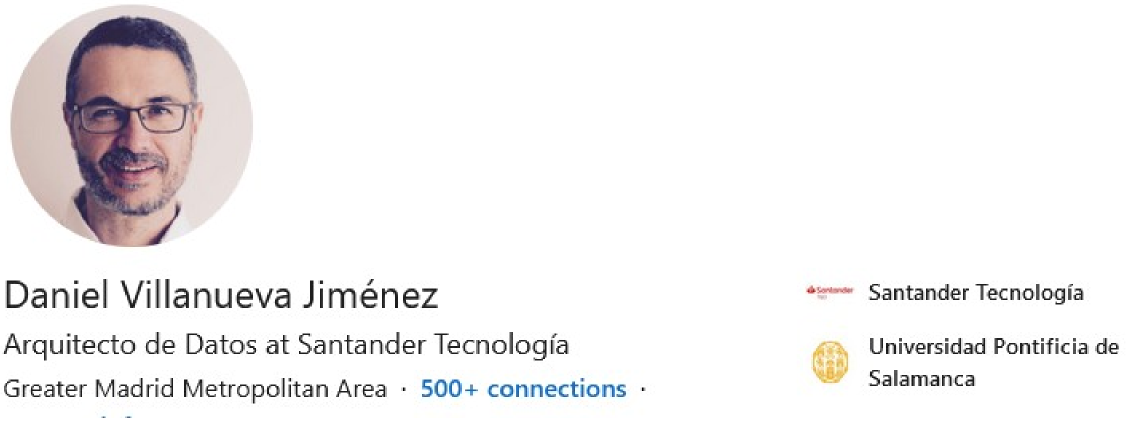
Programming and frameworks for ML

Data Cleaning with Python

About Me

Big Data Consultant at Santander / Big Data Lecturer

- More than 20 years of experience in different environments, technologies, customers, countries ...
- Passionate about data and technology
- Enthusiastic about Big Data world and NoSQL



Daniel Villanueva Jiménez

Arquitecto de Datos at Santander Tecnología

Greater Madrid Metropolitan Area · 500+ connections ·

Santander Tecnología

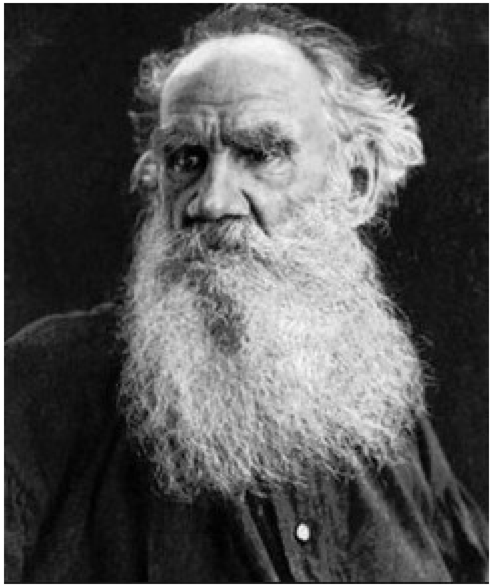
Universidad Pontificia de Salamanca

Agenda

- Introduction
- Widening tables
- Narrowing down tables
- Separating columns
- Joining columns
- Missing data
- Dropping duplicates
- Data Types
- Data Formating
- Regex

Clean data

Happy families are all alike; every unhappy family is unhappy in its own way.



León Tolstói

Clean data

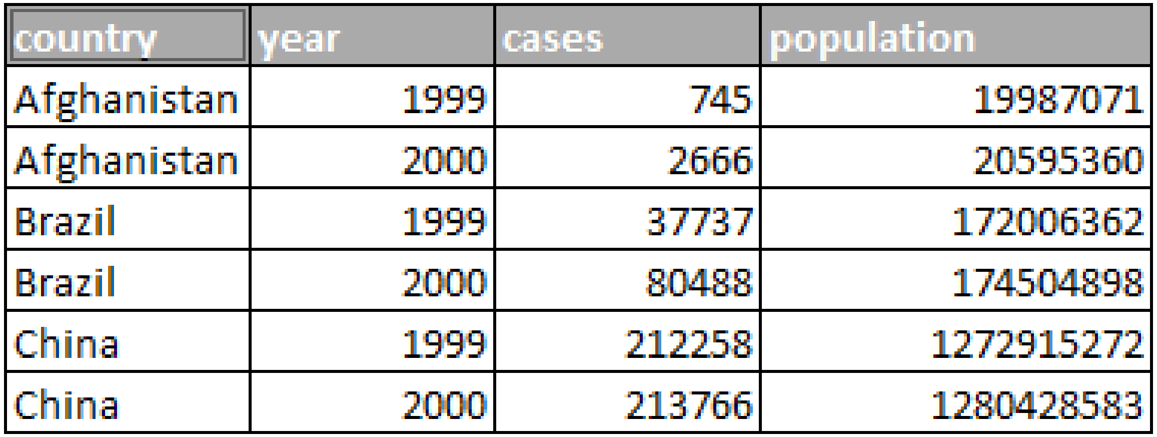
- A clean dataset is easy to analyze, model or visualize

Tidy datasets are all alike, but every messy dataset is messy in its own way.

Hadley Wickham

Definition

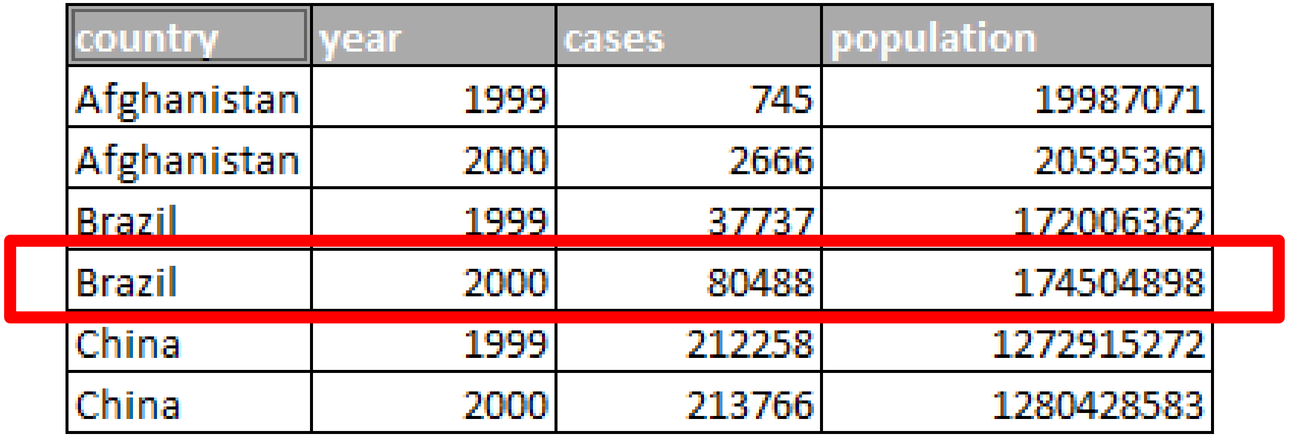
- A **unit of analysis** represents the entity being analysed in a study, and which contains similar features



country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

Definition

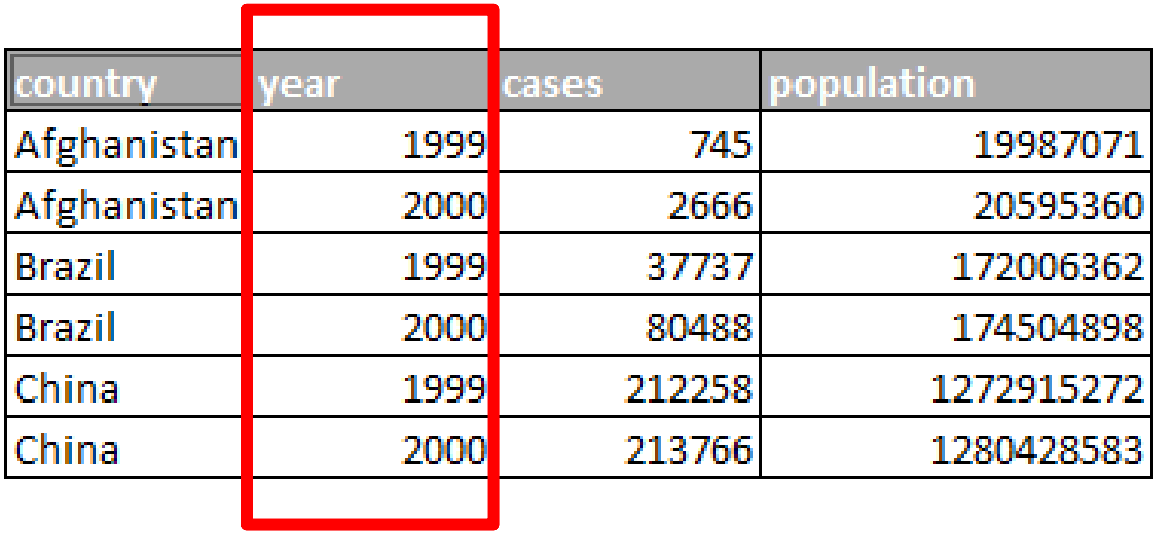
- An **observation** is data collected by observing behavior, events, or physical features.



country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

Definition

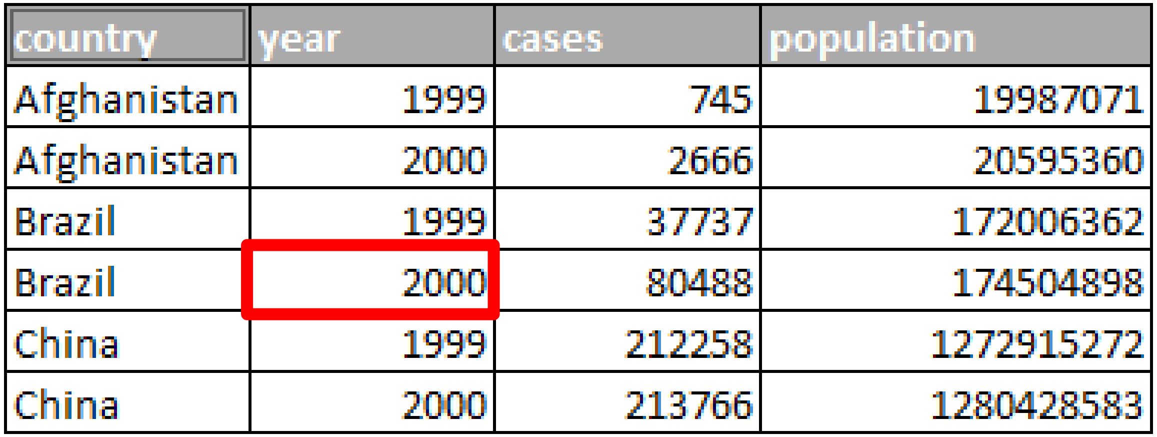
- A **variable** is a property or feature that can change depending on certain factors (the person, the weather, the country, etc.)



country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

Definition

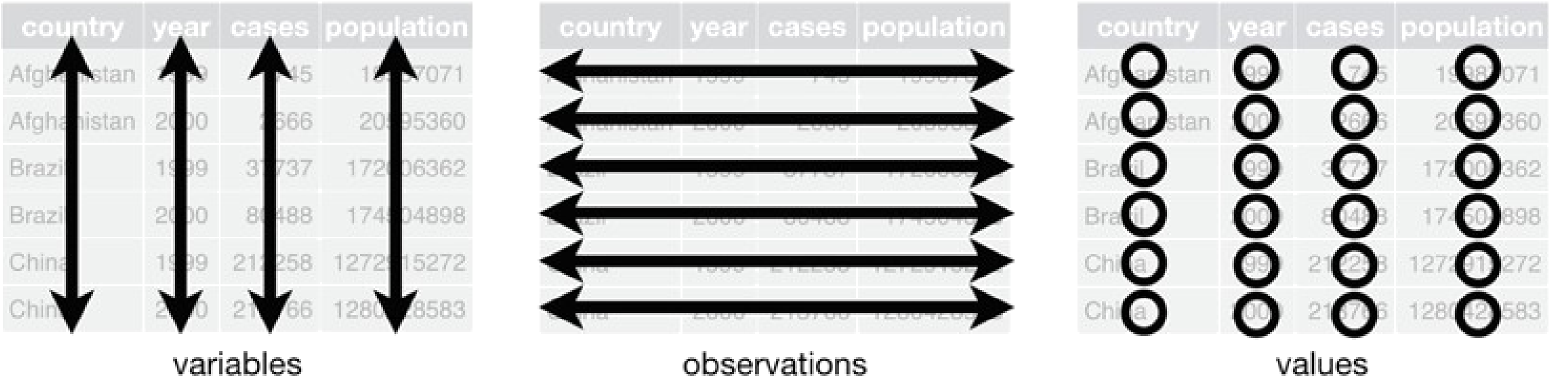
- A variable can take different **values,** which can be measured or observed.



country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

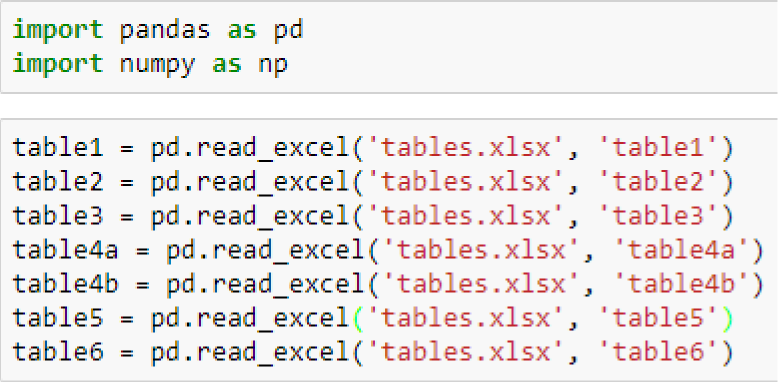
Rules

- Each **variable** must be in its own column and has to have the correct type
- Each **observation** should be in its own row (and cannot be duplicated or empty)
- Each **value** must have its own cell and has to have the correct format
- Each **unit of analysis** must be in its own table



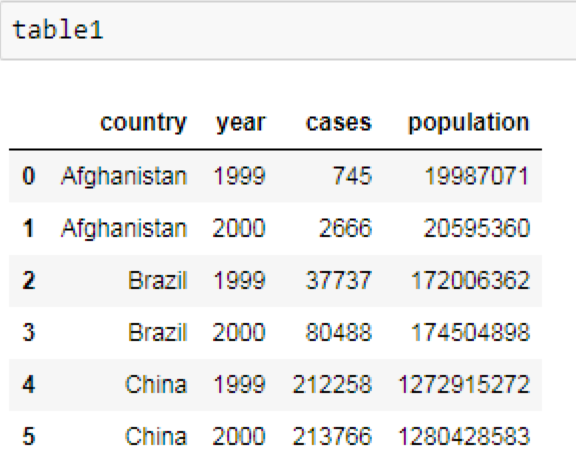
Formats of a dataset

- We will display the same dataset in several formats



```
import pandas as pd
import numpy as np
```

```
table1 = pd.read_excel('tables.xlsx', 'table1')
table2 = pd.read_excel('tables.xlsx', 'table2')
table3 = pd.read_excel('tables.xlsx', 'table3')
table4a = pd.read_excel('tables.xlsx', 'table4a')
table4b = pd.read_excel('tables.xlsx', 'table4b')
table5 = pd.read_excel('tables.xlsx', 'table5')
table6 = pd.read_excel('tables.xlsx', 'table6')
```

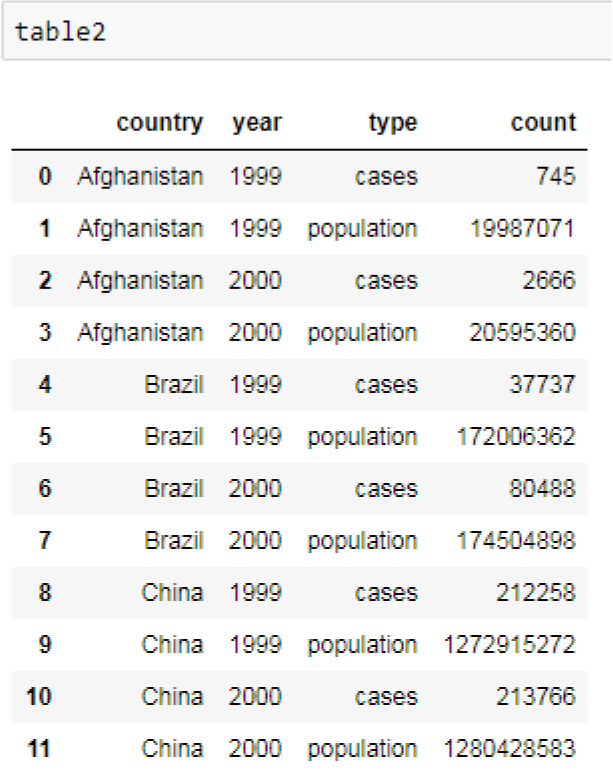


```
table1
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

Formats of a dataset

- Variables such as values ...

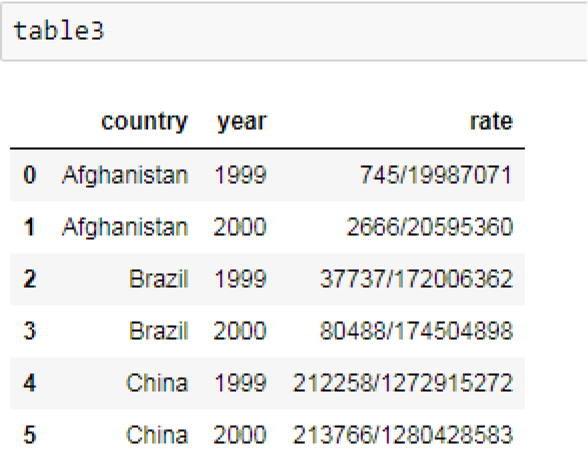


```
table2
```

	country	year	type	count
0	Afghanistan	1999	cases	745
1	Afghanistan	1999	population	19987071
2	Afghanistan	2000	cases	2666
3	Afghanistan	2000	population	20595360
4	Brazil	1999	cases	37737
5	Brazil	1999	population	172006362
6	Brazil	2000	cases	80488
7	Brazil	2000	population	174504898
8	China	1999	cases	212258
9	China	1999	population	1272915272
10	China	2000	cases	213766
11	China	2000	population	1280428583

Formats of a dataset

- A single column with several features ...

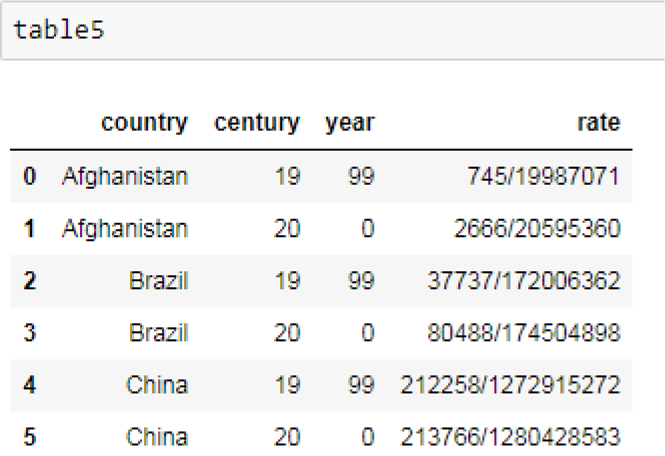


```
table3
```

	country	year	rate
0	Afghanistan	1999	745/19987071
1	Afghanistan	2000	2666/20595360
2	Brazil	1999	37737/172006362
3	Brazil	2000	80488/174504898
4	China	1999	212258/1272915272
5	China	2000	213766/1280428583

Formats of a dataset

- A feature separated into several columns...

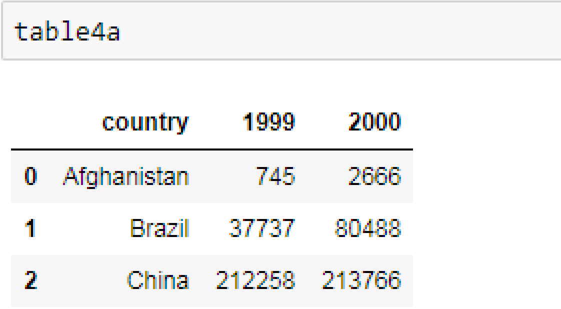


```
table5
```

	country	century	year	rate
0	Afghanistan	19	99	745/19987071
1	Afghanistan	20	0	2666/20595360
2	Brazil	19	99	37737/172006362
3	Brazil	20	0	80488/174504898
4	China	19	99	212258/1272915272
5	China	20	0	213766/1280428583

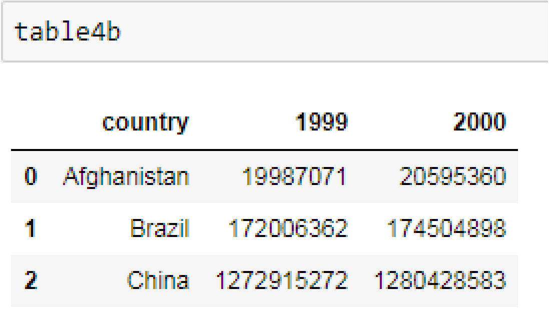
Formats of a dataset

- A separate unit of analysis in several tables
- Values in columns instead of cells ...



```
table4a
```

	country	1999	2000
0	Afghanistan	745	2666
1	Brazil	37737	80488
2	China	212258	213766

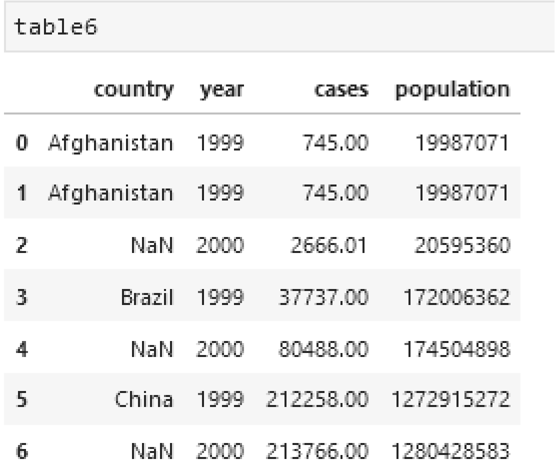


```
table4b
```

	country	1999	2000
0	Afghanistan	19987071	20595360
1	Brazil	172006362	174504898
2	China	1272915272	1280428583

Formats of a dataset

- Features with empty values, duplicated and incorrect format …



```
table6
```

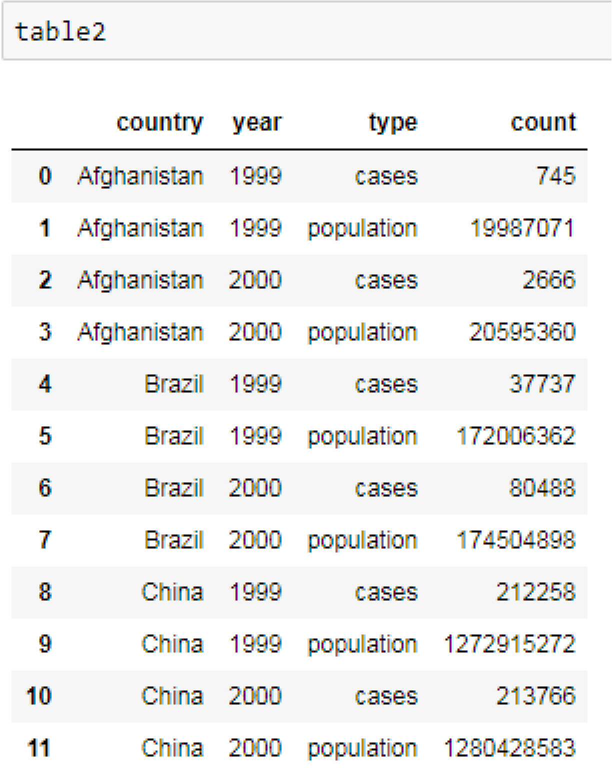
	country	year	cases	population
0	Afghanistan	1999	745.00	19987071
1	Afghanistan	1999	745.00	19987071
2	NaN	2000	2666.01	20595360
3	Brazil	1999	37737.00	172006362
4	NaN	2000	80488.00	174504898
5	China	1999	212258.00	1272915272
6	NaN	2000	213766.00	1280428583

Agenda

- Introduction
- Widening tables
- Narrowing down tables
- Separating columns
- Joining columns
- Missing data
- Dropping duplicates
- Data Types
- Data Formating
- Regex

Widening tables

- Let’s fix the ‘variable as values’ problem …

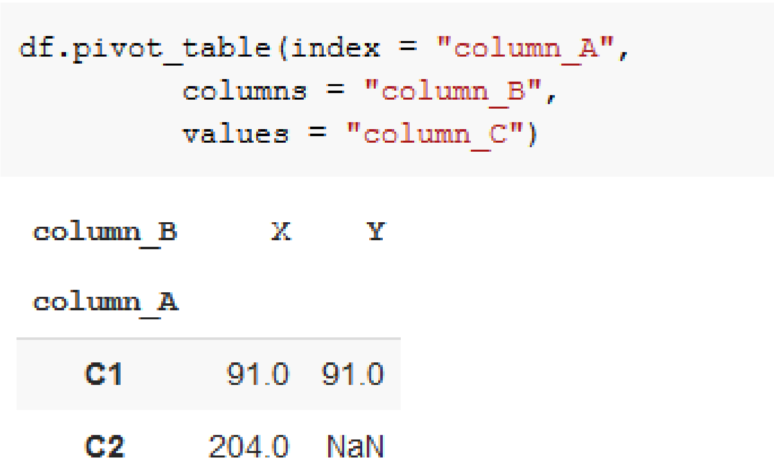


```
table2
```

	country	year	type	count
0	Afghanistan	1999	cases	745
1	Afghanistan	1999	population	19987071
2	Afghanistan	2000	cases	2666
3	Afghanistan	2000	population	20595360
4	Brazil	1999	cases	37737
5	Brazil	1999	population	172006362
6	Brazil	2000	cases	80488
7	Brazil	2000	population	174504898
8	China	1999	cases	212258
9	China	1999	population	1272915272
10	China	2000	cases	213766
11	China	2000	population	1280428583

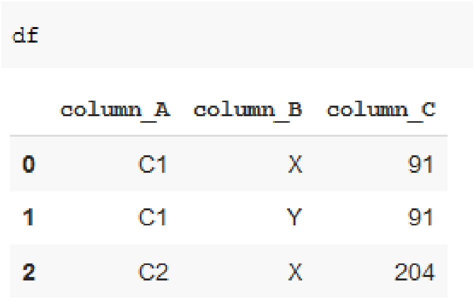
Widening tables

- The **pivot_table**() function is used to distribute a key/value pair across the columns of the table



```
df.pivot_table(index = "column_A",
         columns = "column_B",
         values = "column_C")
```

column_B	X	Y
column_A		
C1	91.0	91.0
C2	204.0	NaN

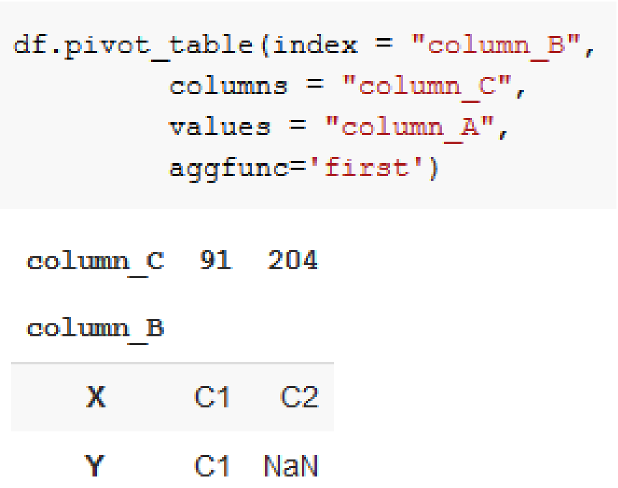


```
df
```

	column_A	column_B	column_C
0	C1	X	91
1	C1	Y	91
2	C2	X	204

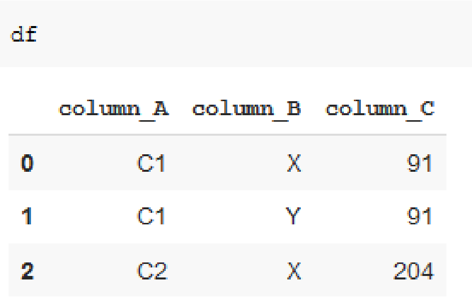
Widening tables

- We have to use the **first** aggregation function if the values are not numbers …



```
df.pivot_table(index = "column_B",
         columns = "column_C",
         values = "column_A",
         aggfunc='first')
```

column_C	91	204
column_B		
X	C1	C2
Y	C1	NaN

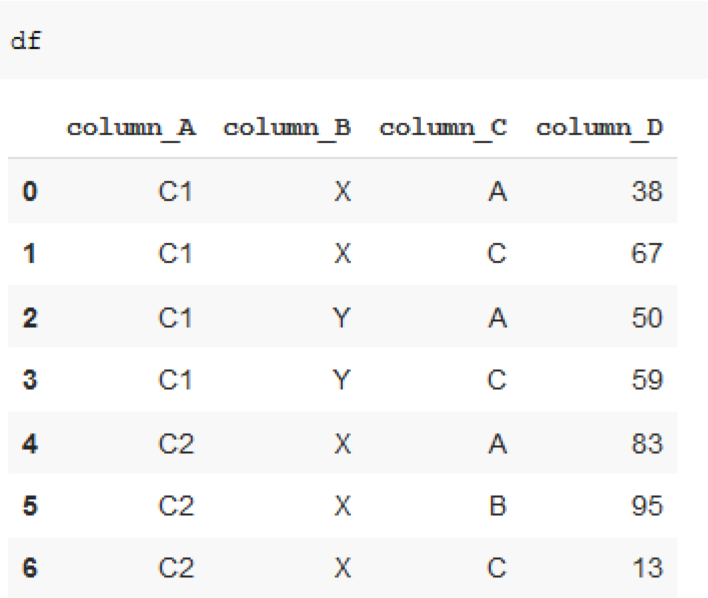


```
df
```

	column_A	column_B	column_C
0	C1	X	91
1	C1	Y	91
2	C2	X	204

Widening tables

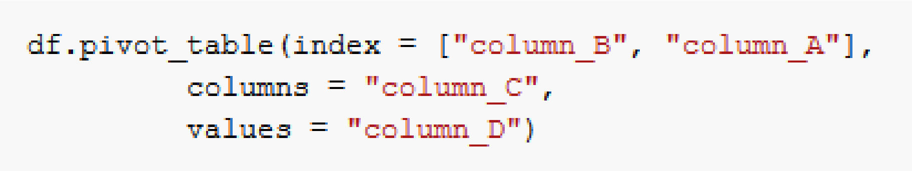
- In the case of having a DataFrame with more than 3 columns ...



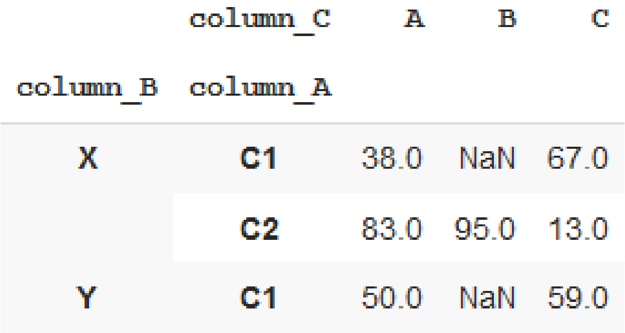
```
df
```

	column_A	column_B	column_C	column_D
0	C1	X	A	38
1	C1	X	C	67
2	C1	Y	A	50
3	C1	Y	C	59
4	C2	X	A	83
5	C2	X	B	95
6	C2	X	C	13

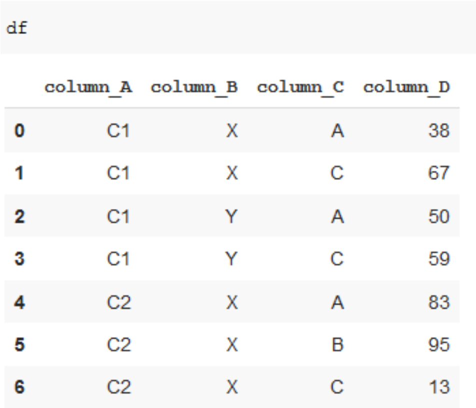
Widening tables



```
df.pivot_table(index = ["column_B", "column_A"],
         columns = "column_C",
         values = "column_D")
```



	column_C	A	B	C
column_B	column_A			
X	C1	38.0	NaN	67.0
	C2	83.0	95.0	13.0
Y	C1	50.0	NaN	59.0

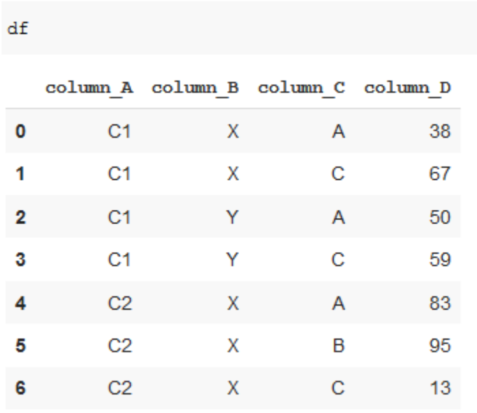


```
df
```

	column_A	column_B	column_C	column_D
0	C1	X	A	38
1	C1	X	C	67
2	C1	Y	A	50
3	C1	Y	C	59
4	C2	X	A	83
5	C2	X	B	95
6	C2	X	C	13

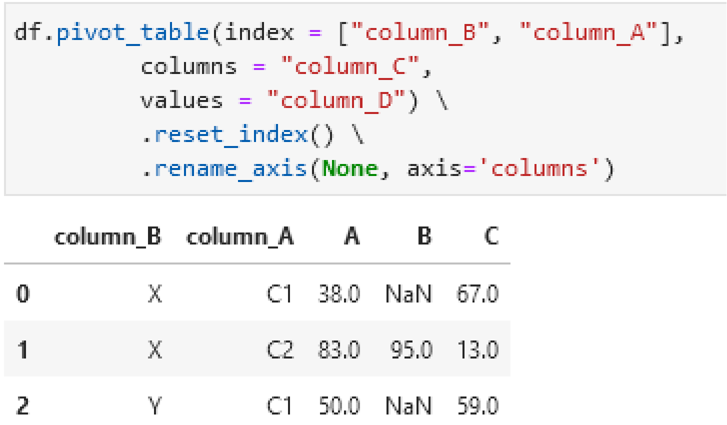
Converting Row names into Columns

- A cleaned dataframe have all variables as columns
- We can reset the index after **df.pivot_table**() is applied using the **reset_index**() and **rename_axis**() functions



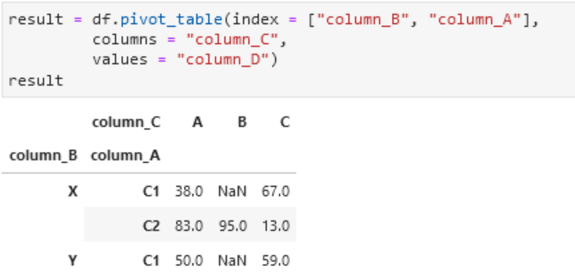
```
df
```

	column_A	column_B	column_C	column_D
0	C1	X	A	38
1	C1	X	C	67
2	C1	Y	A	50
3	C1	Y	C	59
4	C2	X	A	83
5	C2	X	B	95
6	C2	X	C	13



```
df.pivot_table(index = ["column_B", "column_A"],
         columns = "column_C",
         values = "column_D") \
         .reset_index() \
         .rename_axis(None, axis='columns')
```

	column_B	column_A	A	B	C
0	X	C1	38.0	NaN	67.0
1	X	C2	83.0	95.0	13.0
2	Y	C1	50.0	NaN	59.0

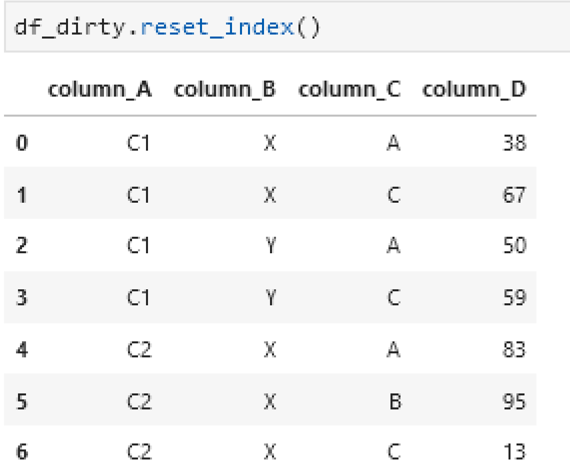


```
result = df.pivot_table(index = ["column_B", "column_A"],
         columns = "column_C",
         values = "column_D")
result
```

	column_C	A	B	C
column_B	**column_A**			
X	**C1**	38.0	NaN	67.0
	C2	83.0	95.0	13.0
Y	**C1**	50.0	NaN	59.0

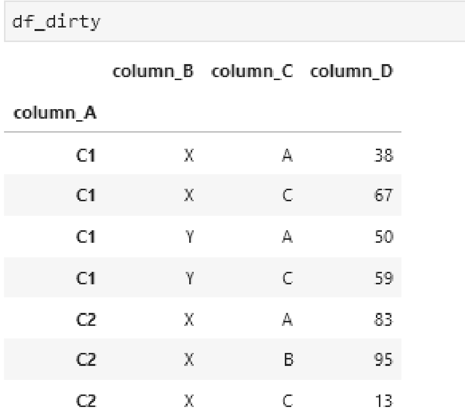
Converting Row names into Columns

- This procedure aplies in case that we have a dataset with variables as row indexes
- In this case only **reset_index**() function is neeed



```
df_dirty.reset_index()
```

	column_A	column_B	column_C	column_D
0	C1	X	A	38
1	C1	X	C	67
2	C1	Y	A	50
3	C1	Y	C	59
4	C2	X	A	83
5	C2	X	B	95
6	C2	X	C	13

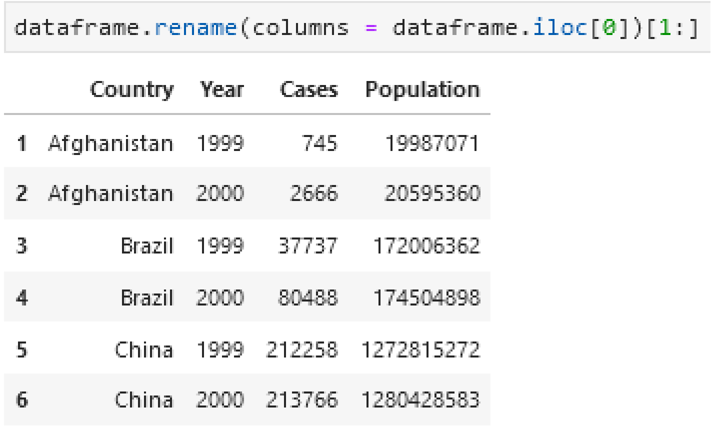


```
df_dirty
```

	column_B	column_C	column_D
column_A			
C1	X	A	38
C1	X	C	67
C1	Y	A	50
C1	Y	C	59
C2	X	A	83
C2	X	B	95
C2	X	C	13

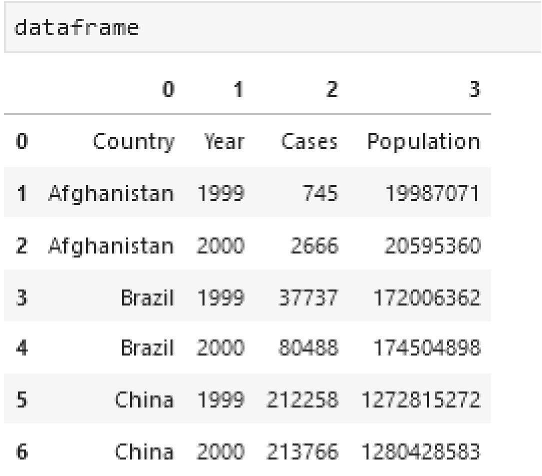
Values as Variables – Especial Case

- A special case is when we find that in the first row of the dataset are our variables
- Pandas does not have a specific function to perform this task. First we have to rename the columns and then delete the row from the dataset



```
dataframe.rename(columns = dataframe.iloc[0])[1:]
```

	Country	Year	Cases	Population
1	Afghanistan	1999	745	19987071
2	Afghanistan	2000	2666	20595360
3	Brazil	1999	37737	172006362
4	Brazil	2000	80488	174504898
5	China	1999	212258	1272815272
6	China	2000	213766	1280428583



```
dataframe
```

	0	1	2	3
0	Country	Year	Cases	Population
1	Afghanistan	1999	745	19987071
2	Afghanistan	2000	2666	20595360
3	Brazil	1999	37737	172006362
4	Brazil	2000	80488	174504898
5	China	1999	212258	1272815272
6	China	2000	213766	1280428583

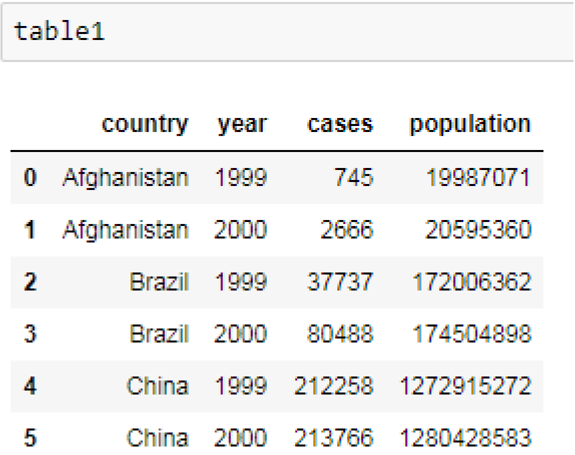
Exercise 1 (1/2)

- Load the following tables from the 'tables.xlsx' file

```
import pandas as pd

table1 = pd.read_excel('Tables.xlsx', 'table1')
table2 = pd.read_excel('Tables.xlsx', 'table2')
table3 = pd.read_excel('Tables.xlsx', 'table3')
table4a = pd.read_excel('Tables.xlsx', 'table4a')
table4b = pd.read_excel('Tables.xlsx', 'table4b')
table5 = pd.read_excel('Tables.xlsx', 'table5')
table6 = pd.read_excel('Tables.xlsx', 'table6')
table7= pd.read_excel('Tables.xlsx', 'table7')
table8 = pd.read_excel('Tables.xlsx', 'table8')
table9 = pd.read_excel('Tables.xlsx', 'table9')
```

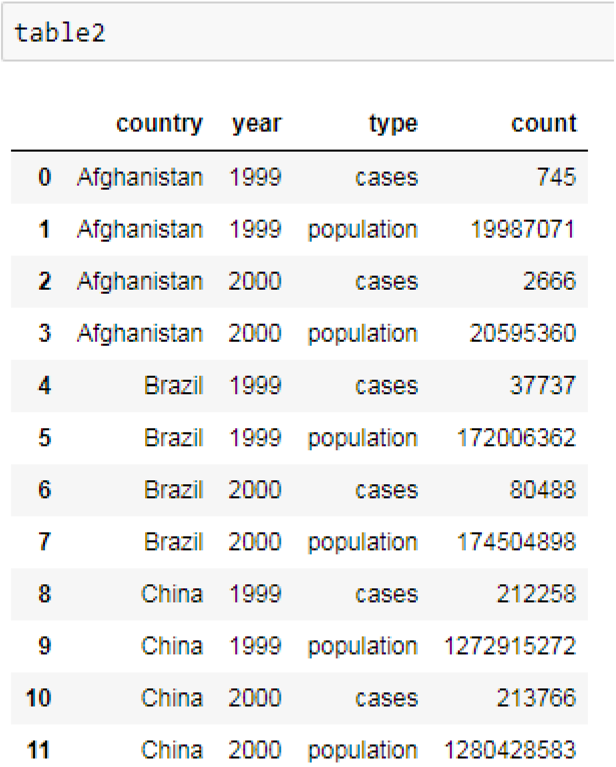
Exercise 1 (2/2)

- Converts the dataset "table2" into a clean dataset, as seen in "table1"



```
table1
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

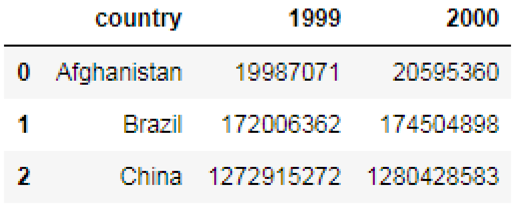


```
table2
```

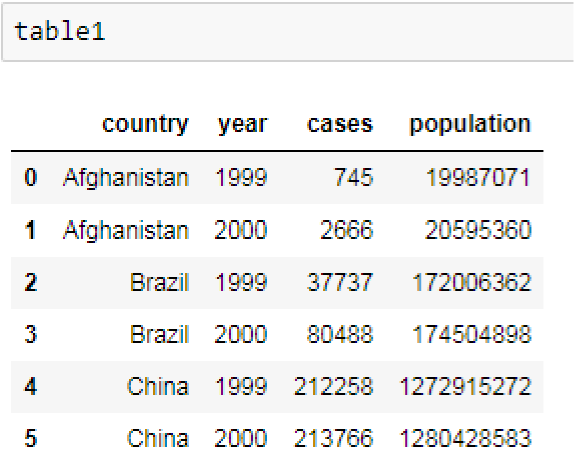
	country	year	type	count
0	Afghanistan	1999	cases	745
1	Afghanistan	1999	population	19987071
2	Afghanistan	2000	cases	2666
3	Afghanistan	2000	population	20595360
4	Brazil	1999	cases	37737
5	Brazil	1999	population	172006362
6	Brazil	2000	cases	80488
7	Brazil	2000	population	174504898
8	China	1999	cases	212258
9	China	1999	population	1272915272
10	China	2000	cases	213766
11	China	2000	population	1280428583

Exercise 2

- Convert the dataset "table1" into another one showing the evolution of the population by years



	country	1999	2000
0	Afghanistan	19987071	20595360
1	Brazil	172006362	174504898
2	China	1272915272	1280428583



```
table1
```

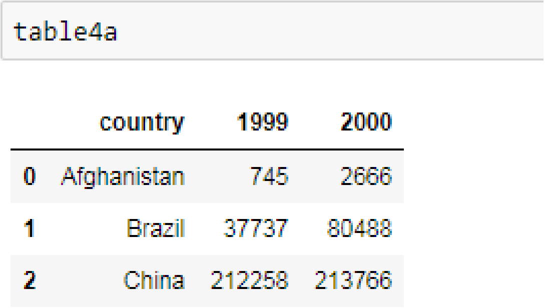
	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

Agenda

- Introduction
- Widening tables
- Narrowing down tables
- Separating columns
- Joining columns
- Missing data
- Dropping duplicates
- Data Types
- Data Formating
- Regex

Narrowing down tables

- Let’s fix the ‘Value as Column’ problem …

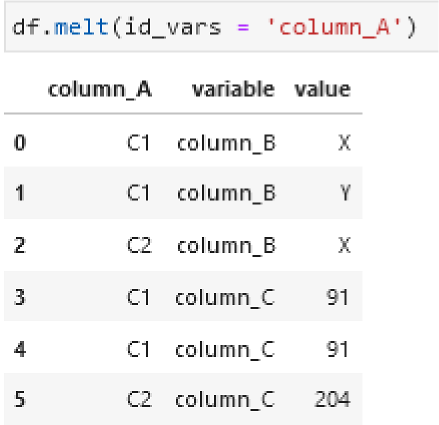


```
table4a
```

	country	1999	2000
0	Afghanistan	745	2666
1	Brazil	37737	80488
2	China	212258	213766

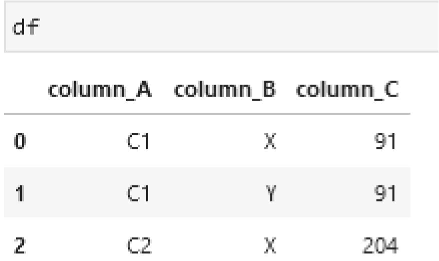
Narrowing down tables

- The **melt**() function takes multiple columns and collects them into a key/value pair



```
df.melt(id_vars = 'column_A')
```

	column_A	variable	value
0	C1	column_B	X
1	C1	column_B	Y
2	C2	column_B	X
3	C1	column_C	91
4	C1	column_C	91
5	C2	column_C	204

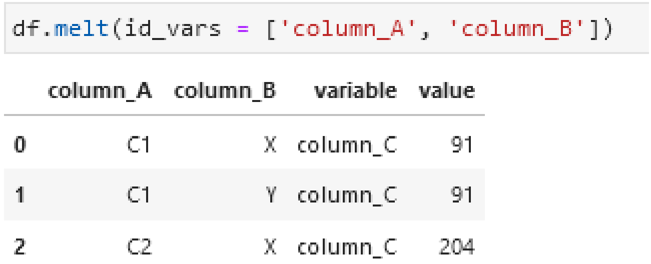


```
df
```

	column_A	column_B	column_C
0	C1	X	91
1	C1	Y	91
2	C2	X	204

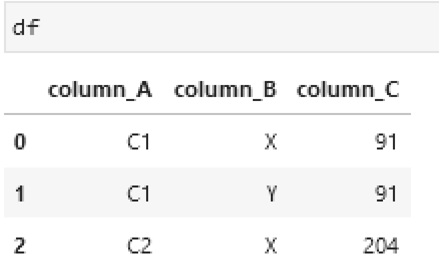
Narrowing down tables

- We can ‘reserve’ as much columns as we want



```
df.melt(id_vars = ['column_A', 'column_B'])
```

	column_A	column_B	variable	value
0	C1	X	column_C	91
1	C1	Y	column_C	91
2	C2	X	column_C	204

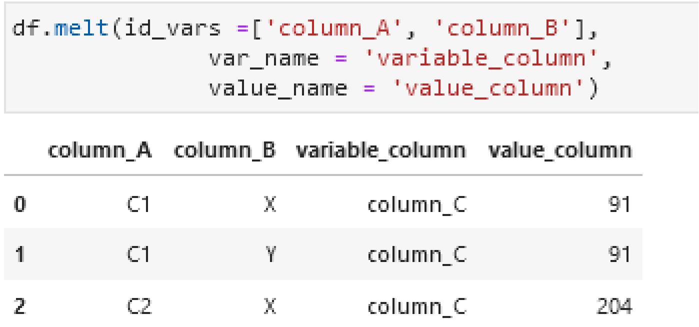


```
df
```

	column_A	column_B	column_C
0	C1	X	91
1	C1	Y	91
2	C2	X	204

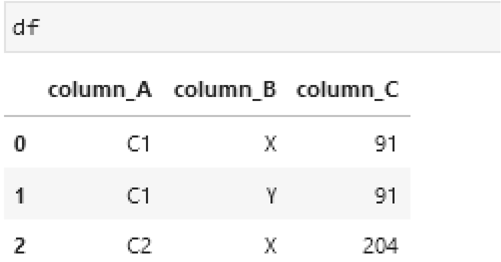
Narrowing down tables

- We can also specify the names of the variable and value columns with the **var_name** and **value_name** parameters



```
df.melt(id_vars =['column_A', 'column_B'],
              var_name = 'variable_column',
              value_name = 'value_column')
```

	column_A	column_B	variable_column	value_column
0	C1	X	column_C	91
1	C1	Y	column_C	91
2	C2	X	column_C	204

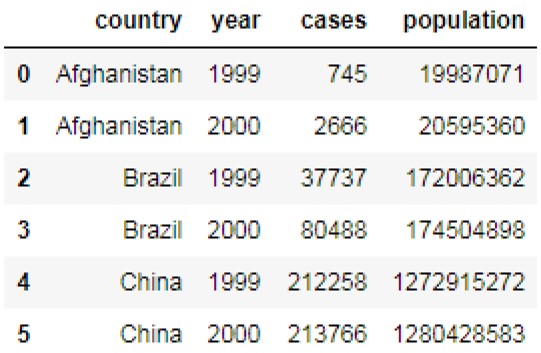


```
df
```

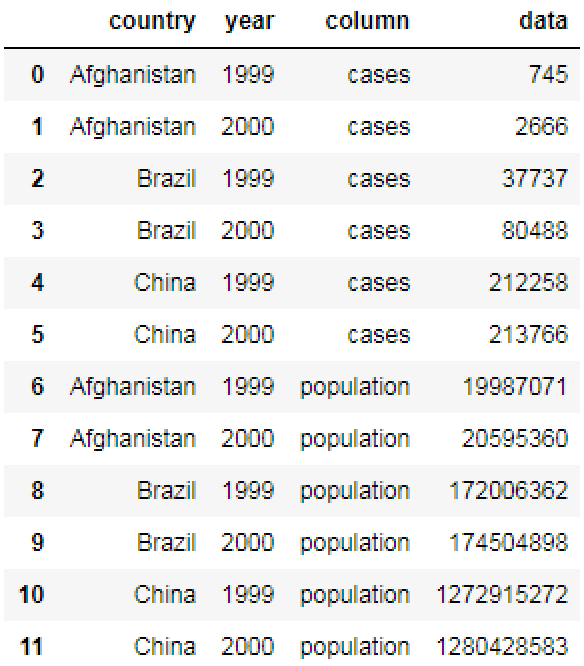
	column_A	column_B	column_C
0	C1	X	91
1	C1	Y	91
2	C2	X	204

Exercise 3

- Convert the dataset "table1" into a narrow table with the following shape:



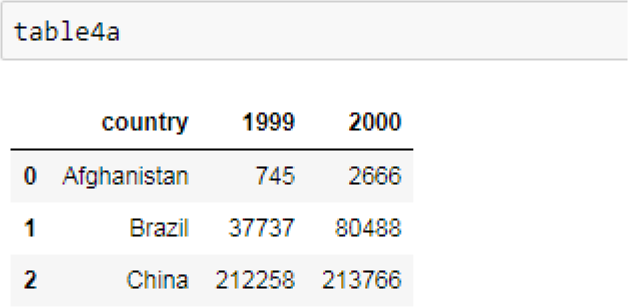
	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583



	country	year	column	data
0	Afghanistan	1999	cases	745
1	Afghanistan	2000	cases	2666
2	Brazil	1999	cases	37737
3	Brazil	2000	cases	80488
4	China	1999	cases	212258
5	China	2000	cases	213766
6	Afghanistan	1999	population	19987071
7	Afghanistan	2000	population	20595360
8	Brazil	1999	population	172006362
9	Brazil	2000	population	174504898
10	China	1999	population	1272915272
11	China	2000	population	1280428583

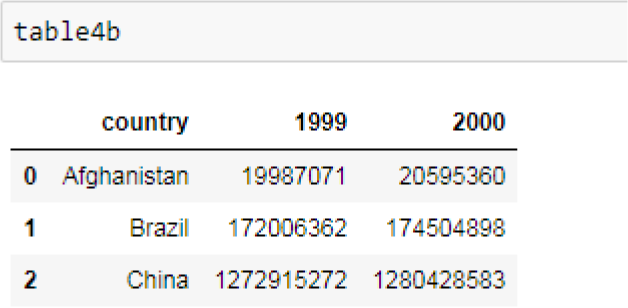
Exercise 4

- Converts the datasets "table4a" and "table4b" into a clean dataset, as seen in "table1"



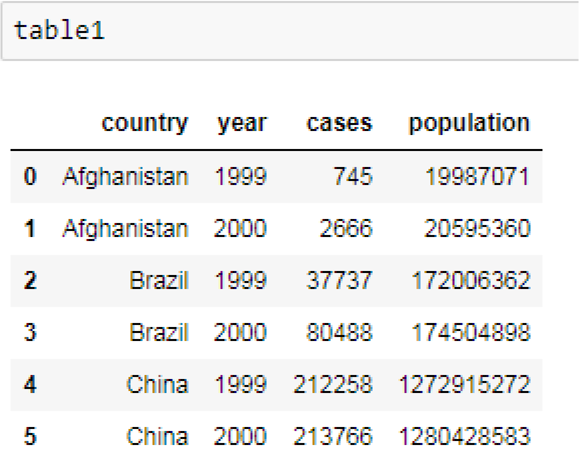
```
table4a
```

	country	1999	2000
0	Afghanistan	745	2666
1	Brazil	37737	80488
2	China	212258	213766



```
table4b
```

	country	1999	2000
0	Afghanistan	19987071	20595360
1	Brazil	172006362	174504898
2	China	1272915272	1280428583



```
table1
```

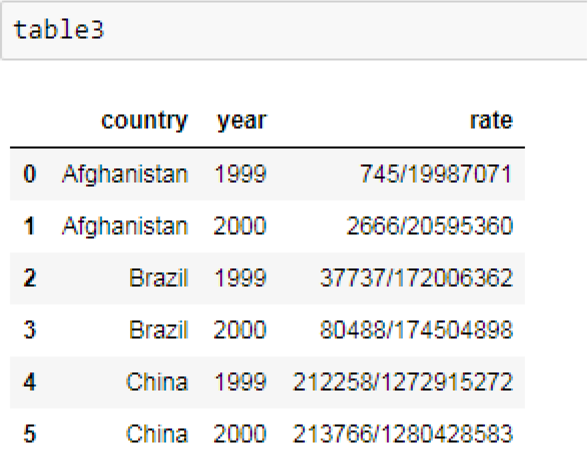
	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

Agenda

- Introduction
- Widening tables
- Narrowing down tables
- Separating columns
- Joining columns
- Missing data
- Dropping duplicates
- Data Types
- Data Formating
- Regex

Separating columns

- We are to fix the ‘Two values in one column’ problem …

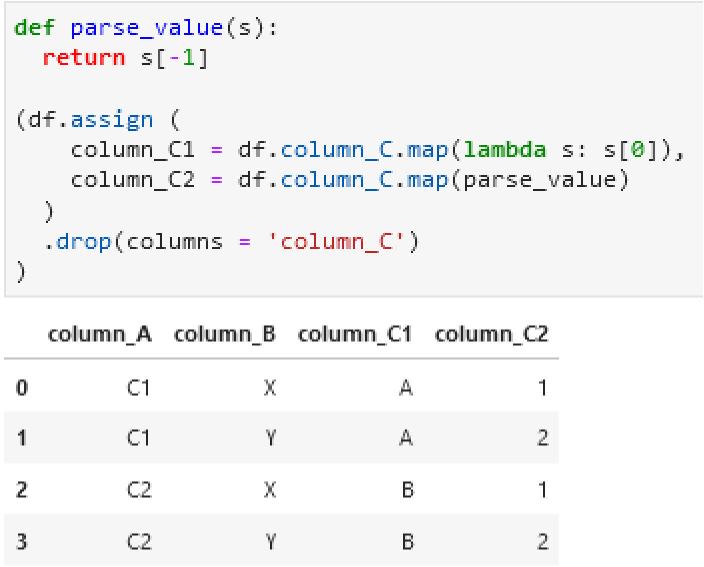


```
table3
```

	country	year	rate
0	Afghanistan	1999	745/19987071
1	Afghanistan	2000	2666/20595360
2	Brazil	1999	37737/172006362
3	Brazil	2000	80488/174504898
4	China	1999	212258/1272915272
5	China	2000	213766/1280428583

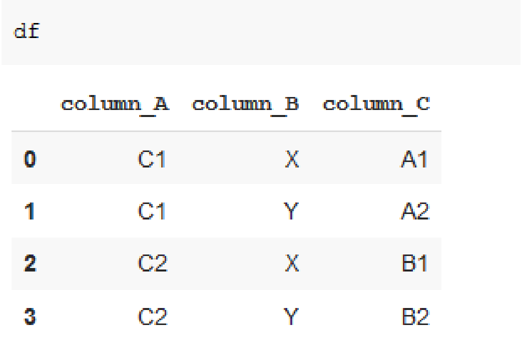
Separating columns

- Another common operation is to separate the value of a column into several columns ...



```
def parse_value(s):
  return s[-1]

(df.assign (
    column_C1 = df.column_C.map(lambda s: s[0]),
    column_C2 = df.column_C.map(parse_value)
  )
  .drop(columns = 'column_C')
)
```

	column_A	column_B	column_C1	column_C2
0	C1	X	A	1
1	C1	Y	A	2
2	C2	X	B	1
3	C2	Y	B	2

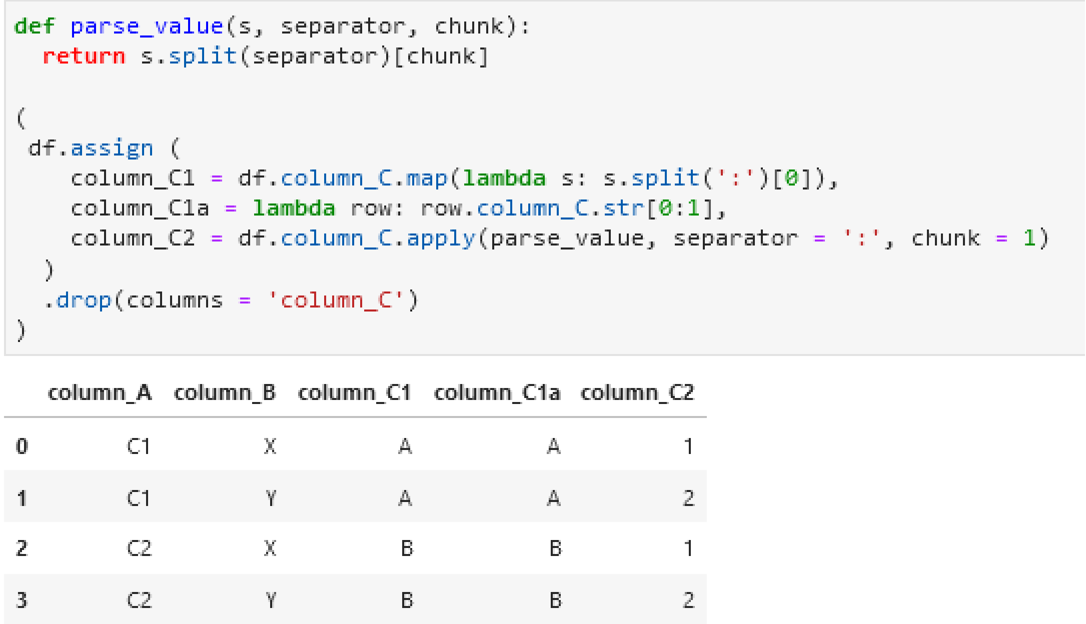


```
df
```

	column_A	column_B	column_C
0	C1	X	A1
1	C1	Y	A2
2	C2	X	B1
3	C2	Y	B2

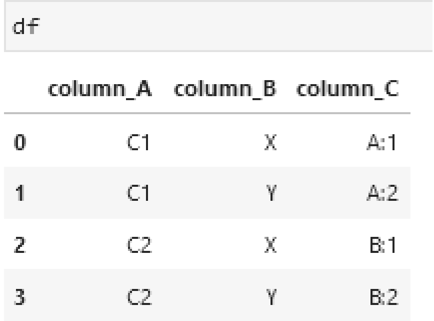
Separating columns

- Another common operation is to separate the value of a column into several columns ...



```
def parse_value(s, separator, chunk):
  return s.split(separator)[chunk]

(
 df.assign (
    column_C1 = df.column_C.map(lambda s: s.split(':')[0]),
    column_C1a = lambda row: row.column_C.str[0:1],
    column_C2 = df.column_C.apply(parse_value, separator = ':', chunk = 1)
  )
  .drop(columns = 'column_C')
)
```

	column_A	column_B	column_C1	column_C1a	column_C2
0	C1	X	A	A	1
1	C1	Y	A	A	2
2	C2	X	B	B	1
3	C2	Y	B	B	2

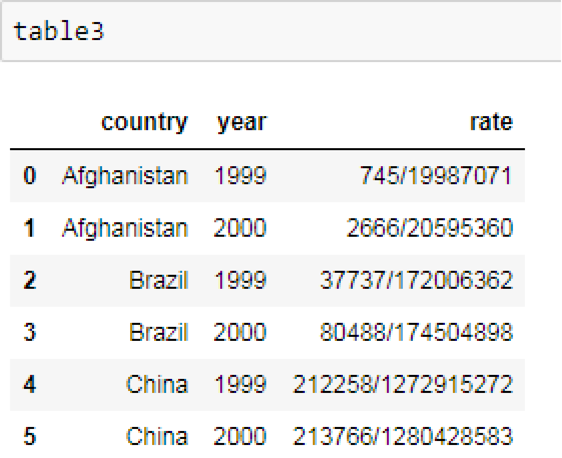


```
df
```

	column_A	column_B	column_C
0	C1	X	A:1
1	C1	Y	A:2
2	C2	X	B:1
3	C2	Y	B:2

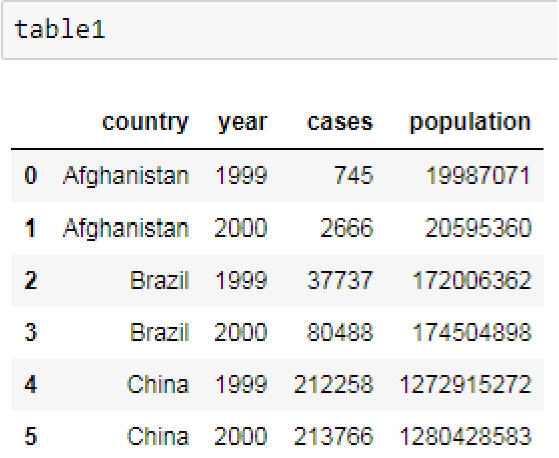
Exercise 5

- Converts the dataset "table3" into a clean dataset, as seen in "table1"
- Make sure the new columns have the int datatype



```
table3
```

	country	year	rate
0	Afghanistan	1999	745/19987071
1	Afghanistan	2000	2666/20595360
2	Brazil	1999	37737/172006362
3	Brazil	2000	80488/174504898
4	China	1999	212258/1272915272
5	China	2000	213766/1280428583



```
table1
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

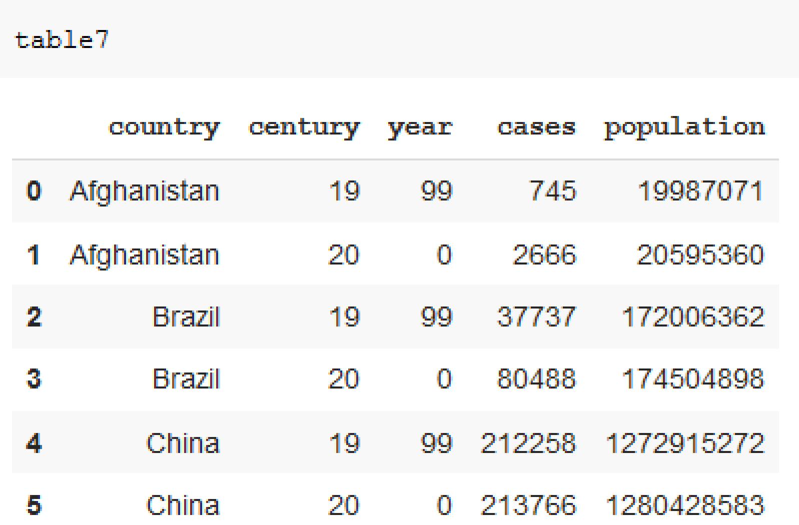
df = table3.copy()

Agenda

- Introduction
- Widening tables
- Narrowing down tables
- Separating columns
- Joining columns
- Missing data
- Dropping duplicates
- Data Types
- Data Formating
- Regex

Joining columns

- We are to fix the ‘Same value in two diferent columns’ problem ...

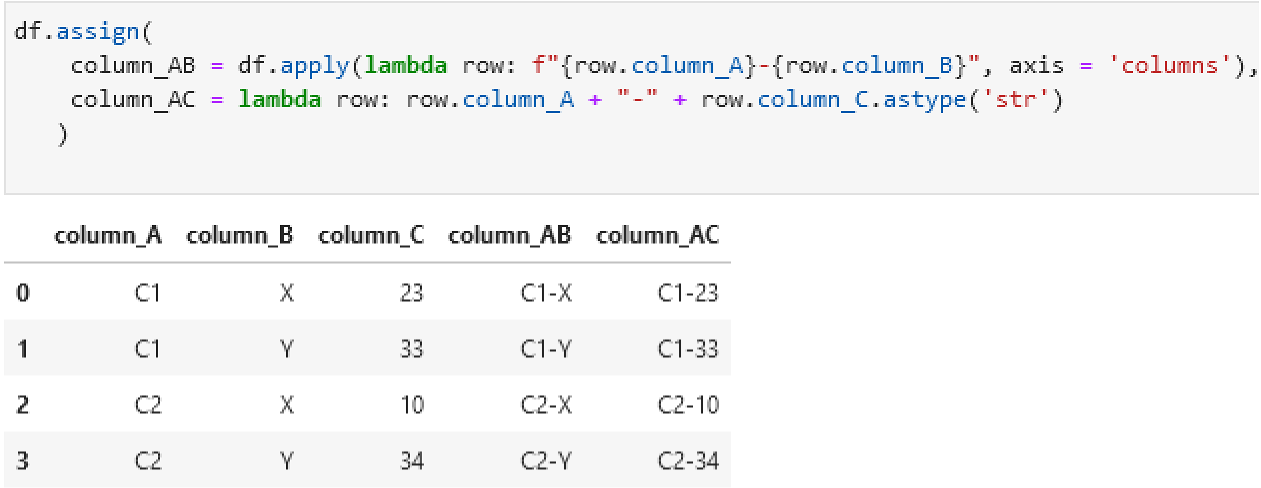


```
table7
```

	country	century	year	cases	population
0	Afghanistan	19	99	745	19987071
1	Afghanistan	20	0	2666	20595360
2	Brazil	19	99	37737	172006362
3	Brazil	20	0	80488	174504898
4	China	19	99	212258	1272915272
5	China	20	0	213766	1280428583

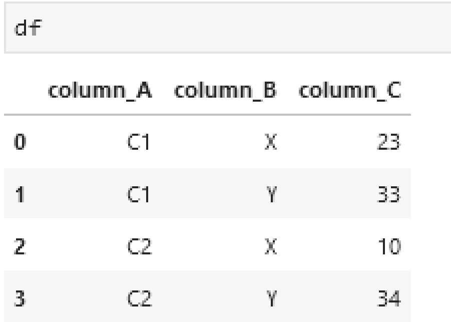
Joining columns

- There are times when we need to join two columns into one...



```
df.assign(
    column_AB = df.apply(lambda row: f"{row.column_A}-{row.column_B}", axis = 'columns'),
    column_AC = lambda row: row.column_A + "-" + row.column_C.astype('str')
   )
```

	column_A	column_B	column_C	column_AB	column_AC
0	C1	X	23	C1-X	C1-23
1	C1	Y	33	C1-Y	C1-33
2	C2	X	10	C2-X	C2-10
3	C2	Y	34	C2-Y	C2-34

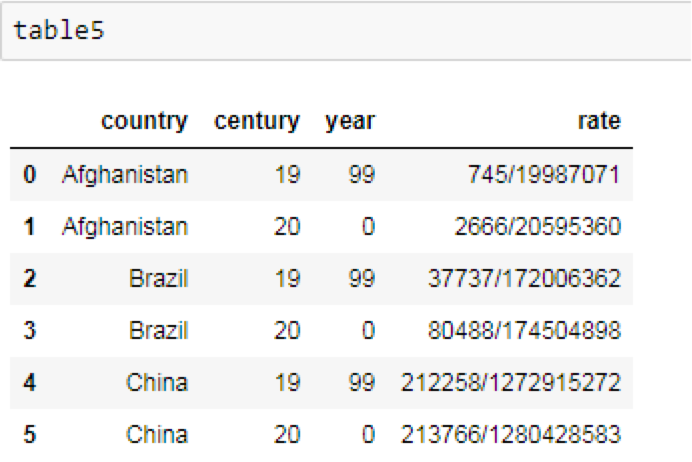


```
df
```

	column_A	column_B	column_C
0	C1	X	23
1	C1	Y	33
2	C2	X	10
3	C2	Y	34

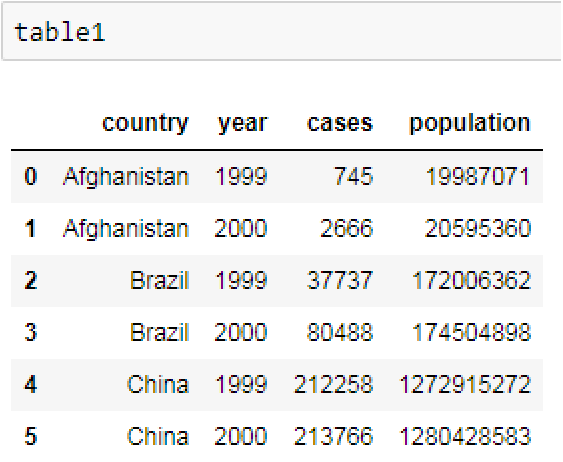
Exercise 6

- Converts the dataset "table5" into a clean dataset, as seen in "table1"
- Make sure the columns are the right type



```
table5
```

	country	century	year	rate
0	Afghanistan	19	99	745/19987071
1	Afghanistan	20	0	2666/20595360
2	Brazil	19	99	37737/172006362
3	Brazil	20	0	80488/174504898
4	China	19	99	212258/1272915272
5	China	20	0	213766/1280428583

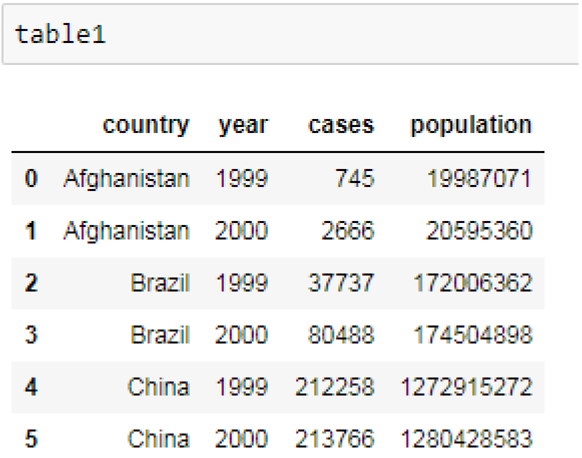


```
table1
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

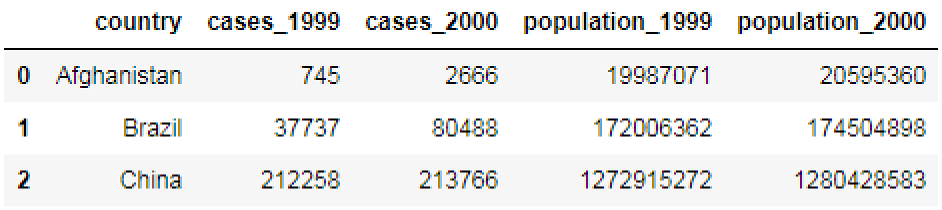
Exercise 7

- Convert the dataset "table1" into a narrow table with the following shape:
- Bunus: Can you done the exercise in one sentence?



```
table1
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583



	country	cases_1999	cases_2000	population_1999	population_2000
0	Afghanistan	745	2666	19987071	20595360
1	Brazil	37737	80488	172006362	174504898
2	China	212258	213766	1272915272	1280428583

Agenda

- Introduction
- Widening tables
- Narrowing down tables
- Separating columns
- Joining columns
- Missing data
- Dropping duplicates
- Data Types
- Data Formating
- Regex

Missing Data

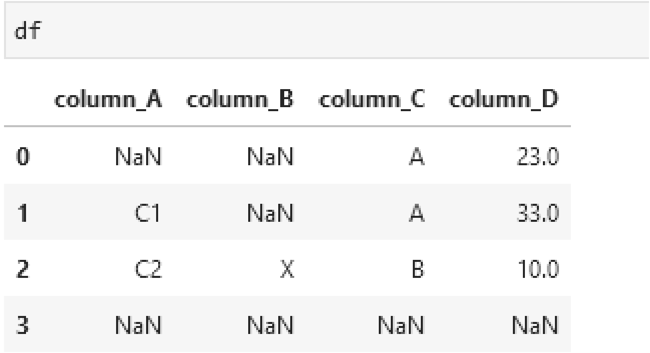
- Missing Data can **generate problems** when trying to represent the data or apply it to an algorithm
- It can hide or represent anomalies in the system
- It is necessary to **identify** and **treat** those missing values (dropping the row or filling the value)

Identifying Missing Data

Pandas provides several methods to identifying null values.

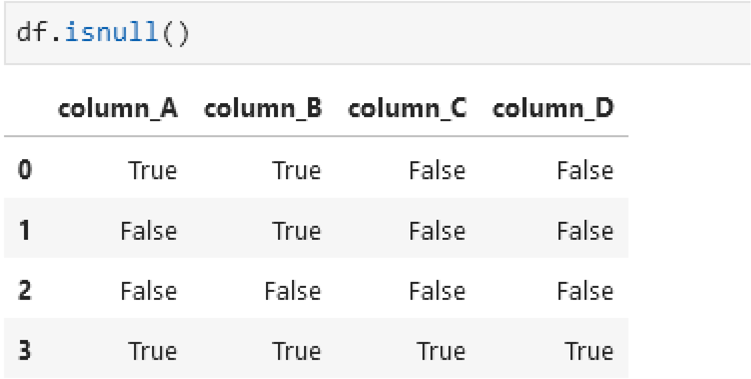
- df.**info**() method to print a summary of a Dataframe
- df.**isnull**() / df.**notnull**() methods to detect missing values

Identifying Missing Data



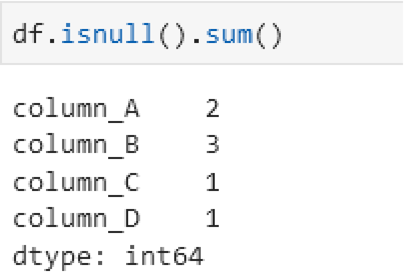
```
df
```

	column_A	column_B	column_C	column_D
0	NaN	NaN	A	23.0
1	C1	NaN	A	33.0
2	C2	X	B	10.0
3	NaN	NaN	NaN	NaN



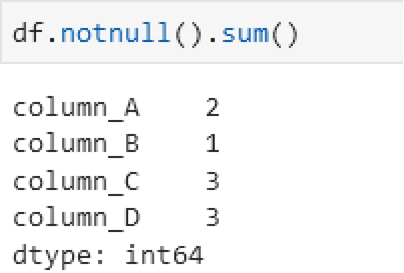
```
df.isnull()
```

	column_A	column_B	column_C	column_D
0	True	True	False	False
1	False	True	False	False
2	False	False	False	False
3	True	True	True	True



```
df.isnull().sum()
```

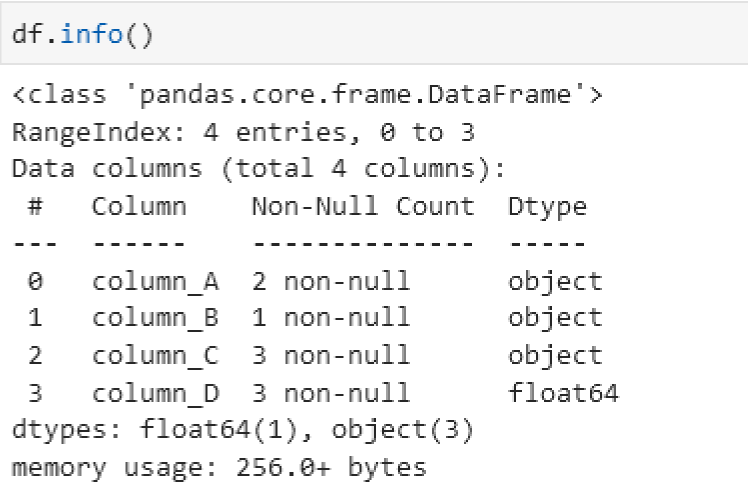
```
column_A    2
column_B    3
column_C    1
column_D    1
dtype: int64
```



```
df.notnull().sum()
```

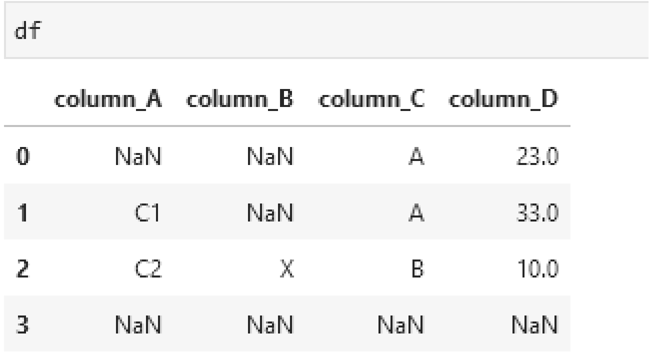
```
column_A    2
column_B    1
column_C    3
column_D    3
dtype: int64
```

Identifying Missing Data



```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   column_A  2 non-null      object 
 1   column_B  1 non-null      object 
 2   column_C  3 non-null      object 
 3   column_D  3 non-null      float64
dtypes: float64(1), object(3)
memory usage: 256.0+ bytes
```

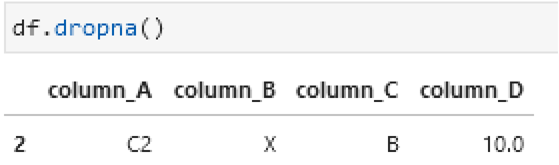


```
df
```

	column_A	column_B	column_C	column_D
0	NaN	NaN	A	23.0
1	C1	NaN	A	33.0
2	C2	X	B	10.0
3	NaN	NaN	NaN	NaN

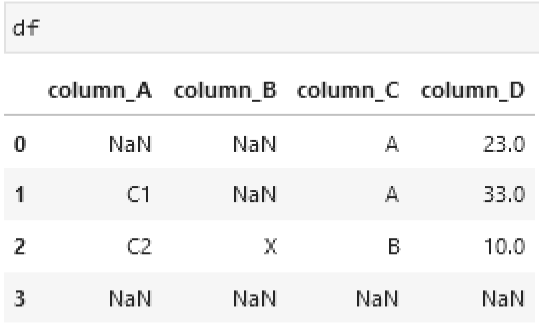
Removing missing data

- The **dropna**() function removes all rows that contain **any** null value
- Note that we remove the full row (not only the columns with missing values)



```
df.dropna()
```

```
   column_A  column_B  column_C  column_D
2        C2         X         B      10.0
```

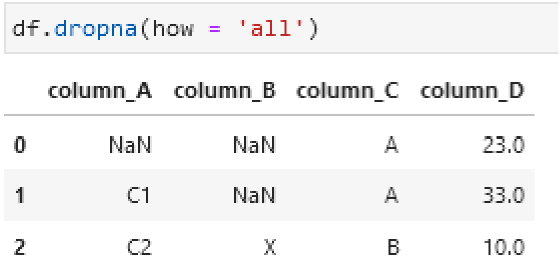


```
df
```

```
   column_A  column_B  column_C  column_D
0       NaN       NaN         A      23.0
1        C1       NaN         A      33.0
2        C2         X         B      10.0
3       NaN       NaN       NaN       NaN
```

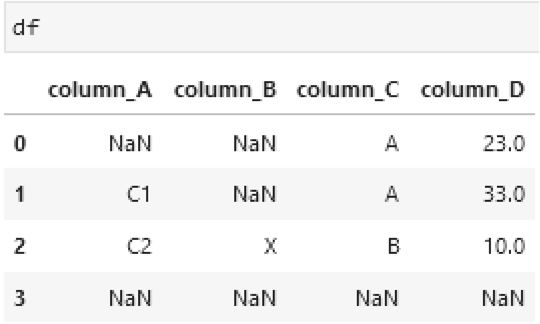
Removing missing data

- The '**how**' parameter allows to specify if we want to remove only the rows with **all values** missing



```
df.dropna(how = 'all')
```

	column_A	column_B	column_C	column_D
0	NaN	NaN	A	23.0
1	C1	NaN	A	33.0
2	C2	X	B	10.0

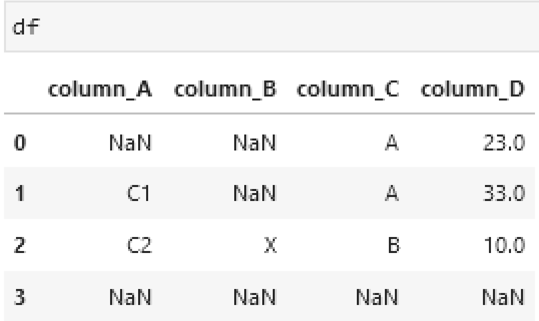


```
df
```

	column_A	column_B	column_C	column_D
0	NaN	NaN	A	23.0
1	C1	NaN	A	33.0
2	C2	X	B	10.0
3	NaN	NaN	NaN	NaN

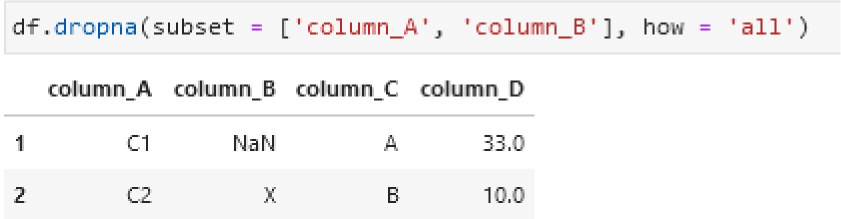
Removing missing data

- The **'subset'** parameter allows you to specify a subset of columns whose value must be null to remove the row



```
df
```

	column_A	column_B	column_C	column_D
0	NaN	NaN	A	23.0
1	C1	NaN	A	33.0
2	C2	X	B	10.0
3	NaN	NaN	NaN	NaN



```
df.dropna(subset = ['column_A', 'column_B'], how = 'all')
```

	column_A	column_B	column_C	column_D
1	C1	NaN	A	33.0
2	C2	X	B	10.0

Filling missing Data

- The **fillna**() function replaces missing values in a dataset.
- This method can be applied to a whole columns in a dataset or an individual column
- In the case of applying it to the entire data set, we have to specify a dictionary where for each column we specify the value that we are going to use to replace a null or missing value

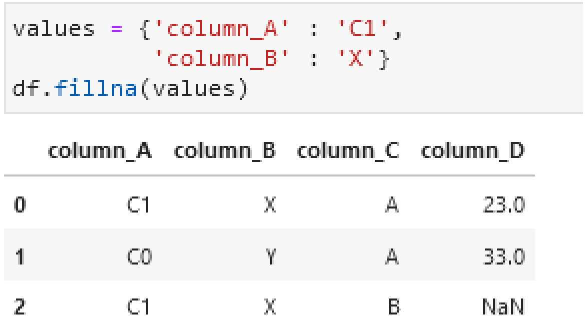
Missing Data Strategies

We can have different strategies to treat missing data:

- Remove the missing data (only when there are enough samples in the dataset)
- Assign a fixed value
- Estimate the missing data with a statistical function (mean, median, most frequent, etc.)
- Estimate the missing data with a more complex method like an interpolation method
- Use the previous or subsequent row

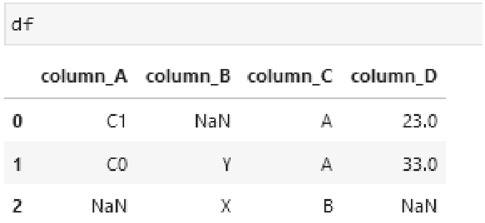
Fixed Values

- In the case of **fixed values** we simply specify the value that we can assign to a column (if the data is missing)



```
values = {'column_A' : 'C1',
          'column_B' : 'X'}
df.fillna(values)
```

	column_A	column_B	column_C	column_D
0	C1	X	A	23.0
1	C0	Y	A	33.0
2	C1	X	B	NaN

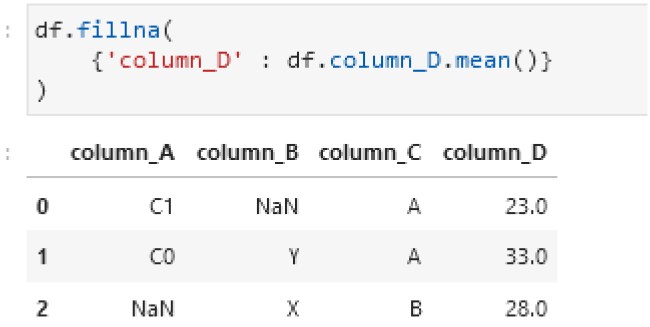


```
df
```

	column_A	column_B	column_C	column_D
0	C1	NaN	A	23.0
1	C0	Y	A	33.0
2	NaN	X	B	NaN

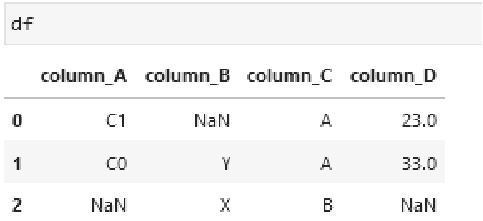
Statistical Function

- In the case of a **statistical function**, we can use a function like the mean or the median



```
df.fillna(
    {'column_D' : df.column_D.mean()}
)
```

	column_A	column_B	column_C	column_D
0	C1	NaN	A	23.0
1	C0	Y	A	33.0
2	NaN	X	B	28.0

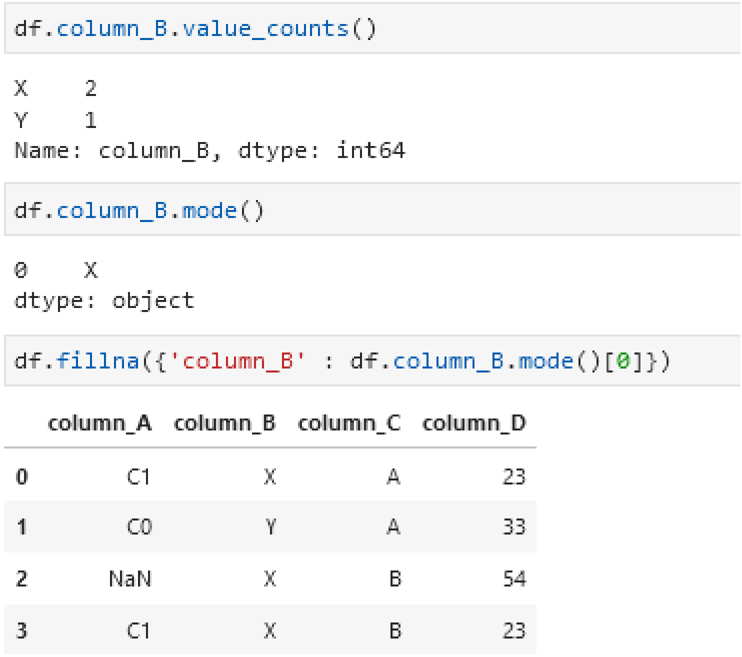


```
df
```

	column_A	column_B	column_C	column_D
0	C1	NaN	A	23.0
1	C0	Y	A	33.0
2	NaN	X	B	NaN

Statistical Function

- In case of **categorical columns** we can not use a mathematical function, we will use the most frequent value of the column (mode)



```
df.column_B.value_counts()
```

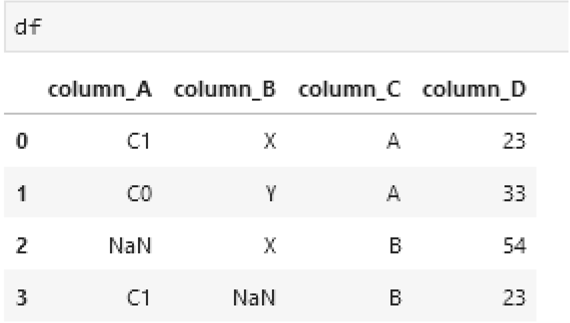
```
X    2
Y    1
Name: column_B, dtype: int64
```

```
df.column_B.mode()
```

```
0    X
dtype: object
```

```
df.fillna({'column_B' : df.column_B.mode()[0]})
```

	column_A	column_B	column_C	column_D
0	C1	X	A	23
1	C0	Y	A	33
2	NaN	X	B	54
3	C1	X	B	23

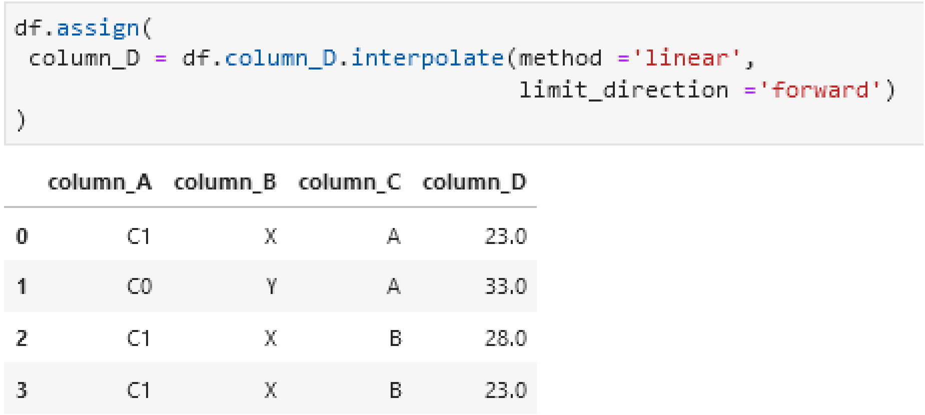


```
df
```

	column_A	column_B	column_C	column_D
0	C1	X	A	23
1	C0	Y	A	33
2	NaN	X	B	54
3	C1	NaN	B	23

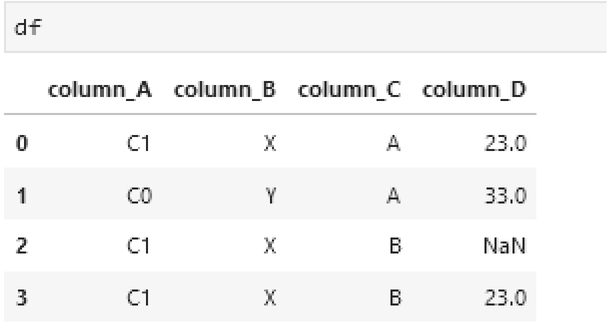
Interpolation Method

- Other posiblitily is estimate missing values using an **interpolation method**



```
df.assign(
 column_D = df.column_D.interpolate(method ='linear',
                                    limit_direction ='forward')
)
```

	column_A	column_B	column_C	column_D
0	C1	X	A	23.0
1	C0	Y	A	33.0
2	C1	X	B	28.0
3	C1	X	B	23.0



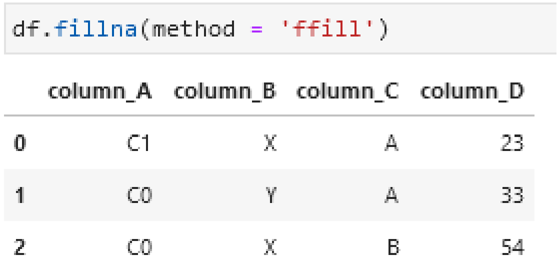
```
df
```

	column_A	column_B	column_C	column_D
0	C1	X	A	23.0
1	C0	Y	A	33.0
2	C1	X	B	NaN
3	C1	X	B	23.0

https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.interpolate.html

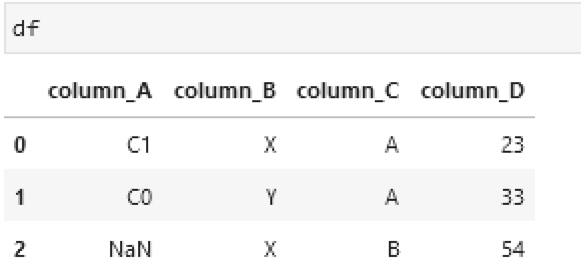
Previous or subsequent row

- We could fill in missing values of a column with the value of the **previous** row (or the **subsequent** row)
- It is a common technique to treat data that comes from Excel



```python
df.fillna(method = 'ffill')
```

	column_A	column_B	column_C	column_D
0	C1	X	A	23
1	C0	Y	A	33
2	C0	X	B	54

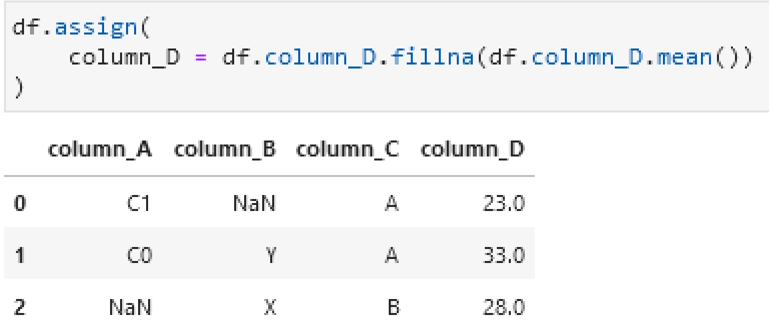


```python
df
```

	column_A	column_B	column_C	column_D
0	C1	X	A	23
1	C0	Y	A	33
2	NaN	X	B	54

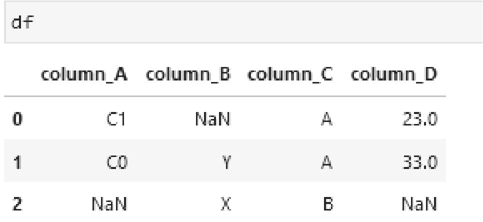
Individual Columns

- The **fillna()** method can also be applied to **an individual column** instead of applying it to all columns at the same time
- We could use any of the strategies we have seen



```
df.assign(
    column_D = df.column_D.fillna(df.column_D.mean())
)
```

	column_A	column_B	column_C	column_D
0	C1	NaN	A	23.0
1	C0	Y	A	33.0
2	NaN	X	B	28.0



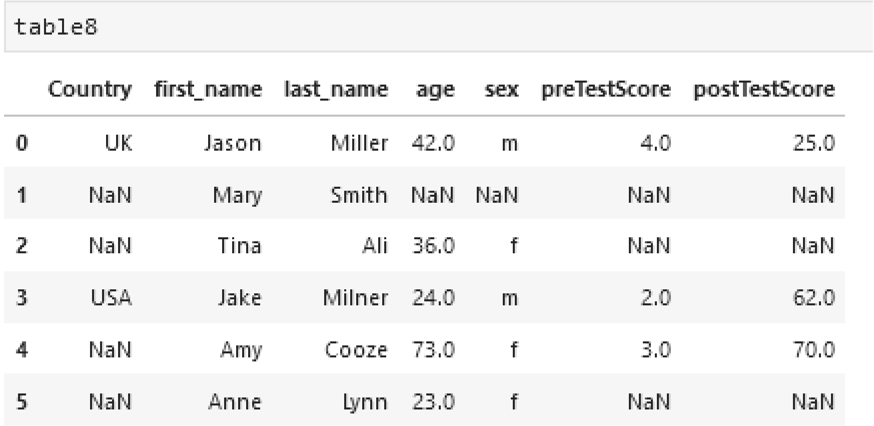
```
df
```

	column_A	column_B	column_C	column_D
0	C1	NaN	A	23.0
1	C0	Y	A	33.0
2	NaN	X	B	NaN

Exercise 8

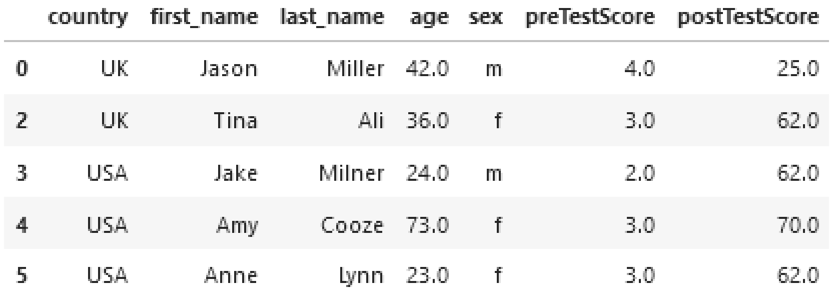
Over the ‘table8’ dataset:

- Determine which column(s) has the greatest number of NaNs.
- Fill the variable 'country' with the value of the subsequent row
- Fill in the null categorical variables with the nost frequent value.
- Fill the variable 'preTestScore' with the mean value
- Fill the variable 'postTestScore' with the median value
- Delete records with missing values in 'age'



```
table8
```

	Country	first_name	last_name	age	sex	preTestScore	postTestScore
0	UK	Jason	Miller	42.0	m	4.0	25.0
1	NaN	Mary	Smith	NaN	NaN	NaN	NaN
2	NaN	Tina	Ali	36.0	f	NaN	NaN
3	USA	Jake	Milner	24.0	m	2.0	62.0
4	NaN	Amy	Cooze	73.0	f	3.0	70.0
5	NaN	Anne	Lynn	23.0	f	NaN	NaN



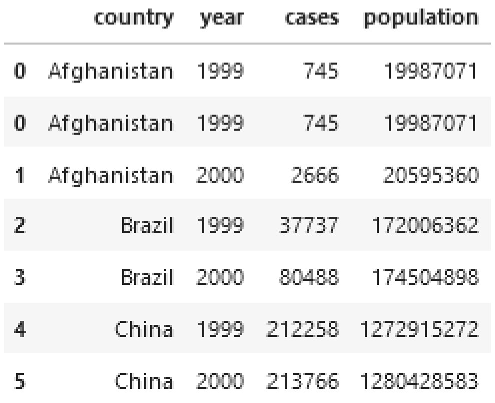
	country	first_name	last_name	age	sex	preTestScore	postTestScore
0	UK	Jason	Miller	42.0	m	4.0	25.0
2	UK	Tina	Ali	36.0	f	3.0	62.0
3	USA	Jake	Milner	24.0	m	2.0	62.0
4	USA	Amy	Cooze	73.0	f	3.0	70.0
5	USA	Anne	Lynn	23.0	f	3.0	62.0

Agenda

- Introduction
- Widening tables
- Narrowing down tables
- Separating columns
- Joining columns
- Missing data
- Dropping duplicates
- Data Types
- Data Formating
- Regex

Droping duplicates

- Dropping duplicates from your data sets is a task you will may have to do as a Data Analyst.
- These duplicates may have been created through lax data integrity or incorrect joining methods during data extraction



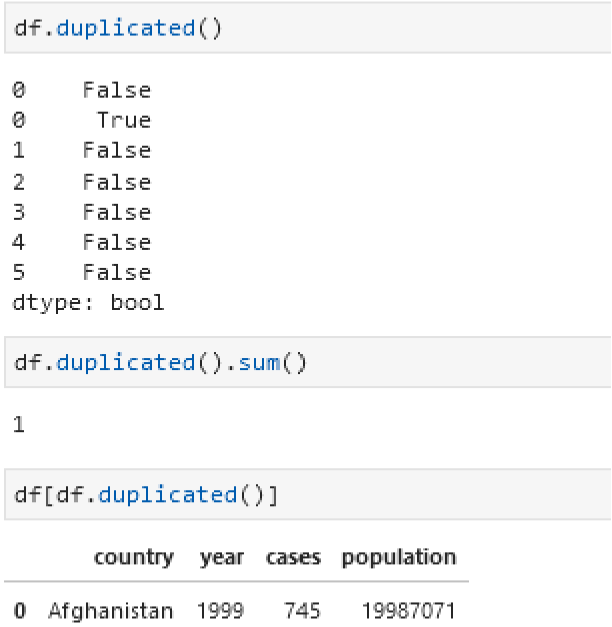
	country	year	cases	population
0	Afghanistan	1999	745	19987071
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

Identifying Duplicates

- Before we remove duplicates, we first need to check whether or not our data set contains duplicates and how we define what a duplicate is.
- Depending on your requirements, a duplicate could either be the duplication of an entire row or duplication based on business rules such as an employee have unique job numbers

Identifying Duplicates

- df.**duplicate**() lets you localize duplicates
- In this case search duplicates on the basis of all columns



```
df.duplicated()
```

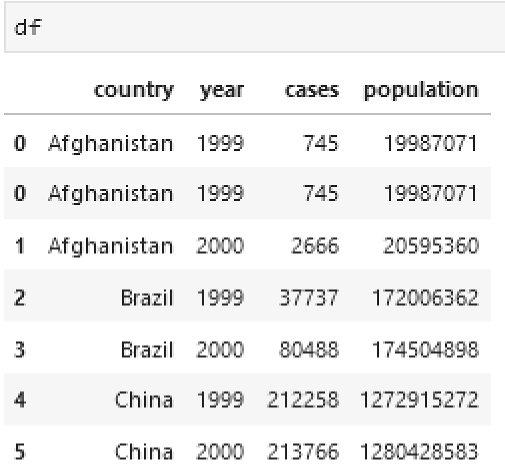
```
0    False
0     True
1    False
2    False
3    False
4    False
5    False
dtype: bool
```

```
df.duplicated().sum()
```

```
1
```

```
df[df.duplicated()]
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071

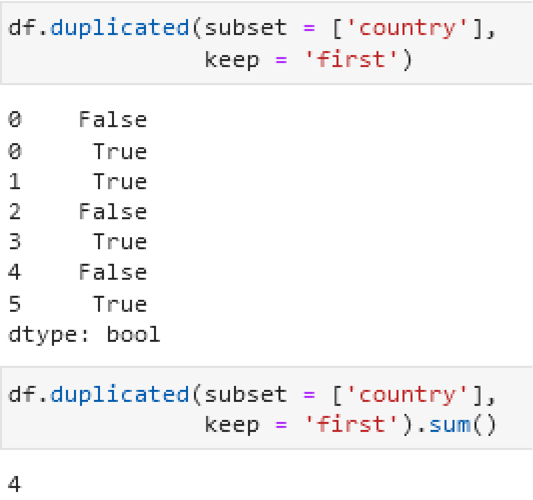


```
df
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

Identifying Duplicates

- **df.duplicates()** can also search for duplicates on the basis of a subset of columns
- “**keep**” parameter specify which row is kept

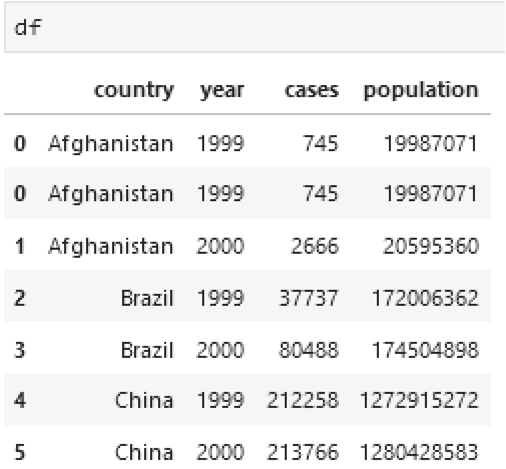


```
df.duplicated(subset = ['country'],
              keep = 'first')
```

```
0    False
0     True
1     True
2    False
3     True
4    False
5     True
dtype: bool
```

```
df.duplicated(subset = ['country'],
              keep = 'first').sum()
```

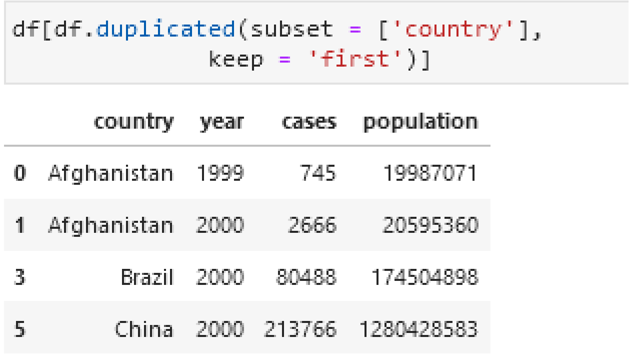
```
4
```



```
df
```

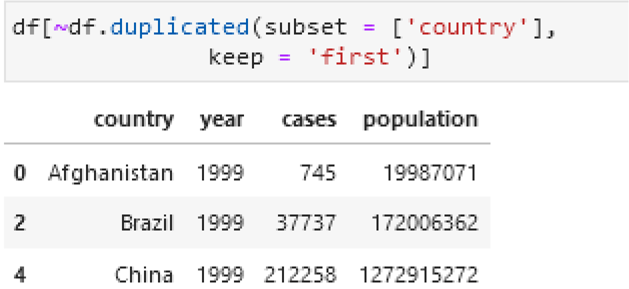
	country	year	cases	population
0	Afghanistan	1999	745	19987071
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

Identifying Duplicates



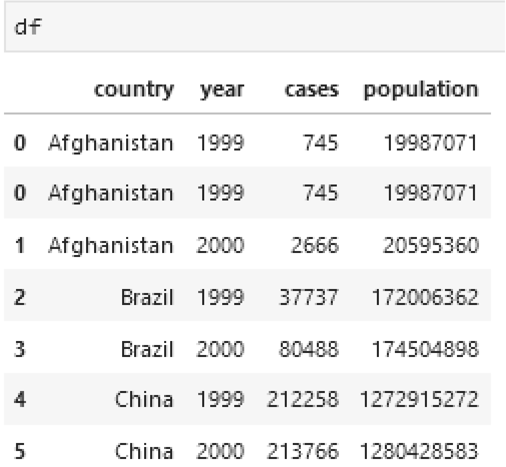
```
df[df.duplicated(subset = ['country'],
              keep = 'first')]
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
3	Brazil	2000	80488	174504898
5	China	2000	213766	1280428583



```
df[~df.duplicated(subset = ['country'],
              keep = 'first')]
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
2	Brazil	1999	37737	172006362
4	China	1999	212258	1272915272



```
df
```

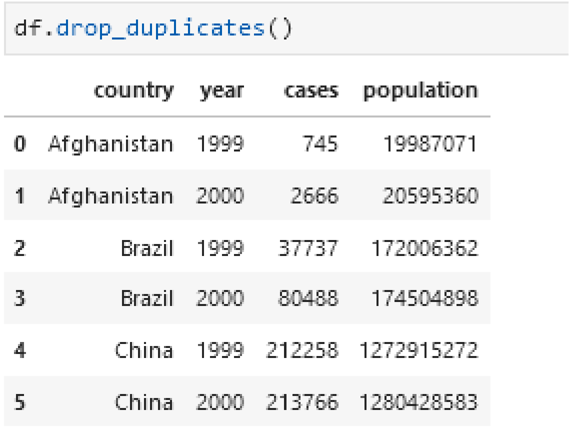
	country	year	cases	population
0	Afghanistan	1999	745	19987071
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

Droping duplicate rows

- To drop duplicates we use the **drop_duplicates**() function.
- We can use different strategies:
  - Drop all duplicates, on the basis of all the columns
  - Drop all duplicates, on the basis of a subset of columns

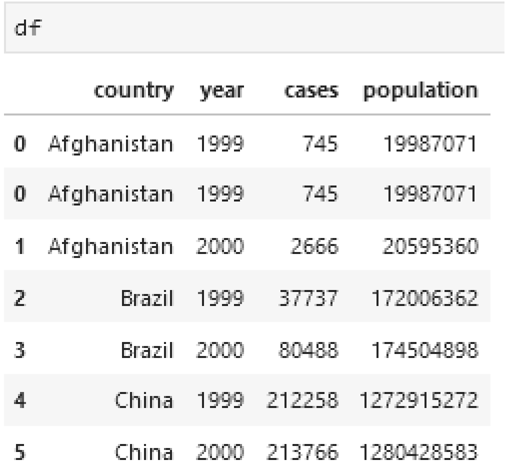
Droping duplicate rows

- Drop all duplicates, on the basis of all the columns



```
df.drop_duplicates()
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

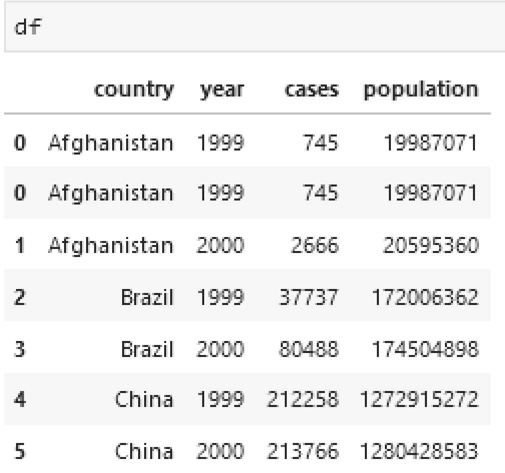


```
df
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

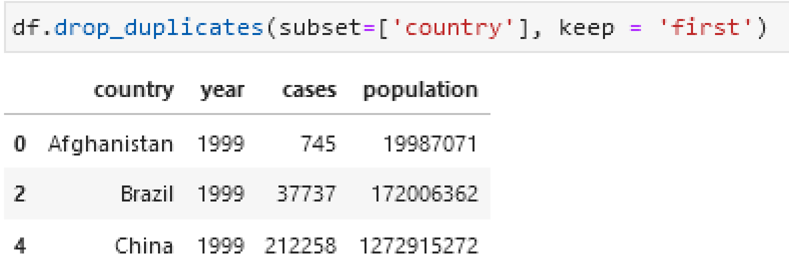
Droping duplicate rows

- Drop all duplicates, on the basis of a subset of columns
- Use the parameter “**keep**” indicating the row to be deleted (‘first’ or ‘last’)
- Order the values of the dataset if you need a specific order



```
df
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583



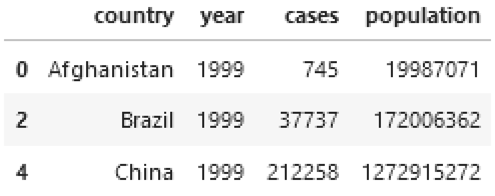
```
df.drop_duplicates(subset=['country'], keep = 'first')
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
2	Brazil	1999	37737	172006362
4	China	1999	212258	1272915272

Exercise 9

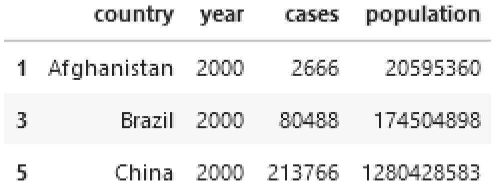
- On table1, keep only one distinct values for the “Country” column (The rows with highest “cases”).
- Identify the rows that are going to be removed

Rows Removed:

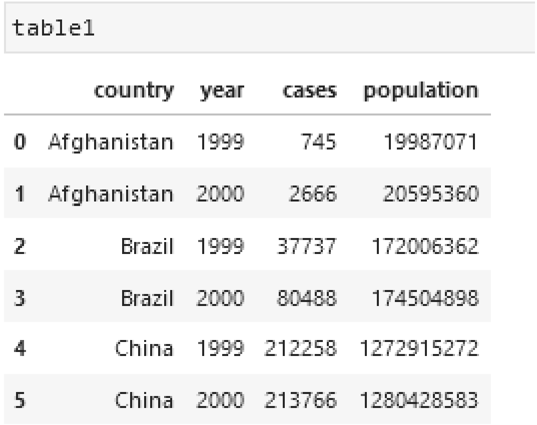


	country	year	cases	population
0	Afghanistan	1999	745	19987071
2	Brazil	1999	37737	172006362
4	China	1999	212258	1272915272

Rows held :



	country	year	cases	population
1	Afghanistan	2000	2666	20595360
3	Brazil	2000	80488	174504898
5	China	2000	213766	1280428583



```
table1
```

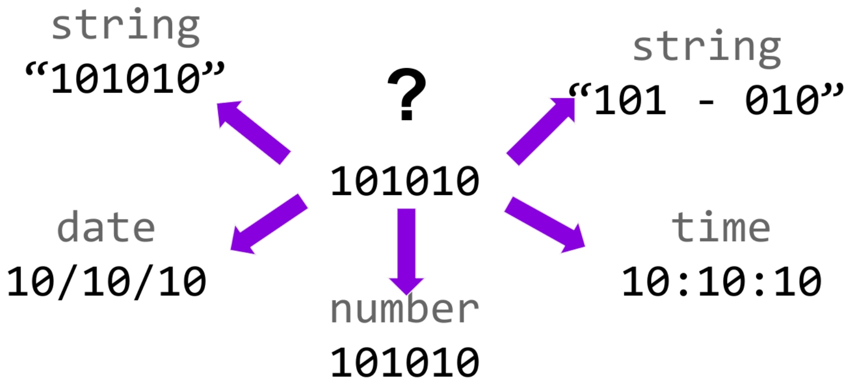
	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

Agenda

- Introduction
- Widening tables
- Narrowing down tables
- Separating columns
- Joining columns
- Missing data
- Dropping duplicates
- Data Types
- Data Formating
- Regex

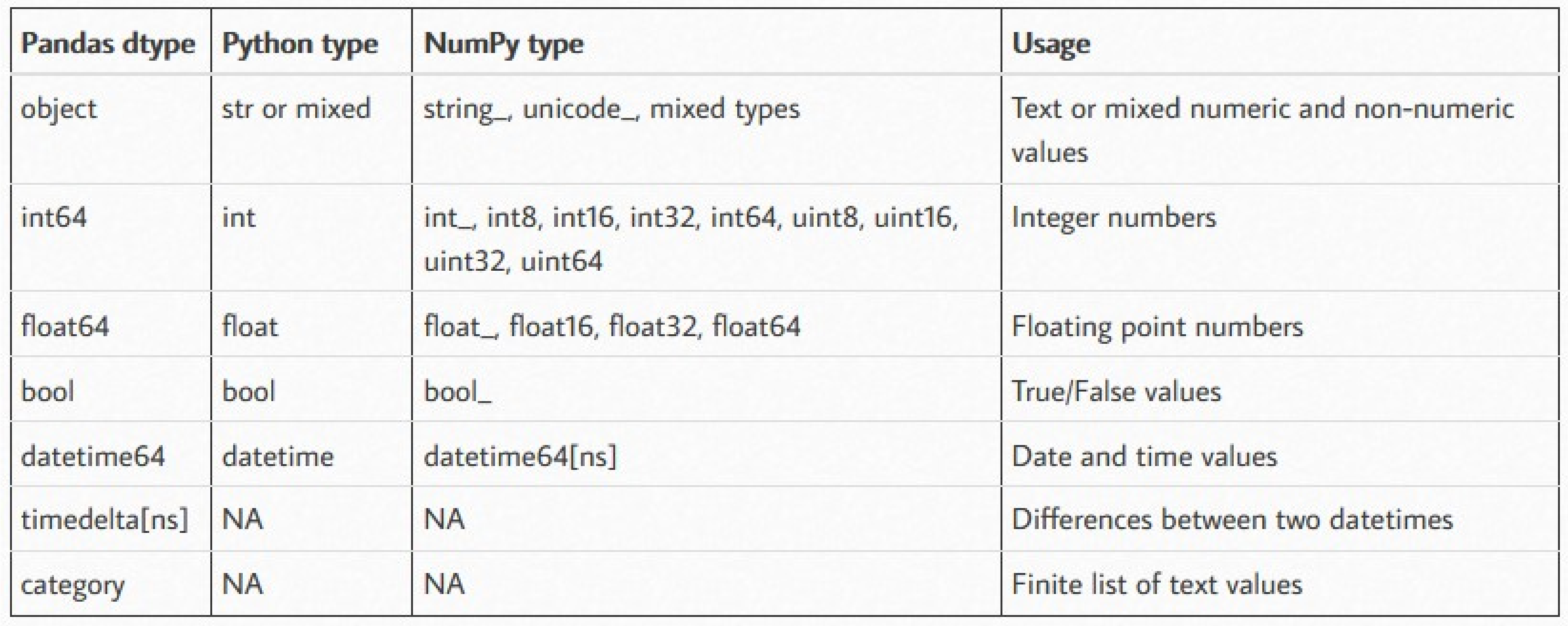
Data Types

- Correctly interpreting the data type is crucial
- We should make sure that every column is assigned to the correct data type
- Data types are one of those things that you don't tend to care about until you get an error or some unexpected results



Data Types

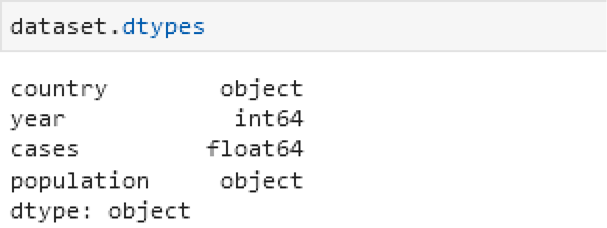
- A data type is essentially an internal construct that a programming language uses to understand how to store and manipulate data



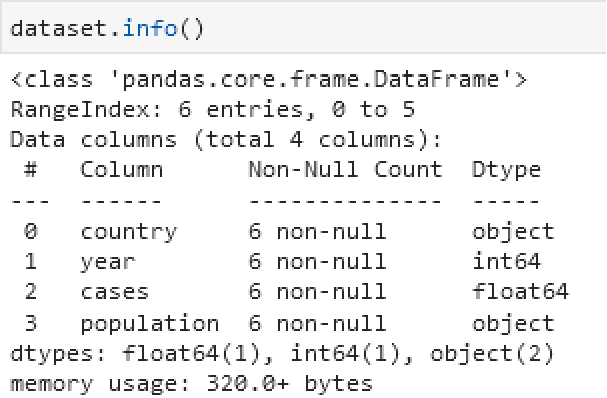
Pandas dtype	Python type	NumPy type	Usage
object	str or mixed	string_, unicode_, mixed types	Text or mixed numeric and non-numeric values
int64	int	int_, int8, int16, int32, int64, uint8, uint16, uint32, uint64	Integer numbers
float64	float	float_, float16, float32, float64	Floating point numbers
bool	bool	bool_	True/False values
datetime64	datetime	datetime64[ns]	Date and time values
timedelta[ns]	NA	NA	Differences between two datetimes
category	NA	NA	Finite list of text values

Identifying Data Types

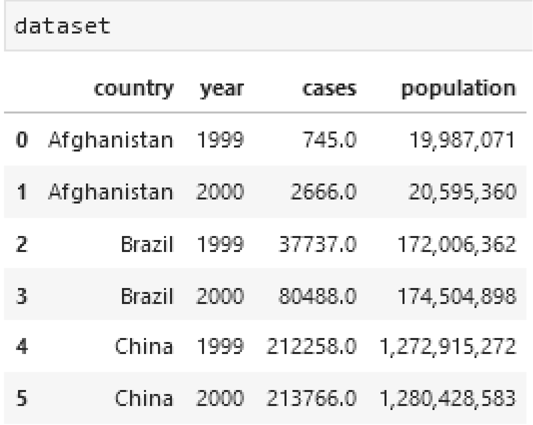
- df.**dtypes** displays all the data types are in a dataframe
- Additionally, the df.**info**() function shows even more useful info



```
dataset.dtypes

country          object
year              int64
cases           float64
population       object
dtype: object
```



```
dataset.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   country     6 non-null      object 
 1   year        6 non-null      int64  
 2   cases       6 non-null      float64
 3   population  6 non-null      object 
dtypes: float64(1), int64(1), object(2)
memory usage: 320.0+ bytes
```

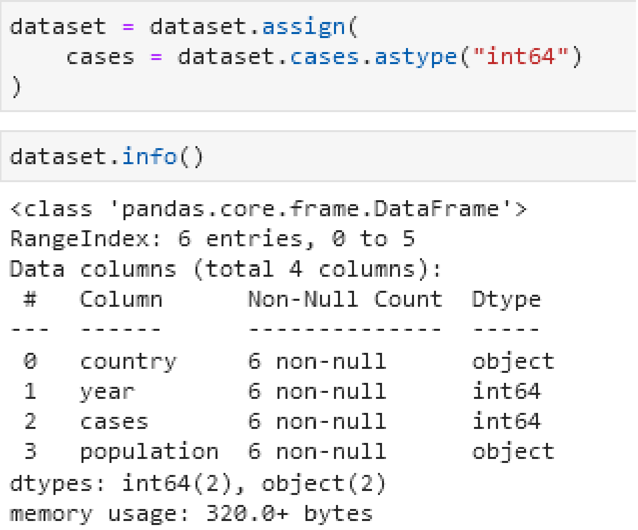


```
dataset
```

	country	year	cases	population
0	Afghanistan	1999	745.0	19,987,071
1	Afghanistan	2000	2666.0	20,595,360
2	Brazil	1999	37737.0	172,006,362
3	Brazil	2000	80488.0	174,504,898
4	China	1999	212258.0	1,272,915,272
5	China	2000	213766.0	1,280,428,583

Converting Data Types

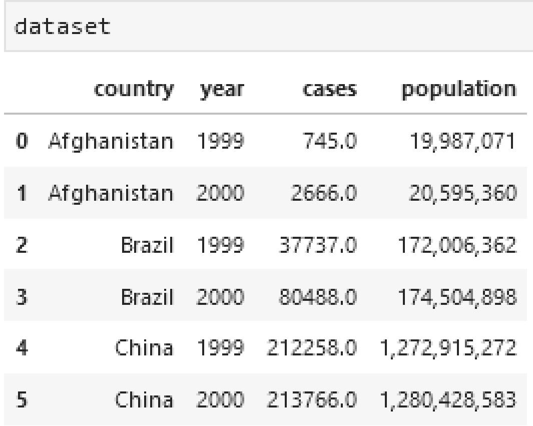
- The simplest way to convert a pandas column of data to a different type is to use **astype**() function



```
dataset = dataset.assign(
    cases = dataset.cases.astype("int64")
)
```

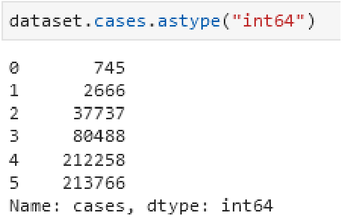
```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   country     6 non-null      object
 1   year        6 non-null      int64
 2   cases       6 non-null      int64
 3   population  6 non-null      object
dtypes: int64(2), object(2)
memory usage: 320.0+ bytes
```



```
dataset
```

	country	year	cases	population
0	Afghanistan	1999	745.0	19,987,071
1	Afghanistan	2000	2666.0	20,595,360
2	Brazil	1999	37737.0	172,006,362
3	Brazil	2000	80488.0	174,504,898
4	China	1999	212258.0	1,272,915,272
5	China	2000	213766.0	1,280,428,583

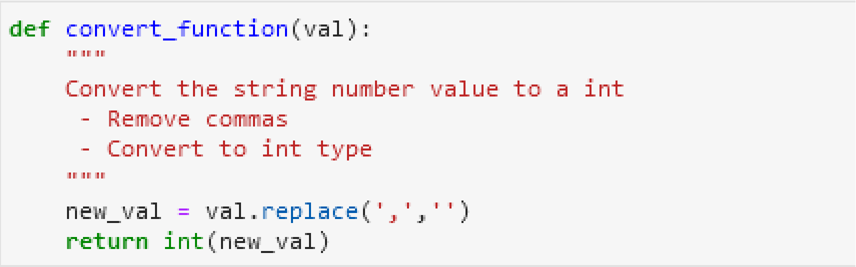


```
dataset.cases.astype("int64")
```

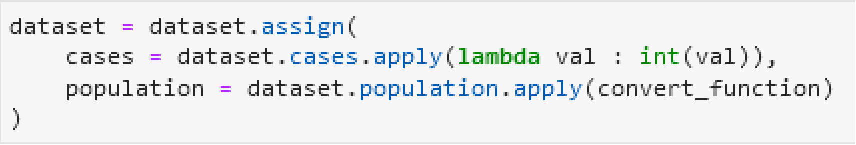
```
0       745
1      2666
2     37737
3     80488
4    212258
5    213766
Name: cases, dtype: int64
```

Converting Data Types

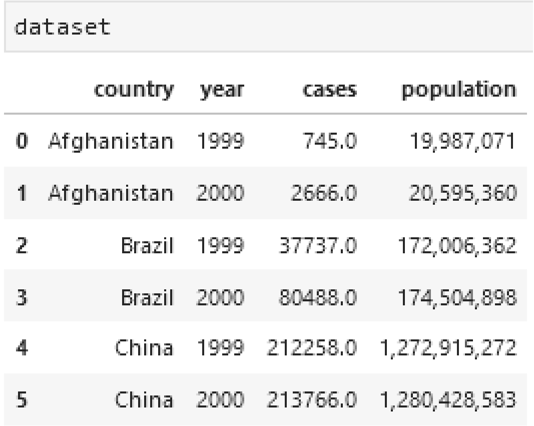
- Since this data is a little more complex to convert, we can build a **custom function** that we apply to each value and convert to the appropriate data type.
- We can use **lambda** functions too



```
def convert_function(val):
    """
    Convert the string number value to a int
     - Remove commas
     - Convert to int type
    """
    new_val = val.replace(',','')
    return int(new_val)
```



```
dataset = dataset.assign(
    cases = dataset.cases.apply(lambda val : int(val)),
    population = dataset.population.apply(convert_function)
)
```

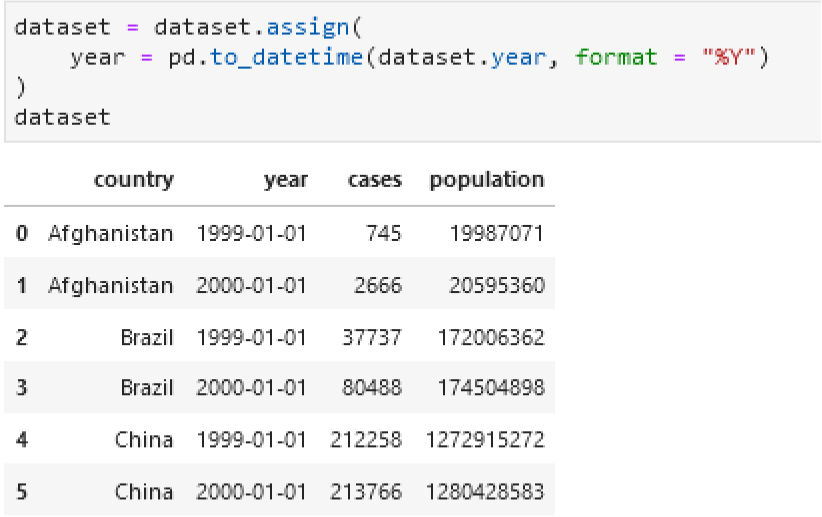


```
dataset
```

	country	year	cases	population
0	Afghanistan	1999	745.0	19,987,071
1	Afghanistan	2000	2666.0	20,595,360
2	Brazil	1999	37737.0	172,006,362
3	Brazil	2000	80488.0	174,504,898
4	China	1999	212258.0	1,272,915,272
5	China	2000	213766.0	1,280,428,583

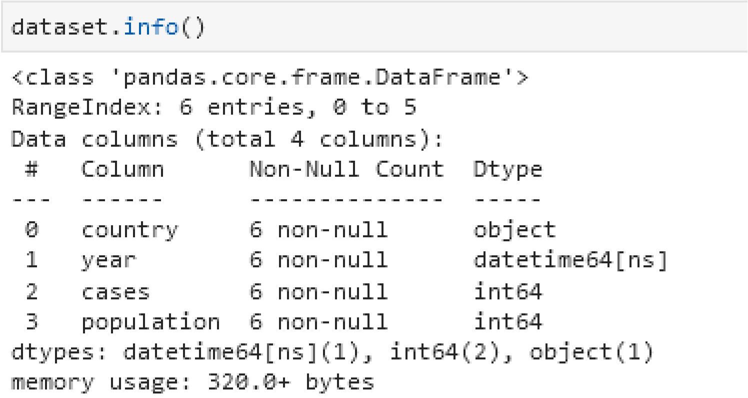
Pandas helper functions

- Pandas has a middle ground between the astype() function and the more complex custom functions
- **pd.to_datetime**() converts its argument to a datetime



```
dataset = dataset.assign(
    year = pd.to_datetime(dataset.year, format = "%Y")
)
dataset
```

	country	year	cases	population
0	Afghanistan	1999-01-01	745	19987071
1	Afghanistan	2000-01-01	2666	20595360
2	Brazil	1999-01-01	37737	172006362
3	Brazil	2000-01-01	80488	174504898
4	China	1999-01-01	212258	1272915272
5	China	2000-01-01	213766	1280428583



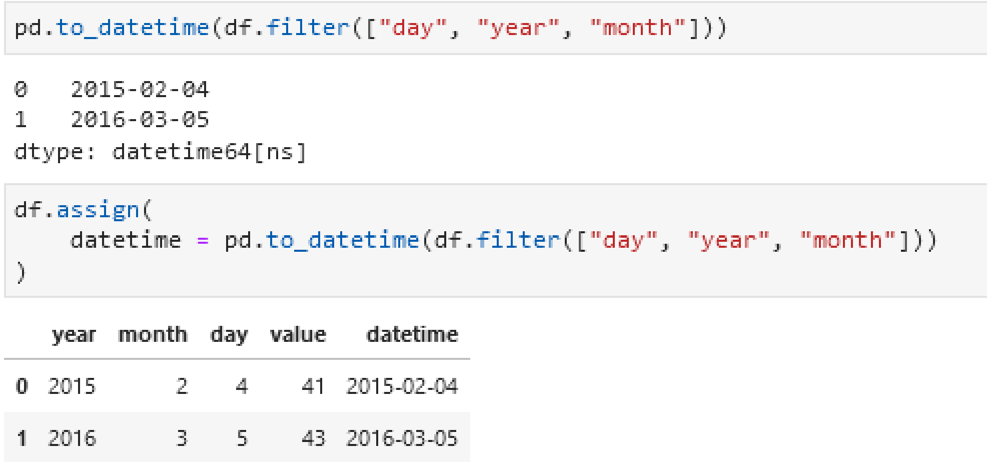
```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   country     6 non-null      object
 1   year        6 non-null      datetime64[ns]
 2   cases       6 non-null      int64
 3   population  6 non-null      int64
dtypes: datetime64[ns](1), int64(2), object(1)
memory usage: 320.0+ bytes
```

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.to_datetime.html

Pandas helper functions

- If we have a dataframe with the columns ‘year’, ‘month’ and ‘day’ we can use **pd.to_datetime**() to get a new datetime column

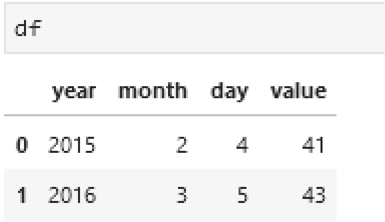


```
pd.to_datetime(df.filter(["day", "year", "month"]))
```

```
0   2015-02-04
1   2016-03-05
dtype: datetime64[ns]
```

```
df.assign(
    datetime = pd.to_datetime(df.filter(["day", "year", "month"]))
)
```

	year	month	day	value	datetime
0	2015	2	4	41	2015-02-04
1	2016	3	5	43	2016-03-05

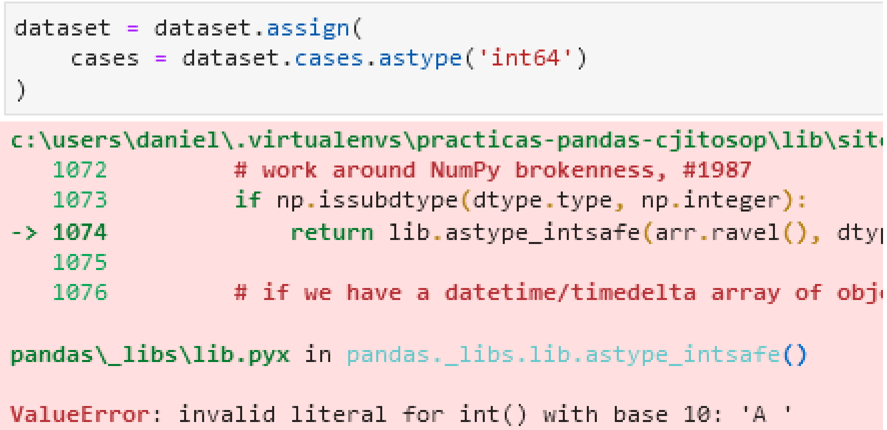


```
df
```

	year	month	day	value
0	2015	2	4	41
1	2016	3	5	43

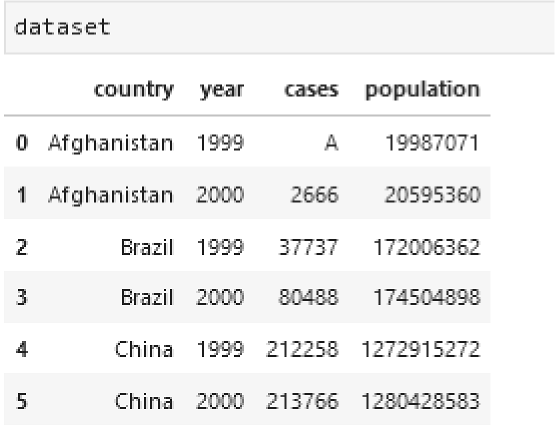
Pandas helper functions

- **pd.to_numeric**() helps us when **astype**() don’t work properly



```
dataset = dataset.assign(
    cases = dataset.cases.astype('int64')
)
```

```
c:\users\daniel\.virtualenvs\practicas-pandas-cjitosop\lib\sit
   1072         # work around NumPy brokenness, #1987
   1073         if np.issubdtype(dtype.type, np.integer):
-> 1074             return lib.astype_intsafe(arr.ravel(), dty
   1075 
   1076         # if we have a datetime/timedelta array of obj

pandas\_libs\lib.pyx in pandas._libs.lib.astype_intsafe()

ValueError: invalid literal for int() with base 10: 'A '
```

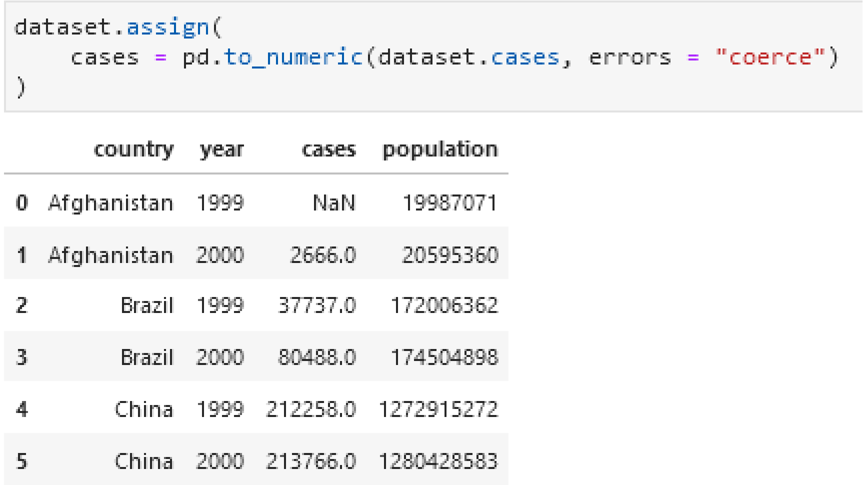


```
dataset
```

	country	year	cases	population
0	Afghanistan	1999	A	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

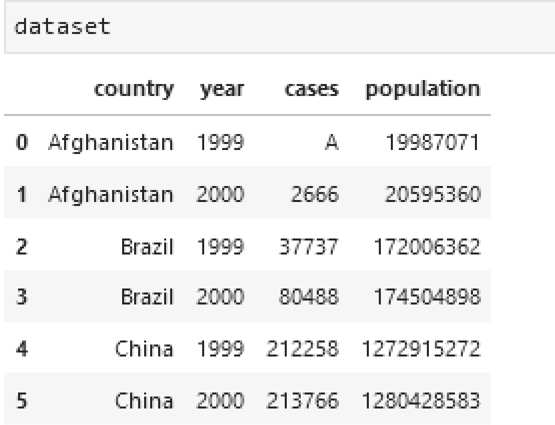
Pandas helper functions

- **pd.to_numeric**() has an argument named ‘**errors**’ that help us deal with convertions errors



```
dataset.assign(
    cases = pd.to_numeric(dataset.cases, errors = "coerce")
)
```

```
       country  year     cases  population
0  Afghanistan  1999       NaN    19987071
1  Afghanistan  2000    2666.0    20595360
2       Brazil  1999   37737.0   172006362
3       Brazil  2000   80488.0   174504898
4        China  1999  212258.0  1272915272
5        China  2000  213766.0  1280428583
```



```
dataset
```

```
       country  year   cases  population
0  Afghanistan  1999       A    19987071
1  Afghanistan  2000    2666    20595360
2       Brazil  1999   37737   172006362
3       Brazil  2000   80488   174504898
4        China  1999  212258  1272915272
5        China  2000  213766  1280428583
```

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.to_numeric.html

Categorical Data

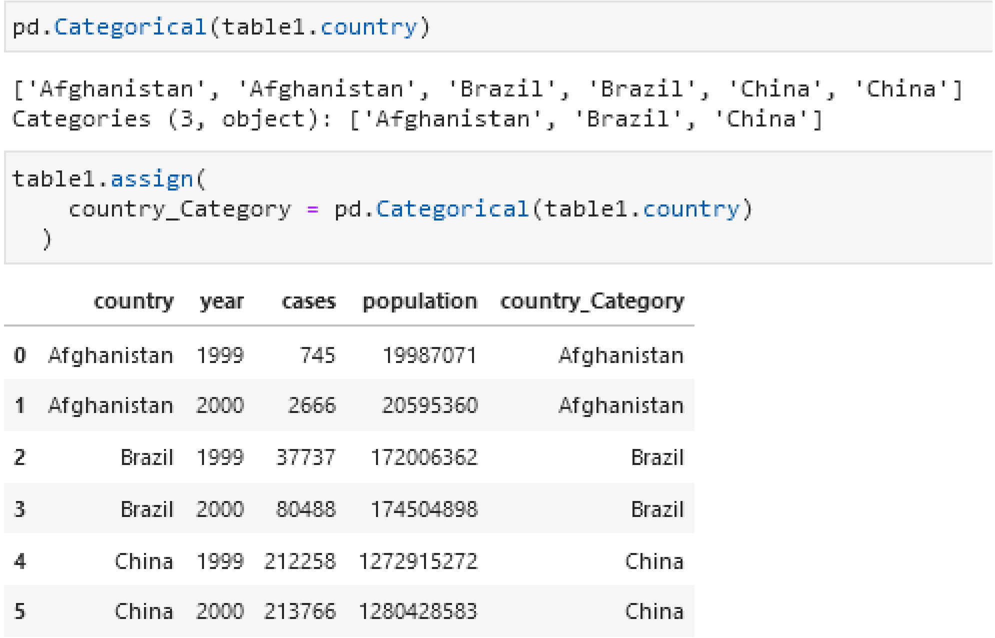
- ***Categoricals*** are a pandas data type corresponding to categorical variables in statistics
- A categorical variable takes on a limited, and fixed, number of possible values
- Categorical data might have an order
- Examples: gender, social class, blood type, country, etc.

Categorical Colums - Pros

- A string variable consisting of only a few different values. Converting such a string variable to a categorical variable will save some memory.
- The lexical order of a variable is not the same as the logical order (“one”, “two”, “three”)
- It is a signal to other Python libraries that this column should be treated as a categorical variable

Categorical Columns

- **pd.Categorical()** convert any column into a category representing a categorical variable

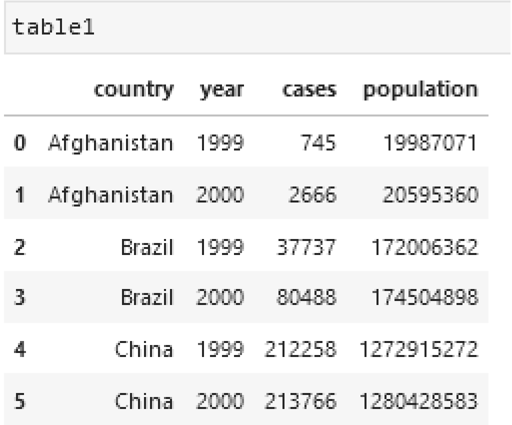


```
pd.Categorical(table1.country)
```

```
['Afghanistan', 'Afghanistan', 'Brazil', 'Brazil', 'China', 'China']
Categories (3, object): ['Afghanistan', 'Brazil', 'China']
```

```
table1.assign(
    country_Category = pd.Categorical(table1.country)
  )
```

	country	year	cases	population	country_Category
0	Afghanistan	1999	745	19987071	Afghanistan
1	Afghanistan	2000	2666	20595360	Afghanistan
2	Brazil	1999	37737	172006362	Brazil
3	Brazil	2000	80488	174504898	Brazil
4	China	1999	212258	1272915272	China
5	China	2000	213766	1280428583	China

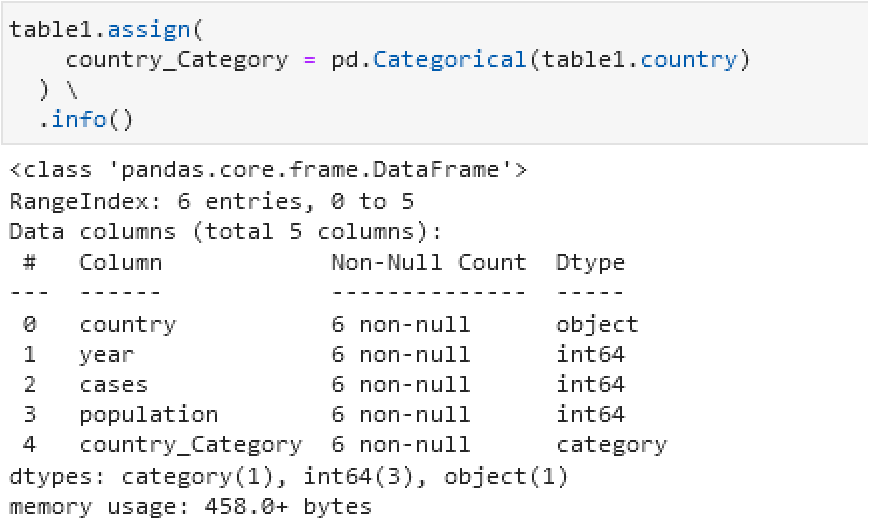


```
table1
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

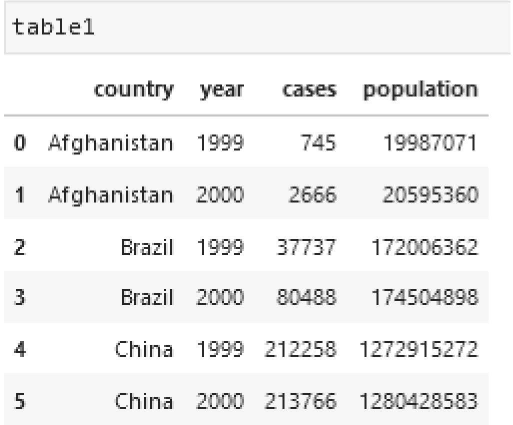
Categorical Columns

- **pd.Categorical()** convert any column into a category representing a categorical variable



```
table1.assign(
    country_Category = pd.Categorical(table1.country)
  ) \
  .info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 5 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   country           6 non-null      object
 1   year              6 non-null      int64
 2   cases             6 non-null      int64
 3   population        6 non-null      int64
 4   country_Category  6 non-null      category
dtypes: category(1), int64(3), object(1)
memory usage: 458.0+ bytes
```

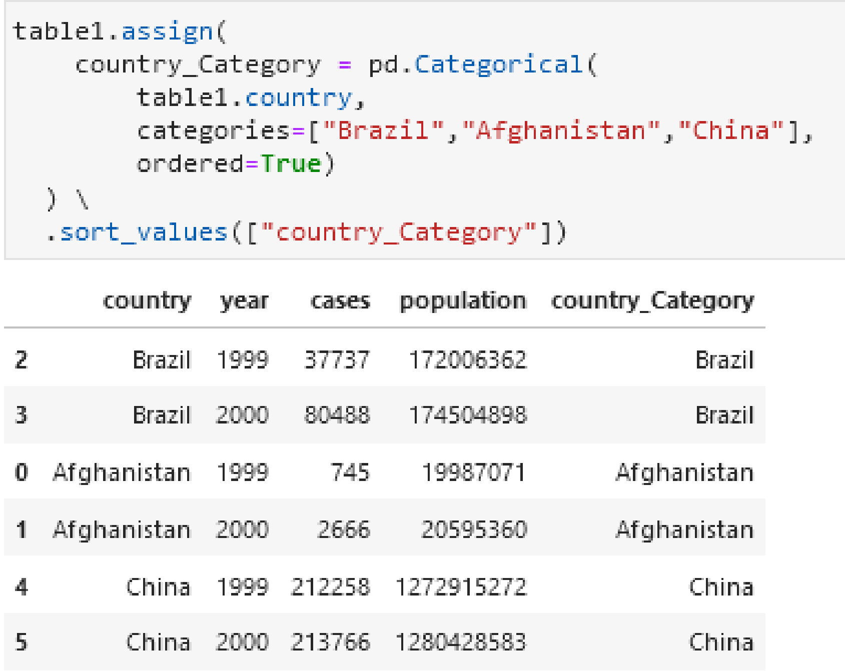


```
table1
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

Categorical Columns

- If we need that the categorical is treated as a ordered categorical column we can use the ‘**ordered**’ param

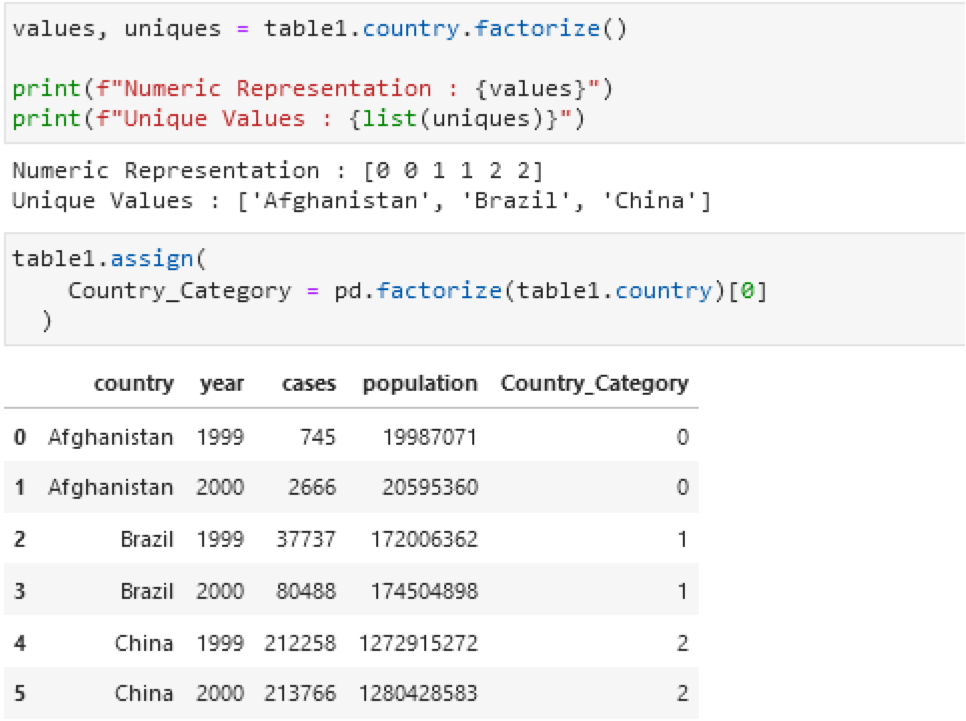


```
table1.assign(
    country_Category = pd.Categorical(
        table1.country,
        categories=["Brazil","Afghanistan","China"],
        ordered=True)
  ) \
  .sort_values(["country_Category"])
```

	country	year	cases	population	country_Category
2	Brazil	1999	37737	172006362	Brazil
3	Brazil	2000	80488	174504898	Brazil
0	Afghanistan	1999	745	19987071	Afghanistan
1	Afghanistan	2000	2666	20595360	Afghanistan
4	China	1999	212258	1272915272	China
5	China	2000	213766	1280428583	China

Factorization

- Another alternative to categorize a column is **factorization** (encode a column with a numerical representation)

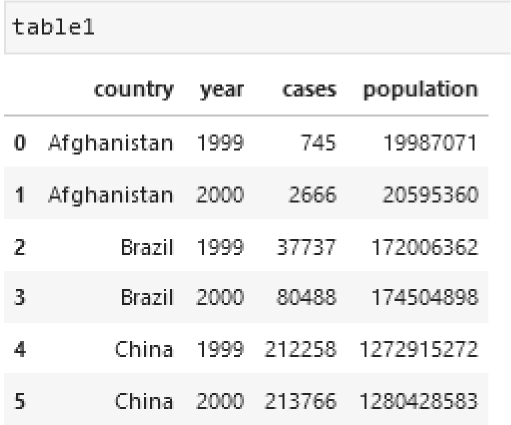


```
values, uniques = table1.country.factorize()

print(f"Numeric Representation : {values}")
print(f"Unique Values : {list(uniques)}")
```

```
Numeric Representation : [0 0 1 1 2 2]
Unique Values : ['Afghanistan', 'Brazil', 'China']
```

```
table1.assign(
    Country_Category = pd.factorize(table1.country)[0]
  )
```

	country	year	cases	population	Country_Category
0	Afghanistan	1999	745	19987071	0
1	Afghanistan	2000	2666	20595360	0
2	Brazil	1999	37737	172006362	1
3	Brazil	2000	80488	174504898	1
4	China	1999	212258	1272915272	2
5	China	2000	213766	1280428583	2



```
table1
```

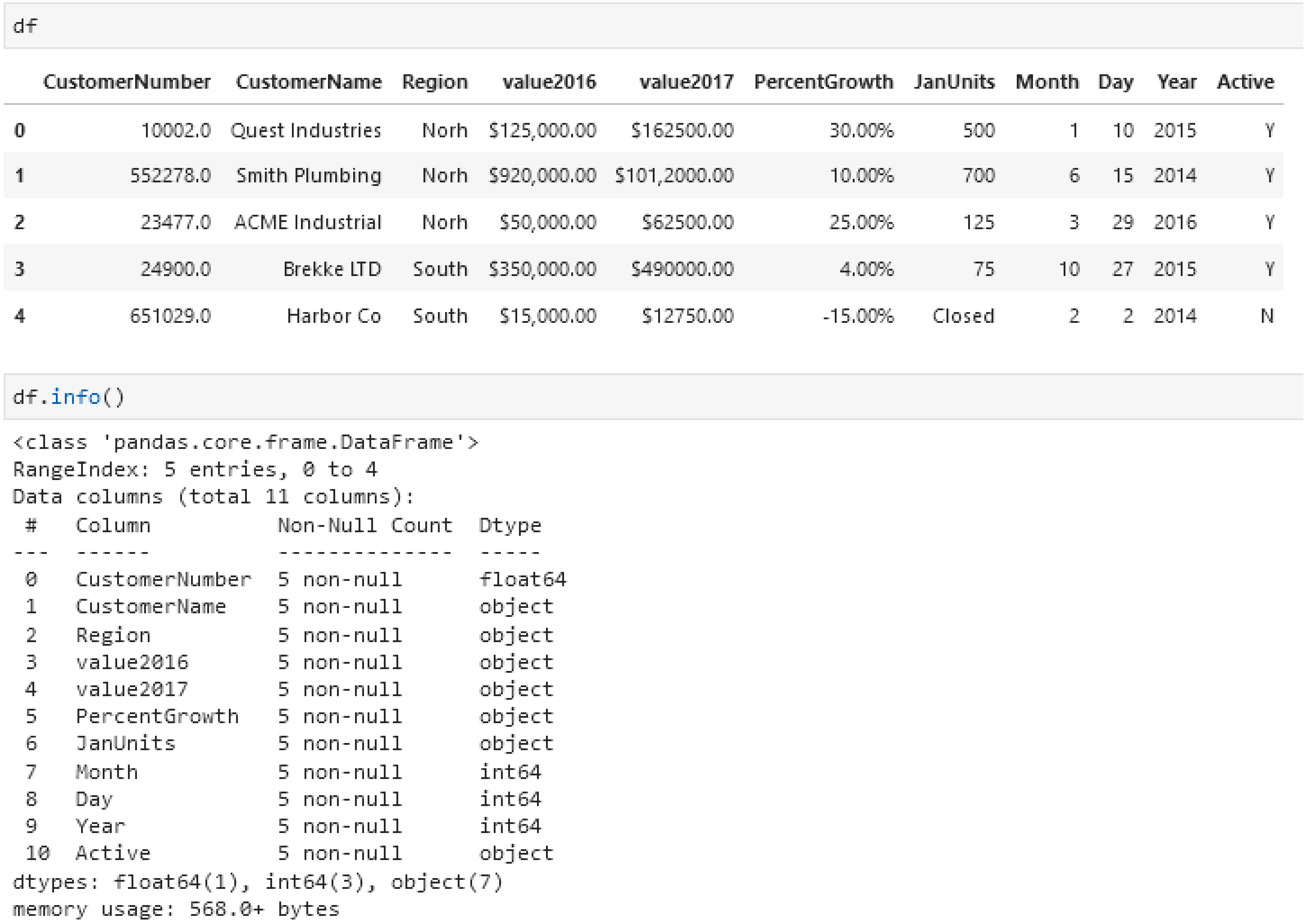
	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

Exercise 10 (1/3)

Clean the ‘sales.csv’ dataset:

- The CustomerNumber is a float64 but it should be an int64
- The value2016 and value2017 columns are stored as objects, not numerical values such as a float64 or int64
- PercentGrowth and JanUnits are also stored as objects not numerical values
- We have Month , Day and Year columns that should be converted to datetime64
- The Active column should be a Boolean
- The Region column should be a category

Exercise 10 (2/3)



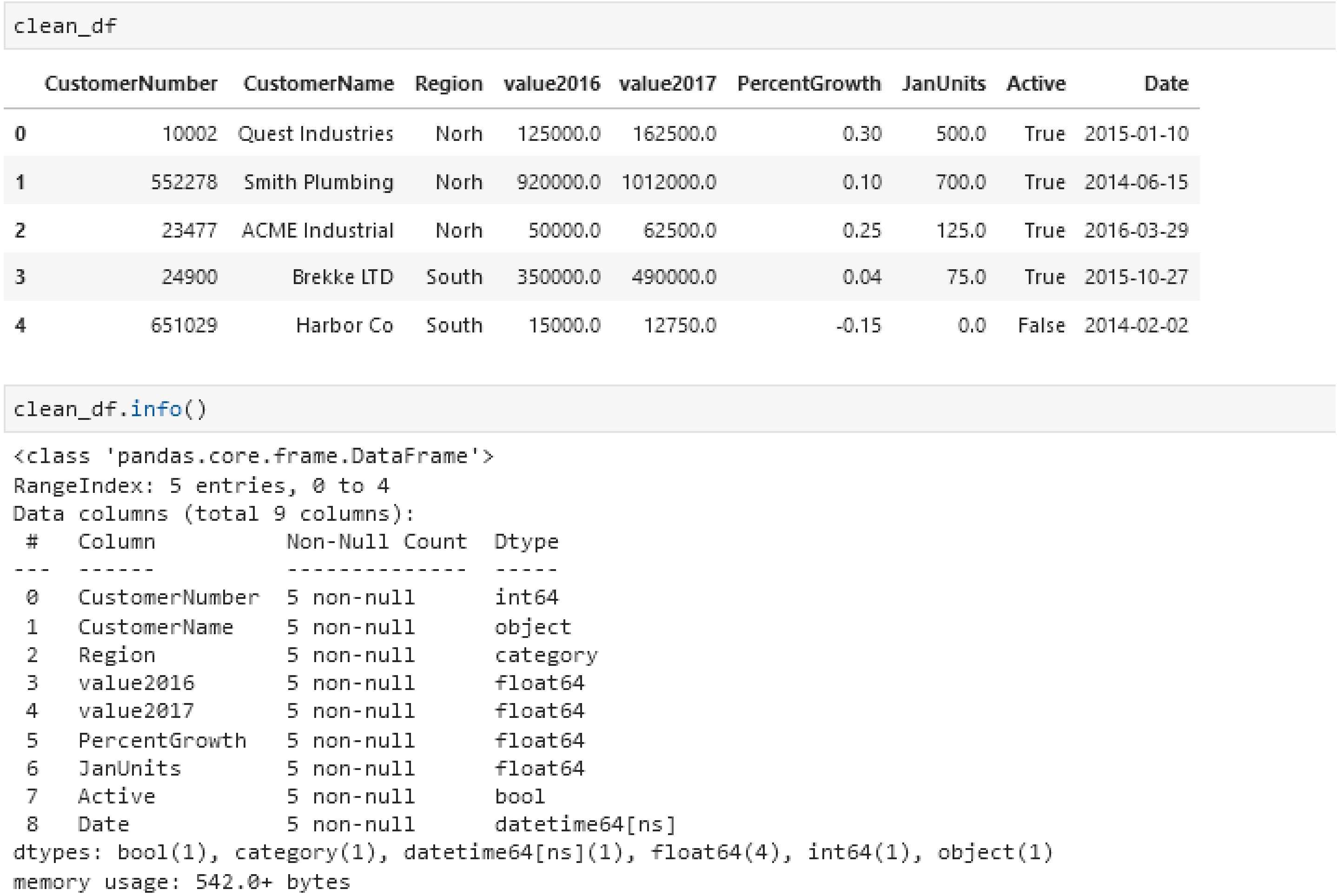
```
df
```

	CustomerNumber	CustomerName	Region	value2016	value2017	PercentGrowth	JanUnits	Month	Day	Year	Active
0	10002.0	Quest Industries	Norh	$125,000.00	$162500.00	30.00%	500	1	10	2015	Y
1	552278.0	Smith Plumbing	Norh	$920,000.00	$101,2000.00	10.00%	700	6	15	2014	Y
2	23477.0	ACME Industrial	Norh	$50,000.00	$62500.00	25.00%	125	3	29	2016	Y
3	24900.0	Brekke LTD	South	$350,000.00	$490000.00	4.00%	75	10	27	2015	Y
4	651029.0	Harbor Co	South	$15,000.00	$12750.00	-15.00%	Closed	2	2	2014	N

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   CustomerNumber  5 non-null      float64
 1   CustomerName    5 non-null      object
 2   Region          5 non-null      object
 3   value2016       5 non-null      object
 4   value2017       5 non-null      object
 5   PercentGrowth   5 non-null      object
 6   JanUnits        5 non-null      object
 7   Month           5 non-null      int64
 8   Day             5 non-null      int64
 9   Year            5 non-null      int64
 10  Active          5 non-null      object
dtypes: float64(1), int64(3), object(7)
memory usage: 568.0+ bytes
```

Exercise 10 (3/3)



```
clean_df
```

	CustomerNumber	CustomerName	Region	value2016	value2017	PercentGrowth	JanUnits	Active	Date
0	10002	Quest Industries	Norh	125000.0	162500.0	0.30	500.0	True	2015-01-10
1	552278	Smith Plumbing	Norh	920000.0	1012000.0	0.10	700.0	True	2014-06-15
2	23477	ACME Industrial	Norh	50000.0	62500.0	0.25	125.0	True	2016-03-29
3	24900	Brekke LTD	South	350000.0	490000.0	0.04	75.0	True	2015-10-27
4	651029	Harbor Co	South	15000.0	12750.0	-0.15	0.0	False	2014-02-02

```
clean_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 9 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   CustomerNumber  5 non-null      int64
 1   CustomerName    5 non-null      object
 2   Region          5 non-null      category
 3   value2016       5 non-null      float64
 4   value2017       5 non-null      float64
 5   PercentGrowth   5 non-null      float64
 6   JanUnits        5 non-null      float64
 7   Active          5 non-null      bool
 8   Date            5 non-null      datetime64[ns]
dtypes: bool(1), category(1), datetime64[ns](1), float64(4), int64(1), object(1)
memory usage: 542.0+ bytes
```

Agenda

- Introduction
- Widening tables
- Narrowing down tables
- Separating columns
- Joining columns
- Missing data
- Dropping duplicates
- Data Types
- Data Formating
- Regex

Data Formating

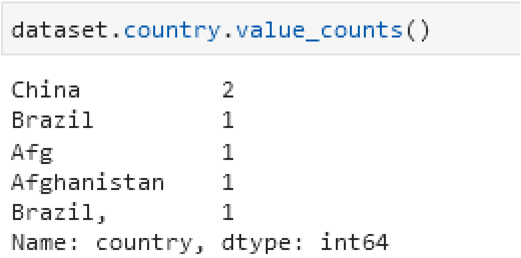
- Data formatting is the process of transforming data into a common format
- We can have different problems. For example:
  - Different values for the same concept

    Example: ‘New York’ & ‘NY’
  - The data is not homogeneous

    Example: ‘91123112’ vs ‘911 231 12’

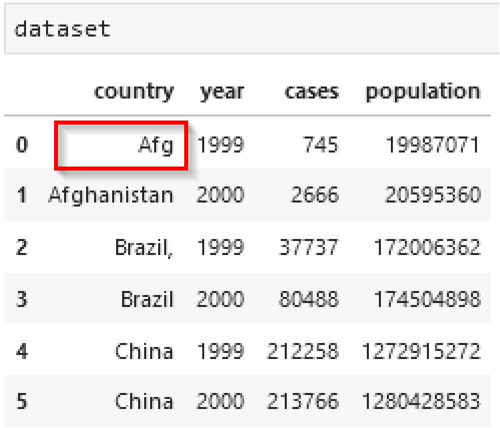
Different values for the same concept

- It may happen that the same concept is represented in different ways
- We can use the **value_counts()** function to list all the values of a column.



```
dataset.country.value_counts()
```

```
China          2
Brazil         1
Afg            1
Afghanistan    1
Brazil,        1
Name: country, dtype: int64
```

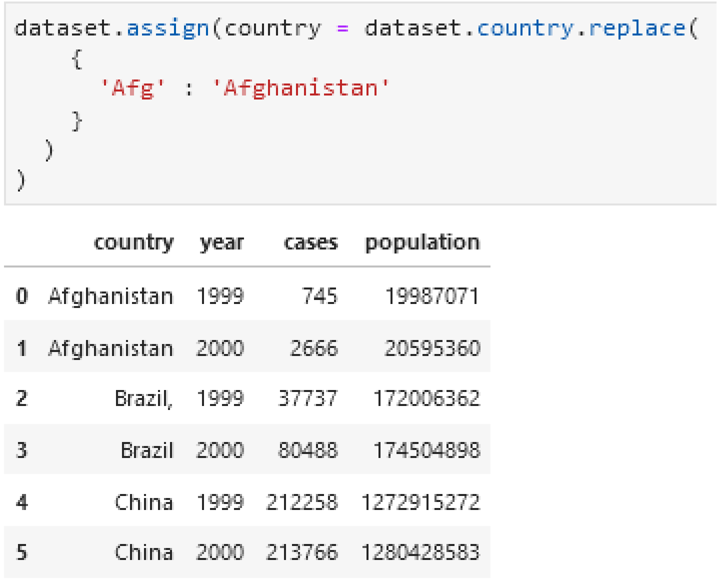


```
dataset
```

	country	year	cases	population
0	Afg	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil,	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

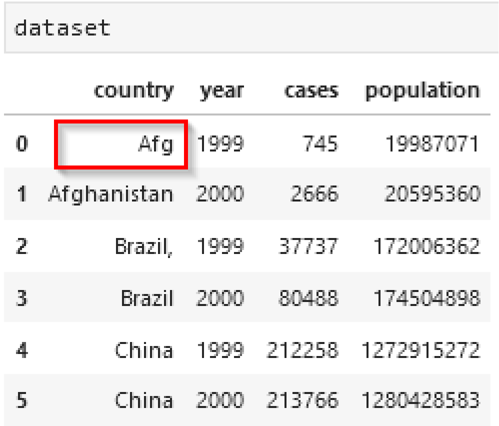
Different values for the same concept

- The **replace()** function is a convenient method to replace values in a column.



```
dataset.assign(country = dataset.country.replace(
    {
      'Afg' : 'Afghanistan'
    }
  )
)
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil,	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

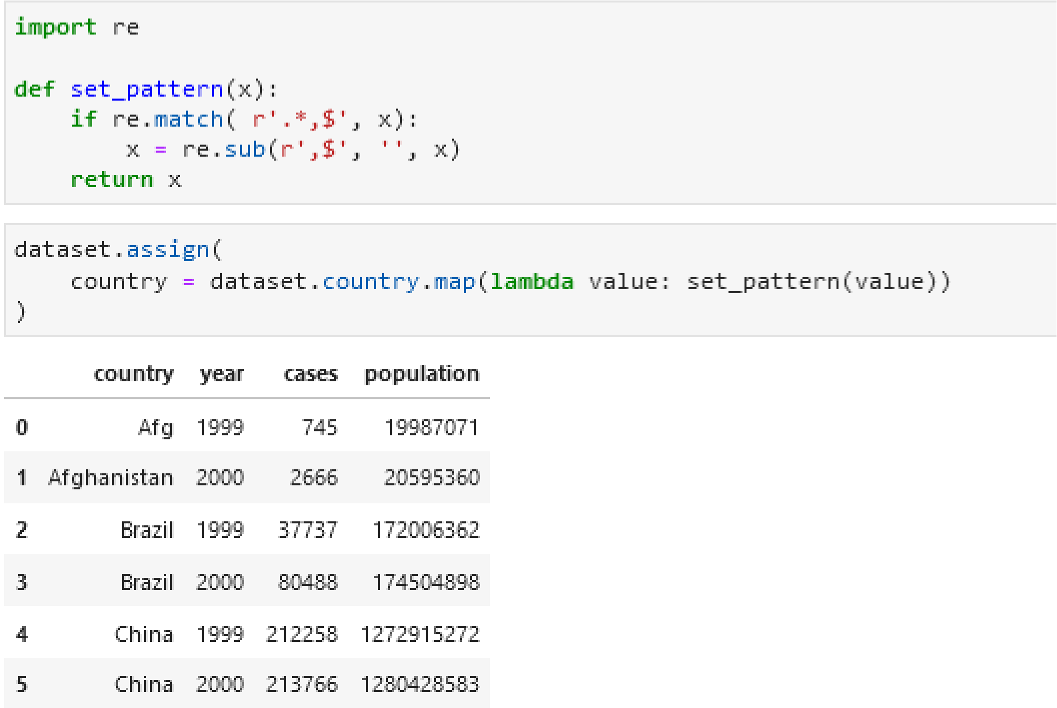


```
dataset
```

	country	year	cases	population
0	Afg	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil,	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

Different values for the same concept

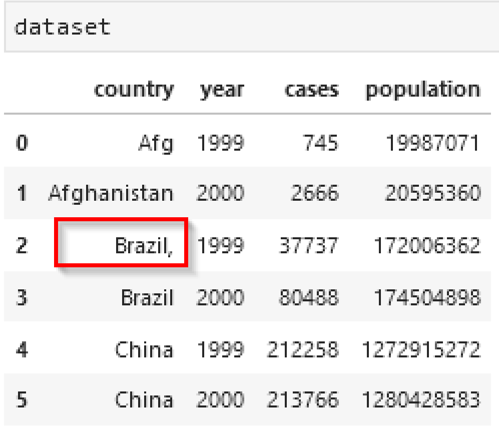
- Another possibility is to use a user function to clean the data …



```
import re

def set_pattern(x):
    if re.match( r'.*,$', x):
        x = re.sub(r',$', '', x)
    return x
```

```
dataset.assign(
    country = dataset.country.map(lambda value: set_pattern(value))
)
```

	country	year	cases	population
0	Afg	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

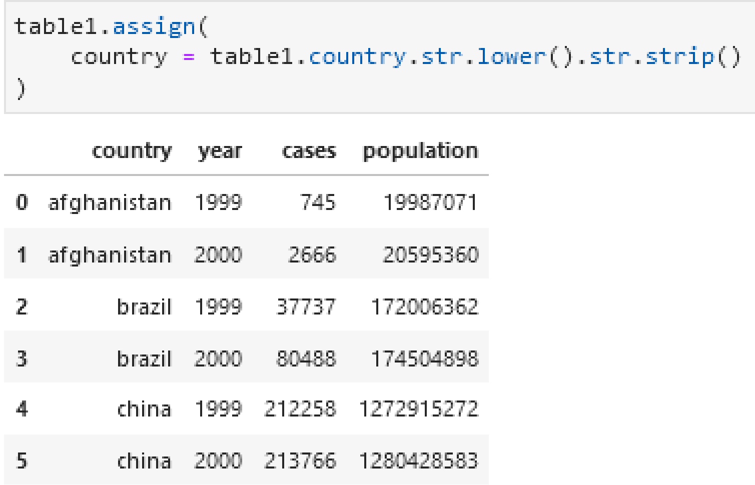


```
dataset
```

	country	year	cases	population
0	Afg	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil,	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

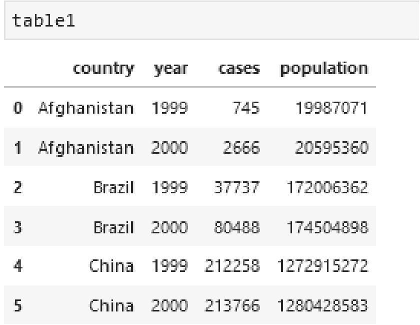
Make the data homogeneous

- This aspect involves numeric and string data
- Text data should have all the same formatting style, such as lower case, or don’t have white spaces at the beginning of string



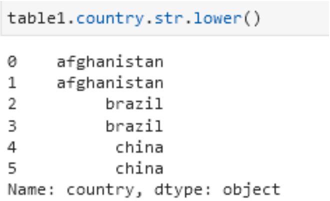
```
table1.assign(
    country = table1.country.str.lower().str.strip()
)
```

	country	year	cases	population
0	afghanistan	1999	745	19987071
1	afghanistan	2000	2666	20595360
2	brazil	1999	37737	172006362
3	brazil	2000	80488	174504898
4	china	1999	212258	1272915272
5	china	2000	213766	1280428583



```
table1
```

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

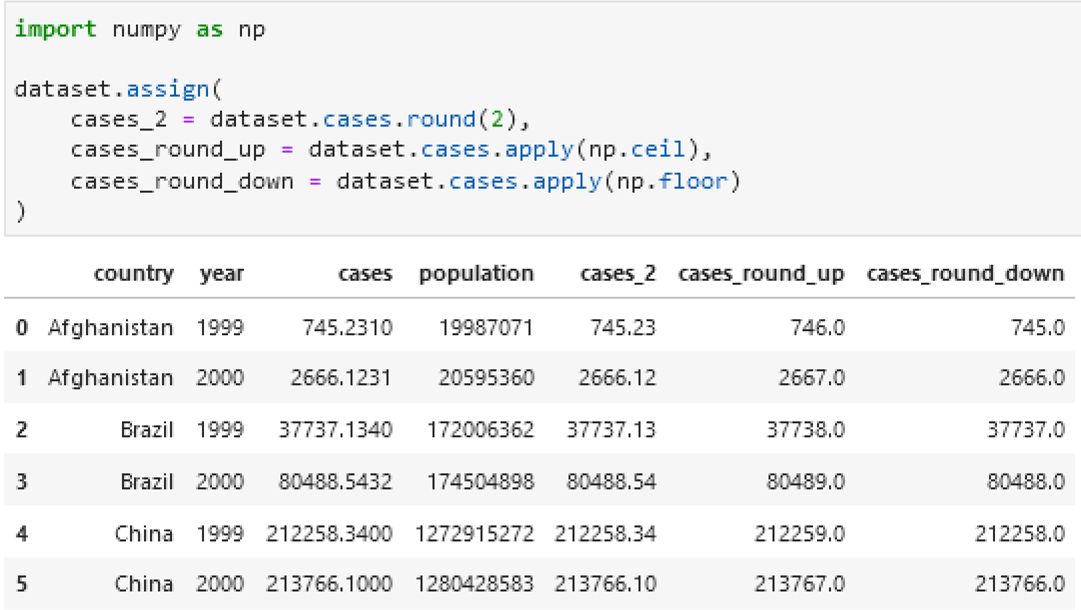


```
table1.country.str.lower()
```

```
0    afghanistan
1    afghanistan
2         brazil
3         brazil
4          china
5          china
Name: country, dtype: object
```

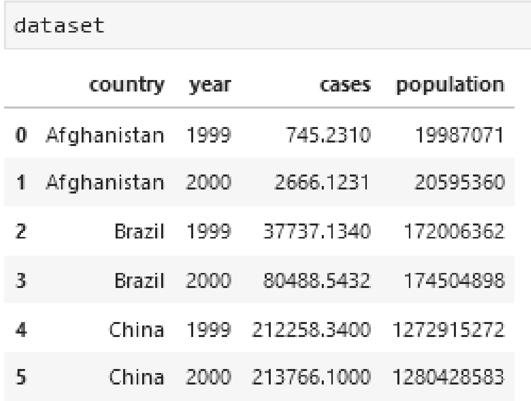
Make the data homogeneous

- Numeric data should have for example the same number of digits after the point.
- Other techniques to make homogeneous numeric data include Round up or Round down



```
import numpy as np

dataset.assign(
    cases_2 = dataset.cases.round(2),
    cases_round_up = dataset.cases.apply(np.ceil),
    cases_round_down = dataset.cases.apply(np.floor)
)
```

	country	year	cases	population	cases_2	cases_round_up	cases_round_down
0	Afghanistan	1999	745.2310	19987071	745.23	746.0	745.0
1	Afghanistan	2000	2666.1231	20595360	2666.12	2667.0	2666.0
2	Brazil	1999	37737.1340	172006362	37737.13	37738.0	37737.0
3	Brazil	2000	80488.5432	174504898	80488.54	80489.0	80488.0
4	China	1999	212258.3400	1272915272	212258.34	212259.0	212258.0
5	China	2000	213766.1000	1280428583	213766.10	213767.0	213766.0



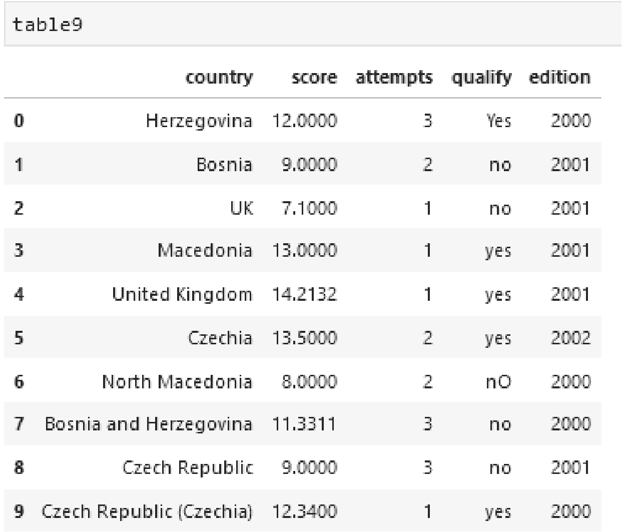
```
dataset
```

	country	year	cases	population
0	Afghanistan	1999	745.2310	19987071
1	Afghanistan	2000	2666.1231	20595360
2	Brazil	1999	37737.1340	172006362
3	Brazil	2000	80488.5432	174504898
4	China	1999	212258.3400	1272915272
5	China	2000	213766.1000	1280428583

Exercise 11

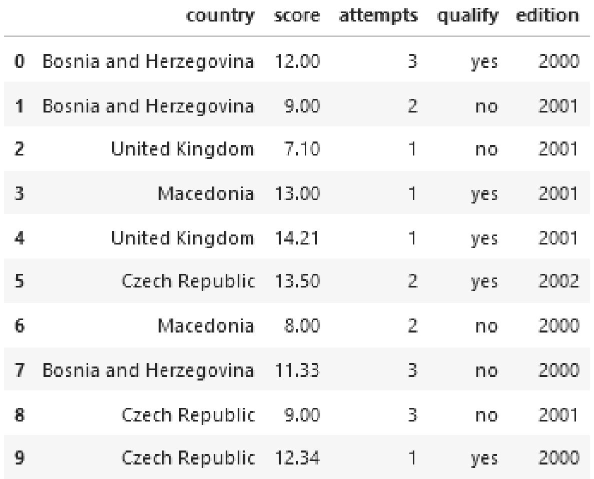
Clean the table9 dataset:

- On the field country, make sure that the same country always has the same value
- Make the filed score homogeneous (2 decimals)
- Make the filed qualify homogeneous (lower case)



```
table9
```

	country	score	attempts	qualify	edition
0	Herzegovina	12.0000	3	Yes	2000
1	Bosnia	9.0000	2	no	2001
2	UK	7.1000	1	no	2001
3	Macedonia	13.0000	1	yes	2001
4	United Kingdom	14.2132	1	yes	2001
5	Czechia	13.5000	2	yes	2002
6	North Macedonia	8.0000	2	nO	2000
7	Bosnia and Herzegovina	11.3311	3	no	2000
8	Czech Republic	9.0000	3	no	2001
9	Czech Republic (Czechia)	12.3400	1	yes	2000



	country	score	attempts	qualify	edition
0	Bosnia and Herzegovina	12.00	3	yes	2000
1	Bosnia and Herzegovina	9.00	2	no	2001
2	United Kingdom	7.10	1	no	2001
3	Macedonia	13.00	1	yes	2001
4	United Kingdom	14.21	1	yes	2001
5	Czech Republic	13.50	2	yes	2002
6	Macedonia	8.00	2	no	2000
7	Bosnia and Herzegovina	11.33	3	no	2000
8	Czech Republic	9.00	3	no	2001
9	Czech Republic	12.34	1	yes	2000

Agenda

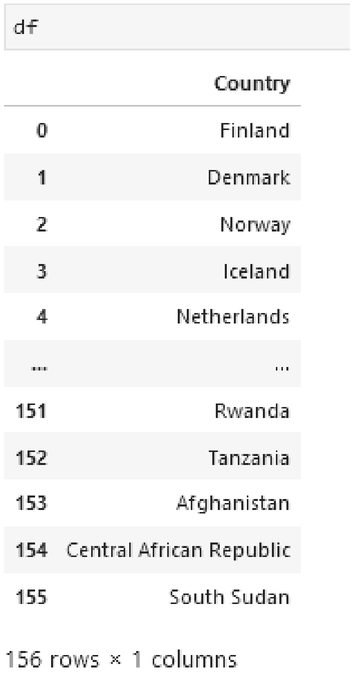
- Introduction
- Widening tables
- Narrowing down tables
- Separating columns
- Joining columns
- Missing data
- Dropping duplicates
- Data Types
- Data Formating
- Regex

Text Data & Regex

- ~80% of data is Text
- Pandas provides a very rich set of functions to manipulate strings (**str** prefix functions)
- There are several **str** methods which accept a regex
- These methods works on the same line as Pythons **re** module
- This will help us to:
  - Check if a text meets a certain pattern
  - Replace certain text pattern with another string
  - Extract information from texts

Match Patterns

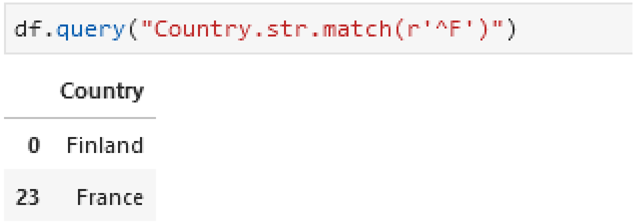
- Check if a text meets a certain pattern will help us, for instance, to find the names starting with a particular character or search for a pattern within a dataframe column.



```
df
```

	Country
0	Finland
1	Denmark
2	Norway
3	Iceland
4	Netherlands
...	...
151	Rwanda
152	Tanzania
153	Afghanistan
154	Central African Republic
155	South Sudan

156 rows × 1 columns

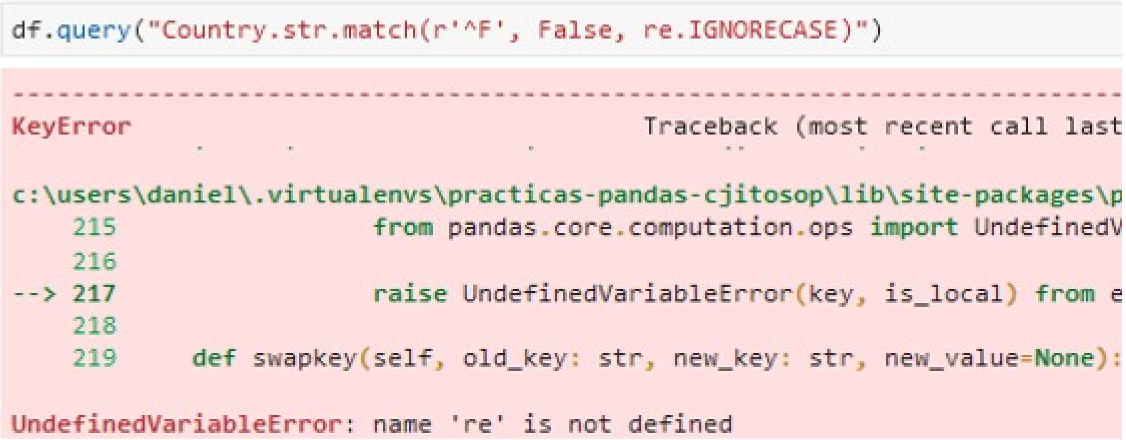


```
df.query("Country.str.match(r'^F')")
```

	Country
0	Finland
23	France

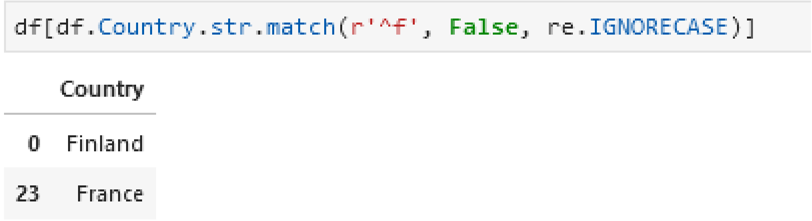
Match Patterns

- If we want to use flags with our regex expression we cannot use **query()** function



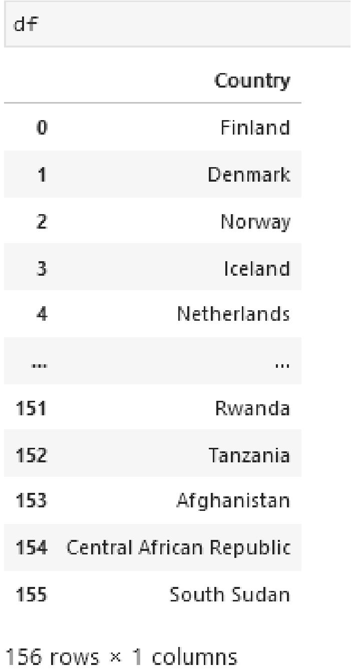
```
df.query("Country.str.match(r'^F', False, re.IGNORECASE)")
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last

c:\users\daniel\.virtualenvs\practicas-pandas-cjitosop\lib\site-packages\p
    215                 from pandas.core.computation.ops import UndefinedV
    216 
--> 217                 raise UndefinedVariableError(key, is_local) from e
    218 
    219     def swapkey(self, old_key: str, new_key: str, new_value=None):

UndefinedVariableError: name 're' is not defined
```



```
df[df.Country.str.match(r'^f', False, re.IGNORECASE)]
```

	Country
0	Finland
23	France



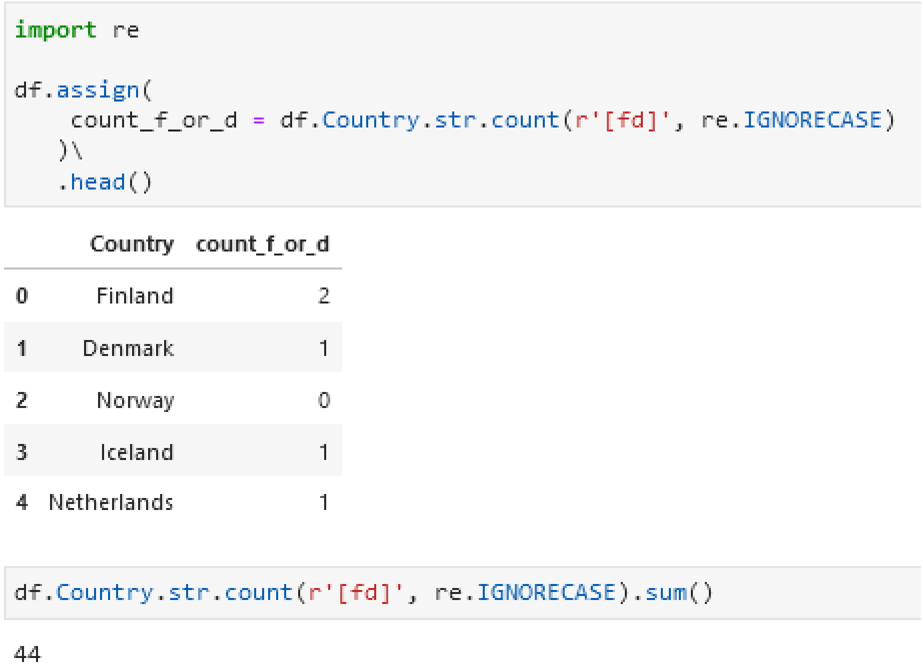
```
df
```

	Country
0	Finland
1	Denmark
2	Norway
3	Iceland
4	Netherlands
...	...
151	Rwanda
152	Tanzania
153	Afghanistan
154	Central African Republic
155	South Sudan

156 rows × 1 columns

Match Patterns

- Sometimes it is very useful to count the number of times a certain pattern appears in a text.

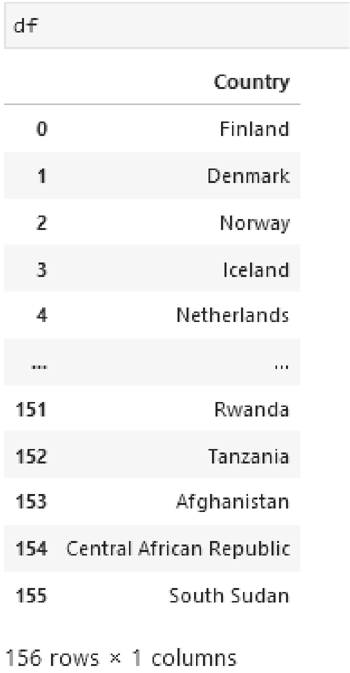


```
import re

df.assign(
    count_f_or_d = df.Country.str.count(r'[fd]', re.IGNORECASE)
   )\
   .head()
```

	Country	count_f_or_d
0	Finland	2
1	Denmark	1
2	Norway	0
3	Iceland	1
4	Netherlands	1

```
df.Country.str.count(r'[fd]', re.IGNORECASE).sum()
```

```
44
```



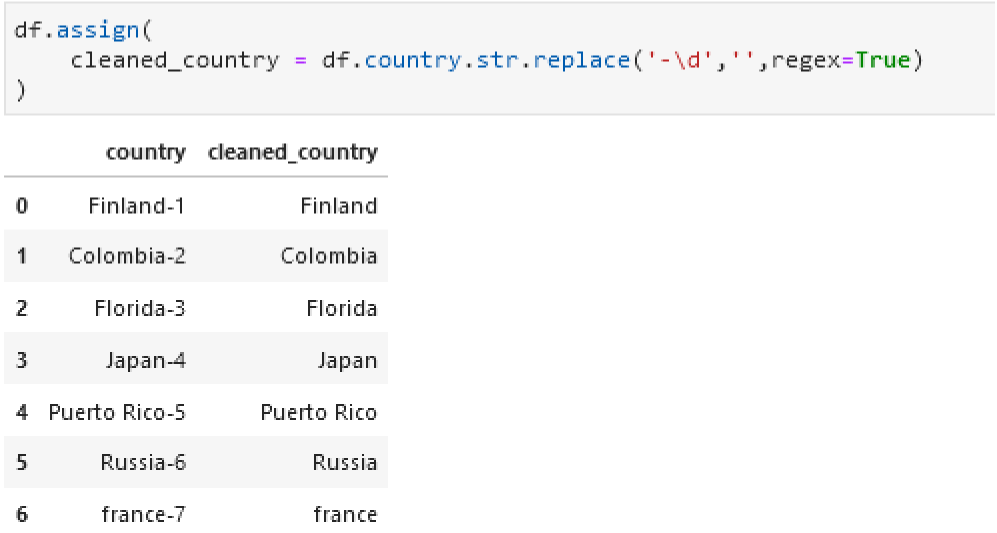
```
df
```

	Country
0	Finland
1	Denmark
2	Norway
3	Iceland
4	Netherlands
...	...
151	Rwanda
152	Tanzania
153	Afghanistan
154	Central African Republic
155	South Sudan

156 rows × 1 columns

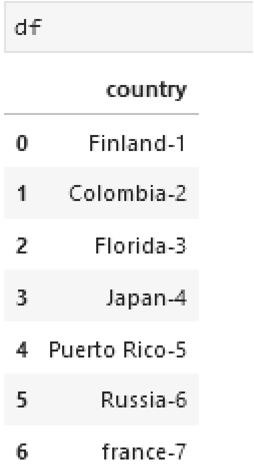
Replacing Text

- Replacing certain text pattern with another string help us to make our data homogeneous



```
df.assign(
    cleaned_country = df.country.str.replace('-\d','',regex=True)
)
```

	country	cleaned_country
0	Finland-1	Finland
1	Colombia-2	Colombia
2	Florida-3	Florida
3	Japan-4	Japan
4	Puerto Rico-5	Puerto Rico
5	Russia-6	Russia
6	france-7	france

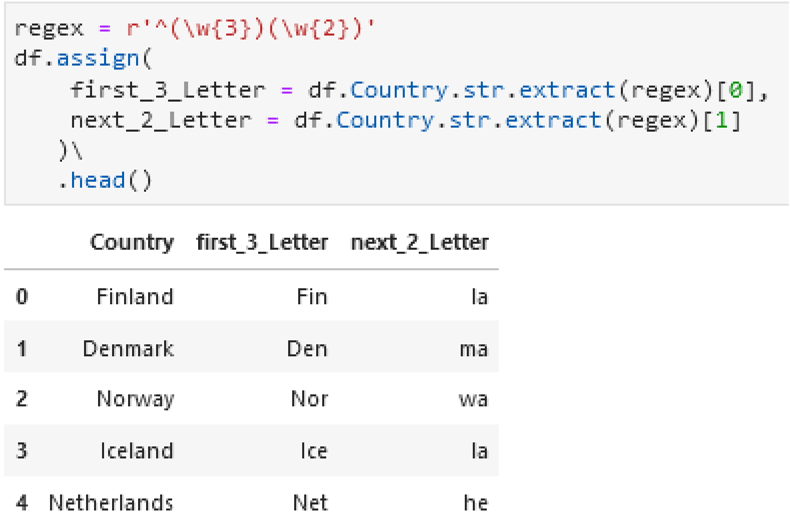


```
df
```

	country
0	Finland-1
1	Colombia-2
2	Florida-3
3	Japan-4
4	Puerto Rico-5
5	Russia-6
6	france-7

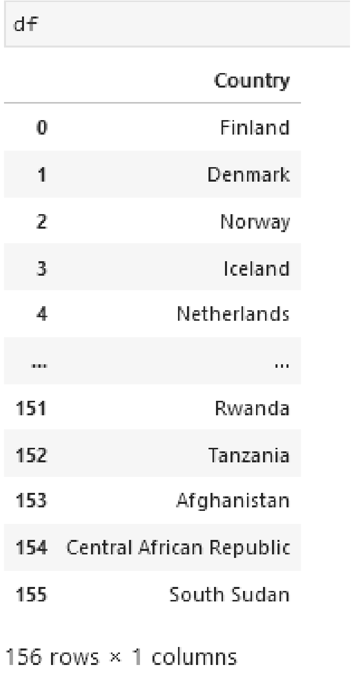
Extracting information

- Extracting information from texts is extremely common in our work as data scientists
- We have to use the capture groups



```
regex = r'^(\w{3})(\w{2})'
df.assign(
    first_3_Letter = df.Country.str.extract(regex)[0],
    next_2_Letter = df.Country.str.extract(regex)[1]
   )\
   .head()
```

	Country	first_3_Letter	next_2_Letter
0	Finland	Fin	la
1	Denmark	Den	ma
2	Norway	Nor	wa
3	Iceland	Ice	la
4	Netherlands	Net	he



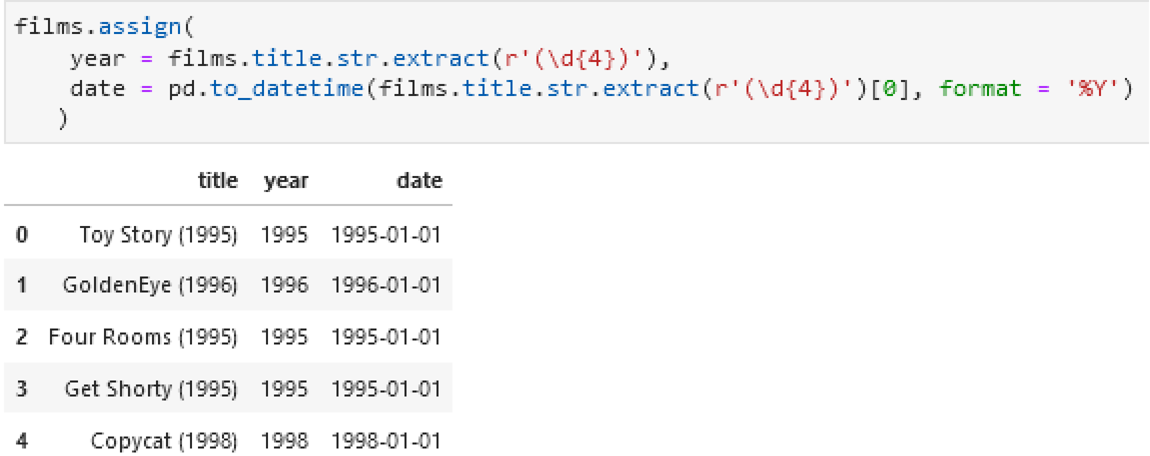
```
df
```

	Country
0	Finland
1	Denmark
2	Norway
3	Iceland
4	Netherlands
...	...
151	Rwanda
152	Tanzania
153	Afghanistan
154	Central African Republic
155	South Sudan

156 rows × 1 columns

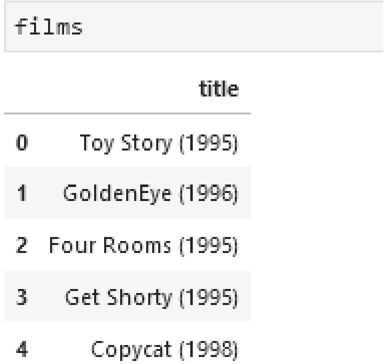
Extracting information

- It will allow us, for example, to extract the dates from a text



```
films.assign(
    year = films.title.str.extract(r'(\d{4})'),
    date = pd.to_datetime(films.title.str.extract(r'(\d{4})')[0], format = '%Y')
   )
```

	title	year	date
0	Toy Story (1995)	1995	1995-01-01
1	GoldenEye (1996)	1996	1996-01-01
2	Four Rooms (1995)	1995	1995-01-01
3	Get Shorty (1995)	1995	1995-01-01
4	Copycat (1998)	1998	1998-01-01

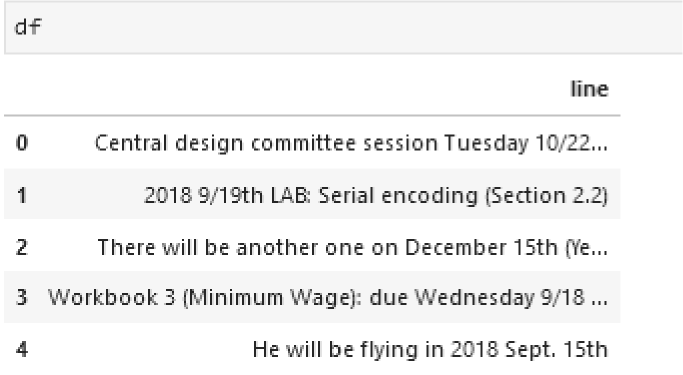


```
films
```

	title
0	Toy Story (1995)
1	GoldenEye (1996)
2	Four Rooms (1995)
3	Get Shorty (1995)
4	Copycat (1998)

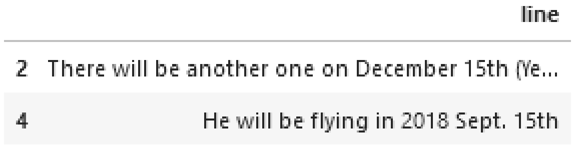
Exercise 12

- Load the file ‘text.txt’ in Pandas
- Search the rows with ‘December’ or ‘Sept.’ literals
- Replace ‘December ’ by ‘12/’ and ‘Sept. ’ by ‘9/’
- Create a new column with the date extracted from every line

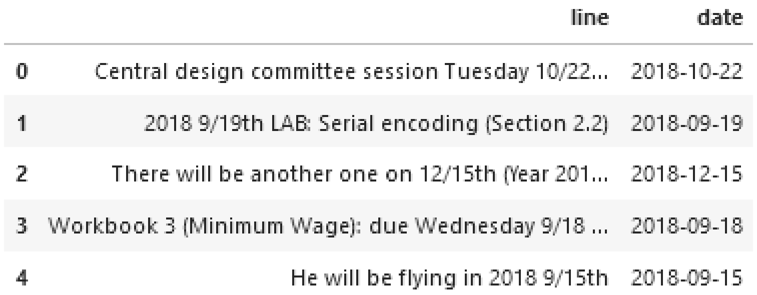


```
df
```

	line
0	Central design committee session Tuesday 10/22...
1	2018 9/19th LAB: Serial encoding (Section 2.2)
2	There will be another one on December 15th (Ye...
3	Workbook 3 (Minimum Wage): due Wednesday 9/18 ...
4	He will be flying in 2018 Sept. 15th

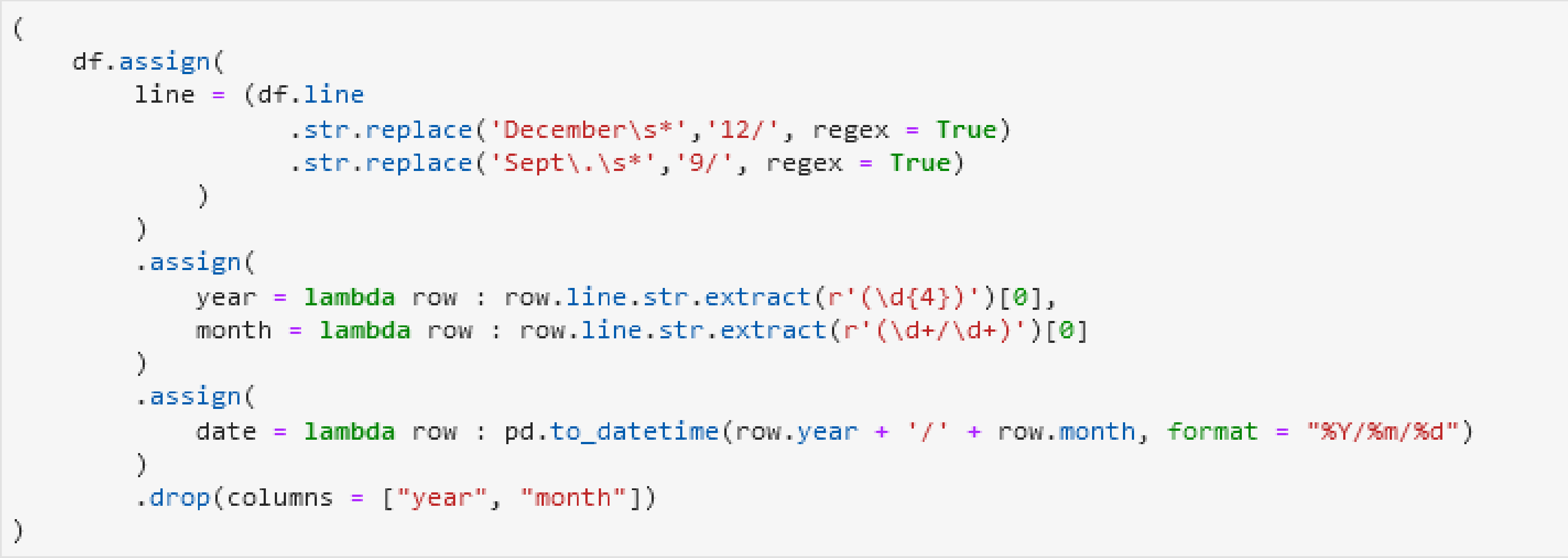


	line
2	There will be another one on December 15th (Ye...
4	He will be flying in 2018 Sept. 15th



	line	date
0	Central design committee session Tuesday 10/22...	2018-10-22
1	2018 9/19th LAB: Serial encoding (Section 2.2)	2018-09-19
2	There will be another one on 12/15th (Year 201...	2018-12-15
3	Workbook 3 (Minimum Wage): due Wednesday 9/18 ...	2018-09-18
4	He will be flying in 2018 9/15th	2018-09-15

Exercise 12



```
(
    df.assign(
        line = (df.line
                  .str.replace('December\s*','12/', regex = True)
                  .str.replace('Sept\.\s*','9/', regex = True)
            )
        )
        .assign(
            year = lambda row : row.line.str.extract(r'(\d{4})')[0],
            month = lambda row : row.line.str.extract(r'(\d+/\d+)')[0]
        )
        .assign(
            date = lambda row : pd.to_datetime(row.year + '/' + row.month, format = "%Y/%m/%d")
        )
        .drop(columns = ["year", "month"])
)
```

THANKS FOR YOUR ATTENTION

Daniel Villanueva Jiménez
daniel.villanueva@immune.institute

@dvillaj