Clean your data efficiently using Python

Everyone has seen messy data before, extras spaces, symbols mixed with numbers, inconsistent spelling or data formats. All of these examples make the analysis of data more difficult.

The data in *bank1* is not "bad" data, but just not ready for analyses yet!

In today's session, I want to help you show how you can check whether your data is "messy" and how to get it ready for your analyses.

Before we begin, I want to show you an example of messy data! Let's have a look at the data, open the file and try to spot some irregularities together. I want you to think of issues such as **unwanted characters, inconsistent spelling, mixed data types, extra whitespace, duplicate rows** and **missing values**. You can download the data here.

While we did the first examination with our own eyes, we can also use pandas (the python module for working with data).

Inspect your data

```
# First, let's import pandas (or install if you have not installed it yet)
import pandas

# Let's read the csv file
df = pandas.read_csv("messy_HR_data.csv")

# Look at the first five rows
df.head()
```

	Name	Age	Salary	Gender	Department	Position	Joining Date	Performance Score	Email
0	grace	25	50000	Male	HR	Manager	April 5, 2018	D	email@example.com
1	david	NaN	65000	Female	Finance	Director	2020/02/20	F	user@domain.com
2	hannah	35	SIXTY THOUSAND	Female	Sales	Director	01/15/2020	C	email@example.com
3	eve	NaN	50000	Female	IT	Manager	April 5, 2018	A	name@company.org
4	grace	NaN	NAN	Female	Finance	Manager	01/15/2020	F	name@company.org

```
# And the general information
df.info()
```

```
<class 'pandas.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 10 columns):
#   Column            Non-Null Count  Dtype
---  ------            --------------  -----
0   Name              1000 non-null   str
1   Age               841 non-null    str
2   Salary            1000 non-null   str
3   Gender            1000 non-null   str
4   Department        1000 non-null   str
5   Position          1000 non-null   str
6   Joining Date      1000 non-null   str
7   Performance Score 1000 non-null   str
8   Email             810 non-null    str
9   Phone Number      813 non-null    str
dtypes: str(10)
memory usage: 78.3 KB
```

Do you see the first problem already? Some columns do not have 1000 rows, but 841 (Age), 810 (Email) and 815 (Phone number instead). Let's remember this and fix it later.

```
# Let's have a look at the summary of the data
df.describe(include="all")
```

	Name	Age	Salary	Gender	Department	Position	Joining Date	Performance Score	Email	P
count	1000	841	1000	1000	1000	1000	1000	1000	810	
unique	10	5	6	3	5	5	5	5	3	
top	alice	thirty	65000	Male	Finance	Assistant	2020/02/20	B	user@domain.com	
freq	118	176	184	355	218	214	232	225	213	

This overview gives us another hint. Now, of course our file only has a couple of rows, but you may wonder (in real data) why a phone number is 2346 times used.

```
# Let's have a look at the unique values of each column and check whether we find
df.Name.unique()
```

```
<StringArray>
[ ' grace ',    ' david ',   ' hannah ',      ' eve ',     ' jack ', ' charlie ',
  ' frank ',    ' bob ',     ' alice ',       ' ivy ']
Length: 10, dtype: str
```

The names look normal to me, but if we look closer, we can see that there are whitespaces around the names. Is that needed? I do not think so! Let's delete them.

WAIT! Logging?

We just wanted to start deleting, however, we want to keep track of everthing. For that I want to write a small function that we can reuse.

```
import datetime

def log_change(description):
    with open("log_file.txt", mode = 'a', encoding = 'utf-8') as log_file:
        log_file.write(description, datetime.datetime.now(), sep = '\t', file = log_file)

log_change("Started process!")
```

Back to deleting whitespace!

Deleting Whitespace

```
# We can do that using str.strip()
df.Name = df.Name.str.strip()
log_change("Deleted whitespace in column Name")

df.Name.unique()
```

```
<StringArray>
['grace',   'david',   'hannah',    'eve',    'jack', 'charlie',   'frank',
  'bob',   'alice',    'ivy']
Length: 10, dtype: str
```

df.Name.str.strip() removes extra whitespace from the beginning and and of each string. We use str to say that this method should be used to every value in the column.

Since we looked at names, we may also want to have them capitalized, right?

Manipulate cases

We can do that using *str.capitalize()*. In a similar manner, we can use lowercase, uppercase or different settings:

- str.lower(): Converts all characters to lowercase.
- str.upper(): Converts all characters to uppercase.
- str.title(): Converts first character of each word to uppercase and remaining to lowercase.
- str.capitalize(): Converts first character to uppercase and remaining to lowercase.
- str.swapcase(): Converts uppercase to lowercase and lowercase to uppercase.

```
df.Name = df.Name.str.capitalize()
log_change("Capitalized strings in column Name")
```

Let's have a look at the age next...

```
df.Age.unique()
```

```
<StringArray>
['25', nan, '35', '40', 'thirty', '50']
Length: 6, dtype: str
```

It seems that age is sometimes written out as a string "thirty" and sometimes saved as a number.

Replacing values

If there are only some instances, it may be handy to just replace the string with a number.

```
df.Age = df.Age.replace("thirty", "30")
log_change("Replacing thirty with 30 in column Age")

df.Age.unique()
```

```
<StringArray>
['25', nan, '35', '40', '30', '50']
Length: 6, dtype: str
```

```
# We can also ask pandas to convert all values to numbers and everything that is n
# 'coerce' means that invalid parsing will be set as NaN.
df.Age = pandas.to_numeric(df.Age, downcast="integer", errors="coerce")
log_change("Converting Age to numeric")

df.Age.unique()
```

```
array([25., nan, 35., 40., 30., 50.])
```

Let's check whether we find a similar problem in the Salary.

```
df.Salary.unique()
```

```
<StringArray>
['50000', '65000', 'SIXTY THOUSAND', ' NAN ', '70000', '55000']
Length: 6, dtype: str
```

```
# Let's replace SIXTY THOUSAND by the number
df.Salary = df.Salary.replace("SIXTY THOUSAND", "60000")
log_change("Replace SIXTY THOUSAND with 60000 in column Salary")

# And tell pandas that we have numerics here
df.Salary = pandas.to_numeric(df.Salary, downcast="integer",errors="coerce")
log_change("Converting Salary to numeric")

df.Salary.unique()
```

```
array([50000., 65000., 60000.,    nan, 70000., 55000.])
```

Let's have a short look at Gender...

```
df.Gender.unique()
```

```
<StringArray>
['Male', 'Female', 'Other']
Length: 3, dtype: str
```

Ok! Looks good to me! But we may want to change it to "m", "f" and "d"? For that, we could create a so-called mapping. We create a dictionary that contains the words that are currently in the dataset as keys and the words that we want to use as values. With this, we can also standardize the spelling!

```
gender_mapping = {'Male': 'm', 'female': 'f', 'Other':'d'}
# Important: To use this function, you need the newest version of pandas (3.0) whi
df.Gender = df.Gender.str.replace(gender_mapping)
log_change("Replace Male, Female and Other with f,m and d in column Gender")

df.Gender
```

```
0      m
1      f
2      f
3      f
4      f
      ..
995    f
996    m
997    m
998    d
999    m
Name: Gender, Length: 1000, dtype: str
```

I did not find any problems in the Department and Position column, however, I do not like that the columns "Joining Date", "Performance Score" and "Phone Number" use two words! That may make the work with them more difficult as we have to use a different approach (df['Joining Date'] instead of df.Joining_date).

Renaming columns

Let's rename them!

```
df = df.rename(columns={"Joining Date": "Joining_Date", "Performance Score": "Perf
log_change("Renaming Column Joining Date, Performance Score and Phone Number to Jo
```

Missing values and using na

While having a first look at our data, we have seen that Age, Salary, Email and Phone Number, were not fully filled and contained empty cells. We also saw that "NaN" has not been used for non-existing values all the time. Let's unify that!

```
# First we want to replace empty cells with a NA value
df = df.replace("", pandas.NA)
log_change("Filling all empty cells with NA")

# We can also decide to name our "NA" different, but NaN works best with pandas. L
df = df.fillna(pandas.NA)
log_change("Unify NAs to pandas NA")

df
```

	Name	Age	Salary	Gender	Department	Position	Joining_Date	Performance_Score	Email
0	Grace	25.0	50000.0	m	HR	Manager	April 5, 2018	D	email@exam
1	David	NaN	65000.0	f	Finance	Director	2020/02/20	F	user@domai
2	Hannah	35.0	60000.0	f	Sales	Director	01/15/2020	C	email@exam
3	Eve	NaN	50000.0	f	IT	Manager	April 5, 2018	A	name@comp
4	Grace	NaN	NaN	f	Finance	Manager	01/15/2020	F	name@comp
...	...	...	...	...	...	...	...	...	...
995	Jack	50.0	65000.0	f	HR	Manager	2020/02/20	F	NaN
996	Jack	30.0	50000.0	m	Finance	Analyst	April 5, 2018	E	NaN
997	Hannah	30.0	70000.0	m	IT	Assistant	01/15/2020	D	user@domai
998	Bob	25.0	65000.0	d	Marketing	Manager	April 5, 2018	D	email@exam
999	Ivy	30.0	60000.0	m	Finance	Manager	2020/02/20	C	user@domai

1000 rows × 10 columns

Unifying time stamps

You may have guessed it already, but the last aspect that we want to clean are the different time stamps. The dataset contains formats such as month, day year and yearmonthday as well as monthdayyear.

Let's unify that using pandas. A possible approach would be df.Joining_Date = pandas.to_datetime(df.Joining_Date, errors="coerce"). However, since our data is a bit more complex, we may have to adapt to different formats.

```
def parse_date(date):
    """Function that takes the date and time and tries out to parse it using every
    for format in ["%B %d, %Y", "%Y/%m/%d","%m/%d/%Y", "%m-%d-%Y", "%Y.%m.%d"]:
        try:
            return pandas.to_datetime(date, format=format)
        except:
            pass
    return pandas.NaT

# Call the function
df.Joining_Date = df.Joining_Date.apply(parse_date)
log_change("Unifying time stamps in column Joining_Date")
```

```
# Unify it to a human-readable version
df.Joining_Date = df.Joining_Date.dt.strftime("%Y-%m-%d")
log_change("Converting time stamps to %Y-%m-%d in column Joining_Date")

# Let's have a final look at our data set! It looks so much cleaner!
df
```

	Name	Age	Salary	Gender	Department	Position	Joining_Date	Performance_Score	Email
0	Grace	25.0	50000.0	m	HR	Manager	2018-04-05	D	email@exam
1	David	NaN	65000.0	f	Finance	Director	2020-02-20	F	user@domai
2	Hannah	35.0	60000.0	f	Sales	Director	2020-01-15	C	email@exam
3	Eve	NaN	50000.0	f	IT	Manager	2018-04-05	A	name@comp
4	Grace	NaN	NaN	f	Finance	Manager	2020-01-15	F	name@comp
...	...	...	...	...	...	...	...	...	...
995	Jack	50.0	65000.0	f	HR	Manager	2020-02-20	F	NaN
996	Jack	30.0	50000.0	m	Finance	Analyst	2018-04-05	E	NaN
997	Hannah	30.0	70000.0	m	IT	Assistant	2020-01-15	D	user@domai
998	Bob	25.0	65000.0	d	Marketing	Manager	2018-04-05	D	email@exam
999	Ivy	30.0	60000.0	m	Finance	Manager	2020-02-20	C	user@domai

1000 rows × 10 columns

```
# Now we can start doing some statistics!
df.describe(include="all")
```

	Name	Age	Salary	Gender	Department	Position	Joining_Date	Performance_Sco
count	1000	841.000000	833.000000	1000	1000	1000	1000	
unique	10	NaN	NaN	3	5	5	5	
top	hannah	NaN	NaN	m	Finance	Assistant	2020-02-20	B
freq	118	NaN	NaN	355	218	214	232	225
mean	NaN	35.802616	62211.086435	NaN	NaN	NaN	NaN	NaN
std	NaN	8.551889	7127.033911	NaN	NaN	NaN	NaN	NaN
min	NaN	25.000000	50000.000000	NaN	NaN	NaN	NaN	NaN
25%	NaN	30.000000	55000.000000	NaN	NaN	NaN	NaN	NaN
50%	NaN	35.000000	60000.000000	NaN	NaN	NaN	NaN	NaN
75%	NaN	40.000000	65000.000000	NaN	NaN	NaN	NaN	NaN
max	NaN	50.000000	70000.000000	NaN	NaN	NaN	NaN	NaN