Data exploration toolkit for cultural data

Stefano Rapisarda & Barbara Romero Ferron (RDM, UU)

2024-10-03

Table of contents

Welcome!

Welcome to the Cultural Data Exploration Toolkit! Throughout this workshop, you'll explore the nuances of data-driven analysis, from constructing your dataset and formulating research inquiries to learning data visualisation techniques. Along this journey, you'll contemplate the creation of personalised data models tailored to your research queries, navigate potential biases within datasets, and, importantly, learn how to effectively interrogate, explore, and analyse gathered information to generate visualisations.

Schedule

Time	Activity
13:00	Welcome - Icebreaker
13:15	Group activity: structuring data
13:45	Discussion
14:00	**Break**
14:10	Introduction to data toolkit
14:50	**Break**
15:00	Explorative Data Analyis
16:30	Recap & Questions

Part I

Preparation

Setting up

Running the analysis locally on your own computer

1. Install python and jupyter notebook. For installation and setup we point at the "Introduction to Python & Data" Installation & Setup page;
2. Create an empty directory called "cultural_data_analysis" (or any other name you prefer);
3. Inside the just created directory, create another directory called "data";
4. Click on this link, this will open a GitHub page. On the toolbar (on the right of the buttons "Raw" and copy), you will find the button for downloading the .csv file containing the data we used during our workshop. Download the file inside the just created data directory;
5. Click on this link, this will open the Jupyter-notebook in one of the instructor google colab environment containing the full analysis performed during the workshop. You can access the file only if you followed the workshop. At this point, you will just need to click on the "File" tab, select "download" (almost at the very end of the menu), select the "Download .ipynb" option, and download the file inside your project directory (the one also containing the data directory);
6. Open the jupyter notebook and have fun!

!!! WARNING !!! In the first cell of the Jupyter-notebook you downloaded, the instructors wrote specific Python instructions to make the notebook work in their Google colab environment. In your case, the only thing you need to do is to find out the full path of your project directory (the one containing the notebook and the data directory you just created) and writing this path instead of "working_directory", as described in the comments of the first cell.

Running the analysis from remote in your Google colab

1. In your personal Google Drive page, create a new directory called "cultural_data_analysis" (or any other name you prefer). For doing that, click on the "New" button on the top left corner of your browser (just below the Drive icon) and select directory;

2. Go inside the just created directory and create another directory called “data”;
3. Click on this link, this will open a GitHub page. On the toolbar (on the right of the buttons “Raw” and copy), you will find the button for downloading the .csv file containing the data we used during our workshop. Download the file inside the just created data directory;
4. Click on this link, this will open the Jupyter-notebook in google colab containing the full analysis performed during the workshop. You can access the file only if you followed the workshop. At this point, you will just need to click on the “File” tab, then click on “download” (almost at the very end of the menu), select the “Download .ipynb” option, and download the file inside your project directory (the one also containing the data directory). Alternatively, if you want to start from scratch creating a black jupyter notebook, click again on the “New” button, select “others”, and then Google collaboratory. Be sure that the just created jupyter notebook is in your project directory (the one containing your data directory);

!!! WARNING !!! In the first cell of the Jupyter-notebook you downloaded, the instructors wrote specific instructions to make the notebook work in their Google colab environment. In your case, the only thing you need to do is to change the value of the variable `work_dir` in the first cell from ‘/det_cultural_data’ to ‘/your_dir_name’. If the name of your directory is det_cultural_data, you do not need to change anything.

1 Introduction to Python

In this session, we will introduce you to Jupyter Notebooks and guide you through some of the essential programming concepts that form the foundation of working with data. Whether you are completely new to programming or looking to strengthen your understanding, this chapter is designed to help you build confidence as you take your first steps into coding.

This session is very python-specific, but (as you will find out in few lines) Jupyter Notebook is compatible with many programming languages and the programming concepts introduced here are fundamentals of every programming language.

A Jupyter Notebook is an interactive environment where you can write and execute code in small, manageable pieces, known as cells. This allows you to see the results of your code immediately, making it an excellent tool for learning, experimenting, and exploring data. It combines text, code, and the results of that code all in one place, making it a popular choice for data scientists, researchers, and educators.

In this session, we will be using Python, a programming language known for its simplicity and readability, which makes it ideal for beginners. We will cover core programming concepts, such as variables, functions, and loops, and you will learn how to apply these concepts to perform basic data exploration tasks.a

Getting familiar with Jupyter Notebook

Jupyter Notebook is an open-source web application that allows you to create and share documents containing live code, equations, visualizations, and narrative text. It supports various programming languages, including Python, R, Julia, and more. However, it is most commonly used with Python.

Jupyter Notebook provides an interactive computing environment where you can write and execute code in a series of cells. Each cell can contain code, markdown text, equations, or visualizations. You can run individual cells or the entire notebook to see the output of the code and the results of any computations.

The name "Jupyter" is a combination of three programming languages: Julia, Python, and R, which were the first three languages supported by the Jupyter project. It was originally developed as part of the IPython project (hence the name "Jupyter"), but has since evolved into a language-agnostic tool that supports multiple programming languages.

In the context of Python, Jupyter Notebook is a popular tool for data analysis, scientific computing, machine learning, education, and research. It allows users to write, test, and document Python code in an interactive and visually appealing manner, making it a valuable tool for both beginners and experienced programmers alike.

Jupyter Notebook cells

Jupyter notebook cells can be either code, markdown, or raw. For the simple purpose of programming and writing text, ignore the raw option. You can easily shift between code and markdown selecting the cell, pressing Esc, and then M for markdown or Y for code.

Markdown is a language for formatting text, it allows you to quickly and easily create formatted documents using simple and intuitive syntax. This current cell and any other cell displaying text in this notebook, is written in markdown. You can learn the basics of markdown syntax in few minutes reading here or simply looking at the content of the text cells in this notebook and see what happens when you select them and run them.

You can tell if your cell selected cell is a code cell because you will see square brackets on its left ([ ]:).

If you want to delete a cell, use Esc + DD (press Esc and then d twice)

WARNING: If your code cell has *empty* squred brackets, it means it has not been run YET.

Main programming concepts

There are some programming concepts that are common to all programming languages and can be found in any program:

- variables and data types;
- sequences of objects;
- functions;
- loops;
- conditional statements;
- packages (also called libraries or modules)

Variables and data types

In programming a variable is a container for a value. This value can either be a number, a string (a word), or any other type of programming object (we will talk about other possible objects later). Let's initialise (define for the first time) some variables:

```
name = 'Stefano'
favourite_planet = 'Saturn'
birth_day = 6
```

In the previous cell we stored the word *Stefano* into the variable `name`, the word *Saturn* into the variable `favourite_planet`, and the value 6 into the variable `birth_day`. From now on, every time we need to use one of these values in our programming, we just need to digit its corresponding variable name.

In Jupyter notebooks, if you want to check the value contained in a variable (so its content), you can simply run a cell with the variable name inside:

```
name
```

```
'Stefano'
```

```
birth_day
```

```
6
```

```
name
birth_day
```

```
6
```

As you can notice, when you write different variable names in the same cell, only the last one will be printed on the screen.

Sequences of objects

We can store single numbers and words inside a variable, but how about we want to store a *sequence* of values or words, or a mix of the two, into a variable? Of course we can, we just need to use a python object called **list**:

```
names = ['Stefano','Pippo','Alfio','Tano']
ages = [20,34,94,'unknown']
```

```
names[0]
```

```
'Stefano'
```

```
ages[3]
```

```
'unknown'
```

In Python lists are defined listing our sequence of values separated by coma inside square brackets (variable_name = [… , … , …]). Values stored in a list can be accessed using **indexing**. In python you count items starting from 0, so that the first item in a list has index 0. This means that for accessing the first item in the list `names` we will digit `name[0]`, and to access the last item in the list `ages` we will digit `ages[3]`

You can create lists of *any* object, even lists of lists:

```
info = [names,ages]
info[0]
```

```
['Stefano', 'Pippo', 'Alfio', 'Tano']
```

```
info[0][0]
```

```
'Stefano'
```

If we want to change a particular value in a list, we first need to access it and then we need to use the operator = to specify the new list value. For example, if we want to change 'Stefano' into 'Steve', we would do:

```
info[0][0] = 'Steve'
info[0]
```

```
['Steve', 'Pippo', 'Alfio', 'Tano']
```

Data types: Dictionaries

In python there are several ways you can store information. We just talked about lists, simple ordered sequences of objects. Another kind of data structure is called **dictionary**. In general a dictionary is a reference or resource that provides information, definitions, or explanations of words, terms, concepts, or objects. A dictionary is usually organised by alphabetically ordered words and by explanations associated to each word. In python a dictionary follows exactly the same organization principle: keyword and value.

```
info_dict = {'name':'Stefano','favourite_number':6}
```

To define a dictionary we use curled brackets ({}) instead of squared brackets. Inside the curled brackets we need to specify couples of key/values separated by comas. To each key we can associate a different python object. Keys need to be unique, while values can by any Python object.

```
info_dict = {'names':names}
```

```
info_dict['names']
```

```
['Steve', 'Pippo', 'Alfio', 'Tano']
```

To access the values contained inside a dictionary you cannot use numerical indices, as you would do for lists. Instead, you must use the name of the key related to the value. In the

previous case the object names (a Python list) is associated with the key 'names'. So, in order to access it, we need to digit `info_dict['names']`.

In a similar way, if you want to change the value related to a key, or create a new key/value couple, you first need to access that value and then to use the '=' sign to assign a new value

```
info_dict['names'] = ['Steve','Josef','Alfonse','Gerrit']
info_dict['names']
```

```
['Steve', 'Josef', 'Alfonse', 'Gerrit']
```

Functions

A function is a python object that performs a single action given some parameters. In python, function names are usually verbs. If variable can be thought as subjects and objects in a sentence, functions are the verbs. Python has already some default functions, functions that are ready to use. Here you can find the build-in Python functions.

```
result = print(name)
result
```

```
Stefano
```

The way a function works is common to all programming languages: you give to the function one or more parameters, the function performs an action, and it returns a result. This happens so fast that, as a matter of fact, you can already think at a function and its parameters as its result. In the previous cell the function `print()` got as an input parameter the variable name and it printed its value on the screen.

```
numbers = [1,2,3,4,5,6,7,8,9,10]
result = sum(numbers)
print(result)
```

```
55
```

In the previous cell, we defined a list of values (the first 10 integers), then we used the function sum to (guess what??) sum all the numbers in the list, and we stored the result into the variable `result`. We finally printed the result using the function `print()`.

Because we know that variable values are printed automatically in Jupyter notebook cells when they contain the variable name, we could write directly:

```
sum(numbers)
```

```
55
```

Indeed `sum(numbers)` represents an operation that returns the value 55 and can be considered equivalent to the value 55 itself, so that when we write it in a cell, we obtain the result printed on the screen.

```
len(numbers)
```

```
10
```

The function `len()` is one of the most used function on objects containing many items. Indeed it tell us how many items are contained in that object (i.e. the length of that object). The function `type()` returns the type of a variable:

```
type(numbers)
```

```
list
```

How many functions are there? thousands, probably millions. Some of them have very intuitive names (like `print()` and `sum()`), some others have more complicated names. However, every function that can be used in Python comes with its own documentation, explaining which parameters it accepts, which additional options you can specify, and which kind of result you get back when applying it. To find about a function just google " Python documentation" or ask ChatGPT about that.

Methods

Methods are functions that are object-specific. What does it mean? There are certain operations that can be perfomed only on a certain type of object. For example, if we consider a function that transforms lower characters into capital letters, it would not make much sense to apply this function to a number.

All objects in python can have their own specific functions and these object-specific functions are called **methods**. To use a method on an object, you need to apply the syntax

`<object_name>.method()`. Do you see the difference compared to a general function syntax? In a general function we have `function(par1,par2,...)`, while in a method we already know that the function, in this case method, will be applied to its object. Therefore, inside the parenthesis we only have additional parameters.

Like functions, methods can accept all kind of parameters, but, of course, their main parameter is the object itself. Let's see some example:

```
name = 'Stefano Rapisarda Arthurus Micaelus'
numbers = [1,2,3,4,5]
```

```
name.split()
```

```
['Stefano', 'Rapisarda', 'Arthurus', 'Micaelus']
```

We initialised two variables: a string made of several words and a list of numbers. The split() method (a string-specific function) divides the string into a list of strings according to a separator. If you dont' specify any separator (like in our case), white spaces will be considered as separators. Let's see another example:

```
numbers.pop(2)
```

```
3
```

```
numbers
```

```
[1, 2, 4, 5]
```

We initialised two variables: a string made of several words and a list of numbers. The `split()` method (a string-specific function) divides the string into a list of strings according to a separator. If you dont' specify any separator (like in our case), white spaces will be considered as separators. The variable `numbers` is already a list and using the method `pop(x)` we can remove the item occupying the 3rd position (index 2). The method affects the list and returns the just removed value.

How can we found about methods if there are so many? Usually a google search can point you at the method or function you need. In general you can always consult python documentation. You will find about string a list methods here and here, respectively.

Loops

One of the potential of using machines is making them repeating the same operation hundreds, millions, or billions of times.

Let's say I have a list of names and I want to print them on the screen one by one:

```
names = ['James','Martin','Sandra','Paul','Chani']
print(names[0])
print(names[1])
print(names[2])
print(names[3])
print(names[4])
```

```
James
Martin
Sandra
Paul
Chani
```

This did not take us much time, because the names are only 5, but imagine you have a list of 1000 names; in that case printing all the names could take hours. Looking at the previous cell we notice that we use repeatedly the function `print()` using as input the values contained in the list `names`. Every time we need to repeat an operation many times, we can use a **loop**, specifically a for loop:

```
for i in range(5):
    print(i,names[i])
```

```
0 James
1 Martin
2 Sandra
3 Paul
4 Chani
```

In the previous cell the same operation (`print()`) is executed 5 times, but at each step, so at each **iteration**, the variable i changes, going from 0 to 4, one step at the time.

In order to achieve this result we need to start declaring `for i`. `i` is the variable name acting as a place holder for a value that will change at every step of the iteration. We chose the letter i, but you can choose any other name. After `for i`, we need to specify which values `i` can

assume at each iteration. `in range(5)` means that `i` will go from 0 to 4, so it will increase of 1 integer per iteration stopping just before 5. Instead of a range of numbers, we can specify any other object containing several objects in it. In that case, the variable `i` (or whatever you will decide to call it), at each iteration, will be initialized with each value contained in the specified object. Let's see some example:

```
for a in range(12): print(a)
```

```
0
1
2
3
4
5
6
7
8
9
10
11
```

```
for value in [0,1,4,5,6,7]:
    print(value)
```

```
0
1
4
5
6
7
```

```
for name in names:
    print(name)
```

```
James
Martin
Sandra
Paul
Chani
```

For *looping* over dictionaries, the concept is the same, but the syntax is a bit different because of the key/value stricture of dictionaries:

```
info_dict = {
    'name':'Stefano',
    'surname':'Rapisarda',
    'favourite_number':6,
    'favourite_planet':'Saturn'
}
for key,value in info_dict.items():
    print(key,':',value)
```

```
name : Stefano
surname : Rapisarda
favourite_number : 6
favourite_planet : Saturn
```

WARNING You noticed that after the for statement, there is an indent of 4 spaces. You can make that indent using the TAB key. That indent tells python that that specific line of code is inside the look and, therefore, needs to be repeated. When you write code without indents, before or after the loop, those lines will be executed normally, i.e. once, one after the other.

```
print('Beginning of the for loop, we will have 10 iterations')
print('='*72)
for i in range(10):
    print('This is iteration number:',i)
    print('The next iteration will be:',i+1)
    print('End of iteration',i)
    print('-'*62)
print('='*72)
print('End of the loop')
```

```
Beginning of the for loop, we will have 10 iterations
========================================================================
This is iteration number: 0
The next iteration will be: 1
End of iteration 0
--------------------------------------------------------------
This is iteration number: 1
The next iteration will be: 2
End of iteration 1
```

```
------------------------------------------------------------
This is iteration number: 2
The next iteration will be: 3
End of iteration 2
------------------------------------------------------------
This is iteration number: 3
The next iteration will be: 4
End of iteration 3
------------------------------------------------------------
This is iteration number: 4
The next iteration will be: 5
End of iteration 4
------------------------------------------------------------
This is iteration number: 5
The next iteration will be: 6
End of iteration 5
------------------------------------------------------------
This is iteration number: 6
The next iteration will be: 7
End of iteration 6
------------------------------------------------------------
This is iteration number: 7
The next iteration will be: 8
End of iteration 7
------------------------------------------------------------
This is iteration number: 8
The next iteration will be: 9
End of iteration 8
------------------------------------------------------------
This is iteration number: 9
The next iteration will be: 10
End of iteration 9
------------------------------------------------------------
============================================================================
End of the loop
```

Conditional statements

We have seen how to store data and information into variables and how to access this information by indexing, so referring to the position of values inside an object. How about selecting information using other criteria? What about if we want to visualize only peoples names if

they are older than 30 or printing the names of towns that start with an 's'? To do that in programming we need to use **conditional statements**. Conditional statements are indeed conditions that need to be satisfied in order for something to happen. What is "something"? Whatever action we want: an operation, a printing function, etc.

```
for key,value in info_dict.items():
    if 'favourite' in key:
        print(key,':',value)
    else:
        print('Not interested!')
```

```
Not interested!
Not interested!
favourite_number : 6
favourite_planet : Saturn
```

We used the same for loop to explore dictionaries keys and values, but this time, inside it, we wrote a conditional statement. The syntax for a conditional statement is:

```
if <condition>:
    action
else:
    other_action
```

`<condition>` is the condition that needs to be satisfied. In this case we want the word 'favourite' to be contained inside the key. If this happens, the condition is True and the "action" is perfomed (in our case, key and value will be printed). If the condition is False, the "other_action" will be perfomed (in our case, the 'Not interested' message will be printed).

You can also make conditions comparing quantities:

```
numbers = [1,2,3,4,5,6,7]
for number in numbers:
    if number < 4:
        print(number,'is smaller than 4')
    elif number > 4:
        print(number,'is larger than 4')
    elif number == 4:
        print(number,'is exactly 4')
```

```
1 is smaller than 4
2 is smaller than 4
3 is smaller than 4
4 is exactly 4
5 is larger than 4
6 is larger than 4
7 is larger than 4
```

Conditional statements may also be combined:

```
for number in numbers:
    if (number < 4) or (number > 4):
        print(number, 'is not 4')
    else:
        print(number, 'must be 4')
```

```
1 is not 4
2 is not 4
3 is not 4
4 must be 4
5 is not 4
6 is not 4
7 is not 4
```

In the previous case we used three conditions that are satisfied if a number is smaller, larger, and equal to 4.

Using loops in combination with conditional statements is particularly useful when it's time to select data. For example, imagine we have data in a table with two columns, one containes years and the other column can be any kind of measurement. In this case, you can use conditional statements to select measurements and very specific time intervals.

Packages

There are millions of functions and objects out there, how can we use them? Python installation does not come with ALL the functions ever written for Python. Functions and objects are usually organized in **packages** (also called libraries or moduli). Each package contains a set of tools specific for certain tasks. There are tools for statistics, machine learning, building website, text-mining, etc. How can we access all these tools? First of all, we need to download

the package into our computer. Usually in the documentation page of the package, there are installation instruction. Once installed, the package needs to be imported.

```
import pandas as pd
from matplotlib import pyplot as plt
```

In the previous cell we imported two packages, pandas and pyplot. When we import something, it is convenient to choose an alias for it, so that, when needed, we don't need to write its entire name. In our case, `pd` will be the alias for pandas.

In the second line we see a slightly sifferent statement. In this case, we import the package pyplot. The package is a sub-package of the massive library matplotlib. Therefore, we need to specify the macro-package containing pyplot. We could also import pyplot in the following way.

```
import matplotlib.pyplot as plt
```

From now on, every time we will need to use a pandas function or object, we just need to specify the alias of the package before the function or object we want to use:

```
df = pd.DataFrame()
```

In the previous case, we initialised a variable called `pd` with a pandas DataFrame. Let's see what happens if we forget to specify `pd`:

```
df = DataFrame()
```

```
NameError: name 'DataFrame' is not defined
```

We obtain an error because Python does not recognize the function name.

Part II

Data Workflow

2 Organising your data

3 Reading data

The first thing we need to do is loading the data. This means opening the file where the data is currently stored and transfer that data here, in our working environment. As we are working with Python in this Jupyter notebook environment, this means transfering all the data into a Python object. Which object? There are Python libraries (Python code written by other developers) that have been specifically designed to perform the task of data analysis. One of these libraries, or ("Pythonically" speaking) **packages**, is called `pandas`. We will use one of the many `pandas` functions to read our `.csv` (coma separated values file) file and we will store the information into a pandas `DataFrame`.

```
<IPython.core.display.HTML object>
```

```
import pandas as pd
data_file = 'data/data.csv'
df = pd.read_csv(data_file)
print(type(df))
```

```
<class 'pandas.core.frame.DataFrame'>
```

We managed to transfer our data into a Python object, specifically a `pandas.core.frame.DataFrame`, or simply (from now on) a DataFrame. However, a lot of things can go wrong when going from one format to another, so it is a good idea to have a first look at the data.

```
<IPython.core.display.HTML object>
```

```
df.head(10)
```

	Year of arrival at port of disembarkation	Voyage ID	Vessel name	Voyage itinerary imputed port
0	1714.0	16109	Freeke Gally	Bristol
1	1713.0	16110	Greyhound Gally	Bristol
2	1714.0	16111	Jacob	Bristol

	Year of arrival at port of disembarkation	Voyage ID	Vessel name	Voyage itinerary imputed port
3	1714.0	16112	Jason Gally	Bristol
4	1713.0	16113	Lawford Gally	Bristol
5	1714.0	16114	Mercy Gally	Bristol
6	1714.0	16115	Mermaid Gally	Bristol
7	1713.0	16116	Morning Star	Bristol
8	1714.0	16117	Peterborough	Bristol
9	1713.0	16118	Resolution	Bristol

Comparing what we see here with our .csv file it seems that everything went well. We have the data organised in rows and columns. Each column has a name and each row and index. Looking at our data, some values are numbers, some are names and places, some contain htmlo tags, some are NaN. It is not time yet to run data analysis, after having loaded the data we still need to correctly interpret the information it contains, then we need to "clean" it, and after that, finally, we can proceed with some data analysis. This is just the beginning, but the best is yet to come!

4 Exploring data

Previous steps

```
import pandas as pd
data_file = 'data/data.csv'
df = pd.read_csv(data_file)
```

```
df.head(5)
```

	Year of arrival at port of disembarkation	Voyage ID	Vessel name	Voyage itinerary imputed port
0	1714.0	16109	Freeke Gally	Bristol
1	1713.0	16110	Greyhound Gally	Bristol
2	1714.0	16111	Jacob	Bristol
3	1714.0	16112	Jason Gally	Bristol
4	1713.0	16113	Lawford Gally	Bristol

Now that we correctly loaded our data in our working environment, it is time to figure out what the data contains. It is always a good idea to look at the dataset documentation (or metadata) to understand where the data comes from, what is the source of all the different records, how data has been collected, and any other possible data related caveat. Diving into the data documentation is up to you, in this chapter what we want to do is understanding as much as we can from the data itself, looking at its columns, rows, and values. Every dataset tells a story. You may think about it like a person with a long experience, but not really willing to talk (well, some datasets "talk" more easily than others). It is your role in this case to "interrogate" the data, let it to talk, to tell a story and to dive into the details of that story, getting as much information as you can. This also depends on how much you need to know: will you be satisfied by a small "chat" or you need to know all kind of details? Let's formulate some questions to begin with.

```
<IPython.core.display.HTML object>
```

```
df.shape
```

```
(36151, 9)
```

```
solution = 'Our DataFrame contains data distributed in 36151 rows and 9 columns. '
question_box(solution=solution)
```

```
<IPython.core.display.HTML object>
```

It is a quite big dataset. Shall we care about how big is our dataset? We should as this may affect our analysis. For example, if we implement a scientific analysis that requires 1 second per row to produce an output, such program would take about 10hrs to analyse the entire dataset, and that is something we should keep in mind. That is why, in general, it is a good idea to test large analysis programs on a small sub-set of data and then, once verified that everything runs smoothly, to perform the analysis on the entire dataset.

Let's continue exploring our DataFrame. We have 9 columns, we saw them displayed in our notebook and, luckily enough, their names are pretty descriptive, therefore, in this case, it is quite intuitive to understand what kind of information they contain. It could be useful to store the column names inside a Python variable and to display their names with a corresponding index (this will be useful later).

```
<IPython.core.display.HTML object>
```

```
column_names = df.columns
print(column_names)
i=0
print("Index ) Column name")
for name in column_names:
    print(i,")",name)
    i = i + 1
```

```
Index(['Year of arrival at port of disembarkation', 'Voyage ID', 'Vessel name',
       'Voyage itinerary imputed port where began (ptdepimp) place',
       'Voyage itinerary imputed principal place of slave purchase (mjbyptimp) ',
       'Voyage itinerary imputed principal port of slave disembarkation (mjslptimp) place',
       'VOYAGEID2', 'Captives arrived at 1st port', 'Captain's name'],
      dtype='object')
Index ) Column name
```

```
0 ) Year of arrival at port of disembarkation
1 ) Voyage ID
2 ) Vessel name
3 ) Voyage itinerary imputed port where began (ptdepimp) place
4 ) Voyage itinerary imputed principal place of slave purchase (mjbyptimp)
5 ) Voyage itinerary imputed principal port of slave disembarkation (mjslptimp) place
6 ) VOYAGEID2
7 ) Captives arrived at 1st port
8 ) Captain's name
```

Now we have the column names nicely listed from top to bottom and with their corresponding index assigned to them. You might be tempted to start the indexing from 1, but as in Python the first element of a list (or any other series of elements) has index 0, we started counting from zero. You can obtain the same result with less lines of code, try it out!

```
print("Index) Column name")
for i,name in enumerate(column_names):
    print(f"{i}) {name}")
```

```
Index) Column name
0) Year of arrival at port of disembarkation
1) Voyage ID
2) Vessel name
3) Voyage itinerary imputed port where began (ptdepimp) place
4) Voyage itinerary imputed principal place of slave purchase (mjbyptimp)
5) Voyage itinerary imputed principal port of slave disembarkation (mjslptimp) place
6) VOYAGEID2
7) Captives arrived at 1st port
8) Captain's name
```

It is now time to figure out what are the rows about. Looking at the column names, we notice that the second one (index 1) is called "Voyage ID". This indicates that this column contains a specific identifier for the ship voyage, implying that each row contains specific information about a single trip. To verify that each row corresponds to a single voyage, we need to check if all the values of the Voyage ID column are different, i.e. if they are unique.

```
<IPython.core.display.HTML object>
```

```
voyage_id = df.iloc[:,1]
print(voyage_id.is_unique)
```

```
True
```

We verified that all the values of the Voyage ID column are unique, this means that each of the rows of our DataFrame refers to a single ship voyage. Looking at the other columns, we also notice that information where the voyage began, the port where slaves have been purchased, and the port where slaves have been desembarked is provided. Looking in particular at the fifth column (index 4, "Voyage itinerary imputed principal place of slave purchase"), we notice it contains several NaNs. NaN stands for "Not a Number", it is a value that appears when something goes wrong in one of the processes ran by our program. If something went wrong, why did not our program stop or told us something about an occuring problem? Because problems may happen more often than you think and if our program stops working everytime it encounters a situation it cannot handle, it would most probably never finish running! In this case, most probably the record does not exist so the data set cell has been filled by NaN, either in our original .csv file or by the `pandas` method `.read_csv()`. NaN are not necesseraly something bad, as they can be easily identified and eventually corrected (or simply ignored). Incorrect or missing data may be much harder to spot and correct. In any case, the presence of NaNs or any other missing value can severely affect our data analysis, for this reason before starting analysing the data we need to find and get rid of those values. This process is usually called "data cleaning" and that is exactly what we are going to do in the next chapter.

5 Cleaning data

Previous steps

```
import pandas as pd
data_file = 'data/data.csv'
df = pd.read_csv(data_file)
print(df.shape)
```

```
(36151, 9)
```

```
column_names = df.columns
df.head(5)
```

	Year of arrival at port of disembarkation	Voyage ID	Vessel name	Voyage itinerary imputed port
0	1714.0	16109	Freeke Gally	Bristol
1	1713.0	16110	Greyhound Gally	Bristol
2	1714.0	16111	Jacob	Bristol
3	1714.0	16112	Jason Gally	Bristol
4	1713.0	16113	Lawford Gally	Bristol

Now that we got some familiarity with our dataset, it is time to clean our data, i.e. to get rid of all those NaN values and anything else that might effect our data analysis. Where to start? Well, inspecting the DataFrame by eye, we see several NaNs in the first 5 rows of our DataFrame. The first column we see NaNs is "Voyage itinerary imputed principal place of slave purchase", the fourth column (index 5). It would be nice to check if also other column have NaNs. Let's start with the first column, "Year of arrival at port of disembarkation" (index 0), let's check if this column contains any NaN and then we will repeat the same process for all the other columns.

```
<IPython.core.display.HTML object>
```

```
arr_year = df.iloc[:,0]
arr_year_na = arr_year.isna()
print(arr_year_na)
print('Total number of NaNs in the first column:',arr_year_na.sum())
```

```
0        False
1        False
2        False
3        False
4        False
         ...
36146    False
36147    False
36148    False
36149    False
36150    False
Name: Year of arrival at port of disembarkation, Length: 36151, dtype: bool
Total number of NaNs in the first column: 1
```

```
solution = 'The first column contains 1 NaN value'
question_box(solution=solution)
```

```
<IPython.core.display.HTML object>
```

In this way we found our that the first column has 1 NaN (or na) value, that would have been quite hard to spot by eye scrolling 36151 lines! It is great that we found 1 NaN in the first column, but where exactly it is located? What's the corresponding Voyage ID of that value?

```
<IPython.core.display.HTML object>
```

```
df[arr_year_na]
```

	Year of arrival at port of disembarkation	Voyage ID	Vessel name	Voyage itinerary imputed port v
32248	NaN	91909	Kitty	Liverpool

```
<IPython.core.display.HTML object>
```

In this way we can inspect NaNs one by one and we can make a decision about how to handle them. In our DataFrame there are thousands of NaNs (as you will see in a minute) and going through ALL of them one by one is not a good idea. Let's first try to figure out if the other columns have also NaNs and how many are they. The process will be quite straightforward as we already did it for one of the columns, so what we need to do now is to repeat the same procedure for all the other columns.

```
<IPython.core.display.HTML object>
```

```
for column_name in column_names:
    selected_column = df[column_name]
    selected_column_na = selected_column.isna()
    n_nan = selected_column_na.sum()
    print(column_name,'has',n_nan,'NaN')
```

```
Year of arrival at port of disembarkation has 1 NaN
Voyage ID has 0 NaN
Vessel name has 1614 NaN
Voyage itinerary imputed port where began (ptdepimp) place has 4508 NaN
Voyage itinerary imputed principal place of slave purchase (mjbyptimp)  has 2210 NaN
Voyage itinerary imputed principal port of slave disembarkation (mjslptimp) place has 4191 Na
VOYAGEID2 has 36101 NaN
Captives arrived at 1st port has 17743 NaN
Captain's name has 4028 NaN
```

and if we want to keep in mind the column index of each column...

```
for i,column_name in enumerate(column_names): \
    print(f"{i}) {column_name} has {df[column_name].isna().sum()} NaN")
```

```
0) Year of arrival at port of disembarkation has 1 NaN
1) Voyage ID has 0 NaN
2) Vessel name has 1614 NaN
3) Voyage itinerary imputed port where began (ptdepimp) place has 4508 NaN
4) Voyage itinerary imputed principal place of slave purchase (mjbyptimp)  has 2210 NaN
5) Voyage itinerary imputed principal port of slave disembarkation (mjslptimp) place has 419
6) VOYAGEID2 has 36101 NaN
7) Captives arrived at 1st port has 17743 NaN
8) Captain's name has 4028 NaN
```

```
<IPython.core.display.HTML object>
```

At this point we have a general idea of the amount of data missing in our DataFrame. The following question is how to deal with this missing data? There are several things we can do, the easiest option would be just exclude it from our DataFrame. However, in order to answer a research question, we often do not need to use or explore ALL the available information and we would usually be interested in some parameters more than others. In this case our data selection could be performed looking at one or more specific columns. What to do with the rest of the NaNs? We can either leave them as they are and trying to figure out how our analysis program will "digest" these values or find good substitute for them. The value of this substitute will depend on the data type of the columns containing the NaN and on our decision. For example the NaN in the columns containing a descriptive string, like the vessel name or the starting port, could be substituted by the string "unknown". NaNs in the "Captives arrived [...]" column could be left as they are (you may be tempted to change them to 0, but zero captives is quite different from unknown number of captives) or substituted by, for example, the average of captives during the same year. Each choice will have different implications to our final results, the most important thing in this stage is to clearly document our criteria for filtering NaN. In our specific case we will be mostly interested in the data containing the number of captives, so we want to filter our all those rows where the number of captives is NaN. We will then exclude the columns VOYAGEID2 as we already have a voyage ID and it is not listed in the data variable description. To resume, here there are our cleaning criteria: - All the rows not containing data about the number of captives have been removed; - All the NaN values in columns with descriptive information (e.g. names) have been substituted with "unknown"; - The column VOYAGEID2 has been removed from the DataFrame.

```
<IPython.core.display.HTML object>
```

```
# Display the name of the columns first
print(df.columns)

# Select our target columns for clearning the data
column_to_remove = 'VOYAGEID2'
column_to_remove_nan = 'Captives arrived at 1st port'

# Perform Data Cleaning visualising the result step by step
# step1, removing column VOYAGEID2 from the DataFrame
cleaned_df_step1 = df.drop(column_to_remove,axis=1)
cleaned_df_step1.head(5)
```

```
Index(['Year of arrival at port of disembarkation', 'Voyage ID', 'Vessel name',
       'Voyage itinerary imputed port where began (ptdepimp) place',
```

```
       'Voyage itinerary imputed principal place of slave purchase (mjbyptimp) ',
       'Voyage itinerary imputed principal port of slave disembarkation (mjslptimp) place',
       'VOYAGEID2', 'Captives arrived at 1st port', 'Captain's name'],
      dtype='object')
```

	Year of arrival at port of disembarkation	Voyage ID	Vessel name	Voyage itinerary imputed port
0	1714.0	16109	Freeke Gally	Bristol
1	1713.0	16110	Greyhound Gally	Bristol
2	1714.0	16111	Jacob	Bristol
3	1714.0	16112	Jason Gally	Bristol
4	1713.0	16113	Lawford Gally	Bristol

```
# step2, removing all the rows haveing NaN in the "Captives arrived at 1st port" column
cleaned_df_step2 = cleaned_df_step1.dropna(subset=[column_to_remove_nan])
cleaned_df_step2.head(5)
```

	Year of arrival at port of disembarkation	Voyage ID	Vessel name	Voyage itinerary imputed port wl
0	1714.0	16109	Freeke Gally	Bristol
2	1714.0	16111	Jacob	Bristol
3	1714.0	16112	Jason Gally	Bristol
5	1714.0	16114	Mercy Gally	Bristol
6	1714.0	16115	Mermaid Gally	Bristol

```
# step3, changing all the other NaN into unknown
cleaned_df = cleaned_df_step2.fillna("unknown")
cleaned_df.head(5)
```

	Year of arrival at port of disembarkation	Voyage ID	Vessel name	Voyage itinerary imputed port wl
0	1714.0	16109	Freeke Gally	Bristol
2	1714.0	16111	Jacob	Bristol
3	1714.0	16112	Jason Gally	Bristol
5	1714.0	16114	Mercy Gally	Bristol
6	1714.0	16115	Mermaid Gally	Bristol

```
# step4, checking how much data we filtered out
print(cleaned_df.shape)
```

```
n_filtered_rows = len(df)-len(cleaned_df)
per_cent = (n_filtered_rows/len(df))*100
print('We filtered out: ',len(df)-len(cleaned_df),', corresponding to about', round(per_ce
```

```
(18408, 8)
We filtered out:  17743 , corresponding to about 49 % of our initial data
```

It seems that because of our filtering, almost half of our data will be excluded from the analysis. This is a quite large percent and we may decide to re-think our filtering criteria to include more data. For example, we could substitue the missing value in the Captives column with an avarage number of captived per trip. For the purpose of our workshop, we will keep the current filtering criteria and keep our filtered DataFrame as it is.

At this point we obtained a "clean" DataFrame, `cleaned_df`, containing 18408 rows with values organised in 8 columns. We can now start diving deep in the analysis of our DataFrame, we are ready to interrogate this dataset and see which kind of story it is going to tell us.

6 Analysing data

Previous steps

```
import pandas as pd
data_file = 'data/data.csv'
df = pd.read_csv(data_file)
cleaned_df = df.drop('VOYAGEID2',axis=1).dropna(subset=['Captives arrived at 1st port']).f
cleaned_col_names = cleaned_df.columns
cleaned_df.head(10)
```

	Year of arrival at port of disembarkation	Voyage ID	Vessel name	Voyage itinerary imputed
0	1714.0	16109	Freeke Gally	Bristol
2	1714.0	16111	Jacob	Bristol
3	1714.0	16112	Jason Gally	Bristol
5	1714.0	16114	Mercy Gally	Bristol
6	1714.0	16115	Mermaid Gally	Bristol
8	1714.0	16117	Peterborough	Bristol
9	1713.0	16118	Resolution	Bristol
10	1714.0	16119	Richard and William	Bristol
11	1713.0	16120	Rotchdale Gally	Bristol
12	1714.0	16121	Tunbridge Gally	Bristol

```
print("Index) Column name")
for i,name in enumerate(cleaned_df.columns):
    print(i,")",name)
```

```
Index) Column name
0 ) Year of arrival at port of disembarkation
1 ) Voyage ID
2 ) Vessel name
3 ) Voyage itinerary imputed port where began (ptdepimp) place
4 ) Voyage itinerary imputed principal place of slave purchase (mjbyptimp)
```

```
5 ) Voyage itinerary imputed principal port of slave disembarkation (mjslptimp) place
6 ) Captives arrived at 1st port
7 ) Captain's name
```

Data Analysis

It is finally time to ask questions to our data. Let's start with some simple ones regaring the time span of our dataset.

```
<IPython.core.display.HTML object>
```

```
arrival_year = cleaned_df.iloc[:,0]
first_year = min(arrival_year)
last_year = max(arrival_year)
year_span = last_year-first_year

print(first_year)
print(last_year)
print(year_span)
```

```
1520.0
1866.0
346.0
```

```
arrival_year_raw = df.iloc[:,0]
first_year_raw = min(arrival_year_raw)
last_year_raw = max(arrival_year_raw)
year_span_raw = last_year_raw-first_year_raw

print(first_year_raw)
print(last_year_raw)
print(year_span_raw)
```

```
1514.0
1866.0
352.0
```

```
<IPython.core.display.HTML object>
```

We can keep asking questions about numerical values. We focused on time in our last question, let's focus on the number of captives this time.

```
<IPython.core.display.HTML object>
```

```
n_captives = cleaned_df.iloc[:,6]
tot_captives = sum(n_captives)
ave_cap_per_voyage = tot_captives/len(cleaned_df)
ave_cap_per_year = tot_captives/year_span
print('Total n. of captives:',tot_captives)
print('Average captives per voyage',round(ave_cap_per_voyage))
print('Average captives per year',round(ave_cap_per_year))
```

```
Total n. of captives: 5082756.0
Average captives per voyage 276
Average captives per year 14690
```

```
filtered_rows = len(df)-len(cleaned_df)
tot_captives_ext = tot_captives + ave_cap_per_voyage*filtered_rows
ave_cap_per_year_adj = tot_captives_ext/year_span_raw
print('Extimated total n. of captives',round(tot_captives_ext))
print('Adjusted average captives per year', round(ave_cap_per_year_adj))
```

```
Extimated total n. of captives 9981894
Adjusted average captives per year 28358
```

```
<IPython.core.display.HTML object>
```

So far we computed numbers, but data can be most effectively described using visualization. In our DataFrame we have information about three different locations: the place where the voyage started, the principal port of slave purchase, and the principal port of slave disembarkation. Let's have a closer look at these locations.

```
<IPython.core.display.HTML object>
```

```
start_port = cleaned_df.iloc[:,3]
start_port_counts = start_port.value_counts()
```

```
print(type(start_port_counts))
start_port_counts
```

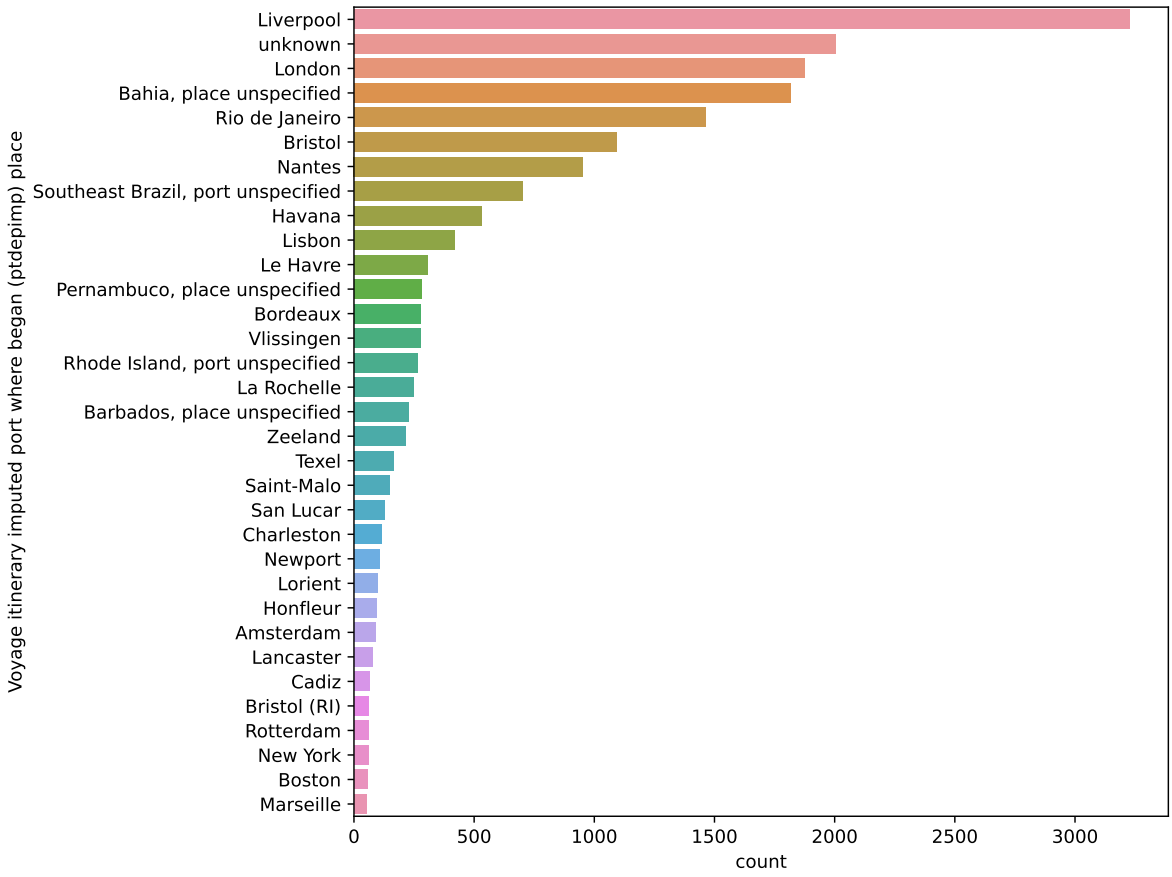
```
<class 'pandas.core.series.Series'>
```

```
Voyage itinerary imputed port where began (ptdepimp) place
Liverpool                          3227
unknown                            2005
London                             1874
Bahia, place unspecified           1815
Rio de Janeiro                     1464
                                   ...
Mangaratiba                           1
Mediterranean coast (France)          1
Canasí                                1
Santa Catarina                        1
Portland                              1
Name: count, Length: 176, dtype: int64
```

```
import seaborn as sns
import matplotlib.pyplot as plt

fig, new_ax = plt.subplots(nrows=1,ncols=1,figsize=(8,8))
filter = start_port_counts > 50
x_data = start_port_counts[filter]
y_data = start_port_counts.index[filter]
sns.barplot(ax=new_ax,x=x_data,y=y_data)
```

```
<Axes: xlabel='count', ylabel='Voyage itinerary imputed port where began (ptdepimp) place'>
```



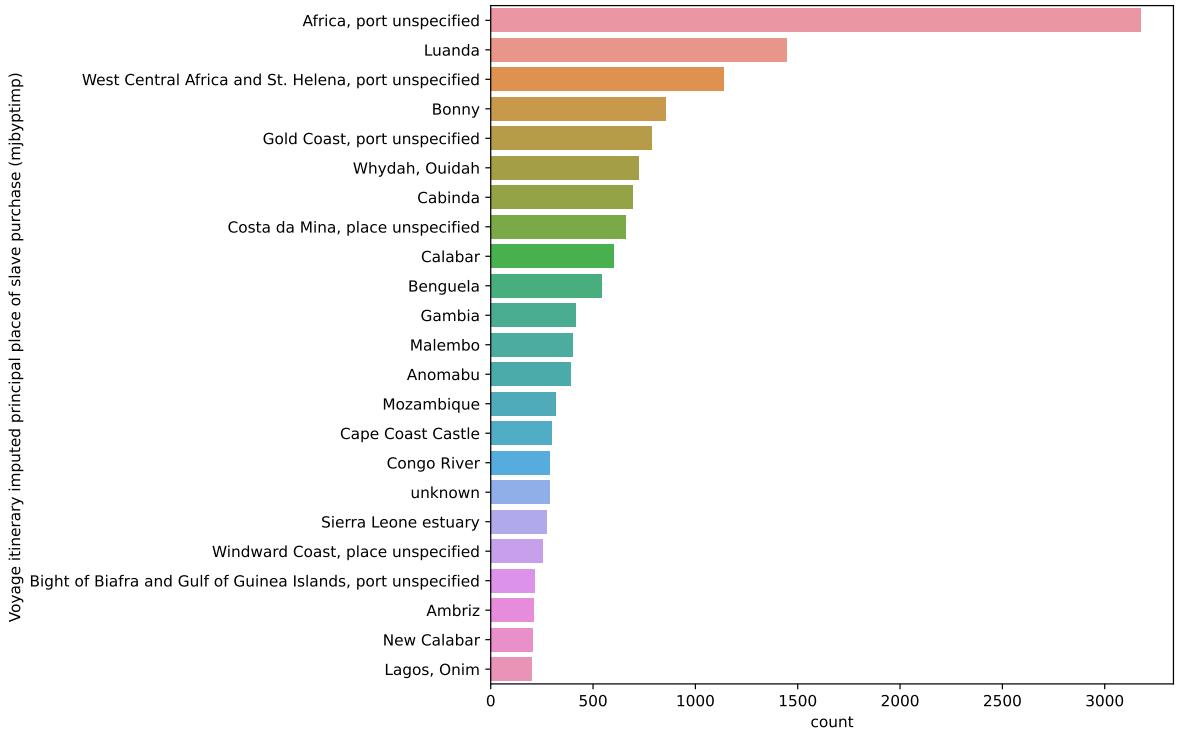

```
main_pur_port = cleaned_df.iloc[:,4]
main_pur_counts = main_pur_port.value_counts()
main_pur_counts
```

```
Voyage itinerary imputed principal place of slave purchase (mjbyptimp)
Africa, port unspecified                                    3177
Luanda                                                      1447
West Central Africa and St. Helena, port unspecified        1139
Bonny                                                        853
Gold Coast, port unspecified                                 787
                                                            ...
Petit Mesurado                                                 1
Eva                                                            1
Pokesoe (Princes Town)                                         1
Sassandra                                                      1
Sugary (Siekere)                                               1
```

```
Name: count, Length: 161, dtype: int64
```

```
fig, ax = plt.subplots(1,1,figsize=(8,8))
filter = main_pur_counts > 200
sns.barplot(ax=ax,x=main_pur_counts[filter],y=main_pur_counts.index[filter])
```

```
<Axes: xlabel='count', ylabel='Voyage itinerary imputed principal place of slave purchase (m
```



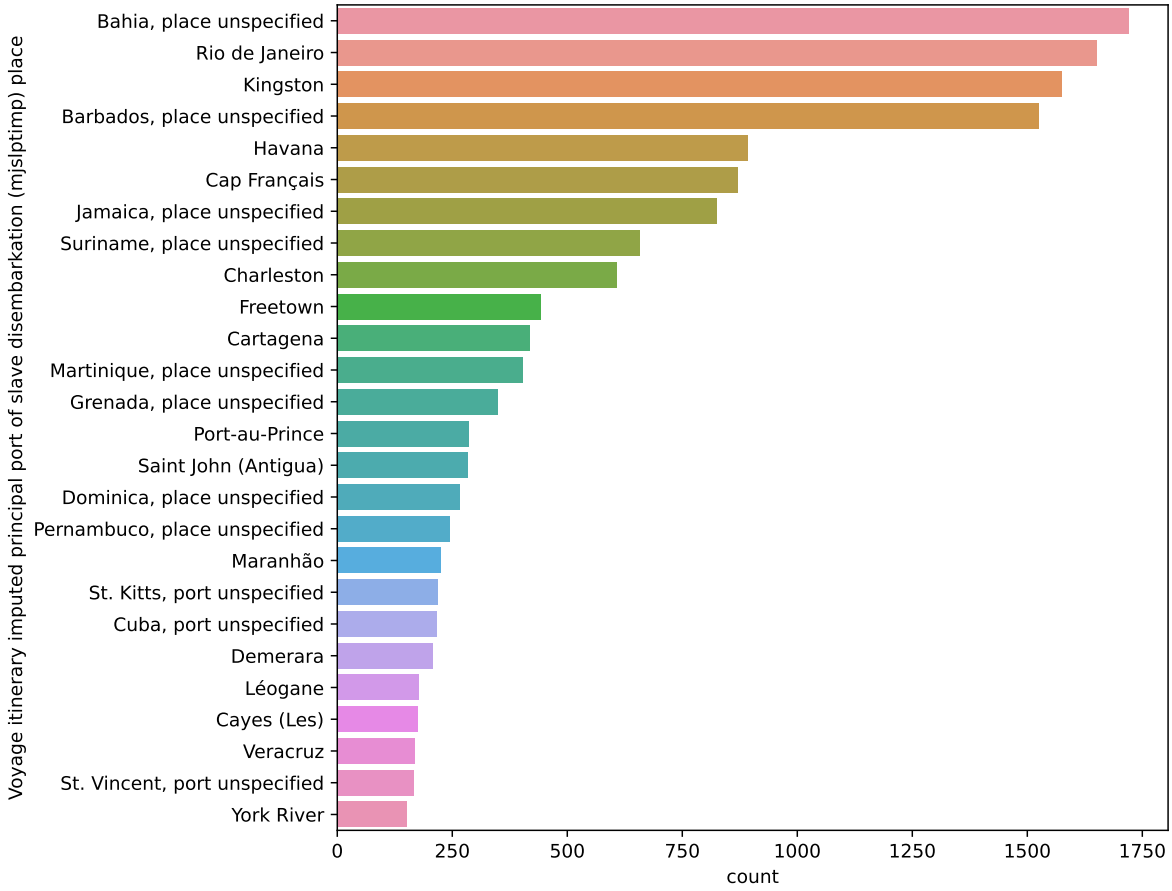

```
main_dis_port = cleaned_df.iloc[:,5]
main_dis_counts = main_dis_port.value_counts()
main_dis_counts
```

```
Voyage itinerary imputed principal port of slave disembarkation (mjslptimp) place
Bahia, place unspecified       1720
Rio de Janeiro                 1651
Kingston                       1576
Barbados, place unspecified    1524
```

```
Havana                          893
                                ...
France, place unspecified         1
Santa Marta                       1
Dois Rios                         1
Maceió                            1
Bonny                             1
Name: count, Length: 240, dtype: int64
```

```
fig, ax = plt.subplots(1,1,figsize=(8,8))
filter = main_dis_counts > 150
sns.barplot(ax=ax,x=main_dis_counts[filter],y=main_dis_counts.index[filter])
```

```
<Axes: xlabel='count', ylabel='Voyage itinerary imputed principal port of slave disembarkatio
```



```
<IPython.core.display.HTML object>
```

Let's try to make now a different type of visualization, a time series, i.e. a plot where we see how parameters change over time

```
<IPython.core.display.HTML object>
```

```
col_to_group = 'Year of arrival at port of disembarkation'
col_to_sum = 'Captives arrived at 1st port'
df_per_year = cleaned_df.groupby(col_to_group)[col_to_sum].sum()
print(df_per_year.shape)
df_per_year
```

```
(298,)

Year of arrival at port of disembarkation
1520.0       44.0
1526.0      115.0
1527.0       46.0
1532.0      589.0
1534.0      354.0
            ...
1862.0    11407.0
1863.0     6739.0
1864.0     3298.0
1865.0      795.0
1866.0      700.0
Name: Captives arrived at 1st port, Length: 298, dtype: float64
```

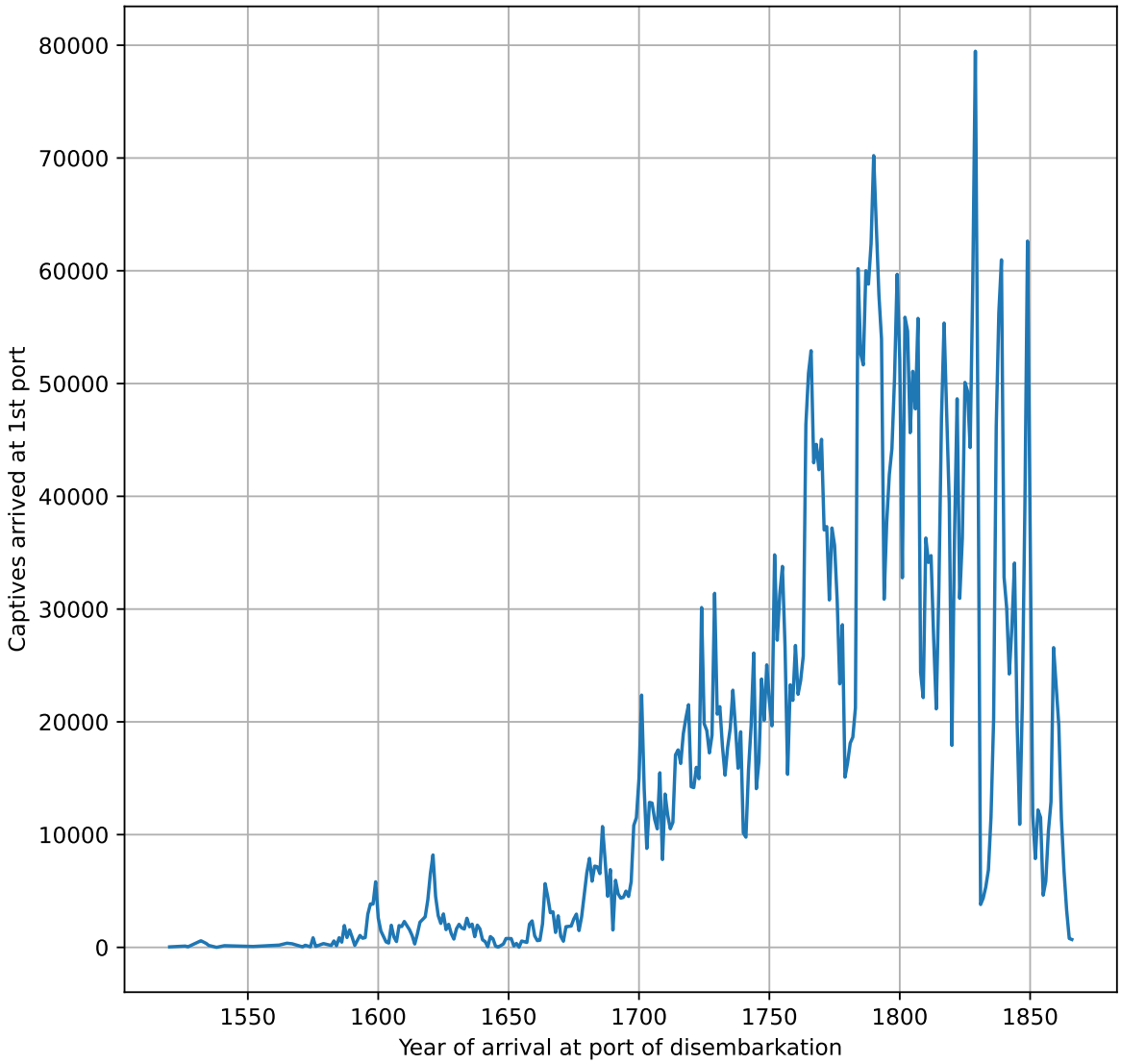
```
fig, ax = plt.subplots(1,1,figsize=(8,8))
sns.lineplot(ax=ax,x=df_per_year.index,y=df_per_year)
plt.grid()
```

```
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119: FutureWarning:

use_inf_as_na option is deprecated and will be removed in a future version. Convert inf value

/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119: FutureWarning:

use_inf_as_na option is deprecated and will be removed in a future version. Convert inf value
```




```
max_index = df_per_year.idxmax()
min_index = df_per_year.idxmin()
min_year = df_per_year[min_index]
max_year = df_per_year[max_index]

print('Min. n. of captives per year:', min_year,'on',min_index)
print('Max. n. of captives per year:', max_year,'on',max_index)
```

```
Min. n. of captives per year: 2.0 on 1538.0
```

```
Max. n. of captives per year: 79472.0 on 1829.0
```

```
solution = 'The total number of captives is almost constant up to 1650, with the exception
and 1622. The number increases steadily up to 1800 and decreases afterwords. The times ser
by low and high peek. The number of captives per year reaches its maximum on 1829 with alm
that year. The minimum is 2 captives on 1538.'
question_box(solution=solution)
```

```
<IPython.core.display.HTML object>
```

Time series are very interesting to describe the trends of phenomema at different scale. Our plot ticks are separated by 50 years, this is fine to visualise trends over centuries, but we cannot see what's happening on decades.

```
<IPython.core.display.HTML object>
```

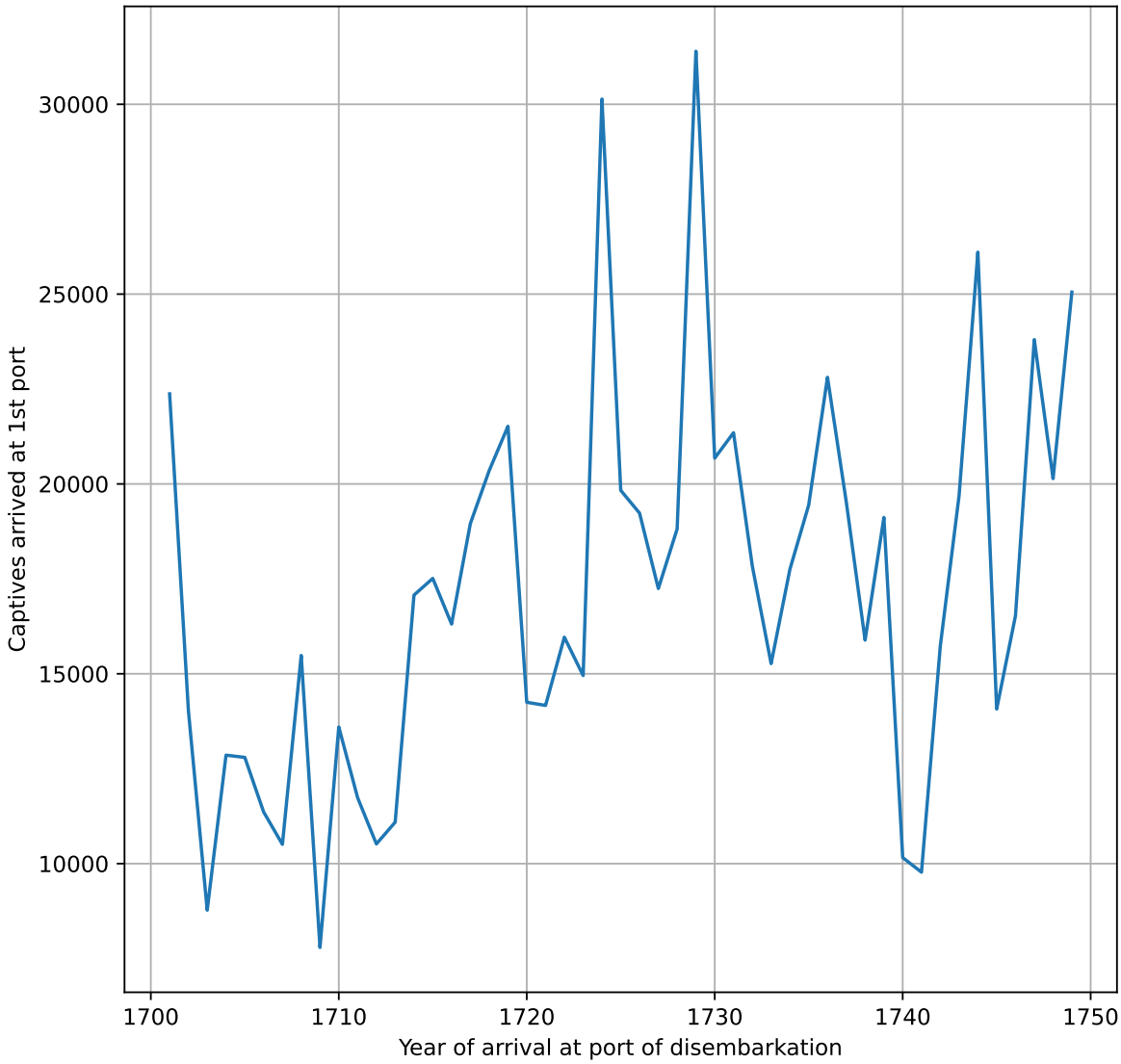
```
time_filter = (df_per_year.index > 1700) & (df_per_year.index < 1750)
fig, ax = plt.subplots(1,1,figsize=(8,8))
x_data = df_per_year.index[time_filter]
y_data = df_per_year[time_filter]
sns.lineplot(ax=ax,x=x_data,y=y_data)
plt.grid()
```

```
/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119: FutureWarning:

use_inf_as_na option is deprecated and will be removed in a future version. Convert inf value

/opt/anaconda3/lib/python3.11/site-packages/seaborn/_oldcore.py:1119: FutureWarning:

use_inf_as_na option is deprecated and will be removed in a future version. Convert inf value
```


30000
25000
20000
15000
10000
Captives arrived at 1st port
1700
1710
1720
1730
1740
1750
Year of arrival at port of disembarkation

Summary

Databases and Data Analysis

In the first part of the workshop, we conducted an interactive exercise to experience the main challenges associated with converting unstructured data into structured, organized, tabular (and generally more mathematical) data. Having structured, machine-readable data is essential for proper data analysis. It is important to note that database creation, which marks the beginning of every data analysis process, can be affected by errors, missing data, and biases driven by human decisions.

In the second part of the workshop, we focused on data analysis. Data analysis is often only considered in relation to programming tools. However, we aimed to emphasize the fundamental principles of data analysis, emphasizing that analyzing data essentially involves querying a dataset. While some information may be easy to retrieve, other information may be hidden or require assumptions and speculation. Python (or any other programming language) is simply a tool to translate our questions into a machine-readable form.

Regardless of our data analysis process and the tools we use, it is crucial to describe and document our choices so that other researchers can reproduce the entire workflow that led to our conclusions.

General data analysis workflow

1. **Define a Research question:**
    - Understand the problem or question you are trying to address;
    - Clearly define the goals, objectives, and sub-task to answer a research question.
2. **Collect/Organise Data:**
    - Collect relevant data from various sources;
    - Ensure data quality, address any missing or inconsistent data, ensure proper data structure.
3. **Clean Data:**
    - Clean and preprocess the data to handle missing values, outliers, and errors;

- Standardize or normalize data formats if necessary.

4. **Explore Data:**
    - Explore the data using statistical and visual methods;
    - Identify patterns, trends, and relationships in the data.
5. **(Model):**
    - Select appropriate models based on the analysis goals;
    - Evaluate the model's performance using metrics relevant to the analysis;
    - Fine-tune the model if necessary.
6. **Interpret Data:**
    - Interpret the results of the analysis in the context of the initial research question;
    - Draw conclusions and make recommendations based on the findings.
7. **Visualization and Reporting:**
    - Create visualizations to communicate key findings;
    - Prepare a comprehensive report summarizing the analysis process, results, and insights.

What's next?

Congratulations! If you are reading these few lines you survived our workshop on analysing cultural data (and you even went through the documentation!). Our workshop was only an introduction, it would have been impossible to cover everything related to analysing cultural data in only four hours, but we hope you now have a general overview of how data analysis is performed and of all the caveats related to data base creation and analysis.

What's next? You can build on top of what we have discussed during the workshop. Here there is a list of possible further steps, good luck!

- **More data**: try to obtain more data in .csv form and perform the data analysis on this new data. You can either get new data in the SlaveVoyages website or download any data in .csv format. Just remember to download it in the "data" directory and to change `data_file` into "data/your_file.csv". You can also use data in a different format (like excel sheet for example) and read it with the corresponding `pandas` tool;
- **Learning more about Python**: if you know nothing about programming and Python, you might consider to invest some time for learning about it. The Utrecht University Library and the Centre for Digital Humanities (CDH) offer free Python courses: have a look at the Research Data Management (RDM) workshop page and at the CDH workshop page;
- **More data questioning**: you can ask your own questions to data and find a way to implement that in Python (or any other programming language you are going to use)

Glossaries

Data Glossary

Controlled vocabularies

Standardised sets of terms or phrases used to ensure consistency and accuracy in categorising and retrieving information.

Data cleaning

The process of identifying and correcting errors, inconsistencies, and inaccuracies in a dataset to improve its quality and reliability.

Data harmonisation

The process of integrating and standardising data from different sources or formats to ensure consistency and compatibility for analysis or other purposes.

Data models

Abstract representations defining the structure, relationships, and constraints of data within a system or database.

Enrichment

The process of enhancing or augmenting existing data with additional information to improve its quality, usability, or value.

Graph database

A database structured around graph theory, where data entities are represented as nodes and their relationships as edges, facilitating complex and interconnected data querying.

ID

Identification or identifier used to uniquely distinguish an entity within a system.

Normalisation

The process of organising data in a database to reduce redundancy and dependency by dividing large tables into smaller ones and defining relationships between them.

Relational database

A type of database management system (DBMS) organised around tables and relationships, adhering to the principles of the relational model.

Programming glossary

Code

Code is like a set of instructions that tells the computer what to do. It's similar to a recipe for the computer. The instructions for a specific task may very according to the programming language you use, that is why you also usually specify the language you are using (e.g. "Python code").

csv

CSV (coma separated values) is a way to store information, like making lists. It's a simple way to organize data, like names and ages, using commas. A file containing data organised in this way, has usually the extention ".csv". Even if the word "coma" is present in the acronym, data can be also separated by other symbols such as ";" or ":" and still be contained in a .csv file.

DataFrame

A DataFrame is a Python object included in the pandas library. It is basically a table where information is organised in rows and columns. Every DataFrame row has an index that can be either a numeric value or a string (i.e. a label)

Initialisation

Initialise basically means getting things ready. It's the starting point before using something in a program. Initialising a variable, in particular, means assigning a value to it for the first time:

```
# We initialise the variable name and age for the first time with a string (a word) and a
name = 'Stefano'
age = 28
```

Library

See package

Loop

A loop is like a computer doing something over and over again. It's a way to repeat a task multiple times.

Object

An object is like a thing in the computer's world. It has characteristics and things it can do. For example, a dog can have a name and bark.

Package

A library or package is like a toolbox with ready-made tools. It's a collection of helpful code (objects and functions) that programmers can use to make their work easier. Packages are made by the programming community and are usually organised according to certain specific tasks. You may find packages specific for time series analysis, text analysis, satellite image analysis, etc. The advantage of using a package is that you do not have to spend time and energy in finding solutions to problems already tackled by other people. Packages do not usually come automatically with the basic programming language installation (so to optimize space), but need to first be downloaded and imported. You may think at packages as building tools. Downloading a package would be similar to buying them from the shop and importing them is similar to get them ready to work (you do not need all your tools ALL the time for ANY house job, right?)

Series

A Series is a DataFrame with a single column. Like DataFrames, a Series is an object belonging to the pandas library. Series rows, like in a DataFrame, have indices that can be either values or labels.

Variable

A variable is like a box where you can keep and change information. It is a container for numbers or words.

Resources

Datasets

- SlaveVoyages website;
- CSV exports of the Getty Provenance Index GitHub page;
- National Gallery of Art Open Data GitHub page.

Programming

- Python webpage;
- Pandas (Python Data Analysis package) webpage;
- Matplotlib (Python visualization package) webpage;
- Seaborn webpage (Python visualization package) webpage.