Tidy Data and Iteration

Using tidyr to create tidy data

Statistical Computing and Empirical Methods
Unit EMATM0061, Data Science MSc

Rihuan Ke
rihuan.ke@bristol.ac.uk

Teaching Block 1, 2024

University of BRISTOL

What we will cover in this lecture

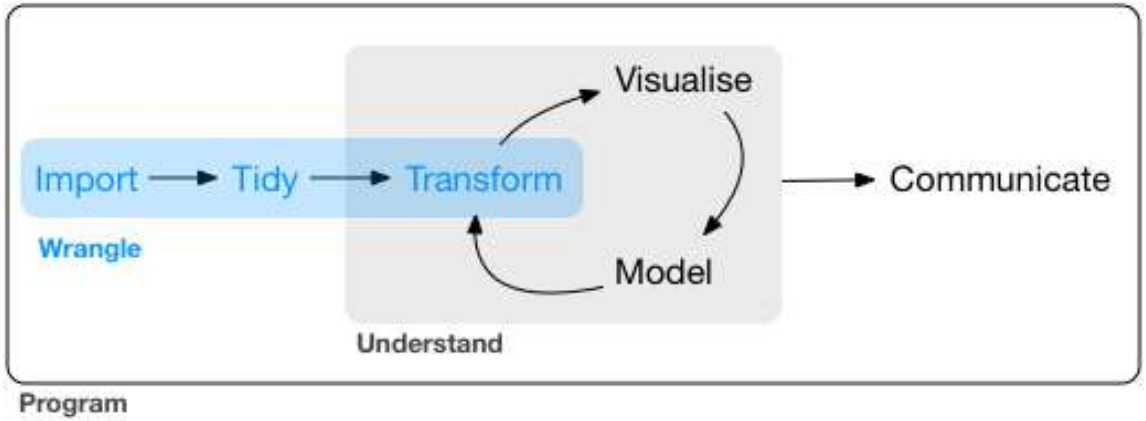
- We will introduce the concepts of tidy data
- We will see how to reshape data frames with the pivot functions.
- We will also look at uniting and separating columns within data.
- We will see how to use the map function for efficient iteration in R.
- We will also look at some basic methods for handling missing data.

Tidying data

Raw data may have complex structures and missing values

It is said that cleaning and preparing data can take 80% of the time in data analysis process

Tidying data aims to make the data well structured and clean to facilitate analysis



source: r4ds.had.co.nz

What is tidy data?

Tidy data is data where

1. Each column corresponds to a variable (a property or quality of individual examples)
2. Each row corresponds to a specific and unique observation (an instance of a specific type of things)

This is tidy data:

species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year
Adelie	Torgersen	40.9	16.8	191	3700	female	2008
Adelie	Biscoe	37.8	20	190	4250	male	2009
Adelie	Dream	36.9	18.6	189	3500	female	2008
Adelie	Torgersen	34.6	17.2	189	3200	female	2008
Adelie	Dream	38.8	20	190	3950	male	2007
Chinstrap	Dream	46.4	17.8	191	3700	female	2008
Chinstrap	Dream	58	17.8	181	3700	female	2007
Chinstrap	Dream	45.6	19.4	194	3525	female	2009
Chinstrap	Dream	52	20.7	210	4800	male	2008
Chinstrap	Dream	52.7	19.8	197	3725	male	2007
Gentoo	Biscoe	43.5	14.2	220	4700	female	2008
Gentoo	Biscoe	45.4	14.6	211	4800	female	2007
Gentoo	Biscoe	46.3	15.8	215	5050	male	2007
Gentoo	Biscoe	50.5	15.9	225	5400	male	2008
Gentoo	Biscoe	49	16.1	216	5550	male	2007

Each row corresponds to a penguin instance, each column corresponds to a property (e.g., bill length)

What is tidy data?

This is not tidy data:

Species	Island	Bill length (mm)	Bill depth (mm)	Flipper length (mm)	Body mass (g)
Adelie	Dream	39.70	17.90	193.00	4250
Adelie	Dream	39.60	18.80	190.00	4600
Adelie	Dream	39.20	21.10	196.00	4150
Adelie	Biscoe	35.30	18.90	187.00	3800
Adelie	Dream	36.50	18.00	182.00	3150
	Average	38.06	18.94	189.60	3990
Chinstrap	Dream	51.30	19.20	193.00	3650
Chinstrap	Dream	46.50	17.90	192.00	3500
Chinstrap	Dream	49.00	19.60	212.00	4300
Chinstrap	Dream	50.80	19.00	210.00	4100
Chinstrap	Dream	45.90	17.10	190.00	3575
	Average	48.70	18.56	199.40	3825
Gentoo	Biscoe	44.40	17.30	219.00	5250
Gentoo	Biscoe	50.80	17.30	228.00	5600
Gentoo	Biscoe	50.40	15.70	222.00	5750
Gentoo	Biscoe	45.80	14.20	219.00	4700
Gentoo	Biscoe	55.90	17.00	228.00	5600
	Average	49.46	16.30	223.20	5380
	Overall average	45.41	17.93	204.07	4398

There are several rows in a different format, and they are not associated with a specific observation

Why tidy data?

Tidy data is typically far easier to manipulate and apply statistical analysis to in R.

- Uniform formats across the rows and columns
- Observations (that are needed by the process of statistical analysis) can be easily accessed

Note: “Tidy data” here is a technical term ie . not just “data that is tidy”/ “data that is well”

In other contexts, non-tidy data has several advantages:

- Non-tidy data can be more accessible visually for non-specialists.
- Non-tidy can offer substantial performance and space advantages in certain contexts.
- Specialist fields e.g. computer vision often have unique standards for storing data.

1. Reshaping data

Reshaping data to transform from narrow data (also called long data) to wide data, and vice versa

Narrow data

```
## # A tibble: 9 × 3
##   species   property  value
##   <fct>     <chr>     <dbl>
## 1 Adelie    bill       38.8
## 2 Adelie    flipper   190.
## 3 Adelie    weight   3701.
## 4 Chinstrap bill       48.8
## 5 Chinstrap flipper   196.
## 6 Chinstrap weight   3733.
## 7 Gentoo    bill       47.5
## 8 Gentoo    flipper   217.
## 9 Gentoo    weight   5076.
```

Reshape

Wide data

```
## # A tibble: 3 × 4
##   species    bill flipper weight
##   <fct>     <dbl>   <dbl>  <dbl>
## 1 Adelie     38.8    190.  3701.
## 2 Chinstrap  48.8    196.  3733.
## 3 Gentoo     47.5    217.  5076.
```

The two regions highlighted in red correspond to the same instance

1. Reshaping data

Suppose we have the following data (called *penguins_summary_narrow*) represented in a narrow format

```
print(penguins_summary_narrow)
```

```
## # A tibble: 9 × 3
##   species   property  value
##   <fct>     <chr>     <dbl>
## 1 Adelie    bill       38.8
## 2 Adelie    flipper   190.
## 3 Adelie    weight   3701.
## 4 Chinstrap bill       48.8
## 5 Chinstrap flipper   196.
## 6 Chinstrap weight   3733.
## 7 Gentoo    bill       47.5
## 8 Gentoo    flipper   217.
## 9 Gentoo    weight   5076.
```

Is this tidy data?

1. Reshaping data

We can reshape the *penguins_summary_narrow* into a wide format using the function pivot_wider:

```
penguins_summary_wide <- penguins_summary_narrow %>%
  pivot_wider(names_from = property, values_from = value)
print(penguins_summary_wide)
```

```
## # A tibble: 3 × 4
##   species    bill flipper weight
##   <fct>     <dbl>   <dbl>  <dbl>
## 1 Adelie     38.8    190.  3701.
## 2 Chinstrap  48.8    196.  3733.
## 3 Gentoo     47.5    217.  5076.
```

1. Reshaping data

It is also possible to reshape data from the wide format to the narrow format:

```
penguins_summary_wide %>%
  pivot_longer(c(bill, flipper, weight), names_to='property', values_to='value')
```

```
## # A tibble: 9 × 3
##   species   property  value
##   <fct>     <chr>     <dbl>
## 1 Adelie    bill       38.8
## 2 Adelie    flipper   190.
## 3 Adelie    weight   3701.
## 4 Chinstrap bill       48.8
## 5 Chinstrap flipper   196.
## 6 Chinstrap weight   3733.
## 7 Gentoo    bill       47.5
## 8 Gentoo    flipper   217.
## 9 Gentoo    weight   5076.
```

```
print(penguins_summary_wide)
```

```
## # A tibble: 3 × 4
##   species    bill flipper weight
##   <fct>     <dbl>   <dbl>  <dbl>
## 1 Adelie     38.8    190.  3701.
## 2 Chinstrap  48.8    196.  3733.
## 3 Gentoo     47.5    217.  5076.
```

2. Uniting and separating data

Separate: separate a character column into multiple columns

Unite: paste together multiple columns into one.

```
## # A tibble: 3 × 3
##   species    bill flipper_over_weight
##   <fct>     <dbl> <chr>
## 1 Adelie     38.8 190/3700.7
## 2 Chinstrap  48.8 195.8/3733.1
## 3 Gentoo     47.5 217.2/5076
```

Separate →

← Unite

```
## # A tibble: 3 × 4
##   species    bill flipper weight
##   <fct>     <dbl>   <dbl>  <dbl>
## 1 Adelie     38.8    190   3701.
## 2 Chinstrap  48.8    196.  3733.
## 3 Gentoo     47.5    217.  5076
```

The flipper and weight columns on the right data frame are combined into a single column call flipper_over_weight on the left

We often encounter data with multiple variables within a single column (e.g., on the left)

However, extracting the individual variables (e.g., flipper, weight) makes it easier to perform tasks such as statistical analysis and visualisation.

Separate

```
print(uni_df)
```

```
## # A tibble: 3 × 3
##   species    bill flipper_over_weight
##   <fct>     <dbl> <chr>
## 1 Adelie     38.8 190/3700.7
## 2 Chinstrap  48.8 195.8/3733.1
## 3 Gentoo     47.5 217.2/5076
```

Suppose that we have a data frame called *uni_df*, and we want to separate one of its columns called *flipper_over_weight:*

The separate function (in the *tidyr* package):

```
sep_df <- uni_df %>%
  separate(flipper_over_weight, into=c("flipper", "weight"), sep="/")
print(sep_df)
```

```
## # A tibble: 3 × 4
##   species    bill flipper weight
##   <fct>     <dbl> <chr>   <chr>
## 1 Adelie     38.8 190     3700.7
## 2 Chinstrap  48.8 195.8   3733.1
## 3 Gentoo     47.5 217.2   5076
```

Note: by default, the separate function preserves the data type of the column (so flipper and weight are character columns).

Separate

Suppose that we have a data frame called *uni_df*, and we want to separate one of its columns called *flipper_over_weight:*

```
print(uni_df)
```

```
## # A tibble: 3 × 3
##   species    bill flipper_over_weight
##   <fct>     <dbl> <chr>
## 1 Adelie     38.8 190/3700.7
## 2 Chinstrap  48.8 195.8/3733.1
## 3 Gentoo     47.5 217.2/5076
```

Use "convert = TRUE" to convert columns into numeric types:

```
sep_df_double <- uni_df %>%
  separate(flipper_over_weight, into=c("flipper", "weight"), sep="/", convert = TRUE)
print(sep_df_double)
```

```
## # A tibble: 3 × 4
##   species    bill flipper weight
##   <fct>     <dbl>   <dbl>  <dbl>
## 1 Adelie     38.8    190   3701.
## 2 Chinstrap  48.8    196.  3733.
## 3 Gentoo     47.5    217.  5076
```

Uinte

Unite: paste together multiple columns into one.

We can also use the unite function to combine columns:

```
print(penguins_summary)
```

```
## # A tibble: 3 × 4
##   species    bill flipper weight
##   <fct>     <dbl>   <dbl>  <dbl>
## 1 Adelie     38.8    190   3701.
## 2 Chinstrap  48.8    196.  3733.
## 3 Gentoo     47.5    217.  5076
```

```
uni_df <- penguins_summary %>%
  unite(flipper_over_weight, flipper, weight, sep="/")
print(uni_df)
```

```
## # A tibble: 3 × 3
##   species    bill flipper_over_weight
##   <fct>     <dbl> <chr>
## 1 Adelie     38.8 190/3700.7
## 2 Chinstrap  48.8 195.8/3733.1
## 3 Gentoo     47.5 217.2/5076
```

Now the original columns *flipper* and *weight* are removed, and a column *flipper_over_weight* is created.

3. Nesting and unnesting

Nest: pack the data of each individual group into a table (data frame)

Unnest: flatten the tables back into regular columns (undo the nest operation)

```
## # A tibble: 4 × 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
## 4 Keith <NA>    guitar
```

Nest →

← Unnest

```
## # A tibble: 4 × 2
## # Groups:   name [4]
##   name  data
##   <chr> <list>
## 1 Mick  <tibble [1 × 2]>
## 2 John  <tibble [1 × 2]>
## 3 Paul  <tibble [1 × 2]>
## 4 Keith <tibble [1 × 2]>
```

Note: a tibble is a special type of data frame in R

Nesting

Nesting can be done in R using the nest function

Suppose that we have a data frame called musicians:

```
print(musicians)
```

```
## # A tibble: 4 × 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
## 4 Keith <NA>    guitar
```

```
musicians_nest <- musicians %>%
  group_by(name) %>%
  nest()
print(musicians_nest)
```

```
## # A tibble: 4 × 2
## # Groups:   name [4]
##   name  data
##   <chr> <list>
## 1 Mick  <tibble [1 × 2]>
## 2 John  <tibble [1 × 2]>
## 3 Paul  <tibble [1 × 2]>
## 4 Keith <tibble [1 × 2]>
```

Note: to use nest() we need to first group the data frame using group_by()

A list of tibbles is created in the column data (corresponds to the individual groups), which is called a list-column

Unnesting

Unnest: flatten the tables back into regular columns (undo the nest operation)

```
print(musicians_nest)
```

```
## # A tibble: 4 × 2
## # Groups:   name [4]
##   name  data
##   <chr> <list>
## 1 Mick  <tibble [1 × 2]>
## 2 John  <tibble [1 × 2]>
## 3 Paul  <tibble [1 × 2]>
## 4 Keith <tibble [1 × 2]>
```

To unnest a data frame, we use the unnest() function

```
musicians_nest %>%
  unnest(cols = data)
```

```
## # A tibble: 4 × 3
## # Groups:   name [4]
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
## 4 Keith <NA>    guitar
```

4. Iteration based on the map function

Iterations are key elements in many programming languages

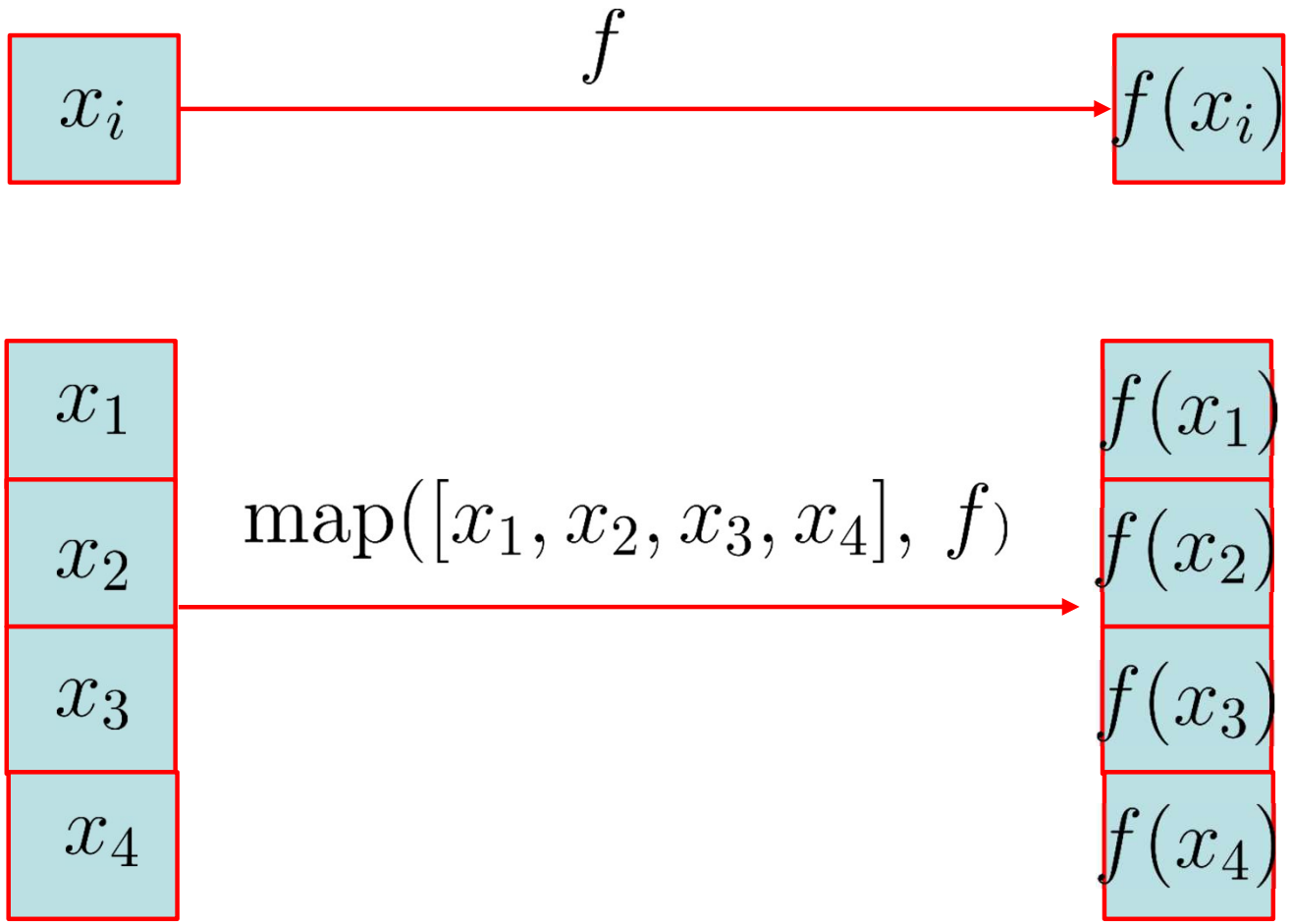
The standard approach to this is through loops e.g. for, while etc.

In R, we can also use vectorized operations and the map function for iterations

- Map based approaches are typically more readable.

The map function in R

In R, the map function transforms its input by applying a function to each element of a list or atomic vector and returning an object of the same length as the input.



4. The map function in R – example

Let's understand the map function through an example:

Code:

```
is_div_2_3 <- function(x){
  if (x%%2==0 | x%%3==0){
    return (TRUE)
  } else {
    return (FALSE)
  }
}
v <- c(1,2,3,5,6)
map(v, is_div_2_3)
```

Output:

```
## [[1]]
## [1] FALSE
##
## [[2]]
## [1] TRUE
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] FALSE
##
## [[5]]
## [1] TRUE
```

= is_div_2_3(v[1])

= is_div_2_3(v[2])

= is_div_2_3(v[3])

= is_div_2_3(v[4])

= is_div_2_3(v[5])

In this example, the map function applies the is_div_2_3 to each individual element of v and returns a list that combines the outputs of the functions.

4. Iteration based on the map function

The function map returns a list.

There are several variants of the map function that return a vector of a specific type, such as

- map_lgl() returns booleans,
- map_int() return integers,
- map_dbl() returns double
- map_chr() return strings

type ?map_int etc to see more

5. Example: Finding variables of maximal correlation

We want to create a function that

1. Takes as input a data frame and a variable name (column name)
2. Computes the correlation with all other numeric variables (columns)
3. Returns the name of the variable with maximal absolute correlation, and the corresponding correlation.

Recall that the correlation between vectors x and y is defined as (Pearson formula):

$$\frac{\sum_1^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2}\sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

In R, the correlation between two vectors can be computed using the function cor (type ?cor for more details)

5. Example: Finding variables of maximal correlation

We will start with a script for a specific case where

- the dataset is the penguins dataset
- the variable (column) name is ‘body_mass_g’.

```
col_name <- 'body_mass_g'
df = penguins

v_col <- select(df, all_of(col_name)) # extract column based on col_name
df_num <- select_if(df, is.numeric) %>% select(-all_of(col_name)) # select all numeric column
s excluding col_name

cor_func <- function(x){ cor(x, v_col, use='complete.obs') } # a function that computes cor b
etween v_col and a vector
correlations <- unlist(map(df_num, cor_func)) # compute correlations with all other numeric c
olumns (with map)
print('the computed correlations are:'); print(correlations)

max_abs_cor_var <- names( which( abs(correlations)==max(abs(correlations))  ) ) # extract the
name of the column with max correlation
cor_val <- as.double(correlations[max_abs_cor_var])
print('\ncolumn with maximal correlation:' ); print(max_abs_cor_var)
```

```
## [1] "the compute correlations are:"
##    bill_length_mm     bill_depth_mm flipper_length_mm              year
##        0.59510982       -0.47191562        0.87120177        0.04220939
## [1] "\ncolumn with maximal correlation:"
## [1] "flipper_length_mm"
```

5. Example: Finding variables of maximal correlation

Then we will define a function called *max_cor_var*; The body of this function is copied from the previous script:

```
max_cor_var <- function(df, col_name){

  v_col <- select(df, all_of(col_name)) # extract column based on col_name
  df_num <- select_if(df, is.numeric) %>% select(-all_of(col_name)) # select all numeric colu
mns excluding col_name

  cor_func <- function(x){ cor(x, v_col, use='complete.obs') } # a function that computes cor
between v_col and a vector
  correlations <- unlist(map(df_num, cor_func)) # compute correlations with all other numeric
columns (with map)

  max_abs_cor_var <- names( which( abs(correlations)==max(abs(correlations))  ) ) # extract t
he name of the column with max correlation
  cor_val <- as.double(correlations[max_abs_cor_var])

  return (data.frame(var_name=max_abs_cor_var, cor=cor_val)) # return as a data frame
}

max_cor_var(penguins, "body_mass_g")
```

```
##            var_name       cor
## 1 flipper_length_mm 0.8712018
```

The variable name and the maximal correlation are returned!

Maximal correlation within each group

The *max_cor_var* defined above can be also applied to each individual group of the data frame

The idea is to use nest() and unnest() to transform the groups into tables, then the apply *max_cor_var to each of the tables*

```
cor_by_group <- penguins %>%
  group_by(species) %>%
  nest() %>%
  mutate(max_cor=map(data, function(x){max_cor_var(x, 'body_mass_g')}))

print(cor_by_group)
```

```
## # A tibble: 3 × 3
## # Groups:   species [3]
##   species   data               max_cor
##   <fct>     <list>             <list>
## 1 Adelie    <tibble [152 × 7]> <df [1 × 2]>
## 2 Gentoo    <tibble [124 × 7]> <df [1 × 2]>
## 3 Chinstrap <tibble [68 × 7]>  <df [1 × 2]>
```

Note that here we have used the map function to apply *max_cor_var* to each group

6. Massing data

Missing data is remarkably common in practical data science applications

Consider for example the following data frame called stocks

```
stocks <- tibble(
  year   = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
  qtr    = c(   1,    2,    3,    4,    2,    3,    4),
  return = c(1.88, 0.59, 0.35,   NA, 0.92, 0.17, 2.66)
)
print(stocks)
```

```
## # A tibble: 7 × 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2015     4  NA
## 5  2016     2   0.92
## 6  2016     3   0.17
## 7  2016     4   2.66
```

The NA value here represents a missing value

Two types of missing data

Missing data can appear either explicitly or implicitly

1. Explicit missing data: The value of an individual variable is replaced with "NA" (not available), e.g., the NA value on the right

2. Implicit missing data: The entire row is missing, e.g., the row corresponding to the first quarter of 2016 is missing

```
## # A tibble: 7 × 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2015     4  NA
## 5  2016     2   0.92
## 6  2016     3   0.17
## 7  2016     4   2.66
```

There are different ways of dealing with the missing data

Making missing data explicit

To make the implicit missing data explicit, we can insert rows that include NA values

We can use the complete function

```
complete(stocks, year, qtr)
```

```
## # A tibble: 8 × 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2015     4  NA
## 5  2016     1  NA
## 6  2016     2   0.92
## 7  2016     3   0.17
## 8  2016     4   2.66
```

The complete() function takes a set of columns and finds all unique combinations. In this example, all unique combinations between (2015,2016) and (1,2,3,4)

A row associated with the **first quarter of 2016** is added, where the return value is NA

Complete case analysis

We can find the row with missing values using the function complete.cases, which returns a logical vector indicating which cases are complete

```
complete.cases(stocks)
```

```
## [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
```

Note: the 4th row is not complete because it contains a NA value

```
## # A tibble: 7 × 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2015     4  NA
## 5  2016     2   0.92
## 6  2016     3   0.17
## 7  2016     4   2.66
```

Removing the incomplete cases

With the complete case analysis, we can remove the incomplete cases using the filter function

```
filter(stocks, complete.cases(stocks))
```

```
## # A tibble: 6 × 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2016     2   0.92
## 5  2016     3   0.17
## 6  2016     4   2.66
```

The 4th row of the original data frame is removed.

Replacing the missing values

In some cases, we may want to replace the missing values with some numbers, instead of deleting them

For example, we can replace a missing value with the mean of its associated column

Let's first define a function called *replace_by_mean* to replace the NA value in a vector

```
replace_by_mean <- function(x){
  mu <- mean(x, na.rm=TRUE) # first compute the mean of x

  impute_f <- function(z){ # imputation on a single element z
    if (is.na(z)){
      return (mu)
    } else {
      return (z)
    }
  }
  return (map_dbl(x, impute_f)) # apply the function to impute across the whole vector x
}

x <- c(1,2,NA,4)
replace_by_mean(x)
```

```
## [1] 1.000000 2.000000 2.333333 4.000000
```

Here the 3rd element of x is replaced with the mean value (1+2+4)/3 = 2.33333

Replacing the missing values

Then we apply the function *replace_by_mean* to the column called return

```
mutate(stocks, return=replace_by_mean(return))
```

```
## # A tibble: 7 × 3
##    year   qtr return
##   <dbl> <dbl>  <dbl>
## 1  2015     1   1.88
## 2  2015     2   0.59
## 3  2015     3   0.35
## 4  2015     4   1.10
## 5  2016     2   0.92
## 6  2016     3   0.17
## 7  2016     4   2.66
```

new

The NA value on the 4th row of the original data frame is replaced.

Note that here we have used the *mutate* function to create a new column called returned

What we have covered

We introduced the concept of tidy data.

We saw how to reshape data with the pivot functions.

We looked at the unite and separate functions and the nest and unnest functions.

We investigated the map function for iteration within the *tidyverse* in R.

We also looked at some basic methods for handling missing data.

Try the examples yourself?

The illustration, codes, and examples are included in the R Markdown file **LectureTidyData.Rmd** which can be downloaded via the course webpage.

Thanks for listening!

Dr. Rihuan Ke

rihuan.ke@bristol.ac.uk

Statistical Computing and Empirical Methods
Unit EMATM0061, MSc Data Science