Initial exploration with pandas

## 📚 Get the info

Immediately after reading the data, you should run **DataFrame.info().** It is a great way to see, at a glance:

* How many rows (“entries”) the DataFrame has.
* The columns of the DataFrame.
* How many null values the DataFrame has.
* The data type of each column.
  + Should any column be numeric, but pandas has read it as an object? If so, you will eventually have to explore why this happened.

At this moment, we recommend not to lose time trying to fix issues you identify, but to simply take note and keep exploring, cleaning and transforming is better done once you have a big picture of the dataset.

## 📚 Nothing like raw data

When datasets are big enough not to be able to take a glimpse at the raw data to acknowledge what’s going on in there, it might be tempting to just aggregate information into digestible summaries and go “insight fishing” right away. But getting a sense of what the data holds is critical.

Use the **DataFrame.head()** method to look at the raw data. Play with its n argument to visualise more rows. Its counterpart, **DataFrame.tail(),** is also useful since sometimes the last rows of a DataFrame have weird stuff in them.

Sometimes, this exploration allows for detecting problems in the data that would otherwise remain undetected: strings with special characters that were not read properly, missing data that was encoded as an empty string (“-” or “NA” are typical too), information that is clearly wrong…

## 📚 Quick tricks

Pandas provides some other functions and methods to explore the data quickly and get an idea of what is going on. Find some examples below, and of course, feel free to explore more by checking the docs. Of course, you should replace DataFrame for the actual name of your pandas DataFrame (e.g. orders or products) and DataFrame.columnName for the actual name of the column (e.g. products.price):

* **DataFrame.describe()** gives basic numerical aggregations. It can be applied to a single column as well.
* **DataFrame.isna().any()** highlights which columns contain missing data.
* **DataFrame.shape** gives the number of rows and columns.
* **DataFrame.columns** gives the column names. Note that a list with new names can be passed to this attribute to rename the columns.
* **DataFrame.columnName.isna().sum()** is a quick way to check the number of missing values in a column
* **DataFrame.columnName.value\_counts()** is a great way to summarise a categorical column. You can use it to discover how many orders are completed, cancelled, pending…
* **DataFrame.columnName.hist()** is an easy way to plot a histogram in a numerical column. Play with the bins argument to change the granularity of the graph.

Find more quick plotting options with pandas in this user guide:

<https://pandas.pydata.org/docs/user_guide/visualization.html#visualization>

## 🧩 Custom exploration

Now that you’ve learned a bit more about using pandas, look at the data and ask yourself: “what do I need to know about it?”. Some questions should start to pop up in your head, along the lines of:

* How many orders are there?
* How many products are there?
* What period of time do these orders comprise?
* How many orders are Completed?
* How should revenue be computed?
* …

As you go through these questions, it is quite normal to detect issues with the data, like numbers that are not read as numbers. When you feel these issues are blocking you from progressing, move on to the next lesson in the platform.

🧩 Cleaning products

You might have noticed during the initial exploration that the products dataset had some serious issues, especially with the price. Here we will guide you through some steps you can take to ensure that you have a dataset you can work with.

Take this guide with a grain of salt, as there is no right or wrong way to clean data. If you feel like taking different (or more) steps, you’re completely free to do it. Just remember to document everything you do and be ready to explain your decisions during your final presentation.

## Remove duplicate rows.

Use [DataFrame.drop\_duplicates()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.drop_duplicates.html?highlight=drop_duplicates#pandas.DataFrame.drop_duplicates) to remove rows that appear twice (or more) in the dataset: that means that these rows contain exactly the same values across all columns. Set the argument **inplace = True** so that changes actually take place. You can count the rows before and after running this code with **len()** and see how many rows were dropped this way.

It is possible that, after doing this, you still have a duplicate **sku**, which you should not have —Stock Keeping Units are meant to be unique.

## Deal with products with two dots in the price

You might have noticed prices that look like this: “304.223.23”. It makes no sense! There are several options here:

* If you think that all prices that look like this have been corrupted with a rule that you can reverse-engineer, go ahead and fix the prices!
* Otherwise, these prices will alter all the analysis that you intend to do. You can drop all the rows with this anomaly. Again, make sure you are documenting the issue and that when you present the results you include a big disclaimer warning about this.

In order to wrangle strings in a pandas DataFrame column, it is highly useful to take advantage of pandas adaptation of python’s string methods. Check out [this user guide](https://pandas.pydata.org/pandas-docs/stable/user_guide/text.html) for some examples and find a summary of all the string methods you can use at the very bottom.

**Regular Expressions** are also an important part of cleaning text.  In the course below, you will learn how to use them together with the Pandas syntax.

## Check for other weird numbers.

Some prices have only one decimal mark, but more than two decimal places (e.g., 99.004). Spotting this should raise suspicions: are these prices realistic for the products they correspond to? Can they just be rounded to 2 decimal places?

Use a combination of Boolean indexing with **.loc** and a string method to filter the rows with just these products, explore them and make a final call: should you keep or remove these rows?

## Deal with missing values.

Explore missing values in the product prices and decide what should be the strategy for dealing with them:

* If you can find a reason by which these prices are missing (e.g. they are products that do not belong to the catalogue and have never been sold), you should act in consequence (just drop them using [**Series.dropna**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.dropna.html?highlight=dropna#pandas.Series.dropna)).
* If you can come up with an imputation method that would fill the prices with a good estimate of their price (e.g. the mean price for products of their same brand), you can fill them using [**Series.fillna()**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.fillna.html?highlight=fillna#pandas.Series.fillna) (potentially, in combination with an indexing method such as .loc).
* If you cannot find any explanation or imputation method for some missing values at this stage, you can drop them for now and come back here in another iteration of your analysis. Do not lose too much time!

## Change prices to numeric.

By now, the price column should only contain valid numbers. You can use[**pd.to\_numeric()**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.to_numeric.html?highlight=to_numeric) to transform the data type of the column and finally be able to explore it in its due depth: making relevant aggregates and plots to discover how cheap or expensive Eniac’s products are.

The products DataFrame is probably the one that needed more cleaning, but the other might still need some work too. Check them one by one and, when you’re done, move on to the next lesson.

🏗 Working with datetimes & grouping data

Grouping and aggregating data is one of the most common exploratory tasks. Answering questions such as “How many products were sold in one particular order?” or “How many orders were placed in January 2017?” is essentially grouping and aggregating data.

In this lesson, we will recap the basics of pandas grouping and then learn how to use it together with dates and times.

## 📝 pandas datetimes

A special kind of grouping happens when you want to split the data based on time. To group by “week”, “month” or “year”, you need pandas to understand that the column where you have your timestamps is indeed a date-time type of data. The [**pd.to\_datetime()** function](https://pandas.pydata.org/docs/reference/api/pandas.to_datetime.html) is going to take care of that if you can make it understand in which format are your dates written.

You will also have to get familiar with the syntax of [the datetime library](https://docs.python.org/3/library/datetime.html) (the built-in Python way to deal with dates and times) and some of its methods.

## 📚 Recap: DataFrame.groupby()

**DataFrame.groupby()** will have to become your bread and butter. [This user guide](https://pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html) from pandas’ documentation site is a great resource to understand what the capabilities of the library are. One of the key insights to understand is that “aggregating” data is a 3-step process: splitting the data into groups, applying a function to each group independently and then combining the results into a single data structure. Let’s see it with an example: In this DataFrame each row is an order, and each order has a “total paid” value and a “state”:

A screenshot of a table

Description automatically generated

If we want to know what the average “total paid” value is for each one of the order “states”, the process to follow is:

1. Split the DataFrame into groups based on the “state” column.
2. Apply the **mean()** function to the “total\_paid” column in each group.
3. Combine the results into a single table.

In practice, you only must take care of the first two steps, asking yourself:

* By which column should you split the data?
* Which numerical column/s do you want to aggregate, and which function should you apply to each group?

In this case, here’s how we would do it:

A screenshot of a computer screen

Description automatically generated

…and this is the resulting DataFrame:

A screenshot of a screenshot of a number

Description automatically generated

data quality

Refers to certain attributes that a dataset or group of datasets possess, such as completeness, consistency, or reliability. In our case, we will have to make sure that the relationships between the different datasets check out.

The Data Frames are connected relationally (each row in orders should correspond to one or more rows in orderlines, and every product sold should be present in the products table…). This is something that SQL databases ensure by having database constraints (Primary and Foreign Keys). CSV files and Pandas Data Frames do not have to follow these rules: we have gained flexibility at the expense of consistency.

In this lesson, we guide you through some steps you can follow to ensure Data Quality. Similarly to the Data Cleaning process, there is not only one single solution for Data Quality. Follow our suggested steps first, but later on, feel free to come up with your own Data Quality approach and implement it.

## Step 1. Define pandas display format.

Checking for data quality often means scrutinising raw data, and using pandas in Jupyter notebooks often prevents you from having this completeness. You can change that by running the following lines at the top of your notebook:

import pandas as pd

pd.set\_option('display.float\_format', lambda x: '%.2f' % x)

pd.set\_option('display.max\_rows', 1000)

User guide: <https://pandas.pydata.org/pandas-docs/stable/user_guide/options.html>

## Step 2. Exclude unwanted orders.

Some of the orders present in the dataset were left in the **shopping cart or cancelled**. As the first step of data cleaning, consider excluding them so that, when you analyse the dataset, you deal with actual purchases from which the company is making actual money. Analysing why some orders were cancelled could be very interesting, but it is out of the scope of this project.

Secondly, make sure that orders in the dataset are in both orders and orderlines. Otherwise, you might consider orders that do not have products associated with or products sold without an order that links them with the customers and sellers involved. How can you perform such an operation? This diagram below should give you a hint.

A diagram of order

Description automatically generated

In summary, these three simple steps should do the trick:

1. orders 🡪 Keep only orders with the states that you want to work with.
2. orderlines 🡪 Keep only orders present in orders.
3. orders 🡪 Keep only orders present in orderlines.

## Step 3. Exclude orders with unknown products.

The **products table** should be the primary reference for all the products being sold. It is likely that, during the Data Cleaning phase, you deleted many rows from products. Any order involving products not present there is susceptible to containing corrupted information.

You might choose to deal with this in many different ways, but a conservative one would be to delete all the potentially corrupted rows. In the example below, you would delete all the rows shadowed in red:

A screenshot of a computer

Description automatically generated

Performing this operation in python as a relative beginner is not easy. Take your time and Google a lot! A starting point would be to look at the **isin** method and use it in combination with **.loc**. Using the **.index** attribute of Series and DataFrames can also be useful.

## Step 4. Explore the revenue from different tables.

All three tables we have seen (products, orderlines and orders) contain information about prices. Shouldn’t this information match? Well… yes and no. There are subtle differences:

* **products.price** –> this is the original price of a product, without any discount or promotion.
* **orderlines.unit\_price** –> this is the actual price at which a product has been sold. It should roughly match the price in the products table, but there might be deviations, as discounts get applied here.
* **orders.total\_paid** –> this is the total amount of the full order. It should roughly translate to the sum of all unitary prices, multiplied by the amount of each product purchased (orderlines.product\_quantity). The key word here is, again, roughly: the total amount of the order might include shipping costs or vouchers.

A good place to start is to compare **orders.total\_paid** **with orderlines.unit\_price \* orderlines.product\_quantity**. This comparison will require you to start by reshaping the orderlines table:

A screenshot of a computer

Description automatically generated

The grouped orderlines table can be merged with the orders table to easily compare the numbers:

A screenshot of a number

Description automatically generated

* What is the average difference between total\_paid and unit\_price\_total?
* What is the distribution of these differences?
* Can all the differences be explained by shipping costs? If not, what are other plausible explanations?
* If there are differences that you can’t explain: what should you do with these orders?

Perform a similar process comparing orderlines.unit\_price with products.price. Differences here should be explained by discounts, which is precisely the main objective of this project!

## Become confident about your data set.

Move forward with the Data Quality process on your own, making sure that data coming from different tables are either consistent or excluded from your dataset. You should document this process: hopefully, this documentation will be used to uncover problems in the data collection process or at any other point in the company’s data pipeline (data extraction is usually one of the weakest links).

## 🧩 Analyse discounts

The time has come to perform the analysis you were asked to do: finding out whether offering discounts is beneficial for the company. Before starting the analysis, let’s establish a couple of things:

Analysing discounts means looking at which products have been sold. Remember that this information is stored in the orderlines table.

Discounts are defined as the difference between orderlines.unit\_price and products.price. Merging both tables and creating a column that contains the discount is, thus, probably a good idea. Drop the columns that you don’t need so that they don’t confuse you.

At this stage, the DataFrame you’re working with should look similar to this one:

A screenshot of a product list

Description automatically generated

Now that you have your main analysis table, it would be a good moment to go back to the Case study and make sure you have the company’s goals in mind. When exploring discounts, make sure to always have the time dimension in consideration: Eniac does not always offer its products at the same price. A critical question, hence, is whether revenue grows whenever discounts increase.

<https://www.bestbuy.com/site/mobile-phone-accessories/iphone-accessories/pcmcat191200050015.c?id=pcmcat191200050015> for categories

🏗 7. Data Visualization with Seaborn

Plots are not just pretty graphics to show in a presentation. Plots have something unique in that they allow analysts to surprise themselves by finding something they were not specifically looking for. In the question-answer explorative process where we are at, this is immensely valuable.

## 📚 From Tableau to Seaborn

You have already visualised data with Tableau. The quickness of Tableau and drag-and-drop tools is great —feel free to upload your data there and assemble some visuals. Visualizing data with Python —and making the graphs look good— often takes some more work, but in some cases, the extra work is worth it. Here are some reasons why visualizing data with Python is a skill worth having:

* **Flexibility.** You can create more types of plots and make them look exactly how you want them to.
* **Reproducibility.** The code to create the plots can be shared and executed by anyone. It is like having step-by-step, automated, foolproof documentation on how the plots were created.
* **Simplicity**. If you are already using Python to apply transformations to the data and then exporting the data to Tableau, you are using two tools to do something you can do in one single tool. This becomes more of a burden when you need to iterate multiple times between these phases (e.g., trying out different data cleaning strategies and then visualizing the data).
* **Open source.** Python and its visualization libraries are open-source, while Tableau and most of the equivalent software are proprietary tools —and the licenses are not cheap. Remember that the free version (Tableau Public) does not preserve the confidentiality of your data.

## 📝 Intro to Seaborn

Use the learnings to further explore Eniac’s data visually. Whenever you decide to use one plot in your presentation, make sure to format it as an explanatory plot. Follow our 6-step recipe for creating great explanatory plots.

Create product categories

The main objective of this project is to use data to define a pricing strategy concerning discounts. However, the analysis of a numerical variable (**product prices**) is highly enriched by the presence of a meaningful categorical variable by which data can be sliced, filtered, and grouped. This categorical variable is going to be **product categories**.

## Tips for creating categories:

📝 As you read the tips below, go through [this notebook](https://colab.research.google.com/drive/18_iOvSitSSt6iTgSDFbVhB-GeWYF7Gus?usp=sharing) and create your own categories.

## 1. Browse through the data

Categories will emerge from the products table, specifically from the name and the desc columns. Explore them and come up with general rules about patterns (words, characters, sequences…) that can reliably tell that a certain product belongs to one of the categories you want to have.

## 2. Tweak options and settings

Set a high number for pandas max\_rows to be able to scroll through the DataFrame and be able to read the full product name and description, and check whether or not the rules you created make sense.

* pd.set\_option('display.max\_rows', 1000)
* pd.set\_option("display.max\_colwidth", 100)

## 3. Write a list of categories

Think about the level of granularity that your pricing analysis should have: would “Accessories” be a good category, or should you have different categories for, let’s say, external hard drives and keyboards? Should you have around 10 categories, or around 100?

## 4. Create simple rules

Look for the simple possible criteria for which you can create categories. For example:

* “Smartphones” –> the name starts with “apple iphone”.
* “Desktops” –> contains “imac” in the name.
* …

Start with rules that are simple to implement in Python!

## 5. Combine regex with pandas string methods

While pandas string methods are intuitive and convenient, Regular Expressions are more powerful and flexible. Good news: you can use both. Here we compile a regex that will look for “iPhone” at the beginning of the string, no matter whether the characters are in upper or lower case:

* iphone\_regex = re.compile(r"^iphone", flags=re.IGNORECASE)
* products.loc[products.name.str.contains(iphone\_regex, regex=True), "category"] = "smartphone"

## 6. Look for mistakes

Turns out that the rule we created to categorize smartphones matched this product:

* iPhone Case Speck Presidio Show 8/7 / 6s / 6 Transparent / Rose Gold

You will have to come up with a more precise rule to distinguish cases and other accessories from the product they relate to!

## 7. Use brands

The first three digits of the SKU refer to the brand of the item. Start by creating a brand column on your DataFrame, and then use it in combination with the name and description to refine your category rules.

## 8. Be agile

Don’t try to be 100% perfect at this stage of the project. Once you have a working script that creates categories, iterating through the rules criteria will be easy. Move on to the next phase of the project, where a potential blocker might await you, and come back to the categorization step once you are sure you are going to meet the deadline.

## 9. [BONUS] Create a function

If you already learned how to work with Python functions, you can dump all of your category creation code in a single function. It will help you keep things organized.

Using apply, as suggested [here](https://stackoverflow.com/questions/26886653/pandas-create-new-column-based-on-values-from-other-columns-apply-a-function-o), is also perfectly valid if you come up with code that does not make use of pandas .str methods.

Apple Accessories

Storage / Server

? / Apple Accessories

Storage

Other

Laptops / Desktops

Apple Accessories

Apple Accessories

Desktops

Apple Accessories

Apple Accessories ?

Monitors

Apple Accessories ?

Headphones

Storage ?

Storage ?

Speakers

Laptops

Repairs

Storage

Other ?

Laptops

Apple Accessories

Storage

Apple Accessories

Other / Accessories

Battery

Charger

Apple Accessories

Tablet

Apple Accessories

Desktops

Apple Accessories

Watches

Watches