (and all the rows)

#AIM: drop the Expression variable based on it's column position in the painters data frame and r
painters[c(1:5), -4] #returns the subsetted df having deselected the 4th column, Expression and r


#syntax for select & deselect based on row variable values
df[which(),] #df=data frame, specify the variable value within the `which()` to subset the df on

#AIM: select all rows where the painters' School is the 'A' category
painters[which(painters$School == "A"),] #returns the subsetted df where equality holds true, i.e

#AIM: deselect all rows where the painters' School is the 'A' category, i.e. return df subset wit
painters[which(painters$School != "A" & painters$Colour > 10),] #returns the subsetted df where e
```

15. Data Frame Frequency Calculations

- `table()`

```
#create new data frame
flavour <- c("choc", "strawberry", "vanilla", "choc", "strawberry", "strawberry")
gender <- c("F", "F", "M", "M", "F", "M")
icecream <- data.frame(flavour, gender) #icecream df made up of 2 factor variables, fl

#AIM: create a frequency distribution table which shows the count of each gender in th
table(icecream$gender)

#AIM: create a frequency distribution table which shows the count of each flavour in th
table(icecream$flavour)

#AIM: create Contingency/2-Way Table showing the counts for each combination of flavou
table(icecream$flavour, icecream$gender)
```

16. Descriptive/Summary Stats Functions

- `mean()`, `median()`, `sd()`, `var()`, `sum()`, `min()`, `max()`, `range()`

```
#re-using the jumble vector from before
jumble <- c(4, 1, 2, 3)

mean(jumble)
median(jumble)
sd(jumble)
var(jumble)
sum(jumble)
min(jumble)
max(jumble)
range(jumble)
```

17. Apply Functions

- `apply()` `apply()` returns a vector, array or list of values where a specified function has been applied to the 'margins' (rows/cols combo) of the original vector/array/list.

```
#re-using the moof matrix from before
moof <- matrix(data = 1:12, nrow=3, ncol=4)

#apply syntax
apply(X, MARGIN, FUN,...) #function arguments: X=an array, MARGIN=1 to apply to rows/2

#AIM: using the moof matrix, apply the sum function to the rows
apply(moof, 1, sum)

#AIM: using the moof matrix, apply the sum function to the columns
apply(moof, 2, sum)
```

18. Apply Functions

- `lapply()` using `list()` A list, a common data structure, is a generic vector containing objects of any types. `lapply()` returns a list where each element returned is the result of applying a specified function to the objects in the list.

```
#create list of various vectors and matrices
bundle <- list(moof, jumble, foo)

#lapply syntax
lapply(X, FUN,...) #function arguments: X=a list, FUN=function to apply

#AIM: using the bundle list, apply the mean function to each object in the list
lapply(bundle, mean)
```

19. Apply Functions

- `tapply()` `tapply()` applies a specified function to specified groups/subsets of a factor variable.

```
#tapply syntax
tapply(X, INDEX, FUN,...) #function arguments: X=an atomic object, INDEX=list of 1+ factors of X

#AIM: calculate the mean Drawing Score of the painters, but grouped by School category
tapply(painters$Drawing, painters$School, mean) #grouping the data by the 8 different Schools, ap
```

20. Programming Tools

- `if` statement, `if...else` statement An `if` statement is used when certain computations are conditional and only execute when a specific condition is met - if the condition is not met, nothing executes. The `if...else` statement extends the `if` statement by adding on a computation to execute when the condition is not met, i.e. the 'else' part of the statement.

```
#if-statement syntax
if ('test expression')
    {
    'statement'
    }

#if...else statement
if ('test expression')
    {
    'statement'
    }else{
    'another statement'
    }
```

```
#AIM: here we want to test if the object, 'condition_to_test' is smaller than 10. If i

#specify the 'test expression'
condition_to_test <- 7

#write your 'if...else' function based on a 'statement' or 'another statement' depende
if (condition_to_test > 5)
    {
    result_after_test = 'Above Average'
    }else{
    result_after_test = 'Below Average'
    }

#call the resulting 'statement' as per the instruction of the 'if...else' statement
result_after_test
```

21. Programming Tools

- `for` loop A `for` loop is an automation method for repeating (looping) a specific set of instructions for each element in a vector.

```
#for loop syntax requires a counter, often called 'i' to denote an index
for ('counter' in 'looping vector')
    {
    'instructions'
    }

#AIM: here we want to print the phrase "In the Year yyyy" 6x, once for each year betwe
#this for loop executes the code chunk 'print(past("In the Year", i)) for each of the
for (i in 2010:2015)
    {
    print(paste("In the Year", i))
    }

#AIM: create an object which contains 10 items, namely each number between 1 and 10 sq
#to store rather than just print results, an empty storage container needs to be creat
container <- NULL
for (i in 1:10)
    {
    container[i] = i^2
    }

container #check results: the loop is instructed to square every element of the loopin
```

22. Programming Tools

- `function()...` User-programmed functions allow you to specify customised arguments and returned values.

```
#AIM: to create a simplified take-home pay calculator (single-band), called 'takehome_pay'. Our f
takehome_pay <- function(tax_rate, income)
    {
    tax = tax_rate * income
    return(income - tax)
    }

takehome_pay(tax_rate = 0.2, income = 25000) #call our function to calculate 'takehome_pay' on a
```

23. Strings

- `grep()`, `tolower()`, `nchar()`

24. Further Data Selection

- `quantile()`, `cut()`, `which()`, `na.omit()`, `complete.cases()`, `sample()`

25. Further Data Creation

- `seq()`, `rep()`

26. Other Apply-related functions

- `split()`, `sapply()`, `aggregate()`

27. More Loops

- `while` loop, `repeat` loop

…..Ad Infinitum!!

Chapter 5

Exploratory data analysis using SQL

What is in a database?
- This set of exercises explores writing functions and stored procedures in SQL servers.

Eexplore data tables: - Select the count of the number of rows

```
SELECT count(*)
  FROM tablename;
```

Counting missing data: - Select the count of ticker, - subtract from the total number of rows, - and alias as missing

```
SELECT count(*) - count(ticker) AS missing
  FROM fortune500;
```

- Select the count of profits_change,
- subtract from total number of rows, and alias as missing

```
SELECT count(*) - count(profits_change) AS missing
  FROM fortune500;
```

- Select the count of industry,
- subtract from total number of rows, and alias as missing

```
SELECT count(*) - count(industry) AS missing
  FROM fortune500;
```

Joining tables:

```
SELECT company.name
-- Table(s) to select from
  FROM company
```

```
       INNER JOIN fortune500
       ON company.ticker=fortune500.ticker;
```

The keys to the database (e.g. foreign vs. primary keys) - Read an entity relationship diagram

```
-- Count the number of tags with each type
SELECT type, count(*) AS count
  FROM tag_type
 -- To get the count for each type, what do you need to do?
 GROUP BY type
 -- Order the results with the most common
 -- tag types listed first
 ORDER BY count DESC;
```

- or:

```
-- Select the 3 columns desired
SELECT name, tag_type.tag, tag_type.type
  FROM company
      -- Join the tag_company and company tables
       INNER JOIN tag_company
       ON company.id = tag_company.company_id
       -- Join the tag_type and company tables
       INNER JOIN tag_type
       ON tag_company.tag = tag_type.tag
  -- Filter to most common type
  WHERE type='cloud';
```

- coalesce function (to combine columns)

```
-- Use coalesce
SELECT coalesce(industry, sector, 'Unknown') AS industry2,
       -- Don't forget to count!
       count(*)
  FROM fortune500
-- Group by what? (What are you counting by?)
 GROUP BY industry2
-- Order results to see most common first
 ORDER BY count DESC
-- Limit results to get just the one value you want
 LIMIT 1;
```

- Coalesce with a self-join:

```
SELECT company_original.name, title, rank
  -- Start with original company information
  FROM company AS company_original
       -- Join to another copy of company with parent
       -- company information
```

```
   LEFT JOIN company AS company_parent
       ON company_original.parent_id = company_parent.id
       -- Join to fortune500, only keep rows that match
       INNER JOIN fortune500
       -- Use parent ticker if there is one,
       -- otherwise original ticker
       ON coalesce(company_parent.ticker,
                   company_original.ticker) =
             fortune500.ticker
 -- For clarity, order by rank
 ORDER BY rank;
```

Column types and constraints - Effects of casting - SELECT CAST(value AS new_type);

```
-- Select the original value
SELECT profits_change,
   -- Cast profits_change
       CAST(profits_change AS integer) AS profits_change_int
  FROM fortune500;
```

- SELECT and divide

```
-- Divide 10 by 3
SELECT 10/3,
       -- Divide 10 cast as numeric by 3
       10::numeric/3;
```

- SELECT value::new_type

```
SELECT '3.2'::numeric,
       '-123'::numeric,
       '1e3'::numeric,
       '1e-3'::numeric,
       '02314'::numeric,
       '0002'::numeric;
```

- Summarize the distribution of numeric values

```
-- Select the count of each value of revenues_change
SELECT revenues_change, count(*)
  FROM fortune500
 GROUP BY revenues_change
 -- order by the values of revenues_change
 ORDER BY revenues_change;
```

- Additional exploration syntax:

```
-- Count rows
SELECT count(*)
  FROM fortune500
```

```
 -- Where...
 WHERE revenues_change > 0;
```

Numeric data types and summary functions

Division

```
-- Select average revenue per employee by sector
SELECT sector,
       avg(revenues/employees::numeric) AS avg_rev_employee
  FROM fortune500
 GROUP BY sector
 -- Use the alias to order the results
 ORDER BY avg_rev_employee;
```

- explore by division:

```
-- Divide unanswered_count by question_count
SELECT unanswered_count/question_count::numeric AS computed_pct,
       -- What are you comparing the above quantity to?
       unanswered_pct
  FROM stackoverflow
 -- eliminate rows where question_count is not 0
 WHERE question_count != 0
 LIMIT 10;
```

Following SQL functions DATEDIFF( ), DATENAME( ), DATEPART( ), CAST( ), CONVERT( ), GETDATE( ) and DATEADD( ) explore transactions per day

```
SELECT
  -- Select the date portion of StartDate
  CONVERT(DATE, StartDate) as StartDate,
  -- Measure how many records exist for each StartDate
  COUNT(ID) as CountOfRows
FROM CapitalBikeShare
-- Group by the date portion of StartDate
GROUP BY CONVERT(DATE, StartDate)
-- Sort the results by the date portion of StartDate
ORDER BY CONVERT(DATE, StartDate);
```

seconds or no? - DATEDIFF() can be used to calculate the trip time by finding the difference between Start and End time - Here, we will use DATEPART() to see how many transactions have seconds greater than zero and how many have them equal to zero

```
SELECT
-- Count the number of IDs
```

```
COUNT(ID) AS Count,
    -- Use DATEPART() to evaluate the SECOND part of StartDate
    "StartDate" = CASE WHEN DATEPART(SECOND, StartDate) = 0 THEN 'SECONDS = 0'
   WHEN DATEPART(SECOND, StartDate) > 0 THEN 'SECONDS > 0' END
FROM CapitalBikeShare
GROUP BY
    -- Complete the CASE statement
CASE WHEN DATEPART(SECOND, StartDate) = 0 THEN 'SECONDS = 0'
 WHEN DATEPART(SECOND, StartDate) > 0 THEN 'SECONDS > 0' END
```

- Which day of week is busiest?

```
SELECT
    -- Select the day of week value for StartDate
DATENAME(weekday, StartDate) as DayOfWeek,
    -- Calculate TotalTripHours
SUM(DATEDIFF(second, StartDate, EndDate))/ 3600 as TotalTripHours
FROM CapitalBikeShare
-- Group by the day of week
GROUP BY DATENAME(weekday, StartDate)
-- Order TotalTripHours in descending order
ORDER BY TotalTripHours DESC
```

- finding the outliers:

```
SELECT
-- Calculate TotalRideHours using SUM() and DATEDIFF()
   SUM(DATEDIFF(SECOND, StartDate, EndDate))/ 3600 AS TotalRideHours,
    -- Select the DATE portion of StartDate
   CONVERT(DATE, StartDate) AS DateOnly,
    -- Select the WEEKDAY
   DATENAME(WEEKDAY, CONVERT(DATE, StartDate)) AS DayOfWeek
FROM CapitalBikeShare
-- Only include Saturday
WHERE DATENAME(WEEKDAY, StartDate) = 'Saturday'
GROUP BY CONVERT(DATE, StartDate);
```

Variables for datetime data: storing data in variables DECLARE & CAST - use CapitalBikeShare table as starting point

```
-- Create @ShiftStartTime
DECLARE @ShiftStartTime AS time = '08:00 AM'

-- Create @StartDate
DECLARE @StartDate AS date

-- Set StartDate to the first StartDate from CapitalBikeShare
```

```
SET
@StartDate = (
     SELECT TOP 1 StartDate
     FROM CapitalBikeShare
     ORDER BY StartDate ASC
)

-- Create ShiftStartDateTime
DECLARE @ShiftStartDateTime AS datetime

-- Cast StartDate and ShiftStartTime to datetime data types
SET @ShiftStartDateTime = CAST(@StartDate AS datetime) + CAST(@ShiftStartTime AS datet

SELECT @ShiftStartDateTime
```

- DECLARE a TABLE:

```
-- Create @Shifts
DECLARE @Shifts TABLE(
    -- Create StartDateTime column
StartDateTime datetime,
    -- Create EndDateTime column
EndDateTime datetime)
-- Populate @Shifts
INSERT INTO @Shifts (StartDateTime, EndDateTime)
SELECT '3/1/2018 8:00 AM', '3/1/2018 4:00 PM'
SELECT *
FROM @Shifts
```

- INSERT INTO ? based on CapitalBikeShare table:

```
-- Create @RideDates
DECLARE @RideDates TABLE(
    -- Create RideStart
RideStart date,
    -- Create RideEnd
RideEnd date)
-- Populate @RideDates
INSERT INTO @RideDates(RideStart, RideEnd)
-- Select the unique date values of StartDate and EndDate
SELECT DISTINCT
    -- Cast StartDate as date
CAST(StartDate as date),
    -- Cast EndDate as date
CAST(EndDate as date)
FROM CapitalBikeShare
SELECT *
FROM @RideDates;
```

Date manipulation - First day of month:

```
-- Find the first day of the current month
SELECT DATEADD(month, DATEDIFF(month, 0, GETDATE()), 0)
-- Or
SELECT DATEDIFF(month, 0, GETDATE()), 0)
-- Or
SELECT DATEDIFF(year, '12/31/2017', '1/1/2019')
-- Or for yesterday use -1
WHERE CAST(year as date) = DATEADD (d, -1, GETDATE())
```

- What was yesterday? Creating a function that returns yesterday's date

```
-- Create GetYesterday()
CREATE FUNCTION GetYesterday()
-- Specify return data type
RETURNS date
AS
BEGIN
-- Calculate yesterday's date value
RETURN(SELECT DATEADD(day, -1, GETDATE()))
END
```

- 1 input/output
- Create a function named SumRideHrsSingleDay() which returns the total ride time in hours for the `?` parameter passed.

```
-- Create SumRideHrsSingleDay
CREATE FUNCTION SumRideHrsSingleDay (@DateParm date)
-- Specify return data type
RETURNS numeric
AS
-- Begin
BEGIN
RETURN
-- Add the difference between StartDate and EndDate
(SELECT SUM(DATEDIFF(second, StartDate, EndDate))/3600
FROM CapitalBikeShare
 -- Only include transactions where StartDate = @DateParm
WHERE CAST(StartDate AS date) = @DateParm)
-- End
END
```

- Multiple inputs/outputs
- Create a function that accepts both StartDate and EndDate then returns the total ride hours for all transactions that occur within the parameter values.

```
-- Create the function
CREATE FUNCTION SumRideHrsDateRange (@StartDateParm datetime, @EndDateParm datetime)
-- Specify return data type
RETURNS numeric
AS
BEGIN
RETURN
-- Sum the difference between StartDate and EndDate
(SELECT SUM(DATEDIFF(second, StartDate, EndDate))/3600
FROM CapitalBikeShare
-- Include only the relevant transactions
WHERE StartDate > @StartDateParm and StartDate <@EndDateParm)
END
```

User defined functions: inline (faster) and multi-statement value functions (slower) - Inline value function:

```
-- Create the function
CREATE FUNCTION SumStationStats(@StartDate AS datetime)
-- Specify return data type
RETURNS TABLE
AS
RETURN
SELECT
StartStation,
    -- Use COUNT() to select RideCount
COUNT(ID) as RideCount,
    -- Use SUM() to calculate TotalDuration
    SUM(DURATION) as TotalDuration
FROM CapitalBikeShare
WHERE CAST(StartDate as Date) = @StartDate
-- Group by StartStation
GROUP BY StartStation;
```

- Multi statement value function

```
-- Create the function
CREATE FUNCTION CountTripAvgDuration (@Month CHAR(2), @Year CHAR(4))
-- Specify return variable
RETURNS @DailyTripStats TABLE(
TripDate date,
TripCount int,
AvgDuration numeric)
AS
BEGIN
-- Insert query results into @DailyTripStats
```

```
INSERT @DailyTripStats
SELECT
    -- Cast StartDate as a date
CAST(StartDate AS date),
    COUNT(ID),
    AVG(Duration)
FROM CapitalBikeShare
WHERE
DATEPART(month, StartDate) = @Month AND
    DATEPART(year, StartDate) = @Year
-- Group by StartDate as a date
GROUP BY CAST(StartDate AS date)
-- Return
RETURN
END
```

User defined functions in action: i.e. execute functions - can use SELECT to execute scalar functions

```
-- Create @BeginDate
DECLARE @BeginDate AS date = '3/1/2018'
-- Create @EndDate
DECLARE @EndDate AS date = '3/10/2018'
SELECT
  -- Select @BeginDate
  @BeginDate AS BeginDate,
  -- Select @EndDate
  @EndDate AS EndDate,
  -- Execute SumRideHrsDateRange()
  dbo.SumRideHrsDateRange(@BeginDate, @EndDate) AS TotalRideHrs
```

- anotehr example:

```
-- Create @RideHrs
DECLARE @RideHrs AS numeric
-- Execute SumRideHrsSingleDay()
EXEC @RideHrs = dbo.SumRideHrsSingleDay @DateParm = '3/5/2018'
SELECT
  'Total Ride Hours for 3/5/2018:',
  @RideHrs
```

- Execute TVF into variable:

```
-- Create @StationStats
DECLARE @StationStats TABLE(
StartStation nvarchar(100),
RideCount int,
TotalDuration numeric)
-- Populate @StationStats with the results of the function
INSERT INTO @StationStats
SELECT TOP 10 *
-- Execute SumStationStats with 3/15/2018
FROM dbo.SumStationStats ('3/15/2018')
ORDER BY RideCount DESC
-- Select all the records from @StationStats
SELECT *
FROM @StationStats
```

Chapter 6

Final Words

6.1 Beginner Resources by Topic

6.1.1 Getting Set-Up with R & RStudio

- **Download & Install R:**
  - https://cran.r-project.org
  - For Mac: click on **Download R for (Mac) OS X**, look at the top link under **Files**, which at time of writing is **R-3.2.4.pkg**, and download this if compatible with your current version mac OS (Mavericks 10.9 or higher). Otherwise download the version beneath it which is compatible for older mac OS versions. Then install the downloaded software.
  - For Windows: click on **Download R for Windows**, then click on the link **install R for the first time**, and download from the large link at the top of the page which at time of writing is **Download R 3.2.4 for Windows**. Then install the downloaded software.
- **Download & Install RStudio:**
  - https://www.rstudio.com/products/rstudio/download/
  - For Mac: under the **Installers for Supported Platforms** heading click the link with **Mac OS X** in it. Install the downloaded software.
  - For Windows: under the **Installers for Supported Platforms** heading click the link with **Windows Vista** in it. Install the downloaded software.
- **Exercises in R: swirl (HIGHLY RECOMMENDED):**
  - http://swirlstats.com/students.html

- **Data Prep**:
  - Intro to dplyr: https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html
  - Data Manipulation (detailed): http://www.sr.bham.ac.uk/~ajrs/R/index.html
  - Aggregation and Restructing Data (base & reshape): http://www.r-statistics.com/2012/01/aggregation-and-restructuring-data-from-r-in-action/
- **Data Types intro**: Vectors, Matrices, Arrays, Data Frames, Lists, Factors: http://www.statmethods.net/input/datatypes.html
- **Using Dates and Times**: http://www.cyclismo.org/tutorial/R/time.html
- **Text Data and Character Strings**: http://gastonsanchez.com/Handling_and_Processing_Strings_in_R.pdf
- **Data Mining**: http://www.rdatamining.com

- **Data Viz**:
  - ggplot2 Cheat Sheet (RECOMMENDED): http://zevross.com/blog/2014/08/04/beautiful-plotting-in-r-a-ggplot2-cheatsheet-3/
  - ggplot2 theoretical tutorial (detailed but RECOMMENDED): http://www.ling.upenn.edu/~joseff/avml2012/
  - Examples of base R, ggplot2, and rCharts: http://patilv.com/Replication-of-few-graphs-charts-in-base-R-ggplot2-and-rCharts-part-1-base-R/
  - Intro to ggplot2: http://heather.cs.ucdavis.edu/~matloff/GGPlot2/GGPlot2Intro.pdf
- **Interactive Visualisations**:
  - Interactive graphics (rCharts, jQuery): http://www.computerworld.com/article/2473365/business-intelligence/business-intelligence-106897-how-to-turn-csv-data-into-interactive-visualizations-with-r-and-rchart.html

---

- **Statistics**:
  - Detailed Statistics Primer: http://health.adelaide.edu.au/psychology/ccs/docs/lsr/lsr-0.3.pdf
  - Beginner guide to statistical topics in R: http://www.cyclismo.org/tutorial/R/
- **Linear Models**: http://data.princeton.edu/R/gettingStarted.html
- **Time Series Analysis**: https://www.otexts.org/fpp/resources
- **Little Book of R series**:
  - Time Series: http://a-little-book-of-r-for-time-series.readthedocs.org/en/latest/
  - Biomedical Statistics: http://a-little-book-of-r-for-biomedical-statistics.readthedocs.org/en/latest/

  - Multivariate Statistics: http://little-book-of-r-for-multivariate-analysis.readthedocs.org/en/latest/

- **RStudio Cheat Sheets**:
  - RStudio IDE: http://www.rstudio.com/wp-content/uploads/2016/01/rstudio-IDE-cheatsheet.pdf
  - Data Wrangling (dplyr & tidyr): https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf
  - Data Viz (ggplot2): https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf
  - Reproducible Reports (markdown): https://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf
  - Interactive Web Apps (shiny): https://www.rstudio.com/wp-content/uploads/2015/02/shiny-cheatsheet.pdf

---

6.1.2 Specialist Topics

- **Google Analytics**: http://online-behavior.com/analytics/r
- **Spatial Cheat Sheet**: http://www.maths.lancs.ac.uk/~rowlings/Teaching/UseR2012/cheatsheet.html
- **Translating between R and SQL**: http://www.burns-stat.com/translating-r-sql-basics/
- **Google's R style guide**: https://google.github.io/styleguide/Rguide.xml

---

6.1.3 Operational Basics

- **Working Directory**:
  Example on a mac = `setwd("~/Desktop/R")` or `setwd("/Users/CRT/Desktop/R")`
  Example on windows = `setwd("C:/Desktop/R")`

- **Help**:
```
?functionName
example(functionName)
args(functionName)
help.search("your search term")
```

- **Assignment Operator**: `<-`

6.2 Getting Your Data into R

1. Loading Existing Local Data

(a) When already in the working directory where the data is

Import a local **csv** file (i.e. where data is separated by **commas**), saving it as an object:

```
#this will create a data frame called "object"
#the header argument is defaulted to TRUE, i.e. read.csv assumes your file has a heade
object <- read.csv("xxx.csv")

#if your csv does not have a header row, add header = FALSE to the command
#in this call default column headers will be assigned which can be changed
object <- read.csv("xxx.csv", header = FALSE)
```

Import a local tab delimited file (i.e. where data is separated by **tabs**), saving is as an object:

(b) When NOT in the working directory where the data is

For example to import and save a local **csv** file from a different working directory you can either need to specify the file path (operating system specific), e.g.:

```
#on a mac
object <- read.csv("~/Desktop/R/data.csv")

#on windows
object <- read.csv("C:/Desktop/R/data.csv")
```

OR

You can use the file.choose() command which will interactively open up the file dialog box for you to browse and select the local file, e.g.:

```
object <- read.csv(file.choose())
```

(c) Copying and Pasting Data

For relatively small amounts of data you can do an equivalent copy paste (operating system specific):

```
#on a mac
object <- read.table(pipe("pbpaste"))

#on windows
object <- read.table(file = "clipboard")
```

2. Loading Non-Numerical Data - character strings

Be careful when loading text data! R may assume character strings are statistical factor variables, e.g. "low", "medium", "high", when are just individual labels like names. To specify text data NOT to be converted into factor variables, add `stringsAsFactor = FALSE` to your `read.csv/read.table` command:

```
object <- read.table("xxx.txt", stringsAsFactors = FALSE)
```

3. Downloading Remote Data

For accessing files from the web you can use the same `read.csv/read.table` commands. However, the file being downloaded does need to be in an R-friendly format (maximum of 1 header row, subsequent rows are the equivalent of one data record per row, no extraneous footnotes etc.). Here is an example downloading an online csv file from Pew Research:

```
object <- read.csv("https://vincentarelbundock.github.io/Rdatasets/csv/datasets/AirPassengers.csv
```

4. Other Formats - Excel, SPSS, SAS etc.

For other file formats, you will need specific R packages to import these data.

Here's a good site for an overview: http://www.statmethods.net/input/importingdata.html

Here's a more detailed site: http://r4stats.com/examples/data-import/

Here's some info on the `foreign` package for loading statistical software file types: http://www.ats.ucla.edu/stat/r/faq/inputdata_R.htm

---

6.3 Getting Your Data out of R

1. Exporting data

Navigate to the working directory you want to save the data table into, then run the command (in this case creating a tab delimited file): - write.table(object, "xxx.txt", sep = "\t")

2. Save down an R object Navigate to the working directory you want to save the object in then run the command:

- save(object, file = "xxx.rda")

reload the object: - load("xxx.rda")
