

# **Introduction to pyspark**

Pedro Duarte Faria

4/4/23

# Table of contents

<b>About this website</b>	<b>6</b>
<b>Preface</b>	<b>7</b>
Introduction . . . . .	7
About the author . . . . .	7
Some conventions of this book . . . . .	8
Python code and terminal commands . . . . .	8
Python objects, functions and methods . . . . .	9
Be aware of differences between OS's! . . . . .	9
Install the necessary software . . . . .	9
<b>1 Key concepts of python</b>	<b>10</b>
1.1 Introduction . . . . .	10
1.2 Scripts . . . . .	10
1.3 How to run a python program . . . . .	10
1.4 Objects . . . . .	11
1.5 Expressions . . . . .	13
1.6 Packages (or libraries) . . . . .	15
1.7 Methods versus Functions . . . . .	18
1.8 Identifying classes and their methods . . . . .	20
<b>2 Introducing Apache Spark</b>	<b>22</b>
2.1 Introduction . . . . .	22
2.2 What is Spark? . . . . .	22
2.3 Spark application . . . . .	23
2.4 Spark application versus <b>pyspark</b> application . . . . .	24
2.5 Core parts of a <b>pyspark</b> program . . . . .	25
2.5.1 Importing the Spark libraries (or packages) . . . . .	25
2.5.2 Starting your Spark Session . . . . .	25
2.5.3 Defining a set of transformations and actions . . . . .	26
2.6 Building your first Spark application . . . . .	26
2.6.1 Writing the code . . . . .	26
2.6.2 Executing the code . . . . .	27
2.7 Overview of <b>pyspark</b> . . . . .	28
2.7.1 Main python modules . . . . .	28

2.7.2	Main python classes . . . . .	29
<b>3</b>	<b>Introducing Spark DataFrames</b>	<b>30</b>
3.1	Introduction . . . . .	30
3.2	Spark DataFrames versus Spark Datasets . . . . .	30
3.3	Partitions of a Spark DataFrame . . . . .	31
3.4	The <b>DataFrame</b> class in <b>pyspark</b> . . . . .	32
3.5	Building a Spark DataFrame . . . . .	34
3.6	Viewing a Spark DataFrame . . . . .	36
3.7	Getting the name of the columns . . . . .	37
3.8	Spark Data Types . . . . .	38
3.9	The DataFrame Schema . . . . .	39
3.9.1	Accessing the DataFrame schema . . . . .	39
3.9.2	Building a DataFrame schema . . . . .	41
3.9.3	Checking your DataFrame schema . . . . .	42
<b>4</b>	<b>Introducing the Column class</b>	<b>43</b>
4.1	Building a column object . . . . .	43
4.2	Columns are strongly related to expressions . . . . .	44
4.3	Literal values versus expressions . . . . .	45
4.4	Passing a literal (or a constant) value to Spark . . . . .	46
4.5	Some key column methods . . . . .	47
<b>5</b>	<b>Transforming your Spark DataFrame</b>	<b>49</b>
5.1	Introduction . . . . .	49
5.2	Defining transformations . . . . .	49
5.3	Triggering calculations with actions . . . . .	51
5.4	Understanding narrow and wide transformations . . . . .	52
5.5	The <b>transf</b> DataFrame . . . . .	54
5.6	Filtering rows of your DataFrame . . . . .	58
5.6.1	Logical operators available . . . . .	59
5.6.2	Connecting multiple logical expressions . . . . .	60
5.6.3	Translating the <b>in</b> keyword to the pythonic way . . . . .	65
5.6.4	Negating logical conditions . . . . .	65
5.6.5	Filtering <b>null</b> values (i.e. missing data) . . . . .	67
5.6.6	Filtering dates and datetimes in your DataFrame . . . . .	70
5.6.7	Searching for a particular pattern in string values . . . . .	71
5.7	Managing the columns of your DataFrame . . . . .	74
5.7.1	Renaming your columns . . . . .	75
5.7.2	Dropping unnecessary columns . . . . .	76
5.7.3	Casting columns to a different data type . . . . .	77
5.7.4	You can add new columns with <b>select()</b> . . . . .	78
5.8	Calculating or adding new columns to your DataFrame . . . . .	79

5.9	Sorting rows of your DataFrame . . . . .	80
5.10	Calculating aggregates . . . . .	82
5.10.1	Using standard DataFrame methods . . . . .	82
5.10.2	Using the <b>agg()</b> method . . . . .	82
5.10.3	Without groups, we calculate a aggregate of the entire DataFrame . . . . .	83
5.10.4	Calculating aggregates per group in your DataFrame . . . . .	83
<b>6</b>	<b>Importing data to Spark</b>	<b>86</b>
6.1	Introduction . . . . .	86
6.2	Reading data from static files . . . . .	86
6.3	An example with a CSV file . . . . .	87
6.4	Import options . . . . .	89
6.5	Setting the separator character for CSV files . . . . .	91
6.6	Setting the encoding of the file . . . . .	92
6.7	Setting the format of dates and timestamps . . . . .	93
<b>7</b>	<b>Working with SQL in pyspark</b>	<b>96</b>
7.1	Introduction . . . . .	96
7.2	The <b>sql()</b> method as the main entrypoint . . . . .	96
7.2.1	A single SQL statement per run . . . . .	97
7.3	Creating SQL Tables in Spark . . . . .	100
7.3.1	<b>TABLEs</b> versus <b>VIEWS</b> . . . . .	101
7.3.2	Temporary versus Persistent sources . . . . .	107
7.3.3	Spark SQL Catalog is the bridge between SQL and <b>pyspark</b> . . . . .	109
7.4	The <b>penguins</b> DataFrame . . . . .	109
7.5	Selecting your Spark DataFrames . . . . .	110
7.6	Executing SQL expressions . . . . .	111
7.7	Every DataFrame transformation in Python can be translated into SQL . . . . .	113
7.7.1	DataFrame methods are usually translated into SQL keywords . . . . .	113
7.7.2	Spark functions are usually translated into SQL functions . . . . .	115
<b>8</b>	<b>Tools for string manipulation</b>	<b>117</b>
8.1	The <b>logs</b> DataFrame . . . . .	117
8.2	Changing the case of letters in a string . . . . .	118
8.3	Calculating string length . . . . .	120
8.4	Trimming or removing spaces from strings . . . . .	120
8.5	Extracting substrings . . . . .	122
8.5.1	A substring based on a start position and length . . . . .	122
8.5.2	A substring based on a delimiter . . . . .	123
8.5.3	Forming an array of substrings . . . . .	126
8.6	Concatenating multiple strings together . . . . .	128
8.7	Introducing regular expressions . . . . .	130
8.7.1	The Java regular expression standard . . . . .	131

8.7.2	Using an invalid regular expression . . . . .	132
8.7.3	Replacing occurrences of a particular regular expression with <b>regex_replace()</b>	132
8.7.4	Introducing capturing groups on <b>pyspark</b> . . . . .	134
8.7.5	Extracting substrings with <b>regex_extract()</b> . . . . .	137
8.7.6	Identifying values that match a particular regular expression with <b>rlike()</b> . . .	139
<b>References</b>		<b>141</b>
<b>Appendices</b>		<b>141</b>
<b>A Opening the terminal of your OS</b>		<b>142</b>
A.0.1	Opening a terminal on Windows . . . . .	143
A.0.2	Opening a terminal on Linux . . . . .	143
A.0.3	Opening a terminal in MacOS . . . . .	145
<b>B How to install Spark and pyspark</b>		<b>147</b>
B.1	What are the steps? . . . . .	147
B.2	On Windows . . . . .	147
B.2.1	Install Java SDK . . . . .	147
B.2.2	Install Python . . . . .	148
B.2.3	Install <b>pyspark</b> . . . . .	148
B.2.4	Download and extract the files of Apache Spark . . . . .	149
B.2.5	Set environment variables . . . . .	149
B.3	On Linux . . . . .	151
B.3.1	Install <b>pyspark</b> . . . . .	151

# About this website

Welcome! This is the initial page for the “Open Access” HTML version of the book “Introduction to **pyspark**”, written by [Pedro Duarte Faria](#). This book provides an introduction to **pyspark**, which is a python API to [Apache Spark](#).

This book is still under active construction. This means that not all chapters are ready yet, and some of its current contents might change in the close future. This book is licensed by the [CC-BY 4.0 Creative Commons Attribution 4.0 International Public License](#).

# Preface

## Introduction

In essence, **pyspark** is a python package that provides an API for Apache Spark. In other words, with **pyspark** you are able to use the python language to write Spark applications and run them on a Spark cluster in a scalable and elegant way. This book focus on teaching the fundamentals of **pyspark**, and how to use it for big data analysis.

This book, also contains a small introduction to key python concepts that are important to understand how **pyspark** is organized and how it works in practice, and, since we will be using Spark under the hood, is very important to understand a little bit of how Spark works, so, we provide a small introduction to Spark as well.

Big part of the knowledge exposed here is extracted from a lot of practical experience of the author, working with **pyspark** to analyze big data at platforms such as Databricks<sup>1</sup>. Another part of the knowledge is extracted from the official documentation of Apache Spark (*Apache Spark Official Documentation* 2022), as well as some established works such as Chambers and Zaharia (2018) and Damji et al. (2020).

## About the author

Pedro Duarte Faria have a bachelor degree in Economics from Federal University of Ouro Preto - Brazil. Currently, he is a Data Engineer at Take Blip, and an Associate Developer for Apache Spark 3.0 certified by Databricks.

The author have more than 3 years of experience in the data analysis market. He developed data pipelines, reports and analysis for research institutions and some of the largest companies in the brazilian financial sector, such as the BMG Bank, Sodexo and Pan Bank, besides dealing with databases that go beyond the billion rows.

Furthermore, Pedro is specialized on the R programming language, and have given several lectures and courses about it, inside graduate centers (such as PPEA-UFOP<sup>2</sup>), in addition to federal and state

---

<sup>1</sup><https://databricks.com/>

<sup>2</sup><https://ppea.ufop.br/>

organizations (such as FJP-MG<sup>3</sup>). As researcher, he have experience in the field of Science, Technology and Innovation Economics.

Personal Website: <https://pedro-faria.netlify.app/>

Twitter: [@PedroPark9](#)

Mastodon: [@pedropark99@fosstodon.org](#)

## Some conventions of this book

### Python code and terminal commands

This book is about **pyspark**, which is a python package. As a result, we will be exposing a lot of python code across the entire book. Examples of python code, are always shown inside a gray rectangle, like this example below.

Every visible result that this python code produce, will be shown outside of the gray rectangle, just below the command that produced that visible result. Besides that, every line of result will always be written in plain black. So in the example below, the value **729** is the only visible result of this python code, and, the statement **print(y)** is the command that triggered this visible result.

```
x = 3
y = 9 ** x

print(y)
```

**729**

Furthermore, all terminal commands that we expose in this book, will always be: pre-fixed by **Terminal\$**; written in black; and, not outlined by a gray rectangle. In the example below, we can easily detect that this command **pip install jupyter** should be inserted in the terminal of the OS (whatever is the terminal that your OS uses), and not in the python interpreter, because this command is prefixed with **Terminal\$**.

```
#| eval: false
Terminal$ pip install jupyter
```

Some terminal commands may produce visible results as well. In that case, these results will be right below the respective command, and will not be pre-fixed with **Terminal\$**. For example, we can see below that the command **echo "Hello!"** produces the result **"Hello!"**.

---

<sup>3</sup><http://fjp.mg.gov.br/>



```
Terminal$ echo "Hello!"
```

## Python objects, functions and methods

When I refer to some python object, function, method or package, I will use a monospaced font. In other words, if I have a python object called “name”, and, I am describing this object, I will use **name** in the paragraph, and not “name”. The same logic applies to functions, methods and package names.

## Be aware of differences between OS's!

Spark is available for all three main operational systems (or OS's) used in the world (Windows, MacOS and Linux). I will use constantly the word OS as an abbreviation to “operational system”.

The snippets of python code shown throughout this book should just run correctly no matter which one of the three OS's you are using. In other words, the python code snippets are made to be portable. So you can just copy and paste them to your computer, no matter which OS you are using.

But, at some points, I may need to show you some terminal commands that are OS specific, and are not easily portable. For example, Linux have a package manager, but Windows does not have one. This means that, if you are on Linux, you will need to use some terminal commands to install some necessary programs (like python). In contrast, if you are on Windows, you will generally download executable files (**.exe**) that make this installation for you.

In cases like this, I will always point out the specific OS of each one of the commands, or, I will describe the necessary steps to be made on each one the OS's. Just be aware that these differences exists between the OS's.

## Install the necessary software

If you want to follow the examples shown throughout this book, you must have Apache Spark and **pyspark** installed on your machine. If you do not know how to do this, you can consult the contents of Appendix [B](#). This appendix give you a tutorial with step-to-step on how to install these tools on Linux and Windows.

# 1 Key concepts of python

## 1.1 Introduction

If you have experience with python, and understands how objects and classes works, you might want to skip this entire chapter. But, if you are new to the language and do not have much experience with it, you might want to stick a little bit, and learn a few key concepts that will help you to understand how the **pyspark** package is organized, and how to work with it.

## 1.2 Scripts

Python programs are written in plain text files that are saved with the **.py** extension. After you save these files, they are usually called “scripts”. So a script is just a text file that contains all the commands that make your python program.

There are many IDEs or programs that help you to write, manage, run and organize this kind of files (like Microsoft Visual Studio Code<sup>1</sup>, PyCharm<sup>2</sup>, Anaconda<sup>3</sup> and RStudio<sup>4</sup>). Many of these programs are free to use, and, are easy to install.

But, if you do not have any of them installed, you can just create a new plain text file from the built-in Notepad program of your OS (operational system), and, save it with the **.py** extension.

## 1.3 How to run a python program

As you learn to write your Spark applications with **pyspark**, at some point, you will want to actually execute this **pyspark** program, to see its result. To do so, you need to execute it as a python program. There are many ways to run a python program, but I will show you the more “standard” way. That is to use the **python** command inside the terminal of your OS (you need to have python already installed).

---

<sup>1</sup><https://code.visualstudio.com/>

<sup>2</sup><https://www.jetbrains.com/pycharm/>

<sup>3</sup><https://www.anaconda.com/products/distribution>

<sup>4</sup><https://www.rstudio.com/>

As an example, let's create a simple “Hello world” program. First, open a new text file then save it somewhere in your machine (with the name **hello.py**). Remember to save the file with the **.py** extension. Then copy and paste the following command into this file:

```
print("Hello World!")
```

It will be much easier to run this script, if you open the terminal inside the folder where you save the **hello.py** file. If you do not know how to do this, look at section [Appendix A](#). After you opened the terminal inside the folder, just run the **python3 hello.py** command. As a result, python will execute **hello.py**, and, the text **Hello World!** should be printed to the terminal:

```
Terminal$ python3 hello.py
```

**Hello World!**

But, if for some reason you could not open the terminal inside the folder, just open a terminal (in any way you can), then, use the **cd** command (stands for “change directory”) with the path to the folder where you saved **hello.py**. This way, your terminal will be rooted in this folder.

For example, if I saved **hello.py** inside my Documents folder, the path to this folder in Windows would be something like this: **"C:\Users\pedro\Documents"**. On the other hand, this path on Linux would be something like **"/usr/pedro/Documents"**. So the command to change to this directory would be:

```
#| eval: false
# On Windows:
Terminal$ cd "C:\Users\pedro\Documents"
# On Linux:
Terminal$ cd "/usr/pedro/Documents"
```

After this **cd** command, you can run the **python hello.py** command in the terminal, and get the exact same result of the previous example.

There you have it! So every time you need to run your python program (or your **pyspark** program), just open a terminal and run the command **python <complete path to your script>**. If the terminal is rooted on the folder where you saved your script, you can just use the **python <name of the script>** command.

## 1.4 Objects

Although python is a general-purpose language, most of its features are focused on object-oriented programming. Meaning that, python is a programming language focused on creating, managing and modifying objects and classes of objects.

So, when you work with python, you are basically applying many operations and functions over a set of objects. In essence, an object in python, is a name that refers to a set of data. This data can be anything that your computer can store (or represent).

Having that in mind, an object is just a name, and this name is a reference, or a key to access some data. To define an object in python, you must use the assignment operator, which is the equal sign (=). In the example below, we are defining, or, creating an object called **x**, and it stores the value 10. Therefore, with the name **x** we can access this value of 10.

```
x = 10
print(x)
```

**10**

When we store a value inside an object, we can easily reuse this value in multiple operations or expressions:

```
# Multiply by 2
print(x * 2)
```

**20**

```
# Divide by 3
print(x / 3)
```

**3.3333333333333335**

```
# Print its class
print(type(x))
```

**<class 'int'>**

Remember, an object can store any type of value, or any type of data. For example, it can store a single string, like the object **salutation** below:

```
salutation = "Hello! My name is Pedro"
```

Or, a list of multiple strings:

```
names = [  
    "Anne", "Vanse", "Elliot",  
    "Carlyle", "Ed", "Memphis"  
]  
  
print(names)
```

```
['Anne', 'Vanse', 'Elliot', 'Carlyle', 'Ed', 'Memphis']
```

Or a dict containing the description of a product:

```
product = {  
    'name': 'Coca Cola',  
    'volume': '2 liters',  
    'price': 2.52,  
    'group': 'non-alcoholic drinks',  
    'department': 'drinks'  
}  
  
print(product)
```

```
{'name': 'Coca Cola', 'volume': '2 liters', 'price': 2.52  
, 'group': 'non-alcoholic drinks', 'department': 'drinks'}
```

And many other things...

## 1.5 Expressions

Python programs are organized in blocks of expressions (or statements). A python expression is a statement that describes an operation to be performed by the program. For example, the expression below describes the sum between 3 and 5.

```
3 + 5
```

8

The expression above is composed of numbers (like 3 and 5) and a operator, more specifically, the sum operator (+). But any python expression can include a multitude of different items. It can be composed of functions (like **print()**, **map()** and **str()**), constant strings (like **"Hello World!"**), logical operators (like **!=**, **<**, **>** and **==**), arithmetic operators (like **\***, **/**, **\*\***, **%**, **-** and **+**), structures (like lists, arrays and dicts) and many other types of commands.

Below we have a more complex example, that contains the **def** keyword (which starts a function definition; in the example below, this new function being defined is **double()**), many built-in functions (**list()**, **map()** and **print()**), a arithmetic operator (**\***), numbers and a list (initiated by the pair of brackets - **[]**).

```
def double(x):  
    return x * 2  
  
print(list(map(double, [4, 2, 6, 1])))
```

**[8, 4, 12, 2]**

Python expressions are evaluated in a sequential manner (from top to bottom of your python file). In other words, python runs the first expression in the top of your file, then, goes to the second expression, and runs it, then goes to the third expression, and runs it, and goes on and on in that way, until it hits the end of the file. So, in the example above, python executes the function definition (initiated at **def double(x):**), before it executes the **print()** statement, because the print statement is below the function definition.

This order of evaluation is commonly referred as “control flow” in many programming languages. Sometimes, this order can be a fundamental part of the python program. Meaning that, sometimes, if we change the order of the expressions in the program, we can produce unexpected results (like an error), or change the results produced by the program.

As an example, the program below prints the result 4, because the print statement is executed before the expression **x = 40**.

```
x = 1  
  
print(x * 4)  
  
x = 40
```

**4**

But, if we execute the expression **x = 40** before the print statement, we then change the result produced by the program.

```
x = 1
x = 40

print(x * 4)
```

160

If we go a little further, and, put the print statement as the first expression of the program, we then get a name error. This error warns us that, the object named **x** is not defined (i.e. it does not exist).

```
print(x * 4)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

```
x = 1
x = 40
```

This error occurs, because inside the print statement, we call the name **x**. But, this is the first expression of the program, and at this point of the program, we did not defined a object called **x**. We make this definition, after the print statement, with **x = 1** and **x = 40**. In other words, at this point, python do not know any object called **x**.

## 1.6 Packages (or libraries)

A python package (or a python “library”) is basically a set of functions and classes that provides important functionality to solve a specific problem. And **pyspark** is one of these many python packages available.

Python packages are usually published (that is, made available to the public) through the PyPI archive<sup>5</sup>. If a python package is published in PyPI, then, you can easily install it through the **pip** tool, that we just used in Appendix B.

To use a python package, you always need to: 1) have this package installed on your machine; 2) import this package in your python script. If a package is not installed in your machine, you will face a **ModuleNotFoundError** as you try to use it, like in the example below.

---

<sup>5</sup><https://pypi.org/>

```
import pandas
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ModuleNotFoundError: No module named 'pandas'

If your program produce this error, is very likely that you are trying to use a package that is not currently installed on your machine. To install it, you may use the **pip install <name of the package>** command on the terminal of your OS.

```
{terminal, eval = FALSE} pip install pandas
```

But, if this package is already installed in your machine, then, you can just import it to your script. To do this, you just include an **import** statement at the start of your python file. For example, if I want to use the **DataFrame** function from the **pandas** package:

```
# Now that I installed the 'pandas' package with 'pip'
# this 'import' statement works without any errors:
import pandas

df = pandas.DataFrame([
    (1, 3214), (2, 4510),
    (1, 9082), (4, 7822)
])

print(df)
```

```
   0    1
0  1  3214
1  2  4510
2  1  9082
3  4  7822
```

Therefore, with **import pandas** I can access any of the functions available in the **pandas** package, by using the dot operator after the name of the package (**pandas.<name of the function>**). However, it can become very annoying to write **pandas.** every time you want to access a function from **pandas**, specially if you use it constantly in your code.

To make life a little easier, python offers some alternative ways to define this **import** statement. First, you can give an alias to this package that is shorter/easier to write. As an example, nowadays, is virtually a industry standard to import the **pandas** package as **pd**. To do this, you use the **as** keyword



in your **import** statement. This way, you can access the **pandas** functionality with **pd.<name of the function>**:

```
import pandas as pd

df = pd.DataFrame([
    (1, 3214), (2, 4510),
    (1, 9082), (4, 7822)
])

print(df)
```

	0	1
0	1	3214
1	2	4510
2	1	9082
3	4	7822

In contrast, if you want to make your life even easier and produce a more “clean” code, you can import (from the package) just the functions that you need to use. In this method, you can eliminate the dot operator, and refer directly to the function by its name. To use this method, you include the **from** keyword in your import statement, like this:

```
from pandas import DataFrame

df = DataFrame([
    (1, 3214), (2, 4510),
    (1, 9082), (4, 7822)
])

print(df)
```

	0	1
0	1	3214
1	2	4510
2	1	9082
3	4	7822

Just to be clear, you can import multiple functions from the package, by listing them. Or, if you prefer, you can import all components of the package (or module/sub-module) by using the star shortcut (**\***):

```
# Import 'search()', 'match()' and 'compile()' functions:
from re import search, match, compile
# Import all functions from the 'os' package
from os import *
```

Some packages may be very big, and includes many different functions and classes. As the size of the package becomes bigger and bigger, developers tend to divide this package in many “modules”. In other words, the functions and classes of this python package are usually organized in “modules”.

As an example, the **pyspark** package is a fairly large package, that contains many classes and functions. Because of it, the package is organized in a number of modules, such as **sql** (to access Spark SQL), **pandas** (to access the Pandas API of Spark), **ml** (to access Spark MLlib).

To access the functions available in each one of these modules, you use the dot operator between the name of the package and the name of the module. For example, to import all components from the **sql** and **pandas** modules of **pyspark**, you would do this:

```
from pyspark.sql import *
from pyspark.pandas import *
```

Going further, we can have sub-modules (or modules inside a module) too. As an example, the **sql** module of **pyspark** have the **functions** and **window** sub-modules. To access these sub-modules, you use the dot operator again:

```
# Importing 'functions' and 'window' sub-modules:
import pyspark.sql.functions as F
import pyspark.sql.window as W
```

## 1.7 Methods versus Functions

Beginners tend mix these two types of functions in python, but they are not the same. So lets describe the differences between the two.

Standard python functions, are **functions that we apply over an object**. A classical example, is the **print()** function. You can see in the example below, that we are applying **print()** over the **result** object.

```
result = 10 + 54
print(result)
```

Other examples of a standard python function would be **map()** and **list()**. See in the example below, that we apply the **map()** function over a set of objects:

```
words = ['apple', 'star', 'abc']
lengths = map(len, words)
list(lengths)
```

[5, 4, 3]

In contrast, a python method is a function registered inside a python class. In other words, this function **belongs to the class itself**, and cannot be used outside of it. This means that, in order to use a method, you need to have an instance of the class where it is registered.

For example, the **startswith()** method belongs to the **str** class (this class is used to represent strings in python). So to use this method, we need to have an instance of this class saved in a object that we can access. Note in the example below, that we access the **startswith()** method through the **name** object. This means that, **startswith()** is a function. But, we cannot use it without an object of class **str**, like **name**.

```
name = "Pedro"
name.startswith("P")
```

True

Note in the example above, that we access any class method in the same way that we would access a sub-module/module of a package. That is, by using the dot operator (**.**).

So, if we have a class called **people**, and, this class has a method called **location()**, we can use this **location()** method by using the dot operator (**.**) with the name of an object of class **people**. If an object called **x** is an instance of **people** class, then, we can do **x.location()**.

But if this object **x** is of a different class, like **int**, then we can no longer use the **location()** method, because this method does not belong to the **int** class. For example, if your object is from class **A**, and, you try to use a method of class **B**, you will get an **AttributeError**.

In the example exposed below, I have an object called **number** of class **int**, and, I try to use the method **startswith()** from **str** class with this object:

```
number = 2
# You can see below that, the 'x' object have class 'int'
type(number)
# Trying to use a method from 'str' class
```

```
number.startswith("P")
```

**AttributeError: 'int' object has no attribute 'startswith'**

## 1.8 Identifying classes and their methods

Over the next chapters, you will realize that **pyspark** programs tend to use more methods than standard functions. So most of the functionality of **pyspark** resides in class methods. As a result, the capability of understanding the objects that you have in your python program, and, identifying its classes and methods will be crucial while you are developing and debugging your Spark applications.

Every existing object in python represents an instance of a class. In other words, every object in python is associated to a given class. You can always identify the class of an object, by applying the **type()** function over this object. In the example below, we can see that, the **name** object is an instance of the **str** class.

```
name = "Pedro"  
type(name)
```

**str**

If you do not know all the methods that a class have, you can always apply the **dir()** function over this class to get a list of all available methods. For example, lets suppose you wanted to see all methods from the **str** class. To do so, you would do this:

```
dir(str)
```

```
[ '__add__', '__class__', '__contains__', '__delattr__', '__dir__',  
  '__doc__', '__eq__', '__format__', '__ge__', '__getattr__',  
  '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',  
  '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',  
  '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',  
  '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__',  
  '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center',  
  'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',  
  'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',  
  'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',  
  'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',  
  'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
```

```
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',  
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

## 2 Introducing Apache Spark

### 2.1 Introduction

In essence, **pyspark** is an API to Apache Spark (or simply Spark). In other words, with **pyspark** we can build Spark applications using the python language. So, by learning a little more about Spark, you will understand a lot more about **pyspark**.

### 2.2 What is Spark?

Spark is a multi-language engine for large-scale data processing that supports both single-node machines and clusters of machines (*Apache Spark Official Documentation* 2022). Nowadays, Spark became the de facto standard for structure and manage big data applications.

It has a number of features that its predecessors did not have, like the capacity for in-memory processing and stream processing (Karau et al. 2015). But, the most important feature of all, is that Spark is an **unified platform** for big data processing (Chambers and Zaharia 2018).

This means that Spark comes with multiple built-in libraries and tools that deals with different aspects of the work with big data. It has a built-in SQL engine<sup>1</sup> for performing large-scale data processing; a complete library for scalable machine learning (**MLib**<sup>2</sup>); a stream processing engine<sup>3</sup> for streaming analytics; and much more;

In general, big companies have many different data necessities, and as a result, the engineers and analysts may have to combine and integrate many tools and techniques together, so they can build many different data pipelines to fulfill these necessities. But this approach can create a very serious dependency problem, which imposes a great barrier to support this workflow. This is one of the big reasons why Spark got so successful. It eliminates big part of this problem, by already including almost everything that you might need to use.

Spark is designed to cover a wide range of workloads that previously required separate distributed systems ... By supporting these workloads in the same engine, Spark makes it easy and inexpensive to combine different processing types, which is often necessary

---

<sup>1</sup><https://spark.apache.org/sql/>

<sup>2</sup><https://spark.apache.org/docs/latest/ml-guide.html>

<sup>3</sup><https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#overview>

in production data analysis pipelines. In addition, it reduces the management burden of maintaining separate tools (Karau et al. 2015).

## 2.3 Spark application

Your personal computer can do a lot of things, but, it cannot efficiently deal with huge amounts of data. For this situation, we need several machines working together, adding up their resources to deal with the volume or complexity of the data. Spark is the framework that coordinates the computations across this set of machines (Chambers and Zaharia 2018). Because of this, a relevant part of Spark's structure is deeply connected to distributed computing models.

You probably do not have a cluster of machines at home. So, while following the examples in this book, you will be running Spark on a single machine (i.e. single node mode). But let's just forget about this detail for a moment.

In every Spark application, you always have a single machine behaving as the driver node, and multiple machines behaving as the worker nodes. The driver node is responsible for managing the Spark application, i.e. asking for resources, distributing tasks to the workers, collecting and compiling the results, .... The worker nodes are responsible for executing the tasks that are assigned to them, and they need to send the results of these tasks back to the driver node.

Every Spark application is distributed into two different and independent processes: 1) a driver process; 2) and a set of executor processes (Chambers and Zaharia 2018). The driver process, or, the driver program, is where your application starts, and it is executed by the driver node. This driver program is responsible for: 1) maintaining information about your Spark Application; 2) responding to a user's program or input; 3) and analyzing, distributing, and scheduling work across the executors (Chambers and Zaharia 2018).

Every time a Spark application starts, the driver process has to communicate with the cluster manager, to acquire workers to perform the necessary tasks. In other words, the cluster manager decides if Spark can use some of the resources (i.e. some of the machines) of the cluster. If the cluster manager allows Spark to use the nodes it needs, the driver program will break the application into many small tasks, and will assign these tasks to the worker nodes.

The executor processes, are the processes that take place within each one of the worker nodes. Each executor process is composed of a set of tasks, and the worker node is responsible for performing and executing these tasks that were assigned to him, by the driver program. After executing these tasks, the worker node will send the results back to the driver node (or the driver program). If they need, the worker nodes can communicate with each other, while performing its tasks.

This structure is represented in Figure 2.1:

When you run Spark on a cluster of computers, you write the code of your Spark application (i.e. your **pyspark** code) on your (single) local computer, and then, submit this code to the driver node. After that, the driver node takes care of the rest, by starting your application, creating your Spark Session,

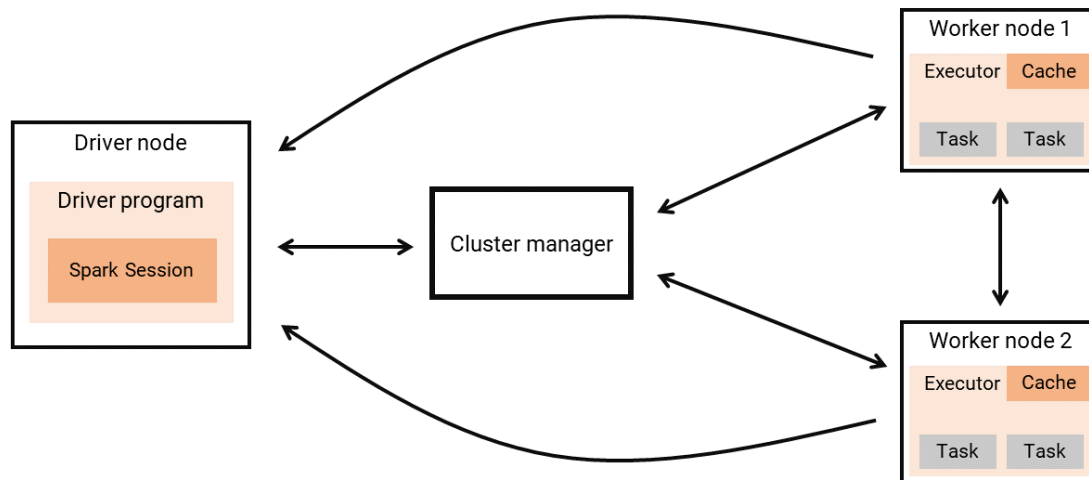


Figure 2.1: Spark application structure on a cluster of computers

asking for new worker nodes, sending the tasks to be performed, collecting and compiling the results and giving back these results to you.

However, when you run Spark on your (single) local computer, the process is very similar. But, instead of submitting your code to another computer (which is the driver node), you will submit to your own local computer. In other words, when Spark is running on single-node mode, your computer becomes the driver and the worker node at the same time.

## 2.4 Spark application versus pyspark application

The **pyspark** package is just a tool to write Spark applications using the python programming language. This means, that every **pyspark** application is a Spark application written in python.

With this conception in mind, you can understand that a **pyspark** application is a description of a Spark application. When we compile (or execute) our python program, this description is translated into a raw Spark application that will be executed by Spark.

To write a **pyspark** application, you write a python script that uses the **pyspark** library. When you execute this python script with the python interpreter, the application will be automatically converted to Spark code, and will be sent to Spark to be executed across the cluster;



## 2.5 Core parts of a pyspark program

In this section, I want to point out the core parts that composes every **pyspark** program. This means that every **pyspark** program that you write will have these “core parts”, which are:

- 1) importing the Spark libraries (or packages);
- 2) starting your Spark Session;
- 3) defining a set of transformations and actions over Spark DataFrames;

### 2.5.1 Importing the Spark libraries (or packages)

Spark comes with a lot of functionality installed. But, in order to use it in your **pyspark** program, you have to import most of these functionalities to your session. This means that you have to import specific packages (or “modules”) of **pyspark** to your python session.

For example, most of the functions used to define our transformations and aggregations in Spark DataFrames, comes from the **pyspark.sql.functions** module.

That is why we usually start our python scripts by importing functions from this module, like this:

```
from pyspark.sql.functions import sum, col
sum_expr = sum(col('Value'))
```

Or, importing the entire module with the **import** keyword, like this:

```
import pyspark.sql.functions as F
sum_expr = F.sum(F.col('Value'))
```

### 2.5.2 Starting your Spark Session

Every Spark application starts with a Spark Session. Basically, the Spark Session is the entry point to your application. This means that, in every **pyspark** program that you write, **you should always start by defining your Spark Session**. We do this, by using the **getOrCreate()** method from **pyspark.sql.Session.builder** module.

Just store the result of this method in any python object. Is very common to name this object as **spark**, like in the example below. This way, you can access all the information and methods of Spark from this **spark** object.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
```

### 2.5.3 Defining a set of transformations and actions

Every **pyspark** program is composed by a set of transformations and actions over a set of Spark DataFrames.

We will explain Spark DataFrames in more depth on the Chapter 3. For now just understand that they are the basic data structure that feed all **pyspark** programs. In other words, on every **pyspark** program we are transforming multiple Spark DataFrames to get the result we want.

As an example, in the script below we begin with the Spark DataFrame stored in the object **students**, and, apply multiple transformations over it to build the **ar\_department** DataFrame. Lastly, we apply the **.show()** action over the **ar\_department** DataFrame:

```
from pyspark.sql.functions import col
# Apply some transformations over
# the `students` DataFrame:
ar_department = students\
    .filter(col('Age') > 22)\
    .withColumn('IsArDepartment', col('Department') == 'AR')\
    .orderBy(col('Age').desc())

# Apply the `.show()` action
# over the `ar_department` DataFrame:
ar_department.show()
```

## 2.6 Building your first Spark application

To demonstrate what a **pyspark** program looks like, let's write and run our first example of a Spark application. This Spark application will build a simple table of 1 column that contains 5 numbers, and then, it will return a simple python list containing this five numbers as a result.

### 2.6.1 Writing the code

First, create a new blank text file in your computer, and save it somewhere with the name **spark-example.py**. Do not forget to put the **.py** extension in the name. This program we are writing together is a python program, and should be treated as such. With the **.py** extension in the name file, you are stating this fact quite clearly to your computer.

After you created and saved the python script (i.e. the text file with the **.py** extension), you can start writing your **pyspark** program. As we noted in the previous section, you should always start your **pyspark** program by defining your Spark Session, with this code:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
```

After you defined your Spark Session, and saved it in an object called **spark**, you can now access all Spark's functionality through this **spark** object.

To create our first Spark table we use the **range()** method from the **spark** object. The **range()** method works similarly as the standard python function called **range()**. It basically creates a sequence of numbers, from 0 to  $n - 1$ . However, this **range()** method from **spark** stores this sequence of numbers as rows in a Spark table (or a Spark DataFrame):

```
table = spark.range(5)
```

After this step, we want to collect all the rows of the resulting table into a python list. And to do that, we use the **collect()** method from the Spark table:

```
result = table.collect()
print(result)
```

So, the entire program is composed of these three parts (or sections) of code. If you need it, the entire program is reproduced below. You can copy and paste all of this code to your python script, and then, save it:

```
# The entire program:
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

table = spark.range(5)
result = table.collect()
print(result)
```

## 2.6.2 Executing the code

Now that you have written your first Spark application with **pyspark**, you want to execute this application and see its results. Yet, to run a **pyspark** program, remember that you need to have the necessary software installed on your machine. I talk about how to install these software at [Appendix B](#).

Anyway, to execute this **pyspark** that you wrote, you need send this script to the python interpreter, and to do this you need to: 1) open a terminal inside the folder where you python script is stored; and, 2) use the python command from the terminal with the name of your python script.

If you do not know how to open a terminal from inside a folder of your computer, you can consult Appendix [A](#) of this book, where I teach you how to do it.

In my current situation, I running Spark on a Ubuntu distribution, and, I saved the **spark-example.py** script inside a folder called **SparkExample**. This folder is located at the path **~/Documentos/Projetos/Livros/Introd-pys** of my computer. This means that, I need to open a terminal that is rooted inside this **SparkExample** folder.

You probably have saved your **spark-example.py** file in a different folder of your computer. This means that you need to open the terminal from a different folder.

After I opened a terminal rooted inside the **SparkExample** folder. I just use the **python3** command to access the python interpreter, and, give the name of the python script that I want to execute. In this case, the **spark-example.py** file. As a result, our first **pyspark** program will be executed:

```
Terminal$ python3 spark-example.py
```

```
[Row(id=0), Row(id=1), Row(id=2), Row(id=3), Row(id=4)]
```

You can see in the above result, that this Spark application produces a sequence of numbers, from 0 to 4, and, returns this sequence as a set of **Row** objects, inside a python list.

Congratulations! You have just run your first Spark application using **pyspark**!

## 2.7 Overview of pyspark

Before we continue, I want to give you a very brief overview of the main parts of **pyspark** that are the most useful and most important to know of.

### 2.7.1 Main python modules

The main python modules that exists in **pyspark** are:

- **pyspark.sql.SparkSession**: the **SparkSession** class that defines your Spark Session, or, the entry point to your Spark application;
- **pyspark.sql.dataframe**: module that defines the **DataFrame** class;
- **pyspark.sql.column**: module that defines the **Column** class;
- **pyspark.sql.types**: module that contains all data types of Spark;

- **pyspark.sql.functions**: module that contains all of the main Spark functions that we use in transformations;
- **pyspark.sql.window**: module that defines the **Window** class, which is responsible for defining windows in a Spark DataFrame;

## 2.7.2 Main python classes

The main python classes that exists in **pyspark** are:

- **DataFrame**: represents a Spark DataFrame, and it is the main data structure in **pyspark**. In essence, they represent a collection of datasets into named columns;
- **Column**: represents a column in a Spark DataFrame;
- **GroupedData**: represents a grouped Spark DataFrame (result of **DataFrame.groupby()**);
- **Window**: describes a window in a Spark DataFrame;
- **DataFrameReader** and **DataFrameWriter**: classes responsible for reading data from a data source into a Spark DataFrame, and writing data from a Spark DataFrame into a data source;
- **DataFrameNaFunctions**: class that stores all main methods for dealing with null values (i.e. missing data);

## 3 Introducing Spark DataFrames

### 3.1 Introduction

In this chapter, you will understand how Spark represents and manages tables (or tabular data). Different programming languages and frameworks use different names to describe a table. But, in Apache Spark, they are referred as Spark DataFrames.

In **pyspark**, these DataFrames are stored inside python objects of class **pyspark.sql.dataframe.DataFrame**, and all the methods present in this class, are commonly referred as the DataFrame API of Spark. This is the most important API of Spark, because much of your Spark applications will heavily use this API to compose your data transformations and data flows (Chambers and Zaharia 2018).

### 3.2 Spark DataFrames versus Spark Datasets

Spark have two notions of structured data: DataFrames and Datasets. In summary, a Spark Dataset, is a distributed collection of data (*Apache Spark Official Documentation* 2022). In contrast, a Spark DataFrame is a Spark Dataset organized into named columns (*Apache Spark Official Documentation* 2022).

This means that, Spark DataFrames are very similar to tables as we know in relational databases - RDBMS, or, in spreadsheets (like Excel). So in a Spark DataFrame, each column has a name, and they all have the same number of rows. Furthermore, all the rows inside a column must store the same type of data, but each column can store a different type of data.

In the other hand, Spark Datasets are considered a collection of any type of data. So a Dataset might be a collection of unstructured data as well, like log files, JSON and XML trees, etc. Spark Datasets can be created and transformed through the Dataset API of Spark. But this API is available only in Scala and Java API's of Spark. For this reason, we do not act directly on Datasets with **pyspark**, only DataFrames. That's ok, because for the most part of applications, we do want to use DataFrames, and not Datasets, to represent our data.

However, what makes a Spark DataFrame different from other dataframes? Like the **pandas** DataFrame? Or the R native **data.frame** structure? Is the **distributed** aspect of it. Spark DataFrames are based on Spark Datasets, and these Datasets are collections of data that are distributed across the cluster. As an example, let's suppose you have the following table stored as a Spark DataFrame:

ID	Name	Value
1	Anne	502
2	Carls	432
3	Stoll	444
4	Percy	963
5	Martha	123
6	Sigrid	621

If you are running Spark in a 4 nodes cluster (one is the driver node, and the other three are worker nodes). Each worker node of the cluster will store a section of this data. So you, as the programmer, will see, manage and transform this table as if it was a single and unified table. But behind the hoods, Spark will split this data and store it as many fragments across the Spark cluster. Figure 3.1 presents this notion in a visual manner.

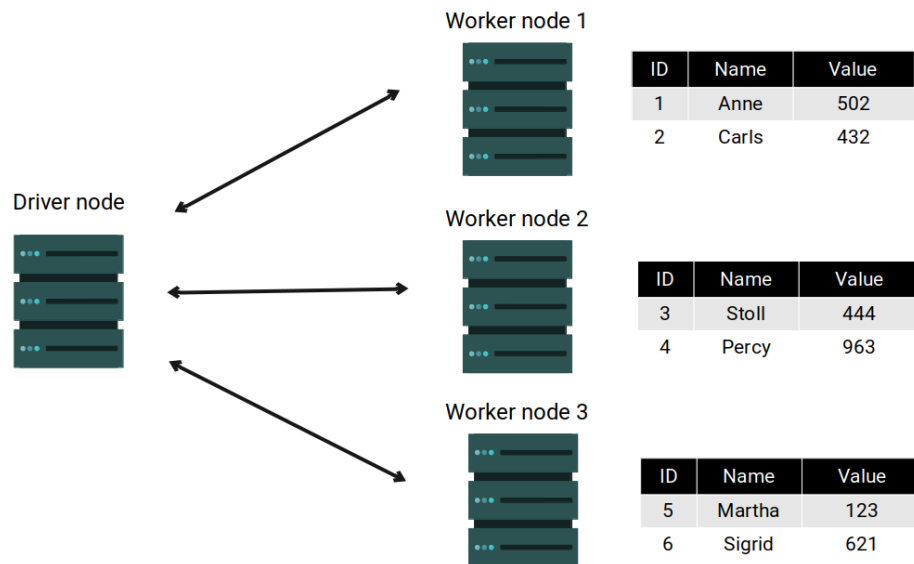


Figure 3.1: A Spark DataFrame is distributed across the cluster

### 3.3 Partitions of a Spark DataFrame

A Spark DataFrame is always broken into many small pieces, and, these pieces are always spread across the cluster of machines. Each one of these small pieces of the total data are considered a DataFrame *partition*.

For the most part, you do not manipulate these partitions manually or individually (Karau et al. 2015), because Spark automatically do this job for you.

As we exposed in Figure 3.1, each node of the cluster will hold a piece of the total DataFrame. If we translate this distribution into a “partition” distribution, this means that each node of the cluster can hold one or multiple partitions of the Spark DataFrame.

If we sum all partitions present in a node of the cluster, we get a chunk of the total DataFrame. The figure below demonstrates this notion:

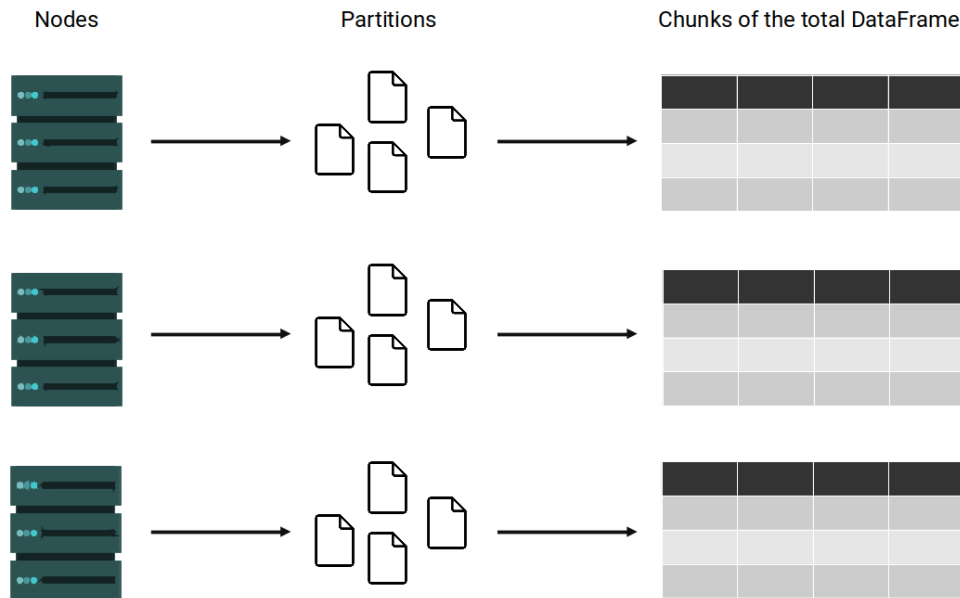


Figure 3.2: Partitions of a DataFrame

If the Spark DataFrame is not big, each node of the cluster will probably store just a single partition of this DataFrame. In contrast, depending on the complexity and size of the DataFrame, Spark will split this DataFrame into more partitions than there are nodes in the cluster. In this case, each node of the cluster will hold more than 1 partition of the total DataFrame.

### 3.4 The DataFrame class in pyspark

In **pyspark**, every Spark DataFrame is stored inside a python object of class **pyspark.sql.dataframe.DataFrame**. Or more succinctly, a object of class **DataFrame**.



Like any python class, the **DataFrame** class comes with multiple methods that are available for every object of this class. This means that you can use any of these methods in any Spark DataFrame that you create through **pyspark**.

As an example, in the code below I expose all the available methods from this **DataFrame** class. First, I create a Spark DataFrame with **spark.range(5)**, and, store it in the object **df5**. After that, I use the **dir()** function to show all the methods that I can use through this **df5** object:

```
df5 = spark.range(5)
available_methods = dir(df5)
print(available_methods)
```

```
['_class__', '__delattr__', '__dict__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getattribute__', '__getitem__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__le__',
 '__lt__', '__module__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
 '_collect_as_arrow', '_jcols', '_jdf', '_jmap',
 '_joinAsOf', '_jseq', '_lazy_rdd', '_repr_html_',
 '_sc', '_schema', '_session', '_sort_cols',
 '_sql_ctx', '_support_repr_html', '_to_corrected_pandas_type', 'agg',
 'alias', 'approxQuantile', 'cache', 'checkpoint',
 'coalesce', 'colRegex', 'collect', 'columns',
 'corr', 'count', 'cov', 'createGlobalTempView',
 'createOrReplaceGlobalTempView', 'createOrReplaceTempView', 'createTempView',
 'crossJoin',
 'crosstab', 'cube', 'describe', 'distinct',
 'drop', 'dropDuplicates', 'drop_duplicates', 'dropna',
 'dtypes', 'exceptAll', 'explain', 'fillna',
 'filter', 'first', 'foreach', 'foreachPartition',
 'freqItems', 'groupBy', 'groupby', 'head',
 'hint', 'inputFiles', 'intersect', 'intersectAll',
 'isEmpty', 'isLocal', 'isStreaming', 'is_cached',
 'join', 'limit', 'localCheckpoint', 'mapInArrow',
 'mapInPandas', 'na', 'observe', 'orderBy',
 'pandas_api', 'persist', 'printSchema', 'randomSplit',
 'rdd', 'registerTempTable', 'repartition', 'repartitionByRange',
 'replace', 'rollup', 'sameSemantics', 'sample',
 'sampleBy', 'schema', 'select', 'selectExpr',
 'semanticHash', 'show', 'sort', 'sortWithinPartitions',
```

```
'sparkSession', 'sql_ctx', 'stat', 'storageLevel',
'subtract', 'summary', 'tail', 'take',
'toDF', 'toJSON', 'toLocalIterator', 'toPandas',
'to_koalas', 'to_pandas_on_spark', 'transform', 'union',
'unionAll', 'unionByName', 'unpersist', 'where',
'withColumn', 'withColumnRenamed', 'withColumns', 'withMetadata',
'withWatermark', 'write', 'writeStream', 'writeTo'],
```

All the methods present in this **DataFrame** class, are commonly referred as the *DataFrame API of Spark*. Remember, this is the most important API of Spark. Because much of your Spark applications will heavily use this API to compose your data transformations and data flows (Chambers and Zaharia 2018).

### 3.5 Building a Spark DataFrame

There are some different methods to create a Spark DataFrame. For example, because a DataFrame is basically a Dataset of rows, we can build a DataFrame from a collection of **Row**'s, through the **createDataFrame()** method from your Spark Session:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
from datetime import date
from pyspark.sql import Row

data = [
    Row(id = 1, value = 28.3, date = date(2021,1,1)),
    Row(id = 2, value = 15.8, date = date(2021,1,1)),
    Row(id = 3, value = 20.1, date = date(2021,1,2)),
    Row(id = 4, value = 12.6, date = date(2021,1,3))
]

df = spark.createDataFrame(data)
```

Remember that a Spark DataFrame in python is a object of class **pyspark.sql.dataframe.DataFrame** as you can see below:

```
type(df)
```

```
pyspark.sql.dataframe.DataFrame
```

If you try to see what is inside of this kind of object, you will get a small description of the columns present in the DataFrame as a result:

```
df
```

```
DataFrame[id: bigint, value: double, date: date]
```

So, in the above example, we use the **Row()** constructor (from **pyspark.sql** module) to build 4 rows. The **createDataFrame()** method, stack these 4 rows together to form our new DataFrame **df**. The result is a Spark DataFrame with 4 rows and 3 columns (**id**, **value** and **date**).

But you can use different methods to create the same Spark DataFrame. As another example, with the code below, we are creating a DataFrame called **students** from two different python lists (**data** and **columns**).

The first list (**data**) is a list of rows. Each row is represent by a python tuple, which contains the values in each column. But the secont list (**columns**) contains the names for each column in the DataFrame.

To create the **students** DataFrame we deliver these two lists to **createDataFrame()** method:

```
data = [
    (12114, 'Anne', 21, 1.56, 8, 9, 10, 9, 'Economics', 'SC'),
    (13007, 'Adrian', 23, 1.82, 6, 6, 8, 7, 'Economics', 'SC'),
    (10045, 'George', 29, 1.77, 10, 9, 10, 7, 'Law', 'SC'),
    (12459, 'Adeline', 26, 1.61, 8, 6, 7, 7, 'Law', 'SC'),
    (10190, 'Mayla', 22, 1.67, 7, 7, 7, 9, 'Design', 'AR'),
    (11552, 'Daniel', 24, 1.75, 9, 9, 10, 9, 'Design', 'AR')
]

columns = [
    'StudentID', 'Name', 'Age', 'Height', 'Score1',
    'Score2', 'Score3', 'Score4', 'Course', 'Department'
]

students = spark.createDataFrame(data, columns)
students
```

```
DataFrame[StudentID: bigint, Name: string, Age: bigint, Height: double
, Score1: bigint, Score2: bigint, Score3: bigint, Score4: bigint, Course: string
, Department: string]
```

You can also use a method that returns a **DataFrame** object by default. Examples are the **table()** and **range()** methods from your Spark Session, like we used in the Section 3.4, to create the **df5** object.

Other examples are the methods used to read data and import it to **pyspark**. These methods are available in the **spark.read** module, like **spark.read.csv()** and **spark.read.json()**. These methods will be described in more depth in Chapter 6.

## 3.6 Viewing a Spark DataFrame

A key aspect of Spark is its laziness. In other words, for most operations, Spark will only check if your code is correct and if it makes sense. Spark will not actually run or execute the operations you are describing in your code, unless you explicitly ask for it with a trigger operation, which is called an “action” (this kind of operation is described in Section 5.3).

You can notice this laziness in the output below:

```
students
```

```
DataFrame[StudentID: bigint, Name: string, Age: bigint, Height: double
, Score1: bigint, Score2: bigint, Score3: bigint, Score4: bigint, Course: string
, Department: string]
```

Because when we call for an object that stores a Spark DataFrame (like **df** and **students**), Spark will only calculate and print a summary of the structure of your Spark DataFrame, and not the DataFrame itself.

So how can we actually see our DataFrame? How can we visualize the rows and values that are stored inside of it? For this, we use the **show()** method. With this method, Spark will print the table as pure text, as you can see in the example below:

```
students.show()
```

```
[Stage 0:>
```

```
(0 + 1) / 1]
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|StudentID|  Name|Age|Height|Score1|Score2|Score3|Score4|  Course|Department|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|   12114|  Anne| 21|  1.56|    8|    9|   10|    9|Economics|      SC|
|   13007| Adrian| 23|  1.82|    6|    6|    8|    7|Economics|      SC|
|   10045| George| 29|  1.77|   10|    9|   10|    7|      Law|      SC|
|   12459|Adeline| 26|  1.61|    8|    6|    7|    7|      Law|      SC|
```

	10190	Mayla	22	1.67	7	7	7	9	Design	AR
	11552	Daniel	24	1.75	9	9	10	9	Design	AR

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

By default, this method shows only the top rows of your DataFrame, but you can specify how much rows exactly you want to see, by using **show(n)**, where **n** is the number of rows. For example, I can visualize only the first 2 rows of **df** like this:

```
df.show(2)
```

```
+---+-----+-----+
| id|value|      date|
+---+-----+-----+
|  1| 28.3|2021-01-01|
|  2| 15.8|2021-01-01|
+---+-----+-----+
```

only showing top 2 rows

### 3.7 Getting the name of the columns

If you need to, you can easily collect a python list with the column names present in your DataFrame, in the same way you would do in a **pandas** DataFrame. That is, by using the **columns** method of your DataFrame, like this:

```
students.columns
```

```
['StudentID',
 'Name',
 'Age',
 'Height',
 'Score1',
 'Score2',
 'Score3',
 'Score4',
 'Course',
 'Department']
```

## 3.8 Spark Data Types

Each column of your Spark DataFrame is associated with a specific data type. Spark supports a large number of different data types. You can see the full list at the official documentation page<sup>1</sup>. For now, we will focus on the most used data types, which are listed below:

- **IntegerType**: Represents 4-byte signed integer numbers. The range of numbers that it can represent is from -2147483648 to 2147483647.
- **LongType**: Represents 8-byte signed integer numbers. The range of numbers that it can represent is from -9223372036854775808 to 9223372036854775807.
- **FloatType**: Represents 4-byte single-precision floating point numbers.
- **DoubleType**: Represents 8-byte double-precision floating point numbers.
- **StringType**: Represents character string values.
- **BooleanType**: Represents boolean values (true or false).
- **TimestampType**: Represents datetime values, i.e. values that contains fields year, month, day, hour, minute, and second, with the session local time-zone. The timestamp value represents an absolute point in time.
- **DateType**: Represents date values, i.e. values that contains fields year, month and day, without a time-zone.

Besides these more “standard” data types, Spark supports two other complex types, which are **ArrayType** and **MapType**:

- **ArrayType(elementType, containsNull)**: Represents a sequence of elements with the type of **elementType**. **containsNull** is used to indicate if elements in a **ArrayType** value can have **null** values.
- **MapType(keyType, valueType, valueContainsNull)**: Represents a set of key-value pairs. The data type of keys is described by **keyType** and the data type of values is described by **valueType**. For a **MapType** value, keys are not allowed to have **null** values. **valueContainsNull** is used to indicate if values of a **MapType** value can have **null** values.

Each one of these Spark data types have a corresponding python class in **pyspark**, which are stored in the **pyspark.sql.types** module. As a result, to access, lets say, type **StringType**, we can do this:

```
from pyspark.sql.types import StringType
s = StringType()
print(s)
```

**StringType()**

---

<sup>1</sup>The full list is available at the link <https://spark.apache.org/docs/3.3.0/sql-ref-datatypes.html#supported-data-types>

## 3.9 The DataFrame Schema

The schema of a Spark DataFrame is the combination of column names and the data types associated with each of these columns. Schemas can be set explicitly by you (that is, you can tell Spark how the schema of your DataFrame should look like), or, they can be automatically defined by Spark while reading or creating your data.

You can get a succinct description of a DataFrame schema, by looking inside the object where this DataFrame is stored. For example, let's look again to the **df** DataFrame.

In the result below, we can see that **df** has three columns (**id**, **value** and **date**). By the description **id: bigint**, we know that **id** is a column of type **bigint**, which translates to the **LongType()** of Spark. Furthermore, by the descriptions **value: double** and **date: date**, we know too that the columns **value** and **date** are of type **DoubleType()** and **DateType()**, respectively.

```
df
```

```
DataFrame[id: bigint, value: double, date: date]
```

You can also visualize a more complete report of the DataFrame schema by using the **printSchema()** method, like this:

```
df.printSchema()
```

```
root
 |-- id: long (nullable = true)
 |-- value: double (nullable = true)
 |-- date: date (nullable = true)
```

### 3.9.1 Accessing the DataFrame schema

So, by calling the object of your DataFrame (i.e. an object of class **DataFrame**) you can see a small description of the schema of this DataFrame. But, how can you access this schema programmatically?

You do this, by using the **schema** method of your DataFrame, like in the example below:

```
df.schema
```

```
StructType([StructField('id', LongType(), True), StructField('value', DoubleType(), True), StructField('date', DateType(), True)])
```

The result of the **schema** method, is a **StructType()** object, that contains some information about each column of your DataFrame. More specifically, a **StructType()** object is filled with multiple **StructField()** objects. Each **StructField()** object stores the name and the type of a column, and a boolean value (**True** or **False**) that indicates if this column can contain any null value inside of it.

You can use a **for** loop to iterate through this **StructType()** and get the information about each column separately.

```
schema = df.schema
for column in schema:
    print(column)
```

```
StructField('id', LongType(), True)
StructField('value', DoubleType(), True)
StructField('date', DateType(), True)
```

You can access just the data type of each column by using the **dataType** method of each **StructField()** object.

```
for column in schema:
    datatype = column.dataType
    print(datatype)
```

```
LongType()
DoubleType()
DateType()
```

And you can do the same for column names and the boolean value (that indicates if the column can contain “null” values), by using the **name** and **nullable** methods, respectively.

```
# Accessing the name of each column
for column in schema:
    print(column.name)
```

```
id
value
date
```



```
# Accessing the boolean value that indicates
# if the column can contain null values
for column in schema:
    print(column.nullable)
```

```
True
True
True
```

### 3.9.2 Building a DataFrame schema

When Spark creates a new DataFrame, it will automatically guess which schema is appropriate for that DataFrame. In other words, Spark will try to guess which are the appropriate data types for each column. But, this is just a guess, and, sometimes, Spark go way off.

Because of that, in some cases, you have to tell Spark how exactly you want this DataFrame schema to be like. To do that, you need to build the DataFrame schema by yourself, with **StructType()** and **StructField()** constructors, alongside with the Spark data types (i.e. **StringType()**, **DoubleType()**, **IntegerType()**, ...). Remember, all of these python classes come from the **pyspark.sql.types** module.

In the example below, the **schema** object represents the schema of the **registers** DataFrame. This DataFrame have three columns (**ID**, **Date**, **Name**) of types **IntegerType**, **DateType** and **StringType**, respectively.

You can see below that I deliver this **schema** object that I built to **spark.createDataFrame()**. Now **spark.createDataFrame()** will follow the schema I described in this **schema** object when building the **registers** DataFrame.

```
from pyspark.sql.types import StructType, StructField
from pyspark.sql.types import DateType, StringType, IntegerType
from datetime import date

data = [
    (1, date(2022, 1, 1), 'Anne'),
    (2, date(2022, 1, 3), 'Layla'),
    (3, date(2022, 1, 15), 'Wick'),
    (4, date(2022, 1, 11), 'Paul')
]

schema = StructType([
    StructField('ID', IntegerType(), True),
```

```

    StructField('Date', DateType(), True),
    StructField('Name', StringType(), True)
])

registers = spark.createDataFrame(data, schema = schema)

```

Having this example in mind, in order to build a DataFrame schema from scratch, you have to build the equivalent **StructType()** object that represents the schema you want.

### 3.9.3 Checking your DataFrame schema

In some cases, you need to include in your **pyspark** program, some checks that certifies that your Spark DataFrame have the expected schema. In other words, you want to take actions if your DataFrame have a different schema that might cause a problem in your program.

To check if a specific column of your DataFrame is associated with the data type  $x$ , you have to use the DataFrame schema to check if the respective column is an “instance” of the python class that represents that data type  $x$ . Lets use the **df** DataFrame as an example.

Suppose you wanted to check if the **id** column is of type **IntegerType**. To do this check, we use the python built-in function **isinstance()** with the python class that represents the Spark **IntegerType** data type. But, you can see in the result below, that the **id** column is not of type **IntegerType**.

```

from pyspark.sql.types import IntegerType
schema = df.schema
id_column = schema[0]
isinstance(id_column.dataType, IntegerType)

```

**False**

This unexpected result happens, because the **id** column is actually from the “big integer” type, or, the **LongType** (which are 8-byte signed integer). You can see below, that now the test results in true:

```

from pyspark.sql.types import LongType
isinstance(id_column.dataType, LongType)

```

**True**

## 4 Introducing the Column class

As we described at the introduction of Chapter 3, you will massively use the methods from the **DataFrame** class in your Spark applications to manage, modify and calculate your Spark DataFrames.

However, there is one more python class that provides some very useful methods that you will regularly use, which is the **Column** class, or more specifically, the **pyspark.sql.column.Column** class.

The **Column** class is used to represent a column in a Spark DataFrame. This means that, each column of your Spark DataFrame is a object of class **Column**.

We can confirm this statement, by taking the **df** DataFrame that we showed at Section 3.5, and look at the class of any column of it. Like the **id** column:

```
type(df.id)
```

```
pyspark.sql.column.Column
```

### 4.1 Building a column object

You can refer to or create a column, by using the **col()** and **column()** functions from **pyspark.sql.functions** module. These functions receive a string input with the name of the column you want to create/refer to.

Their result are always a object of class **Column**. For example, the code below creates a column called **ID**:

```
from pyspark.sql.functions import col
id_column = col('ID')
print(id_column)
```

```
Column<'ID'>
```

## 4.2 Columns are strongly related to expressions

Many kinds of transformations that we want to apply over a Spark DataFrame, are usually described through expressions, and, these expressions in Spark are mainly composed by **column transformations**. That is why the **Column** class, and its methods, are so important in Apache Spark.

Columns in Spark are so strongly related to expressions that the columns themselves are initially interpreted as expressions. If we look again at the column **id** from **df** DataFrame, Spark will bring an expression as a result, and not the values hold by this column.

```
df.id
```

```
Column<'id'>
```

Having these ideas in mind, when I created the column **ID** on the previous section, I created a “column expression”. This means that **col("ID")** is just an expression, and as consequence, Spark does not know which are the values of column **ID**, or, where it lives (which is the DataFrame that this column belongs?). For now, Spark is not interested in this information, it just knows that we have an expression referring to a column called **ID**.

These ideas relates a lot to the **lazy aspect** of Spark that we talked about in Section 3.6. Spark will not perform any heavy calculation, or show you the actual results/values from you code, until you trigger the calculations with an action (we will talk more about these “actions” on Section 5.3). As a result, when you access a column, Spark will only deliver an expression that represents that column, and not the actual values of that column.

This is handy, because we can store our expressions in variables, and, reuse it latter, in multiple parts of our code. For example, I can keep building and merging a column with different kinds of operators, to build a more complex expression. In the example below, I create an expression that doubles the values of **ID** column:

```
expr1 = id_column * 2
print(expr1)
```

```
Column<'(ID * 2)'>
```

Remember, with this expression, Spark knows that we want to get a column called **ID** somewhere, and double its values. But Spark will not perform that action right now.

Logical expressions follow the same logic. In the example below, I am looking for rows where the value in column **Name** is equal to **'Anne'**, and, the value in column **Grade** is above 6.

Again, Spark just checks if this is a valid logical expression. For now, Spark does not want to know where are these **Name** and **Grade** columns. Spark does not evaluate the expression, until we ask for it with an action:

```
expr2 = (col('Name') == 'Anne') & (col('Grade') > 6)
print(expr2)
```

```
Column<'((Name = Anne) AND (Grade > 6))'>
```

## 4.3 Literal values versus expressions

We know now that columns of a Spark DataFrame have a deep connection with expressions. But, on the other hand, there are some situations that you write a value (it can be a string, a integer, a boolean, or anything) inside your **pyspark** code, and you might actually want Spark to interpret this value as a constant (or a literal) value, rather than a expression.

As an example, lets suppose you control the data generated by the sales of five different stores, scattered across different regions of Belo Horizonte city (in Brazil). Now, lets suppose you receive a batch of data generated by the 4th store in the city, which is located at Amazonas Avenue, 324. This batch of data is exposed below:

```
path = '../Data/sales.json'
sales = spark.read.json(path)
sales.show(5)
```

```
+-----+-----+-----+-----+-----+-----+
|price|product_id|product_name|sale_id|          timestamp|units|
+-----+-----+-----+-----+-----+-----+
| 3.12|      134| Milk 1L Mua| 328711|2022-02-01T22:10:02|   1|
| 1.22|      110|  Coke 350ml| 328712|2022-02-03T11:42:09|   3|
| 4.65|      117|   Pepsi 2L| 328713|2022-02-03T14:22:15|   1|
| 1.22|      110|  Coke 350ml| 328714|2022-02-03T18:33:08|   1|
| 0.85|      341|Trident Mint| 328715|2022-02-04T15:41:36|   1|
+-----+-----+-----+-----+-----+-----+
```

If you look at this batch... there is no indication that these sales come from the 4th store. In other words, this information is not present in the data, is just in your mind. It certainly is a very bad idea to leave this data as is, without any identification of the source of it. So, you might want to add some labels and new columns to this batch of data, that can easily identify the store that originated these sales.

For example, we could add two new columns to this **sales** DataFrame. One for the number that identifies the store (4), and, another to keep the store address. Considering that all rows in this batch comes from the 4th store, we should add two “constant” columns, meaning that these columns should have a constant value across all rows in this batch. But, how can we do this? How can we create a “constant” column? The answer is: by forcing Spark to interpret the values as literal values, instead of a expression.

In other words, I can not use the **col()** function to create these two new columns. Because this **col()** function receives a column name as input. **It interprets our input as an expression that refers to a column name.** This function does not accept some sort of description of the actual values that this column should store.

## 4.4 Passing a literal (or a constant) value to Spark

So how do we force Spark to interpret a value as a literal (or constant) value, rather than a expression? To do this, you must write this value inside the **lit()** (short for “literal”) function from the **pyspark.sql.functions** module.

In other words, when you write in your code the statement **lit(4)**, Spark understand that you want to create a new column which is filled with 4’s. In other words, this new column is filled with the constant integer 4.

With the code below, I am creating two new columns (called **store\_number** and **store\_address**), and adding them to the **sales** DataFrame.

```
from pyspark.sql.functions import lit
store_number = lit(4).alias('store_number')
store_address = lit('Amazonas Avenue, 324').alias('store_address')

sales = sales\
    .select(
        '*', store_number, store_address
    )

sales\
    .select(
        'product_id', 'product_name',
        'store_number', 'store_address'
    )\
    .show(5)
```

+-----+-----+-----+-----+

product_id	product_name	store_number	store_address
134	Milk 1L Mua	4	Amazonas Avenue, 324
110	Coke 350ml	4	Amazonas Avenue, 324
117	Pepsi 2L	4	Amazonas Avenue, 324
110	Coke 350ml	4	Amazonas Avenue, 324
341	Trident Mint	4	Amazonas Avenue, 324

In essence, you normally use the **lit()** function when you want to write a literal value in places where Spark expects a column name. In the example above, instead of writing a name to an existing column in the **sales** DataFrame, I wanted to write the literal values **'Amazonas Avenue, 324'** and **4**, and I used the **lit()** function to make this intention very clear to Spark. If I did not use the **lit()** function, the **withColumn()** method would interpret the value **'Amazonas Avenue, 324'** as an existing column named **Amazonas Avenue, 324**.

## 4.5 Some key column methods

Because many transformations that we want to apply over our DataFrames are expressed as column transformations, the methods from the **Column** class will be quite useful on many different contexts. You will see many of these methods across the next chapters, like **desc()**, **alias()** and **cast()**.

Remember, you can always use the **dir()** function to see the complete list of methods available in any python class. It's always useful to check the official documentation too<sup>1</sup>. There you will have a more complete description of each method.

But since they are so important in Spark, let's just give you a brief overview of some of the most popular methods from the **Column** class (these methods will be described in more detail in later chapters):

- **desc()** and **asc()**: methods to order the values of the column in a descending or ascending order (respectively);
- **cast()** and **astype()**: methods to cast (or convert) the values of the column to a specific data type;
- **alias()**: method to rename a column;
- **substr()**: method that returns a new column with the sub string of each value;
- **isNull()** and **isNotNull()**: logical methods to test if each value in the column is a null value or not;
- **startswith()** and **endswith()**: logical methods to search for values that starts with or ends with a specific pattern;
- **like()** and **rlike()**: logical methods to search for a specific pattern or regular expression in the values of the column;

<sup>1</sup><https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html>

- **isin()**: logical method to test if each value in the column is some of the listed values;



# 5 Transforming your Spark DataFrame

## 5.1 Introduction

Virtually every data analysis or data pipeline will include some ETL (*Extract, Transform, Load*) process, and the T is an essential part of it. Because, you almost never have an input data, or a initial DataFrame that perfectly fits your needs.

This means that you always have to transform the initial data that you have, to a specific format that you can use in your analysis. In this chapter, you will learn how to apply some of these basic transformations to your Spark DataFrame.

## 5.2 Defining transformations

Spark DataFrames are **immutable**, meaning that, they cannot be directly changed. But you can use an existing DataFrame to create a new one, based on a set of transformations. In other words, you define a new DataFrame as a transformed version of an older DataFrame.

Basically every **pyspark** program that you write will have such transformations. Spark support many types of transformations, however, in this chapter, we will focus on four basic transformations that you can apply to a DataFrame:

- Filtering rows;
- Sorting rows;
- Adding or deleting columns;
- Calculating aggregates;

Therefore, when you apply one of the above transformations to an existing DataFrame, you will get a new DataFrame as a result. You usually combine multiple transformations together to get your desired result. As a first example, lets get back to the **df** DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
from datetime import date
from pyspark.sql import Row
```

```
data = [
    Row(id = 1, value = 28.3, date = date(2021,1,1)),
    Row(id = 2, value = 15.8, date = date(2021,1,1)),
    Row(id = 3, value = 20.1, date = date(2021,1,2)),
    Row(id = 4, value = 12.6, date = date(2021,1,3))
]

df = spark.createDataFrame(data)
```

In the example below, to create a new DataFrame called **big\_values**, we begin with the **df** DataFrame, then, we filter its rows where **value** is greater than 15, then, we select **date** and **value** columns, then, we sort the rows based on the **value** column. So, this set of sequential transformations (filter it, then, select it, then, order it, ...) defines what this new **big\_values** DataFrame is.

```
from pyspark.sql.functions import col
# You define a chain of transformations to
# create a new DataFrame
big_values = df\
    .filter(col('value') > 15)\
    .select('date', 'value')\
    .orderBy('value')
```

Thus, to apply a transformation to an existing DataFrame, we use DataFrame methods such as **select()**, **filter()**, **orderBy()** and many others. Remember, these are methods from the python class that defines Spark DataFrame's (i.e. the **pyspark.sql.dataframe.DataFrame** class).

This means that you can apply these transformations only to Spark DataFrames, and no other kind of python object. For example, if you try to use the **orderBy()** method in a standard python string (i.e. an object of class **str**), you will get an **AttributeError** error. Because this class of object in python, does not have a **orderBy()** method:

```
s = "A python string"
s.orderBy('value')
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'orderBy'
```

Each one of these DataFrame methods create a *lazily evaluated transformation*. Once again, we see the **lazy** aspect of Spark doing its work here. All these transformation methods are lazily evaluated, meaning that, Spark will only check if they make sense with the initial DataFrame that you have. Spark

will not actually perform these transformations on your initial DataFrame, not until you trigger these transformations with an **action**.

## 5.3 Triggering calculations with actions

Therefore, Spark will avoid performing any heavy calculation until such calculation is really needed. But how or when Spark will face this decision? **When it encounters an action**. An action is the tool you have to trigger Spark to actually perform the transformations you have defined.

An action instructs Spark to compute the result from a series of transformations. (Chambers and Zaharia 2018).

There are four kinds of actions in Spark:

- Showing an output in the console;
- Writing data to some file or data source;
- Collecting data from a Spark DataFrame to native objects in python (or Java, Scala, R, etc.);
- Counting the number of rows in a Spark DataFrame;

You already know the first type of action, because we used it before with the **show()** method. This **show()** method is an action by itself, because you are asking Spark to show some output to you. So we can make Spark to actually calculate the transformations that defines the **big\_values** DataFrame, by asking Spark to show this DataFrame to us.

```
big_values.show()
```

[Stage 0:>

(0 + 12) / 12]

```
+-----+-----+
|      date|value|
+-----+-----+
|2021-01-01| 15.8|
|2021-01-02| 20.1|
|2021-01-01| 28.3|
+-----+-----+
```

Another very useful action is the **count()** method, that gives you the number of rows in a DataFrame. To be able to count the number of rows in a DataFrame, Spark needs to access this DataFrame in the first place. That is why this **count()** method behaves as an action. Spark will perform the transformations that defines **big\_values** to access the actual rows of this DataFrame and count them.

```
big_values.count()
```

### 3

Furthermore, sometimes, you want to collect the data of a Spark DataFrame to use it inside python. In other words, sometimes you need to do some work that Spark cannot do by itself. To do so, you collect part of the data that is being generated by Spark, and store it inside a normal python object to use it in a standard python program.

That is what the **collect()** method do. It transfers all the data of your Spark DataFrame into a standard python list that you can easily access with python. More specifically, you get a python list full of **Row()** values:

```
data = big_values.collect()
print(data)
```

```
[Row(date=datetime.date(2021, 1, 1), value=15.8), Row(date=datetime.date(2021, 1, 2), value=20.1), Row(date=datetime.date(2021, 1, 1), value=28.3)]
```

The **take()** method is very similar to **collect()**. But you usually apply **take()** when you need to collect just a small section of your DataFrame (and not the entire thing), like the first **n** rows.

```
n = 1
first_row = big_values.take(n)
print(first_row)
```

```
[Row(date=datetime.date(2021, 1, 1), value=15.8)]
```

The last action would be the **write** method of a Spark DataFrame, but we will explain this method latter at Chapter 6.

## 5.4 Understanding narrow and wide transformations

There are two kinds of transformations in Spark: narrow and wide transformations. Remember, a Spark DataFrame is divided into many small parts (called partitions), and, these parts are spread across the cluster. The basic difference between narrow and wide transformations, is if the transformation forces Spark to read data from multiple partitions to generate a single part of the result of that transformation, or not.

## Narrow transformations

are 1 to 1 transformations

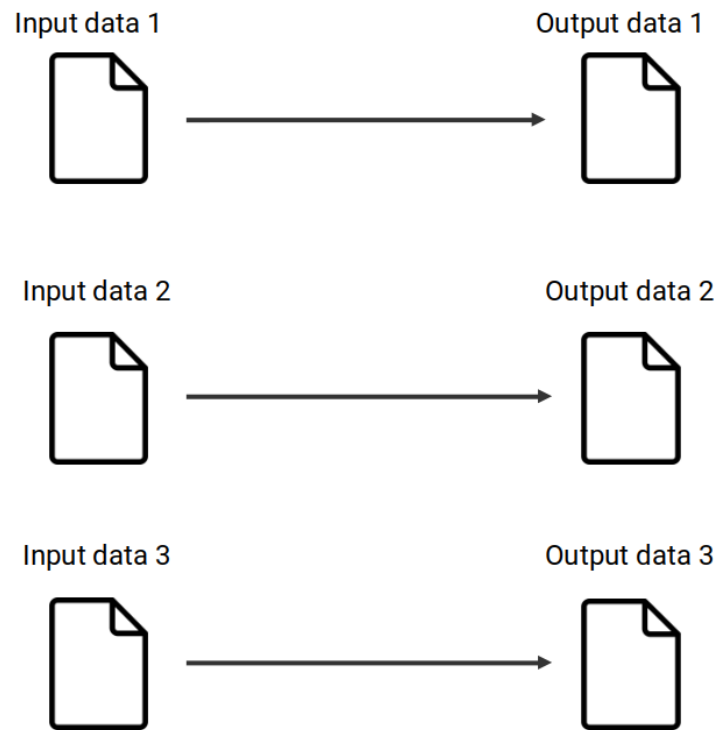


Figure 5.1: Presenting narrow transformations

More technically, narrow transformations are simply transformations where 1 input data (or 1 partition of the input DataFrame) contributes to only 1 partition of the output.

In other words, each partition of your input DataFrame will be used (*separately*) to generate one individual part of the result of your transformation. As another perspective, you can understand narrow transformations as those where Spark does not need to read the entire input DataFrame to generate a single and small piece of your result.

A classic example of narrow transformation is a filter. For example, suppose you have three students (Anne, Carls and Mike), and that each one has a bag full of blue, orange and red balls mixed. Now, suppose you asked them to collect all the red balls of these bags, and combined them in a single bag.

To do this task, Mike does not need to know what balls are inside of the bag of Carls or Anne. He just need to collect the red balls that are solely on his bag. At the end of the task, each student will have a part of the end result (that is, all the red balls that were in his own bag), and they just need to combine all these parts to get the total result.

The same thing applies to filters in Spark DataFrames. When you filter all the rows where the column **state** is equal to "**Alaska**", Spark will filter all the rows in each partition separately, and then, will combine all the outputs to get the final result.

In contrast, wide transformations are the opposite of that. In wide transformations, Spark needs to use more than 1 partition of the input DataFrame to generate a small piece of the result.

When this kind of transformation happens, each worker node of the cluster needs to share his partition with the others. In other words, what happens is a partition shuffle. Each worker node sends his partition to the others, so they can have access to it, while performing their assigned tasks.

Partition shuffles are a very popular topic in Apache Spark, because they can be a serious source of inefficiency in your Spark application (Chambers and Zaharia 2018). In more details, when these shuffles happens, Spark needs to write data back to the hard disk of the computer, and this is not a very fast operation. It does not mean that wide transformations are bad or slow, just that the shuffles they are producing can be a problem.

A classic example of wide operation is a grouped aggregation. For example, lets suppose we had a DataFrame with the daily sales of multiple stores spread across the country, and, we wanted to calculate the total sales per city/region. To calculate the total sales of a specific city, like “São Paulo”, Spark would need to find all the rows that corresponds to this city, before adding the values, and these rows can be spread across multiple partitions of the cluster.

## 5.5 The **transf** DataFrame

To demonstrate some of the next examples in this chapter, we will use a different DataFrame called **transf**. The data that represents this DataFrame is freely available as a CSV file. You can download

## Wide transformations

are 1 to N transformations

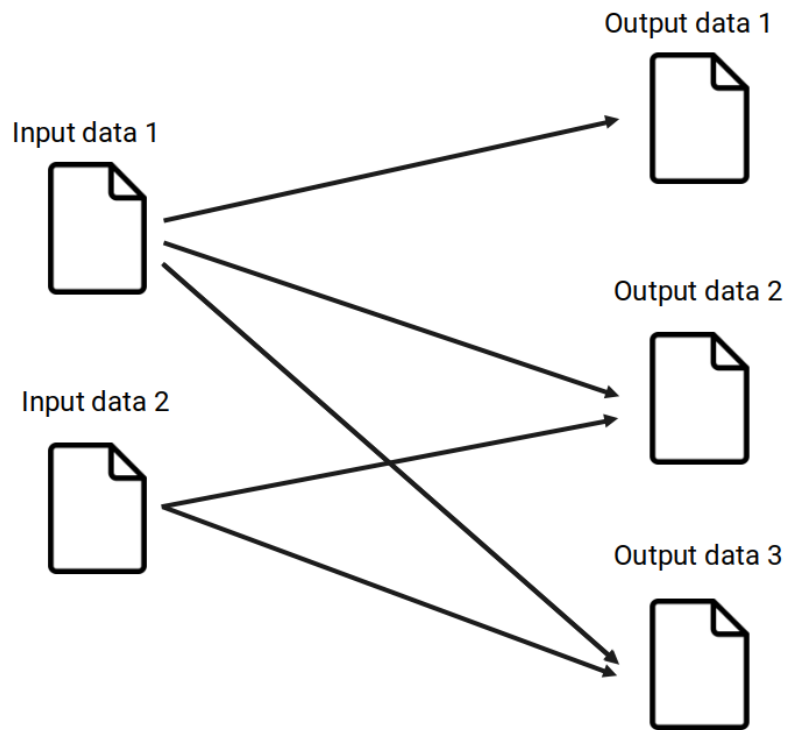


Figure 5.2: Presenting wide transformations

this CSV at the repository of this book<sup>1</sup>.

With the code below, you can import the data from the **transf.csv** CSV file, to recreate the **transf** DataFrame in your Spark Session:

```
from pyspark.sql.types import StructType, StructField
from pyspark.sql.types import DoubleType, StringType
from pyspark.sql.types import LongType, TimestampType, DateType
path = "../Data/transf.csv"
schema = StructType([
    StructField('dateTransfer', DateType(), False),
    StructField('datetimeTransfer', TimestampType(), False),
    StructField('clientNumber', LongType(), False),
    StructField('transferValue', DoubleType(), False),
    StructField('transferCurrency', StringType(), False),
    StructField('transferID', LongType(), False),
    StructField('transferLog', StringType(), False),
    StructField('destinationBankNumber', LongType(), False),
    StructField('destinationBankBranch', LongType(), False),
    StructField('destinationBankAccount', StringType(), False)
])

transf = spark.read\
    .csv(path, schema = schema, sep = ";", header = True)
```

You could also use the **pandas** library to read the DataFrame directly from GitHub, without having to manually download the file:

```
import pandas as pd
url = 'https://raw.githubusercontent.com/'
url = url + 'pedropark99/Introd-pyspark/'
url = url + 'main/Data/transf.csv'

transf_pd = pd.read_csv(
    url, sep = ';',
    dtype = str,
    keep_default_na = False
)

transf_pd['transferValue'] = transf_pd['transferValue'].astype('float')
```

---

<sup>1</sup><https://github.com/pedropark99/Introd-pyspark/tree/main/Data>



```

columns_to_int = [
    'clientNumber',
    'transferID',
    'destinationBankNumber',
    'destinationBankBranch'
]
for column in columns_to_int:
    transf_pd[column] = transf_pd[column].astype('int')

from pyspark.sql import SparkSession
from pyspark.sql.functions import col
spark = SparkSession.builder.getOrCreate()
transf = spark.createDataFrame(transf_pd)\
    .withColumn('dateTransfer', col('dateTransfer').cast('date'))\
    .withColumn('datetimeTransfer', col('datetimeTransfer').cast('timestamp'))

```

This **transf** DataFrame contains bank transfer records from a fictitious bank. Before I show you the actual data of this DataFrame, is useful to give you a quick description of each column that it contains:

- **dateTransfer**: the date when the transfer occurred;
- **datetimeTransfer**: the date and time when the transfer occurred;
- **clientNumber** the unique number that identifies a client of the bank;
- **transferValue**: the nominal value that was transferred;
- **transferCurrency**: the currency of the nominal value transferred;
- **transferID**: an unique ID for the transfer;
- **transferLog**: store any error message that may have appeared during the execution of the transfer;
- **destinationBankNumber**: the transfer destination bank number;
- **destinationBankBranch**: the transfer destination branch number;
- **destinationBankAccount**: the transfer destination account number;

Now, to see the actual data of this DataFrame, we can use the **show()** action as usual.

```
transf.show(5)
```

```

+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-31|2022-12-31 14:00:24|      5516|      7794.31|          zing f|
|  2022-12-31|2022-12-31 10:32:07|      4965|       7919.0|          zing f|
|  2022-12-31|2022-12-31 07:37:02|      4608|       5603.0|        dollar $|
|  2022-12-31|2022-12-31 07:35:05|      1121|       4365.22|        dollar $|

```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-31	2022-12-31 02:53:44	1121	4620.0	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

## 5.6 Filtering rows of your DataFrame

To filter specific rows of a DataFrame, **pyspark** offers two equivalent DataFrame methods called **where()** and **filter()**. In other words, they both do the same thing, and work in the same way. These methods receives as input a logical expression that translates what you want to filter.

As a first example, lets suppose you wanted to inspect all the rows from the **transf** DataFrame where **transferValue** is less than 1000. To do so, you can use the following code:

```
transf\
    .filter("transferValue < 1000")\
    .show(5)
```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-18	2022-12-18 08:45:30	1297	142.66	dollar \$
2022-12-13	2022-12-13 20:44:23	5516	992.15	dollar \$
2022-11-24	2022-11-24 20:01:39	1945	174.64	dollar \$
2022-11-07	2022-11-07 16:35:57	4862	570.69	dollar \$
2022-11-04	2022-11-04 20:00:34	1297	854.0	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

Writing simple SQL logical expression inside a string is the most easy and “clean” way to create a filter expression in **pyspark**. However, you could also write the same exact expression in a more “pythonic” way, using the **col()** function from **pyspark.sql.functions** module.

```
from pyspark.sql.functions import col

transf\
    .filter(col("transferValue") < 1000)\
    .show(5)
```

```

+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-18|2022-12-18 08:45:30|      1297|      142.66|      dollar $|
|  2022-12-13|2022-12-13 20:44:23|      5516|      992.15|      dollar $|
|  2022-11-24|2022-11-24 20:01:39|      1945|      174.64|      dollar $|
|  2022-11-07|2022-11-07 16:35:57|      4862|      570.69|      dollar $|
|  2022-11-04|2022-11-04 20:00:34|      1297|      854.0|      dollar $|
+-----+-----+-----+-----+-----+

```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

You still have a more verbose alternative, that does not require the **col()** function. With this method, you refer to the specific column using the dot operator (**.**), like in the example below:

```

# This will give you the exact
# same result of the examples above
transf\
    .filter(transf.transferValue < 1000)

```

### 5.6.1 Logical operators available

As we saw in the previous section, there are two ways to write logical expressions in **pyspark**: 1) write a SQL logical expression inside a string; 2) or, write a python logical expression using the **col()** function.

If you choose to write a SQL logical expressions in a string, you need to use the logical operators of SQL in your expression (not the logical operators of python). In the other hand, if you choose to write in the “python” way, then, you need to use the logical operators of python instead.

The logical operators of SQL are described in the table below:

Table 5.1: List of logical operators of SQL

Operator	Example of expression	Meaning of the expression
<	<b>x &lt; y</b>	is <b>x</b> less than <b>y</b> ?
>	<b>x &gt; y</b>	is <b>x</b> greater than <b>y</b> ?
<=	<b>x &lt;= y</b>	is <b>x</b> less than or equal to <b>y</b> ?
>=	<b>x &gt;= y</b>	is <b>x</b> greater than or equal to <b>y</b> ?
==	<b>x == y</b>	is <b>x</b> equal to <b>y</b> ?
!=	<b>x != y</b>	is <b>x</b> not equal to <b>y</b> ?

Operator	Example of expression	Meaning of the expression
in	<b>x in y</b>	is <b>x</b> one of the values listed in <b>y</b> ?
and	<b>x and y</b>	both logical expressions <b>x</b> and <b>y</b> are true?
or	<b>x or y</b>	at least one of logical expressions <b>x</b> and <b>y</b> are true?
not	<b>not x</b>	is the logical expression <b>x</b> not true?

And, the logical operators of python are described in the table below:

Table 5.2: List of logical operators of python

Operator	Example of expression	Meaning of the expression
<	<b>x &lt; y</b>	is <b>x</b> less than <b>y</b> ?
>	<b>x &gt; y</b>	is <b>x</b> greater than <b>y</b> ?
<=	<b>x &lt;= y</b>	is <b>x</b> less than or equal to <b>y</b> ?
>=	<b>x &gt;= y</b>	is <b>x</b> greater than or equal to <b>y</b> ?
==	<b>x == y</b>	is <b>x</b> equal to <b>y</b> ?
!=	<b>x != y</b>	is <b>x</b> not equal to <b>y</b> ?
&	<b>x &amp; y</b>	both logical expressions <b>x</b> and <b>y</b> are true?
	<b>x   y</b>	at least one of logical expressions <b>x</b> and <b>y</b> are true?
~	<b>~x</b>	is the logical expression <b>x</b> not true?

### 5.6.2 Connecting multiple logical expressions

Sometimes, you need to write more complex logical expressions to correctly describe the rows you are interested in. That is, when you combine multiple logical expressions together.

As an example, lets suppose you wanted all the rows in **transf** DataFrame from client of number 1297 where the transfer value is smaller than 1000, and the date of the transfer is after 20 of February 2022. These conditions are dependent, that is, they are connected to each other (the client number, the transfer value and the date of the transfer). That is why I used the **and** keyword between each condition in the example below (i.e. to connect these three conditions together).

```
condition = '''
    transferValue < 1000
    and clientNumber == 1297
    and dateTransfer > '2022-02-20'
'''

transf\
    .filter(condition)\
```

```
.show()
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-18|2022-12-18 08:45:30|      1297|      142.66|      dollar $|
|  2022-11-04|2022-11-04 20:00:34|      1297|       854.0|      dollar $|
|  2022-02-27|2022-02-27 13:27:44|      1297|       697.21|      dollar $|
+-----+-----+-----+-----+-----+
... with 5 more columns: transferID, transferLog, destinationBankNumber
                        destinationBankBranch, destinationBankAccount
```

I could translate this logical expression into the “pythonic” way (using the `col()` function). However, I would have to surround each individual expression by parentheses, and, use the `&` operator to substitute the `and` keyword.

```
transf\
  .filter(
    (col('transferValue') < 1000) &
    (col('clientNumber') == 1297) &
    (col('dateTransfer') > '2022-02-20')
  )\
  .show()
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-18|2022-12-18 08:45:30|      1297|      142.66|      dollar $|
|  2022-11-04|2022-11-04 20:00:34|      1297|       854.0|      dollar $|
|  2022-02-27|2022-02-27 13:27:44|      1297|       697.21|      dollar $|
+-----+-----+-----+-----+-----+
... with 5 more columns: transferID, transferLog, destinationBankNumber
                        destinationBankBranch, destinationBankAccount
```

This a **very important detail**, because it is very easy to forget. When building your complex logical expressions in the “python” way, always **remember to surround each expression by a pair of parentheses**. Otherwise, you will get a very confusing and useless error message, like this:

```
transf\
  .filter(
```

```

col('transferValue') < 1000 &
col('clientNumber') == 1297 &
col('dateTransfer') > '2022-02-20'
)\
.show(5)

```

Py4JError: An error occurred while calling o216.and. Trace:

```

py4j.Py4JException: Method and([class java.lang.Integer]) does not exist
  at py4j.reflection.ReflectionEngine.getMethod(ReflectionEngine.java:318)
  at py4j.reflection.ReflectionEngine.getMethod(ReflectionEngine.java:326)
  at py4j.Gateway.invoke(Gateway.java:274)
  at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)
  at py4j.commands.CallCommand.execute(CallCommand.java:79)
  at py4j.ClientServerConnection.waitForCommands(ClientServerConnection.java:182)
  at py4j.ClientServerConnection.run(ClientServerConnection.java:106)
  at java.base/java.lang.Thread.run(Thread.java:829)

```

In the above examples, we have logical expressions that are dependent on each other. But, let's suppose these conditions were independent. In this case, we would use the **or** keyword, instead of **and**. Now, Spark will look for every row of **transf** where **transferValue** is smaller than 1000, or, **clientNumber** is equal to 1297, or, **dateTransfer** is greater than 20 of February 2022.

```

condition = '''
transferValue < 1000
or clientNumber == 1297
or dateTransfer > '2022-02-20'
'''

transf\
.filter(condition)\
.show(5)

```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-31	2022-12-31 14:00:24	5516	7794.31	zing f
2022-12-31	2022-12-31 10:32:07	4965	7919.0	zing f
2022-12-31	2022-12-31 07:37:02	4608	5603.0	dollar \$
2022-12-31	2022-12-31 07:35:05	1121	4365.22	dollar \$
2022-12-31	2022-12-31 02:53:44	1121	4620.0	dollar \$

only showing top 5 rows

... with 5 more columns: **transferID**, **transferLog**, **destinationBankNumber**  
**destinationBankBranch**, **destinationBankAccount**

To translate this expression into the pythonic way, we have to substitute the **or** keyword by the **|** operator, and surround each expression by parentheses again:

```
transf\  
  .filter(  
    (col('transferValue') < 1000) |  
    (col('clientNumber') == 1297) |  
    (col('dateTransfer') > '2022-02-20')  
  )\  
  .show(5)
```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-31	2022-12-31 14:00:24	5516	7794.31	zing f
2022-12-31	2022-12-31 10:32:07	4965	7919.0	zing f
2022-12-31	2022-12-31 07:37:02	4608	5603.0	dollar \$
2022-12-31	2022-12-31 07:35:05	1121	4365.22	dollar \$
2022-12-31	2022-12-31 02:53:44	1121	4620.0	dollar \$

only showing top 5 rows

... with 5 more columns: **transferID**, **transferLog**, **destinationBankNumber**  
**destinationBankBranch**, **destinationBankAccount**

You can also increase the complexity of your logical expressions by mixing dependent expressions with independent expressions. For example, to filter all the rows where **dateTransfer** is greater than or equal to 01 of October 2022, and **clientNumber** is either 2727 or 5188, you would have the following code:

```
condition = '''  
  (clientNumber == 2727 or clientNumber == 5188)  
  and dateTransfer >= '2022-10-01'  
  '''  
  
transf\  
  .filter(condition)\  
  .show(5)
```

```

+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-29|2022-12-29 10:22:02|      2727|      4666.25|          euro €|
|  2022-12-27|2022-12-27 03:58:25|      5188|      7821.69|          dollar $|
|  2022-12-26|2022-12-25 23:45:02|      2727|      3261.73| british pound £|
|  2022-12-23|2022-12-23 05:32:49|      2727|       8042.0|          dollar $|
|  2022-12-22|2022-12-22 06:02:47|      5188|      8175.67|          dollar $|
+-----+-----+-----+-----+-----+

```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

If you investigate the above condition carefully, maybe, you will identify that this condition could be rewritten in a simpler format, by using the **in** keyword. This way, Spark will look for all the rows where **clientNumber** is equal to one of the listed values (2727 or 5188), and, that **dateTransfer** is greater than or equal to 01 of October 2022.

```

condition = '''
    clientNumber in (2727, 5188)
    and dateTransfer >= '2022-10-01'
'''

transf\
    .filter(condition)\
    .show(5)

```

```

+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-29|2022-12-29 10:22:02|      2727|      4666.25|          euro €|
|  2022-12-27|2022-12-27 03:58:25|      5188|      7821.69|          dollar $|
|  2022-12-26|2022-12-25 23:45:02|      2727|      3261.73| british pound £|
|  2022-12-23|2022-12-23 05:32:49|      2727|       8042.0|          dollar $|
|  2022-12-22|2022-12-22 06:02:47|      5188|      8175.67|          dollar $|
+-----+-----+-----+-----+-----+

```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount



### 5.6.3 Translating the in keyword to the pythonic way

Python does have a **in** keyword just like SQL, but, this keyword does not work as expected in **pyspark**. To write a logical expression, using the pythonic way, that filters the rows where a column is equal to one of the listed values, you can use the **isin()** method.

This method belongs to the **Column** class, so, you should always use **isin()** after a column name or a **col()** function. In the example below, we are filtering the rows where **destinationBankNumber** is 290 or 666:

```
transf\  
  .filter(col('destinationBankNumber').isin(290, 666))\  
  .show(5)
```

```
+-----+-----+-----+-----+-----+  
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|  
+-----+-----+-----+-----+-----+  
|  2022-12-31|2022-12-31 07:37:02|      4608|      5603.0|      dollar $|  
|  2022-12-31|2022-12-31 07:35:05|      1121|      4365.22|      dollar $|  
|  2022-12-31|2022-12-31 02:44:46|      1121|      7158.0|        zing f|  
|  2022-12-31|2022-12-31 01:02:06|      4862|      6714.0|      dollar $|  
|  2022-12-31|2022-12-31 00:48:47|      3294|     10882.52|      dollar $|  
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

### 5.6.4 Negating logical conditions

In some cases, is easier to describe what rows you **do not want** in your filter. That is, you want to negate (or invert) your logical expression. For this, SQL provides the **not** keyword, that you place before the logical expression you want to negate.

For example, we can filter all the rows of **transf** where **clientNumber** is not equal to 3284. Remember, the methods **filter()** and **where()** are equivalents or synonymous (they both mean the same thing).

```
condition = '''  
  not clientNumber == 3284  
  '''  
  
transf\  
  .where(condition)
```

```
.show(5)
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-31|2022-12-31 14:00:24|      5516|      7794.31|          zing f|
|  2022-12-31|2022-12-31 10:32:07|      4965|       7919.0|          zing f|
|  2022-12-31|2022-12-31 07:37:02|      4608|       5603.0|        dollar $|
|  2022-12-31|2022-12-31 07:35:05|      1121|       4365.22|        dollar $|
|  2022-12-31|2022-12-31 02:53:44|      1121|       4620.0|        dollar $|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

To translate this expression to the pythonic way, we use the `~` operator. However, because we are negating the logical expression as a whole, is important to surround the entire expression with parentheses.

```
transf\
  .where(~(col('clientNumber') == 3284))\
  .show(5)
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-31|2022-12-31 14:00:24|      5516|      7794.31|          zing f|
|  2022-12-31|2022-12-31 10:32:07|      4965|       7919.0|          zing f|
|  2022-12-31|2022-12-31 07:37:02|      4608|       5603.0|        dollar $|
|  2022-12-31|2022-12-31 07:35:05|      1121|       4365.22|        dollar $|
|  2022-12-31|2022-12-31 02:53:44|      1121|       4620.0|        dollar $|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

If you forget to add the parentheses, Spark will think you are negating just the column (e.g. `~col('clientNumber')`), and not the entire expression. That would not make sense, and, as a result, Spark would throw an error:

```
transf\
  .where(~col('clientNumber') == 3284)\
  .show(5)
```

**AnalysisException: cannot resolve '(NOT clientNumber)' due to data type mismatch: argument 1 requires b  
'Filter (NOT clientNumber#210L = 3284)**

Because the `~` operator is a little discrete and can go unnoticed, I sometimes use a different approach to negate my logical expressions. I make the entire expression equal to **False**. This way, I get all the rows where that particular expression is **False**. This makes my intention more visible in the code, but, is harder to write it.

```
# Filter all the rows where 'clientNumber' is not equal to
# 2727 or 5188.
transf\
  .where( (col('clientNumber').isin(2727, 5188)) == False )\
  .show(5)
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-31|2022-12-31 14:00:24|      5516|    7794.31|         zing f|
|  2022-12-31|2022-12-31 10:32:07|      4965|     7919.0|         zing f|
|  2022-12-31|2022-12-31 07:37:02|      4608|     5603.0|        dollar $|
|  2022-12-31|2022-12-31 07:35:05|      1121|     4365.22|        dollar $|
|  2022-12-31|2022-12-31 02:53:44|      1121|     4620.0|        dollar $|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

### 5.6.5 Filtering null values (i.e. missing data)

Sometimes, the **null** values play an important role in your filter. You either want to collect all these **null** values, so you can investigate why they are null in the first place, or, you want to completely eliminate them from your DataFrame.

Because this is a special kind of value in Spark, with a special meaning (the “absence” of a value), you need to use a special syntax to correctly filter these values in your DataFrame. In SQL, you can use the **is** keyword to filter these values:

```
transf\
  .where('transferLog is null')\
  .show(5)
```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-31	2022-12-31 14:00:24	5516	7794.31	zing f
2022-12-31	2022-12-31 10:32:07	4965	7919.0	zing f
2022-12-31	2022-12-31 07:37:02	4608	5603.0	dollar \$
2022-12-31	2022-12-31 07:35:05	1121	4365.22	dollar \$
2022-12-31	2022-12-31 02:53:44	1121	4620.0	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

However, if you want to remove these values from your DataFrame, then, you can just negate (or invert) the above expression with the **not** keyword, like this:

```
transf\
  .where('not transferLog is null')\
  .show(5)
```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-12-05	2022-12-05 00:51:00	2197	8240.62	zing f
2022-09-20	2022-09-19 21:59:51	5188	7583.9	dollar \$
2022-09-03	2022-09-03 06:07:59	3795	3654.0	zing f
2022-07-02	2022-07-02 15:29:50	4465	5294.0	dollar \$
2022-06-14	2022-06-14 10:21:55	1121	7302.0	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

The **is** and **not** keywords in SQL have a special relation. Because you can create the same negation/inversion of the expression by inserting the **not** keyword in the middle of the expression (you can do this too in expressions with the **in** keyword). In other words, you might see, in someone else's code, the same expression above written in this form:

```
transf\
  .where('transferLog is not null')\
  .show(5)
```

Both forms are equivalent and valid SQL logical expressions. But the latter is a strange version. Because we cannot use the **not** keyword in this manner on other kinds of logical expressions. Normally, we put the **not** keyword **before** the logical expression we want to negate, not in the middle of it. Anyway, just have in mind that this form of logical expression exists, and, that is a perfectly valid one.

When we translate the above examples to the “pythonic” way, many people tend to use the **null** equivalent of python, that is, the **None** value, in the expression. But as you can see in the result below, this method does not work as expected:

```
transf\
  .where(col('transferLog') == None)\
  .show(5)
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
... with 5 more columns: transferID, transferLog, destinationBankNumber
                        destinationBankBranch, destinationBankAccount
```

The correct way to do this in **pyspark**, is to use the **isNull()** method from the **Column** class.

```
transf\
  .where(col('transferLog').isNull())\
  .show(5)
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-31|2022-12-31 14:00:24|      5516|      7794.31|      zing f|
|  2022-12-31|2022-12-31 10:32:07|      4965|       7919.0|      zing f|
|  2022-12-31|2022-12-31 07:37:02|      4608|       5603.0|     dollar $|
|  2022-12-31|2022-12-31 07:35:05|      1121|       4365.22|     dollar $|
|  2022-12-31|2022-12-31 02:53:44|      1121|       4620.0|     dollar $|
+-----+-----+-----+-----+-----+
only showing top 5 rows
... with 5 more columns: transferID, transferLog, destinationBankNumber
                        destinationBankBranch, destinationBankAccount
```

If you want to eliminate the **null** values, just use the inverse method **isNotNull()**.

```
transf\  
  .where(col('transferLog').isNotNull())\  
  .show(5)
```

```
+-----+-----+-----+-----+-----+  
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|  
+-----+-----+-----+-----+-----+  
|  2022-12-05|2022-12-05 00:51:00|      2197|      8240.62|          zing f|  
|  2022-09-20|2022-09-19 21:59:51|      5188|       7583.9|        dollar $|  
|  2022-09-03|2022-09-03 06:07:59|      3795|       3654.0|          zing f|  
|  2022-07-02|2022-07-02 15:29:50|      4465|       5294.0|        dollar $|  
|  2022-06-14|2022-06-14 10:21:55|      1121|       7302.0|        dollar $|  
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

### 5.6.6 Filtering dates and datetimes in your DataFrame

Just as a quick side note, when you want to filter rows of your DataFrame that fits into a particular date, you can easily write this particular date as a single string, like in the example below:

```
df_0702 = transf\  
  .where(col('dateTransfer') == '2022-07-02')
```

When filtering datetimes you can also write the datetimes as strings, like this:

```
later_0330_pm = transf.where(  
  col('datetimeTransfer') > '2022-07-02 03:30:00'  
)
```

However, is a better practice to write these particular values using the built-in **date** and **datetime** classes of python, like this:

```
from datetime import date, datetime  
  
d = date(2022,2,7)  
dt = datetime(2022,2,7,3,30,0)
```

```

# Filter all rows where `dateTransfer`
# is equal to "2022-07-02"
df_0702 = transf.where(
    col('dateTransfer') == d
)

# Filter all rows where `datetimeTransfer`
# is greater than "2022-07-02 03:30:00"
later_0330_pm = transf.where(
    col('datetimeTransfer') > dt
)

```

When you translate the above expressions to SQL, you can also write the date and datetime values as strings. However, is also a good idea to use the **CAST()** SQL function to convert these string values into the correct data type before the actual filter. like this:

```

condition_d = '''
dateTransfer == CAST("2022-07-02" AS DATE)
'''

condition_dt = '''
dateTransfer > CAST("2022-07-02 03:30:00" AS TIMESTAMP)
'''

# Filter all rows where `dateTransfer`
# is equal to "2022-07-02"
df_0702 = transf.where(condition_d)

# Filter all rows where `datetimeTransfer`
# is greater than "2022-07-02 03:30:00"
later_0330_pm = transf.where(condition_dt)

```

In other words, with the SQL expression **CAST("2022-07-02 03:30:00" AS TIMESTAMP)** we are telling Spark to convert the literal string "2022-07-02 03:30:00" into an actual timestamp value.

### 5.6.7 Searching for a particular pattern in string values

Spark offers different methods to search a particular pattern within a string value. In this section, I want to describe how you can use these different methods to find specific rows in your DataFrame, that fit into the description of these patterns.

### 5.6.7.1 Starts with, ends with and contains

You can use the column methods **startswith()**, **endswith()** and **contains()**, to search for rows where a input string value starts with, ends with, or, contains a particular pattern, respectively.

These three methods returns a boolean value that indicates if the input string value matched the input pattern that you gave to the method. And you can use these boolean values they return to filter the rows of your DataFrame that fit into the description of these patterns.

Just as an example, in the following code, we are creating a new DataFrame called **persons**, that contains the description of 3 persons (Alice, Bob and Charlie). And I use these three methods to search for different rows in the DataFrame:

```
from pyspark.sql.functions import col

persons = spark.createDataFrame(
    [
        ('Alice', 25),
        ('Bob', 30),
        ('Charlie', 35)
    ],
    ['name', 'age']
)

# Filter the DataFrame to include only rows
# where the "name" column starts with "A"
persons.filter(col('name').startswith('A'))\
    .show()
```

```
+-----+
| name|age|
+-----+
|Alice| 25|
+-----+
```

```
# Filter the DataFrame to include only rows
# where the "name" column ends with "e"
persons.filter(col('name').endswith('e'))\
    .show()
```

```
+-----+
|  name|age|
```



```
+-----+----+
| Alice| 25|
|Charlie| 35|
+-----+-----+
```

```
# Filter the DataFrame to include only rows
# where the "name" column contains "ob"
persons.filter(col('name').contains('ob'))\
.show()
```

```
+-----+----+
|name|age|
+-----+----+
| Bob| 30|
+-----+-----+
```

### 5.6.7.2 Using regular expressions or SQL LIKE patterns

In Spark, you can also use a particular “SQL LIKE pattern” or a regular pattern (a.k.a. regex) to filter the rows of a DataFrame, by using the **Column** methods **like()** and **rlike()**.

In essence, the **like()** method is the **pyspark** equivalent of the **LIKE** SQL operator. As a result, this **like()** method expects a SQL pattern as input. This means that you can use the SQL metacharacters **%** (to match any number of characters) and **\_** (to match exactly one character) inside this pattern.

```
transf\
  .where(col('transferCurrency').like('british%'))\
  .show(5)
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
| 2022-12-30|2022-12-30 11:30:23|      1455|      5141.0| british pound £|
| 2022-12-30|2022-12-30 02:35:23|      5986|      6076.0| british pound £|
| 2022-12-29|2022-12-29 15:24:04|      4862|      5952.0| british pound £|
| 2022-12-29|2022-12-29 14:16:46|      2197|      8771.0| british pound £|
| 2022-12-29|2022-12-29 06:51:24|      5987|      2345.0| british pound £|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

Although the style of pattern matching used by **like()** being very powerful, you might need to use more powerful and flexible patterns. In those cases, you can use the **rlike()** method, which accepts a regular expression as input. In the example below, I am filtering rows where **destinationBankAccount** starts by the characters **54**.

```
regex = '^54([0-9]{3})-[0-9]$\ntransf\
.where(col('destinationBankAccount').rlike(regex))\
.show(5)
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-29|2022-12-29 02:54:23|      2197|      5752.0| british pound £|
|  2022-12-27|2022-12-27 04:51:45|      4862|     11379.0|        dollar $|
|  2022-12-05|2022-12-05 05:50:27|      4965|      5986.0| british pound £|
|  2022-12-04|2022-12-04 14:31:42|      4965|       8123.0|        dollar $|
|  2022-11-29|2022-11-29 16:23:07|      4862|      8060.0|         zing f|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

## 5.7 Managing the columns of your DataFrame

Sometimes, you need manage or transform the columns you have. For example, you might need to change the order of these columns, or, to delete/rename some of them. To do this, you can use the **select()** and **drop()** methods of your DataFrame.

The **select()** method works very similarly to the **SELECT** statement of SQL. You basically list all the columns you want to keep in your DataFrame, in the specific order you want.

```
transf\
.select(
  'transferID', 'datetimeTransfer',
  'clientNumber', 'transferValue'
)\
.show(5)
```

```
+-----+-----+-----+-----+-----+
```

transferID	datetimeTransfer	clientNumber	transferValue
20223563	2022-12-31 14:00:24	5516	7794.31
20223562	2022-12-31 10:32:07	4965	7919.0
20223561	2022-12-31 07:37:02	4608	5603.0
20223560	2022-12-31 07:35:05	1121	4365.22
20223559	2022-12-31 02:53:44	1121	4620.0

only showing top 5 rows

### 5.7.1 Renaming your columns

Realize in the example above, that the column names can be delivered directly as strings to **select()**. This makes life pretty easy, but, it does not give you extra options.

For example, you might want to rename some of the columns, and, to do this, you need to use the **alias()** method from **Column** class. Since this is a method from **Column** class, you need to use it after a **col()** or **column()** function, or, after a column name using the dot operator.

```
transf\
  .select(
    'datetimeTransfer',
    col('transferID').alias('ID_of_transfer'),
    transf.clientNumber.alias('clientID')
  )\
  .show(5)
```

datetimeTransfer	ID_of_transfer	clientID
2022-12-31 14:00:24	20223563	5516
2022-12-31 10:32:07	20223562	4965
2022-12-31 07:37:02	20223561	4608
2022-12-31 07:35:05	20223560	1121
2022-12-31 02:53:44	20223559	1121

only showing top 5 rows

By using this **alias()** method, you can rename multiple columns within a single **select()** call. However, you can use the **withColumnRenamed()** method to rename just a single column of your DataFrame. The first argument of this method, is the current name of this column, and, the second argument, is the new name of this column.

```
transf\
  .withColumnRenamed('clientNumber', 'clientID')\
  .show(5)
```

```
+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientID|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-31|2022-12-31 14:00:24|   5516|      7794.31|          zing f|
|  2022-12-31|2022-12-31 10:32:07|   4965|      7919.0|          zing f|
|  2022-12-31|2022-12-31 07:37:02|   4608|      5603.0|        dollar $|
|  2022-12-31|2022-12-31 07:35:05|   1121|      4365.22|        dollar $|
|  2022-12-31|2022-12-31 02:53:44|   1121|      4620.0|        dollar $|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

## 5.7.2 Dropping unnecessary columns

In some cases, your DataFrame just have too many columns and you just want to eliminate a few of them. In a situation like this, you can list the columns you want to drop from your DataFrame, inside the **drop()** method, like this:

```
transf\
  .drop('dateTransfer', 'clientNumber')\
  .show(5)
```

```
+-----+-----+-----+-----+-----+
|  datetimeTransfer|transferValue|transferCurrency|transferID|transferLog|
+-----+-----+-----+-----+-----+
|2022-12-31 14:00:24|      7794.31|          zing f|  20223563|      null|
|2022-12-31 10:32:07|      7919.0|          zing f|  20223562|      null|
|2022-12-31 07:37:02|      5603.0|        dollar $|  20223561|      null|
|2022-12-31 07:35:05|      4365.22|        dollar $|  20223560|      null|
|2022-12-31 02:53:44|      4620.0|        dollar $|  20223559|      null|
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 3 more columns: destinationBankNumber, destinationBankBranch  
destinationBankAccount

### 5.7.3 Casting columns to a different data type

Spark try to do its best when guessing which is correct data type for the columns of your DataFrame. But, obviously, Spark can get it wrong, and, you end up deciding by your own which data type to use for a specific column.

To explicit transform a column to a specific data type, you can use **cast()** or **astype()** methods inside **select()**. The **astype()** method is just an alias to **cast()**. This **cast()** method is very similar to the **CAST()** function in SQL, and belongs to the **Column** class, so, you should always use it after a column name with the dot operator, or, a **col()/column()** function:

```
transf\  
  .select(  
    'transferValue',  
    col('transferValue').cast('long').alias('value_as_integer'),  
    transf.transferValue.cast('boolean').alias('value_as_boolean')  
  )\  
  .show(5)
```

```
+-----+-----+-----+  
|transferValue|value_as_integer|value_as_boolean|  
+-----+-----+-----+  
|      7794.31|           7794|           true|  
|      7919.0|           7919|           true|  
|      5603.0|           5603|           true|  
|      4365.22|           4365|           true|  
|      4620.0|           4620|           true|  
+-----+-----+-----+
```

only showing top 5 rows

To use **cast()** or **astype()** methods, you give the name of the data type (as a string) to which you want to cast the column. The main available data types to **cast()** or **astype()** are:

- **'string'**: correspond to **StringType()**;
- **'int'**: correspond to **IntegerType()**;
- **'long'**: correspond to **LongType()**;
- **'double'**: correspond to **DoubleType()**;
- **'date'**: correspond to **DateType()**;
- **'timestamp'**: correspond to **TimestampType()**;
- **'boolean'**: correspond to **BooleanType()**;
- **'array'**: correspond to **ArrayType()**;
- **'dict'**: correspond to **MapType()**;

### 5.7.4 You can add new columns with `select()`

When I said that `select()` works in the same way as the `SELECT` statement of SQL, I also meant that you can use `select()` to select columns that do not currently exist in your DataFrame, and add them to the final result.

For example, I can select a new column (called `by_1000`) containing `value` divided by 1000, like this:

```
transf\  
  .select(  
    'transferValue',  
    (col('transferValue') / 1000).alias('by_1000')  
  )\  
  .show(5)
```

```
+-----+-----+  
|transferValue|by_1000|  
+-----+-----+  
|      7794.31|7.79431|  
|      7919.0| 7.919|  
|      5603.0| 5.603|  
|      4365.22|4.36522|  
|      4620.0| 4.62|  
+-----+-----+  
only showing top 5 rows
```

This `by_1000` column do not exist in `transf` DataFrame. It was calculated and added to the final result by `select()`. The formula `col('transferValue') / 1000` is the equation that defines what this `by_1000` column is, or, how it should be calculated.

Besides that, `select()` provides a useful shortcut to reference all the columns of your DataFrame. That is, the star symbol (\*) from the `SELECT` statement in SQL. This shortcut is very useful when you want to maintain all columns, and, add a new column, at the same time.

In the example below, we are adding the same `by_1000` column, however, we are bringing all the columns of `transf` together.

```
transf\  
  .select(  
    '*',  
    (col('transferValue') / 1000).alias('by_1000')  
  )\  
  .show(5)
```

```

+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-31|2022-12-31 14:00:24|      5516|      7794.31|        zing f|
|  2022-12-31|2022-12-31 10:32:07|      4965|       7919.0|        zing f|
|  2022-12-31|2022-12-31 07:37:02|      4608|       5603.0|       dollar $|
|  2022-12-31|2022-12-31 07:35:05|      1121|       4365.22|       dollar $|
|  2022-12-31|2022-12-31 02:53:44|      1121|       4620.0|       dollar $|
+-----+-----+-----+-----+-----+

```

only showing top 5 rows

... with 6 more columns: transferID, transferLog, destinationBankNumber  
 destinationBankBranch, destinationBankAccount, by\_1000

## 5.8 Calculating or adding new columns to your DataFrame

Although you can add new columns with **select()**, this method is not specialized to do that. As consequence, when you want to add many new columns, it can become pretty annoying to write **select('\*', new\_column)** over and over again. That is why **pyspark** provides a special method called **withColumn()**.

This method has two arguments. First, is the name of the new column. Second, is the formula (or the equation) that represents this new column. As an example, I could reproduce the same **by\_1000** column like this:

```

transf\
  .withColumn('by_1000', col('transferValue') / 1000)\
  .show(5)

```

```

+-----+-----+-----+-----+-----+
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|
+-----+-----+-----+-----+-----+
|  2022-12-31|2022-12-31 14:00:24|      5516|      7794.31|        zing f|
|  2022-12-31|2022-12-31 10:32:07|      4965|       7919.0|        zing f|
|  2022-12-31|2022-12-31 07:37:02|      4608|       5603.0|       dollar $|
|  2022-12-31|2022-12-31 07:35:05|      1121|       4365.22|       dollar $|
|  2022-12-31|2022-12-31 02:53:44|      1121|       4620.0|       dollar $|
+-----+-----+-----+-----+-----+

```

only showing top 5 rows

... with 6 more columns: transferID, transferLog, destinationBankNumber  
 destinationBankBranch, destinationBankAccount, by\_1000

A lot of the times we use the functions from **pyspark.sql.functions** module to produce such formulas used by **withColumn()**. You can checkout the complete list of functions present in this module, by visiting the official documentation of **pyspark**<sup>2</sup>.

You will see a lot more examples of formulas and uses of **withColumn()** throughout this book. For now, I just want you to know that **withColumn()** is a method that adds a new column to your DataFrame. The first argument is the name of the new column, and, the second argument is the calculation formula of this new column.

## 5.9 Sorting rows of your DataFrame

Spark, or, **pyspark**, provides the **orderBy()** and **sort()** DataFrame method to sort rows. They both work the same way: you just give the name of the columns that Spark will use to sort the rows.

In the example below, Spark will sort **transf** according to the values in the **transferValue** column. By default, Spark uses an ascending order while sorting your rows.

```
transf\  
  .orderBy('transferValue')\  
  .show(5)
```

```
+-----+-----+-----+-----+-----+  
|dateTransfer|  datetimeTransfer|clientNumber|transferValue|transferCurrency|  
+-----+-----+-----+-----+-----+  
|  2022-07-22|2022-07-22 16:06:25|      3795|        60.0|      dollar $|  
|  2022-05-09|2022-05-09 14:02:15|      3284|       104.0|      dollar $|  
|  2022-09-16|2022-09-16 20:35:40|      3294|       129.09|        zing f|  
|  2022-12-18|2022-12-18 08:45:30|      1297|       142.66|      dollar $|  
|  2022-08-20|2022-08-20 09:27:55|      2727|       160.0|      dollar $|  
+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

Just to be clear, you can use the combination between multiple columns to sort your rows. Just give the name of each column (as strings) separated by commas. In the example below, Spark will sort the rows according to **clientNumber** column first, then, is going to sort the rows of each **clientNumber** according to **transferValue** column.

---

<sup>2</sup><https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql.html#functions>



```
transf\
.orderBy('clientNumber', 'transferValue')\
.show(5)
```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-03-30	2022-03-30 11:57:22	1121	461.0	euro €
2022-05-23	2022-05-23 11:51:02	1121	844.66	british pound £
2022-08-24	2022-08-24 13:51:30	1121	1046.93	euro €
2022-09-23	2022-09-23 19:49:19	1121	1327.0	british pound £
2022-06-25	2022-06-25 17:07:08	1121	1421.0	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

If you want to use a descending order in a specific column, you need to use the **desc()** method from **Column** class. In the example below, Spark will sort the rows according to **clientNumber** column using an ascending order. However, it will use the values from **transferValue** column in a descending order to sort the rows in each **clientNumber**.

```
transf\
.orderBy('clientNumber', col('transferValue').desc())\
.show(5)
```

dateTransfer	datetimeTransfer	clientNumber	transferValue	transferCurrency
2022-08-18	2022-08-18 13:57:12	1121	11490.37	zing f
2022-11-05	2022-11-05 08:00:37	1121	10649.59	dollar \$
2022-05-17	2022-05-17 10:27:05	1121	10471.23	british pound £
2022-05-15	2022-05-15 00:25:49	1121	10356.0	dollar \$
2022-06-10	2022-06-09 23:51:39	1121	10142.0	dollar \$

only showing top 5 rows

... with 5 more columns: transferID, transferLog, destinationBankNumber  
destinationBankBranch, destinationBankAccount

This means that you can mix ascending orders with descending orders in **orderBy()**. Since the ascending order is the default, if you want to use a descending order in all of the columns, then, you need to apply the **desc()** method to all of them.

## 5.10 Calculating aggregates

To calculate aggregates of a Spark DataFrame we have two main paths: 1) we can use some standard DataFrame methods, like **count()** or **sum()**, to calculate a single aggregate; 2) or, we can use the **agg()** method to calculate multiple aggregates at the same time.

### 5.10.1 Using standard DataFrame methods

The Spark DataFrame class by itself provides a single aggregate method, which is **count()**. With this method, you can find out how many rows your DataFrame have. In the example below, we can see that **transf** have 2421 rows.

```
transf.count()
```

```
2421
```

However, if you have a **grouped** DataFrame (we will learn more about these objects very soon), **pyspark** provides some more aggregate methods, which are listed below:

- **mean()**: calculate the average value of each numeric column;
- **sum()**: return the total sum of a column;
- **count()**: count the number of rows;
- **max()**: compute the maximum value of a column;
- **min()**: compute the minimum value of a column;

This means that you can use any of the above methods after a **groupby()** call, to calculate aggregates *per group* in your Spark DataFrame. For now, let's forget about this “groupby” detail, and learn how to calculate different aggregates by using the **agg()** method.

### 5.10.2 Using the agg() method

With the **agg()** method, we can calculate many different aggregates at the same time. In this method, you should provide an expression that describes what aggregate measure you want to calculate.

In most cases, this “aggregate expression” will be composed of functions from the **pyspark.sql.functions** module. So, having familiarity with the functions present in this module will help you to compose the formulas of your aggregations in **agg()**.

In the example below, I am using the **sum()** and **mean()** functions from **pyspark.sql.functions** to calculate the total sum and the total mean of the **transferValue** column in the **transf** DataFrame.

I am also using the `countDistinct()` function to calculate the number of distinct values in the `clientNumber` column.

```
import pyspark.sql.functions as F

transf.agg(
    F.sum('transferValue').alias('total_value'),
    F.mean('transferValue').alias('mean_value'),
    F.countDistinct('clientNumber').alias('number_of_clients')
)\
.show()
```

```
+-----+-----+-----+
|      total_value|      mean_value|number_of_clients|
+-----+-----+-----+
|1.5217690679999998E7|6285.704535315985|          26|
+-----+-----+-----+
```

### 5.10.3 Without groups, we calculate a aggregate of the entire DataFrame

When we do not define any group for the input DataFrame, `agg()` always produce a new DataFrame with one single row (like in the above example). This happens because we are calculating aggregates of the entire DataFrame, that is, a set of single values (or single measures) that summarizes (in some way) the entire DataFrame.

In the other hand, when we define groups in a DataFrame (by using the `groupby()` method), the calculations performed by `agg()` are made inside each group in the DataFrame. In other words, instead of summarizing the entire DataFrame, `agg()` will produce a set of single values that describes (or summarizes) each group in the DataFrame.

This means that each row in the resulting DataFrame describes a specific group in the original DataFrame, and, `agg()` usually produces a DataFrame with more than one single row when its calculations are performed by group. Because our DataFrames usually have more than one single group.

### 5.10.4 Calculating aggregates per group in your DataFrame

But how you define the groups inside your DataFrame? To do this, we use the `groupby()` and `groupBy()` methods. These methods are both synonymous (they do the same thing).

These methods, produce a **grouped** DataFrame as a result, or, in more technical words, a object of class **pyspark.sql.group.GroupedData**. You just need to provide, inside this **groupby()** method, the name of the columns that define (or “mark”) your groups.

In the example below, I am creating a grouped DataFrame per client defined in the **clientNumber** column. This means that each distinct value in the **clientNumber** column defines a different group in the DataFrame.

```
transf_per_client = transf.groupBy('clientNumber')
transf_per_client
```

<pyspark.sql.group.GroupedData at 0x7f533424cac0>

At first, it appears that nothing has changed. But the **groupBy()** method always returns a new object of class **pyspark.sql.group.GroupedData**. As a consequence, we can no longer use some of the DataFrame methods that we used before, like the **show()** method to see the DataFrame.

That’s ok, because we usually do not want to keep this grouped DataFrame for much time. This grouped DataFrame is just a passage (or a bridge) to the result we want, which is, to calculate aggregates **per group** of the DataFrame.

As an example, I can use the **max()** method, to find out which is the highest value that each user have tranfered, like this:

```
transf_per_client\
    .max('transferValue')\
    .show(5)
```

```
+-----+-----+
|clientNumber|max(transferValue)|
+-----+-----+
|      1217|      12601.0|
|      2489|      12644.56|
|      3284|      12531.84|
|      4608|      10968.31|
|      1297|      11761.0|
+-----+-----+
only showing top 5 rows
```

Remember that, by using standard DataFrame methods (like **max()** in the example above) we can calculate only a single aggregate value. But, with **agg()** we can calculate more than one aggregate value

at the same time. Since our **transf\_per\_client** object is a **grouped** DataFrame, **agg()** will calculate these aggregates per group.

As an example, if I apply **agg()** with the exact same expressions exposed on Section 5.10.2 with the **transf\_per\_client** DataFrame, instead of a DataFrame with one single row, I get a new DataFrame with nine rows. In each row, I have the total and mean values for a specific user in the input DataFrame.

```
transf_per_client\  
  .agg(  
    F.sum('transferValue').alias('total_value'),  
    F.mean('transferValue').alias('mean_value')  
  )\  
  .show(5)
```

```
+-----+-----+-----+  
|clientNumber|    total_value|    mean_value|  
+-----+-----+-----+  
|      1217|575218.2099999998| 6185.142043010751|  
|      2489|546543.0900000001| 6355.152209302327|  
|      3284|581192.5700000001| 6054.089270833334|  
|       4608|    448784.44| 6233.117222222222|  
|       1297|594869.6699999999|6196.5590624999995|  
+-----+-----+-----+
```

only showing top 5 rows

# 6 Importing data to Spark

## 6.1 Introduction

Another way of creating Spark DataFrames, is to read (or import) data from somewhere and convert it to a Spark DataFrame. Spark can read a variety of file formats, including CSV, Parquet, JSON, ORC and Binary files. Furthermore, Spark can connect to other databases and import tables from them, using ODBC/JDBC connections.

To read (or import) any data to Spark, we use a “read engine”, and there are many different read engines available in Spark. Each engine is used to read a specific file format, or to import data from a specific type of data source, and we access these engines by using the **read** module from your Spark Session object.

## 6.2 Reading data from static files

Static files are probably the easiest way to transport data from one computer to another. Because you just need to copy and paste this file to the other machine, or download it from the internet.

But in order to Spark read any type of static file stored inside your computer, **it always need to know the path to this file**. Every OS have its own file system, and every file in your computer is stored in a specific address of this file system. The “path” to this file is the path (or “steps”) that your computer needs to follow to reach this specific address, where the file is stored.

As we pointed out earlier, to read any static file in Spark, you use one of the available “read engines”, which are in the **spark.read** module of your Spark Session. This means that, each read engine in this module is responsible for reading a specific file format.

If you want to read a CSV file for example, you use the **spark.read.csv()** engine. In contrast, if you want to read a JSON file, you use the **spark.read.json()** engine instead. But no matter what read engine you use, you always give the path to your file to any of these “read engines”.

The main read engines available in Spark for static files are:

- **spark.read.json()**: to read JSON files;
- **spark.read.csv()**: to read CSV files;
- **spark.read.parquet()**: to read Apache Parquet files;

- **spark.read.orc()**: to read ORC (Apache *Optimized Row Columnar* format) files;

For example, to read a JSON file called **sales.json** that is stored in my **Data** folder, I can do this:

```
json_data = spark.read.json("../Data/sales.json")
json_data.show()
```

[Stage 0:>

(0 + 1) / 1]

```
+-----+-----+-----+-----+-----+-----+
|price|product_id|product_name|sale_id|          timestamp|units|
+-----+-----+-----+-----+-----+-----+
| 3.12|      134| Milk 1L Mua| 328711|2022-02-01T22:10:02|  1|
| 1.22|      110|   Coke 350ml| 328712|2022-02-03T11:42:09|  3|
| 4.65|      117|    Pepsi 2L| 328713|2022-02-03T14:22:15|  1|
| 1.22|      110|   Coke 350ml| 328714|2022-02-03T18:33:08|  1|
| 0.85|      341|Trident Mint| 328715|2022-02-04T15:41:36|  1|
+-----+-----+-----+-----+-----+-----+
```

## 6.3 An example with a CSV file

As an example, I have the following CSV file saved in my computer:

```
name,age,job
Jorge,30,Developer
Bob,32,Developer
```

This CSV was saved in a file called **people.csv**, inside a folder called **Data**. So, to read this static file, Spark needs to know the path to this **people.csv** file. In other words, Spark needs to know where this file is stored in my computer, to be able to read it.

In my specific case, considering where this **Data** folder is in my computer, a relative path to it would be **"../Data/"**. Having the path to the folder where **people.csv** is stored, I just need to add this file to the path, resulting in **"../Data/people.csv"**. See in the example below, that I gave this path to the **read.csv()** method of my Spark Session. As a result, Spark will visit this address, and, read the file that is stored there:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
```

```
path = "../Data/people.csv"
```

```
df = spark.read.csv(path)
df.show()
```

```
+-----+-----+
| _c0|_c1|    _c2|
+-----+-----+
| name|age|    job|
|Jorge| 30|Developer|
|  Bob| 32|Developer|
+-----+-----+
```

In the above example, I gave a relative path to the file I wanted to read. But you can provide an absolute path<sup>1</sup> to the file, if you want to. The **people.csv** is located at a very specific folder in my Linux computer, so, the absolute path to this file is pretty long as you can see below. But, if I were in my Windows machine, this absolute path would be something like **"C:\Users\pedro\Documents\Projects\..."**.

```
# The absolute path to `people.csv`:
```

```
path = "/home/pedro/Documents/Projets/Books/Introd-pyspark/Data/people.csv"
```

```
df = spark.read.csv(path)
df.show()
```

```
+-----+-----+
| _c0|_c1|    _c2|
+-----+-----+
| name|age|    job|
|Jorge| 30|Developer|
|  Bob| 32|Developer|
+-----+-----+
```

If you give an invalid path (that is, a path that does not exist in your computer), you will get a **AnalysisException**. In the example below, I try to read a file called **"weird-file.csv"** that (in theory) is located at my current working directory. But when Spark looks inside my current directory, it does not find any file called **"weird-file.csv"**. As a result, Spark raises a **AnalysisException** that warns me about this mistake.

---

<sup>1</sup>That is, the complete path to the file, or, in other words, a path that starts in the root folder of your hard drive.



```
df = spark.read.csv("weird-file.csv")
```

Traceback (most recent call last):

```
...
```

**pyspark.sql.utils.AnalysisException: Path does not exist: file:/home/pedro/Documents/Projects/Books/I**

Every time you face this “Path does not exist” error, it means that Spark did not find the file that you described in the path you gave to **spark.read**. In this case, is very likely that you have a typo or a mistake in your path. Maybe you forgot to add the **.csv** extension to the name of your file. Or maybe you forgot to use the right angled slash (/) instead of the left angled slash (\). Or maybe, you gave the path to folder *x*, when in fact, you wanted to reach the folder *y*.

Sometimes, is useful to list all the files that are stored inside the folder you are trying to access. This way, you can make sure that you are looking at the right folder of your file system. To do that, you can use the **listdir()** function from **os** module of python. As an example, I can list all the files that are stored inside of the **Data** folder in this way:

```
from os import listdir
listdir("../Data/")
```

```
['accounts.csv',
 'user-events.json',
 'transf.csv',
 'people.csv',
 'penguins.csv',
 'livros.txt',
 'transf_reform.csv',
 'logs.json',
 'sales.json',
 'books.txt']
```

## 6.4 Import options

While reading and importing data from any type of data source, Spark will always use the default values for each import option defined by the read engine you are using, unless you explicit ask it to use a different value. Each read engine has its own read/import options.

For example, the **spark.read.orc()** engine has a option called **mergeSchema**. With this option, you can ask Spark to merge the schemas collected from all the ORC part-files. In contrast, the

**spark.read.csv()** engine does not have such option. Because this functionality of “merging schemas” does not make sense with CSV files.

This means that, some import options are specific (or characteristic) of some file formats. For example, the **sep** option (where you define the *separator* character) is used only in the **spark.read.csv()** engine. Because you do not have a special character that behaves as the “separator” in the other file formats (like ORC, JSON, Parquet...). So it does not make sense to have such option in the other read engines.

In the other hand, some import options can co-exist in multiple read engines. For example, the **spark.read.json()** and **spark.read.csv()** have both an **encoding** option. The encoding is a very important information, and Spark needs it to correctly interpret your file. By default, Spark will always assume that your files use the UTF-8 encoding system. Although, this may not be true for your specific case, and for these cases you use this **encoding** option to tell Spark which one to use.

In the next sections, I will break down some of the most used import options for each file format. If you want to see the complete list of import options, you can visit the *Data Source Option* section in the specific part of the file format you are using in the Spark SQL Guide<sup>2</sup>.

To define, or, set a specific import option, you use the **option()** method from a **DataFrameReader** object. To produce this kind of object, you use the **spark.read** module, like in the example below. Each call to the **option()** method is used to set a single import option.

Notice that the “read engine” of Spark (i.e. **csv()**) is the last method called at this chain (or sequence) of steps. In other words, you start by creating a **DataFrameReader** object, then, set the import options, and lastly, you define which “read engine” you want to use.

```
# Creating a `DataFrameReader` object:
df_reader = spark.read
# Setting the import options:
df_reader = df_reader\
    .option("sep", "$")\
    .option("locale", "pt-BR")

# Setting the "read engine" to be used with `.csv()`:
my_data = df_reader\
    .csv("../Data/a-csv-file.csv")
```

If you prefer, you can also merge all these calls together like this:

```
spark.read\ # a `DataFrameReader` object
    .option("sep", "$")\ # Setting the `sep` option
    .option("locale", "pt-BR")\ # Setting the `locale` option
```

---

<sup>2</sup>For example, this *Data Source Option* for Parquet files is located at: <https://spark.apache.org/docs/latest/sql-data-sources-parquet.html#data-source-option>

```
.csv("../Data/a-csv-file.csv") # The "read engine" to be used
```

There are many different import options for each read engine, and you can see the full list in the official documentation for Spark<sup>3</sup>. But let's just give you a brief overview of the probably most popular import options:

- **sep**: sets the separator character for each field and value in the CSV file (defaults to ",");
- **encoding**: sets the character encoding of the file to be read (defaults to "UTF-8");
- **header**: boolean (defaults to **False**), should Spark consider the first line of the file as the header of the DataFrame (i.e. the name of the columns) ?
- **dateFormat** and **timestampFormat**: sets the format for dates and timestamps in the file (defaults to "yyyy-MM-dd" and "yyyy-MM-dd'T'HH:mm:ss[.SSS][XXX]" respectively);

## 6.5 Setting the separator character for CSV files

In this section, we will use the **transf\_reform.csv** file to demonstrate how to set the separator character of a CSV file. This file, contains some data of transfers made in a fictitious bank. It's worth mentioning that this **sep** import option is only available for CSV files.

Let's use the **peek\_file()** function defined below to get a quick peek at the first 5 lines of this file. If you look closely to these lines, you can identify that this CSV file uses the ";" character to separate fields and values, and not the american standard "," character.

```
def peek_file(path, n_lines = 5):
    with open(path) as file:
        lines = [next(file) for i in range(n_lines)]
        text = ''.join(lines)
        print(text)

peek_file("../Data/transf_reform.csv")
```

```
datetime;user;value;transferid;country;description
2018-12-06T22:19:19Z;Eduardo;598.5984;116241629;Germany;
2018-12-06T22:10:34Z;Júlio;4610.955;115586504;Germany;
2018-12-06T21:59:50Z;Nathália;4417.866;115079280;Germany;
2018-12-06T21:54:13Z;Júlio;2739.618;114972398;Germany;
```

This is usually the standard adopted by countries that use a comma to define decimal places in real numbers. In other words, in some countries, the number **3.45** is usually written as **3,45**.

---

<sup>3</sup><https://spark.apache.org/docs/latest/sql-data-sources-csv.html>

Anyway, we know now that the **transf\_reform.csv** file uses a different separator character, so, to correctly read this CSV file into Spark, we need to set the **sep** import option. Since this file comes with the column names in the first line, I also set the **header** import option to read this first line as the column names as well.

```
transf = spark.read\  
    .option("sep", ";")\  
    .option("header", True)\  
    .csv("../Data/transf_reform.csv")  
  
transf.show(5)
```

```
+-----+-----+-----+-----+-----+-----+  
|      datetime|   user|   value|transferid|country|description|  
+-----+-----+-----+-----+-----+-----+  
|2018-12-06T22:19:19Z| Eduardo|598.5984| 116241629|Germany|      null|  
|2018-12-06T22:10:34Z|   Júlio|4610.955| 115586504|Germany|      null|  
|2018-12-06T21:59:50Z|Nathália|4417.866| 115079280|Germany|      null|  
|2018-12-06T21:54:13Z|   Júlio|2739.618| 114972398|Germany|      null|  
|2018-12-06T21:41:27Z|    Ana|1408.261| 116262934|Germany|      null|  
+-----+-----+-----+-----+-----+-----+
```

only showing top 5 rows

## 6.6 Setting the encoding of the file

Spark will always assume that your static files use the UTF-8 encoding system. But, that might not be the case for your specific file. In this situation, you have to tell Spark which is the appropriate encoding system to be used while reading the file. This **encoding** import option is available both for CSV and JSON files.

To do this, you can set the **encoding** import option, with the name of the encoding system to be used. As an example, let's look at the file **books.txt**, which is a CSV file encoded with the ISO-8859-1 system (i.e. the Latin 1 system).

If we use the defaults in Spark, you can see in the result below that some characters in the **Title** column are not correctly interpreted. Remember, this problem occurs because of a mismatch in encoding systems. Spark thinks **books.txt** is using the UTF-8 system, but, in reality, it uses the ISO-8859-1 system:

```
books = spark.read\  
    .option("header", True)
```

```
.csv("../Data/books.txt")
```

```
books.show()
```

```
+-----+-----+-----+
|          Title|          Author| Price|
+-----+-----+-----+
|      O Hobbit|    J. R. R. Tolkien| 40.72|
|Matem tica para E...|Carl P. Simon and...|139.74|
|Microeconomia: um...|      Hal R. Varian| 141.2|
|      A Luneta  mbar|    Philip Pullman| 42.89|
+-----+-----+-----+
```

But if we tell Spark to use the ISO-8859-1 system while reading the file, then, all problems are solved, and all characters in the file are correctly interpreted, as you see in the result below:

```
books = spark.read\
    .option("header", True)\
    .option("encoding", "ISO-8859-1")\
    .csv("../Data/books.txt")
```

```
books.show()
```

```
+-----+-----+-----+
|          Title|          Author| Price|
+-----+-----+-----+
|      O Hobbit|    J. R. R. Tolkien| 40.72|
|Matem tica para E...|Carl P. Simon and...|139.74|
|Microeconomia: um...|      Hal R. Varian| 141.2|
|      A Luneta  mbar|    Philip Pullman| 42.89|
+-----+-----+-----+
```

## 6.7 Setting the format of dates and timestamps

The format that humans write dates and timestamps vary drastically over the world. By default, Spark will assume that the dates and timestamps stored in your file are in the format described by the ISO-8601 standard. That is, the “YYYY-mm-dd”, or, “year-month-day” format.

But this standard might not be the case for your file. For example: the brazilian people usually write dates in the format “dd/mm/YYYY”, or, “day/month/year”; some parts of Spain write dates in the

format “YYYY/dd/mm”, or, “year/day/month”; on Nordic countries (i.e. Sweden, Finland) dates are written in “YYYY.mm.dd” format.

Every format of a date or timestamp is defined by using a string with the codes of each part of the date/timestamp, like the letter ‘Y’ which represents a 4-digit year, or the letter ‘d’ which represents a 2-digit day. You can see the complete list of codes and their description in the official documentation of Spark<sup>4</sup>.

As an example, let's look into the **user-events.json** file. We can see that the dates and timestamps in this file are using the “dd/mm/YYYY” and “dd/mm/YYYY HH:mm:ss” formats respectively.

```
peek_file("../Data/user-events.json", n_lines=3)
```

```
{"dateOfEvent": "15/06/2022", "timeOfEvent": "15/06/2022 14:33:10"
, "userId": "b902e51e-d043-4a66-afc4-a820173e1bb4", "nameOfEvent": "entry"}
{"dateOfEvent": "15/06/2022", "timeOfEvent": "15/06/2022 14:40:08"
, "userId": "b902e51e-d043-4a66-afc4-a820173e1bb4", "nameOfEvent": "click: shop"}
{"dateOfEvent": "15/06/2022", "timeOfEvent": "15/06/2022 15:48:41"
, "userId": "b902e51e-d043-4a66-afc4-a820173e1bb4"
, "nameOfEvent": "select: payment-method"}
```

Date variables are usually interpreted by Spark as string variables. In other words, Spark usually does not convert data that contains dates to the date type of Spark. In order to Spark

```
from pyspark.sql.types import StructType, StructField
from pyspark.sql.types import DateType, StringType, TimestampType

schema = StructType([
    StructField('dateOfEvent', DateType(), True),
    StructField('timeOfEvent', TimestampType(), True),
    StructField('userId', StringType(), True),
    StructField('nameOfEvent', StringType(), True)
])

user_events = spark.read\
    .option("dateFormat", "d/M/y")\
    .option("timestampFormat", "d/M/y k:m:s")\
    .json("../Data/user-events.json", schema = schema)

user_events.show()
```

---

<sup>4</sup><https://spark.apache.org/docs/latest/sql-ref-datetime-pattern.html>

dateOfEvent	timeOfEvent	userId	nameOfEvent
2022-06-15	2022-06-15 14:33:10	b902e51e-d043-4a6...	entry
2022-06-15	2022-06-15 14:40:08	b902e51e-d043-4a6...	click: shop
2022-06-15	2022-06-15 15:48:41	b902e51e-d043-4a6...	select: payment-m...

# 7 Working with SQL in pyspark

## 7.1 Introduction

As we discussed in Chapter 2, Spark is a **multi-language** engine for large-scale data processing. This means that we can build our Spark application using many different languages (like Java, Scala, Python and R). Furthermore, you can also use the Spark SQL module of Spark to translate all of your transformations into pure SQL queries.

In more details, Spark SQL is a Spark module for structured data processing (*Apache Spark Official Documentation* 2022). Because this module works with Spark DataFrames, using SQL, you can translate all transformations that you build with the DataFrame API into a SQL query.

Therefore, you can mix python code with SQL queries very easily in Spark. Virtually all transformations exposed in python throughout this book, can be translated into a SQL query using this module of Spark. We will focus a lot on this exchange between Python and SQL in this chapter.

However, this also means that the Spark SQL module does not handle the transformations produced by the unstructured APIs of Spark, i.e. the Dataset API. Since the Dataset API is not available in **pyspark**, it is not covered in this book.

## 7.2 The sql() method as the main entrypoint

The main entrypoint, that is, the main bridge that connects Spark SQL to Python is the **sql()** method of your Spark Session. This method accepts a SQL query inside a string as input, and will always output a new Spark DataFrame as result. That is why I used the **show()** method right after **sql()**, in the example below, to see what this new Spark DataFrame looked like.

As a first example, lets look at a very basic SQL query, that just select a list of code values:

```
SELECT *  
FROM (  
  VALUES (11), (31), (24), (35)  
) AS List(Codes)
```



To run the above SQL query, and see its results, I must write this query into a string, and give this string to the **sql()** method of my Spark Session. Then, I use the **show()** action to see the actual result rows of data generated by this query:

```
sql_query = '''
SELECT *
FROM (
  VALUES (11), (31), (24), (35)
) AS List(Codes)
'''

spark.sql(sql_query).show()
```

```
+-----+
|Codes|
+-----+
|   11|
|   31|
|   24|
|   35|
+-----+
```

If you want to execute a very short SQL query, is fine to write it inside a single pair of quotation marks (for example "**SELECT \* FROM sales.per\_day**"). However, since SQL queries usually take multiple lines, you can write your SQL query inside a python docstring (created by a pair of three quotation marks), like in the example above.

Having this in mind, every time you want to execute a SQL query, you can use this **sql()** method from the object that holds your Spark Session. So the **sql()** method is the bridge between **pyspark** and SQL. You give it a pure SQL query inside a string, and, Spark will execute it, considering your Spark SQL context.

### 7.2.1 A single SQL statement per run

Is worth pointing out that, although being the main bridge between Python and SQL, the Spark Session **sql()** method can execute only a single SQL statement per run. This means that if you try to execute two sequential SQL statements at the same time with **sql()**, then, Spark SQL will automatically raise a **ParseException** error, which usually complains about an “extra input”.

In the example below, we are doing two very basic steps to SQL. We first create a dummy database with a **CREATE DATABASE** statement, then, we ask SQL to use this new database that we created as the default database of the current session, with a **USE** statement.

```
CREATE DATABASE `dummy`;  
USE `dummy`;
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>  
File "/opt/spark/python/pyspark/sql/session.py", line 1034, in sql  
    return DataFrame(self._jsparkSession.sql(sqlQuery), self)  
File "/opt/spark/python/lib/py4j-0.10.9.5-src.zip/py4j/java_gateway.py", line 1321, in __call__  
File "/opt/spark/python/pyspark/sql/utils.py", line 196, in deco  
    raise converted from None  
pyspark.sql.utils.ParseException:  
Syntax error at or near 'USE': extra input 'USE'(line 3, pos 0)
```

== SQL ==

```
CREATE DATABASE `dummy`;  
USE `dummy`;  
^^^
```

In the example above, Spark complains with a **ParseException**, indicating that we have a syntax error in our query. However, there is nothing wrong about the above SQL statements. They are both correct and valid SQL statements, both semantically and syntactically. In other words, the case above results in a **ParseException** error solely because it contains two different SQL statements.

In essence, the **spark.sql()** method always expect a single SQL statement as input, and, therefore, it will try to parse this input query as a single SQL statement. If it finds multiple SQL statements inside this input string, the method will automatically raise the above error.

Now, be aware that some SQL queries can take multiple lines, but, **still be considered a single SQL statement**. A query started by a **WITH** clause is usually a good example of a SQL query that can group multiple **SELECT** statements, but still be considered a single SQL statement as a whole:

```
-- The query below would execute  
-- perfectly fine inside spark.sql():  
WITH table1 AS (  
    SELECT *  
    FROM somewhere  
)  
  
filtering AS (  
    SELECT *  
    FROM table1
```

```

WHERE dateOfTransaction == CAST("2022-02-02" AS DATE)
)

SELECT *
FROM filtering

```

Another example of a usually big and complex query, that can take multiple lines but still be considered a single SQL statement, is a single **SELECT** statement that selects multiple subqueries that are nested together, like this:

```

-- The query below would also execute
-- perfectly fine inside spark.sql():
SELECT *
FROM (
  -- First subquery.
  SELECT *
  FROM (
    -- Second subquery..
    SELECT *
    FROM (
      -- Third subquery...
      SELECT *
      FROM (
        -- Ok this is enough....
      )
    )
  )
)

```

However, if we had multiple separate **SELECT** statements that were independent on each other, like in the example below, then, **spark.sql()** would issue an **ParseException** error if we tried to execute these three **SELECT** statements inside the same input string.

```

-- These three statements can NOT be executed
-- at the same time inside spark.sql()
SELECT * FROM something;
SELECT * FROM somewhere;
SELECT * FROM sometime;

```

As a conclusion, if you want to easily execute multiple statements, you can use a **for** loop which calls **spark.sql()** for each single SQL statement:

```
statements = '''SELECT * FROM something;
SELECT * FROM somewhere;
SELECT * FROM sometime;'''

statements = statements.split('\n')
for statement in statements:
    spark.sql(statement)
```

## 7.3 Creating SQL Tables in Spark

In real life jobs at the industry, is very likely that your data will be allocated inside a SQL-like database. Spark can connect to a external SQL database through JDBC/ODBC connections, or, read tables from Apache Hive. This way, you can sent your SQL queries to this external database.

However, to expose more simplified examples throughout this chapter, we will use **pyspark** to create a simple temporary SQL table in our Spark SQL context, and use this temporary SQL table in our examples of SQL queries. This way, we avoid the work to connect to some existing SQL database, and, still get to learn how to use SQL queries in **pyspark**.

First, lets create our Spark Session. You can see below that I used the **config()** method to set a specific option of the session called **spark.sql.catalogImplementation**, to the value **"hive"**. This option controls the implementation of the Spark SQL Catalog, which is a core part of the SQL functionality of Spark <sup>1</sup>.

Spark usually complain with a **AnalysisException** error when you try to create SQL tables with this option undefined (or not configured). So, if you decide to follow the examples of this chapter, please always set this option right at the start of your script<sup>2</sup>.

```
from pyspark.sql import SparkSession
spark = SparkSession\
    .builder\
    .config("spark.sql.catalogImplementation", "hive")\
    .getOrCreate()
```

<sup>1</sup>There are some very good materials explaining what is the Spark SQL Catalog, and which is the purpose of it. For a soft introduction, I recommend Sarfaraz Hussain post: <https://medium.com/@sarfarazhussain211/metastore-in-apache-spark-9286097180a4>. For a more technical introduction, see <https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/spark-sql-Catalog.html>.

<sup>2</sup>You can learn more about why this specific option is necessary by looking at this StackOverflow post: <https://stackoverflow.com/questions/50914102/why-do-i-get-a-hive-support-is-required-to-create-hive-table-as-select-error>.

### 7.3.1 TABLEs versus VIEWs

To run a complete SQL query over any Spark DataFrame, you must register this DataFrame in the Spark SQL Catalog of your Spark Session. You can register a Spark DataFrame into this catalog as a physical SQL **TABLE**, or, as a SQL **VIEW**.

If you are familiar with the SQL language and Relational DataBase Management Systems - RDBMS (such as MySQL), you probably already heard of these two types (**TABLE** or **VIEW**) of SQL objects. But if not, we will explain each one in this section. It is worth pointing out that choosing between these two types **does not affect** your code, or your transformations in any way. It just affects the way that Spark SQL stores the table/DataFrame itself.

#### 7.3.1.1 VIEWs are stored as SQL queries or memory pointers

When you register a DataFrame as a SQL **VIEW**, the query to produce this DataFrame is stored, not the DataFrame itself. There are also cases where Spark stores a memory pointer instead, that points to the memory address where this DataFrame is stored in memory. In this perspective, Spark SQL uses this pointer every time it needs to access this DataFrame.

Therefore, when you call (or access) this SQL **VIEW** inside your SQL queries (for example, with a **SELECT \* FROM** statement), Spark SQL will automatically get this SQL **VIEW** “on the fly” (or “on runtime”), by executing the query necessary to build the initial DataFrame that you stored inside this **VIEW**, or, if this DataFrame is already stored in memory, Spark will look at the specific memory address it is stored.

In other words, when you create a SQL **VIEW**, Spark SQL does not store any physical data or rows of the DataFrame. It just stores the SQL query necessary to build your DataFrame. In some way, you can interpret any SQL **VIEW** as an abbreviation to a SQL query, or a “nickname” to an already existing DataFrame.

As a consequence, for most “use case scenarios”, SQL **VIEWs** are easier to manage inside your data pipelines. Because you usually do not have to update them. Since they are calculated from scratch, at the moment you request for them, a SQL **VIEW** will always translate the most recent version of the data.

This means that the concept of a **VIEW** in Spark SQL is very similar to the concept of a **VIEW** in other types of SQL databases, such as the MySQL database. If you read the [official documentation for the CREATE VIEW statement at MySQL](https://dev.mysql.com/doc/refman/8.0/en/create-view.html)<sup>3</sup> you will get a similar idea of a **VIEW**:

The `select_statement` is a **SELECT** statement that provides the definition of the view. (Selecting from the view selects, in effect, using the **SELECT** statement.) ...

---

<sup>3</sup><https://dev.mysql.com/doc/refman/8.0/en/create-view.html>

The above statement, tells us that selecting a **VIEW** causes the SQL engine to execute the expression defined at **select\_statement** using the **SELECT** statement. In other words, in MySQL, a SQL **VIEW** is basically an alias to an existing **SELECT** statement.

### 7.3.1.2 Differences in Spark SQL VIEWS

Although a Spark SQL **VIEW** being very similar to other types of SQL **VIEW** (such as the MySQL type), on Spark applications, SQL **VIEWS** are usually registered as **TEMPORARY VIEWS**<sup>4</sup> instead of standard (and “persistent”) SQL **VIEW** as in MySQL.

At MySQL there is no notion of a “temporary view”, although other popular kinds of SQL databases do have it, [such as the PostgreSQL database](#)<sup>5</sup>. So, a temporary view is not a exclusive concept of Spark SQL. However, is a special type of SQL **VIEW** that is not present in all popular kinds of SQL databases.

In other words, both Spark SQL and MySQL support the **CREATE VIEW** statement. In contrast, statements such as **CREATE TEMPORARY VIEW** and **CREATE OR REPLACE TEMPORARY VIEW** are available in Spark SQL, but not in MySQL.

### 7.3.1.3 Registering a Spark SQL VIEW in the Spark SQL Catalog

In **pyspark**, you can register a Spark DataFrame as a temporary SQL **VIEW** with the **createTempView()** or **createOrReplaceTempView()** DataFrame methods. These methods are equivalent to **CREATE TEMPORARY VIEW** and **CREATE OR REPLACE TEMPORARY VIEW** SQL statements of Spark SQL, respectively.

In essence, these methods register your Spark DataFrame as a temporary SQL **VIEW**, and have a single input, which is the name you want to give to this new SQL **VIEW** you are creating inside a string:

```
# To save the `df` DataFrame as a SQL VIEW, use one of the methods below:
df.createTempView('example_view')
df.createOrReplaceTempView('example_view')
```

After we executed the above statements, we can now access and use the **df** DataFrame in any SQL query, like in the example below:

```
sql_query = '''
SELECT *
FROM example_view
WHERE value > 20
```

---

<sup>4</sup>I will explain more about the meaning of “temporary” at Section [7.3.2](#).

<sup>5</sup><https://www.postgresql.org/docs/current/sql-createview.html>

```
...
```

```
spark.sql(sql_query).show()
```

[Stage 0:>

(0 + 1) / 1]

```
+---+-----+-----+
| id|value|    date|
+---+-----+-----+
|  1| 28.3|2021-01-01|
|  3| 20.1|2021-01-02|
+---+-----+-----+
```

So you use the `createTempView()` or `createOrReplaceTempView()` methods when you want to make a Spark DataFrame created in `pyspark` (that is, a python object), available to Spark SQL.

Besides that, you also have the option to create a temporary **VIEW** by using pure SQL statements through the `sql()` method. However, when you create a temporary **VIEW** using pure SQL, you can only use (inside this **VIEW**) native SQL objects that are already stored inside your Spark SQL Context.

This means that you cannot make a Spark DataFrame created in python available to Spark SQL, by using a pure SQL inside the `sql()` method. To do this, you have to use the DataFrame methods `createTempView()` and `createOrReplaceTempView()`.

As an example, the query below uses pure SQL statements to create the **active\_brazilian\_users** temporary **VIEW**, which selects an existing SQL table called **hubspot.user\_mails**:

```
CREATE TEMPORARY VIEW active_brazilian_users AS
SELECT *
FROM hubspot.user_mails
WHERE state == 'Active'
AND country_location == 'Brazil'
```

Temporary **VIEWS** like the one above (which are created from pure SQL statements being executed inside the `sql()` method) are kind of unusual in Spark SQL. Because you can easily avoid the work of creating a **VIEW** by using Common Table Expression (CTE) on a **WITH** statement, like in the query below:

```
WITH active_brazilian_users AS (
  SELECT *
```

```

FROM hubspot.user_mails
WHERE state == 'Active'
AND country_location == 'Brazil'
)

SELECT A.user, SUM(sale_value), B.user_email
FROM sales.sales_per_user AS A
INNER JOIN active_brazilian_users AS B
GROUP BY A.user, B.user_email

```

Just as a another example, you can also run a SQL query that creates a persistent SQL **VIEW** (that is, without the **TEMPORARY** clause). In the example below, I am saving the simple query that I showed at the beginning of this chapter inside a **VIEW** called **list\_of\_codes**. This **CREATE VIEW** statement, register a persistent SQL **VIEW** in the SQL Catalog.

```

sql_query = '''
CREATE OR REPLACE VIEW list_of_codes AS
SELECT *
FROM (
  VALUES (11), (31), (24), (35)
) AS List(Codes)
'''

spark.sql(sql_query)

```

**DataFrame[]**

Now, every time I want to use this SQL query that selects a list of codes, I can use this **list\_of\_codes** as a shortcut:

```
spark.sql("SELECT * FROM list_of_codes").show()
```

```

+-----+
|Codes|
+-----+
|  11 |
|  31 |
|  24 |
|  35 |
+-----+

```



#### 7.3.1.4 TABLEs are stored as physical tables

In the other hand, SQL **TABLEs** are the “opposite” of SQL **VIEWS**. That is, SQL **TABLEs** are stored as physical tables inside the SQL database. In other words, each one of the rows of your table are stored inside the SQL database.

Because of this characteristic, when dealing with huge amounts of data, SQL **TABLEs** are usually faster to load and transform. Because you just have to read the data stored on the database, you do not need to calculate it from scratch every time you use it.

But, as a collateral effect, you usually have to physically update the data inside this **TABLE**, by using, for example, **INSERT INTO** statements. In other words, when dealing with SQL **TABLEs** you usually need to create (and manage) data pipelines that are responsible for periodically update and append new data to this SQL **TABLE**, and this might be a big burden to your process.

#### 7.3.1.5 Registering Spark SQL TABLEs in the Spark SQL Catalog

In **pyspark**, you can register a Spark DataFrame as a SQL **TABLE** with the **write.saveAsTable()** DataFrame method. This method accepts, as first input, the name you want to give to this SQL **TABLE** inside a string.

```
# To save the `df` DataFrame as a SQL TABLE:  
df.write.saveAsTable('example_table')
```

As you expect, after we registered the DataFrame as a SQL table, we can now run any SQL query over **example\_table**, like in the example below:

```
spark.sql("SELECT SUM(value) FROM example_table").show()
```

```
+-----+  
|sum(value)|  
+-----+  
|      76.8|  
+-----+
```

You can also use pure SQL queries to create an empty SQL **TABLE** from scratch, and then, feed this table with data by using **INSERT INTO** statements. In the example below, we create a new database called **examples**, and, inside of it, a table called **code\_brazil\_states**. Then, we use multiple **INSERT INTO** statements to populate this table with few rows of data.

```

all_statements = '''CREATE DATABASE `examples`;
USE `examples`;
CREATE TABLE `code_brazil_states` (`code` INT, `state_name` STRING);
INSERT INTO `code_brazil_states` VALUES (31, "Minas Gerais");
INSERT INTO `code_brazil_states` VALUES (15, "Pará");
INSERT INTO `code_brazil_states` VALUES (41, "Paraná");
INSERT INTO `code_brazil_states` VALUES (25, "Paraíba");'''

statements = all_statements.split('\n')
for statement in statements:
    spark.sql(statement)

```

We can see now this new physical SQL table using a simple query like this:

```

spark\
.sql('SELECT * FROM examples.code_brazil_states')\
.show()

```

```

+----+-----+
|code| state_name|
+----+-----+
| 41|      Paraná|
| 31| Minas Gerais|
| 15|        Pará|
| 25|     Paraíba|
+----+-----+

```

### 7.3.1.6 The different save “modes”

There are other arguments that you might want to use in the `write.saveAsTable()` method, like the `mode` argument. This argument controls how Spark will save your data into the database. By default, `write.saveAsTable()` uses the `mode = "error"` by default. In this mode, Spark will look if the table you referenced already exists, before it saves your data.

Let’s get back to the code we showed before (which is reproduced below). In this code, we asked Spark to save our data into a table called `"example_table"`. Spark will look if a table with this name already exists. If it does, then, Spark will raise an error that will stop the process (i.e. no data is saved).

```

df.write.saveAsTable('example_table')

```

Raising an error when you do not want to accidentally affect a SQL table that already exist, is a good practice. But, you might want to not raise an error in this situation. In case like this, you might want to

just ignore the operation, and get on with your life. For cases like this, `write.saveAsTable()` offers the `mode = "ignore"`.

So, in the code example below, we are trying to save the `df` DataFrame into a table called `example_table`. But if this `example_table` already exist, Spark will just silently ignore this operation, and will not save any data.

```
df.write.saveAsTable('example_table', mode = "ignore")
```

In addition, `write.saveAsTable()` offers two more different modes, which are `mode = "overwrite"` and `mode = "append"`. When you use one these two modes, Spark **will always save your data**, no matter if the SQL table you are trying to save into already exist or not. In essence, these two modes control whether Spark will delete or keep previous rows of the SQL table intact, before it saves any new data.

When you use `mode = "overwrite"`, Spark will automatically rewrite/replace the entire table with the current data of your DataFrame. In contrast, when you use `mode = "append"`, Spark will just append (or insert, or add) this data into the table. The subfigures at Figure 7.1 demonstrates these two modes visually.

You can see the full list of arguments of `write.SaveAsTable()`, and their description by [looking at the documentation](#)<sup>6</sup>.

### 7.3.2 Temporary versus Persistent sources

When you register any Spark DataFrame as a SQL **TABLE**, it becomes a persistent source. Because the contents, the data, the rows of the table are stored on disk, inside a database, and can be accessed any time, even after you close or restart your computer (or your Spark Session). In other words, it becomes “persistent” as in the sense of “it does not die”.

As another example, when you save a specific SQL query as a SQL **VIEW** with the **CREATE VIEW** statement, this SQL **VIEW** is saved inside the database. As a consequence, it becomes a persistent source as well, and can be accessed and reused in other Spark Sessions, unless you explicit drop (or “remove”) this SQL **VIEW** with a **DROP VIEW** statement.

However, with methods like `createTempView()` and `createOrReplaceTempView()` you register your Spark DataFrame as a *temporary* SQL **VIEW**. This means that the life (or time of existence) of this **VIEW** is tied to your Spark Session. In other words, it will exist in your Spark SQL Catalog only for the duration of your Spark Session. When you close your Spark Session, this **VIEW** just dies. When you start a new Spark Session it does not exist anymore. As a result, you have to register your DataFrame again at the catalog to use it one more time.

---

<sup>6</sup><https://spark.apache.org/docs/3.1.2/api/python/reference/api/pyspark.sql.DataFrameWriter.saveAsTable>

mode = "overwrite"



(a) Mode overwrite

mode = "append"



(b) Mode append

Figure 7.1: How Spark saves your data with different "save modes"

### 7.3.3 Spark SQL Catalog is the bridge between SQL and pyspark

Remember, to run SQL queries over any Spark DataFrame, you must register this DataFrame into the Spark SQL Catalog. Because of it, this Spark SQL Catalog works almost as the bridge that connects the python objects that hold your Spark DataFrames to the Spark SQL context. Without it, Spark SQL will not find your Spark DataFrames. As a result, it can not run any SQL query over it.

When you try to use a DataFrame that is not currently registered at the Spark SQL Catalog, Spark will automatically raise a **AnalysisException**, like in the example below:

```
spark\  
  .sql("SELECT * FROM this.does_not_exist")\  
  .show()
```

**AnalysisException: Table or view not found**

The methods **saveAsTable()**, **createTempView()** and **createOrReplaceTempView()** are the main methods to register your Spark DataFrame into this Spark SQL Catalog. This means that you have to use one of these methods before you run any SQL query over your Spark DataFrame.

## 7.4 The penguins DataFrame

Over the next examples in this chapter, we will explore the **penguins** DataFrame. This is the **penguins** dataset from the [palmerpenguins R library](https://allisonhorst.github.io/palmerpenguins/reference/penguins.html). It stores data of multiple measurements of penguin species from the islands in Palmer Archipelago.

These measurements include size (flipper length, body mass, bill dimensions) and sex, and they were collected by researchers of the Antarctica LTER program, a member of the Long Term Ecological Research Network. If you want to understand more about each field/column present in this dataset, I recommend you to read the [official documentation of this dataset](https://allisonhorst.github.io/palmerpenguins/reference/penguins.html)<sup>7</sup>.

To get this data, you can download the CSV file called **penguins.csv** (remember that this CSV can be downloaded from the book repository<sup>8</sup>). In the code below, I am reading this CSV file and creating a Spark DataFrame with its data. Then, I register this Spark DataFrame as a SQL temporary view (called **penguins\_view**) using the **createOrReplaceTempView()** method.

```
path = "../Data/penguins.csv"  
penguins = spark.read\  
  .csv(path, header = True)
```

<sup>7</sup><https://allisonhorst.github.io/palmerpenguins/reference/penguins.html>

<sup>8</sup><https://github.com/pedropark99/Introd-pyspark/tree/main/Data>

```
penguins.createOrReplaceTempView('penguins_view')
```

After these commands, I have now a SQL view called **penguins\_view** registered in my Spark SQL context, which I can query it, using pure SQL:

```
spark.sql('SELECT * FROM penguins_view').show(5)
```

```
+-----+-----+-----+-----+-----+-----+
|species|  island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|
+-----+-----+-----+-----+-----+-----+
| Adelie|Torgersen|      39.1|      18.7|        181|      3750|
| Adelie|Torgersen|      39.5|      17.4|        186|      3800|
| Adelie|Torgersen|      40.3|       18|        195|      3250|
| Adelie|Torgersen|     null|     null|        null|       null|
| Adelie|Torgersen|      36.7|      19.3|        193|      3450|
+-----+-----+-----+-----+-----+-----+
```

only showing top 5 rows

... with 2 more columns: sex, year

## 7.5 Selecting your Spark DataFrames

An obvious way to access any SQL **TABLE** or **VIEW** registered in your Spark SQL context, is to select it, through a simple **SELECT \* FROM** statement, like we saw in the previous examples. However, it can be quite annoying to type “SELECT \* FROM” every time you want to use a SQL **TABLE** or **VIEW** in Spark SQL.

That is why Spark offers a shortcut to us, which is the **table()** method of your Spark session. In other words, the code **spark.table("table\_name")** is a shortcut to **spark.sql("SELECT \* FROM table\_name")**. They both mean the same thing. For example, we could access **penguins\_view** as:

```
spark\
  .table('penguins_view')\
  .show(5)
```

```
+-----+-----+-----+-----+-----+-----+
|species|  island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|
+-----+-----+-----+-----+-----+-----+
| Adelie|Torgersen|      39.1|      18.7|        181|      3750|
| Adelie|Torgersen|      39.5|      17.4|        186|      3800|
+-----+-----+-----+-----+-----+-----+
```

Adelie Torgersen	40.3	18	195	3250
Adelie Torgersen	null	null	null	null
Adelie Torgersen	36.7	19.3	193	3450

+-----+-----+-----+-----+-----+-----+

only showing top 5 rows  
... with 2 more columns: sex, year

## 7.6 Executing SQL expressions

As I noted at Section 4.2, columns of a Spark DataFrame (or objects of class **Column**) are closely related to expressions. As a result, you usually use and execute expressions in Spark when you want to transform (or mutate) columns of a Spark DataFrame.

This is no different for SQL expressions. A SQL expression is basically any expression you would use on the **SELECT** statement of your SQL query. As you can probably guess, since they are used in the **SELECT** statement, these expressions are used to transform columns of a Spark DataFrame.

There are many column transformations that are particularly verbose and expensive to write in “pure” **pyspark**. But you can use a SQL expression in your favor, to translate this transformation into a more short and concise form. Virtually any expression you write in **pyspark** can be translated into a SQL expression.

To execute a SQL expression, you give this expression inside a string to the **expr()** function from the **pyspark.sql.functions** module. Since expressions are used to transform columns, you normally use the **expr()** function inside a **withColumn()** or a **select()** DataFrame method, like in the example below:

```
from pyspark.sql.functions import expr

spark\
  .table('penguins_view')\
  .withColumn(
    'specie_island',
    expr("CONCAT(species, '_', island)")
  )\
  .withColumn(
    'sex_short',
    expr("CASE WHEN sex == 'male' THEN 'M' ELSE 'F' END")
  )\
  .select('specie_island', 'sex_short')\
  .show(5)
```

```

+-----+-----+
|  specie_island|sex_short|
+-----+-----+
|Adelie_Torgersen|      M|
|Adelie_Torgersen|      F|
|Adelie_Torgersen|      F|
|Adelie_Torgersen|      F|
|Adelie_Torgersen|      F|
+-----+-----+

```

only showing top 5 rows

I particularly like to write “if-else” or “case-when” statements using a pure **CASE WHEN** SQL statement inside the **expr()** function. By using this strategy you usually get a more simple statement that translates the intention of your code in a cleaner way. But if I wrote the exact same **CASE WHEN** statement above using pure **pyspark** functions, I would end up with a shorter (but “less clean”) statement using the **when()** and **otherwise()** functions:

```

from pyspark.sql.functions import (
    when, col,
    concat, lit
)

spark\
    .table('penguins_view')\
    .withColumn(
        'specie_island',
        concat('species', lit('_'), 'island')
    )\
    .withColumn(
        'sex_short',
        when(col("sex") == 'male', 'M')\
            .otherwise('F')
    )\
    .select('specie_island', 'sex_short')\
    .show(5)

```

```

+-----+-----+
|  specie_island|sex_short|
+-----+-----+
|Adelie_Torgersen|      M|
|Adelie_Torgersen|      F|
|Adelie_Torgersen|      F|

```



```
|Adelie_Torgersen|      F|
|Adelie_Torgersen|      F|
+-----+
only showing top 5 rows
```

## 7.7 Every DataFrame transformation in Python can be translated into SQL

All DataFrame API transformations that you write in Python (using **pyspark**) can be translated into SQL queries/expressions using the Spark SQL module. Since the DataFrame API is a core part of **pyspark**, the majority of python code you write with **pyspark** can be translated into SQL queries (if you want to).

Is worth pointing out, that, no matter which language you choose (Python or SQL), they are both further compiled to the same base instructions. The end result is that the Python code you write and his SQL translated version **will perform the same** (they have the same efficiency), because they are compiled to the same instructions before being executed by Spark.

### 7.7.1 DataFrame methods are usually translated into SQL keywords

When you translate the methods from the python **DataFrame** class (like **orderBy()**, **select()** and **where()**) into their equivalents in Spark SQL, you usually get SQL keywords (like **ORDER BY**, **SELECT** and **WHERE**).

For example, if I needed to get the top 5 penguins with the biggest body mass at **penguins\_view**, that had sex equal to **"female"**, and, ordered by bill length, I could run the following python code:

```
from pyspark.sql.functions import col
top_5 = penguins\
    .where(col('sex') == 'female')\
    .orderBy(col('body_mass_g').desc())\
    .limit(5)

top_5\
    .orderBy('bill_length_mm')\
    .show()
```

```
+-----+-----+-----+-----+-----+-----+
|species|island|bill_length_mm|bill_depth_mm|flipper_length_mm|body_mass_g|
+-----+-----+-----+-----+-----+-----+
|Adelie  |Torgersen|190|18|181|3500|
|Adelie  |Torgersen|195|19|186|3800|
|Adelie  |Torgersen|198|20|193|3800|
|Adelie  |Torgersen|201|21|197|3900|
|Adelie  |Torgersen|206|22|201|4200|
```

	Gentoo Biscoe	44.9	13.3	213	5100
	Gentoo Biscoe	45.1	14.5	207	5050
	Gentoo Biscoe	45.2	14.8	212	5200
	Gentoo Biscoe	46.5	14.8	217	5200
	Gentoo Biscoe	49.1	14.8	220	5150

+-----+-----+-----+-----+-----+-----+

... with 2 more columns: sex, year

I could translate the above python code to the following SQL query:

```
WITH top_5 AS (
  SELECT *
  FROM penguins_view
  WHERE sex == 'female'
  ORDER BY body_mass_g DESC
  LIMIT 5
)

SELECT *
FROM top_5
ORDER BY bill_length_mm
```

Again, to execute the above SQL query inside **pyspark** we need to give this query as a string to the **sql()** method of our Spark Session, like this:

```
query = '''
WITH top_5 AS (
  SELECT *
  FROM penguins_view
  WHERE sex == 'female'
  ORDER BY body_mass_g DESC
  LIMIT 5
)

SELECT *
FROM top_5
ORDER BY bill_length_mm
'''

# The same result of the example above
spark.sql(query)
```

```
DataFrame[species: string, island: string, bill_length_mm: string
, bill_depth_mm: string, flipper_length_mm: string, body_mass_g: string
, sex: string, year: string]
```

## 7.7.2 Spark functions are usually translated into SQL functions

Every function from the **pyspark.sql.functions** module you might use to describe your transformations in python, can be directly used in Spark SQL. In other words, every Spark function that is accessible in python, is also accessible in Spark SQL.

When you translate these python functions into SQL, they usually become a pure SQL function with the same name. For example, if I wanted to use the **regexp\_extract()** python function, from the **pyspark.sql.functions** module in Spark SQL, I just use the **REGEXP\_EXTRACT()** SQL function. The same occurs to any other function, like the **to\_date()** function for example.

```
from pyspark.sql.functions import to_date, regexp_extract
# `df1` and `df2` are both equal. Because they both
# use the same `to_date()` and `regexp_extract()` functions
df1 = (spark
      .table('penguins_view')
      .withColumn(
        'extract_number',
        regexp_extract('bill_length_mm', '[0-9]+', 0)
      )
      .withColumn('date', to_date('year', 'y'))
      .select(
        'bill_length_mm', 'year',
        'extract_number', 'date'
      )
    )

df2 = (spark
      .table('penguins_view')
      .withColumn(
        'extract_number',
        expr("REGEXP_EXTRACT(bill_length_mm, '[0-9]+', 0)")
      )
      .withColumn('date', expr("TO_DATE(year, 'y')"))
      .select(
        'bill_length_mm', 'year',
        'extract_number', 'date'
      )
    )
```

```
)
```

```
df2.show(5)
```

```
+-----+-----+-----+-----+
|bill_length_mm|year|extract_number|      date|
+-----+-----+-----+-----+
|          39.1|2007|          39|2007-01-01|
|          39.5|2007|          39|2007-01-01|
|          40.3|2007|          40|2007-01-01|
|          null|2007|         null|2007-01-01|
|          36.7|2007|          36|2007-01-01|
+-----+-----+-----+-----+
```

only showing top 5 rows

This is very handy. Because for every new python function from the **pyspark.sql.functions** module, that you learn how to use, you automatically learn how to use in Spark SQL as well, because is the same function, with the basically the same name and arguments.

As an example, I could easily translate the above transformations that use the **to\_date()** and **regexp\_extract()** python functions, into the following SQL query (that I could easily execute trough the **sql()** Spark Session method):

```
SELECT
  bill_length_mm, year,
  REGEXP_EXTRACT(bill_length_mm, '[0-9]+', 0) AS extract_number,
  TO_DATE(year, 'y') AS date
FROM penguins_view
```

## 8 Tools for string manipulation

Many of the world's data is represented (or stored) as text (or string variables). As a consequence, is very important to know the tools available to process and transform this kind of data, in any platform you use. In this chapter, we will focus on these tools.

Most of the functionality available in **pyspark** to process text data comes from functions available at the **pyspark.sql.functions** module. This means that processing and transforming text data in Spark usually involves applying a function on a column of a Spark DataFrame (by using DataFrame methods such as **withColumn()** and **select()**).

### 8.1 The logs DataFrame

Over the next examples in this chapter, we will use the **logs** DataFrame, which contains various log messages registered at a fictitious IP address. The data that represents this DataFrame is freely available through the **logs.json** file, which you can download from the official repository of this book<sup>1</sup>.

Each line of this JSON file contains a message that was recorded by the logger of a fictitious system. Each log message have three main parts, which are: 1) the type of message (warning - **WARN**, information - **INFO**, error - **ERROR**); 2) timestamp of the event; 3) the content of the message. In the example below, we have an example of message:

[INFO]: 2022-09-05 03:35:01.43 Looking for workers at South America region;

To import **logs.json** file into a Spark DataFrame, I can use the following code:

```
path = '../Data/logs.json'
logs = spark.read.json(path)
n_truncate = 50
logs.show(5, truncate = n_truncate)
```

```
+-----+-----+
|          ip|          message|
+-----+-----+
| 1.0.104.27 |[INFO]: 2022-09-05 03:35:01.43 Looking for work...|
```

<sup>1</sup><https://github.com/pedropark99/Introd-pyspark/tree/main/Data>

```
| 1.0.104.27 |[WARN]: 2022-09-05 03:35:58.007 Workers are una...|
| 1.0.104.27 |[INFO]: 2022-09-05 03:40:59.054 Looking for wor...|
| 1.0.104.27 |[INFO]: 2022-09-05 03:42:24 3 Workers were acqu...|
| 1.0.104.27 |[INFO]: 2022-09-05 03:42:37 Initializing instan...|
+-----+
only showing top 5 rows
```

By default, when we use the **show()** action to see the contents of our Spark DataFrame, Spark will always truncate (or cut) any value in the DataFrame that is more than 20 characters long. Since the logs messages in the **logs.json** file are usually much longer than 20 characters, I am using the **truncate** argument of **show()** in the example above, to avoid this behaviour.

By setting this argument to 50, I am asking Spark to truncate (or cut) values at the 50th character (instead of the 20th). By doing this, you (reader) can actually see a much more significant part of the logs messages in the result above.

## 8.2 Changing the case of letters in a string

Probably the most basic string transformation that exists is to change the case of the letters (or characters) that compose the string. That is, to raise specific letters to upper-case, or reduce them to lower-case, and vice-versa.

As a first example, lets go back to the **logs** DataFrame, and try to change all messages in this DataFrame to lower case, upper case and title case, by using the **lower()**, **upper()**, and **initcap()** functions from the **pyspark.sql.functions** module.

```
from pyspark.sql.functions import (
    lower,
    upper,
    initcap
)

m = logs.select('message')
# Change to lower case:
m.withColumn('message', lower('message'))\
  .show(5, truncate = n_truncate)
```

```
+-----+
|                                     message|
+-----+
|[info]: 2022-09-05 03:35:01.43 looking for work...|
```

```

|[warn]: 2022-09-05 03:35:58.007 workers are una...|
|[info]: 2022-09-05 03:40:59.054 looking for wor...|
|[info]: 2022-09-05 03:42:24 3 workers were acqu...|
|[info]: 2022-09-05 03:42:37 initializing instan...|
+-----+
only showing top 5 rows

```

```

# Change to upper case:
m.withColumn('message', upper('message'))\
  .show(5, truncate = n_truncate)

```

```

+-----+
|                                     message|
+-----+
|[INFO]: 2022-09-05 03:35:01.43 LOOKING FOR WORK...|
|[WARN]: 2022-09-05 03:35:58.007 WORKERS ARE UNA...|
|[INFO]: 2022-09-05 03:40:59.054 LOOKING FOR WOR...|
|[INFO]: 2022-09-05 03:42:24 3 WORKERS WERE ACQU...|
|[INFO]: 2022-09-05 03:42:37 INITIALIZING INSTAN...|
+-----+
only showing top 5 rows

```

```

# Change to title case
# (first letter of each word is upper case):
m.withColumn('message', initcap('message'))\
  .show(5, truncate = n_truncate)

```

```

+-----+
|                                     message|
+-----+
|[info]: 2022-09-05 03:35:01.43 Looking For Work...|
|[warn]: 2022-09-05 03:35:58.007 Workers Are Una...|
|[info]: 2022-09-05 03:40:59.054 Looking For Wor...|
|[info]: 2022-09-05 03:42:24 3 Workers Were Acqu...|
|[info]: 2022-09-05 03:42:37 Initializing Instan...|
+-----+
only showing top 5 rows

```

## 8.3 Calculating string length

In Spark, you can use the **length()** function to get the length (i.e. the number of characters) of a string. In the example below, we can see that the first log message is 74 characters long, while the second log message have 112 characters.

```
from pyspark.sql.functions import length
logs\
  .withColumn('length', length('message'))\
  .show(5)
```

```
+-----+-----+-----+
|          ip|          message|length|
+-----+-----+-----+
|  1.0.104.27 |[INFO]: 2022-09-0...|    74|
|  1.0.104.27 |[WARN]: 2022-09-0...|   112|
|  1.0.104.27 |[INFO]: 2022-09-0...|    75|
|  1.0.104.27 |[INFO]: 2022-09-0...|    94|
|  1.0.104.27 |[INFO]: 2022-09-0...|    65|
+-----+-----+-----+
```

only showing top 5 rows

## 8.4 Trimming or removing spaces from strings

The process of removing unnecessary spaces from strings is usually called “trimming”. In Spark, we have three functions that do this process, which are:

- **trim()**: removes spaces from both sides of the string;
- **ltrim()**: removes spaces from the left side of the string;
- **rtrim()**: removes spaces from the right side of the string;

```
from pyspark.sql.functions import (
    trim, rtrim, ltrim
)

logs\
  .select('ip')\
  .withColumn('ip_trim', trim('ip'))\
  .withColumn('ip_ltrim', ltrim('ip'))\
  .withColumn('ip_rtrim', rtrim('ip'))\
```



```
.show(5)
```

```

+-----+-----+-----+-----+
|          ip|  ip_trim|  ip_ltrim|  ip_rtrim|
+-----+-----+-----+-----+
|  1.0.104.27 |1.0.104.27|1.0.104.27 |  1.0.104.27|
|  1.0.104.27 |1.0.104.27|1.0.104.27 |  1.0.104.27|
|  1.0.104.27 |1.0.104.27|1.0.104.27 |  1.0.104.27|
|  1.0.104.27 |1.0.104.27|1.0.104.27 |  1.0.104.27|
|  1.0.104.27 |1.0.104.27|1.0.104.27 |  1.0.104.27|
+-----+-----+-----+-----+

```

only showing top 5 rows

For the most part, I tend to remove these unnecessary strings when I want to: 1) tidy the values; 2) avoid weird and confusing mistakes in filters on my DataFrame. The second case is worth describing in more details.

Let's suppose you wanted to filter all rows from the **logs** DataFrame where **ip** is equal to the **1.0.104.27** IP address. However, you can see in the result above, that I get nothing. Not a single row of result.

```
from pyspark.sql.functions import col
logs.filter(col('ip') == "1.0.104.27")\
.show(5)
```

```

+---+-----+
| ip|message|
+---+-----+
+---+-----+

```

But if you see the result of the previous example (where we applied the three versions of “trim functions”), you know that this IP address **1.0.104.27** exists in the DataFrame. You know that the filter above should find values for this IP address. So why it did not find any rows?

The answer is these annoying (and hidden) spaces on both sides of the values from the **ip** column. If we remove these unnecessary spaces from the values of the **ip** column, we suddenly find the rows that we were looking for.

```
logs.filter(trim(col('ip')) == "1.0.104.27")\
.show(5)
```

```

+-----+-----+
|          ip|          message|
+-----+-----+
| 1.0.104.27 |[INFO]: 2022-09-0...|
| 1.0.104.27 |[WARN]: 2022-09-0...|
| 1.0.104.27 |[INFO]: 2022-09-0...|
| 1.0.104.27 |[INFO]: 2022-09-0...|
| 1.0.104.27 |[INFO]: 2022-09-0...|
+-----+-----+
only showing top 5 rows

```

## 8.5 Extracting substrings

There are five main functions that we can use in order to extract substrings of a string, which are:

- **substring()** and **substr()**: extract a single substring based on a start position and the length (number of characters) of the collected substring<sup>2</sup>;
- **substring\_index()**: extract a single substring based on a single delimiter<sup>3</sup>;
- **split()**: extract one or multiple substrings based on common delimiter;
- **regexp\_extract()**: extracts substrings from a given string that match a specified regular expression pattern;

You can obviously extract a substring that matches a particular regex (regular expression) by using the **regexp\_extract()** function. However, I will describe this function, and the regex functionality available in **pyspark** at Section 8.7, or, more specifically, at Section 8.7.5. For now, just understand that you can also use regex to extract substrings from your text data.

### 8.5.1 A substring based on a start position and length

The **substring()** and **substr()** functions they both work the same way. However, they come from different places. The **substring()** function comes from the **spark.sql.functions** module, while the **substr()** function is actually a method from the **Column** class.

One interesting aspect of these functions, is that they both use a one-based index, instead of a zero-based index. This means that the first character in the full string is identified by the index 1, instead of the index 0.

The first argument in both function is the index that identifies the start of the substring. If you set this argument to, let's say, 4, it means that the substring you want to extract starts at the 4th character in the input string.

<sup>2</sup>These functions uses a one based index, and not zero based index.

<sup>3</sup>These functions uses a one based index, and not zero based index.

The second argument is the amount of characters in the substring, or, in other words, it's length. For example, if you set this argument to 10, it means that the function will extract the substring that is formed by walking  $10 - 1 = 9$  characters ahead from the start position you specified at the first argument. We can also interpret this as: the function will walk ahead on the string, from the start position, until it gets a substring that is 10 characters long.

In the example below, we are extracting the substring that starts at the second character (index 2) and ends at the sixth character (index 6) in the string.

```
from pyspark.sql.functions import col, substring
# `df1` and `df2` are equal, because
# they both mean the same thing
df1 = (logs
      .withColumn('sub', col('message').substr(2, 5))
      )

df2 = (logs
      .withColumn('sub', substring('message', 2, 5))
      )

df2.show(5)
```

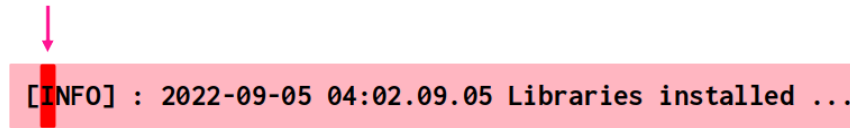
```
+-----+-----+-----+
|          ip|          message|  sub|
+-----+-----+-----+
|  1.0.104.27 | [INFO]: 2022-09-0... | INFO|
|  1.0.104.27 | [WARN]: 2022-09-0... | WARN|
|  1.0.104.27 | [INFO]: 2022-09-0... | INFO|
|  1.0.104.27 | [INFO]: 2022-09-0... | INFO|
|  1.0.104.27 | [INFO]: 2022-09-0... | INFO|
+-----+-----+-----+
only showing top 5 rows
```

Just to be very clear on how **substring()** and **substr()** both works. The Figure 8.1 illustrates the result of the above code.

## 8.5.2 A substring based on a delimiter

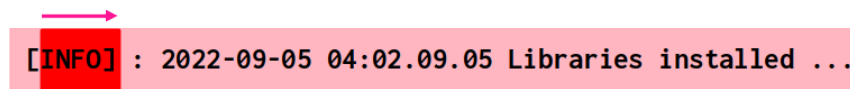
The **substring\_index()** function works very differently. It collects the substring formed between the start of the string, and the nth occurrence of a particular character.

Start here at the 2nd character



[INFO] : 2022-09-05 04:02.09.05 Libraries installed ...

Then, move forward until we have a 5 characters long substring



[INFO] : 2022-09-05 04:02.09.05 Libraries installed ...

Figure 8.1: How **substring()** and **substr()** works

For example, if you ask **substring\_index()** to search for the 3rd occurrence of the character **\$** in your string, the function will return to you the substring formed by all characters that are between the start of the string until the 3rd occurrence of this character **\$**. You can also ask **substring\_index()** to read backwards. That is, to start the search on the end of the string, and move backwards in the string until it gets to the 3rd occurrence of this character **\$**.

As an example, let's look at the 10th log message present in the **logs** DataFrame. I used the **collect()** DataFrame method to collect this message into a raw python string, so we can see the full message.

```
from pyspark.sql.functions import monotonically_increasing_id

mes_10th = (
    logs
    .withColumn(
        'row_id',
        monotonically_increasing_id()
    )
    .where(col('row_id') == 9)
)

message = mes_10th.collect()[0]['message']
print(message)
```

```
[INFO]: 2022-09-05 04:02:09.05 Libraries installed: pandas, flask, numpy
, spark_map, pyspark
```

We can see that this log message is listing a set of libraries that were installed somewhere. Suppose you want to collect the first and the last libraries in this list. How would you do it?

A good start to this objective, is to isolate the list of libraries from the rest of the message. In other words, there is a bunch of characters in the start of the log message, that we do not care about. So let's get rid of them.

If you look closely to the message, you can see that the character `:` appears twice within the message. One close to the start of the string, and another time right before the start of the list of the libraries. We can use this character as our first delimiter, to collect the third substring that it creates within the total string, which is the substring that contains the list of libraries.

This first stage is presented visually at Figure 8.2.

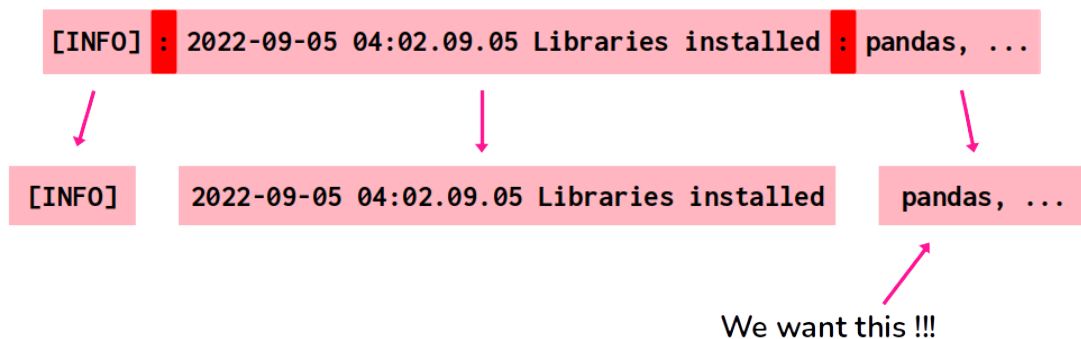


Figure 8.2: The first stage of subsetting

Now that we identified the substrings produced by the “delimiter character”, we just need to understand better which index we need to use in `substring_index()` to get this third substring that we want. The Figure 8.3 presents in a visual manner how the count system of `substring_index()` works.

When you use a positive index, `substring_index()` will count the occurrences of the delimiter character from left to right. But, when you use a negative index, the opposite happens. That is, `substring_index()` counts the occurrences of the delimiter character from right to left.

The index 1 represents the first substring that is before the 1st occurrence of the delimiter (`[INFO]`). The index 2 represents everything that is before the 2nd occurrence of the delimiter (`[INFO]: 2022-09-05 04:02.09.05 Libraries installed`). etc.

In contrast, the index -1 represents everything that is after the 1st occurrence of the delimiter, counting from right to left (`pandas, flask, numpy, spark_map, pyspark`). The index -2 represents everything that is after the 2nd occurrence of the delimiter (`2022-09-05 04:02.09.05 Libraries installed: pandas, flask, numpy, spark_map, pyspark`). Again, counting from right to left.

Having all these informations in mind, we can conclude that the following code fit our first objective. Note that I applied the `trim()` function over the result of `substring_index()`, to ensure that the result substring does not contain any unnecessary spaces at both ends.

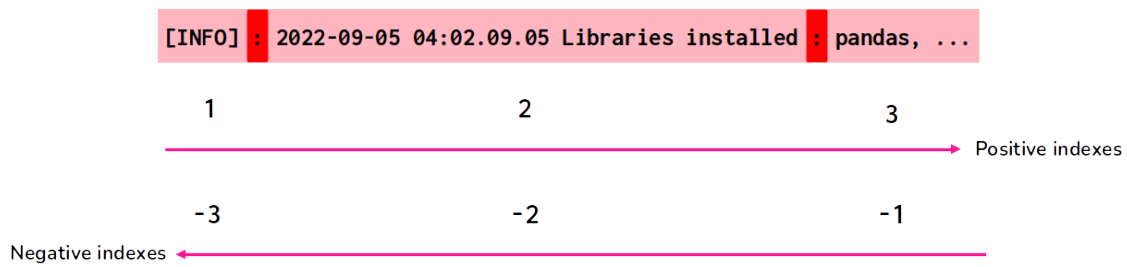


Figure 8.3: The count system of **substring\_index()**

```
from pyspark.sql.functions import substring_index
mes_10th = mes_10th\
    .withColumn(
        'list_of_libraries',
        trim(substring_index('message', ':', -1))
    )

mes_10th.select('list_of_libraries')\
    .show(truncate = n_truncate)
```

```
+-----+
|               list_of_libraries|
+-----+
|pandas, flask, numpy, spark_map, pyspark|
+-----+
```

### 8.5.3 Forming an array of substrings

Now is a good time to introduce the **split()** function, because we can use it to extract the first and the last library from the list libraries of stored at the **mes\_10th** DataFrame. Basically, this function also uses a delimiter character to cut the total string into multiple pieces, and store these pieces in a array of substrings. With this strategy, we can now access each substring (or each piece of the total string) individually.

If we look again at the string that we stored at the **list\_of\_libraries** column, we have a list of libraries, separated by a comma.

```
mes_10th\
  .select('list_of_libraries')\
  .show(truncate = n_truncate)
```

```
+-----+
|               list_of_libraries|
+-----+
|pandas, flask, numpy, spark_map, pyspark|
+-----+
```

The comma character (,) plays an important role in this string, by separating each value in the list. And we can use this comma character as the delimiter inside **split()**, to get an array of substrings. Each element of this array is one of the many libraries in the list. The Figure 8.4 presents this process visually.

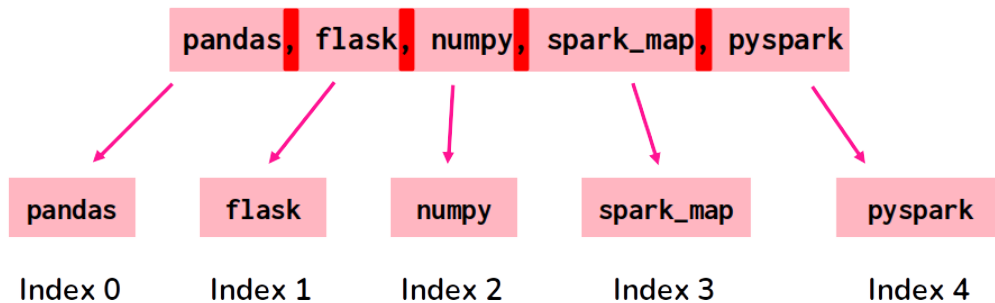


Figure 8.4: Building an array of substrings with **split()**

The code to make this process is very straightforward. In the example below, the column **array\_of\_libraries** becomes a column of data type **ArrayType(StringType)**, that is, an array of string values.

```
from pyspark.sql.functions import split
mes_10th = mes_10th\
  .withColumn(
    'array_of_libraries',
    split('list_of_libraries', ',')
  )

mes_10th\
```

```
.select('array_of_libraries')\
.show(truncate = n_truncate)
```

```
+-----+
|                array_of_libraries|
+-----+
|[pandas, flask, numpy, spark_map, pyspark]|
+-----+
```

By having this array of substring, we can very easily select a specific element in this array, by using the `getItem()` column method, or, by using the open brackets as you would normally use to select an element in a python list.

You just need to give the index of the element you want to select, like in the example below that we select the first and the fifth libraries in the array.

```
mes_10th\
.withColumn('lib_1', col('array_of_libraries')[0])\
.withColumn('lib_5', col('array_of_libraries').getItem(4))\
.select('lib_1', 'lib_5')\
.show(truncate = n_truncate)
```

```
+-----+-----+
| lib_1|  lib_5|
+-----+-----+
|pandas|pyspark|
+-----+-----+
```

## 8.6 Concatenating multiple strings together

Sometimes, we need to concatenate multiple strings together, to form a single and longer string. To do this process, Spark offers two main functions, which are: `concat()` and `concat_ws()`. Both of these functions receives a list of columns as input, and will perform the same task, which is to concatenate the values of each column in the list, sequentially.

However, the `concat_ws()` function have an extra argument called `sep`, where you can define a string to be used as the separator (or the “delimiter”) between the values of each column in the list. In some way, this `sep` argument and the `concat_ws()` function works very similarly to the `join()` string method of python<sup>4</sup>.

<sup>4</sup><https://docs.python.org/3/library/stdtypes.html#str.join>



Let's come back to the **penguins** DataFrame to demonstrate the use of these functions:

```
path = "../Data/penguins.csv"
penguins = spark.read\
    .csv(path, header = True)

penguins.select('species', 'island', 'sex')\
    .show(5)
```

```
+-----+-----+-----+
|species|  island|   sex|
+-----+-----+-----+
| Adelie|Torgersen|  male|
| Adelie|Torgersen|female|
| Adelie|Torgersen|female|
| Adelie|Torgersen|  null|
| Adelie|Torgersen|female|
+-----+-----+-----+
only showing top 5 rows
```

Suppose you wanted to concatenate the values of the columns **species**, **island** and **sex** together, and, store these new values on a separate column. All you need to do is to list these columns inside the **concat()** or **concat\_ws()** function.

If you look at the example below, you can see that I also used the **lit()** function to add a underline character (**\_**) between the values of each column. This is more verbose, because if you needed to concatenate 10 columns together, and still add a “delimiter character” (like the underline) between the values of each column, you would have to write **lit('\_')** for 9 times on the list.

In contrast, the **concat\_ws()** offers a much more succinct way of expressing this same operation. Because the first argument of **concat\_ws()** is the character to be used as the delimiter between each column, and, after that, we have the list of columns to be concatenated.

```
from pyspark.sql.functions import (
    concat,
    concat_ws,
    lit
)

penguins\
    .withColumn(
        'using_concat',
        concat(
```

```

        'species', lit('_'), 'island',
        lit('_'), 'sex')
    )\
    .withColumn(
        'using_concat_ws',
        concat_ws(
            '_', # The delimiter character
            'species', 'island', 'sex' # The list of columns
        )
    )\
    .select('using_concat', 'using_concat_ws')\
    .show(5, truncate = n_truncate)

```

```

+-----+-----+
|      using_concat|      using_concat_ws|
+-----+-----+
| Adeline_Torgersen_male| Adeline_Torgersen_male|
|Adeline_Torgersen_female|Adeline_Torgersen_female|
|Adeline_Torgersen_female|Adeline_Torgersen_female|
|              null|      Adeline_Torgersen|
|Adeline_Torgersen_female|Adeline_Torgersen_female|
+-----+-----+

```

only showing top 5 rows

If you look closely to the result above, you can also see, that **concat()** and **concat\_ws()** functions deal with null values in different ways. If **concat()** finds a null value for a particular row, in any of the listed columns to be concatenated, the end result of the process is a null value for that particular row.

On the other hand, **concat\_ws()** will try to concatenate as many values as he can. If he does find a null value, he just ignores this null value and go on to the next column, until it hits the last column in the list.

## 8.7 Introducing regular expressions

Spark also provides some basic regex (*regular expressions*) functionality. Most of this functionality is available through two functions that comes from the **pyspark.sql.functions** module, which are:

- **regexp\_replace()**: replaces all occurrences of a specified regular expression pattern in a given string with a replacement string.;
- **regexp\_extract()**: extracts substrings from a given string that match a specified regular expression pattern;

There is also a column method that provides an useful way of testing if the values of a column matches a regular expression or not, which is the **rlike()** column method. You can use the **rlike()** method in conjunction with the **filter()** or **where()** DataFrame methods, to find all values that fit (or match) a particular regular expression, like we demonstrated at Section 5.6.7.2.

### 8.7.1 The Java regular expression standard

At this point, is worth remembering a basic fact about Apache Spark that we introduced at Chapter 2. Apache Spark is written in Scala, which is a modern programming language deeply connected with the Java programming language. One of the many consequences from this fact, is that all regular expression functionality available in Apache Spark is based on the Java **java.util.regex** package.

This means that you should always write regular expressions on your **pyspark** code that follows the Java regular expression syntax, and not the Python regular expression syntax, which is based on the python module **re**.

Although this detail is important, these two flavors of regular expressions (Python syntax versus Java syntax) are very, very similar. So, for the most part, you should not see any difference between these two syntaxes.

If for some reason, you need to consult the full list of all metacharacters available in the Java regular expression standard, you can always check the Java documentation for the **java.util.regex** package. More specifically, the [documentation for the java.util.regex.Pattern class](https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html)<sup>5</sup>.

The following list gives you a quick description of a small fraction of the available metacharacters in the Java syntax, and, as a result, metacharacters that you can use in **pyspark**:

- **.** : Matches any single character;
- **\*** : Matches zero or more occurrences of the preceding character or pattern;
- **+** : Matches one or more occurrences of the preceding character or pattern;
- **?** : Matches zero or one occurrence of the preceding character or pattern;
- **|** : Matches either the expression before or after the **|**;
- **[]** : Matches any single character within the brackets;
- **\d** : Matches any digit character;
- **\b** : Matches a word boundary character;
- **\w** : Matches any word character. Equivalent to the **"\b([a-zA-Z\_0-9]+)\b"** regular expression;
- **\s** : Matches any whitespace character;
- **()** : Groups a series of pattern elements to a single element;

---

<sup>5</sup><https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>

### 8.7.2 Using an invalid regular expression

When you write an invalid regular expression in your code, Spark usually complains with a **java.util.regex.PatternSyntaxException** runtime error. The code presented below is an example of code that produces such error.

In this example, the regular expression `\b([a-z]` is invalid because it is missing a closing parenthesis. If you try to execute this code, Spark will raise a with the message “Unclosed group near index 7”. This error message indicates that there is a syntax error in the regular expression, due to an unclosed group (i.e., a missing closing parenthesis).

```
from pyspark.sql.functions import col
weird_regex = '\b([a-z]
logs\
    .filter(col('message').rlike(weird_regex))\
    .show(5)
```

```
Py4JJavaError: An error occurred while calling o261.showString.
: java.util.regex.PatternSyntaxException: Unclosed group near index 7
([a-z]
```

To avoid these runtime errors, due to invalid regular expressions, is always a good idea to test your regular expressions, before you use them in your **pyspark** code. You can easily test your regular expressions by using online tools, such as the [Regex101 website](https://regex101.com/)<sup>6</sup>.

### 8.7.3 Replacing occurrences of a particular regular expression with `regexp_replace()`

One of the most essential actions with regular expression is to find text that fits into a particular regular expression, and, rewriting this text into a different format, or, even removing it completely from the string.

The **regexp\_replace()** function (from the **pyspark.sql.functions** module) is the function that allows you to perform this kind of operation on string values of a column in a Spark DataFrame.

This function replaces all occurrences of a specified regular expression pattern in a given string with a replacement string, and it takes three different arguments:

- The input column name or expression that contains the string values to be modified;
- The regular expression pattern to search for within the input string values;

---

<sup>6</sup><https://regex101.com/>

- The replacement string that will replace all occurrences of the matched pattern in the input string values;

As an example, let's suppose we want to remove completely the type of the message in all log messages present in the **logs** DataFrame. To that, we first need to get a regular expression capable of identifying all possibilities for these types.

A potential candidate would be the regular expression '\\[(INFO|ERROR|WARN)\\]: ', so let's give it a shot. Since we are trying to **remove** this particular part from all log messages, we should replace this part of the string by an empty string (''), like in the example below:

```
from pyspark.sql.functions import regexp_replace

type_regex = '\\[(INFO|ERROR|WARN)\\]: '

logs\
  .withColumn(
    'without_type',
    regexp_replace('message', type_regex, '')
  )\
  .select('message', 'without_type')\
  .show(truncate = 30)
```

message	without_type
[INFO]: 2022-09-05 03:35:01...   2022-09-05 03:35:01.43 Look...	2022-09-05 03:35:01.43 Look...
[WARN]: 2022-09-05 03:35:58...   2022-09-05 03:35:58.007 Wor...	2022-09-05 03:35:58.007 Wor...
[INFO]: 2022-09-05 03:40:59...   2022-09-05 03:40:59.054 Loo...	2022-09-05 03:40:59.054 Loo...
[INFO]: 2022-09-05 03:42:24...   2022-09-05 03:42:24 3 Worke...	2022-09-05 03:42:24 3 Worke...
[INFO]: 2022-09-05 03:42:37...   2022-09-05 03:42:37 Initial...	2022-09-05 03:42:37 Initial...
[WARN]: 2022-09-05 03:52:02...   2022-09-05 03:52:02.98 Libr...	2022-09-05 03:52:02.98 Libr...
[INFO]: 2022-09-05 04:00:33...   2022-09-05 04:00:33.210 Lib...	2022-09-05 04:00:33.210 Lib...
[INFO]: 2022-09-05 04:01:15...   2022-09-05 04:01:15 All clu...	2022-09-05 04:01:15 All clu...
[INFO]: 2022-09-05 04:01:35...   2022-09-05 04:01:35.022 Mak...	2022-09-05 04:01:35.022 Mak...
[INFO]: 2022-09-05 04:02:09...   2022-09-05 04:02:09.05 Libr...	2022-09-05 04:02:09.05 Libr...
[INFO]: 2022-09-05 04:02:09...   2022-09-05 04:02:09.05 The ...	2022-09-05 04:02:09.05 The ...
[INFO]: 2022-09-05 04:02:09...   2022-09-05 04:02:09.05 An e...	2022-09-05 04:02:09.05 An e...
[ERROR]: 2022-09-05 04:02:1...   2022-09-05 04:02:12 A task ...	2022-09-05 04:02:12 A task ...
[ERROR]: 2022-09-05 04:02:3...   2022-09-05 04:02:34.111 Err...	2022-09-05 04:02:34.111 Err...
[ERROR]: 2022-09-05 04:02:3...   2022-09-05 04:02:34.678 Tra...	2022-09-05 04:02:34.678 Tra...
[ERROR]: 2022-09-05 04:02:3...   2022-09-05 04:02:35.14 Quit...	2022-09-05 04:02:35.14 Quit...

Is useful to remind that this **regexp\_replace()** function searches for **all occurrences** of the regular expression on the input string values, and replaces all of these occurrences by the input replacement string that you gave. However, if the function does not find any matches for your regular expression inside a particular value in the column, then, the function simply returns this value intact.

## 8.7.4 Introducing capturing groups on pyspark

One of the many awesome functionalities of regular expressions, is the capability of enclosing parts of a regular expression inside groups, and actually store (or cache) the substring matched by this group. This process of grouping parts of a regular expression inside a group, and capturing substrings with them, is usually called of “grouping and capturing”.

Is worth pointing out that this capturing groups functionality is available both in **regexp\_replace()** and **regexp\_extract()**.

### 8.7.4.1 What is a capturing group ?

Ok, but, what is this group thing? You create a group inside a regular expression by enclosing a particular section of your regular expression inside a pair of parentheses. The regular expression that is written inside this pair of parentheses represents a capturing group.

A capturing group inside a regular expression is used to capture a specific part of the matched string. This means that the actual part of the input string that is matched by the regular expression that is inside this pair of parentheses, is captured (or cached, or saved) by the group, and, can be reused later.

Besides grouping part of a regular expression together, parentheses also create a numbered capturing group. It stores the part of the string matched by the part of the regular expression inside the parentheses. .... The regex “Set(Value)?” matches “Set” or “SetValue”. In the first case, the first (and only) capturing group remains empty. In the second case, the first capturing group matches “Value”. (Goyvaerts 2023).

So, remember, to use capturing groups in a regular expression, you must enclose the part of the pattern that you want to capture in parentheses **()**. Each set of parentheses creates a new capturing group. This means that you can create multiple groups inside a single regular expression, and, then, reuse latter the substrings captured by all of these multiple groups. Awesome, right?

Each new group (that is, each pair of parentheses) that you create in your regular expression have a different index. That means that the first group is identified by the index 1, the second group, by the index 2, the third group, by the index 3, etc.

Just to quickly demonstrate these capturing groups, here is a quick example, in pure Python:

```
import re

# A regular expression that contains
# three different capturing groups
regex = r"(\d{3})-(\d{2})-(\d{4})"

# Match the regular expression against a string
text = "My social security number is 123-45-6789."
match = re.search(regex, text)

# Access the captured groups
group1 = match.group(1) # "123"
group2 = match.group(2) # "45"
group3 = match.group(3) # "6789"
```

In the above example, the regular expression `r"(\d{3})-(\d{2})-(\d{4})"` contains three capturing groups, each enclosed in parentheses. When the regular expression is matched against the string `"My social security number is 123-45-6789."`, the first capturing group matches the substring `"123"`, the second capturing group matches `"45"`, and the third capturing group matches `"6789"`.

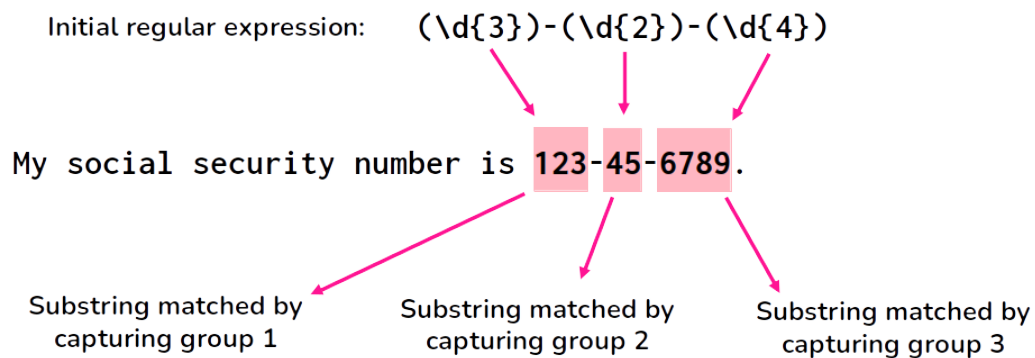


Figure 8.5: Example of capturing groups

In Python, we can access the captured groups using the `group()` method of the `Match` object returned by `re.search()`. In this example, `match.group(1)` returns the captured substring of the first capturing group (which is `"123"`), `match.group(2)` returns second `"45"`, and `match.group(3)` returns `"6789"`.

#### 8.7.4.2 How can we use capturing groups in pyspark ?

Ok, now that we understood what capturing groups is, how can we use them in **pyspark**? First, remember, capturing groups will be available to you, only if you enclose a part of your regular expression

in a pair of parentheses. So the first part is to make sure that the capturing groups are present in your regular expressions.

After that, you can access the substring matched by the capturing group, by using the reference index that identifies this capturing group you want to use. In pure Python, we used the **group()** method with the group index (like 1, 2, etc.) to access these values.

But in **pyspark**, we access these groups by using a special pattern formed by the group index preceded by a dollar sign (\$). That is, the text **\$1** references the first capturing group, **\$2** references the second capturing group, etc.

As a first example, let's go back to the regular expression we used at Section 8.7.3: **\\[(INFO|ERROR|WARN)\\]:.** This regular expression contains one capturing group, which captures the type label of the log message: **(INFO|ERROR|WARN)**.

If we use the special pattern **\$1** to reference this capturing group inside of **regexp\_replace()**, what is going to happen is: **regexp\_replace()** will replace all occurrences of the input regular expression found on the input string, by the substring matched by the first capturing group. See in the example below:

```
logs\
  .withColumn(
    'using_groups',
    regexp_replace('message', type_regex, 'Type Label -> $1 | ')
  )\
  .select('message', 'using_groups')\
  .show(truncate = 30)
```

message	using_groups
[INFO]: 2022-09-05 03:35:01...	Type Label -> INFO   2022-0...
[WARN]: 2022-09-05 03:35:58...	Type Label -> WARN   2022-0...
[INFO]: 2022-09-05 03:40:59...	Type Label -> INFO   2022-0...
[INFO]: 2022-09-05 03:42:24...	Type Label -> INFO   2022-0...
[INFO]: 2022-09-05 03:42:37...	Type Label -> INFO   2022-0...
[WARN]: 2022-09-05 03:52:02...	Type Label -> WARN   2022-0...
[INFO]: 2022-09-05 04:00:33...	Type Label -> INFO   2022-0...
[INFO]: 2022-09-05 04:01:15...	Type Label -> INFO   2022-0...
[INFO]: 2022-09-05 04:01:35...	Type Label -> INFO   2022-0...
[INFO]: 2022-09-05 04:02:09...	Type Label -> INFO   2022-0...
[INFO]: 2022-09-05 04:02:09...	Type Label -> INFO   2022-0...
[INFO]: 2022-09-05 04:02:09...	Type Label -> INFO   2022-0...
[ERROR]: 2022-09-05 04:02:1...	Type Label -> ERROR   2022-...



```

|[ERROR]: 2022-09-05 04:02:3...|Type Label -> ERROR | 2022-...|
|[ERROR]: 2022-09-05 04:02:3...|Type Label -> ERROR | 2022-...|
|[ERROR]: 2022-09-05 04:02:3...|Type Label -> ERROR | 2022-...|
+-----+-----+

```

In essence, you can reuse the substrings matched by the capturing groups, by using the special patterns **\$1**, **\$2**, **\$3**, etc. This means that you can reuse the substrings captured by multiple groups at the same time inside **regexp\_replace()** and **regexp\_extract()**. For example, if we use the replacement string **"\$1, \$2, \$3"** inside **regexp\_replace()**, we would get the substrings matched by the first, second and third capturing groups, separated by commas.

However, it is also good to emphasize a small limitation that this system has. When you need to reuse the substrings captured by multiple groups together, it is important that you make sure to add some amount of space (or some delimiter character) between each group reference, like **"\$1 \$2 \$3"**.

Because if you write these group references one close to each other (like in **"\$1\$2\$3"**), it is not going to work. In other words, Spark will not understand that you are trying to access a capturing group. It will interpret the text **"\$1\$2\$3"** as the literal value **"\$1\$2\$3"**, and not as a special pattern that references multiple capturing groups in the regular expression.

### 8.7.5 Extracting substrings with **regexp\_extract()**

Another very useful regular expression activity is to extract a substring from a given string that match a specified regular expression pattern. The **regexp\_extract()** function is the main method used to do this process.

This function takes three arguments, which are:

- The input column name or expression that contains the string to be searched;
- The regular expression pattern to search for within the input string;
- The index of the capturing group within the regular expression pattern that corresponds to the substring to extract;

You may (or may not) use capturing groups inside of **regexp\_replace()**. However, on the other hand, the **regexp\_extract()** function is based on the capturing groups functionality. As a consequence, when you use **regexp\_extract()**, you must give a regular expression that **contains some capturing group**. Because otherwise, the **regexp\_extract()** function becomes useless.

In other words, the **regexp\_extract()** function extracts substrings that are matched by the capturing groups present in your input regular expression. If you want, for example, to use **regexp\_extract()** to extract the substring matched by a entire regular expression, then, you just need to surround this entire regular expression by a pair of parentheses. This way you transform this entire regular expression in a capturing group, and, therefore, you can extract the substring matched by this group.

As an example, let's go back again to the regular expression we used in the **logs** DataFrame: `\\[(INFO|ERROR|WARN)\\]:`. We can extract the type of log message label, by using the index 1 to reference the first (and only) capturing group in this regular expression.

```
from pyspark.sql.functions import regexp_extract

logs\
  .withColumn(
    'message_type',
    regexp_extract('message', type_regex, 1)
  )\
  .select('message', 'message_type')\
  .show(truncate = 30)
```

message	message_type
[[INFO]: 2022-09-05 03:35:01...	INFO
[[WARN]: 2022-09-05 03:35:58...	WARN
[[INFO]: 2022-09-05 03:40:59...	INFO
[[INFO]: 2022-09-05 03:42:24...	INFO
[[INFO]: 2022-09-05 03:42:37...	INFO
[[WARN]: 2022-09-05 03:52:02...	WARN
[[INFO]: 2022-09-05 04:00:33...	INFO
[[INFO]: 2022-09-05 04:01:15...	INFO
[[INFO]: 2022-09-05 04:01:35...	INFO
[[INFO]: 2022-09-05 04:02:09...	INFO
[[INFO]: 2022-09-05 04:02:09...	INFO
[[INFO]: 2022-09-05 04:02:09...	INFO
[[ERROR]: 2022-09-05 04:02:1...	ERROR
[[ERROR]: 2022-09-05 04:02:3...	ERROR
[[ERROR]: 2022-09-05 04:02:3...	ERROR
[[ERROR]: 2022-09-05 04:02:3...	ERROR

As another example, let's suppose we wanted to extract not only the type of the log message, but also, the timestamp and the content of the message, and store these different elements in separate columns.

To do that, we could build a more complete regular expression. An expression capable of matching the entire log message, and, at the same time, capture each of these different elements inside a different capturing group. The code below is an example that produces such regular expression, and applies it over the **logs** DataFrame.

```

type_regex = r'\[(INFO|ERROR|WARN)\]: '

date_regex = r'\d{4}-\d{2}-\d{2}'
time_regex = r' \d{2}:\d{2}:\d{2}([\d+])?'
timestamp_regex = date_regex + time_regex
timestamp_regex = r'(' + timestamp_regex + r')'

regex = type_regex + timestamp_regex + r'(.+)$'

logs\
  .withColumn(
    'message_type',
    regexp_extract('message', regex, 1)
  )\
  .withColumn(
    'timestamp',
    regexp_extract('message', regex, 2)
  )\
  .withColumn(
    'message_content',
    regexp_extract('message', regex, 4)
  )\
  .select('message_type', 'timestamp', 'message_content')\
  .show(5, truncate = 30)

```

```

+-----+-----+-----+
|message_type|      timestamp|message_content|
+-----+-----+-----+
|      INFO|2022-09-05 03:35:01.43| Looking for workers at Sou...|
|      WARN|2022-09-05 03:35:58.007| Workers are unavailable at...|
|      INFO|2022-09-05 03:40:59.054| Looking for workers at Sou...|
|      INFO|      2022-09-05 03:42:24| 3 Workers were acquired at...|
|      INFO|      2022-09-05 03:42:37| Initializing instances in ...|
+-----+-----+-----+

```

only showing top 5 rows

### 8.7.6 Identifying values that match a particular regular expression with `rlike()`

The `rlike()` column method is useful for checking if a string value in a column matches a specific regular expression. We briefly introduced this method at Section 5.6.7.2. This method has only one input, which is the regular expression you want to apply over the column values.

As an example, let's suppose you wanted to identify timestamp values inside your strings. You could use a regular expression pattern to find which text values had these kinds of values inside them.

A possible regular expression candidate for it would be `"[0-9]{2}:[0-9]{2}:[0-9]{2}([.][0-9]+)?"`. This regex matches timestamp values in the format "hh:mm:ss.sss". This pattern consists of the following building blocks, or, elements:

- `[0-9]{2}`: Matches any two digits from 0 to 9.
- `:`: Matches a colon character.
- `([.][0-9]+)?`: Matches an optional decimal point followed by one or more digits.

If we apply this pattern over all log messages stored in the **logs** DataFrame, we would find that all log messages matches this particular regular expression. Because all log messages contains a timestamp value at the start of the message:

```
from pyspark.sql.functions import col

pattern = "[0-9]{2}:[0-9]{2}:[0-9]{2}([.][0-9]+)?"
logs\
  .withColumn(
    'does_it_match?',
    col("message").rlike(pattern)
  )\
  .select('message', 'does_it_match?')\
  .show(5, truncate = n_truncate)
```

```
+-----+-----+
|                                     message|does_it_match?|
+-----+-----+
|[INFO]: 2022-09-05 03:35:01.43 Looking for work...|      true|
|[WARN]: 2022-09-05 03:35:58.007 Workers are una...|      true|
|[INFO]: 2022-09-05 03:40:59.054 Looking for wor...|      true|
|[INFO]: 2022-09-05 03:42:24 3 Workers were acqu...|      true|
|[INFO]: 2022-09-05 03:42:37 Initializing instan...|      true|
+-----+-----+
```

only showing top 5 rows

## References

- Apache Spark Official Documentation*. 2022. Documentation for Apache Spark 3.2.1. <https://spark.apache.org/docs/latest/>.
- Chambers, Bill, and Matei Zaharia. 2018. *Spark: The Definitive Guide: Big Data Processing Made Simple*. Sebastopol, CA: O'Reilly Media.
- Damji, Jules, Brooke Wenig, Tathagata Das, and Denny Lee. 2020. *Learning Spark: Lightning-Fast Data Analytics*. Sebastopol, CA: O'Reilly Media.
- Goyvaerts, Jan. 2023. "Regular-Expressions.info." <https://www.regular-expressions.info/>.
- Karau, Holden, Andy Konwinski, Patrick Wendell, and Matei Zaharia. 2015. *Learning Spark: Lightning-Fast Data Analytics*. Sebastopol, CA: O'Reilly Media.

## A Opening the terminal of your OS

Every operating system (OS) comes with a terminal (or a command prompt). A terminal is usually just a black screen where you can send commands to be executed by your OS. As an example, Figure A.1 shows a print screen of the terminal in Ubuntu Linux. The terminal on Windows is very similar to this.

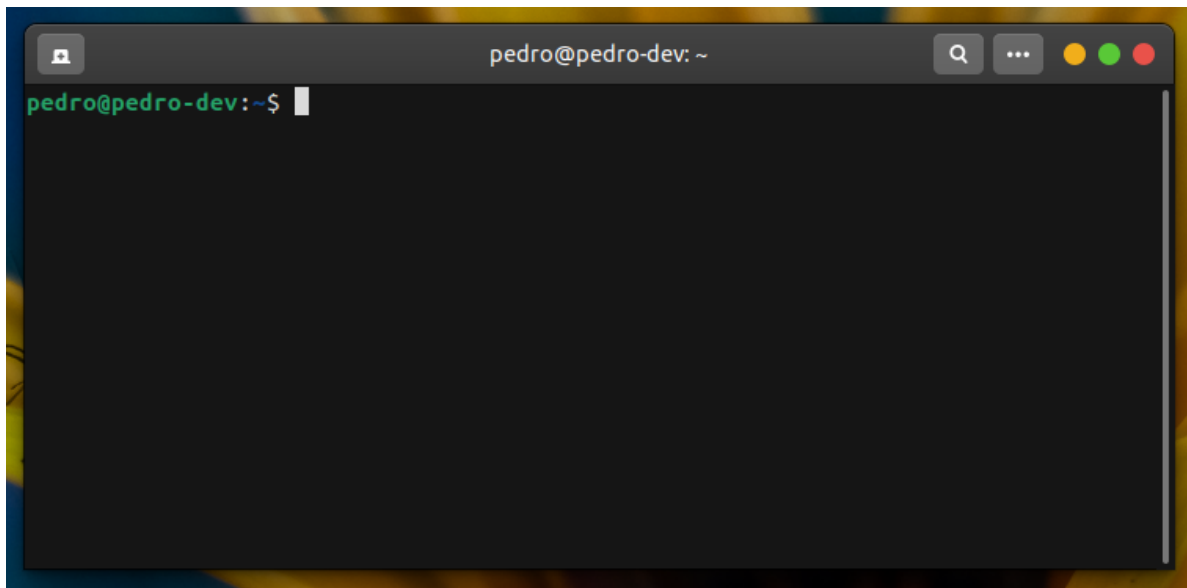


Figure A.1: A terminal on Ubuntu

Terminals are incredibly flexible tools, and you will find this tool very useful for a number of things. However, since they are just a black and empty screen, they give you no clue of what you should do, or what commands are available to you.

That is ok, for now, you should not worry about that. I will expose many terminal commands in this book, and these commands should be used inside these terminals that comes with your OS. The next sub-sections will show you how to open a terminal in each one of the OS's where Spark is available. Lets begin with Windows.

### A.0.1 Opening a terminal on Windows

There are some different approaches to do this, but, the one that I find the most useful, is to open a terminal from inside a folder, using the default File Explorer program of Windows.

This is very useful, because we normally use terminal commands to affect a set of files that are stored inside a specific folder in our computer.

As a result, when we open a terminal from inside a folder, using the File Explorer, the terminal opened is already rooted inside the folder where our files are stored. In other words, we already have easy access to the files that we want to affect/use in our command, and we do not have the work to change or adjust directories in this terminal.

For example, lets suppose that we want to use a terminal command to use a file called **hello.py**, and, that this **hello.py** file is stored inside a folder called **HelloPython**. You can see in Figure A.2, that this folder is in path **C:\Users\pedro\Documents\HelloPython**. So, the first step, is to use the File Explorer of Windows, to open this folder, like in Figure A.2.

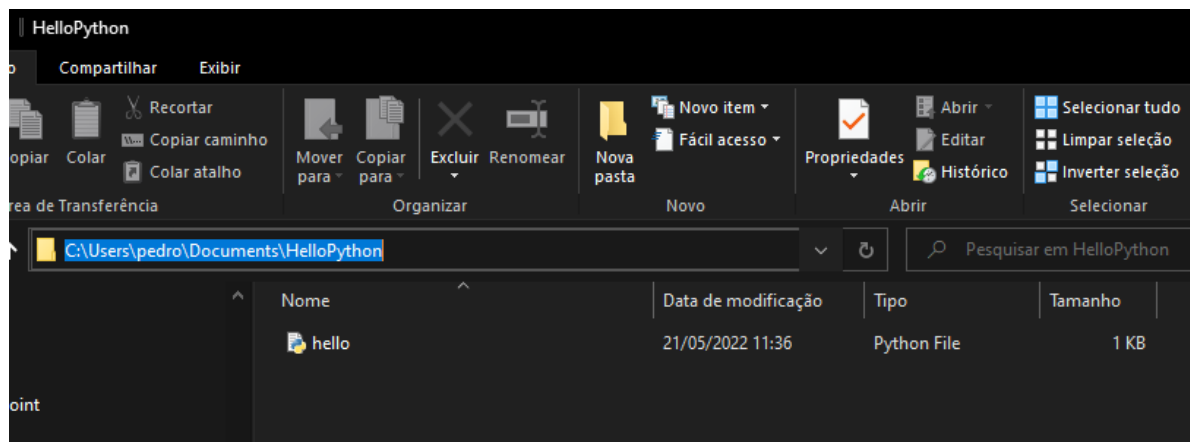


Figure A.2: Opening the **HelloPython** folder in Windows

After you opened this folder, substitute the path to the folder in the search box with the word “cmd”, like in Figure A.3, and them, press Enter in the keyboard.

As a result, a new terminal will open. See in Figure A.4, that this new terminal is already looking to (or is already rooted on) this **HelloPython** folder. This way, we can easily access the files stored inside this folder, like the **hello.py** file.

### A.0.2 Opening a terminal on Linux

Is fairly easy to open a terminal on a Linux distribution. Again, is very useful when you open the terminal from inside the folder you are interested in. Because you will have an easier access to all the files that are stored inside this folder.

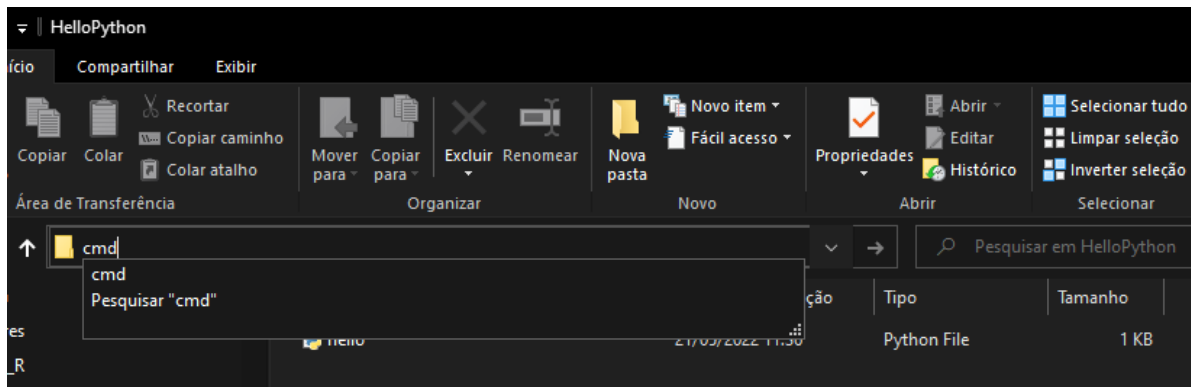


Figure A.3: Opening a terminal inside a Windows folder - Part 1

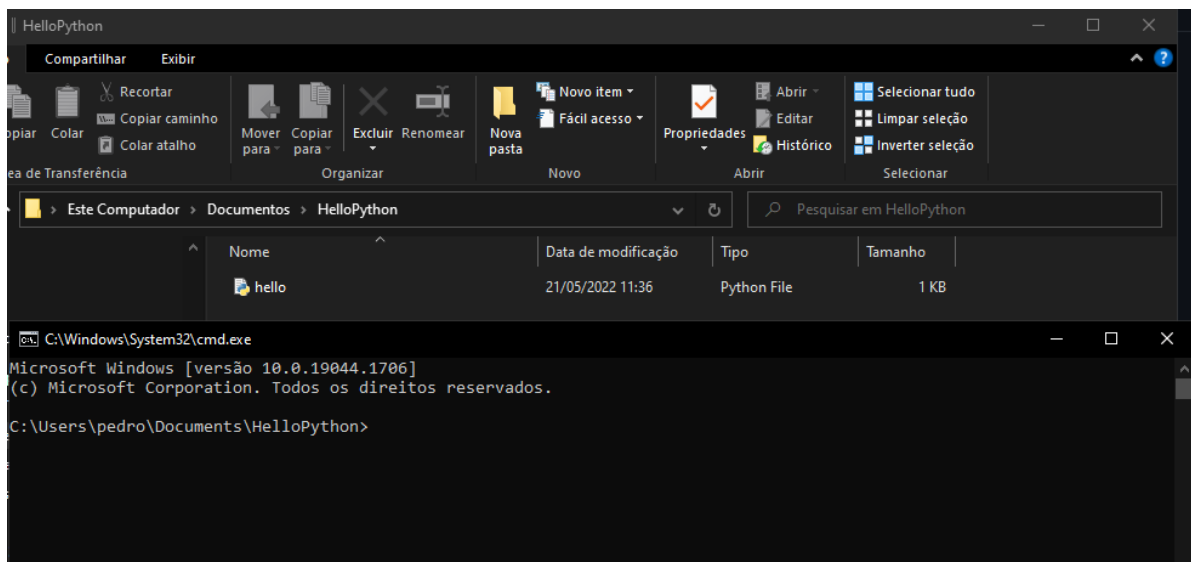


Figure A.4: Opening a terminal inside a Windows folder - Part 2



To do this in Linux, you use the built-in File Explorer to open the folder where you want to root your terminal. At the moment, I am using an Ubuntu distribution. I just opened the same **HelloPython** folder, with the same **hello.py** file, in the File Explorer of Linux. As shown in Figure A.5:

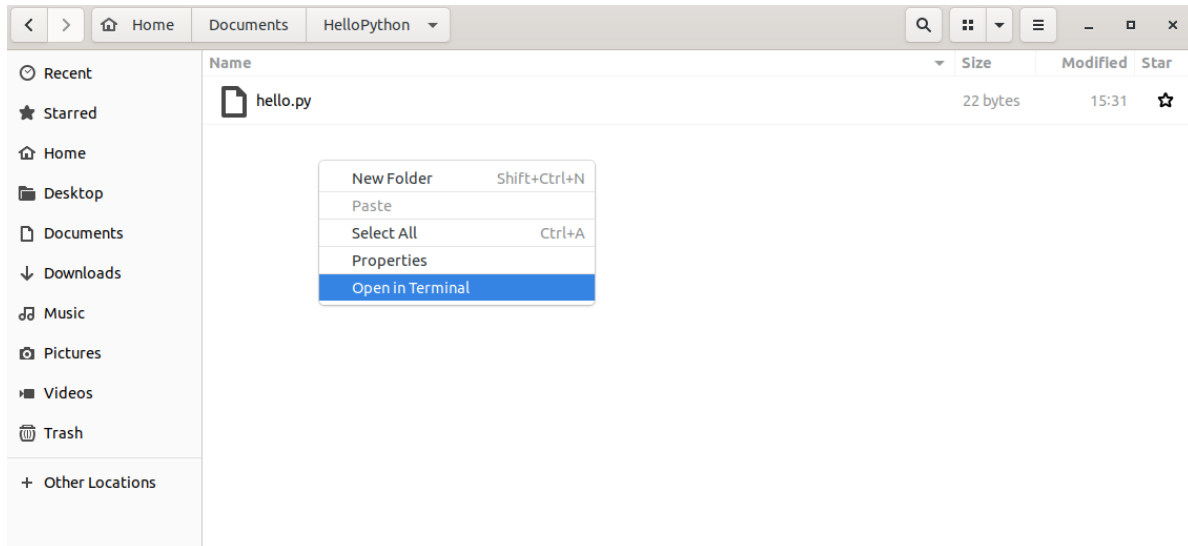


Figure A.5: Opening the **HelloPython** folder in Linux

After you opened the folder, just click with the right button of your mouse, and select the “Open in Terminal” option, and a new terminal should appear on your screen. See in Figure A.6, that the new terminal is already looking to the **HelloPython** folder, as we expected.

### A.0.3 Opening a terminal in MacOS

Unfortunately, I do not have a Mac machine in my possession, so I cannot easily show you how to open a terminal in MacOS. But there a lot of articles available in the internet discussing how to open a terminal in MacOS. For example, there is a article from the support of Apple<sup>1</sup>, or this other article from iDownloadBlog<sup>2</sup>.

---

<sup>1</sup><https://support.apple.com/en-ie/guide/terminal/apd5265185d-f365-44cb-8b09-71a064a42125/mac>

<sup>2</sup><https://www.idownloadblog.com/2019/04/19/ways-open-terminal-mac/>

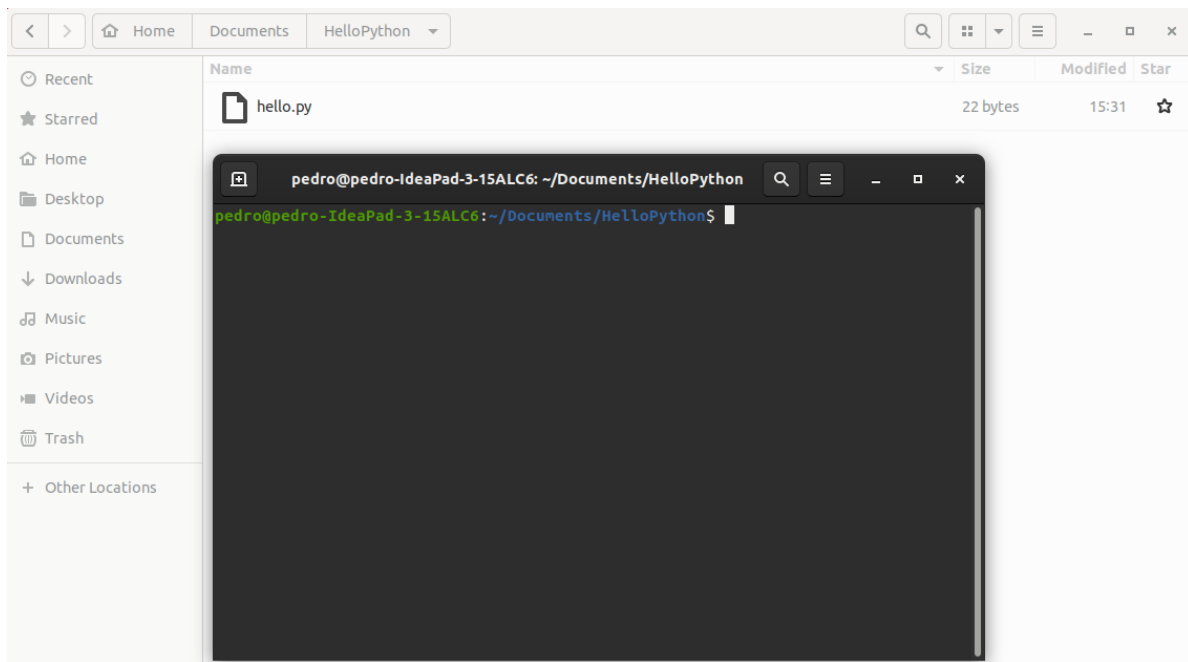


Figure A.6: Opening the terminal in Linux

## B How to install Spark and pyspark

In order to build and run any Spark application through **pyspark**, you have to install Apache Spark in your computer. Apache Spark is available to all three major operating systems in the market today (macOS, Windows and Linux).

The process of installation is kind of similar in all three OS's. But some steps work differently in each OS. Currently, I (the author of this book) do not have access to a macOS machine, and because of that, I will not describe the installation process of Spark on this platform here. If you have access to a macOS machine, and are willing to describe this installation process, I would be happy to review a PR with such content.

### B.1 What are the steps?

In short, the next steps for installing Spark are:

1. Install Java;
2. Install Python;
3. Install **pyspark**;
4. Download and extract Apache Spark;
5. Set a few environment variables;

The next sections describes these steps for each operating system.

### B.2 On Windows

#### B.2.1 Install Java SDK

Apache Spark is written in Scala, which is a fairly modern programming language that have powerful interoperability with the Java programming language. Because of this characteristic, some of the functionalities of Spark require you to have Java SDK (*Software Development Kit*) installed in your machine.

In other words, you must have Java SDK installed to use Spark. If you do not have Java SDK, Spark will likely fail when you try to start it. However, since Java is a very popular technology across the world,

is possible that you already have it installed in your machine. To check if Java is already installed, you can run the following command in your OS terminal:

```
#| eval: false
Terminal$ java -version
```

If the above command outputs something similar to the text exposed below, than, you already have Java installed in your machine, and you can proceed to the next step.

```
java version "19.0.1" 2022-10-18
Java(TM) SE Runtime Environment (build 19.0.1+10-21)
Java HotSpot(TM) 64-Bit Server VM (build 19.0.1+10-21, mixed mode, sharing)
```

But, if something different comes up in your terminal, than, is likely that you do not have Java installed. To fix this, [download Java from the official website](#)<sup>1</sup>, and install it.

## B.2.2 Install Python

You can easily install Python on Windows, by downloading the installer available at the [official website of the language](#)<sup>2</sup>, and executing it.

## B.2.3 Install pyspark

Installing the **pyspark** python package is pretty straightforward. Just open a terminal (if you need help to open the terminal check [Appendix A](#)), and use the **pip** command to do it:

```
#| eval: false
Terminal$ pip install pyspark
```

If you try to run the above command (inside a terminal of any OS), and a message like **pip: command not found** appears, this means that you do not have the **pip** tool installed on your machine. Hence, if you face this kind of message, you need to install **pip** before you even install **pyspark**.

The **pip** tool is automatically installed with Python on Windows. So, if you face this message (**pip: command not found**), then, is very likely that you do not have Python correctly installed on your machine. Or maybe, Python is not installed at all in any shape or size in your system. So, you should comeback to previous section, and re install it.

---

<sup>1</sup><https://www.oracle.com/java/technologies/downloads/>

<sup>2</sup><https://www.python.org/downloads/>

## B.2.4 Download and extract the files of Apache Spark

First, you need to download Apache Spark from the [official website of the project](https://spark.apache.org/downloads)<sup>3</sup>. Currently, the Apache Spark does not offers installers or package programs to install the software for you.

We usually install external software on Windows by using installers (i.e. executable files - **.exe**) that perform all the necessary steps to install the software in your machine. However, currently, the Apache Spark project does not offers such installers. This means that you have to install it yourself.

When you download Apache Spark from the official website, you will get all files of the program compacted inside a TAR file (**.tgz**), which is similar to a ZIP file (**.zip**). After you downloaded Spark to your machine, you need to extract all files from the TAR file (**.tgz**) to a specific location of your computer. It can be anywhere, just choose a place. As an example, I will extract the files to a folder called **Spark** directly at my hard disk **C:/**.

To extract these files, you can use very popular UI tools like [7zip](https://7-zip.org/)<sup>4</sup> or [WinRAR](https://www.win-rar.com/)<sup>5</sup>. However, if you use a modern version of Windows, there is a **tar** command line tool available at the terminal that you can use to do this process in a programatic fashion. As an example, I can extract and move all files of Spark with these commands:

```
#| eval: false
Terminal$ tar -x -f spark-3.3.1-bin-hadoop3.tgz
Terminal$ mv spark-3.3.1-bin-hadoop3 C:/Spark/spark-3.3.1-bin-hadoop3
```

## B.2.5 Set environment variables

Apache Spark will always look for two specific environment variables in your system: **SPARK\_HOME** and **JAVA\_HOME**. This means that you must have these two environment variables defined and correctly configured in your machine. To configure an environment variable on Windows, I recommend you to use the menu that you can find by searching for “environment variables” in the Windows search box. At Figure [B.1](#) you can see the path to this menu on Windows:

During the process of scheduling and running your Spark application, Spark will execute a set of scripts located at the home directory of Spark itself. Spark does this by looking for a environment variable called **SPARK\_HOME** in your system. If you do not have this variable configured, Spark will not be able to locate its home directory, and as a consequence, it throws an runtime error.

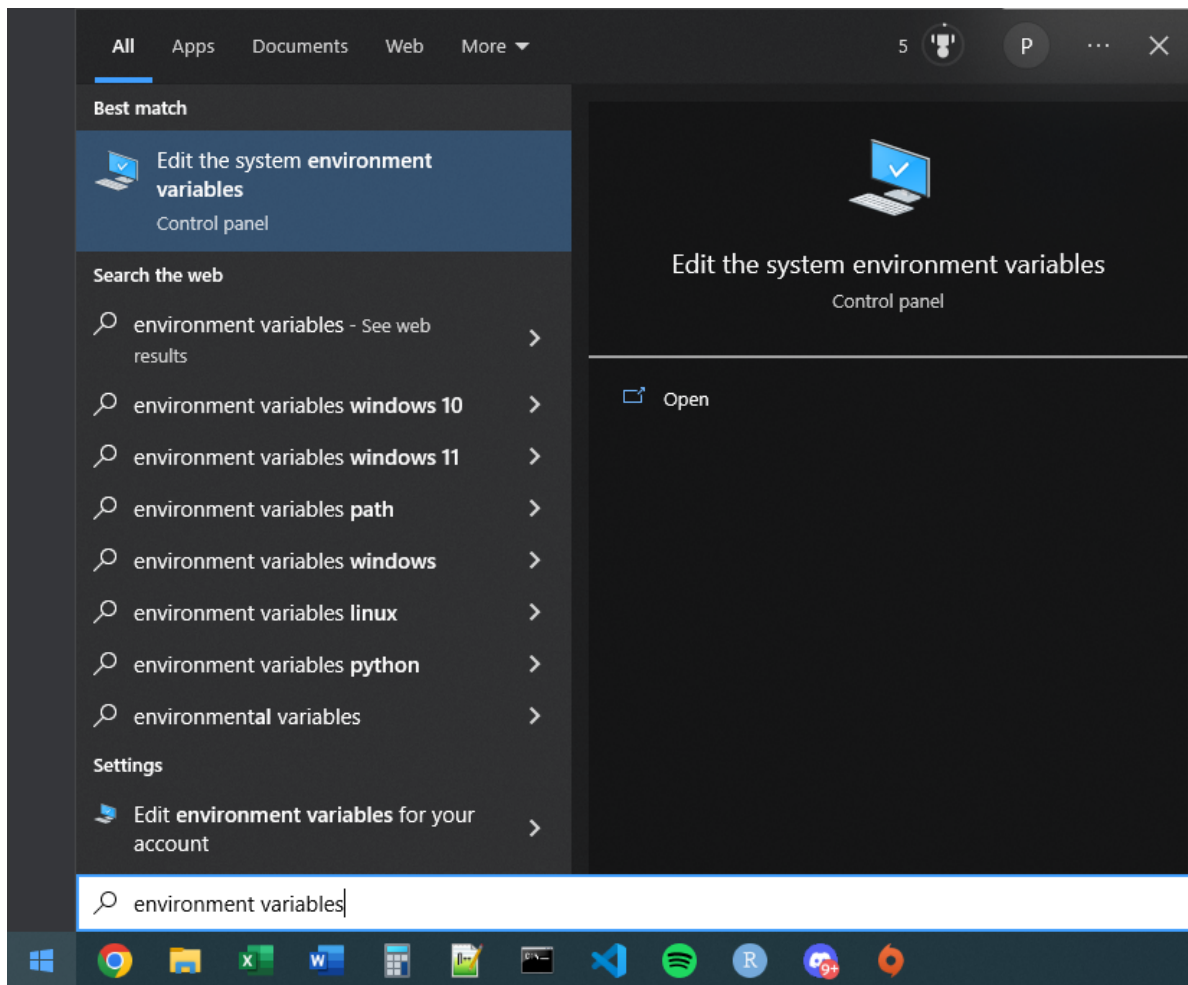
In other words, you need to set a variable with name **SPARK\_HOME** in your system. Its value should be the path to the home (or “root”) directory where you installed (or unzipped) the Spark files. In my case I unzipped the files into a folder called **C:/Spark/spark-3.3.1-bin-hadoop3**, and that is the folder that I going to use in **SPARK\_HOME**, as you can see at Figure [B.2](#):

---

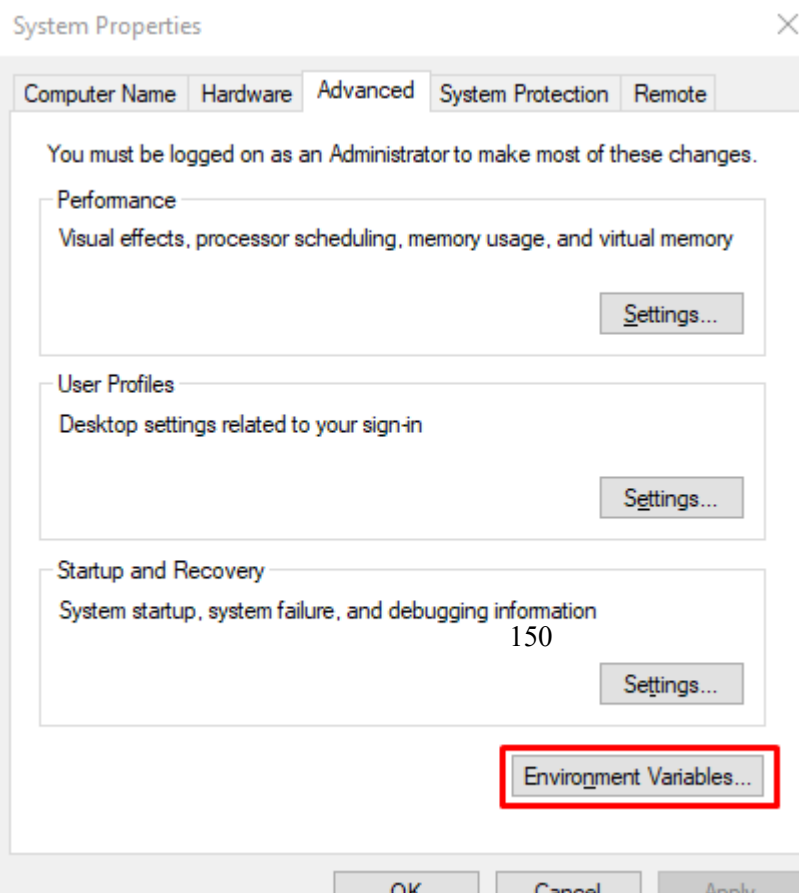
<sup>3</sup><https://spark.apache.org/downloads>

<sup>4</sup><https://7-zip.org/>

<sup>5</sup><https://www.win-rar.com/>



windows-env1, fig-env="subfigure"}



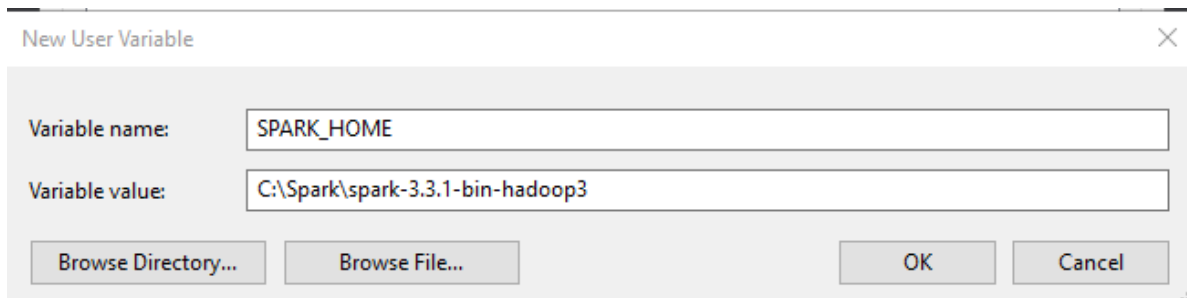


Figure B.2: Setting the **SPARK\_HOME** environment variable

Now, a lot Spark functionality is closely related to Java SDK, and to find it, Spark will look for an environment variable called **JAVA\_HOME**. On my machine, I have Java SDK installed at the folder **C:\Program Files\Java\jdk-19**. This might be not your case, and you may find the “jdk” folder for Java at a different location of your computer. Just find where it is, and use this path while setting this **JAVA\_HOME** variable, like I did at Figure B.3:

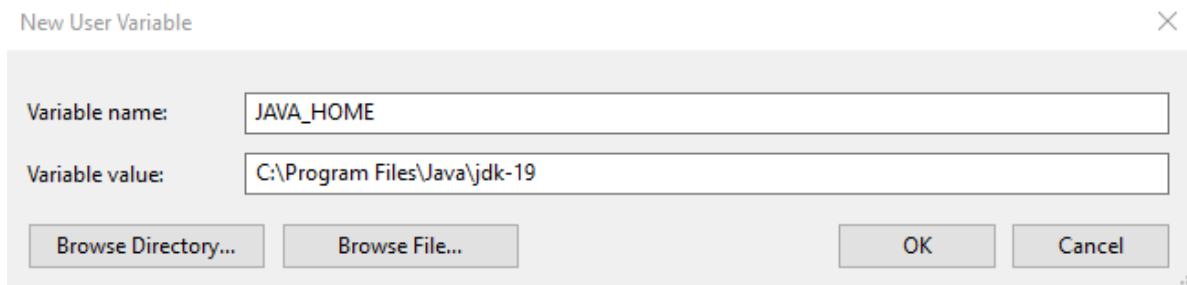


Figure B.3: Setting the **JAVA\_HOME** environment variable

With these two environment variables setted, you should be able to start and use Spark already. Just open a brand new terminal, and type the **spark** command. An interactive Spark Session will initiate after this command, and give you a command prompt where you can start coding the Spark application you want to execute.

## B.3 On Linux

### B.3.1 Install pyspark

In Linux systems, installing **pip** is very easy, because you can use the built-in package manager to do this for you. In Debian like distros (e.g. Ubuntu), you use the **apt** tool, and, in Arch-Linux like distros (e.g. Arch Linux, Manjaro) you would use the **pacman** tool. Both possibilities are exposed below:

**{terminal, eval = FALSE} # If you are in a Debian like distro of Linux # and need to install `pip`, use this command: Terminal\$ sudo apt install python3-pip # If you are in a Arch-Linux like distro of Linux # and need to install `pip`, use this command: Terminal\$ pacman -S python-pip**