# [***1.1. Perform an Initial Data Analysis***](https://openclassrooms.com/en/courses/6037301-perform-an-initial-data-analysis)

# **Identify Different Types of Errors**

Welcome!

In this course, you will be learning how to **cleanse** and **describe** your data.

When you begin analyzing your data, you need to cleanse them of all errors. If not, the code you write to make your lovely graphs (and other representations) will crash. Worse, if your sample contains errors, your analyses may well contain errors, too.

Every data analyst or data scientist will tell you that, unfortunately, we spend more time cleansing our data than analyzing them. Cleansing your dataset is a crucial component in data analysis, even though it may not be the most exciting part. :honte:

It would be false to say that data cleansing is performed before data analysis. In most cases, we have to go back and forth between the cleansing phase and the description (analysis) phase. During the analysis phase, we often encounter new errors, and are forced to go back again and cleanse. In addition, the cleansing required for analysis differs from one statistical process to the next: that’s why you have to go back and forth!

Wait, why do we have errors in the first place?

It’s all comes down to a question of *data source*.

If you have data input by humans, it is highly likely to have errors. Imagine you are volunteering as a poll worker for a local election and are asked to gather elections results on paper and then manually type them into a spreadsheet program. At some point, some way, an error is bound to occur!

Consider another situation where data is provided by sensors (for example, a geolocation system on your cell phone, a thermometer, or speed sensors on your car). An error can occur if the sensor has deteriorated over time and is no longer or no longer works at all (is no longer sending data).

### ***1.2.Identify the Different Error Types***

We are going to look at a few different types of errors. There is no need to learn them by heart or remember their names: these details aren’t important!

Let’s take the example of a sample of people described by a number of different variables:

| **First Name** | **Email** | **Date of Birth** | **Country** | **Height** |
| --- | --- | --- | --- | --- |
| Leila | leila@example.com | 23/01/1990 | France | 1.49m |
| Samuel | samuel\_329@example.com | 20/09/2001 |  | 1.67m |
| Rodney | choupipoune@supermail.eu | 12 Sept. 1984 | Madagascar | 5'2 |
| Mark | marco23@example.com, mc23@supermail.eu | 10/02/1978 | 24 | 1.65m |
| Harry | helloworld@mail.example.com | 04/25/1975 | Germany | 1.34m |
| Hannah | hannah2019@supermail.eu | 01/01/1970 | Canada | 2.8m |
| Samuël | samuel\_329@example.com |  | Benin | 1.45m |

Well... You can see that this sample is not super clean, right?

Can you point out a few inconsistencies? Write them down a few and check your answers below!

1. First, there are empty cells for the "country" and "date of birth variables". We call these **missing attributes**.
2. If you look at the "Country" column, you see a cell that contains 24. “24” is definitely not a country! This is known as a **lexical error**.
3. Next, you may notice in the "Height" column that there is an entry with a different unit of measure. Indeed, Rodney's height is recorded in feet and inches while the rest are recorded in meters. This is an **irregularity error** because the unit of measures are not uniform.
4. Mark has two email addresses. It’s is not necessarily a problem, but if you forget about this and code an analysis program based on the assumption that each person has only one email address, your program will probably crash! This is called a **formatting error**.
5. Look at the "date of birth" variable. There is also a **formatting error** here as Rodney’s date of birth is not recorded in the same format as the others.
6. Samuel appears on two different rows. But, how can we be sure this is the same Samuel? By his email address, of course! This is called a **duplication error**. But look closer, Samuel’s two rows each give a different value for the "height variable": 1.67m and 1.45m. This is called a **contradiction error**.
7. Hannah is apparently 9'1". This height diverges greatly from the normal heights of human beings. This value is, therefore, referred to as an **outlier**.

The term **outlier** can indicate two different things: an **atypical** value and an **aberration**.

### **Deal With These Errors**

I’ll tell you right away that, when it comes to cleansing data sets, there is no set rule. Everything you do depends on how you plan to use your data. No two data analysts will cleanse the same data set the same way—not if their objectives are different!

So there’s no set rule, but I can give you a few pointers:

1. 1. Missing attributes will be addressed in the following chapter.
2. 2. For the invalid country, it’s possible to supply a list of authorized countries in advance, then eliminate all of the values that are not found on this list (hint: 24 will not be found). Such a list is often referred to as a dictionary.
3. For irregularity errors, it’s more complicated! You can, for example, set a fixed format (here: a decimal number followed by the letter “m” for “meter”) and eliminate values that don’t adhere to it. But we can do better, by first detecting what unit the value is expressed in (meters or centimeters) then converting everything to the same unit.
4. For the formatting error of the duplicate email address, it all depends on what you want to do. If you won’t be looking at emails in your future analysis, there’s no need to correct this error. If, on the other hand, you want to know the proportion of people whose address ends in, for example @example.com, or @supermail.eu, etc., then you can choose between:
   1. Taking the first email address and forgetting the second one.
   2. Keeping all email addresses.
5. Let’s move on to the Date of Birth variable. Aaaaaaah, dates! Believe me, they will always give you headaches! There are many different formats; each country has its own custom when it comes to writing dates (France and North America, for example, do not use the same format). Add to this the problem of time zones! In our case, the simplest solution would be to eliminate dates that are not in the desired format month/day/year.
6. Duplicates will be discussed in the next chapter.
7. Outliers will also be discussed in the next chapter!

Some countries have come together and agreed upon a uniform standard for dates. It’s called the ISO 8601 format, and it looks like this: 1977-04-22T06:00:00Z.

As a general rule, if a variable contains few errors and the variable is not of crucial importance to your analysis, you can allow yourself to delete erroneous values. You will then be left with missing attributes. You will see what to do with these in the next chapter.

However, if there are many errors of the same type, you might as well create a program to correct them. For example, if 60% of the heights are expressed in meters, 35% in centimeters, and 5% in other units, then 35% of the errors are of the same type (35% of the values are expressed in centimeters instead of meters). In this case, you might as well write a few lines of code to convert the centimeters to meters. If you are motivated and the data set is worth it to you, go ahead and correct the remaining 5%, although that might take you longer!

### **Take It Further: External Resources**

In this course, we will cleanse our data using Python.

But you should know that there is a very good tool for cleansing data, accessible to non-programmers, called [OpenRefine.](http://openrefine.org/)

## **1.3. Deal with Missing Attributes, Outliers, and Duplicates**

In the last chapter, we saw that a sample can contain missing values, outliers, and duplicates. What to do?

Some of the following methods delete information from your sample, so be sure to always make a copy of your sample. If you are thinking of deleting information, make a copy of the sample, then delete whatever you want: the new sample you obtain will be called a sub-sample (or subset).

### **Missing Attributes**

When the sample contains missing attributes, there is unfortunately no miracle cure! However, there are several possible approaches you can take.

For a given variable (for instance, last chapter’s example of Date of Birth), if the proportion of missing attributes is low, you can just forget about them and do nothing: leave the sample intact. You will then be working with a data set that has “holes,” like a Swiss cheese. Depending on the statistical process you plan to apply, this solution might be acceptable, or it might not.

#### **Forget a Variable**

However, if for this same variable, the proportion of missing attributes is way too high, you’d better just forget about it—provided that the variable is not too important to your analysis. This is the same as not including a column in a table, as we saw in the last chapter.

#### **Forget Individuals**

If the variable with the missing data is crucial to your analysis, it’s better to create a sub-sample, removing the individuals for whom this variable is missing. For example, if you are analyzing your bank statements by looking at the amount of money you spend/earn, the “transaction amount” variable will be crucial. If the transaction amount is unknown for some of the rows of your statement, it’s better to create a sub-sample that removes all of the offending rows.

However, this method is risky. You might find yourself with a number of individuals (a number of rows) so small that your analysis no longer meaningful. In addition, your sample might no longer be representative of the overall population. To find out why, go to the Take It Further section at the end of this chapter.

#### **Guess**

A more adventurous approach consists of filling in your holes with values you have guessed. This is pretty much the method for daredevils! :zorro: Of course, these values will not correspond to actual values, but some methods manage to create values that are not too far off. Guessing a missing attribute is referred to as Imputation.

For example, we can replace the missing attributes of the height variable with the average height of the individuals in our sample. In our example, to correct Hannah’s heiDeal with Missing Attributes, Outliers, and Duplicates

In the last chapter, we saw that a sample can contain missing values, outliers, and duplicates. What to do?

Some of the following methods delete information from your sample, so be sure to always make a copy of your sample. If you are thinking of deleting information, make a copy of the sample, then delete whatever you want: the new sample you obtain will be called a sub-sample (or subset).

**Missing Attributes**

When the sample contains missing attributes, there is unfortunately no miracle cure! However, there are several possible approaches you can take.

For a given variable (for instance, last chapter’s example of Date of Birth), if the proportion of missing attributes is low, you can just forget about them and do nothing: leave the sample intact. You will then be working with a data set that has “holes,” like a Swiss cheese. Depending on the statistical process you plan to apply, this solution might be acceptable, or it might not.

**Forget a Variable**

However, if for this same variable, the proportion of missing attributes is way too high, you’d better just forget about it—provided that the variable is not too important to your analysis. This is the same as not including a column in a table, as we saw in the last chapter.

**Forget Individuals**

If the variable with the missing data is crucial to your analysis, it’s better to create a sub-sample, removing the individuals for whom this variable is missing. For example, if you are analyzing your bank statements by looking at the amount of money you spend/earn, the “transaction amount” variable will be crucial. If the transaction amount is unknown for some of the rows of your statement, it’s better to create a sub-sample that removes all of the offending rows.

However, this method is risky. You might find yourself with a number of individuals (a number of rows) so small that your analysis no longer meaningful. In addition, your sample might no longer be representative of the overall population. To find out why, go to the Take It Further section at the end of this chapter.

**Guess**

A more adventurous approach consists of filling in your holes with values you have guessed. This is pretty much the method for daredevils! :zorro: Of course, these values will not correspond to actual values, but some methods manage to create values that are not too far off. Guessing a missing attribute is referred to as Imputation.

For example, we can replace the missing attributes of the height variable with the average height of the individuals in our sample. In our example, to correct Hannah’s height (which we assume is erroneous), we would replace it with the average height of the other individuals in the sample, which is 1.52 m. This is known as Mean Imputation.

**Guess Based on Other Variables**

But we can do better! To replace a given variable, we can look at the variables around it. A number of methods apply this principle.

Imagine a new individual named Luke, born in 1991, whose height is unknown. Rather than assign him the mean of the entire sample (1.52 m), we can assign him the average height of people who are about his age. So let’s assign him the average height of people born between 1990 and 2000, or 1.49 m. Here, we looked at the value of the date\_of\_birth variable to come up with a value for the height variable.

Other methods also deduce the value of a variable by looking at other variables. These include Hot-deck, and methods based on linear regression.

Guessing (imputing) values changes your sample, because the imputed values are false. In particular, your calculations of variances and correlations will be false. You must therefore use this method only sparingly.

In all cases, you must specify which method you used for each of the analysis results you present. It’s a question of intellectual integrity ^^.

**Outliers**

Hannah is 3.45 meters tall. You think that’s not so tall? You’re wrong. It’s very tall compared with the heights of other human beings.

But proceed cautiously, because an outlier value isn’t always necessarily false! Hannah might actually be 3.45 meters tall. Okay, that’s hard to believe—but it’s possible.

An outlier can be:

An aberration: a value that’s obviously false

An atypical value: a value that “deviates from the norm,” but is not necessarily false.

Ideally, outliers should be checked to determine whether or not they’re erroneous. For example, a thermometer in Canada in April might indicate 40°C, but this could be due to a defective temperature sensor, or it could be an actual value....(although it usually a little colder in Canada in the spring ;)).

So what should we do with outliers? If we are sure that the value is erroneous (input error or flawed sensor, for example) and we can’t find the actual value, it has to be deleted. If we are not sure whether it’s erroneous, we can choose between:

Deleting the value. We then find ourselves with a missing attribute, to which we can impute a value, as we saw previously. But imputation isn’t mandatory.

Keeping the value.

How to choose between these two options? It all depends on how you will be processing your data after that. Some methods are considered “robust,” meaning they are not destabilized by outliers. For example, we will see below that the mean is very sensitive to outliers, while the median is not. So if you want to find a mean, create a sub-sample in which you don’t include outliers. But if you also want to calculate a median, work with the original sample.

When you present your analysis, don’t hesitate to point out outliers if they are interesting. For example, “The temperature statements contain two outliers of 42°C, corresponding to two days of extreme temperatures.”

What about Duplicates?

In our example, Samuel appears twice. That’s a problem, because this duplicate compromises the analysis, in particular by falsifying the sample’s average height.

Duplicates have to be removed. However, there is no precise rule for detecting them: you alone can find them, based on the structure of your data and your knowledge of how the data were collected. But sometimes, it will be impossible.

A little example: if your sample contains an “identifier” variable, then it’s easy to detect duplicates. They are the ones with the same identifier . In our example, we can consider the email address to be an individual’s identifier. In our example, the two rows containing the email address samuel\_329@example.com constitute a duplicate.

If you are familiar with databases, you are probably familiar with the concept of keys (primary keys, candidate keys). Two individuals that have the same value for a key constitute a duplicate!

Another example: Say you are analyzing temperature records from a small town. The town has two weather stations. Station 1 operated for many years until January 15, 2019, and then was taken offline, due to age. Because this was expected, Station 2 had already been installed (in the same place) to take over for it. Station 2 began operating on January 2, 2019. Your sample is therefore made up of records from both stations. However, records made between January 2 and January 15 are duplicates, because both stations were operating at the same time. For each date in this interim period, therefore, you must delete one of the two records.

Yes but which of our two rows containing samuel\_329@example.com do we delete? Do we just pick one at random?

Cases like this call for greater attention. It’s better, in fact, to group the duplicates in one row. Of the two rows in our sample, the first informs us that Samuel was born on 20/09/2001, and the second informs us that Samuel lives in Benin (information that is missing from the first row). The real problem has to do with the height: the first row tells us that Samuel is 1.67 meters tall, while the second tells us that he’s only 1.45 meters tall. That’s a contradiction. If there is no other means of verifying Samuel’s height, we can, for example, choose to take the mean of these two values.

Take It Further: The Consequences of Removing Individuals

Imagine a sample of people represented in the same form as in the last chapter:

First Name Country Date of Birth Height

Albert France 23/09/1930 1.45m

Sophia USA 01/20/1959 1.68m

Donald USA 02/16/2002 1.65m

Ali France 16/02/2000 1.57m

Doriane Togo 17/08/1978 1.58m

You decide to delete all of the dates of birth that don’t conform to the format day/month/year, which creates missing attributes for the date of birth variable. Then you decide to delete all of the rows (all of the individuals) that have a missing date of birth. You will probably end up removing all of the people who live in the United States, because their dates are expressed in a different format from those of French-speaking countries. If you then perform an analysis on the heights, your sample will no longer be representative, because people from the United States surely have a different average height than those of other countries.

# **1.4. Cleanse Your Dataset using Python**

We are now going to cleanse the data set we saw in the previous chapter. We will illustrate this in Python.

We will begin by loading the sample [persons.csv](https://s3-eu-west-1.amazonaws.com/static.oc-static.com/prod/courses/files/Data+Analyst+EN%C2%A0Path/Courses/Perform+an+Initial+Data+Analysis/persons.csv) (which you can find here) into a variable we will call data . This variable will therefore be a dataframe.

Next, we will comb through each of the columns looking for errors, correcting them, and updating the columns accordingly. Whether you are working in Python or R, updating a column in a dataframe is performed like this:

data["name\_column"] = new\_column

Here, we want to replace the values of the name\_column column (or variable). If the

dataframe has 7 lines, then name\_column column will contain 7 values. To replace them,

new\_column has to be a list of 7 values.

### **Use the Apply and Map Methods**

We still need to know how to populate new\_column . In fact, this will be calculated from

name\_column . We need to comb through each name\_column value, verify whether it is correct or not, and correct it as needed. For this, we use the apply method. This method applies a function to each value of a dataframe column. Alternatively, we can use the map method, which is ([more or less](https://stackoverflow.com/questions/19798153/difference-between-map-applymap-and-apply-methods-in-pandas)) equivalent. It applies a check / correct function to each value:

import pandas as pd # the Pandas Libraries is imported and aliased 'pd'.

def lower\_case(value):

print('Here is the value I am processing : ', value)

return value.lower()

data = pd.DataFrame([['A',1],

['B',2],

['C',3]], columns = ['letter','position'])

new\_column = data['letter'].apply(lower\_case)

new\_column = new\_column.values

print(new\_column)

data['letter'] = new\_column

print(data)

This code is provided in the cleanse folder of the downloadable archive in the Download the data chapter, in the following separate files: R\_cleanse and python\_cleanse.

On line 7, we create our dataframe. This is a table with 2 columns ('letter' and 'position') and 3 rows. On line 3, we create a function named lower\_case , which takes as a parameter a value , displays it (line 4), converts it to lower case (line 5), then returns it. Next, we select the letter column from data , call the apply method, and specify that each of the values be sent, one at a time, to the lower\_case function (line 13).

On line 11, new\_column is “column” type (because the apply method returns a column). In the Pandas library, the exact column type is Series . To obtain the values for this column in the form of a list, we call new\_column.values (line 12). Here is what the program will display:

Here is the value I am processing : A

Here is the value I am processing : B

Here is the value I am processing : C

['a' 'b' 'c']

letter position

0 a 1

1 b 2

2 c 3

Lines 1 to 3 display what the lower\_case function is doing; line 4 displays the processing result, that is, the three lower-case letters; and the other lines display the dataframe in which the letter column has been updated to lowercase!

Line 12 converts new\_column from a Series object to a list. This line is optional, in fact, because the syntax of line 13 works just as well whether new\_column is a list or a Series object!

### **Attack!**

#### **Load the Data**

Begin by downloading the CSV file that corresponds to the example in previous chapters (provided at the beginning of the chapter), then load it using these lines of code:

# importation of libraries we will need

import pandas as pd

import numpy as np

import re

# loading and display of data

data = pd.read\_csv('persons.csv')

print(data)

#### **Process Country Names**

As you have no doubt understood, we will need one function per process. Let’s forget lower\_case , and write a function that verifies whether the countries in the Country column are correct. To do this, we need a list of valid country names:

VALID\_COUNTRIES = ['France', 'Madagascar', 'Benin', 'Germany'

, 'Canada']

def check\_country(country):

if country not in VALID\_COUNTRIES:

print(' - "{}" is not a valid country, we delete it.' \

.format(country))

return np.NaN

return country

Here, if the country in the country variable is not on the VALID\_COUNTRIES list, we display that message on lines 6 and 7. Then, we return np.NaN , which is the value used by the Numpy and Pandas libraries to indicate that a value is unknown. It is roughly equivalent to None.

More specifically, "NaN" means "Not a Number.” You will also see pd.NaT , "Not a Time" (for date columns).

Otherwise, if the country is valid, we simply return it (line 9)!

#### **Process Emails**

Now it’s the emails’ turn! The problem with this column is that there are sometimes two email addresses per row. We only want to take the first one. We will therefore create the first function:

def first(string):

parts = string.split(',')

first\_part = parts[0]

if len(parts) >= 2:

print(' - There are several parts in "{}", we are only keeping {}.'\

.format(parts,first\_part))

return first\_part

When there is more than one email per line, they are separated by commas. We therefore separate the character string of the string variable according to commas using the split method (line 2). The result is a list with as many items as there are email addresses; this list is placed in the parts variable.

Since parts contains at least one item, we place it in the first\_part variable. Then we count the number of items in parts using the len function. If there are at least two items, we display the message shown on lines 5 and 6. Finally, we return first\_part , which contains the first email!

#### **Process Heights**

Here we will have two functions: convert\_height , which will convert character strings of type 5'4 to decimal numbers, and fill\_height , which will replace the missing attributes with the average height (mean) of the sample.

def convert\_height(height):

found = re.search('\d\.\d{2}m', height)

if found is None:

print('{} is not in the right format. It will be ignored.'.format(height))

return np.NaN

else:

value = height[:-1] # the last character is removed: 'm'

return float(value)

def fill\_height(height, replacement):

if pd.isnull(height):

print('Imputation by the mean : {}'.format(replacement))

return replacement

return height

The first function is a little more elaborate. You can just blindly trust it, or you can attempt to pierce its veil of mystery by reading the *Take It Further* section at the end of this chapter. :magicien:

Let’s move on to the second function. Ah! It takes two parameters: height and replacement . The first is the height, as usual. The second is the value to be returned if there is a missing attribute. Line 11 checks to see whether the height value is missing (None, NaN or NaT). If it is, we return the replacement value (line 13). Otherwise, we return height .

#### **Apply All Functions**

Now that these functions are defined, let’s execute them! At the end of your program, add the following:

data['email'] = data['email'].apply(first)

data['country'] = data['country'].apply(check\_country)

data['height'] = [convert\_height(t) for t in data['height']]

data['height'] = [t if t<3 else np.NaN for t in data['height']]

mean\_height = data['height'].mean()

data['height'] = [fill\_height(t, mean\_height) for t in data['height']]

data['date\_of\_birth'] = pd.to\_datetime(data['date\_of\_birth'],

format='%d/%m/%Y', errors='coerce')

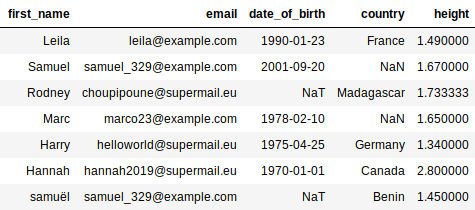
print(data)

Do you recall the syntax for updating a column we saw at the very beginning of this chapter? We use it here in lines 1 to 4, 6 and 7. You are familiar with the syntax used in lines 1 and 2. However, for lines 3, 4 and 6 you may need to refresh your memory about list comprehensions. If this term means nothing to you, scroll down to the *Take It Further* section.

What else is there left to tell you? You have all of the essentials. Well, except for a few minor details:

* h if h<3 else np.NaN returns h if h is less than 3, otherwise it returns np.NaN . This is used to delete heights in excess of 7'0", which are aberrations.
* data['height'].mean() returns a unique value, which is the mean of all the heights.
* The date\_of\_birth column contains character strings. We convert them to dates, specifying the date format. Character strings that do not conform to this format will be converted in pd.NaT (this is the case for Rodney’s date of birth).

Line 9 displays the final result:



### **Take It Further: List Comprehension**

List comprehension is a very practical syntax, because it can be used to write a loop that creates a list, in one line. For example, line 3 of the last bit of code above is equivalent to 4 lines (lines 3 to 6):

data = pd.read\_csv('persons.csv')

new\_column = []

for t in data['height']:

new\_column.append(convert\_height(t))

data['height'] = new\_column

Would could very well have used apply :

data = pd.read\_csv('persons.csv')

data['height'] = data['height'].apply(convert\_height)

### **Take It Further: Processing heights**

Let’s go back to the function we left behind:

def convert\_height(height):

found = re.search('\d\.\d{2}m', height)

if found is None:

print('{} is not in the right format. It will be ignored.'.format(height))

return np.NaN

else:

value = height[:-1] # the last character is removed: 'm'

return float(value)

Normally, Pandas automatically detects whether a column from a CSV file contains numbers or character strings. But here, the Height column contains "m"s (for “meters”). Because “m” is a letter, Pandas considers "1.34 m" to be a character string, not a number! We must therefore convert it ourselves.

So... it’s true that line 2 is hard to understand. It checks to see whether the height is properly formatted, that is, in the form of a number followed by a decimal point, then two numbers, then an "m." Thus, "1.34 m" is correct; "153 cm" is not.

This is a regular expression. No need to learn about them for this course, but you should know all the same that regular expressions are very practical—indispensable if you want to become a Data Analyst.

You now hold the keys to understanding the rest of this function. Note that float(value) is used to convert a character string representing a number to... a real number (whose type is “float”)!