Data Management in R

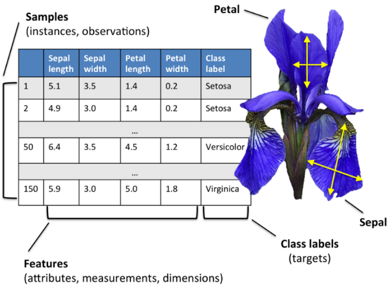
importing, cleaning, renaming, labelling, recoding, sorting, joining (merging, appending)

Henry Athiany

2022-04-13

Gentle introduction: Iris dataset

- The **Iris** flower dataset or Fisher's **Iris data** was initially introduced by **Ronald Fisher** in his **1936** paper: *The use of multiple measurements in taxonomic problems*.
- The dataset contains three plant species (setosa, virginica, versicolor), and four features measured for each of the 50 samples. In total, we have 150 observaions in the dataset.
- It is one of the most famous multivariate dataset used for data mining, classification, clustering among others (see below).


Petal
Samples
(instances, observations)
Sepal
Class labels
(targets)
Features
(attributes, measurements, dimensions)

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica

Data cleaning

Data cleaning

- An important process in data analysis
- It involves transforming inaccurate raw data into reliable data useful for analysis
- Improves data quality and overall reliability
- Multiple R packages available for this task, e.g `dyplr`
- In data cleaning, one needs to have this in mind:
  - Be familiar with the dataset to be cleaned (domain knowledge: **is it about sheep? malaria? datatype?**)
  - Be aware of structural errors (**mislabeled variables, faulty data types, non-unique IDs, string inconsistencies**)
  - Be aware of data irregularities (**invalid values - no logical sense, outliers**)
  - Think of how to deal with missing values: (**no single best way of dealing with this, but. . .**)
  - Keep record of cleaning procedures: (**Good research is reproducible**)

Data cleaning

We intent to use the **Iris dataset** to show all these steps involved in data cleaning

Poorly prepared data gives unreliable result - ***Garbage in, garbage out***

Load the iris dataset

- Previously, we learnt on how we could import various sources of data into R (Excel, Stata, SAS e.t.c)
- Base R has a package known as `datasets`. It has the Iris dataset already
- To view this, we use the code

```
# Load the datasets package to access the data
library(datasets)
data(iris)
#lets view the head of the iris dataset
head(iris, n=3)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Spec
## 1          5.1         3.5          1.4         0.2  set
## 2          4.9         3.0          1.4         0.2  set
## 3          4.7         3.2          1.3         0.2  set
```

Load the iris dataset

- Notice the first three(3) observations of the dataset, and the variable names.
- Next, let's open and view the `csv` file first in Ms Excel
- Now, let's import the `csv` file that will be used for the training.
- Name the imported dataset as `iris.imported`, to avoid mixing names with the inbuilt dataset

```
library(readr)
iris.imported <- read_csv("iris.csv", col_names = FALSE)
```

```
## Rows: 150 Columns: 5
## -- Column specification ---------------------------------
## Delimiter: ","
## chr (1): X1
## dbl (4): X2, X3, X4, X5
##
## i Use `spec()` to retrieve the full column specificatio
## i Specify the column types or set `show_col_types = FALS
```

Cleaning data

- View the imported dataset (iris2)

```
iris2<-iris.imported
#keep original dataset intact, but create new datasets for
head(iris2, n=3)
```

```
## # A tibble: 3 x 5
##   X1          X2    X3    X4    X5
##   <chr>    <dbl> <dbl> <dbl> <dbl>
## 1 Setosa     5.1   3.5   1.4   0.2
## 2 Setosa     4.9   3     1.4   0.2
## 3 Setosa     4.7   3.2   1.3   0.2
```

- We note that the variable names are X1,. . . ,X5 and order of the variables changed
- Also imported as **`A tibble: 6 x 5`**

Renaming

Renaming variables

```r
# get the current variables' names
names(iris2)
```

```
## [1] "X1" "X2" "X3" "X4" "X5"
```

```r
#Name of variable number 4
names(iris2)[4]
```

```
## [1] "X4"
```

```r
#Rename the variables
names(iris2)[names(iris2) == "X1"] <- "Species"
names(iris2)[names(iris2) == "X2"] <- "Sepal.Length"
names(iris2)[names(iris2) == "X3"] <- "Sepal.Width"
names(iris2)[names(iris2) == "X4"] <- "Petal.Length"
names(iris2)[names(iris2) == "X5"] <- "Petal.Width"
names(iris2)
```

```
## [1] "Species"      "Sepal.Length" "Sepal.Width"  "Petal.
```

Renaming (single) variables

- Alternatively;

```
library(reshape)
iris3<-iris.imported
#Rename the variables
rename(iris3, c(X1 = "Species")) #rename one variable
```

```
## # A tibble: 150 x 5
##    Species    X2    X3    X4    X5
##    <chr>   <dbl> <dbl> <dbl> <dbl>
##  1 Setosa    5.1   3.5   1.4   0.2
##  2 Setosa    4.9   3     1.4   0.2
##  3 Setosa    4.7   3.2   1.3   0.2
##  4 Setosa    4.6   3.1   1.5   0.2
##  5 Setosa    5     3.6   1.4   0.2
##  6 Setosa    5.4   3.9   1.7   0.4
##  7 Setosa    4.6   3.4   1.4   0.3
##  8 Setosa    5     3.4   1.5   0.2
##  9 Setosa    4.4   2.9   1.4   0.2
```

Renaming (several) variables

- Alternatively;

```
rename(iris3, c(X2= "Sepal.Length", X3= "Sepal.Width", X4=
```

```
## # A tibble: 150 x 5
##    X1     Sepal.Length Sepal.Width Petal.Length Petal.Wi
##    <chr>         <dbl>       <dbl>        <dbl>        <c
##  1 Setosa          5.1         3.5          1.4
##  2 Setosa          4.9         3            1.4
##  3 Setosa          4.7         3.2          1.3
##  4 Setosa          4.6         3.1          1.5
##  5 Setosa          5           3.6          1.4
##  6 Setosa          5.4         3.9          1.7
##  7 Setosa          4.6         3.4          1.4
##  8 Setosa          5           3.4          1.5
##  9 Setosa          4.4         2.9          1.4
## 10 Setosa          4.9         3.1          1.5
## # ... with 140 more rows
```

Renaming variables

```
names(iris3)
```

```
## [1] "X1" "X2" "X3" "X4" "X5"
```

```
head(iris3)
```

```
## # A tibble: 6 x 5
##   X1         X2    X3    X4    X5
##   <chr>  <dbl> <dbl> <dbl> <dbl>
## 1 Setosa   5.1   3.5   1.4   0.2
## 2 Setosa   4.9   3     1.4   0.2
## 3 Setosa   4.7   3.2   1.3   0.2
## 4 Setosa   4.6   3.1   1.5   0.2
## 5 Setosa   5     3.6   1.4   0.2
## 6 Setosa   5.4   3.9   1.7   0.4
```

- **Ex:** try typing the command `fix(iris3)` at the command prompt and rename one of the variables. What's the danger of this approach?

Removing and adding variables

```
iris4<-iris.imported
names(iris4)
```

```
## [1] "X1" "X2" "X3" "X4" "X5"
```

```
names(iris4) <- NULL
names(iris4)
```

```
## NULL
```

```
names(iris4) <- c("Species", "Sepal.Length", "Sepal.Width",
head(iris4)
```

```
## # A tibble: 6 x 5
##   Species Sepal.Length Sepal.Width Petal.Length Petal.Wi
##   <chr>          <dbl>       <dbl>        <dbl>       <
## 1 Setosa           5.1         3.5          1.4
## 2 Setosa           4.9         3            1.4
## 3 Setosa           4.7         3.2          1.3
## 4 Setosa           4.6         3.1          1.5
```

Deleting columns, rows and data values

Deleting column(s) by name

Method I :

- The most easiest way to drop columns is by using `subset()` function

```
iris2<-iris.imported
names(iris2)
```

```
## [1] "X1" "X2" "X3" "X4" "X5"
```

```
iris2_new = subset(iris2, select = -c(X4,X5))
names(iris2_new)
```

```
## [1] "X1" "X2" "X3"
```

Delete column(s) by name

Method II

```
iris3<-iris.imported
names(iris3)
```

```
## [1] "X1" "X2" "X3" "X4" "X5"
```

```
iris3_new = iris3[,!(names(iris3) %in% c("X2","X3"))]
iris3_new
```

```
## # A tibble: 150 x 3
##    X1          X4    X5
##    <chr>   <dbl> <dbl>
##  1 Setosa    1.4   0.2
##  2 Setosa    1.4   0.2
##  3 Setosa    1.3   0.2
##  4 Setosa    1.5   0.2
##  5 Setosa    1.4   0.2
##  6 Setosa    1.7   0.4
##  7 Setosa    1.4   0.3
```

Drop columns by column index numbers

```
#drop variables in position 2-5
names(iris4)
```

```
## [1] "Species"      "Sepal.Length" "Sepal.Width"  "Petal.
```

```
iris4_new <- iris4[ -c(2:5) ]
names(iris4_new)
```

```
## [1] "Species"
```

```
#Only one variable is left
```

- To this end, we have three new datasets
  - iris2_new (3 variables)
  - iris3_new (3 variables)
  - iris4_new (1 variable)

Exercises (drop or keep)

1. Use the following sytax to keep the names

- Keep column by name: `iris4_new = iris4[c("X1","X2")]`
- Keep columns by column index number: `iris4_new <- iris4[c(2,4)]`

2. Practice on how to Keep or Delete columns with `dplyr` package `library(dplyr)`

- then use the syntax: `mydata2 = select(mydata, -1, -3:-4)`

Keep or delete columns with `dplyr` package

```r
iris5<-iris.imported
# delete first, third and fourth column
iris5.1 = select(iris5, -1, -3:-4)
# delete named columns
iris5.2 = select(iris5, -X1, -X2, -X3)
# or
iris5.3 = select(iris5, -c(X1, X2, X3))
#or
iris5.4 = select(iris5, -X1:-X3)
# keep named columns
iris5.5 = select(iris5, X1, X3:X4)
```

- All the four dataframes `iris5.1 to iris5.4` should be the same. Check!

Keep / drop columns by `name pattern`

```
# Keeping columns whose name starts with "S"
iris4.s = iris4[,grepl("^S",names(iris4))]
names(iris4.s)
```

```
## [1] "Species"      "Sepal.Length" "Sepal.Width"
```

- The `grepl()` function is used to search for matches to a pattern. In this case `Se`.

```
# Dropping columns whose name begin with the letter "S"
iris4.s1 = iris4[,!grepl("*S",names(iris4))]
names(iris4.s1)
```

```
## [1] "Petal.Length" "Petal.Width"
```

Data structure

Structure of the dataset

```
# Getting to know the stucture of the imported dataset
iris.str<-iris.imported
names(iris.str) <- c("Species", "Sepal.Length", "Sepal.Widt
str(iris.str)
```

```
## spec_tbl_df [150 x 5] (S3: spec_tbl_df/tbl_df/tbl/data.f
##  $ Species     : chr [1:150] "Setosa" "Setosa" "Setosa"
##  $ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5
##  $ Sepal.Width : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3
##  $ Petal.Length: num [1:150] 1.4 1.4 1.3 1.5 1.4 1.7 1.4
##  $ Petal.Width : num [1:150] 0.2 0.2 0.2 0.2 0.2 0.4 0.3
##  - attr(*, "spec")=
##   .. cols(
##   ..   X1 = col_character(),
##   ..   X2 = col_double(),
##   ..   X3 = col_double(),
##   ..   X4 = col_double(),
##   ..   X5 = col_double()
##   .. )
```

Structure of the dataset

- Notice that Species has been imported as character
- We need to change it to 'Factor

```
iris.str<-iris.imported
# getting the class of the vector
class(iris.str$X1)
```

```
## [1] "character"
```

```
iris.str$X1<-as.factor(iris.str$X1)
class(iris.str$X1)
```

```
## [1] "factor"
```

```
str(iris.str)
```

```
## spec_tbl_df [150 x 5] (S3: spec_tbl_df/tbl_df/tbl/data.
##  $ X1: Factor w/ 3 levels "Setosa","Versicolor",..: 1 1
##  $ X2: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9
##  $ X3: num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.
##  $ X4: num [1:150] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4
```

Subsetting (vectors, matrix, lists and dataframes)

Subsetting

- We have several operators that one can use to extract subsets of R objects
- For instance;
    - [returns an object of the same class as the original; can be used to select more than one element
    - [[used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame
    - $ used to extract elements of a list or data frame by name

```r
x <- c("a", "b", "c", "c", "d", "a")
x[1];x[2] ;x[1:4]
```

```
## [1] "a"
```

```
## [1] "b"
```

```
## [1] "a" "b" "c" "c"
```

```r
x[x > "a"]
```

```
## [1] "b" "c" "c" "d"
```

Subsetting a matrix

- To subset a matrix, you can use the usual way of indices (i,j)

```r
x <- matrix(1:9, 3, 3) #default, column wise
x[1, 2] # try x[1, 2, drop = FALSE]
```

```
## [1] 4
```

```r
x[2, 1]
```

```
## [1] 2
```

Indices can also be missing.

```r
x[1, ] #try x[1, , drop = FALSE]
```

```
## [1] 1 4 7
```

```r
x[, 2]
```

```
## [1] 4 5 6
```

Subsetting cases from a dataframe

Now, let's use the Iris data to show the tricks

- First get a summary of the variable of interest
- This will also help in knowing the factor names

```
iris.str<-iris.imported
names(iris.str) <- c("Species", "Sepal.Length", "Sepal.Widt
iris.str$Species<-as.factor(iris.str$Species)
summary(iris.str$Species)
```

```
##     Setosa Versicolor  Virginica
##         50         50         50
```

Subsetting cases from a dataframe

```
names(iris.str) <- c("Species", "Sepal.Length", "Sepal.Widt
# Get first few rows of each subset
iris.str$Species<-as.factor(iris.str$Species)
subset(iris.str, Species == "Setosa")[1:2,]
```

```
## # A tibble: 2 x 5
##   Species Sepal.Length Sepal.Width Petal.Length Petal.Wi
##   <fct>          <dbl>       <dbl>        <dbl>       <c
## 1 Setosa           5.1         3.5          1.4
## 2 Setosa           4.9         3            1.4
```

```
subset(iris.str, Species == "Versicolor")[1:2,]
```

```
## # A tibble: 2 x 5
##   Species    Sepal.Length Sepal.Width Petal.Length Petal
##   <fct>             <dbl>       <dbl>        <dbl>
## 1 Versicolor          7           3.2          4.7
## 2 Versicolor          6.4         3.2          4.5
```

```
#subset(iris.str, Species == "Virginica")[1:2,]
```

Subsetting cases from a dataframe

- Let's do the same for the inbuilt `iris` dataset

```
subset(iris, Species == "setosa")[1:2,]
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Spec
## 1          5.1         3.5          1.4         0.2  set
## 2          4.9         3.0          1.4         0.2  set
```

```
subset(iris, Species == "versicolor")[1:2,]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width
## 51          7.0         3.2          4.7         1.4 ver
## 52          6.4         3.2          4.5         1.5 ver
```

```
subset(iris, Species == "virginica")[1:2,]
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width
## 101          6.3         3.3          6.0         2.5 vi
## 102          5.8         2.7          5.1         1.9 vi
```

```
# run this then try again: iris <- as_tibble(iris)
```

Variable labels

- A variable label could be specified for any vector using `var_label()` .

```
library(labelled)
var_label(iris.str$Sepal.Length) <- "Length of sepal"
```

- It's also possible to add a variable label to several columns of a data frame using a named list.

```
#var_label(iris) <- list(Petal.Length = "Length of petal",
var_label(iris.str) <- list(Petal.Length = "Length of peta
var_label(iris$Petal.Width)
```

```
## NULL
```

Variable labels

```
var_label(iris.str)
```

```
## $Species
## NULL
##
## $Sepal.Length
## [1] "Length of sepal"
##
## $Sepal.Width
## NULL
##
## $Petal.Length
## [1] "Length of petal"
##
## $Petal.Width
## [1] "Width of Petal"
```

- in RStudio, use `View(iris.str)` to display the variable labels in the data viewer

Variable labels

- To remove a variable label, use NULL.

```
var_label(iris.str$Sepal.Length) <- NULL
```

- To display and search through variable names and labels with `look_for()`

```
look_for(iris.str)
```

```
##  pos variable     label           col_type values
##  1   Species      -               fct      Setosa
##                                            Versicolor
##                                            Virginica
##  2   Sepal.Length -               dbl
##  3   Sepal.Width  -               dbl
##  4   Petal.Length Length of petal dbl
##  5   Petal.Width  Width of Petal  dbl
```

Variable labels

```r
look_for(iris.str, "Set")
```

```
##  pos variable label col_type values
##  1   Species  -     fct      Setosa
##                              Versicolor
##                              Virginica
```

```r
look_for(iris.str, details = FALSE)
```

```
##  pos variable     label
##  1   Species      -
##  2   Sepal.Length -
##  3   Sepal.Width  -
##  4   Petal.Length Length of petal
##  5   Petal.Width  Width of Petal
```

Deduplicates

Deduplicates

Find and drop duplicate elements

```
#Given the following vector:
x <- c(1, 1, 4, 5, 4, 6, 6, 2,3,4)
#To find the position of duplicate elements in x, use this
duplicated(x)
```

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

```
#Extract duplicate elements:
x[duplicated(x)]
```

```
## [1] 1 4 6 4
```

```
# To remove duplicated elements
x[!duplicated(x)]
```

```
## [1] 1 4 5 6 2 3
```

Deduplicates: dataframe

- check for duplicates, we can use the base R function `duplicated()`, which will return a logical vector telling us which rows are duplicate rows

```
suppressPackageStartupMessages(library(tidyverse))
iris.dup<-iris %>% slice(1:3)
iris.dup = subset(iris.dup, select = -c(Sepal.Length,Sepal
iris.dup
```

```
##   Petal.Length Petal.Width Species
## 1          1.4         0.2  setosa
## 2          1.4         0.2  setosa
## 3          1.3         0.2  setosa
```

```
iris.dup %>% distinct()
```

```
##   Petal.Length Petal.Width Species
## 1          1.4         0.2  setosa
## 2          1.3         0.2  setosa
```

Duplicates: dataframe (removing duplicates)

```
suppressPackageStartupMessages(library(tidyverse))
iris.dup<-iris %>% slice(1:3)
iris.dup = subset(iris.dup, select = -c(Sepal.Length,Sepal.
iris.dup
```

```
##   Petal.Length Petal.Width Species
## 1          1.4         0.2  setosa
## 2          1.4         0.2  setosa
## 3          1.3         0.2  setosa
```

```
#with duplicated() function
iris.dup <- iris.dup[!duplicated(iris.dup$Petal.Length), ]
iris.dup
```

```
##   Petal.Length Petal.Width Species
## 1          1.4         0.2  setosa
## 3          1.3         0.2  setosa
```

```
#with distinct() function
iris.dup <- iris.dup %>% distinct(Petal.Length, .keep_all =
```

Extract unique elements

```
#Given the following vector:
x <- c(1, 1, 4, 5, 4, 6)
# You can extract unique elements as follow:
unique(x)
```

```
## [1] 1 4 5 6
```

```
# It's also possible to apply unique() on a data frame, +
#for removing duplicated rows as follow:
suppressPackageStartupMessages(library(tidyverse))
iris.dup<-iris %>% slice(1:3)
iris.dup = subset(iris.dup, select = -c(Sepal.Length,Sepal
unique(iris.dup)
```

```
##   Petal.Length Petal.Width Species
## 1          1.4         0.2  setosa
## 3          1.3         0.2  setosa
```

Recoding

Recoding a categorical variable to another categorical variable

- We want to recode the factor variable Species. `setosa` will be recoded as `set`, `virginica` to `virg` and `versicolor` to `versi`
- This can be done with the recode() function from the dplyr package:

```
suppressPackageStartupMessages(library(dplyr))
iris.recod = subset(iris, select = -c(Sepal.Length,Sepal.Wi
#Getting to know the levels before recoding
levels(iris.str$Species)
```

```
## [1] "Setosa"     "Versicolor" "Virginica"
```

```
iris.recod<-recode(iris.recod$Species, setosa = "set", virg
#levels after recoding
levels(iris.recod)
```

```
## [1] "set"   "versi" "virg"
```

Categorize numeric data with `cut()` function

```
library(data.table)
iris.str$iris.group <- cut(iris.str$Petal.Length, breaks =
labels = c("small", "medium", "large"), include.lowest=TRU
```

- If a data value falls outside of the specified bounds, it's categorized as `NA`.
- The result of `cut()` is a factor, and you can see from the example that the factor levels are named after the bounds: `str(iris.str$iris.group)`
- What happens when we omit the part `include.lowest=TRUE`?

Categorize numeric data with `cut()` function

```
iris.str$iris.group <- cut(iris.str$Petal.Length, breaks =
labels = c("small", "medium", "large"), include.lowest=TRU
#Drop few variables to view full list
iris.str= iris.str[,!grepl("*Se",names(iris.str))]
head(iris.str, n=1)
```

```
## # A tibble: 1 x 4
##   Species Petal.Length Petal.Width iris.group
##   <fct>          <dbl>       <dbl> <fct>
## 1 Setosa           1.4         0.2 small
```

- You can also categorise using the following functions: `discretize()`, `group_var()` and `frq()` amongst others

Let's take a 10 min break

Sorting

Sorting data

- R has a function called **`sort`** that is used to sort data in either ascending or descending order,
- The variable by which you sort can be a **numeric, string or factor** variable,
- We also have some options on how missing values are handled: they can be listed **first, last or removed**,
- We use our `iris.str` dataset to show this.

Sorting data

```
iris.str[1:5, ] #please note the arguments in the square br
```

```
## # A tibble: 5 x 4
##   Species Petal.Length Petal.Width iris.group
##   <fct>          <dbl>       <dbl> <fct>
## 1 Setosa           1.4         0.2 small
## 2 Setosa           1.4         0.2 small
## 3 Setosa           1.3         0.2 small
## 4 Setosa           1.5         0.2 small
## 5 Setosa           1.4         0.2 small
```

```
sort(iris.str$Petal.Width)
```

```
##   [1] 0.1 0.1 0.1 0.1 0.1 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.
##  [19] 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.
##  [37] 0.3 0.3 0.3 0.3 0.3 0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.
##  [55] 1.0 1.0 1.0 1.1 1.1 1.1 1.2 1.2 1.2 1.2 1.2 1.3 1.
##  [73] 1.3 1.3 1.3 1.3 1.3 1.3 1.4 1.4 1.4 1.4 1.4 1.4 1.
##  [91] 1.5 1.5 1.5 1.5 1.5 1.5 1.5 1.5 1.6 1.6 1.6 1.6 1.
```

Sorting data

```
iris.str[1:5, ]
```

```
## # A tibble: 5 x 4
##   Species Petal.Length Petal.Width iris.group
##   <fct>          <dbl>       <dbl> <fct>
## 1 Setosa           1.4         0.2 small
## 2 Setosa           1.4         0.2 small
## 3 Setosa           1.3         0.2 small
## 4 Setosa           1.5         0.2 small
## 5 Setosa           1.4         0.2 small
```

- Data still the same as above, not what we anticipated
- Variable `Petal.Length` sorted independent of the dataframe
- In most applications, we use the **order** function and not the **sort** function to sort data in a data frame

Sorting data

```
sorted.iris.str <- iris.str[order(iris.str$Petal.Width) , ]
sorted.iris.str[1:5, ]
```

```
## # A tibble: 5 x 4
##   Species Petal.Length Petal.Width iris.group
##   <fct>          <dbl>       <dbl> <fct>
## 1 Setosa           1.5         0.1 small
## 2 Setosa           1.4         0.1 small
## 3 Setosa           1.1         0.1 small
## 4 Setosa           1.5         0.1 small
## 5 Setosa           1.4         0.1 small
```

Sorting data

- We can also sort the data frame by more than one variable, say `Species` then `Petal.Length`

```
#sorted.iris.str <- iris.str[order(iris.str$Species,iris.s
sorted.iris.str <- iris.str[order(iris.str$Species,iris.st
sorted.iris.str[1:5, ]
```

```
## # A tibble: 5 x 4
##   Species Petal.Length Petal.Width iris.group
##   <fct>          <dbl>       <dbl> <fct>
## 1 Setosa           1           0.2 small
## 2 Setosa           1.1         0.1 small
## 3 Setosa           1.2         0.2 small
## 4 Setosa           1.2         0.2 small
## 5 Setosa           1.3         0.2 small
```

Sorting data: descending

- We can also sort in the reverse order by using the (-) before the variable

```
sorted.iris.str <- iris.str[order(iris.str$Species,-(iris.s
sorted.iris.str[1:5, ]
```

```
## # A tibble: 5 x 4
##   Species Petal.Length Petal.Width iris.group
##   <fct>          <dbl>       <dbl> <fct>
## 1 Setosa           1.9         0.2 small
## 2 Setosa           1.9         0.4 small
## 3 Setosa           1.7         0.4 small
## 4 Setosa           1.7         0.3 small
## 5 Setosa           1.7         0.2 small
```

- How do we handle **ties** in the dataset?

Sorting data: descending

- What is different in this case?

```
sorted.iris.str <- iris.str[order(iris.str$Petal.Length,-(i
sorted.iris.str[5:10, ]
```

```
## # A tibble: 6 x 4
##   Species Petal.Length Petal.Width iris.group
##   <fct>          <dbl>       <dbl> <fct>
## 1 Setosa           1.3         0.4 small
## 2 Setosa           1.3         0.3 small
## 3 Setosa           1.3         0.3 small
## 4 Setosa           1.3         0.2 small
## 5 Setosa           1.3         0.2 small
## 6 Setosa           1.3         0.2 small
```

- Sorted by first variable but because of ties, second variable sorted in descending order

Missing values

Missing values

Missing values are denoted by NA or NaN for undefined mathematical operations.

- `is.na()` is used to test objects if they are NA
- `is.nan()` is used to test for NaN
- `NA` values have a class also, so there are integer NA, character NA, etc.
- A `NaN` value is also NA but the converse is not true

Missing values

```r
x <- c(1, 2, NA, 10, 3)
is.na(x)
```

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

```r
is.nan(x)
```

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

```r
x <- c(1, 2, NaN, NA, 4)
is.na(x)
```

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

```r
is.nan(x)
```

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

- lets get back to our dataset

Handling missing data: NA

- Note that missing data are denoted as NA in R, regardless of the type of variable (even in numeric variables)
- Since we don't have missing observations, let's create some for this exercise using the variable Petal.Width

```
iris.str$Petal.Width[2:5] <- NA
iris.str[1:5, ]
```

```
## # A tibble: 5 x 4
##   Species Petal.Length Petal.Width iris.group
##   <fct>          <dbl>       <dbl> <fct>
## 1 Setosa           1.4         0.2 small
## 2 Setosa           1.4        NA   small
## 3 Setosa           1.3        NA   small
## 4 Setosa           1.5        NA   small
## 5 Setosa           1.4        NA   small
```

Sorting data with missing values

- We can sort the data frame such that the missing data are at the top, the bottom, or deleted from the frame

```
iris.str$Petal.Width[2:5] <- NA
iris.str <- iris.str[order(iris.str$Petal.Width, na.last=FA
head(iris.str)
```

```
## # A tibble: 6 x 4
##   Species Petal.Length Petal.Width iris.group
##   <fct>          <dbl>       <dbl> <fct>
## 1 Setosa           1.4        NA   small
## 2 Setosa           1.3        NA   small
## 3 Setosa           1.5        NA   small
## 4 Setosa           1.4        NA   small
## 5 Setosa           1.5         0.1 small
## 6 Setosa           1.4         0.1 small
```

Sorting data with missing values: descending

- We can sort the data frame such that the missing data are at the top, the bottom, or deleted from the frame

```
iris.str$Petal.Width[2:5] <- NA
iris.str <- iris.str[order(iris.str$Petal.Width, na.last=TR
tail(iris.str)
```

```
## # A tibble: 6 x 4
##   Species   Petal.Length Petal.Width iris.group
##   <fct>            <dbl>       <dbl> <fct>
## 1 Virginica          5.7         2.5 medium
## 2 Setosa             1.4        NA   small
## 3 Setosa             1.3        NA   small
## 4 Setosa             1.5        NA   small
## 5 Setosa             1.4        NA   small
## 6 Setosa             1.5        NA   small
```

Handling missing values

- Let's create a new variable called `id` and then give missing values

```
iris.str$id <- 1:nrow(iris.str)
iris.str$id[2:145] <- NA
iris.str <- iris.str[order(iris.str$id, na.last=NA) , ]
iris.str
```

```
## # A tibble: 6 x 5
##   Species Petal.Length Petal.Width iris.group    id
##   <fct>          <dbl>       <dbl> <fct>      <int>
## 1 Setosa           1.4         0.1 small          1
## 2 Setosa           1.4        NA   small        146
## 3 Setosa           1.3        NA   small        147
## 4 Setosa           1.5        NA   small        148
## 5 Setosa           1.4        NA   small        149
## 6 Setosa           1.5        NA   small        150
```

Dealing with dates in R

Creating date/times

- There are three types of date/time data that refer to an instant in time:
- A **date**. Tibbles print this as `<date>`.
- A **time** within a day. Tibbles print this as `<time>`.
- A **date-time** is a date plus a time: it uniquely identifies an instant in time (typically to the nearest second). Tibbles print this as `<dttm>`. Elsewhere in R these are called POSIXctt.
- Dates are represented as the number of days since 1970-01-01, with negative values for earlier dates.

Knowing the current date/time

```r
#lubridate is a package in R meant to deal with dates
suppressPackageStartupMessages(library(lubridate))
Sys.Date() #get current date
```

```
## [1] "2022-04-13"
```

```r
today()  #get current date
```

```
## [1] "2022-04-13"
```

```r
Sys.time() #get current time
```

```
## [1] "2022-04-13 18:23:29 EAT"
```

```r
now() #get current time
```

```
## [1] "2022-04-13 18:23:29 EAT"
```

```r
leap_year(2018) # check whether 2018 is a leap year
```

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

Create and format dates

- To create a `Date` object from a simple character string in R, you can use the `as.Date()` function.
- The character string has to obey a format that can be defined using a set of symbols (the examples correspond to 11 January, 1992):
  - `%Y`: 4-digit year (1992)
  - `%y`: 2-digit year (92)
  - `%m`: 2-digit month (01)
  - `%d`: 2-digit day of the month (11)
  - `%A`: weekday (Wednesday)
  - `%a`: abbreviated weekday (Wed)
  - `%B`: month (January)
  - `%b`: abbreviated month (Jan)

Create and format dates

Lets create the same Date object for the 11th day in January of 1992:

```r
as.Date("1992-01-11")
```

```
## [1] "1992-01-11"
```

```r
as.Date("Jan-11-92", format = "%b-%d-%y")
```

```
## [1] "1992-01-11"
```

```r
as.Date("1 January, 1992", format = "%d %B, %Y")
```

```
## [1] "1992-01-01"
```

Dates formats

```
library(readr)
dates <- read_csv("dates.csv")
dates
```

```
## # A tibble: 7 x 4
##   mdyy    dmyy    day_time      yymd
##   <chr>   <chr>   <chr>         <date>
## 1 7/6/21  6/7/21  6/7/21 03:00  2021-07-06
## 2 7/7/21  7/7/21  7/7/21 03:00  2021-07-07
## 3 7/8/21  8/7/21  8/7/21 03:00  2021-07-08
## 4 7/9/21  9/7/21  9/7/21 03:00  2021-07-09
## 5 7/10/21 10/7/21 10/7/21 03:00 2021-07-10
## 6 7/11/21 11/7/21 11/7/21 03:00 2021-07-11
## 7 7/12/21 12/7/21 12/7/21 03:00 2021-07-12
```

```
#format(dates, format="%B %d %Y")
f_col <- as.Date(dates$mdyy, format="%m/%d/%y")
f_col
```

```
## [1] "2021-07-06" "2021-07-07" "2021-07-08" "2021-07-09"
```

Date arithmetic

- We can do some arithmetic with the dates as follows

```
course_start    <- as.Date('2017-04-12')
course_end      <- as.Date('2017-04-21')
course_duration <- course_end - course_start
course_duration
```

```
## Time difference of 9 days
```

- Suppose course start date has been moved forward by 2 days and the start date given 3 more days, what would be the course duration?

```
course_start    <- as.Date('2017-04-12')
course_end      <- as.Date('2017-04-21')
course_duration <- (course_end+3) - (course_start-2)
course_duration
```

```
## Time difference of 14 days
```

lubtridate

- Getting R to agree that your data contains the dates and times you think it does can be tricky
- Package in R dealing specifically with times
- Has in built functions for dealing with dates
- Expands the type of mathematical operations that can be performed with date-time objects
- Easy and fast parsing of date-times: `ymd()`, `ymd_hms`, `dmy()`, `dmy_hms`, `mdy()`

```
ymd(20201215)
```

```
## [1] "2020-12-15"
```

```
mdy("4/1/17")
```

```
## [1] "2017-04-01"
```

lubtridate

- Simple functions to get and set components of a date-time, such as `year()`, `month()`, `mday()`, `hour()`, `minute()` and `second()`:

```
bday <- dmy("14/10/2019")
month(bday)
```

```
## [1] 10
```

```
wday(bday, label = TRUE)
```

```
## [1] Mon
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

```
year(bday) <- 2016
wday(bday, label = TRUE)
```

```
## [1] Fri
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

lubridate

- It introduces three new time span classes.
- **durations**, which measure the exact amount of time between two points
- **periods**, which accurately track clock times despite leap years, leap seconds, and day light savings time
- **intervals**, a protean summary of the time information between two points

Example: Time intervals

- Consider the followig

```
arrive <- ymd_hms("2022-06-04 12:00:00", tz = "Pacific/Auck
#arrive
leave <- ymd_hms("2022-08-10 14:00:00", tz = "Pacific/Auckl
#leave
```

- Suppose I plan to travel for a conference in Auckland, and my stay will be from June 4, 2022 to August 10, 2022

```
auckland <- interval(arrive, leave)
auckland
```

```
## [1] 2022-06-04 12:00:00 NZST--2022-08-10 14:00:00 NZST
```

```
auckland <- arrive %--% leave
auckland
```

```
## [1] 2022-06-04 12:00:00 NZST--2022-08-10 14:00:00 NZST
```

Example: Time intervals

- My collaborator at the University of Auckland, Darren, will also be travelling to various conferences this year including the International Biometric Conference (IBC).
- This will take him out of the country from July 20 until the end of August.
- Will my visit overlap with his travels?

```
ibc <- interval(ymd(20220720, tz = "Pacific/Auckland"), ymc
ibc
```

```
## [1] 2022-07-20 NZST--2022-08-31 NZST
```

```
int_overlaps(ibc, auckland)
```

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

- Yes it will overlap

Example: Time intervals

- What part of my visit will he be available?

```
setdiff(auckland, ibc)
```

```
## [1] 2022-06-04 12:00:00 NZST--2022-07-20 NZST
```

- Other functions that work with intervals include `int_start`, `int_end`, `int_flip`, `int_shift`, `int_aligns`, `union`, `intersect`, `setdiff`, and `%within%`.

joining(merging, appending)

"merge" dataframes

- In most cases, all of your data might not come from a single source. You might have to merge the data after you load it into R.
- For instance, we previously created three different datasets named: `iris2_new, iris3_new and iris4_new`
- Suppose we want to join the datasets, what would we do? merge or append?

Merging dataframes

- There are many ways to combine multiple dataframes, from the `rbind` function to left outer join to logical vector combinations.
- However, to ensure that every key column and variable from your multiple datasets are combined correctly, there are three main techniques to consider;
    - `cbind()`: combining the columns of two data frames side-by-side
    - `rbind()`: stacking two data frames on top of each other, appending one to the other (appening)
    - `merge()`: joining two data frames using a common column

Using cbind() to merge two R data frames

- lets consuder the two datasets previously created
- We can see the first few observations of the sub-set dataset previously created

```
head(iris2_new,2)
```

```
## # A tibble: 2 x 3
##   X1          X2    X3
##   <chr>  <dbl> <dbl>
## 1 Setosa   5.1   3.5
## 2 Setosa   4.9   3
```

```
head(iris3_new,2)
```

```
## # A tibble: 2 x 3
##   X1          X4    X5
##   <chr>  <dbl> <dbl>
## 1 Setosa   1.4   0.2
## 2 Setosa   1.4   0.2
```

Using cbind() to merge two R data frames

```r
# Using cbind(), we can merge the two as follows
merged.cbind<-cbind(iris2_new, iris3_new)
head(merged.cbind,2)
```

```
##       X1  X2  X3     X1  X4  X5
## 1 Setosa 5.1 3.5 Setosa 1.4 0.2
## 2 Setosa 4.9 3.0 Setosa 1.4 0.2
```

Using rbind() to merge two R data frames

```
#setosa rows
seto <- iris2_new[iris2_new$X1 == "Setosa",]
#random subsample of non-setosa rows of size n("Setosa")
nonseto <- iris2_new[sample(which(iris2_new$X1 != "Setosa")
merged.rbind <- rbind(seto, nonseto)
head(merged.rbind, 1)
```

```
## # A tibble: 1 x 3
##   X1         X2    X3
##   <chr>  <dbl> <dbl>
## 1 Setosa   5.1   3.5
```

```
#tail(merged.rbind, 1)
```

- That was simple, right ?
- Well, the `rbind()` function works well when the structure of the data sets is exactly the same – same set of columns.

merge dataframes in R

- Note that the main method of the R `merge` function is for `data frames`.
- However, merge is a generic function that can be also used with other objects (like vectors or matrices), but they will be coerced to data.frame class.
- To illustrate this, we will consider two dataframes named;
  - **df_1**, that represents the **id, name and monthly salary** of some employees of a company and
  - **df_2**, that shows the **id, name, age and position** of some employees

merge dataframes

```
set.seed(01)
employee_id <- 1:10
employee_name <- c("Andrew", "Susan", "John", "Joe", "Jack'
                   "Jacob", "Mary", "Kate", "Jacqueline", '
employee_salary <- round(rnorm(10, mean = 1500, sd = 200))
employee_age <- round(rnorm(10, mean = 50, sd = 8))
employee_position <- c("CTO", "CFO", "Administrative", rep(
df_1 <- data.frame(id = employee_id[1:8], name = employee_n
                   month_salary = employee_salary[1:8])
df_2 <- data.frame(id = employee_id[-5], name = employee_na
                   age = employee_age[-5], position = emplo
head(df_1,2); head(df_2,2)
```

```
##   id   name month_salary
## 1  1 Andrew         1375
## 2  2  Susan         1537
```

```
##   id   name age position
## 1  1 Andrew  62      CTO
```

merge dataframes

- In reality, all ids will be unique but the names can be repeated.
- We usualy have either an:
  - Inner join (intersection)- most usual join of data sets
  - Full (outer) join(Union)
  - Left (outer) join in R - matching main dataset
  - Right (outer) join in R - matching `using` dataset
  - Cross join - one to all other data points on either side

merge (inner join)

```r
# merges by the common column names
merge(x = df_1, y = df_2)
```

```
##   id   name month_salary age       position
## 1  1 Andrew         1375  62            CTO
## 2  2  Susan         1537  53            CFO
## 3  3   John         1333  45 Administrative
## 4  4    Joe         1819  32     Technician
## 5  6  Jacob         1336  50     Technician
## 6  7   Mary         1597  50     Technician
## 7  8   Kate         1648  58     Technician
```

```r
merge(x = df_1, y = df_2, by = c("id", "name")) # Equivale
```

```
##   id   name month_salary age       position
## 1  1 Andrew         1375  62            CTO
## 2  2  Susan         1537  53            CFO
## 3  3   John         1333  45 Administrative
## 4  4    Joe         1819  32     Technician
```

merge (full (outer) join)

```
# merges all the columns of both data sets into one
merge(x = df_1, y = df_2, all = TRUE)
```

```
##    id       name month_salary age       position
## 1   1     Andrew         1375  62            CTO
## 2   2      Susan         1537  53            CFO
## 3   3       John         1333  45 Administrative
## 4   4        Joe         1819  32     Technician
## 5   5       Jack         1566  NA           <NA>
## 6   6      Jacob         1336  50     Technician
## 7   7       Mary         1597  50     Technician
## 8   8       Kate         1648  58     Technician
## 9   9 Jacqueline           NA  57     Technician
## 10 10        Ivy           NA  55     Technician
```

```
# to create a full outer join of the two data frames
# in R you have to set the argument all to TRUE
```

merge (left (outer) join)

```r
# to create the join, set all.x = TRUE
merge(x = df_1, y = df_2, all.x = TRUE)
```

```
##   id   name month_salary age       position
## 1  1 Andrew         1375  62            CTO
## 2  2  Susan         1537  53            CFO
## 3  3   John         1333  45 Administrative
## 4  4    Joe         1819  32     Technician
## 5  5   Jack         1566  NA           <NA>
## 6  6  Jacob         1336  50     Technician
## 7  7   Mary         1597  50     Technician
## 8  8   Kate         1648  58     Technician
```

```r
# What would you say about employee of id=5?
```

merge (right (outer), cross join and multiple dataframes)

```
# Right join
head(merge(x = df_1, y = df_2, all.y = TRUE),3)
```

```
##   id   name month_salary age       position
## 1  1 Andrew         1375  62            CTO
## 2  2  Susan         1537  53            CFO
## 3  3   John         1333  45 Administrative
```

```
#cross join
head(Merged <- merge(x = df_1, y = df_2, by = NULL),3)
```

```
##   id.x name.x month_salary id.y name.y age position
## 1    1 Andrew         1375    1 Andrew  62      CTO
## 2    2  Susan         1537    1 Andrew  62      CTO
## 3    3   John         1333    1 Andrew  62      CTO
```

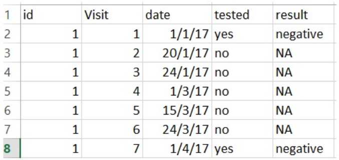
```
# merge several dataframes

#merge(x, merge(y, z, all = TRUE), all = TRUE)
```

Case study

Case study: Reshape, join, label, missing values

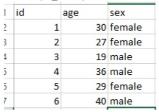
- **Purpose**: To explain how data can be appended, variables encoded and recoded,data converted from wide to long format and vice-versa
- There are three Ms Excel files named **reshape_example, reshape_example2 and reshap_example3** as shown below

reshape_example



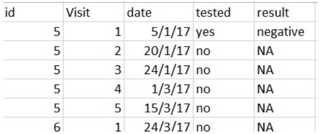
	id	Visit	date	tested	result
2	1	1	1/1/17	yes	negative
3	1	2	20/1/17	no	NA
4	1	3	24/1/17	no	NA
5	1	4	1/3/17	no	NA
6	1	5	15/3/17	no	NA
7	1	6	24/3/17	no	NA
8	1	7	1/4/17	yes	negative

reshape_example2



	id	age	sex
2	1	30	female
3	2	27	female
4	3	19	male
5	4	36	male
6	5	29	female
7	6	40	male

reshape_example3



id	Visit	date	tested	result
5	1	5/1/17	yes	negative
5	2	20/1/17	no	NA
5	3	24/1/17	no	NA
5	4	1/3/17	no	NA
5	5	15/3/17	no	NA
6	1	24/3/17	no	NA

Data import and date formating

```
#importing an excel file
library(readxl)
reshape_example <- read_excel("reshape_example.xlsx")
typeof(reshape_example$date)
```

```
## [1] "double"
```

```
#changing the date format to dd/mm/yyy
reshape_example$date <- format(as.Date(reshape_example$date
head(reshape_example, n=3)
```

```
## # A tibble: 3 x 5
##      id Visit date        tested result
##   <dbl> <dbl> <chr>       <chr>  <chr>
## 1     1     1 01-Jan-2017 yes    negative
## 2     1     2 20-Jan-2017 no     NA
## 3     1     3 24-Jan-2017 no     NA
```

```
save(reshape_example, file = "reshape_example.RData")
```

import the third dataset

```
#importing an excel file
library(readxl)
reshape_example3 <- read_excel("reshape_example3.xlsx")
typeof(reshape_example3$date)
```

```
## [1] "double"
```

```
reshape_example3$date <- format(as.Date(reshape_example3$da
save(reshape_example3, file = "reshape_example3.RData")
```

Append reshape_example3 to reshape_example1

- When it comes to appending data frames, the `rbind()` and `cbind()` function comes to mind because they can concatenate the data frames horizontally and vertically.

```
appended_reshape <- rbind(reshape_example, reshape_example3
head(appended_reshape)
```

```
## # A tibble: 6 x 5
##      id Visit date        tested result
##   <dbl> <dbl> <chr>       <chr>  <chr>
## 1     1     1 01-Jan-2017 yes    negative
## 2     1     2 20-Jan-2017 no     NA
## 3     1     3 24-Jan-2017 no     NA
## 4     1     4 01-Mar-2017 no     NA
## 5     1     5 15-Mar-2017 no     NA
## 6     1     6 24-Mar-2017 no     NA
```

recoding character variable to numeric

```
suppressPackageStartupMessages(library(dplyr))
appended_reshape<- appended_reshape%>% mutate(tested2 = tes
appended_reshape$tested <- recode(appended_reshape$tested,
appended_reshape$tested2 <- recode(appended_reshape$tested2
#table(appended_reshape$tested, appended_reshape$tested2)#
#drop the original column
appended_reshape$tested <- NULL
#rename the new variable back to tested
rename(appended_reshape, tested = tested2)
```

```
## # A tibble: 38 x 5
##       id Visit date        result   tested
##    <dbl> <dbl> <chr>       <chr>     <dbl>
##  1     1     1 01-Jan-2017 negative      1
##  2     1     2 20-Jan-2017 NA            0
##  3     1     3 24-Jan-2017 NA            0
##  4     1     4 01-Mar-2017 NA            0
##  5     1     5 15-Mar-2017 NA            0
```

recoding character variable to numeric

```
appended_reshape$result <- recode(appended_reshape$result,
```

```
## Warning: Unreplaced values treated as NA as `.x` is not
## Please specify replacements exhaustively or supply `.de
```

```
#Testing for Missing Values
is.na(appended_reshape$result)
```

```
##  [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FA
## [13]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  T
## [25]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  T
## [37]  TRUE FALSE
```

```
#appended_reshape$tested2 <- NULL
#appended_reshape$result <- NULL
```

Reshape the data set from Long to Wide format

- Within the reshape function, we have to specify the name of our data frame (i.e. data), the idvar argument (i.e. id), the timevar argument (i.e. Visit), and the direction (i.e. "wide")

```
suppressPackageStartupMessages(library(reshape2))
data_reshape1 <- reshape(data=appended_reshape,
                         idvar = "id",
                         v.names = c("date", "result", "tes
                         timevar = "Visit",
                         direction = "wide")
```

Reshape the data set from Long to Wide format

```
suppressPackageStartupMessages(library(dplyr))
#data_wide <- spread(appended_reshape, id, c("Visit", "dat

#data_wide <- spread(appended_reshape,  key=Visit, value=d

appended_reshape %>%  pivot_wider(names_from = c(Visit,date
```

```
## # A tibble: 6 x 29
##      id `1_01-Jan-2017_0` `2_20-Jan-2017_NA` `3_24-Jan-2
##   <dbl>             <dbl>              <dbl>
## 1     1                 1                  0
## 2     2                NA                 NA
## 3     3                NA                 NA
## 4     5                NA                  0
## 5     6                NA                 NA
## 6     7                NA                 NA
## # ... with 24 more variables: `5_15-Mar-2017_NA` <dbl>,
## #   `6_24-Mar-2017_NA` <dbl>, `7_01-Apr-2017_0` <dbl>,
```

Reshape the data set from Long to Wide format

```
reshape_example2 <- read_excel("reshape_example2.xlsx")
typeof(reshape_example2$id)
```

```
## [1] "double"
```

```
merged<-merge(appended_reshape, reshape_example2)
head(merged)
```

```
##   id Visit        date result tested2 age    sex
## 1  1     1 01-Jan-2017      0       1  30 female
## 2  1     5 15-Mar-2017     NA       0  30 female
## 3  1     2 20-Jan-2017     NA       0  30 female
## 4  1     3 24-Jan-2017     NA       0  30 female
## 5  1     4 01-Mar-2017     NA       0  30 female
## 6  1     8 20-Apr-2017     NA       0  30 female
```

```
#finally sort the data by id then visit
merged[order(merged$id,merged$Visit), ]
```

```
##    id Visit        date result tested2 age    sex
```

Thanks!