Data

301113 Programming for Data Science

WESTERN SYDNEY
UNIVERSITY

School of Computer, Data and Mathematical Sciences

Week 6

Outline

WESTERN SYDNEY UNIVERSITY

1 Loading Data

2 Viewing Data

3 Temperature at JFK airport

4 Tidyverse

5 Exporting Data

What is data?

Data Science is the science of turning data in to information. Data is any sequence of symbols. For data to be useful, it must have meaning.

Most data is a sample from some population. It is possible to obtain data from the whole population, but is rare (usually due to population sizes).

Data can be used:

- to make inferences about the population (does smoking cause cancer?).
- to make future predictions (will customers buy this product).
- to detect anomalies (was there unauthorised access to a machine?).

This is done by first assuming a model of the population then reverse engineering the details of the population using the sample data.

Examining Data

A typical start to a project requires:

1. Loading data
2. Examining the data
3. Arranging the data to suit our analysis
4. Analysing the data
5. Modelling the data

Outline

1 Loading Data

2 Viewing Data

3 Temperature at JFK airport

4 Tidyverse

5 Exporting Data

Data Structures: Tabular Data

The types of data can be categories based on the way that it is organised. The most common form of data is tabular data. Tabular data consists of a set of observations, with measurements of specific variables.

- The observations are the rows of the table.
- The measurements are the columns of the table.

The objects are the items found in the population (e.g. people, computers, viruses). The variables are the things that we measure from the objects (e.g. age, operating system, infection rate).

Data Structures: Tabular Data Files

WESTERN SYDNEY UNIVERSITY

Tabular data can be stored in text files, but also exists in proprietary formats such as Excel spreadsheets.

The simplest and most accessible file format is Comma Separated Value (CSV) format.

CSV files contain:

- one line for each row
- the items in the rows are separated by commas

If commas exist in the data, then the data is encased in quotes (e.g. “Smith, Steve”).

CSV files might contain a header line, with column names.

Example

Here is an example of a CSV file `https://archive.ics.uci.edu/ml/datasets/Abalone`. Let’s view it in a text editor.

Reading CSV files

CSV files are read using the function:

```
d <- read.csv(fileName)
```

The file contents are stored in a data frame within the variable `d` (or which ever variable name you choose). The function assumes that the CSV file has a header, if not, the argument `header` must be set to `FALSE`.

There are also arguments for changing the separator character, and for ignoring comments in the CSV file.

Reading Excel files

The function to read Excel files is not included with the R base libraries, so a library must be downloaded and activated.

```
library("readxl") # load library of functions for excel files

d <- read_excel(fileName)
```

The function provides arguments for selecting a specific sheet, a specific range of cells and a locgical variable declaring if the first row contains column names.

Note that the data will be stored in a data frame.

Data Structures: Hierarchical Data

Hierarchical data (such as taxonomies) form a tree structure and so are not suitable to be stored in a table.

Commonly used hierarchical data formats are:

- XML
- JSON
- YAML

There are libraries than contain functions to read these file formats (e.g. jsonlite). The data is stored in a list data structure.

Example

A JSON file containing the description of the Magic the Gathering card "Gush"

Data Structures: Graphs

A graph contains a set of objects and the relationships (edges) between each object, where the edges can be weighted or signed (positive or negative).

Graph data describes the relationships between objects. It usually comes in one of two forms.

- A matrix with one row and one column representing each item. The elements of the graph represent the absence or presence of an edge, or the weight, or sign of the edge.
- A table of edge pairs, where each row contains two edges that are connected and possibly a third element showing the weight or sign of the edge.

Data Structures: Graphs

A graph contains a set of objects and the relationships (edges) between each object, where the edges can be weighted or signed (positive or negative).

Graph data describes the relationships between objects. It usually comes in one of two forms.

- A matrix with one row and one column representing each item. The elements of the graph represent the absence or presence of an edge, or the weight, or sign of the edge.
- A table of edge pairs, where each row contains two edges that are connected and possibly a third element showing the weight or sign of the edge.

There are libraries that contain functions for storing and manipulating graphs (e.g. igraph).

Example

Example graph data files are found at `http://snap.stanford.edu/data/index.html`

Outline

1 Loading Data

2 Viewing Data

3 Temperature at JFK airport

4 Tidyverse

5 Exporting Data

Quick Checks

After loading data, it is good practice to examine the data to make sure that:

- the data is the data that you want,
- the data variables are the correct type,

A quick check of the contents can be performed using the function `head`.

```r
head(mtcars) #display the top six rows of the data
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Examining variable type

In order to perform the correct analysis, we must ensure that the variables within the data frame are all of the correct type.

```r
ls.str(mtcars)
```

```
am :  num [1:32] 1 1 1 0 0 0 0 0 0 0 ...
carb :  num [1:32] 4 4 1 1 2 1 4 2 2 4 ...
cyl :  num [1:32] 6 6 4 6 8 6 8 4 4 6 ...
disp :  num [1:32] 160 160 108 258 360 ...
drat :  num [1:32] 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
gear :  num [1:32] 4 4 4 3 3 3 3 4 4 4 ...
hp :  num [1:32] 110 110 93 110 175 105 245 62 95 123 ...
mpg :  num [1:32] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
qsec :  num [1:32] 16.5 17 18.6 19.4 17 ...
vs :  num [1:32] 0 0 1 1 0 1 0 1 1 1 ...
wt :  num [1:32] 2.62 2.88 2.32 3.21 3.44 ...
```

This data frame contains all numeric variables, but the `vs` (engine shape) and `am` (transmission type) should be factors (categorical), not numeric.

Correcting Data Types

We can explicitly tell R the types of the variables when loading a table by setting the `colClasses` argument when running `read.csv`, or we can correct them after loading.

```r
## Use factor to convert to categorical variables
mtcars$vs <- as.factor(mtcars$vs)
mtcars$am <- as.factor(mtcars$am)
```

We can also use `as.numeric()`, `as.character()`, `as.Date()` if required.

Outline

1 Loading Data

2 Viewing Data

3 Temperature at JFK airport

4 Tidyverse

5 Exporting Data

Analysis Required

We will demonstrate a typical processes used when arranging data.

We have been supplied with data containing flight and weather information from New York City.

Our first task is to identify if the maximum temperatures for each day are roughly the same at each of the NYC airports.

We need to:

1. Load data
2. Examine the data
3. Arrange the data to suit our analysis
4. Analyse the data

All of the code required should be written in an R script file.

Loading the Weather data

The data is in CSV format, so we load the file using.

```r
weather <- read.csv("data/nycweather.csv")
head(weather, 2)
```

```
  origin year month day hour  temp  dewp humid wind_dir wind_speed wind_gust
1    EWR 2013     1   1    1 39.02 26.06 59.37      270   10.35702        NA
2    EWR 2013     1   1    2 39.02 26.96 61.63      250    8.05546        NA
  precip pressure visib
1      0   1012.0    10
2      0   1012.3    10
```

Working Directory

Remember that the function `read.csv` is looking for the file in your current working directory. We wrote "data/nycweather.csv" to tell R to look in the "data" directory within the working directory. Check the name of you working directory using `getwd()`. If you want to change the working directory use `setwd("path to dir")`.

Examine the Weather Data

Before we can arrange the data, we must familiarise ourselves with the data.

1. Use `head` to examine the first few rows of the data.
2. Identify the name of the variable that contains the airport names.
3. Identify the name of the variable that contains the temperature.
4. Identify the different airport names.

Categories of a variable

Rather than printing out the variable (which could be long), we can use the functions `unique` or `table` to show the set of categories within a string variable.

Extracting the Data

After the examining the data, we find that the records are associated to each airport using the `origin` variable (column).

We can extract the JFK airport data using:

```
jfkWeather <- subset(weather, subset = (origin == "JFK"))
```

Problem

To compute the maximum daily temperature at each airport, we will first extract the data for each airport and store it in its own variable.
Show how to extract the rows for each airport using either:

- a for loop
- the function `lapply`.

Obtaining the Date

We need to compute the maximum temperature for each day, but we find that there is no date variable. The data contains the year, month and day variables, so we want to combine them to form a date variable.

Date variable type

- Date variables have their own type. There are many problems that can occur when using historical dates due to changes in calendars.
- Dates can be provided in many formats, but the default format follows the rules of the ISO 8601 international standard which expresses a day as ' "2001-02-23" '.
- To convert a string to a date, us `as.Date`.
- We can use many mathematical operations on dates.
- Plots also make use of the date data type.

Merging Columns for the Date

We need to form the date as the string "YYYY-MM-DD", taking each part from the year, month and day columns of the data.

Recall that `paste` can be used to combine strings. Therefore, we can use a for loop or lapply to combine the year, month and day into a date string for each row.

Merging Columns for the Date

We need to form the date as the string "YYYY-MM-DD", taking each part from the year, month and day columns of the data.

Recall that `paste` can be used to combine strings. Therefore, we can use a for loop or lapply to combine the year, month and day into a date string for each row.

We find that `paste` is vectorised, meaning that we can provide vectors of strings and it will provide a vector of combined strings (so we don't need the for loop or lapply).

```r
dateString <- paste(weather$year, weather$month, weather$day, sep = "-")
weatherDate <- as.Date(dateString) # convert the strings to date type
```

Dates for all Airports

We have split the `weather` data into three data frames; one for each airport. Now we need to add columns to each data frame containing the date.

Problem

Use a for loop or lapply to add the date column to each airport data frame.

Maximum Temperature

Finally, we need to compute the maximum temperature for each day. We can do this by finding the maximum temperature for each grouping of date. A for loop can be used for this, but this task is perfect for `tapply`

`tapply` for Maximum Temperature

`tapply` takes a vector of values that we want to apply a function to, a grouping vector showing which of the values we group, and a function to apply to the groups of values.

```
maxTemp <- tapply(weather$temp, weather$date, max)
```

We want the `max` temperature grouped by date.

Maximum Temperature for each Airport

WESTERN SYDNEY UNIVERSITY

Problem

Given that we have the data frames for each airport containing the temperatures and dates, compute the maximum temperature for each day for each airport. Use either a for loop or lapply.

Visualising the Data

Now that we have the daily maximum temperature for each airport, we can plot them to visualise the relationship. Before we can do that, we need the dates associated to each temperature.

`tapply` has included the dates as the `names` of each maximum temperature, but as a string,

```
head(maxTemp)
```

```
2013-01-01 2013-01-02 2013-01-03 2013-01-04 2013-01-05 2013-01-06
     41.00      35.06      33.98      39.92      44.06      48.02
```

So we need to convert them to a date again.

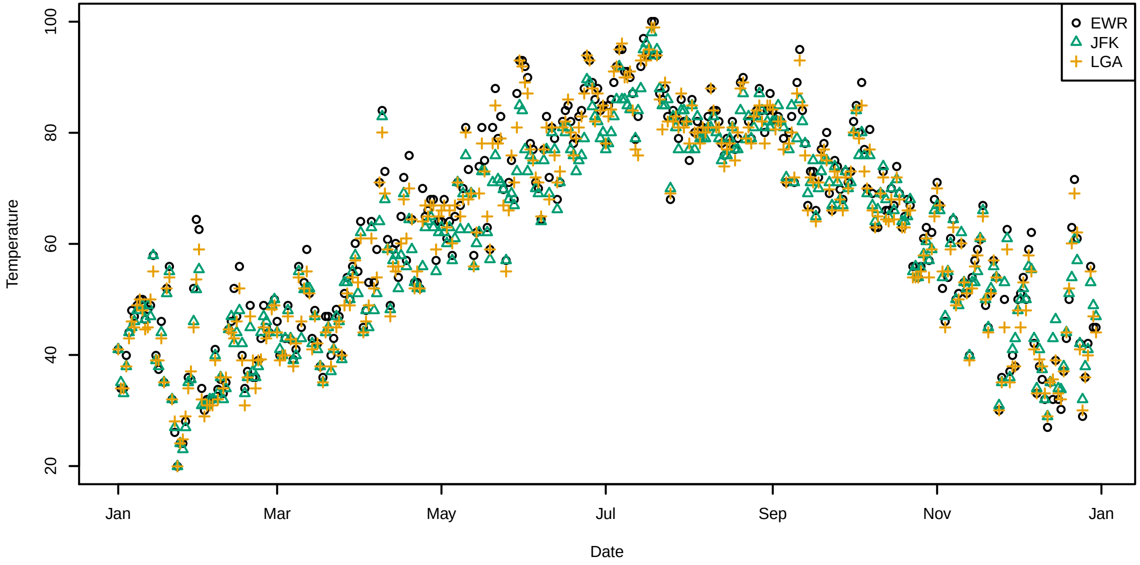
```
computeMaxTemp <- function(weather) {
    maxTemp <- tapply(weather$temp, weather$date, max)
    dateTemp <- as.Date(names(maxTemp)) # extract the dates
    df <- data.frame(date = dateTemp, temp = maxTemp)
    return(df)
}

airportMaxTemp <- lapply(airportWeather, computeMaxTemp)
```

Still visualising the data

We can now plot the temperatures for each airport. We should choose different colours and point shapes for each airport.

```r
## Note that plot creates the plot, but points plots over the current plot
plot(airportMaxTemp[[1]]$date, airportMaxTemp[[1]]$temp, col = 1, pch = 1,
    xlab = "Date", ylab = "Temperature")
points(airportMaxTemp[[2]]$date, airportMaxTemp[[2]]$temp, col = 2, pch = 2)
points(airportMaxTemp[[3]]$date, airportMaxTemp[[3]]$temp, col = 3, pch = 3)
legend("topright", legend = airportsNames, col = 1:3, pch = 1:3)
```

Maximum Temperatures for each Day



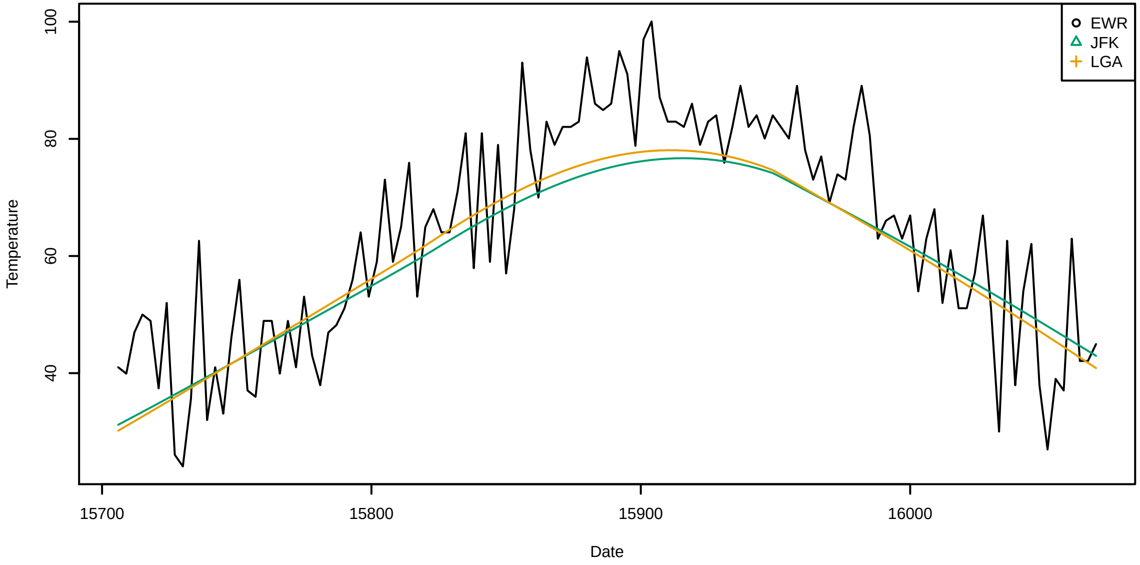
Smoothed Maximum Temperatures for each day

There is not much that we can see from the plot due to the high variation in maximum temperatures. To get an indication of the trend of the data, we can plot a smoothed version of the data.

R provides a function `lowess` that provides a set of smoothed points, The “smoothness” is dependent on the argument `f` which tells `lowess` what proportion of the data to use to smooth each point. Larger `f` leads to smoother data.

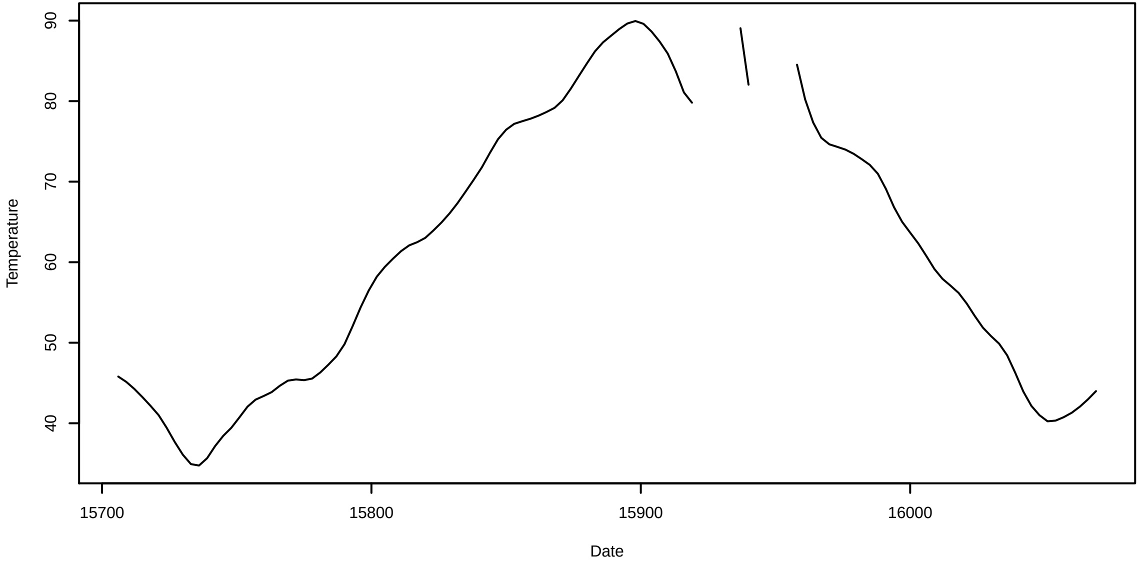
```r
plot(lowess(airportMaxTemp[[1]]$date, airportMaxTemp[[1]]$temp), type = "l", col = 1,
    xlab = "Date", ylab = "Temperature")
lines(lowess(airportMaxTemp[[2]]$date, airportMaxTemp[[2]]$temp), col = 2, pch = 2)
lines(lowess(airportMaxTemp[[3]]$date, airportMaxTemp[[3]]$temp), col = 3, pch = 3)
legend("topright", legend = airportsNames, col = 1:3, pch = 1:3)
```

Smoothed Maximum Temperatures for each day



Still not smooth

The JFK and LGA lines are smooth, but the EWR is not. Let’s change the `f` value to 0.1.



Missing Values

We have found that there is a discontinuity in the data. Further examination reveals that there are missing temperatures.

```r
## check the position of missing values appearing in the maximum temperatures
which(is.na(airportMaxTemp[[1]]$temp))
```

```
[1] 234
```

What can we do about this?

Missing Values

We have found that there is a discontinuity in the data. Further examination reveals that there are missing temperatures.

```r
## check the position of missing values appearing in the maximum temperatures
which(is.na(airportMaxTemp[[1]]$temp))
```

```
[1] 234
```

What can we do about this?

Since the temperatures we have are maximums, we can check the original hourly data, to examine if the missing values are likely to effect the maximum.

```r
## check the position of missing values appearing in the hourly data
which(is.na(airportWeather[[1]]$temp))
```

```
[1] 5592
```

Missing Values for the Day

```
## Find the day that the missing temperature occured
airportWeather[[1]][5592,]

     origin year month day hour temp dewp humid wind_dir wind_speed wind_gust
5592    EWR 2013     8  22    9   NA   NA    NA      320   12.65858        NA
     precip pressure visib       date
5592   0.13       NA     7 2013-08-22
## Examing temperatures from the day that the missing temperature occured
subset(airportWeather[[1]], subset = ((month == 8) & (day == 22)))

     origin year month day hour  temp  dewp humid wind_dir wind_speed wind_gust
5583    EWR 2013     8  22    0 77.00 69.08 76.57      230    6.90468        NA
5584    EWR 2013     8  22    1 77.00 69.98 78.96      230    6.90468        NA
5585    EWR 2013     8  22    2 75.92 69.08 79.37      220    6.90468        NA
5586    EWR 2013     8  22    3 75.02 69.98 84.34      220    8.05546        NA
5587    EWR 2013     8  22    4 75.02 69.98 84.34      240    5.75390        NA
5588    EWR 2013     8  22    5 75.02 69.98 84.34      230    8.05546        NA
5589    EWR 2013     8  22    6 75.02 69.98 84.34      210    5.75390        NA
```

Dealing with the missing value

We find that the missing value is not likely to effect the maximum temperature for the day, so we can adjust our code to ignore it.

All this requires is including `na.rm = TRUE` when we use `max`.

Adjust the code and and examine the results.

Missing Values

There are many times that we will have to deal with missing values and we can see that it depends on how we are using the data. The main rule is to make sure that what ever we do, our changes don't effect the results of the analysis.

Outline

1 Loading Data

2 Viewing Data

3 Temperature at JFK airport

4 Tidyverse

5 Exporting Data

Introduction to Tidyverse

Over the past decade the Tidyverse libraries have become popular for data wrangling in R.

Benefits of the tidyverse way:

- Many of the underlying functions in tidyverse packages are written in C and so are very fast.
- The functions encourage piping (connecting the output of a function to the input of another), making the processing pipeline more obvious.

To begin using the tidyverse libraries, first install them:

```
install.packages("tidyverse")
```

Then in each script file:

```
library("tidyverse")
```

Traditional vs Tidyverse Analysis

There are functions that can be used in R to manipulate data for analysis where the data is provided to the function and the modified data is returned.

Tidyverse libraries allow us to set up the data processing using pipes. For example:

```
modifiedData <- data %>% filter() %>% sort() %>% clean()
```

The data is piped through the function `filter()`, then the output is piped into the function `sort()`, then the output is piped through the function `clean()` and finally assigned to `modifedData`.

Using pipes means that we don't have to store the intermediate results in a variable.

Commonly used functions

Many of the data manipulation functions come from the tidyverse library dplyr

- mutate() adds new variables that are functions of existing variables.
- select() picks variables based on their names.
- filter() picks cases based on their values.
- summarise() reduces multiple values down to a single summary.
- arrange() changes the ordering of the rows.

There is also the function `group_by()` that sets the groups that the next function will use.

A useful summary of the functions is found here
`https://github.com/rstudio/cheatsheets/blob/master/data-transformation.pdf`.

Maximum temperature using Tidyverse

We will now demonstrate the use of tidyverse and piping by repeating the previous analysis (maximum temperature for each airport).

The steps taken are:

- Create a new variable containing the date (using mutate and unite)
- Compute the maximum temperature for each day (using summarise)
- Extract each airport (using filter).

Tidyverse example

Below is an example of how to compute the maximum temperatures using tidyverse function and pipes (“%>%”).

```
maxTemp <- weather %>%
     unite(date, year, month, day, sep = "-") %>% # combine columns
     mutate(date = as.Date(date)) %>% # create a new column
     group_by(origin, date) %>% summarize(maxTemp = max(temp, na.rm = TRUE)) # reduce

maxTempJFK <- maxTemp %>% filter(origin == "JFK") # select airport
```

We can treat this as is the data frame `weather` is being piped through each function one at a time and when it reached the end of the pipeline, it is assigned to the variable `maxTemp`

Explaining each function

The previous pipeline can be explained further.

- `unite(date, year, month, day, sep = "-")` combines the year, month and day colummns into a string and separates them with a "-".
- `mutate(date = as.Date(date))` creates a new column called `date` containing the columns `date` passed through the function `as.Date`.
- `group_by(origin, date)` sets the variables origin and date as variables to be grouped by, for the next function.
- `summarize(maxTemp = max(temp, na.rm = TRUE))` creates the new column `maxTemp` containing the maxium of the column `temp`, and also follows any groupings previously assigned.

Further explanation of these functions and more are here
`https://r4ds.had.co.nz/transform.html`

Explaining each function

The previous pipeline can be explained further.

- `unite(date, year, month, day, sep = "-")` combines the year, month and day colummns into a string and separates them with a "-".
- `mutate(date = as.Date(date))` creates a new column called `date` containing the columns `date` passed through the function `as.Date`.
- `group_by(origin, date)` sets the variables origin and date as variables to be grouped by, for the next function.
- `summarize(maxTemp = max(temp, na.rm = TRUE))` creates the new column `maxTemp` containing the maxium of the column `temp`, and also follows any groupings previously assigned.

Further explanation of these functions and more are here
`https://r4ds.had.co.nz/transform.html`

Problem

Using the data `weather`, write the code to create a new column containing the temperature measured in degrees Celsius.

Outline

1 Loading Data

2 Viewing Data

3 Temperature at JFK airport

4 Tidyverse

5 **Exporting Data**

Finishing an analysis

When finishing an analysis, we want to keep the results that have been generated. There are two options.

1. Save the data frames to CSV files.
2. Don't save the data frames, but ensure that your code is in a state where you can easily generate the results again.

If the analysis is quick, the preference should be option 2. If it takes three months to generate the results, then option 1 is a better choice.

Ensuring that your code can be used to reproduce the results from the original data means that others can verify your work, bugs can fixed, and if the original data changes, then your code can easily be run again.

Writing data frames to a file.

Data frames are simply tables, therefore, we can export them to a CSV file.

```r
write.csv(weather, file = "/tmp/weather.csv")
```

Note that the above function also includes row numbers in the CSV file, which is not always wanted, but it has arguments to change that behaviour.

Working Directory

- `write.csv` will create the file in your current working directory.
- If the file already exists, it will be overwritten!
- If the path is provided, it will be created at the provided path.

Summary

- Data comes in many formats, the most typical is tabular data.
- When loading data, we must make sure that R has assigned the correct variable types.
- Analysis usually requires manipulating the data.
- The method that we use to deal with missing values should not effect our results.
- The Tidyverse libraries provide another way of manipulating data.
- Data tables can be saved to files, but it is important to keep you code in a script so that the analysis is reproducible.