



PYTHON & PANDAS

TIPS & TRICKS



pandas

30+ Data Science Tips and Tricks
to make your life easier

Zoumana Keita

ZoumDataScience.com



1. Remove duplicates from a list

When trying to remove duplicates from a list, you might attempt to use the **for** loop approach.

This works but is inefficient ✗ when dealing with very large data.

Instead, use **set()** ✓ which natively does not accept duplicates.

```
cities = ["Abidjan", "Bamako", "Dakar", "Abidjan", "Dakar"]

# First Approach
unique_cities = []

for city in cities:
    if city not in unique_cities:
        unique_cities.append(city)

print(unique_cities)

# Second Approach
unique_cities = list(set(cities))
print(unique_cities)
```

Don't do this ✗

Do this ✓

['Abidjan', 'Bamako', 'Dakar']

Using **set()** to remove duplicates from lists is a great approach.

⚠️ But be careful with using it as it will NOT ✗ preserve the original order. Only use it when you don't care about the order of the elements in your list.

Instead, use **dict.fromkeys()** ✓ to preserve the original order.

```
countries = ["Ivory Coast", "Mali", "Senegal", "Ivory Coast",
             "Benin", "Mali", "Senegal"]
```

=====

Duplicate removal process

=====

1- IF Order DOES NOT matter

```
list(set(countries)) → ['Mali', 'Senegal', 'Ivory Coast', 'Benin']
```

Didnt keep the
original order ✗

2- IF Order MATTERS

```
['Ivory Coast', 'Mali', 'Senegal', 'Benin']
```

```
list(dict.fromkeys(countries))
```

Kept the
original order ✓

2. Check if an element exists in a list

When trying to **check if an item exists in a list**, you might attempt to use the **for** loop and **if** condition approach.

This works but is inefficient ✗ when dealing with very large data.

Instead, use the **in** ✓ approach which natively returns a boolean.

```
cities = ["Abidjan", "Bamako", "Dakar", "Abidjan", "Dakar"]

# First Approach
search_criteria = "Bamako"

result = False

for city in cities:

    if(city == search_criteria):

        result = True

print(result) → True
```

```
# Second Approach
result = search_criteria in cities

print(result) → True
```

Don't do this ✗

Do this ✓
Natively returns a boolean

3. Get the N largest and smallest values in a list

The maximum and minimum values of a list in Python can be found using the **max()** and **min()** functions respectively.

However, when it comes to getting the **N largest** or **smallest** values of you might think of a two-way approach:

- 1 Sort the list in decreasing or increasing order.
- 2 Retrieve the N largest or smallest values.

Good strategy, BUT not efficient ❌ when dealing with large data.

- ✓ Instead, you can use the **nlargest** and **nsmallest** functions from the built-in Python module **heapq** which is fast 💡 and memory efficient 👍

Relevant functions

```
from heapq import nlargest, nsmallest  
my_list = [45, 10, 436, 89, 100, 43546, 855]
```

```
# Get the 3 largest values  
print(nlargest(3, my_list))
```

Call for the **3 largest** values

[43546, 855, 436]

```
# Get the 3 smallest values  
print(nsmallest(3, my_list))
```

Call for the **3 smallest** values

[10, 45, 89]

4. Describe numerical and categorical columns

Applying the **describe()** function without a parameter naturally returns statistics related to numerical columns only.

This restricts **🚫** our understanding of the data set since most of the time we deal with categorical columns as well.

-  To solve this issue, you can proceed with a two-way approach:
 - 1** Use **describe()** for numerical columns.
 - 2** Set the parameter **include=[object]** to provide information about categorical ones.

```
import pandas as pd
candidates_df = pd.read_csv("candidates.csv")

# Data column types
candidates_df.dtypes
```

→

Name	object
Degree	object
From	object
Years_exp	int64
From_office(min)	int64
Application_date	datetime64[ns]
dtype:	object

```
# Statistics of numerical columns
display(candidates_df.describe())
```

```
# Statistics of categoricals
display(candidates_df.describe(include=['object']))
```

	Years_exp	From_office(min)
count	6.000000	6.000000
mean	2.833333	84.000000
std	1.722401	29.223278
min	0.000000	34.000000
25%	2.250000	76.250000
50%	3.000000	87.500000
75%	3.750000	98.750000
max	5.000000	120.000000

Default result

Including "categorical"

	Name	Degree	From	Application_date
count	6	6	6	6
unique	6	3	5	6
top	Ismael	Master	Abidjan	11/17/2022
freq	1	3	2	1

5. Avoid for loops when creating new columns

When working with Pandas dataframes, creating new columns from existing ones is mainly part of the process.

The way these columns are created can affect the efficiency of the overall computation time . Some may use loops to generate those derived columns.

However, this might not be the right approach  because of the time complexity , especially when working with large data.

- Adopting the vectorization approach is much better.



```
import pandas as pd  
import time
```

Data with 16598 Rows

```
game_df = pd.read_csv("data/vgsales.csv")
```

```
# OPTION 1: FOR LOOP
```

```
begin = time.time()
```

Avoid this

```
for index, row in game_df.iterrows():  
    # Create new column  
    game_df.loc[index, 'Sum_NA_EU'] = row["NA_Sales"] + row["EU_Sales"]
```

```
print(time.time() - begin) ➔ ⏳ 2.63 seconds 😞
```

```
# OPTION 2: VECTORIZATION
```

```
begin = time.time()
```

Do this

```
game_df["Sum_NA_EU"] = game_df["NA_Sales"] + game_df["EU_Sales"]
```

```
print(time.time() - begin) ➔ ⏳ 0.0006 seconds ✨✨✨ 😊
```

```
display(game_df.head())
```

New column

Genre	Publisher	NA_Sales	EU_Sales	JP_Sales	Other_Sales	Global_Sales	Sum_NA_EU
Sports	Nintendo	41.49	29.02	3.77	8.46	82.74	70.51
Platform	Nintendo	29.08	3.58	6.81	0.77	40.24	32.66
Racing	Nintendo	15.85	12.88	3.79	3.31	35.82	28.73
Sports	Nintendo	15.75	11.01	3.28	2.96	33.00	26.76
Role-Playing	Nintendo	11.27	8.89	10.22	1.00	31.37	20.16

6. Save a subset of Pandas columns

Sometimes we are interested in saving only a subset of columns from the original data frame rather than the whole data.

One way of doing that is to create a new data frame with the columns of interest.

But, this approach adds another layer of complexity



- ✓ This issue can be solved by specifying the columns argument.

The screenshot shows a Jupyter Notebook cell with the following code:

```
import pandas as pd  
  
candidates_df = pd.read_csv("candidates.csv")  
  
# Columns of interest  
cols_subset = ["Name", "Degree", "From"]  
  
# Create new file  
candidates_df.to_csv("column_subset.csv", columns = cols_subset, index=False)
```

A yellow callout points to the first line of code: "Original data with all the columns". A green callout points to the second line of code: "Columns to be considered". Below the code, two tables are shown side-by-side. The left table, with a yellow border, contains all the original columns: Name, Degree, From, Years_exp, From_office(min), and Application_date. The right table, with a green border, contains only the three columns specified in the code: Name, Degree, and From.

	Name	Degree	From	Years_exp	From_office(min)	Application_date
0	Aida	Master	Abidjan	2	120	11/17/2022
1	Mamadou	Master	Dakar	3	95	09/23/2022
2	Ismael	Bachelor	Bamako	0	75	12/2/2021
3	Aicha	PhD	Abidjan	5	80	08/25/2022
4	Fatou	Master	Konakry	4	100	01/07/2022
5	Khalil	PhD	Lomé	3	34	12/26/2022

	Name	Degree	From
0	Aida	Master	Abidjan
1	Mamadou	Master	Dakar
2	Ismael	Bachelor	Bamako
3	Aicha	PhD	Abidjan
4	Fatou	Master	Konakry
5	Khalil	PhD	Lomé

7. Convert Tabular data from the webpage into Pandas Dataframe

If you want to extract tables from a webpage  as Pandas Dataframes, you can use the `read_html()` function of Pandas.

- It returns a list of all the tables from the webpage.

```
import pandas as pd

URL = "https://en.wikipedia.org/wiki/World_energy_supply_and_consumption"

world_energy_consumptions = pd.read_html(URL)

print(f"Number of Tables: {len(world_energy_consumptions)}")

world_energy_consumptions[6]

Number of Tables: 8
Table from the 6th index
Final Pandas Dataframe
```

Country	Fuel Mtoe	of which renewable	Electricity Mtoe	of which renewable
Germany	156	10%	45	46%
France	100	12%	38	21%
United Kingdom	95	5%	26	40%
Italy	87	9%	25	39%
Spain	60	10%	21	43%
Poland	58	12%	12	16%
Ukraine	38	5%	10	12%
Netherlands	36	4%	9	16%
Belgium	26	8%	7	23%
Sweden	20	35%	11	72%
Austria	20	19%	5	86%
Romania	19	20%	4	57%
Finland	18	34%	7	39%
Portugal	11	20%	4	67%
Denmark	11	15%	3	71%
Norway	8	16%	10	100%

Country	Fuel Mtoe	of which renewable	Electricity Mtoe	of which renewable,1
Germany	156	10%	45	46%
France	100	12%	38	21%
United Kingdom	95	5%	26	40%
Italy	87	9%	25	39%
Spain	60	10%	21	43%
Poland	58	12%	12	16%
Ukraine	38	5%	10	12%
Netherlands	36	4%	9	16%
Belgium	26	8%	7	23%
Sweden	20	35%	11	72%
Austria	20	19%	5	86%
Romania	19	20%	4	57%
Finland	18	34%	7	39%
Portugal	11	20%	4	67%
Denmark	11	15%	3	71%
Norway	8	16%	10	100%

8. Replace values from a dataframe based on conditions

If you want to replace values from a dataframe based on conditions

- ✓ You can use the built-in **mask()** function from Pandas.

The screenshot shows a Jupyter Notebook cell with the following code:

```
# Import libraries
import pandas as pd
import numpy as np

# Create sample dataframe
df = pd.DataFrame({'A': [4, 7, -6, 2, -9],
                    'B': [12, -7, 7, -6, 23],
                    'C': [54, -21, 34, 32, -12]})

# Replace negative values in all columns with NaN
df.mask(df < 0, np.Nan)
```

An orange arrow labeled "Original data" points from the original DataFrame on the right to the code. Below the code, three numbered circles (1, 2, 3) point to three colored boxes:

- Yellow box: "Apply mask"
- Green box: "Condition"
- Purple box: "Replacement value"

The word "Result" is centered below the numbered circles. At the bottom, the resulting DataFrame is shown:

	A	B	C
0	4.0	12.0	54.0
1	7.0	NaN	NaN
2	NaN	7.0	34.0
3	2.0	NaN	32.0
4	NaN	23.0	NaN

9. Apply colors to your dataframe

- ✓ **Pandas.style** is a built-in module that provides a high-level interface for styling your dataframe.

This can be helpful to quickly find some information JUST by looking at your dataframe.

The screenshot shows a Jupyter Notebook cell with the following code:

```
import pandas as pd
import numpy as np

# create a sample dataframe
df = pd.DataFrame({'A': [4, 7, -6],
                    'B': [12, -7, 7],
                    'C': [54, -21, -34]})

# Function to apply RED color to negative and GREEN to others
def color_negative_to_red(val):
    color = 'red' if val < 0 else 'green'
    return 'color: %s' % color

# After applying the function
df.style.applymap(color_negative_to_red)
```

Annotations and callouts:

- Original data**: An orange arrow points from the text "Original data" to the original DataFrame on the right.
- Your dataframe**: A yellow box contains the line `df.style`.
- Styling interface**: A green box contains the method `.applymap()`.
- Applies custom function**: A pink box contains the function definition `color_negative_to_red`.
- Custom function**: A red box contains the condition in the function definition `if val < 0`.

Below the code, four numbered circles (1, 2, 3, 4) point to the resulting styled DataFrame:

- 1: Points to the first row of the DataFrame.
- 2: Points to the second row of the DataFrame.
- 3: Points to the third row of the DataFrame.
- 4: Points to the last column of the DataFrame.

Result: The final styled DataFrame is shown below, where negative values in columns B and C are colored red, and positive values are colored green.

	A	B	C
0	4	12	54
1	7	-7	-21
2	-6	7	-34

10. Print dataframe in Markdown

It is always better to print your data frame in a way that makes it easier to understand.

- ✓ One way of doing that is to render it in a markdown format using the `.to_markdown()` function.

The screenshot shows a Jupyter Notebook cell with the following code:

```
import pandas as pd

# Read the dataframe
candiates_df = pd.read_csv("candidates.csv")

# Print the original dataframe
print(candiates_df)
```

A red box highlights the output of the second print statement, which is a raw DataFrame representation:

	Name	Degree	From	Years_exp	From_office(min)
0	Aida	Master	Abidjan	2	120
1	Mamadou	Master	Dakar	3	95
3	Aicha	PhD	Abidjan	5	80

To the right of this output is a red speech bubble with the text "No markdown" and a red X icon.

Below the original print statement, another print statement uses the `to_markdown()` method:

```
# Print the dataframe in markdown
print(candiates_df.to_markdown(tablefmt = "grid"))
```

This output is highlighted with three colored boxes:

- Yellow box: "Your dataframe"
- Green box: "Markdown function"
- Pink box: "Table formatting using \"grid\""

A bracket groups these three boxes under the word "Result".

The resulting output is a well-formatted grid table:

	Name	Degree	From	Years_exp	From_office(min)
0	Aida	Master	Abidjan	2	120
1	Mamadou	Master	Dakar	3	95
3	Aicha	PhD	Abidjan	5	80

To the right of this output is a green speech bubble with the text "With markdown" and a green checkmark icon.

11. SQL-like queries on dataframe

You might want to filter 🔎 through your data to find relevant insights.

✓ This can be achieved using the built-in **query()** function in Pandas.

It runs queries based on boolean expressions, as you would write a natural language sentence! 💬

The screenshot shows a Jupyter Notebook cell with the following code:

```
# Import pandas library
import pandas as pd

# Read the data
candidate_df = pd.read_csv("candidates.csv")
```

An orange arrow labeled "Original data" points from the code to the resulting DataFrame on the right. The DataFrame has the following columns and data:

	Name	Degree	From	Years_exp	From_office(min)
0	Aida	Master	Abidjan	2	120
1	Mamadou	Master	Dakar	3	95
2	Ismael	Bachelor	Bamako	0	75
3	Aicha	PhD	Abidjan	5	80
4	Fatou	Master	Konakry	4	100
5	Khalil	PhD	Lomé	3	34

Below the DataFrame, the code continues:

```
# Filter on Degree and Years of experience
candidate_df.query("Degree == 'Master' and Years_exp > 2")
```

This code uses the **query()** function. A callout box highlights "Function used to make queries". Another callout box highlights the filter condition "Degree == 'Master' and Years_exp > 2".

The result of the query is a "Filtered data" DataFrame:

	Name	Degree	From	Years_exp	From_office(min)
1	Mamadou	Master	Dakar	3	95
4	Fatou	Master	Konakry	4	100

12. Extract time periods from datetime columns

Days, weeks, months, or quarters 📅, Each one can play an important role depending on the tasks at hand.

- With the **to_period()** function, you can extract from the date column each of such relevant information.

The screenshot shows a Jupyter Notebook cell with the following code:

```
# Read Pandas library
import pandas as pd

# Read the data
df = pd.read_csv("data.csv")

# Get months from date column
df["month"] = df["Application_date"].dt.to_period("M")

# Get quarters from date column
df["quarter"] = df["Application_date"].dt.to_period("Q")
```

An orange arrow labeled "Original data" points from the left towards the first table. A yellow callout box with a checkmark says "Final data with Quarters and Months columns". Another yellow callout box with a checkmark says "Get quarters (Q) and Months (M)".

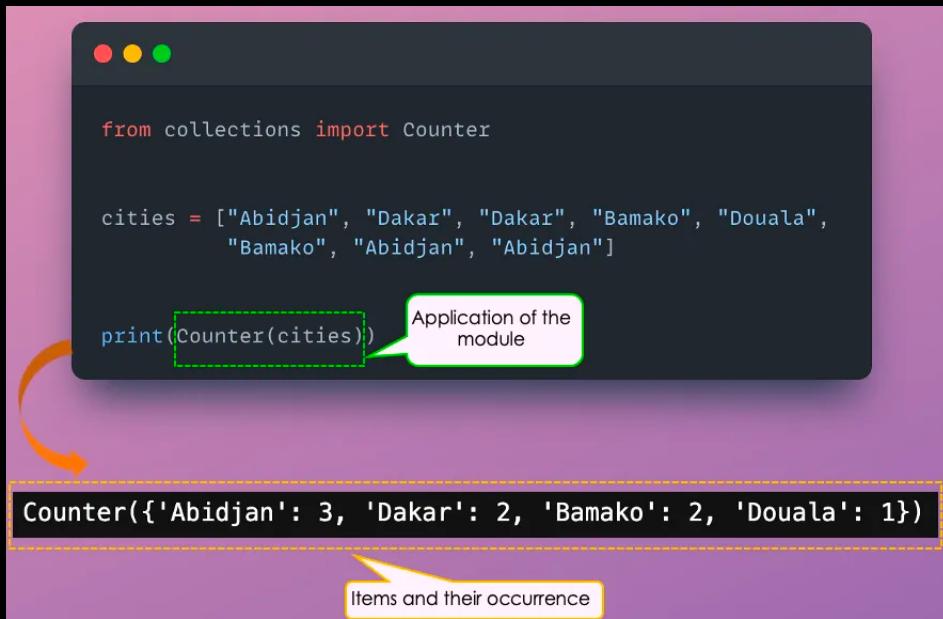
Name	Degree	Application_date	month	quarter
Aida	Master	2022-11-17	2022-11	2022Q4
Mamadou	Master	2022-09-23	2022-09	2022Q3
Ismael	Bachelor	2021-12-02	2021-12	2021Q4
Aicha	PhD	2022-08-25	2022-08	2022Q3
Fatou	Master	2022-01-07	2022-01	2022Q1
Khalil	PhD	2022-12-26	2022-12	2022Q4

13. Number of elements in a list

Still using loops  to determine how often each item occurs in a list?

Maybe there is a better and much more elegant Pythonic  way!

- ✓ You can use the **Counter** class from Python to compute the counts of the elements in a list.



The screenshot shows a Jupyter Notebook cell with the following code:

```
from collections import Counter

cities = ["Abidjan", "Dakar", "Dakar", "Bamako", "Douala",
         "Bamako", "Abidjan", "Abidjan"]

print(Counter(cities))
```

A green callout bubble points to the line `print(Counter(cities))` with the text "Application of the module". A yellow arrow points from the output area to the resulting Counter object with the text "Items and their occurrence".

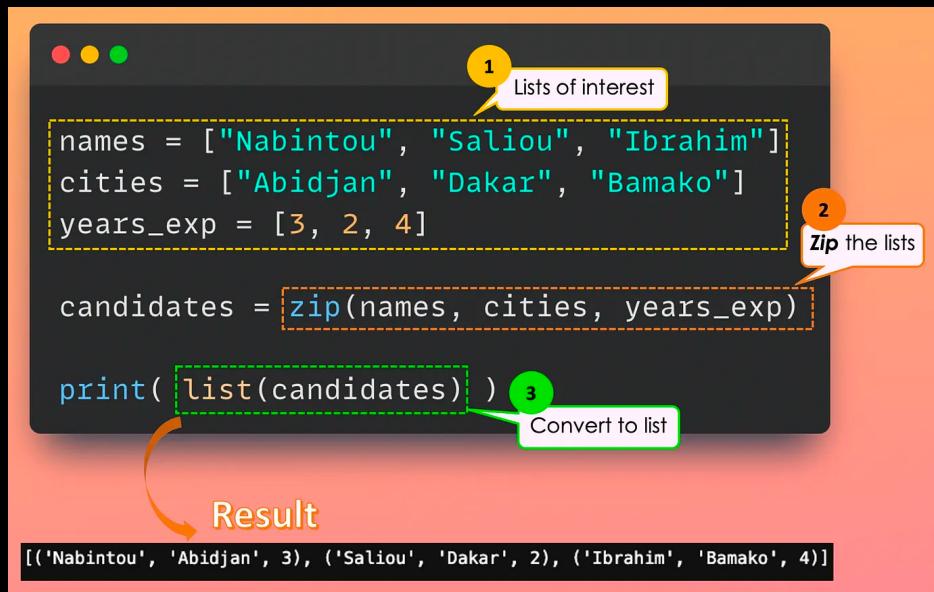
The output of the code is:

```
Counter({'Abidjan': 3, 'Dakar': 2, 'Bamako': 2, 'Douala': 1})
```

14. Combine elements from multiple lists

Are you trying to aggregate elements from multiple lists?

- ✗ Stop using **for** loops  and adopt the following approach.
- ✓ The Python built-in **zip()** function.



The screenshot shows a code editor window with the following code:

```
names = ["Nabintou", "Saliou", "Ibrahim"]
cities = ["Abidjan", "Dakar", "Bamako"]
years_exp = [3, 2, 4]

candidates = zip(names, cities, years_exp)

print(list(candidates))
```

Annotations explain the steps:

- 1 Lists of interest: Points to the three lists (`names`, `cities`, `years_exp`) at the top of the code.
- 2 Zip the lists: Points to the `zip` function call in the second line.
- 3 Convert to list: Points to the `list` function call in the third line.

Result: `[('Nabintou', 'Abidjan', 3), ('Saliou', 'Dakar', 2), ('Ibrahim', 'Bamako', 4)]`

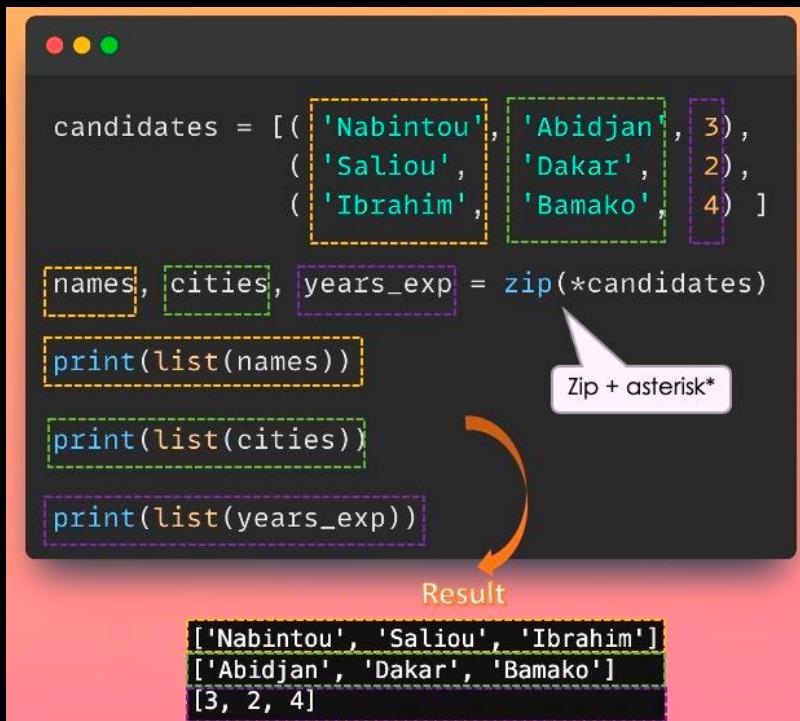
15. Create multiple lists from aggregated elements

When trying to aggregate elements from multiple lists, the most elegant and Pythonic way is to use the built-in `zip()` function.

Now, what if you want to proceed the other way around: create multiple lists from those aggregated elements ?

✗ Forget `for` loops 

✓ Just combine the `zip()` function with **asterisk ***



```
names, cities, years_exp = zip(*candidates)

print(list(names))
print(list(cities))
print(list(years_exp))
```

Zip + asterisk*

Result

['Nabintou', 'Saliou', 'Ibrahim']
['Abidjan', 'Dakar', 'Bamako']
[3, 2, 4]

16. Conditional data replacement

If you want to replace values from a dataframe based on conditions

- ✓ You can use the built-in **mask()** function from Pandas.

The screenshot shows a Jupyter Notebook cell with the following code:

```
# Import libraries
import pandas as pd
import numpy as np

# Create sample dataframe
df = pd.DataFrame({'A': [4, 7, -6, 2, -9],
                   'B': [12, -7, 7, -6, 23],
                   'C': [54, -21, 34, 32, -12]})

# Replace negative values in all columns with NaN
df.mask(df < 0, np.NaN)
```

Annotations explain the steps:

- Original data**: Points to the initial DataFrame on the right.
- 1**: Points to the condition `df < 0` in the `df.mask()` call.
- 2**: Points to the `np.NaN` in the `df.mask()` call.
- 3**: Points to the `df` in the `df.mask()` call.
- Apply mask**: Points to the first button in the toolbar.
- Condition**: Points to the second button in the toolbar.
- Replacement value**: Points to the third button in the toolbar.

The resulting DataFrame is shown below:

	A	B	C
0	4.0	12.0	54.0
1	7.0	NaN	NaN
2	NaN	7.0	34.0
3	2.0	NaN	32.0
4	NaN	23.0	NaN

17. Change columns type

The wrong data format is a common challenge when dealing with real-world 🌎 data.

For instance, you might have a numerical value that is stored as a string such as “34” instead of 34.

- ✓ Using the ***astypefunction***, you can easily convert data from one type to another (e.g. string to numerical).

The screenshot shows a Jupyter Notebook cell with the following code:

```
import pandas as pd

info = {'Age': ['23', '43', '31'],
        'Degree': ["MS", "BS", "PhD"]}

df = pd.DataFrame(info)

# Check data types
df.dtypes
```

A Pandas DataFrame is shown with three rows and two columns:

	Age	Degree
0	23	MS
1	43	BS
2	31	PhD

The code `df["Age"] = df["Age"].astype(int)` is highlighted with a green box. A green arrow points from this line to the resulting output:

```
# Convert "Age" column data type
df["Age"] = df["Age"].astype(int)

# Check data types
df.dtypes
```

The output shows the data types for each column:

	Age	Degree
dtype:	int64	object

An orange callout bubble says "Age column to integer".

18. Equality of two dataframes

Two columns with the same name may not contain the same values, and two rows with the same index may not be identical.

To know if two DataFrames are equal, you need to go deeper  to check if they have the same shape and same elements.

This is where the Pandas **equals()** function comes in handy.

-  It returns True if the two DataFrames are equal.
-  It returns False if they are not equal.



Scan and follow



```
import pandas as pd
```

```
candidates_1 = {
```

```
    'Age': ["23", "43", "31"],
```

```
    'Degree': ["MS", "BS", "PhD"]
```

```
}
```

- Same name
- Same values
- Different Format

```
candidates_2 = {
```

```
    'Age': [23, 43, 31],
```

```
    'Degree': ["MS", "BS", "PhD"]
```

```
}
```

```
first_candidates = pd.DataFrame(candidates_1)
```

	Age	Degree
0	23	MS
1	43	BS
2	31	PhD

	Age	Degree
0	23	MS
1	43	BS
2	31	PhD

```
second_candidates = pd.DataFrame(candidates_2)
```

```
# Check the similarity of the two datasets
```

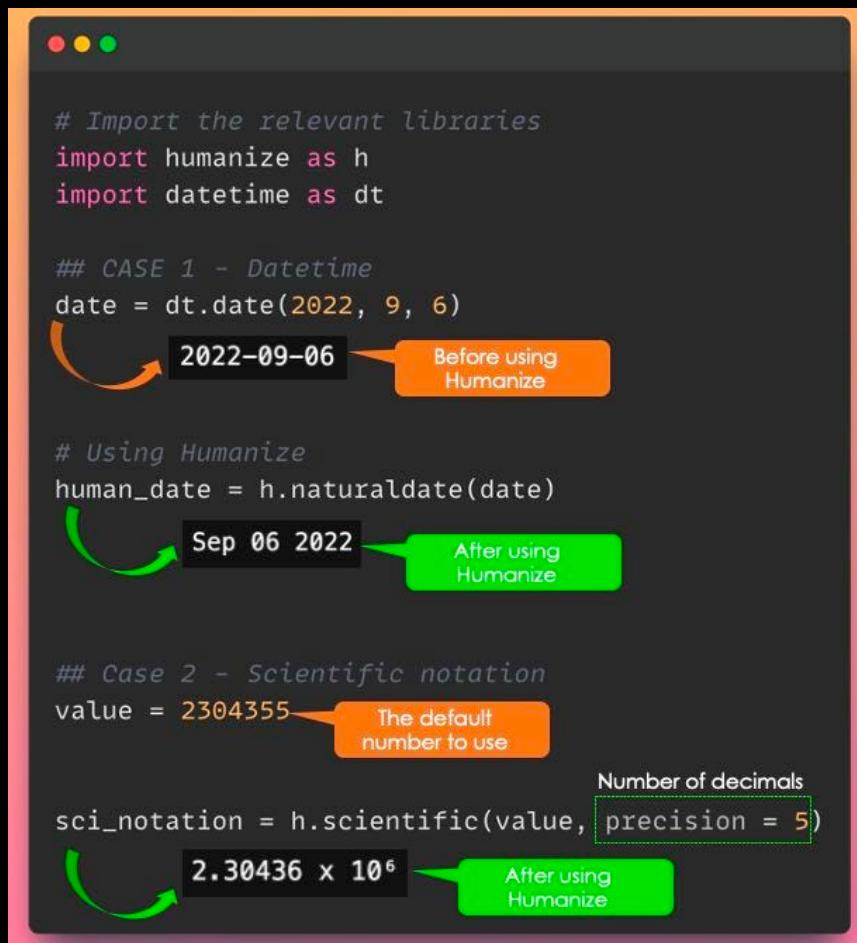
```
first_candidates.equals(second_candidates)
```

The dataframes are
different, hence
returns False

19. Make Python output more human-readable

Sometimes it is necessary to go beyond the default output provided by Python to make it more understandable by humans .

- ✓ This can be achieved using the `humanize` library.



```
# Import the relevant libraries
import humanize as h
import datetime as dt

## CASE 1 - Datetime
date = dt.date(2022, 9, 6)
2022-09-06 Before using Humanize

# Using Humanize
human_date = h.naturaldate(date)
Sep 06 2022 After using Humanize

## Case 2 - Scientific notation
value = 2304355 The default number to use
Number of decimals
sci_notation = h.scientific(value, precision = 5)
2.30436 x 106 After using Humanize
```

The screenshot shows a Jupyter Notebook cell with Python code demonstrating the `humanize` library. It includes two examples: one for date conversion and one for scientific notation. Annotations with arrows point from specific outputs to callouts. The first example shows the date `2022-09-06` before and after using `humanize.naturaldate`, with the result being `Sep 06 2022`. The second example shows the value `2304355` before and after using `humanize.scientific`, with the result being `2.30436 x 106`.

20. Convert natural language to numerical values

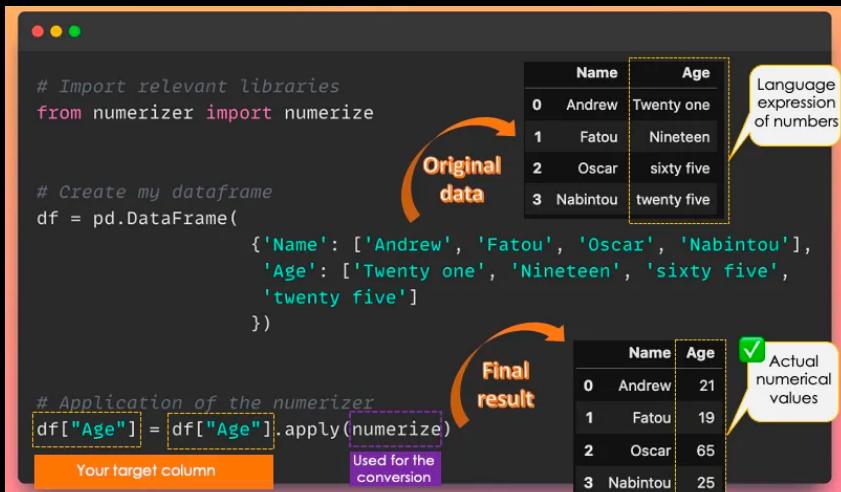
Natural language 🧑‍💻 is everywhere 🌎, even in our DataFrames.

This is not a bad thing itself because it is the perfect  type of data when performing natural language processing tasks.

However, their limitations 🤢🚫 become obvious when trying to perform numerical computation.

 To tackle this issue, you can use the **numerize()** function from the python library **numerizer**.

 It converts natural language expressions of numbers into their actual numerical values.



The screenshot shows a Jupyter Notebook cell with the following code:

```
# Import relevant libraries
from numerizer import numerize

# Create my dataframe
df = pd.DataFrame(
    {'Name': ['Andrew', 'Fatou', 'Oscar', 'Nabintou'],
     'Age': ['Twenty one', 'Nineteen', 'sixty five',
             'twenty five']}
)

# Application of the numerizer
df["Age"] = df["Age"].apply(numerize)
```

Annotations explain the process:

- A callout points to the 'Age' column in the original DataFrame: "Language expression of numbers".
- A callout points to the 'Age' column in the final DataFrame: "Actual numerical values".
- An orange box highlights "Your target column" under the DataFrame.
- A purple box highlights "Used for the conversion" under the code.

Name	Age
0 Andrew	Twenty one
1 Fatou	Nineteen
2 Oscar	sixty five
3 Nabintou	twenty five

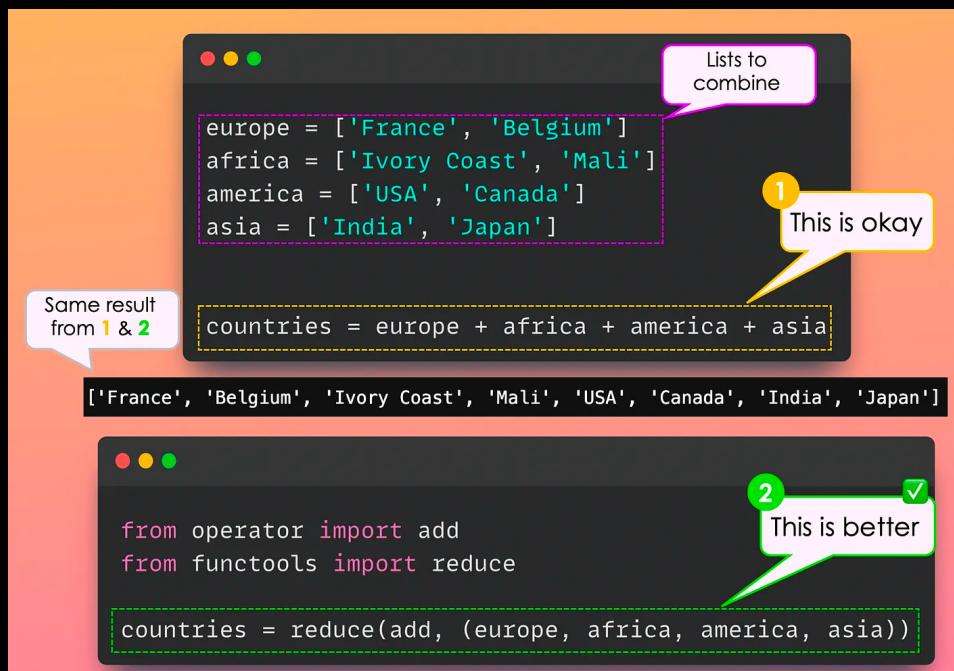
Name	Age
0 Andrew	21
1 Fatou	19
2 Oscar	65
3 Nabintou	25

21. Combine multiple lists

Using the + sign is probably the most common approach to combining  lists.

However, typing the + sign all the time can become easily boring when you have to deal with multiple lists.

✓ Instead, you can use the **add** and **reduce** functions respectively from the **operator** and **functools** modules.



The image shows a terminal window with two examples of list concatenation. The first example uses the + operator, and the second uses the add and reduce functions from the operator and functools modules respectively.

Example 1 (Using + operator):

```
lists to combine
1 This is okay
Same result from 1 & 2
countries = europe + africa + america + asia
['France', 'Belgium', 'Ivory Coast', 'Mali', 'USA', 'Canada', 'India', 'Japan']
```

Example 2 (Using add and reduce):

```
2 This is better
from operator import add
from functools import reduce
countries = reduce(add, (europe, africa, america, asia))
```

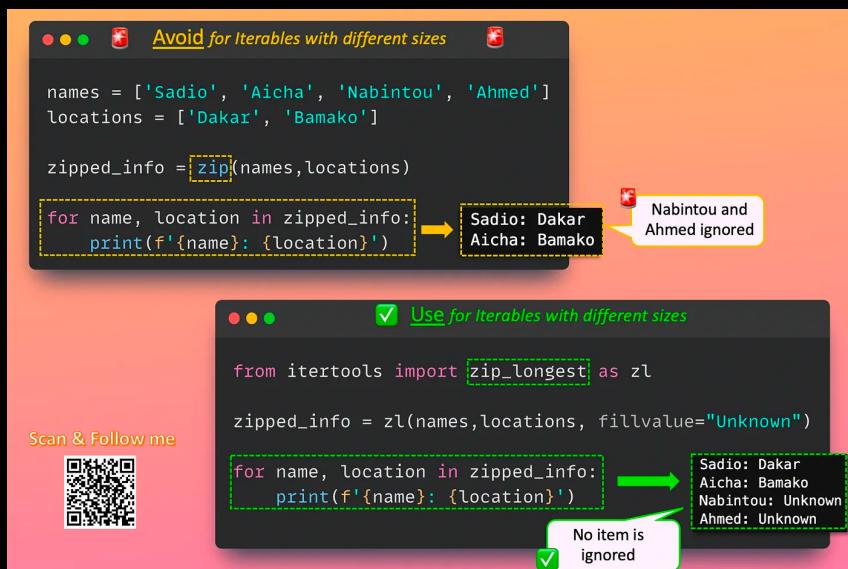
22. Zip Iterables of different sizes

If you have been using the **zip()** function, then you might be aware of this limitation: it does not work with **iterables** of different sizes, which can lead to information loss.

  You can tackle this issue with the **zip** function's cousin: **zip_longest()** function from the **itertools** module.

Instead of ignoring the remaining items, their values are replaced with **None**

That's good but can be even amazing to use the **fillvalue** parameter to replace the **None** with a meaningful value.



The slide features two code snippets side-by-side, each with a title and a callout box.

Avoid for Iterables with different sizes

```
names = ['Sadio', 'Aicha', 'Nabintou', 'Ahmed']
locations = ['Dakar', 'Bamako']

zipped_info = zip(names, locations)

for name, location in zipped_info:
    print(f'{name}: {location}')
```

Nabintou and Ahmed ignored

Use for Iterables with different sizes

```
from itertools import zip_longest as zl

zipped_info = zl(names, locations, fillvalue="Unknown")

for name, location in zipped_info:
    print(f'{name}: {location}')
```

No item is ignored

Scan & Follow me 

23. Combine SQL statements and Pandas

My gut feeling is telling me that more than 80% of the Data Scientists use Pandas in their daily Data Science activities.

And, I believe that this is because of the benefits it offers of being part of the wider range of the Python universe, making it accessible to many people.

What about SQL?

Even though not everyone uses it in their daily life (because not every company has necessarily a SQL Database?), SQL's performance is undeniable.

Also, it is human-readable which makes it easily understood by even non-tech people.

❓ What if we could find a way to ***combine the benefits of both Pandas and SQL*** statements?

✓ Here is where **pandasql** comes in handy

```
import pandas as pd
from pandasql import sqldf
```

`students_df` = pd.DataFrame({
 'Students':["Sira", "Ibrahim", ...],
 ...})

	Students	Gender	Age	Email
0	Sira	Female	18	sira@info.com
1	Ibrahim	Male	27	ib@info.com
2	Moussa	Male	19	mouss@info.com
3	Mamadou	Male	22	mam@info.com
4	Nabintou	Female	21	nab@info.com

```
teaching_assistant_df = pd.DataFrame{  
    'Teacher':["Ibrahim", "Nabintou", ...],  
    ...})
```

SQL query to execute

```
query = """ SELECT st.Students, st.Gender,  
                st.Email, st.Age,  
                tat.Department  
        FROM students_df st  
        INNER JOIN teaching_assistant_df tat  
        ON st.Email = tat.Email;  
    """
```

sqldf library

```
result = sqldf(query)
```

Final result

	Students	Gender	Email	Age	Department
0	Ibrahim	Male	ib@info.com	27	Business
1	Mamadou	Male	mam@info.com	22	Comp Sc
2	Nabintou	Female	nab@info.com	21	Statistics

@zoumdatascience

24. Update a given dataframe with another dataframe

There are multiple ways of replacing missing values  in Pandas, from simple imputation to more advanced methods.

But ... 

Sometimes, you just want to replace them using non-NA values from another DataFrame.

- ✓ This can be achieved using the built-in update function from Pandas.

It aligns both DataFrames on their index and columns before performing the update.

General syntax  below:

`first_dataframe.update(second_dataframe)`

 missing values from **`first_dataframe`** dataframe are replaced with non-missing values from **`second_dataframe`**

 **`overwrite=True`** will overwrite **`first_dataframe`**'s values from using **`second_dataframe`** data, and this is the default value. If **`overwrite=False`** only the missing values are replaced.

```

import pandas as pd

# Create two DataFrames
first_dataframe = pd.DataFrame({'A': [10, None, 30],
                                 'B': [None, 50, None],
                                 'C': [70, None, None]})



Before: Data with missing values



| A    | B    | C    |
|------|------|------|
| 10.0 | NaN  | 70.0 |
| NaN  | 50.0 | NaN  |
| 30.0 | NaN  | NaN  |



second_dataframe = pd.DataFrame({'A': [1, 2, 3],
                                 'B': [4, 5, 6],
                                 'C': [7, 8, 9]})



# Update the first dataframe


first_dataframe.update(second_dataframe, overwrite=False)

```

After: Missing values replaced



A	B	C
10.0	4.0	70.0
2.0	50.0	8.0
30.0	6.0	9.0

Do not overwrite non-missing values

25. From unstructured to structured data

Data preprocessing is full of challenges 🔥

Imagine you have this data with candidates' information in the following format:

‘**Adja Kone: has Master in Statistics and is 23 years old**’

...

‘**Fanta Traore: has PhD in Statistics and is 30 years old**’

Then, your task is to generate a table with the following information per candidate for further analysis:

- ❖ The first and last name
- ❖ The degree and field of study
- ❖ The Age

 Performing such a task can be daunting 😱

 This is where the **str.extract()** function in Pandas can help!

It is a powerful text-processing function for extracting structured information from unstructured textual data.

```
import pandas as pd
import re

candidates_df = pd.DataFrame({
    'Details': [
        'Adja Kone: has Master in Statistics and is 23 ...',
        'Issa Cisse: has Master in Economics and is 25 ...',
        'Frank Kouadio: has PhD in Finance and is 30 ye...',
        'Fanta Traore: has PhD in Statistics and is 30 ...'
    ]
})
```

Pattern of desired data format

```
pattern = re.compile(r"""
    (?P<First_name>\w+) \s (?P<Last_name>\w+): .*?
    (?P<Degree>Master|PhD).*?
    (?P<Field>\w+(?:\s\w+)?).*?
    (?P<Age>\d+) """, re.VERBOSE)
```

```
structured_data = candidates_df['Details'].str.extract(pattern)
```

Structured data

✓

First_name	Last_name	Degree	Field	Age
Adja	Kone	Master	in Statistics	23
Issa	Cisse	Master	in Economics	25
Frank	Kouadio	PhD	in Finance	30
Fanta	Traore	PhD	in Statistics	30

Apply pattern to extract key information



@zoumdatascience

@zoumana_keita_

26. Perform multiple aggregations with the agg() function

If you want to perform multiple aggregation functions like **sum**, **average**, **count** ... on one or multiple columns.

- ✓ You can combine **groupby()** and **agg()** functions from Pandas in one line of code.

Here is a Scenario  

Let's imagine this student's data containing information about:

- ❖ Students' areas of study
- ❖ Their grades
- ❖ The graduation years and the age of each student

And, you have been requested to compute the following information per area of study and year:

- The number of students
- The average grade
- The average age

```

import pandas as pd
# Students' information
students_data = pd.DataFrame({
    'Area': ['Math', 'Finance', 'Finance', 'Math', 'Math'],
    'Grade': [90, 80, 85, 95, 88],
    'Year': [2021, 2021, 2022, 2021, 2021],
    'Age': [18, 19, 20, 19, 21]
})
# Combining groupby() + agg() functions
aggregated_data = students_data.groupby(['Area', 'Year']).agg(
    Average_Grade=('Grade', 'mean'),
    Number_of_students=('Grade', 'count'),
    Average_Age=('Age', 'mean')
)
# Resetting the index for better readability
aggregated_data = aggregated_data.reset_index()

```

Area	Grade	Year	Age
Math	90	2021	18
Finance	80	2021	19
Finance	85	2022	20
Math	95	2021	19
Math	88	2021	21

Aggregation constraints

Aggregation applied to groupby



Area	Year	Avg_Grade	Student_Count	Avg_Age
Finance	2021	80.0	1	19.000000
Finance	2022	85.0	1	20.000000
Math	2021	91.0	3	19.333333



27. Select data between two dates

When working with time series data, you might want to select observations between two specified times for further analysis.

- ✓ This can be quickly achieved using the `between_time()` function

The screenshot shows a Jupyter Notebook cell with Python code. A green arrow labeled "Original data" points to the DataFrame definition. A green arrow labeled "Filtered data" points to the filtered DataFrame. A callout box explains the effect of the `pd.date_range` call.

```
import pandas as pd

# Create the DataFrame
attendance_df = pd.DataFrame({
    'Major': ['CS', 'Math', 'Finance', 'Physics', 'Biology'],
    'Location': ['Abidjan', 'Dakar', 'Bamako', 'Lome', 'Cotonou'],
    'Attendance': pd.date_range('2023-04-09', periods=5, freq='H')
})

# Set the Attendance column as index for filtering
attendance_df = attendance_df.set_index('Attendance')

# Filter observations between desired 1:00 & 3:30
filtered_df = attendance_df.between_time('1:00', '3:30').reset_index()
```

Major	Location	Attendance
CS	Abidjan	2023-04-09 00:00:00
Math	Dakar	2023-04-09 01:00:00
Finance	Bamako	2023-04-09 02:00:00
Physics	Lome	2023-04-09 03:00:00
Biology	Cotonou	2023-04-09 04:00:00

Attendance	Major	Location
2023-04-09 01:00:00	Math	Dakar
2023-04-09 02:00:00	Finance	Bamako
2023-04-09 03:00:00	Physics	Lome

Creates 5 rows from April 9th. Current row is 1h apart from the next row.

@zoumdatagridscience @zoumana_keita_

28. Check if all elements meet a certain condition

✗ The combination of **for** loops and **if** statements is not always the most elegant way when writing Python code.

For instance, let's say that you want to check if all the elements of an iterable meet a certain condition.

Two possibilities may arise:

1 Either use for loop and if statement.

OR

2 Use the all() built-in function

```
# List to check
numbers = [3, 5, 7, 11, 15]

# Check if all elements are odd using for loop
all_odd = True

for num in numbers:
    if num % 2 == 0:
        all_odd = False
        break

print("All numbers are odd:", all_odd)
# → All numbers are odd: True
```

✗ **for** loop and **if** statement


```
# Check if all elements are odd
all_odd = all(num % 2 != 0 for num in numbers)
print("All elements are odd:", all_odd)
# → All numbers are odd: True
```

✓ More elegant

29. Check if any element meets a certain condition

Similarly to the previous case, if you want to check if at least one element of an iterable meet a certain condition.

- ✓ Then use the `any()` built-in function which is more elegant than using for loop and if statement.

The illustration is similar to the above image.

Avoid nested for loops

Writing nested `for` loops is almost inevitable when your program becomes bigger and more complicated.

- ✗ This can also make your code difficult to read and maintain.

- ✓ A better alternative is to use the built-in `product()` function instead.

```
from itertools import product
```

```
first_list = [12, 46, 11, 7, 9]
second_list = [1, 62, 2, 8]
third_list = [7, 4, 3, 54, 87]
```

One level of
for loop

```
for fst, snd, trd in product(first_list, second_list, third_list):
    if(fst * snd * trd == 126):
        print(f' 1st: {fst} 2nd: {snd} 3rd: {trd}')
    ....
```

→ 1st: 9 2nd: 2 3rd: 7

```
for fst in first_list:
    for snd in second_list:
        for trd in third_list:
            if(fst * snd * trd == 126):
                print(f' 1st: {fst} 2nd: {snd} 3rd: {trd}')
    ....
```

→ 1st: 9 2nd: 2 3rd: 7

3-level nested
for loop

30. Automatically handle index in a list

Imagine you have to access elements in a list and their indexes at the same time.

One way of doing it is handling manually the indexes in a for loop.

✓ Instead, you can use the **enumerate()** built-in function.

This has two main benefits (I can think of).

- ★ First it automatically handles the index variable.
- ★ Then makes the code more readable.

The diagram illustrates the difference between manually managing indices and using the `enumerate()` function.

Index managed manually: A red X marks this approach as incorrect. The code shows a for loop where the index is tracked manually with a variable `index = 0` and updated inside the loop body. The output shows the index and city name.

```
big_cities = ['Abidjan', 'Dakar', 'Bamako']
index = 0
for big_city in big_cities:
    print(f"Index: {index}, Big City: {big_city}")
    index += 1
```

Index managed automatically: A green checkmark indicates this is the correct approach. The code uses the `enumerate()` function to handle the index automatically. The output is identical to the manual method.

```
for index, big_city in enumerate(big_cities):
    print(f"Index: {index}, Big City: {big_city}")
```

Both sections show the output:
Index: 0, Big City: Abidjan
Index: 1, Big City: Dakar
Index: 2, Big City: Bamako

A green arrow labeled "Same result" points from the manual section's output to the automatic section's output.

Thanks for reading!

If you enjoy this e-book, you will love my newsletter about *AI, Data Science & Analytics*.

Newsletter 

ZoumDataScience.com

