ST2195 Programming for Data Science
Block 6 Introduction to Data Wrangling

VLE Resources

There are extra resources for this Block on the VLE – including quizzes and videos please make sure you review them as part of your learning. You can find them here: https://emfss.elearning.london.ac.uk/course/view.php?id=382#section-6

Data Wrangling

Data wrangling (also known as data munging or data carpentry) refers to the design, implementation, and execution of processes that take us from raw, typically unstructured, data to data that are more appropriate for subsequent data-analytic processes. We have already seen an instance of data wrangling in "Structured, Semi-Structured and Unstructured Data," where we used the **tm** R package to convert unstructured text to a document-term matrix, which is a structured data format that can be used for topic modelling.

As with everything in data science, it is extremely important to:

- Think carefully and design what data wrangling processes are appropriate for what you need to do;
- Implement the processes in a way that is reproducible, reusable and shareable.

Investing time to implement and share the data wrangling processes using open-source tools (e.g. git), technologies (e.g. Markdown and R Markdown), and programming languages (like Python and R) will get you a long way in the above directions.

The article "Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions" by Benureau, F. C. Y and Rougier, N. P (Benureau & Rougier, 2018) is a highly-recommended read about reproducibility and reusability of code.

The main data wrangling activities involve at least a combination of the following:

- Discovering patterns in data: For example, identifying correlations and patterns through basic data analysis and visualizations
- Structuring data: For example, subsetting, merging, re-ordering, transforming, reshaping, etc
- Cleaning and validating data: For example, identifying missing or misrecorded data that can impact the accuracy of subsequent data-analytic processes
- Enriching data: For example, thinking of what other data sources should be used to maximize the value of the data

You have already learned how to go about most of those operations in previous weeks, and you will learn more in the upcoming weeks. It is important to know that during data wrangling activities most often ad-hoc decisions need to be taken! It is good practice to record what decisions have been taken, or even “parameterize” those decisions in order to come back to them, for example to test how they impact the outcome of an analysis.

For example, the data in `heart_rates.csv` is a runner’s heart rates as recorded by a smart watch during jogging.

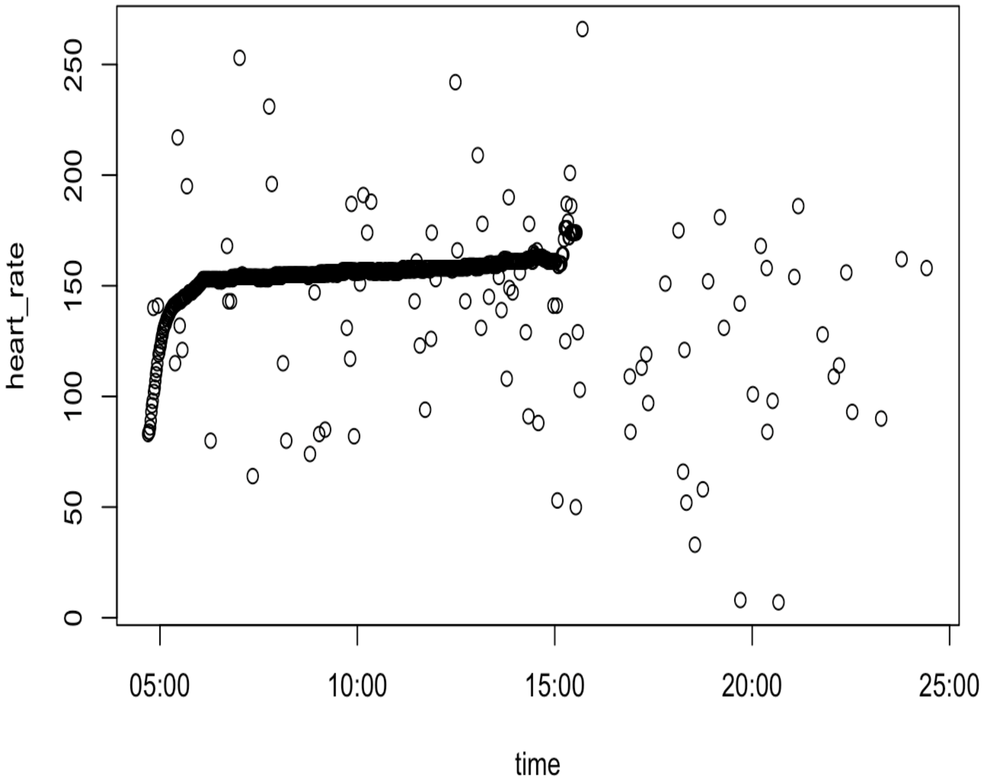
```
heart_rates <- read.csv("heart_rates.csv")
# convert times from character into POSIX (see ?as.POSIXct)
heart_rates$time <- as.POSIXct(heart_rates$time)
str(heart_rates)
'data.frame':   1191 obs. of  2 variables:
 $ time      : POSIXct, format: "2013-06-08 08:04:42" "2013-06-08 08:04:43"
...
 $ heart_rate: int  83 84 84 86 89 93 96 98 140 102 ...
head(heart_rates)
                 time heart_rate
1 2013-06-08 08:04:42         83
2 2013-06-08 08:04:43         84
3 2013-06-08 08:04:44         84
4 2013-06-08 08:04:45         86
5 2013-06-08 08:04:46         89
6 2013-06-08 08:04:47         93
```

From a quick look using `str()` and `head()` we see that `heart_rate` is a data frame with times and the corresponding heart rate recordings. However, a closer look reveals some issues

```
summary(heart_rates)
      time                       heart_rate
 Min.   :2013-06-08 08:04:42   Min.   :  7.0
 1st Qu.:2013-06-08 08:09:38   1st Qu.:153.0
 Median :2013-06-08 08:14:35   Median :156.0
 Mean   :2013-06-08 08:14:34   Mean   :151.5
 3rd Qu.:2013-06-08 08:19:31   3rd Qu.:158.0
 Max.   :2013-06-08 08:24:27   Max.   :266.0
                               NA's   :501
```

There are 501 missing heart rate values, which may be because the heart rate monitor could not take a measurement at that particular timestamp. A more serious issue, though, is that the minimum recorded heart rate is `r min(heart_rates$heart_rate, na.rm = TRUE)`, which seems a bit low for someone jogging, and the maximum recorded heart rate is `r` `max(heart_rates$heart_rate, na.rm = TRUE)`, which is a bit high for a human! A plot of the heart rates versus time reveals some issues.

```
plot(heart_rates)
```



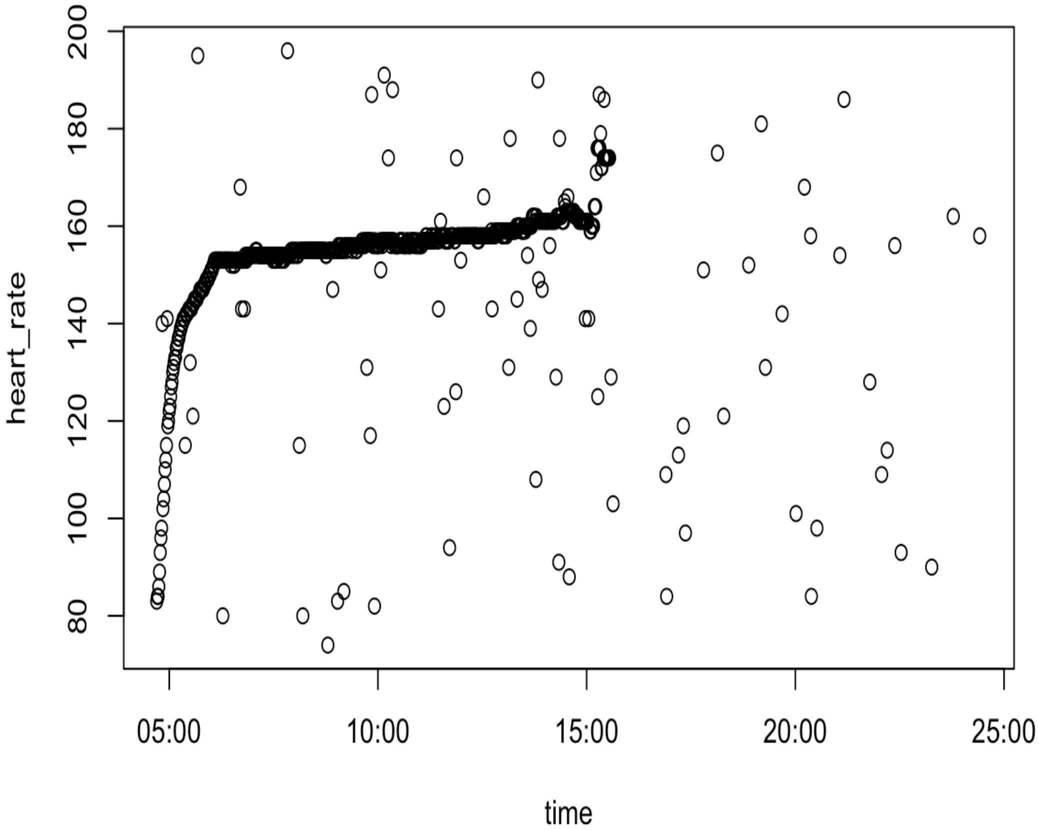
First, there seems to be a clear evolution of the heart rates for about 10 minutes from around 70 to 180, which seems normal. But the clear evolution stops abruptly after minute 15, possibly because the heart rate monitor fell off. Also there appears to be some noisy heart rate observations during the workout, which may point to an issue with the device.

Depending on what we want to do with this data, using it as is in statistical modelling may lead to misleading conclusions. We can alleviate some of those problems by writing a short function that removes "implausible" observations. For example, the below function replaces any heart rate outside the interval `(min_hr, max_hr)` with `NA`.

```
trim <- function(x, min_hr, max_hr) {
    within(heart_rates, {
        heart_rate[heart_rate < min_hr] <- NA
        heart_rate[heart_rate > max_hr] <- NA
    })
}
```

So, we can now `NA` heart rates that are less than 70 and above 200 as implausible, before passing the data to further analysis and modelling.

```
hr <- trim(heart_rates, min_hr = 70, max_hr = 200)
plot(hr)
```



In addition to that, after we complete the analysis and modelling, we can come back and adjust the values of `min_hr` and `max_hr` and repeat the analyses to see how our decisions regarding the data impact our conclusions!

In the following weeks we will discuss how missing values can be handled also as part of machine learning pipelines.

Useful Links and Resources

- [Wikipedia's data wrangling page]
- [Jenny Bryan's course on "Data wrangling, exploration, and analysis with R"]

References

Benureau, F. C. Y., & Rougier, N. P. (2018). Re-run, repeat, reproduce, reuse, replicate: Transforming code into scientific contributions. *Frontiers in Neuroinformatics*, *11*, 69.

Data Wrangling Operations in R

Introduction

In this page, we will focus on some R packages and functionality that is extremely useful during data wrangling processes. In particular, we will introduce:

- The `*apply` family of R functions that allow us to apply functions to specific dimensions of matrices and arrays, and to data frames and lists
- Reshaping data sets
- Stacking data sets
- Subsetting data sets
- The verbs of the **dplyr** R package

`*apply` Functions

The `*apply()` family is a collection of functions in R that is used to manipulate slices of data from matrices, arrays, lists and data frames in a repetitive way without explicitly writing loops. These functions apply a function to each element of the selected input data. Below, we focus on the following `*apply()` functions:

- `apply()`: Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.
- `lapply()`: Returns a list of the same length as the selected input data, each element of which is the result of applying the chosen function to the corresponding element of the selected input data.
- `sapply()`: A more user-friendly version of `lapply()`. Returns a vector or matrix by default.
- `mapply()`: This is the multivariate version of `sapply()`. It applies a function to multiple lists or vector arguments.

Please check the help pages of each of these functions for a complete description of their arguments.

`apply()`

The `apply()` function takes a data frame or matrix as input and returns an output of vector, list or array type. The `apply()` function is mainly used to avoid explicit uses of loops.

```
# Create a simple matrix
my_matrix <- matrix(c(1:20), nrow = 5)
my_matrix
     [,1] [,2] [,3] [,4]
```

```
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
# Using apply() to calculate the sum of each row
apply(X = my_matrix, MARGIN = 1, FUN = sum)
[1] 34 38 42 46 50
# Using apply() to calculate the sum of each column
apply(X = my_matrix, MARGIN = 2, FUN = sum)
[1] 15 40 65 90
```

In the above code chunk, we use the `apply()` function to calculate the sum of each row of the matrix by setting `MARGIN = 1`, calculate the sum of each column of the matrix by setting `MARGIN = 2`.

In both examples, we applied the `sum()` function to the margins. However, we could have used any other function, even user-defined ones, as long as they can work with the inputs specified by the margin we use. For example, if we want to get basic summaries of each of the columns we can do

```
apply(my_matrix, 2, summary)
        [,1] [,2] [,3] [,4]
Min.       1    6   11   16
1st Qu.    2    7   12   17
Median     3    8   13   18
Mean       3    8   13   18
3rd Qu.    4    9   14   19
Max.       5   10   15   20
```

As we would expect from the `summary()` function applied to a vector, the third column of the output above has

```
min(my_matrix[, 3])
[1] 11
quantile(my_matrix[, 3], 0.25)
25%
 12
median(my_matrix[, 3])
[1] 13
mean(my_matrix[, 3])
[1] 13
quantile(my_matrix[, 3], 0.75)
75%
 14
max(my_matrix[, 3])
[1] 15
```

lapply()

The function `lapply()` takes a vector or list `X` and applies the function `FUN` to each of its elements. Then, `lapply()` will output a list which is of the same length as `X`, where each element is the outcome of applying the function `FUN` on the corresponding element of `X`.

```
First_name <- c("John", "Jane", "Tim", "Michael", "Emma")
Last_name <- c("Doe", "Smith", "Williams", "Taylor", "Wilson")
```

```
# Create a list from the names
Name_list <- list(First_name, Last_name)
Name_list
[[1]]
[1] "John"    "Jane"    "Tim"     "Michael" "Emma"

[[2]]
[1] "Doe"      "Smith"    "Williams" "Taylor"   "Wilson"
# Convert to lower case
Names_lower <- lapply(X = Name_list, FUN = tolower)
Names_lower
[[1]]
[1] "john"    "jane"    "tim"     "michael" "emma"

[[2]]
[1] "doe"      "smith"    "williams" "taylor"   "wilson"
str(Names_lower)
List of 2
 $ : chr [1:5] "john" "jane" "tim" "michael" ...
 $ : chr [1:5] "doe" "smith" "williams" "taylor" ...
```

In the above code chunk, we worked with a list of strings containing the first and last name of five people. We used `lapply()` together with the `tolower()` function to convert the names to lower case.

`sapply()`

The function `sapply()` works in the same way as `lapply()` but simplifies (hence the `s`) the output to return a vector or a matrix.

```
# Create a simple function to raise the input to a chosen power
raise_power <- function(x, power) {
    x^power
}
# Examples
raise_power(x = 2, power = 3)
[1] 8
raise_power(x = 4, power = 2)
[1] 16
raise_power(x = 5, power = 4)
[1] 625
# Create a simple list containing numbers
numbers <- list(1:5, 6:10)
numbers
[[1]]
[1] 1 2 3 4 5

[[2]]
[1]  6  7  8  9 10
# Using the sapply() function to raise each element of the numbers vector
to the power of 3
sapply(X = numbers, FUN = raise_power, power = 3)
     [,1] [,2]
[1,]    1  216
[2,]    8  343
[3,]   27  512
[4,]   64  729
[5,]  125 1000
```

In the above example, we created a function called `raise_power()` that takes the arguments `x` and `power` and returns `x` raised to `power`.

We then created a list called `numbers`, which contains two vectors, and used `sapply()` to apply the `raise_power()` function to each element of `numbers`. We set `power = 3` just after we have specified `X = numbers` and `FUN = raise_power`. The output is a \(5 \times 2\) matrix, where each column represents the elements of the first and second vectors, respectively, raised to the power of three.

mapply()

The function `mapply()` is the multivariate version of `sapply()`. It applies a function in parallel over a set of arguments.

```
# Short demo of rep()
rep(1, 3)
[1] 1 1 1
rep(2, 3)
[1] 2 2 2
rep(3, 3)
[1] 3 3 3
# Create a 3x3 matrix
matrix(c(rep(1, 3), rep(2, 3), rep(3, 3)), 3, 3)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
# Do the same thing via mapply()
mapply(rep, 1:3, 3)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
[3,]    1    2    3
```

In this example, we created a matrix `mat`, using the `rep()` function thrice. We then create the same matrix `mat1` by using `mapply()`. Basically, `mapply()` applies the `rep()` function to a vector containing the numbers 1 to 3 with additional argument 3 specifying how many times each number should be repeated. Another example is

```
mapply(rep, 1:3, (1:3)^2)
[[1]]
[1] 1

[[2]]
[1] 2 2 2 2

[[3]]
[1] 3 3 3 3 3 3 3 3 3
```

sweep() and **aggregate()**

The `sweep()` and `aggregate()` functions are closely related to the `apply()` family. See their help files and the useful links and resources for more information.

Reshaping Data Sets

R provides a variety of methods for reshaping your data before analysis. Base R provides the `reshape()` function which, as the help file states reshapes a data frame between “wide” format with repeated measurements in separate columns of the same record, and “long” format with the repeated measurements in separate records. The **reshape2** R package provides similar functionality but with a simplified interface. To install the **reshape2** package, type in `install.packages("reshape2").`

The **reshape2** package makes it easy to transform data between wide and long formats:

- Wide-format data has a column for each variable
- Long-format data has a column for possible variable types and a column for the values for those variables. However, long-format data isn’t necessarily only two columns

As you may have already figured out, you may need wide-format data for some type of analyses and long-format data for others. For example, it is easy to work with **ggplot2** (we will discuss **ggplot2** graphics) with long-format data. Also, most statistical modelling functions like `lm()`, `glm()` and `gam()` work well with long-format data. However, many people often find wide-format data easier and more intuitive for recording data. Therefore, it is important to be able to work with both and be able to transform data from and into each of these two formats.

reshape2 has two key functions:

- `melt()`: Convert wide- to long-format data
- `*cast()`: Convert long- to wide-format data

Wide to Long

To illustrate how to use the `melt()` function in the **reshape2** package, we will consider the built-in R dataset `airquality`. This data set consists of daily air quality measurements in New York, from May to September 1973. This data is in wide format because for each data point, we have a column for each variable. In **reshape2**, to convert wide-format data to long-format data, we use the `melt()` function:

```
library("reshape2")
# print out the first few records of the airquality data frame
head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
# use the melt function to convert the data into long format
airquality_long <- melt(airquality)
No id variables; using all as measure variables
# print out the first few records of the new airquality_long data frame
```

```
head(airquality_long)
  variable value
1    Ozone    41
2    Ozone    36
3    Ozone    12
4    Ozone    18
5    Ozone    NA
6    Ozone    28
# print out the last few records of the new airquality_long data frame
tail(airquality_long)
    variable value
913      Day    25
914      Day    26
915      Day    27
916      Day    28
917      Day    29
918      Day    30
```

We see now our data is in long-format where we have one column named `variable` which records the name of the variable and a column named value which records the value of that variable. By default, the `melt()` function assumes that all columns are numeric with numeric values are variables with values. Often, we may want to know specific values and keep them as columns. We can identify these by using the `id.vars` argument. Here we can specify the ID variables, which are variables that identify individual rows of the data. Suppose we want to have the month and the day as ID variables:

```
# use the melt function to convert the data into long format with
specifying ID variables Month and Day
airquality_long <- melt(airquality, id.vars = c('Month', 'Day'))
head(airquality_long)
  Month Day variable value
1     5   1    Ozone    41
2     5   2    Ozone    36
3     5   3    Ozone    12
4     5   4    Ozone    18
5     5   5    Ozone    NA
6     5   6    Ozone    28
```

We can see that by passing in the `Month` and `Day` variables into the `id.vars` argument, we kept `Month` and `Day` as columns and the rest of the data was converted into long format. We may also wish to control the column names in our long format by passing in the column names for the variable and value into the `variable.name` and `value.name` arguments respectively:

```
# use the melt function to convert the data into long format with
specifying ID variables Month and Day
airquality_long <- melt(airquality, id.vars = c('Month', 'Day'),
                        variable.name = 'climate_var',
                        value.name = 'climate_value')
head(airquality_long)
  Month Day climate_var climate_value
1     5   1       Ozone            41
2     5   2       Ozone            36
3     5   3       Ozone            12
4     5   4       Ozone            18
5     5   5       Ozone            NA
6     5   6       Ozone            28
```

```
tail(airquality_long)
    Month Day climate_var climate_value
607     9  25        Temp            63
608     9  26        Temp            70
609     9  27        Temp            77
610     9  28        Temp            75
611     9  29        Temp            76
612     9  30        Temp            68
```

Long to Wide

Going from long to wide format can take a bit more thought than going from wide- to long-format data. In **reshape2**, there are multiple cast functions. To work with data frames, we use the `dcast()` function—there are also the `acast()` function to return a vector, matrix or array. To illustrate how to use the `dcast()` function, we work with the `airquality_long` data frame we created earlier.

The `dcast()` function uses a formula to describe the shape of the data. In this function we need to tell `dcast()` what are ID variables and what is the variable column that describes the measured variables. `dcast()` will use the last remaining column as the column that contains the values by default, but we can also pass this into the `value.var` argument. We can print this out and see that we can recover the original `airquality` data frame (but with the columns in different order):

```
# use the melt function to convert the data into long format with
specifying ID variables Month and Day
airquality_long <- melt(airquality, id.vars = c('Month', 'Day'),
                        variable.name = 'climate_var',
                        value.name = 'climate_value')
head(airquality_long)
  Month Day climate_var climate_value
1     5   1       Ozone            41
2     5   2       Ozone            36
3     5   3       Ozone            12
4     5   4       Ozone            18
5     5   5       Ozone            NA
6     5   6       Ozone            28
# use the dcast function to convert the data back into wide format
airquality_wide <- dcast(airquality_long,
                         formula = Month + Day ~ climate_var,
                         value.var = 'climate_value')
head(airquality_wide)
  Month Day Ozone Solar.R Wind Temp
1     5   1    41     190  7.4   67
2     5   2    36     118  8.0   72
3     5   3    12     149 12.6   74
4     5   4    18     313 11.5   62
5     5   5    NA      NA 14.3   56
6     5   6    28      NA 14.9   66
```

A common problem users may face is when you cast a data set where there is more than one value per data cell. For instance, consider only using `Month` as an ID variable:

```
# use the melt function to convert the data into long format with
specifying ID variables Month and Day
airquality_long <- melt(airquality, id.vars = c('Month', 'Day'),
```

```
                        variable.name = 'climate_var',
                        value.name = 'climate_value')
head(airquality_long)
  Month Day climate_var climate_value
1     5   1       Ozone            41
2     5   2       Ozone            36
3     5   3       Ozone            12
4     5   4       Ozone            18
5     5   5       Ozone            NA
6     5   6       Ozone            28
# use the dcast function to convert the data back into wide format
airquality_wide <- dcast(airquality_long,
                         formula = Month ~ climate_var,
                         value.var = 'climate_value')
Aggregation function missing: defaulting to length
head(airquality_wide)
  Month Ozone Solar.R Wind Temp
1     5    31      31   31   31
2     6    30      30   30   30
3     7    31      31   31   31
4     8    31      31   31   31
5     9    30      30   30   30
```

Here we got the warning: "Aggregation function missing: defaulting to length." By only using `Month` as the ID variable, we can see here that the cells are now filled with the number of data rows for each month. This is because we did not pass an aggregation function into the `fun.aggregate` argument which tells R how to aggregate the data.

If you cast your data and there are multiple values per cell, you will also need to tell `dcast()` how to aggregate the data. Depending on the aim of the analysis, we can use a range of scalar summaries, like the `mean()`, `median()`, `sum()`, etc. Here, we take the mean using the `mean()` function and we also use `na.rm = TRUE` to remove the NA values:

```
# use the melt function to convert the data into long format with
specifying ID variables Month and Day
airquality_long <- melt(airquality, id.vars = c('Month', 'Day'),
                        variable.name = 'climate_var',
                        value.name = 'climate_value')
head(airquality_long)
  Month Day climate_var climate_value
1     5   1       Ozone            41
2     5   2       Ozone            36
3     5   3       Ozone            12
4     5   4       Ozone            18
5     5   5       Ozone            NA
6     5   6       Ozone            28
# use the dcast function to convert the data back into wide format
airquality_wide <- dcast(airquality_long,
                         formula = Month ~ climate_var,
                         value.var = 'climate_value',
                         fun.aggregate = mean,
                         na.rm = TRUE)
head(airquality_wide)
  Month    Ozone  Solar.R      Wind     Temp
1     5 23.61538 181.2963 11.622581 65.54839
2     6 29.44444 190.1667 10.266667 79.10000
3     7 59.11538 216.4839  8.941935 83.90323
4     8 59.96154 171.8571  8.793548 83.96774
```

```
5      9 31.44828 167.4333 10.180000 76.90000
```

We see here that we now obtained the mean value of the variables for each month.

tidyr Package

There is also another popular package for data reshaping and more general data tidying, called **tidyr**. **tidyr** provides functionality for:

- Pivoting data (i.e. reshaping);
- Rectangling data (e.g. turning deeply nested lists into rectangular data);
- Nesting and unnesting (e.g. converting grouped data to a few where each group becomes a single row with a nested data frame, and the reverse);
- Splitting and combining character columns (e.g. pulling a single character column into multiple columns);
- Making implicit missing values explicit and vice versa;

Useful Links and Resources, below, provides some links of the basic functionality from **tidyr**.

Stacking

The `stack()` and `unstack()` functions in R are handy when working with data frames. Applying `stack()` to a data frame simply stacks the columns vectors on top of each other to form a single vector along with a factor indicating where each observation came from. As expected, the `unstack()` function does the complete opposite.

Below is a demonstration using the built-in `PlantGrowth` dataset. The dataset contains the weights of 30 plants from 3 different groups, namely `ctrl` (control), `trt1` (treatment 1) and `trt2` (treatment 2). Because of the way `stack()` works, we first convert the group variable from factor to character to avoid getting a warning message.

```
# Use the built-in PlantGrowth dataset
head(PlantGrowth)
  weight group
1   4.17  ctrl
2   5.58  ctrl
3   5.18  ctrl
4   6.11  ctrl
5   4.50  ctrl
6   4.61  ctrl
# Converting group variable from factor to character
PlantGrowth$group <- as.character(PlantGrowth$group)
# Stacking
stack(PlantGrowth)
   values    ind
1    4.17 weight
2    5.58 weight
3    5.18 weight
4    6.11 weight
5     4.5 weight
6    4.61 weight
```

```
7    5.17 weight
8    4.53 weight
9    5.33 weight
10   5.14 weight
11   4.81 weight
12   4.17 weight
13   4.41 weight
14   3.59 weight
15   5.87 weight
16   3.83 weight
17   6.03 weight
18   4.89 weight
19   4.32 weight
20   4.69 weight
21   6.31 weight
22   5.12 weight
23   5.54 weight
24    5.5 weight
25   5.37 weight
26   5.29 weight
27   4.92 weight
28   6.15 weight
29    5.8 weight
30   5.26 weight
31   ctrl  group
32   ctrl  group
33   ctrl  group
34   ctrl  group
35   ctrl  group
36   ctrl  group
37   ctrl  group
38   ctrl  group
39   ctrl  group
40   ctrl  group
41   trt1  group
42   trt1  group
43   trt1  group
44   trt1  group
45   trt1  group
46   trt1  group
47   trt1  group
48   trt1  group
49   trt1  group
50   trt1  group
51   trt2  group
52   trt2  group
53   trt2  group
54   trt2  group
55   trt2  group
56   trt2  group
57   trt2  group
58   trt2  group
59   trt2  group
60   trt2  group
# Unstacking
unstack(PlantGrowth)
   ctrl trt1 trt2
1  4.17 4.81 6.31
2  5.58 4.17 5.12
3  5.18 4.41 5.54
4  6.11 3.59 5.50
```

```
5  4.50 5.87 5.37
6  4.61 3.83 5.29
7  5.17 6.03 4.92
8  4.53 4.89 6.15
9  5.33 4.32 5.80
10 5.14 4.69 5.26
```

Check the help file of `stack()` for more options and examples.

Subsetting

R has some powerful indexing features that allows you to quickly access object elements. In this subsection, we will learn how to efficiently select or exclude particular elements in vectors or data frames.

Subsetting Vectors

Consider a simple numeric vector:

```
x <- c(1.4, 5.6, 7.8, 2.6)
```

The three ways in which we can subset elements of a vector in R are:

- Using positive integers: We use a vector of positive integers to specify the index of the elements we want to return / keep.
- Using negative integers: We use a vector of negative integers to specify the index of the elements we want to exclude.
- Using logical vectors: We use a vector of logical values where elements are selected if the corresponding logical value is `TRUE`.

```
# define x
x <- c(1.4, 5.6, 7.8, 2.6)
# select the 2nd and 4th element
x[c(2,4)]
[1] 5.6 2.6
# exclude the 2nd and 4th element
x[-c(2,4)]
[1] 1.4 7.8
# select the 2nd and 4th element
x[c(FALSE, TRUE, FALSE, TRUE)]
[1] 5.6 2.6
# select the elements that are strictly greater than 5
x[x > 5]
[1] 5.6 7.8
```

By running the above code, we can see that a very useful way to subset vectors is by using logical vectors. In the last line, we first evaluated the conditional statement `x > 5` and then we used the result to subset `x`.

Subsetting Data Frames

The most basic way of subsetting a data frame in R is using square brackets `[ ]`:

```
dataframe[x, y]
```

where `dataframe` is a data frame in R, `x` is the rows that we want to be returned, and `y` is a vector of the columns that we want returned.

```
# create dataframe
dataframe <- data.frame(v1 = 1:5, v2 = 6:10, v3 = 11:15)
# select the first 3 rows
dataframe[1:3, ]
  v1 v2 v3
1  1  6 11
2  2  7 12
3  3  8 13
# exclude the first 3 rows
dataframe[-(1:3),]
  v1 v2 v3
4  4  9 14
5  5 10 15
# select the first 3 rows and first 2 columns
dataframe[1:3, 1:2]
  v1 v2
1  1  6
2  2  7
3  3  8
# select the rows using a logical vector (condition: if the variable v3 is
divisible by 3
dataframe[dataframe$v3 %% 3 == 0,]
  v1 v2 v3
2  2  7 12
5  5 10 15
# select columns v1 and v3
dataframe[, c("v1", "v3")]
  v1 v3
1  1 11
2  2 12
3  3 13
4  4 14
5  5 15
# select columns v1 and v3 (alternative)
dataframe[c("v1", "v3")]
  v1 v3
1  1 11
2  2 12
3  3 13
4  4 14
5  5 15
```

As we have seen in “Data Structures in R,” we can also subset the rows of a data frame using the `subset()` function.

dplyr Verbs

One of the most useful packages to manipulate data is **dplyr**. This package contains the so-called **dplyr** verbs:

- `mutate()`: Adds new variables that are functions of existing variables
- `select()`: Picks variables based on their names.

- `filter()`: Picks cases based on their values.
- `summarize()`: Reduces multiple values down to a single summary.
- `arrange()`: Changes the ordering of the rows.

When working with **dplyr** verbs, we typically want to use the result of a verb applied to a data frame as input for another verb. In such cases, it is more intuitive to use the pipe `%>%` operator from the **magrittr** R package to chain operations. The **magrittr** package is loaded automatically when **dplyr** is loaded.

The variable to the left of `%>%` operator is passed as the first argument to the function on the right of the `%>%` operator.

We have already encountered some of the those verbs and functionality when working with databases in R. Here, we take a more in-depth view of **dplyr**'s functionality.

Throughout this section, we use the `mtcars` data set to illustrate the capabilities of the dplyr verbs. Here is a preview of the dataset and its structure:

```
head(mtcars)
                   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
str(mtcars)
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

`mutate()`

The variable `mpg` stands for miles per gallon. Suppose we want to add another variable called `kpg` which stands for kilometres per gallon. We know that 1 mile is approximately equal to 1.61 kilometres. We can use `mutate()` to add the variable `kpg`:

```
# Loading the dplyr package
library(dplyr)

Attaching package: 'dplyr'
The following objects are masked from 'package:stats':

    filter, lag
The following objects are masked from 'package:base':
```

```
    intersect, setdiff, setequal, union
# Adding kpg
mtcars1 <- mutate(mtcars, kpg = 1.61 * mpg)
head(mtcars1)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb    kpg
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4 33.810
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4 33.810
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1 36.708
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1 34.454
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2 30.107
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1 29.141
# Alternative approach using the pipe %>% operator
mtcars2 <- mtcars %>% mutate(kpg = 1.61 * mpg)
head(mtcars2)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb    kpg
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4 33.810
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4 33.810
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1 36.708
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1 34.454
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2 30.107
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1 29.141
# mtcars1 is identical to mtcars2
identical(mtcars1, mtcars2)
[1] TRUE
```

select()

Suppose we only want to select the `mpg`, `cyl`, `hp`, `vs` and `gear` columns. We can use the `select()` verb to do so.

```
mtcars %>%
    select(mpg, cyl, hp, vs, gear)
                     mpg cyl  hp vs gear
Mazda RX4           21.0   6 110  0    4
Mazda RX4 Wag       21.0   6 110  0    4
Datsun 710          22.8   4  93  1    4
Hornet 4 Drive      21.4   6 110  1    3
Hornet Sportabout   18.7   8 175  0    3
Valiant             18.1   6 105  1    3
Duster 360          14.3   8 245  0    3
Merc 240D           24.4   4  62  1    4
Merc 230            22.8   4  95  1    4
Merc 280            19.2   6 123  1    4
Merc 280C           17.8   6 123  1    4
Merc 450SE          16.4   8 180  0    3
Merc 450SL          17.3   8 180  0    3
Merc 450SLC         15.2   8 180  0    3
Cadillac Fleetwood  10.4   8 205  0    3
Lincoln Continental 10.4   8 215  0    3
Chrysler Imperial   14.7   8 230  0    3
Fiat 128            32.4   4  66  1    4
Honda Civic         30.4   4  52  1    4
Toyota Corolla      33.9   4  65  1    4
Toyota Corona       21.5   4  97  1    3
Dodge Challenger    15.5   8 150  0    3
AMC Javelin         15.2   8 150  0    3
Camaro Z28          13.3   8 245  0    3
Pontiac Firebird    19.2   8 175  0    3
Fiat X1-9           27.3   4  66  1    4
Porsche 914-2       26.0   4  91  0    5
```

```
Lotus Europa        30.4   4 113  1    5
Ford Pantera L      15.8   8 264  0    5
Ferrari Dino        19.7   6 175  0    5
Maserati Bora       15.0   8 335  0    5
Volvo 142E          21.4   4 109  1    4
```

`filter()`

We can use the `filter()` function to get only observations that match a condition (or multiple conditions). `filter()` is the **dplyr** equivalent to `subset()`, and allows us to write more intuitive code.

```
# Get observations where gear is equal to 4 and disp is less than 80
mtcars %>% filter(gear == 4, disp < 80)
                mpg cyl disp hp drat    wt  qsec vs am gear carb
Fiat 128       32.4   4 78.7 66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4 75.7 52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4 71.1 65 4.22 1.835 19.90  1  1    4    1
Fiat X1-9      27.3   4 79.0 66 4.08 1.935 18.90  1  1    4    1
## this is equivalent to
subset(mtcars, gear == 4 & disp < 80)
                mpg cyl disp hp drat    wt  qsec vs am gear carb
Fiat 128       32.4   4 78.7 66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4 75.7 52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4 71.1 65 4.22 1.835 19.90  1  1    4    1
Fiat X1-9      27.3   4 79.0 66 4.08 1.935 18.90  1  1    4    1
```

`arrange()`

We can arrange our observations in any convenient way using the `arrange()` function:

```
# Arrange mtcars in descending order of mpg
mtcars %>%
    arrange(mpg) %>%
    head()
                     mpg cyl disp  hp drat    wt  qsec vs am gear carb
Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
Maserati Bora       15.0   8  301 335 3.54 3.570 14.60  0  1    5    8
# Arrange mtcars in descending order of disp
mtcars %>%
    arrange(desc(disp)) %>%
    head()
                     mpg cyl disp  hp drat    wt  qsec vs am gear carb
Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
Pontiac Firebird    19.2   8  400 175 3.08 3.845 17.05  0  0    3    2
Hornet Sportabout   18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
```

`summarize()`

We can compute several summaries using the `summarize()` function. For example, if we want to calculate the maximum `mpg` value, the minimum value `disp`, and the mean of `qsec` in `mtcars` we do:

```
mtcars %>%
    summarize(maximum_mpg = max(mpg),
          minimum_disp = min(disp),
          average_qsec = mean(qsec))
  maximum_mpg minimum_disp average_qsec
1        33.9         71.1     17.84875
```

`summarize()` combines well with the `group_by()` function. For example,

```
iris %>%
    group_by(Species) %>%
    summarize(average_Sepal_Length = mean(Sepal.Length),
              Average_Sepal_Width = mean(Sepal.Width),
              average_Petal_Length = mean(Petal.Length),
              average_Petal_Width = mean(Petal.Width))
# A tibble: 3 x 5
  Species  average_Sepal_Le… Average_Sepal_Wi… average_Petal_L…
average_Petal_W…
  <fct>                <dbl>             <dbl>            <dbl>
<dbl>
1 setosa                5.01              3.43             1.46
0.246
2 versico…              5.94              2.77             4.26
1.33
3 virgini…              6.59              2.97             5.55
2.03
```

Try to understand exactly what the above code chunk does.

`*_join()`

dplyr also provides verbs for joining data sets. Below, we use the `band_members` and `band_instruments` `tibble`s (which are `data.frame` structures) to illustrate.

```
band_members
# A tibble: 3 x 2
  name  band
  <chr> <chr>
1 Mick  Stones
2 John  Beatles
3 Paul  Beatles
band_instruments
# A tibble: 3 x 2
  name  plays
  <chr> <chr>
1 John  guitar
2 Paul  bass
3 Keith guitar
```

Suppose that we want to find all rows of `band_members` for which there are matching values in `band_instruments`, and return all columns from the two `tibble`s. We can achieve that using `inner_join`:

```
band_members %>% inner_join(band_instruments)
Joining, by = "name"
# A tibble: 2 x 3
  name  band    plays
  <chr> <chr>   <chr>
1 John  Beatles guitar
2 Paul  Beatles bass
```

We can also use `left_join()` to return all rows of `band_members` and all columns from the two `tibble`s, with `NA` in the new columns if there are no matches in the rows of `band_members` and `band_instruments`:

```
band_members %>% left_join(band_instruments)
Joining, by = "name"
# A tibble: 3 x 3
  name  band    plays
  <chr> <chr>   <chr>
1 Mick  Stones  <NA>
2 John  Beatles guitar
3 Paul  Beatles bass
```

We can use `right_join()` to return all rows of `band_instruments` and all columns from the two `tibble`s, with `NA` in the new columns if there are no matches in the rows of `band_instruments` and `band_members`

```
band_members %>% right_join(band_instruments)
Joining, by = "name"
# A tibble: 3 x 3
  name  band    plays
  <chr> <chr>   <chr>
1 John  Beatles guitar
2 Paul  Beatles bass
3 Keith <NA>    guitar
```

Similarly, we can return all rows and columns of both `band_members` and `band_instruments` with `NA` in the new columns if there are no matches in the rows of `band_instruments` and `band_members`

```
band_members %>% full_join(band_instruments)
Joining, by = "name"
# A tibble: 4 x 3
  name  band    plays
  <chr> <chr>   <chr>
1 Mick  Stones  <NA>
2 John  Beatles guitar
3 Paul  Beatles bass
4 Keith <NA>    guitar
```

Notice that the same join (with different ordering of rows) can be obtained by

```
band_instruments %>% full_join(band_members)
Joining, by = "name"
# A tibble: 4 x 3
  name  plays  band
  <chr> <chr>  <chr>
1 John  guitar Beatles
2 Paul  bass   Beatles
```

```
3 Keith guitar <NA>
4 Mick   <NA>   Stones
```

All examples above also work with `data.frame`s. More information and examples on **dplyr** verbs for joining data sets can be found at **dplyr**'s join pages.

Useful Links and Resources

- Datacamp's tutorial on the `*apply()` family and associated functions
- MarinStatsLectures in Youtube on the `apply` function
- Xianjun Dong's post on **reshape2** and **tidy2**
- Overview of the **tidyr** R package
- The chapter on subsetting in *Advanced R*
- **dplyr**'s join pages for verbs to join `data.frame`s and `tibble`s

Data Wrangling Operations in Python

** Note: The code chunks below should be run in the following order **

Introduction

In this page we will use the same dataset `airquality`, from the unit on data wrangling in R. We first export the data from R to CSV files so that the dataset can be accessible for Python:

```
write.table(airquality, file = "airquality.csv", sep = ",", row.names = FALSE)
```

We then load the data to Python, and rearrange the columns so that `Month` and `Day` information are in the first two columns:

```
import pandas as pd
import numpy as np

airquality = pd.read_csv("airquality.csv")
airquality = airquality[['Month','Day','Ozone','Solar.R','Temp','Wind']]
airquality.head()
   Month  Day  Ozone  Solar.R  Temp  Wind
0      5    1   41.0    190.0    67   7.4
1      5    2   36.0    118.0    72   8.0
2      5    3   12.0    149.0    74  12.6
3      5    4   18.0    313.0    62  11.5
4      5    5    NaN      NaN    56  14.3
```

There is some missing data in the dataset. We can see the data as in the wide format as as it has a column for each variable (`Ozone`, `Solar.R`, `Temp`, `Wind`). In each row, we get the values for `Ozone`, `Solar.R`, `Temp`, `Wind` level for a particular day. For example, for example, the first row has `Ozone = 41.0`, `Solar.R = 190.0`, `Temp = 67`, `Wind = 7.4` for the day 1 May.

In this page we will first show how we can reshape the dataset. We then show how we can select a subset of the data. Finally we show how we can handle the missing data.

Reshaping Dataset

Python *pandas* provides a variety of methods/functions for reshaping your data before analysis, and many of the methods are very similar to the R functions. We will reshape the data with the following methods:

- Wide to long: `pandas.melt()`, `stack()`
- Long to wide: `pivot_table()`

Wide to Long

We can convert the data from wide to long by the `pandas.melt()` function:

```
pd.melt(airquality)
    variable  value
0      Month    5.0
1      Month    5.0
2      Month    5.0
3      Month    5.0
4      Month    5.0
..       ...    ...
913     Wind    6.9
914     Wind   13.2
915     Wind   14.3
916     Wind    8.0
917     Wind   11.5

[918 rows x 2 columns]
```

Similar to `melt()` in R, `pandas.melt()` in Python will set the first column named `variable` which records the name of the variable and a second column named value which records the value of that variable. For this data set it is not really useful to convert the data to the `DataFrame` above. Instead we should keep `Month` and `Day` as columns:

```
airquality_long = pd.melt(airquality, id_vars = ['Month', 'Day'], value_vars = ['Oz
one', 'Solar.R', 'Wind', 'Temp'],
                         var_name='climate_var', value_name='climate_value')
airquality_long.head()
   Month  Day climate_var  climate_value
0      5    1       Ozone           41.0
1      5    2       Ozone           36.0
2      5    3       Ozone           12.0
3      5    4       Ozone           18.0
4      5    5       Ozone            NaN
```

In this representation, each row gives the reading of a particular measure for a particular day. For example, the first row tells us that the ozone level is `41.0` on 1 May. This is done by providing extra arguments to the function `pandas.melt()`. The `id_vars = ['Month', 'Day']` argument specifies that `Month` and `Day` columns should be kept. By providing the argument `value_vars = ['Ozone', 'Solar.R', 'Wind', 'Temp']`, the other columns (`Ozone`, `Solar.R`, `Temp`, `Wind`) are converted into long format . We set the column names in our long format by passing the additional arguments `var_name='climate_var', value_name='climate_value'`. You may notice the syntax for `pandas.melt()` is very similar to the one used in `melt()` in R.

Long to Wide

We can use the method `pivot_table()` from `pandas.DataFrame` to convert data from long to wide format:

```
airquality_wide = airquality_long.pivot_table(values='climate_value', index=['Month
', 'Day'], columns=['climate_var'])

airquality_wide.head()

climate_var  Ozone  Solar.R  Temp  Wind

Month Day

5      1       41.0    190.0  67.0   7.4

       2       36.0    118.0  72.0   8.0

       3       12.0    149.0  74.0  12.6

       4       18.0    313.0  62.0  11.5

       5        NaN      NaN  56.0  14.3
```

Note that while the resulting table `airquality_wide` is very similar to the original table `airquality`, we now have `Multindex` which groups `Month` and `Day` together for `airquality_wide`:

```
airquality_wide.index

MultiIndex([(5,  1),

            (5,  2),

            (5,  3),

            (5,  4),

            (5,  5),

            (5,  6),

            (5,  7),

            (5,  8),

            (5,  9),

            (5, 10),

            ...

            (9, 21),

            (9, 22),

            (9, 23),

            (9, 24),
```

```
            (9, 25),
            (9, 26),
            (9, 27),
            (9, 28),
            (9, 29),
            (9, 30)],
           names=['Month', 'Day'], length=153)
```

This is because we set `index=['Month', 'Day']` so that two columns are used as key. For the other arguments, `columns=['climate_var']` specifies that the data should be grouped by the values in the column `climates_var` (i.e. `Ozone`, `Solar.R`, `Temp`, `Wind`) for the columns, and `values='climate_value'` specifies that the values from the column `climate_value` should be used to fill in the table. In our example, there is only one instance for the combination of row and column (e.g. `Month`, `Day`, `Ozone`), so the value from `climate_value` is used directly. For `airquality_wide`, each cell in the table represents the reading of a air quality measure (read from the column name) on a particular day (read from the `MultiIndex` in the row). For example, the value `41.0`in the first cell is the reading of `Ozone` on 1 May.

What if there is more than one instance for the combination of row and column? For example, if we consider using only `Month` for the index, then for a given `Month` and `Ozone`, we will have multiple values `41.0`, `36.0`, etc from different values of `Day`. What does `pivot_table()` return in this case?

```
airquality_month = airquality_long.pivot_table(values='climate_value', index=['Month'], columns=['climate_var'])
airquality_month.head()
climate_var      Ozone     Solar.R       Temp       Wind
Month
5            23.615385  181.296296  65.548387  11.622581
6            29.444444  190.166667  79.100000  10.266667
7            59.115385  216.483871  83.903226   8.941935
8            59.961538  171.857143  83.967742   8.793548
9            31.448276  167.433333  76.900000  10.180000
```

It seems that the value inside the table is the average value. We can confirm it by considering the following code, which we specify using mean to aggregating the data by the argument `aggfunc=np.mean`:

```
airquality_month = airquality_long.pivot_table(values='climate_value', index=['Month'], columns=['climate_var'], aggfunc=np.mean)
airquality_month.head()
climate_var      Ozone     Solar.R       Temp       Wind
Month
5            23.615385  181.296296  65.548387  11.622581
6            29.444444  190.166667  79.100000  10.266667
7            59.115385  216.483871  83.903226   8.941935
8            59.961538  171.857143  83.967742   8.793548
```

```
9            31.448276  167.433333  76.900000  10.180000
```

As we can see, the return values are the same, showing us indeed averaging is used by default if we have multiple values for a given combination. For the table above, the value in the first cell (23.6) represents the monthly average ozone reading for `Month = 5` (i.e. May).

We can use other aggregating function, for example `np.max` to get the maximum value:

```
airquality_month = airquality_long.pivot_table(values='climate_value', index=['Month'], columns=['climate_var'], aggfunc=np.max)

airquality_month.head()

climate_var  Ozone  Solar.R  Temp  Wind

Month

5            115.0    334.0  81.0  20.1

6             71.0    332.0  93.0  20.7

7            135.0    314.0  92.0  14.9

8            168.0    273.0  97.0  15.5

9             96.0    259.0  93.0  16.6
```

For the table above, the value in the first cell (115.0) represents the maximum ozone reading for `Month = 5` (i.e. May).

Stack and Unstack

Like R, we can stack the `DataFrame`. All the columns except the index (or MultiIndex) are stacked into one new column:

```
airquality_stack = airquality_wide.stack()

airquality_stack

Month  Day  climate_var

5      1    Ozone           41.0

            Solar.R        190.0

            Temp            67.0

            Wind             7.4

       2    Ozone           36.0

                           ...

9      29   Wind             8.0

       30   Ozone           20.0

            Solar.R        223.0

            Temp            68.0

            Wind            11.5

Length: 568, dtype: float64
```

We can turn it back to the original data structure by `unstack()`:

```
airquality_stack.unstack()
```

```
climate_var  Ozone  Solar.R  Temp  Wind
Month Day
5     1       41.0    190.0  67.0   7.4
      2       36.0    118.0  72.0   8.0
      3       12.0    149.0  74.0  12.6
      4       18.0    313.0  62.0  11.5
      5        NaN      NaN  56.0  14.3
...            ...      ...   ...   ...
9     26      30.0    193.0  70.0   6.9
      27       NaN    145.0  77.0  13.2
      28      14.0    191.0  75.0  14.3
      29      18.0    131.0  76.0   8.0
      30      20.0    223.0  68.0  11.5

[153 rows x 4 columns]
```

Subsetting

Like in R, *pandas* in Python allows users to access a subset of data very easily.

Subsetting `DataFrame` Columns

We can get a column from a `DataFrame` by simply giving the column name in the square bracket:

```
airquality['Ozone']
0      41.0
1      36.0
2      12.0
3      18.0
4       NaN
       ...
148    30.0
149     NaN
150    14.0
151    18.0
152    20.0
Name: Ozone, Length: 153, dtype: float64
```

Note that the above code chunk returns a `Series` (i.e. not a `DataFrame`)

```
type(airquality['Ozone'])
```

```
<class 'pandas.core.series.Series'>
```

To return a `DataFrame` use:

```
airquality[['Ozone']]
     Ozone
0     41.0
1     36.0
2     12.0
3     18.0
4      NaN
..     ...
148   30.0
149    NaN
150   14.0
151   18.0
152   20.0

[153 rows x 1 columns]
```

To select more than one column, use `[['col1', 'col2',..]]`:

```
airquality[['Ozone', 'Temp']]
     Ozone  Temp
0     41.0    67
1     36.0    72
2     12.0    74
3     18.0    62
4      NaN    56
..     ...   ...
148   30.0    70
149    NaN    77
150   14.0    75
151   18.0    76
152   20.0    68

[153 rows x 2 columns]
```

Alternative, you can use the following:

```
airquality.iloc[:,np.array([2,4])]
     Ozone  Temp
```

```
0      41.0    67
1      36.0    72
2      12.0    74
3      18.0    62
4       NaN    56
..      ...   ...
148    30.0    70
149     NaN    77
150    14.0    75
151    18.0    76
152    20.0    68

[153 rows x 2 columns]
```

In the above, inside the square brackets, the rows are specified first (here `:` means all) and, after the coma, the columns are specified by `(np.array([2,4])` that corresponds to the indices of the `Ozone` and `Temp` columns.

Subsetting `DataFrame` Rows:

We can get rows from a `DataFrame` by simply giving the row indexes in the square bracket:

```
airquality[:5]
   Month  Day  Ozone  Solar.R  Temp  Wind
0      5    1   41.0    190.0    67   7.4
1      5    2   36.0    118.0    72   8.0
2      5    3   12.0    149.0    74  12.6
3      5    4   18.0    313.0    62  11.5
4      5    5    NaN      NaN    56  14.3
```

We can also use `.loc[]` and `.iloc[]` to select rows. `.loc[]` is a label-based way to get the specific rows (or columns as well), whereas `.iloc[]` is an *index*-based way to get the specific rows (or columns as well). For example, for the `airquality` dataset, we can get the first five rows by

```
airquality.loc[:5]
   Month  Day  Ozone  Solar.R  Temp  Wind
0      5    1   41.0    190.0    67   7.4
1      5    2   36.0    118.0    72   8.0
2      5    3   12.0    149.0    74  12.6
3      5    4   18.0    313.0    62  11.5
4      5    5    NaN      NaN    56  14.3
5      5    6   28.0      NaN    66  14.9
```

or

```
airquality.iloc[:5]
   Month  Day  Ozone  Solar.R  Temp  Wind
0      5    1   41.0    190.0    67   7.4
1      5    2   36.0    118.0    72   8.0
2      5    3   12.0    149.0    74  12.6
3      5    4   18.0    313.0    62  11.5
4      5    5    NaN      NaN    56  14.3
```

However, if we use the same method on `airquality_wide` instead, the result is different for `.loc[]`:

```
airquality_wide.loc[:5]
climate_var  Ozone  Solar.R  Temp  Wind
Month Day
5     1       41.0    190.0  67.0   7.4
      2       36.0    118.0  72.0   8.0
      3       12.0    149.0  74.0  12.6
      4       18.0    313.0  62.0  11.5
      5        NaN      NaN  56.0  14.3
      6       28.0      NaN  66.0  14.9
      7       23.0    299.0  65.0   8.6
      8       19.0     99.0  59.0  13.8
      9        8.0     19.0  61.0  20.1
      10       NaN    194.0  69.0   8.6
      11       7.0      NaN  74.0   6.9
      12      16.0    256.0  69.0   9.7
      13      11.0    290.0  66.0   9.2
      14      14.0    274.0  68.0  10.9
      15      18.0     65.0  58.0  13.2
      16      14.0    334.0  64.0  11.5
      17      34.0    307.0  66.0  12.0
      18       6.0     78.0  57.0  18.4
      19      30.0    322.0  68.0  11.5
      20      11.0     44.0  62.0   9.7
      21       1.0      8.0  59.0   9.7
      22      11.0    320.0  73.0  16.6
      23       4.0     25.0  61.0   9.7
      24      32.0     92.0  61.0  12.0
      25       NaN     66.0  57.0  16.6
```

```
      26        NaN    266.0  58.0  14.9
      27        NaN      NaN  57.0   8.0
      28       23.0     13.0  67.0  12.0
      29       45.0    252.0  81.0  14.9
      30      115.0    223.0  79.0   5.7
      31       37.0    279.0  76.0   7.4
airquality_wide.iloc[:5]
climate_var  Ozone  Solar.R  Temp  Wind
Month Day
5     1       41.0    190.0  67.0   7.4
      2       36.0    118.0  72.0   8.0
      3       12.0    149.0  74.0  12.6
      4       18.0    313.0  62.0  11.5
      5        NaN      NaN  56.0  14.3
```

Why is this the case? Note that the *label* of the index for `airquality_wide` is:

```
airquality_wide.index
MultiIndex([(5,  1),
            (5,  2),
            (5,  3),
            (5,  4),
            (5,  5),
            (5,  6),
            (5,  7),
            (5,  8),
            (5,  9),
            (5, 10),
            ...
            (9, 21),
            (9, 22),
            (9, 23),
            (9, 24),
            (9, 25),
            (9, 26),
            (9, 27),
            (9, 28),
            (9, 29),
            (9, 30)],
           names=['Month', 'Day'], length=153)
```

so `airquality_wide.loc[:5]` gets all rows with `Month = 5`. See here for more details.

We can also get the rows by condition, for example:

```
airquality.loc[airquality.Day == 3, 'Ozone']
2       12.0
33       NaN
63      32.0
94      16.0
125     73.0
Name: Ozone, dtype: float64
```

The above code chunk returns the ozone reading on 3 May, 3 June, …, 3 September.

Subsetting `DataFrame` Rows and Columns

Above we only specify one value in `.loc[]` and `.iloc[]`. If we specify one more value, we can specify the columns as well. For example, if we want to get the first row with ozone readings, we can use the `.loc[]`

```
airquality.loc[:5, 'Ozone']
0     41.0
1     36.0
2     12.0
3     18.0
4      NaN
5     28.0
Name: Ozone, dtype: float64
```

Or the `.iloc[]`:

```
airquality.iloc[:5, 2]
0     41.0
1     36.0
2     12.0
3     18.0
4      NaN
Name: Ozone, dtype: float64
```

Handling Missing Values

You probably can see that there are some missing value in our dataset:

```
airquality.head(10)
   Month  Day  Ozone  Solar.R  Temp  Wind
```

```
0       5    1   41.0    190.0    67   7.4
1       5    2   36.0    118.0    72   8.0
2       5    3   12.0    149.0    74  12.6
3       5    4   18.0    313.0    62  11.5
4       5    5    NaN      NaN    56  14.3
5       5    6   28.0      NaN    66  14.9
6       5    7   23.0    299.0    65   8.6
7       5    8   19.0     99.0    59  13.8
8       5    9    8.0     19.0    61  20.1
9       5   10    NaN    194.0    69   8.6
```

We can remove them by `.dropna()`, which removes rows containing *any* missing data.

```
airquality_no_na = airquality.dropna()
airquality_no_na.head(10)
    Month  Day  Ozone  Solar.R  Temp  Wind
0       5    1   41.0    190.0    67   7.4
1       5    2   36.0    118.0    72   8.0
2       5    3   12.0    149.0    74  12.6
3       5    4   18.0    313.0    62  11.5
6       5    7   23.0    299.0    65   8.6
7       5    8   19.0     99.0    59  13.8
8       5    9    8.0     19.0    61  20.1
11      5   12   16.0    256.0    69   9.7
12      5   13   11.0    290.0    66   9.2
13      5   14   14.0    274.0    68  10.9
```

Sometimes, removing the rows with missing data may not be appropriate. Instead we may want to fill in the missing values in a systematic way. For example, we can replace the missing values with previous value:

```
airquality_fill_na = airquality.fillna(method='ffill')
airquality_fill_na.head(10)
   Month  Day  Ozone  Solar.R  Temp  Wind
0       5    1   41.0    190.0    67   7.4
1       5    2   36.0    118.0    72   8.0
2       5    3   12.0    149.0    74  12.6
3       5    4   18.0    313.0    62  11.5
4       5    5   18.0    313.0    56  14.3
5       5    6   28.0    313.0    66  14.9
6       5    7   23.0    299.0    65   8.6
7       5    8   19.0     99.0    59  13.8
```

```
8       5    9    8.0     19.0    61   20.1
9       5   10    8.0    194.0    69    8.6
```

There are different settings for the fill rule. See the pandas fillna() method for DataFrame for more details. Alternatively we can use interpolation to fill the missing values. For example:

```
airquality_linear = airquality.interpolate(method='linear')
airquality_linear.head(10)
   Month  Day  Ozone     Solar.R  Temp  Wind
0       5    1   41.0  190.000000    67   7.4
1       5    2   36.0  118.000000    72   8.0
2       5    3   12.0  149.000000    74  12.6
3       5    4   18.0  313.000000    62  11.5
4       5    5   23.0  308.333333    56  14.3
5       5    6   28.0  303.666667    66  14.9
6       5    7   23.0  299.000000    65   8.6
7       5    8   19.0   99.000000    59  13.8
8       5    9    8.0   19.000000    61  20.1
9       5   10    7.5  194.000000    69   8.6
```

Here we use linear method to interpolate, with the missing ozone data on 5 May calculated as `(18+28)/2 = 23`.

Combining Different Datasets

We downloaded the coronavirus new cases and new deaths data from https://coronavirus.data.gov.uk. We load the data into `cases_df` and `death_df` via *pandas*

```
import pandas as pd
cases_df = pd.read_csv("covid_new_cases.csv", index_col = 0)
deaths_df = pd.read_csv("covid_new_deaths.csv", index_col = 0)
```

Both `DataFrame` has `date` as the index (i.e. the row labels), although the `case_df` has more data.

```
print(cases_df)
            new_cases
date
2020-01-30          2
2020-01-31          0
2020-02-01          0
2020-02-02          0
2020-02-03          0
...               ...
```

```
2020-10-25      15654
2020-10-26      26467
2020-10-27      23757
2020-10-28      22887
2020-10-29      19525

[274 rows x 1 columns]
print(deaths_df)
            new_deaths
date
2020-02-29           0
2020-03-01           0
2020-03-02           1
2020-03-03           2
2020-03-04           0
...                ...
2020-10-25         234
2020-10-26         253
2020-10-27         227
2020-10-28         216
2020-10-29         217

[244 rows x 1 columns]
```

Here we want to join the two datasets together. `pandas` has fast and full-featured functions for combining datasets via the method `join()`, and these joins can be one-to-one, many-to-one or many-to-many. These merges can be done in a number of ways shown in the table below:

Method	**Behaviour**
left	Use calling frame's index (or column if on is specified)
right	Use other's index
outer	Use union of keys from both `DataFrame`
inner	Use intersection of keys from both `DataFrame`

For example, we can join the `cases_df` and `death_df` by:

```
df_1 = cases_df.join(deaths_df)
print(df_1)
            new_cases  new_deaths
date
2020-01-30          2         NaN
2020-01-31          0         NaN
2020-02-01          0         NaN
2020-02-02          0         NaN
```

```
2020-02-03           0          NaN
...                ...          ...
2020-10-25       15654        234.0
2020-10-26       26467        253.0
2020-10-27       23757        227.0
2020-10-28       22887        216.0
2020-10-29       19525        217.0

[274 rows x 2 columns]
```

In the above we use all the row labels from `cases_df`, as the default is `left` join. We can see that the number of rows of the resulting `DataFrame` is the same as the number of rows in `cases_df`, as we use all the row labels from `cases_df`. We also see that there are some NA data for the `new_deaths` column in `df_1` , as for the dates like `2020-01-30` there is no data available from the `deaths_df`.

If we want to use all the keys of `deaths_df` instead, we can add the argument `how = 'right'`:

```
df_2 = cases_df.join(deaths_df, how = 'right')
print(df_2)
            new_cases  new_deaths
date
2020-02-29          5           0
2020-03-01         22           0
2020-03-02         40           1
2020-03-03         56           2
2020-03-04         56           0
...               ...         ...
2020-10-25      15654         234
2020-10-26      26467         253
2020-10-27      23757         227
2020-10-28      22887         216
2020-10-29      19525         217

[244 rows x 2 columns]
```

Now we have the number of rows of `df_2` equals to the number of rows in `deaths_df`, as we use all the row labels from `deaths_df`.

If we want to the keys that are found in both `cases_df` and `deaths_df`, we can use `how = 'inner'):

```
df_3 = cases_df.join(deaths_df, how = 'inner')
print(df_3)
            new_cases  new_deaths
```

```
date
2020-02-29          5           0
2020-03-01         22           0
2020-03-02         40           1
2020-03-03         56           2
2020-03-04         56           0
...               ...         ...
2020-10-25      15654         234
2020-10-26      26467         253
2020-10-27      23757         227
2020-10-28      22887         216
2020-10-29      19525         217

[244 rows x 2 columns]
```

For this particular example, `df_2` and `df_3` are the same as the intersection of the dates (rows) from `cases_df` and `deaths_df`, is the same as the dates in `deaths_df`.

Useful Links and Resources

- McKinney, W. (2013). Python for data analysis. Chapter 7.
- `pandas.DataFrame` official documentation