DAFS Course manual

Report subtitle

J. van der Pol

2023-10-17

Table of contents

Preface

This manual regroups all the theory and exercises for the course "Data Analysis for Sustainability" of the UU. You will find here all the exercices and readings for the modules Text Mining and Programming.

```
1 + 1
```

```
[1] 2
```

1 Introduction

This is a book created from markdown and executable code.

To search for carbon reduction technologies we use the following query in lens.org

```
 ( Title: ( "carbon capture" ) OR ( Abstract: ( "carbon capture" ) OR ( Keyword: ( "carbon
```

Or Scopus:

```
(TITLE-ABS-KEY("carbon capture") OR TITLE-ABS-KEY("carbon capture and storage") OR TITLE-A
```

```
https://www.kaggle.com/datasets/anshtanwar/global-data-on-sustainable-energy
```

2 Summary

In summary, this book has no content whatsoever.

```
1 + 1
```

```
[1] 2
```

Part I

The Programming module

This chapter discusses how to extract terms with high value from the text

```
1 + 1
```

```
[1] 2
```

3 Basic Operations

This chapter shows how to use basic operations in both R and Python. We will focus on operations that we encounter in a data analysis setting: arithmetic, comparisons, logic.

You are expected to be able to use all the operations described in this chapter.

3.0.1 Basic Mathematical operations

In both R and Python, basic mathematical operations can be written directly in the

3.0.1.1 Arithmetic operators

```
# In R
1 + 1
```

```
[1] 2
```

```
4 * 2
```

```
[1] 8
```

```
4 / 2
```

```
[1] 2
```

```python
# In python
1 + 1
```

```
2
```

```python
4 * 2
```

```
8
```

```python
4 / 2
```

```
2.0
```

Pay Attention

For more complex mathematical operators we might need some functions that are not provided in the base version of the software we use. For instance, the ***ln()*** function and ***exp()*** function are not available in base python. We therefore need to load a package which contains these functions, this will be the **maths** package in Python. For R, both the ***ln()*** and ***exp()*** functions are part of the base package, we therefor do not need to load an additional package, we can use these functions right away.

```r
# Base R
log(12)
```

```
[1] 2.484907
```

```
exp(12)
```

```
[1] 162754.8
```

```
sqrt(12) # square root
```

```
[1] 3.464102
```

```
import math
math.log(12)
```

```
2.4849066497880004
```

```
math.exp(12)
```

```
162754.79141900392
```

```
math.sqrt(12)
```

```
3.4641016151377544
```

3.0.1.2 Comparison Operators

Comparison operations function in the same way in both languages. The only difference we note is in the assignment of the values to a variable in which it is recommended in R to use <-.

Warning

Even though = works for simple operations in R, you might run into trouble with more complex code. For a detailed explanation see this post. We recommend you to always use the <- operator when programming in R.

```r
# R
x <- 5
y <- 5
# Equal to
x == y
```

```
[1] TRUE
```

```r
# Not equal to
x != y
```

```
[1] FALSE
```

```r
# Greater than
x > y
```

```
[1] FALSE
```

```r
# Less than
x < y
```

```
[1] FALSE
```

```
# Greater than or equal to
x >= y
```

```
[1] TRUE
```

```
# Less than or equal to
x <= y
```

```
[1] TRUE
```

```
# Python
x = 5
y = 5
# Equal to
x == y
```

```
True
```

```
# Not equal to
x != y
```

```
False
```

```
# Greater than
x > y
```

```
False
```

```
# Less than
x < y
```

```
False
```

```
# Greater than or equal to
x >= y
```

```
True
```

```
# Less than or equal to
x <= y
```

```
True
```

3.0.1.3 Logic Operators

Logical operators are commonly used in data analysis, especially when sub-setting datasets. For example when we want to extract documents that are from the year 2000 which have the term "sustainability" and the term "climate change" but not the term "fossil fuel". Combining these operators is important, and so is understanding how they work.

Pay Attention

Some differences between R and Python become apparent here. In R, **TRUE** and **FALSE** must be written in all caps to be recognised as the logical operator. In Python, **True** and **False** must start with a capitalized letter. **or**, **and**, **not** should also be written exactly in this manner.If these operators are written differently, they will be recognized as objects.

```r
x <- 4
y <- 8
# Equal (==)
x == y
```

```
[1] FALSE
```

```r
# And (&)
x == 4 & y == 8
```

```
[1] TRUE
```

```r
# Or (|)
x == 4 | y == 8
```

```
[1] TRUE
```

```r
# Not (!)
!y
```

```
[1] FALSE
```

```r
# Combine
z <- "plop"
x == 4 & (y == 8 | z == "plop")
```

```
[1] TRUE
```

```
x = 4
y = 8
# Equal (==)
x == y
```

```
False
```

```
# And
x == 4 and y == 8
```

```
True
```

```
# Or
x == 4 or y == 8
```

```
True
```

```
# Not
not x
```

```
False
```

```
# Combine
z = "plop"
x == 4 and (y == 8 or z == "plop")
```

```
True
```

In data analysis, we usually use operators to subset data. This means that we compare a variable to a value to check if it fits our criteria. For example, if we have a column that contains a year, and we only want observations with the year 2003, we will search for year == 2003. In this setting the R operators we just described will be the same. It is possible that these operators varie when different packages are used in python. For instance, in the context of the pandas package, **and** becomes **&**, **or** becomes |, **not** becomes ~. We will adress these variations in the database manipulation chapter.

4 Loops, if-else and Mapping

This chapter discusses how to create loops and program with if-else statements. These chapters only contain elements that we expect you to know at the end of the course. More detailled books and references exist. For more detailes explanations on specific points, please check: R for Data Science.

Loops are a useful tool for programming. It allows us to automatically repeat operations. For example if we want to compute indicators for each year in a dataset, we can automatically subset per year, compute indicators, store the results and move on to the next year.

Warning

The biggest difference between R and Python resides in the fact that Python starts at 0 while R starts at 1. You can see that in the results below. ***range(5)*** gives 0,1,2,3,4 in python.

4.1 Making loops

4.1.1 For Loops

A loop in R starts with the **for** operator. This is followed by a condition that determines how the loop runs ("the loop runs for…"). We start by defining a variable that will take the different values in the loop. Suppose we want to print the value 1 to 5. This requires a loop that takes a variables that starts at 1, and increases by one with each iteration. The **in** operator is used to define the values the variables will take.

In the following code we will show different ways to loop:

```r
# range of numbers
for (i in 1:5) {
  print(i)
}
```

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

```r
# items in a vector
fruits <- c("apple", "banana", "cherry", "date")
for (fruit in fruits) {
  print(fruit)
}
```

```
[1] "apple"
[1] "banana"
[1] "cherry"
[1] "date"
```

```r
# Loop with an index
languages <- c("Python", "Java", "C++", "Ruby")
for (i in 1:length(languages)) {
  cat("Language", i, "is", languages[i], "\n")
}
```

```
Language 1 is Python
Language 2 is Java
Language 3 is C++
Language 4 is Ruby
```

```r
# custom step
for (number in seq(1, 10, by = 2)) {
  cat("Odd number:", number, "\n")
}
```

```
Odd number: 1
Odd number: 3
Odd number: 5
Odd number: 7
Odd number: 9
```

```r
# Nested loops
for (i in 1:3) {
  for (j in 1:2) {
    cat("i =", i, ", j =", j, "\n")
  }}
```

```
i = 1 , j = 1
i = 1 , j = 2
i = 2 , j = 1
i = 2 , j = 2
i = 3 , j = 1
i = 3 , j = 2
```

```r
# break statement
# it is possible to stop the loop given a certain condition
numbers <- c(3, 7, 1, 9, 4, 2)
for (num in numbers) {
  if (num == 9) {
    cat("Found 9. Exiting the loop.\n")
    break
  }
  cat("Processing", num, "\n")
}
```

```
Processing 3
Processing 7
Processing 1
Found 9. Exiting the loop.
```

A loop in Python starts with the **for** operator. This is followed by a condition that determines how the loop runs ("the loop runs for…"). We start by defining a variable that will take the different values in the loop. Suppose we want to print the value 0 to 4. This requires a loop that takes a variables that starts at 0, and increases by one with each iteration. The **in** operator is used to define the values the variables will take.

In the following code we will show different ways to loop:

```
# a range of numbers

for i in range(5):
    print("Iteration", i)
```

```
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

```
# items in a list
fruits = ["apple", "banana", "cherry", "date"]

for fruit in fruits:
    print("I like", fruit)
```

```
I like apple
I like banana
I like cherry
I like date
```

```
# with an index
languages = ["Python", "Java", "C++", "Ruby"]

for i, language in enumerate(languages):
    print("Language", i + 1, "is", language)
```

```
Language 1 is Python
Language 2 is Java
Language 3 is C++
Language 4 is Ruby
```

```python
# with a custom step

for number in range(1, 11, 2):
    print("Odd number:", number)
```

```
Odd number: 1
Odd number: 3
Odd number: 5
Odd number: 7
Odd number: 9
```

```python
# Nested loops

for i in range(3):
    for j in range(2):
        print("i =", i, ", j =", j)
```

```
i = 0 , j = 0
i = 0 , j = 1
i = 1 , j = 0
i = 1 , j = 1
i = 2 , j = 0
i = 2 , j = 1
```

```python
# a break statement
numbers = [3, 7, 1, 9, 4, 2]


for num in numbers:
    if num == 9:
        print("Found 9. Exiting the loop.")
        break
    print("Processing", num)
```

```
Processing 3
Processing 7
Processing 1
Found 9. Exiting the loop.
```

Pay Attention

While R, in general, does not care about the indentation of your code, Python does. If the code is not properly indented, the script will not run.

4.1.2 While loops

While loops continue to loop as long as a specific condition is satisfied. The therefore differ from the for loops which have a specified stopping point. The danger with these loops is that they can theoretically run forever if the conditions is always verified. The basic logic of these loops is **while** followed by a condition and then the code to execute while this condition is verified:

```r
# Example 1: Simple while loop
count <- 1
while (count <= 5) {
  cat("Iteration", count, "\n")
  count <- count + 1}
```

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

```r
# Example 3: Loop with a condition and next statement (equivalent to continue in Python)
i <- 0
while (i < 10) {
  i <- i + 1
  if (i %% 2 == 0) {
    next }  # Skip even numbers
  cat(i, "\n")}
```

```
1
3
5
7
9
```

```
# Example 4: Nested while loops
row <- 1
while (row <= 3) {
  col <- 1
  while (col <= 3) {
    cat("Row", row, "Column", col, "\n")
    col <- col + 1 }
  row <- row + 1}
```

```
Row 1 Column 1
Row 1 Column 2
Row 1 Column 3
Row 2 Column 1
Row 2 Column 2
Row 2 Column 3
Row 3 Column 1
Row 3 Column 2
Row 3 Column 3
```

```
# Example 1: Simple while loop
count = 1
while count <= 5:
    print("Iteration", count)
    count += 1
```

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

```python
# Example 3: Loop with a condition and continue statement
i = 0
while i < 10:
    i += 1
    if i % 2 == 0:
        continue  # Skip even numbers
    print(i)
```

```
1
3
5
7
9
```

```python
# Example 4: Nested while loops
row = 1
while row <= 3:
    col = 1
    while col <= 3:
        print("Row", row, "Column", col)
        col += 1
    row += 1
```

```
Row 1 Column 1
Row 1 Column 2
Row 1 Column 3
Row 2 Column 1
Row 2 Column 2
Row 2 Column 3
Row 3 Column 1
Row 3 Column 2
Row 3 Column 3
```

4.2 Conditional statements (if, else, elif)

```r
# Using if statement
x <- 10
if (x > 5) {
  cat("x is greater than 5\n")}
```

```
x is greater than 5
```

```r
# Using if-else statement
y <- 3
if (y > 5) {
  cat("y is greater than 5\n")
} else {
  cat("y is not greater than 5\n")}
```

```
y is not greater than 5
```

```r
# Using if-else if-else statement
z <- 7
if (z > 10) {
  cat("z is greater than 10\n")
} else if (z > 5) {
  cat("z is greater than 5 but not greater than 10\n")
} else {
  cat("z is not greater than 5\n")}
```

```
z is greater than 5 but not greater than 10
```

```
# Nested if statements
a <- 15
b <- 6
if (a > 10) {
  if (b > 5) {
    cat("Both a and b are greater than 10 and 5, respectively\n")
  } else {
    cat("a is greater than 10, but b is not greater than 5\n")}}
```

```
Both a and b are greater than 10 and 5, respectively
```

```
# Using a conditional expression (ternary operator)
age <- 18
status <- if (age >= 18) "adult" else "minor"
cat("You are an", status, "\n")
```

```
You are an adult
```

```
# Using if statement
x = 10
if x > 5:
    print("x is greater than 5")
```

```
x is greater than 5
```

```
# Using if-else statement
y = 3
if y > 5:
    print("y is greater than 5")
else:
    print("y is not greater than 5")
```

```
y is not greater than 5
```

```
# Using if-elif-else statement
z = 7
if z > 10:
    print("z is greater than 10")
elif z > 5:
    print("z is greater than 5 but not greater than 10")
else:
    print("z is not greater than 5")
```

```
z is greater than 5 but not greater than 10
```

```
# Nested if statements
a = 15
b = 6
if a > 10:
    if b > 5:
        print("Both a and b are greater than 10 and 5, respectively")
    else:
        print("a is greater than 10, but b is not greater than 5")
```

```
Both a and b are greater than 10 and 5, respectively
```

```
# Using a conditional expression (ternary operator)
age = 18
status = "adult" if age >= 18 else "minor"
print("You are an", status)
```

```
You are an adult
```

4.2.1 Mapping functions

When working with data, we often want to apply a function to all rows of a column, or even on a whole dataframe. For example if we want to remove the capital letters from text or round up numbers to a set number of digits. This could be achieved by looping over all the cells in the dataframe, but there are often more efficient ways of doing this. There are functions that allow us to apply another function to each cell of the column, or dataframe, that we select. This function takes care of the looping behind the scenes.

In R we have multiple options for this, some in base R, some from different packages. We will discuss the most notable functions here.

apply sapply mapply lapply

map

```r
# All these functions are
# present in base R

# Example 1: apply function
matrix_data <- matrix(1:9, nrow = 3)

# Apply a function to rows (axis 1)
row_sums <- apply(matrix_data, 1, sum)
print("Row Sums:")
```

```
[1] "Row Sums:"
```

```r
print(row_sums)
```

```
[1] 12 15 18
```

```r
# Apply a function to columns (axis 2)
col_sums <- apply(matrix_data, 2, sum)
```

```r
print("Column Sums:")
```

```
[1] "Column Sums:"
```

```r
print(col_sums)
```

```
[1]  6 15 24
```

```r
# Example 2: lapply function
data_list <- list(a = 1:3, b = 4:6, c = 7:9)

# Apply a function to each element of the list
squared_list <- lapply(data_list, function(x) x^2)
print("Squared List:")
```

```
[1] "Squared List:"
```

```r
print(squared_list)
```

```
$a
[1] 1 4 9

$b
[1] 16 25 36

$c
[1] 49 64 81
```

```r
# Example 3: sapply function
# Similar to lapply, but attempts to simplify the result
squared_vector <- sapply(data_list, function(x) x^2)
print("Squared Vector:")
```

```
[1] "Squared Vector:"
```

```r
print(squared_vector)
```

```
     a  b  c
[1,] 1 16 49
[2,] 4 25 64
[3,] 9 36 81
```

```r
# Example 4: tapply function
# Used for applying a function by factors
sales_data <- data.frame(product = c("A", "B", "A", "B", "A"),
                         sales = c(100, 150, 120, 180, 90))

# Apply a function to calculate total sales by product
total_sales <- tapply(sales_data$sales, sales_data$product, sum)
print("Total Sales by Product:")
```

```
[1] "Total Sales by Product:"
```

```r
print(total_sales)
```

```
  A   B 
310 330 
```

```python
import numpy as np
import pandas as pd

# Example 1: NumPy equivalent of apply
matrix_data = np.array([[1, 4, 7], [2, 5, 8], [3, 6, 9]])

# Apply a function to rows (axis 1)
```

```python
row_sums = np.apply_along_axis(np.sum, axis=1, arr=matrix_data)
print("Row Sums:")
```

```
Row Sums:
```

```python
print(row_sums)
```

```
[12 15 18]
```

```python
# Apply a function to columns (axis 0)
col_sums = np.apply_along_axis(np.sum, axis=0, arr=matrix_data)
print("Column Sums:")
```

```
Column Sums:
```

```python
print(col_sums)
```

```
[ 6 15 24]
```

```python
# List
data_list = {'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]}

# Apply a function to each element of the dictionary values
squared_list = {key: [x**2 for x in value] for key, value in data_list.items()}
print("Squared Dictionary:")
```

```
Squared Dictionary:
```

```python
print(squared_list)
```

```
{'a': [1, 4, 9], 'b': [16, 25, 36], 'c': [49, 64, 81]}
```

```python
# Example 3: pandas
data_dict = {'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]}
df = pd.DataFrame(data_dict)

# Apply a function to each column (series) and return a DataFrame
squared_df = df.apply(lambda x: x**2)
print(squared_df)
```

```
   a   b   c
0  1  16  49
1  4  25  64
2  9  36  81
```

```python
# pandas
sales_data = pd.DataFrame({'product': ['A', 'B', 'A', 'B', 'A'],
                           'sales': [100, 150, 120, 180, 90]})

# Apply a function to calculate total sales by product
total_sales = sales_data.groupby('product')['sales'].sum()
print("Total Sales by Product:")
```

```
Total Sales by Product:
```

```python
print(total_sales)
```

```
product
A    310
B    330
Name: sales, dtype: int64
```

4.2.1.1 The map() function in R

In R we also have the **map()** function, this function is part of the **purrr** package. **map()** is able to apply a function of your chosing to each element of a list, vector or other data strucutr. The basic syntax for the function is:

```
map(.x, .f, ...)
```

The ***.x*** argument is the data structure you wish to apply the function to. The ***.f*** argument will be the function you will be applying to the dataframe (for example, **gsub()**, **as.integer()**, **tolower()**, etc.)

The main advantage of **map()** is that it is a one format solution while the base R functions (apply, lappply, sapply and tapply) all have different syntaxes and behaviour. **map()** works on all and behaves identically on all datastructures.

Here is an example of how to use this function:

```
library(purrr)
test_data = list(3,9,12,15,18)

# we will apply the sqrt() function to these numbers

test_data_sqrt = map(test_data, ~ sqrt(.))
print(test_data_sqrt)
```

```
[[1]]
[1] 1.732051

[[2]]
[1] 3

[[3]]
[1] 3.464102

[[4]]
[1] 3.872983

[[5]]
[1] 4.242641
```

It also works for dataframes:

```
test_dataframe = data.frame("value" = c(3,9,12,15,18))
test_data_sqrt = map(test_dataframe, ~ sqrt(.))
print(test_data_sqrt)
```

```
$value
[1] 1.732051 3.000000 3.464102 3.872983 4.242641
```

It also works for a matrix:

```r
test_matrix = matrix(c(3,9,12,15,18), nrow = 5, ncol = 1)
test_data_sqrt = map(test_matrix, ~ sqrt(.))
print(test_data_sqrt)
```

```
[[1]]
[1] 1.732051

[[2]]
[1] 3

[[3]]
[1] 3.464102

[[4]]
[1] 3.872983

[[5]]
[1] 4.242641
```

Pay Attention

It is possible to use **map()** on a specific colomn in a dataframe. You will need to pay close attention to select this column correctly. Applying the function to a matrix will result in the matrix being reverted back to a list. For this reason we add the **unlist()** function so that the results can be returned in matrix format.

```r
test_dataframe = data.frame("value" = c(3,9,12,15,18), "year" = c(2000,2004,2018,2021,2025
test_dataframe$value = map(test_dataframe$value, ~ sqrt(.))
print(test_dataframe)
```

```
     value year
1 1.732051 2000
2        3 2004
3 3.464102 2018
4 3.872983 2021
5 4.242641 2025
```

```r
test_matrix = matrix(c(3,9,12,15,18,2000,2004,2018,2021,2025), nrow = 5, ncol = 2)
test_matrix[,1] = unlist(map(test_matrix[,1],  ~ sqrt(.)))
# Using pipes we can also write an equivalent form:
# test_matrix[,1] = test_matrix[,1] %>% map( ~ sqrt(.)) %>% unlist()
print(test_matrix)
```

```
         [,1] [,2]
[1,] 1.732051 2000
[2,] 3.000000 2004
[3,] 3.464102 2018
[4,] 3.872983 2021
[5,] 4.242641 2025
```

5 Random numbers

How to generate random numbers. The seed.

https://www.w3schools.com/python/module_random.asp

6 Making functions

This chapter discusses how to create a function

```
1 + 1
```

```
[1] 2
```

7 Data Import and preparation

How to import different dataformats in python and R.

8 Importing in R

8.1 TXT, CSV, TSV

For the import of data from *txt*, *csv* or *tsv* formats we will use the **read_delim()** function from the readr package. The main difference between these data formats resides in the separator of the data. For txt data, this is often undefined and needs to be specified by the user. The main arguments of the **read_delim()** are the file and the separator. The separator is the character string that defines where a line should be cut. For example, if our raw data looks like this:

Year;Class;Value 2002;B60C;42 2003;B29K;21 2009;C08K;12

Then we see that each column is separated by a ";". By splitting the data whenever there is a ";" we create the following dataframe:

Year	Class	Value
2002	B60C	42
2003	B29K	21
2009	C08K	12

Seperators that we commonly find in data are the semicolon: **";"**, the comma: **","**, the vertical bar (or pipe) **"|"**, the space **" ", and tabs which are coded with**"\t".

Pay Attention

Even though csv stands for Comma Separated Values, this does not mean that the separator is always a comma, often enough it is in fact a semicolon. Always check your data to make sure you have the right one.

```r
library(readr)
# csv and various other separators
data <- read_delim(file, sep = ",")

data <- read_delim(file, sep = "|")

data <- read_delim(file, sep = ";")
```