PYTHON & PANDAS

TIPS & TRICKS

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

Zoumana Keita

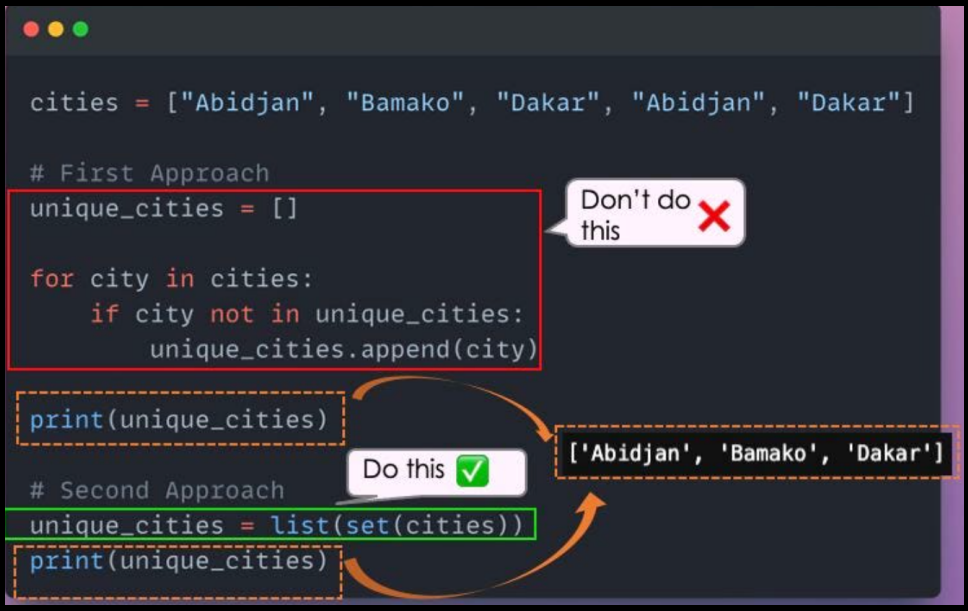
ZoumDataScience.com

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.



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.

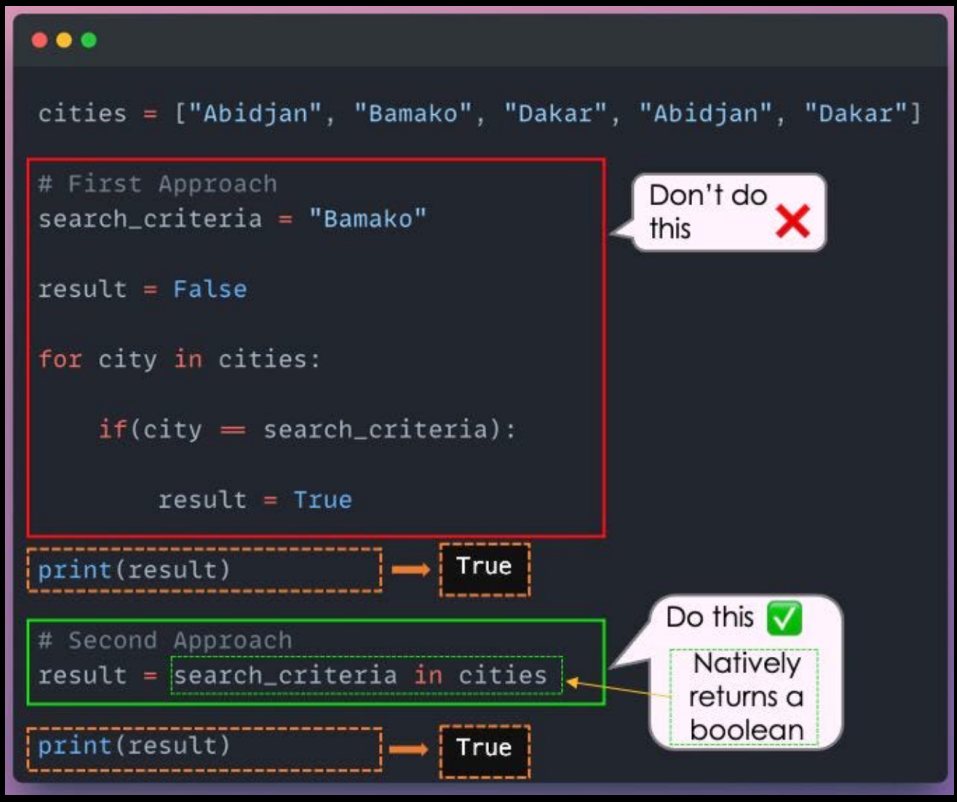
Instead, use **dict.fromkeys**() ✅ to preserve the original order.

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.



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.

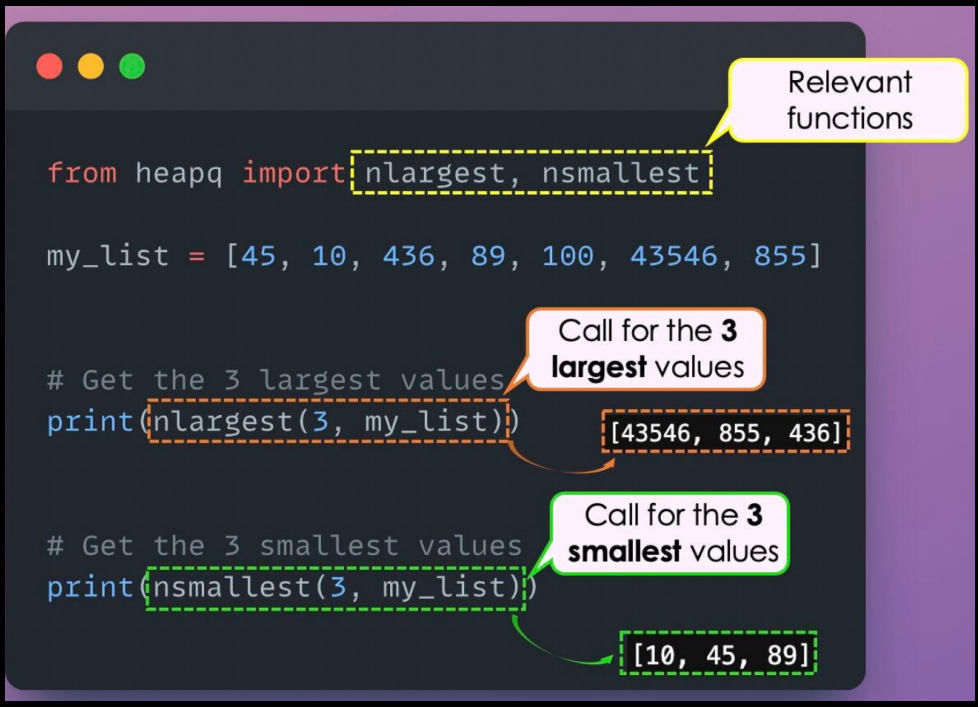
However, when it comes to getting the **N largest** or **smallest** values of you might think of a two-way approach:

1️⃣ Sort the list in decreasing or increasing order.

2️⃣ 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]
Call for the 3 largest values
# Get the 3 largest values
print(nlargest(3, my_list))
[43546, 855, 436]
Call for the 3 smallest values
# Get the 3 smallest values
print(nsmallest(3, my_list))
[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:

1️⃣ Use **describe**() for numerical columns.

2️⃣ Set the parameter **include**=[**object**] to provide information about categorical ones.

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

# Data column types
candidates_df.dtypes

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

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

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

Default result

	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

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.

```python
import pandas as pd
import time

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

# OPTION 1: FOR LOOP

begin = time.time()

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)

# OPTION 2: VECTORIZATION

begin = time.time()

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

print(time.time() - begin)

display(game_df.head())
```

Data with 16598 Rows

Avoid this

2.63 seconds

Do this

0.0006 seconds

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

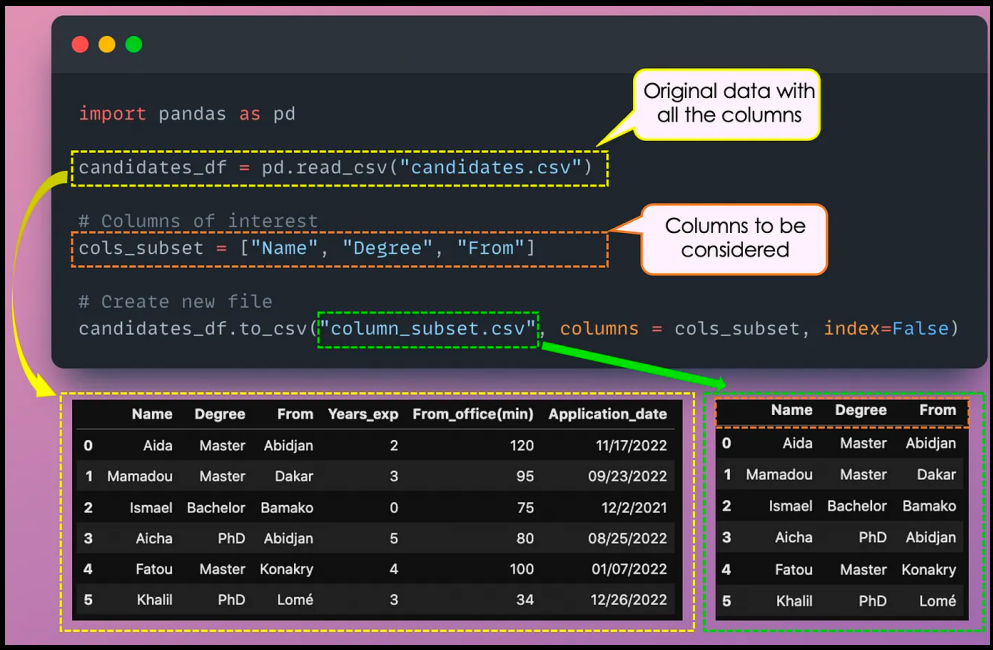
6. **Save a subset of Pandas columns**

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.



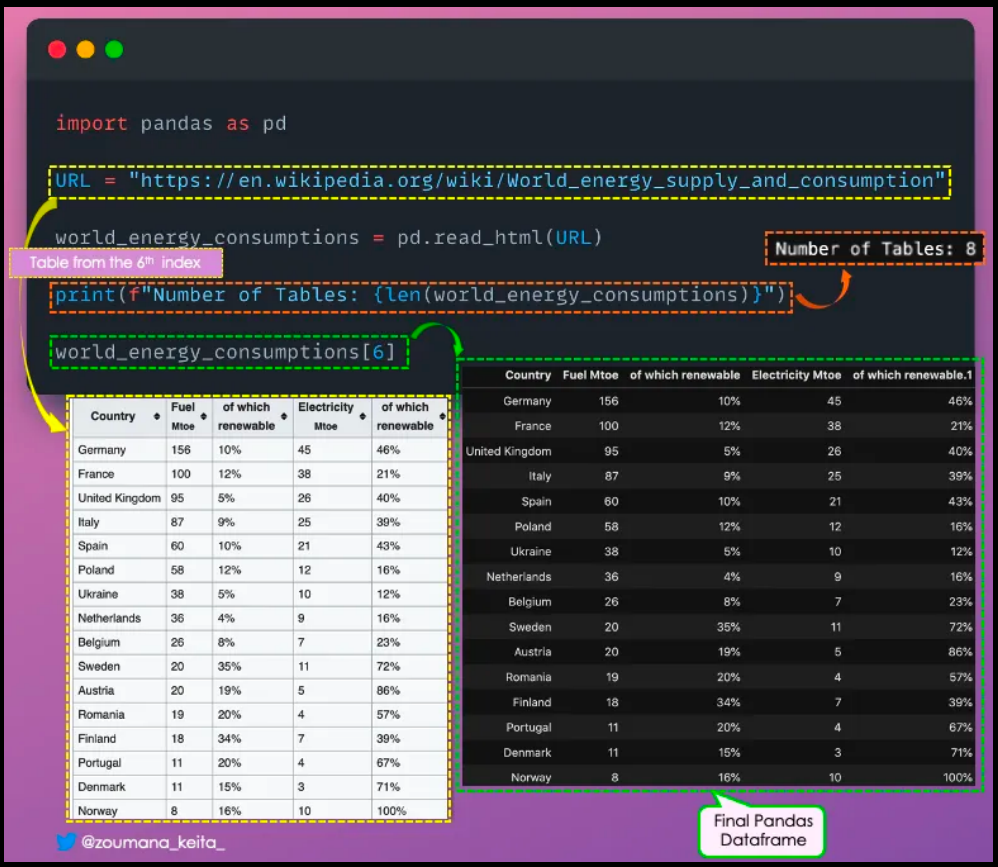
	Name	Degree	From	Years_exp	From_office(min)	Application_date
0	Aida	Master	Abidjan	2	120	11/17/2022
1	Mamadou	Master	Dakar	3	95	09/23/2022
2	Ismael	Bachelor	Bamako	0	75	12/2/2021
3	Aicha	PhD	Abidjan	5	80	08/25/2022
4	Fatou	Master	Konakry	4	100	01/07/2022
5	Khalil	PhD	Lomé	3	34	12/26/2022

	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.

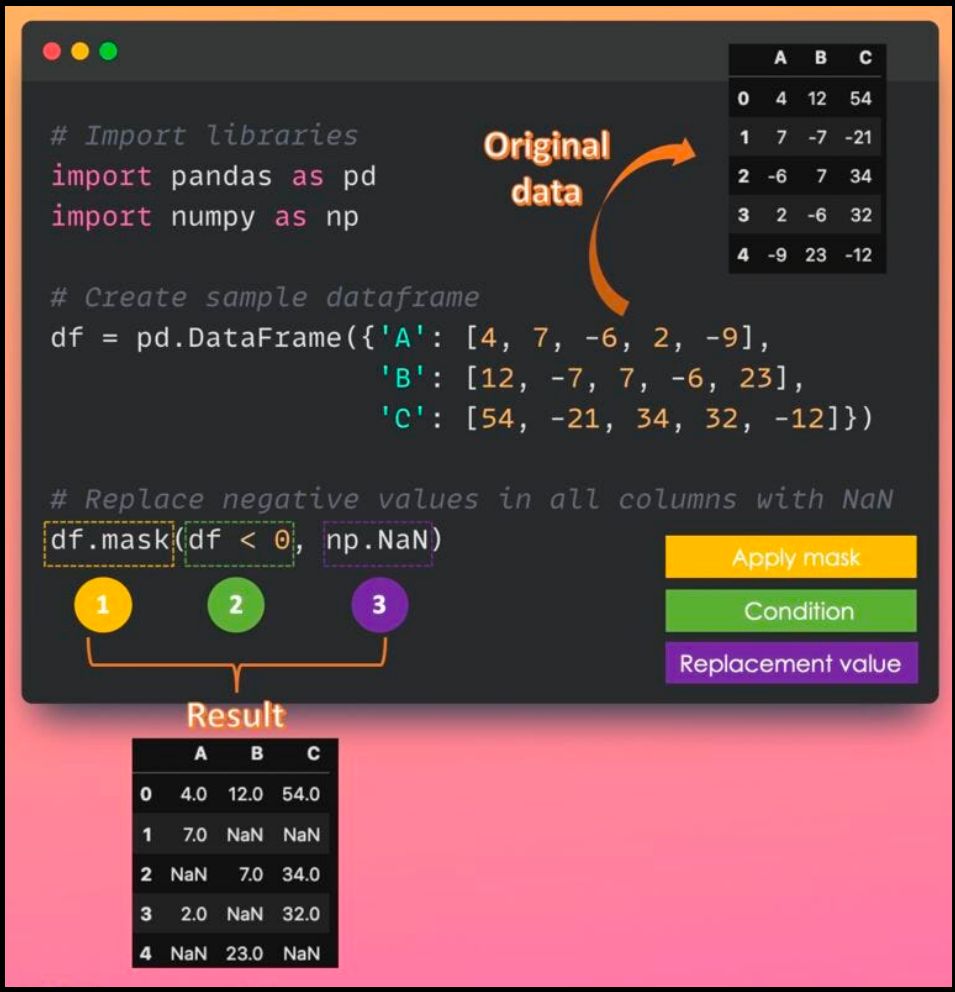


Country	Fuel Mtoe	of which renewable	Electricity Mtoe	of which renewable
Germany	156	10%	45	46%
France	100	12%	38	21%
United Kingdom	95	5%	26	40%
Italy	87	9%	25	39%
Spain	60	10%	21	43%
Poland	58	12%	12	16%
Ukraine	38	5%	10	12%
Netherlands	36	4%	9	16%
Belgium	26	8%	7	23%
Sweden	20	35%	11	72%
Austria	20	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%

8. **Replace values from a dataframe based on conditions**

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

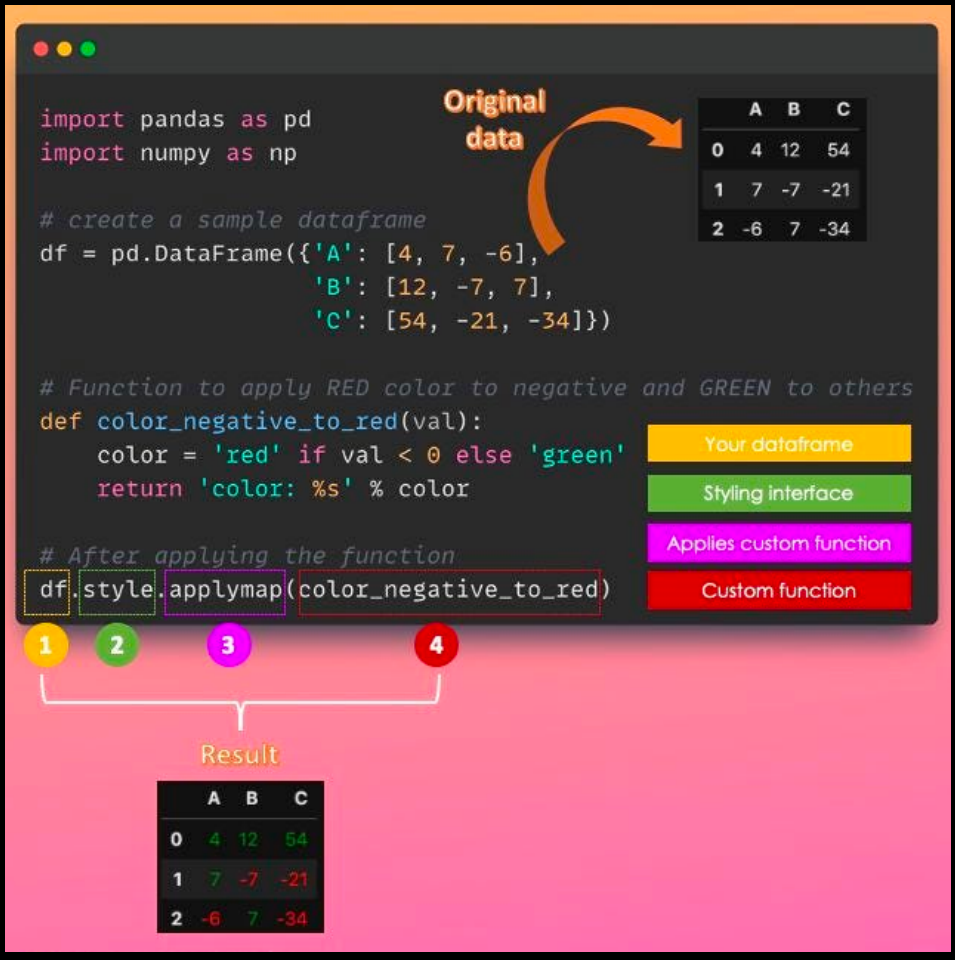
✅ You can use the built-in **mask**() function from Pandas.



9. **Apply colors to your dataframe**

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

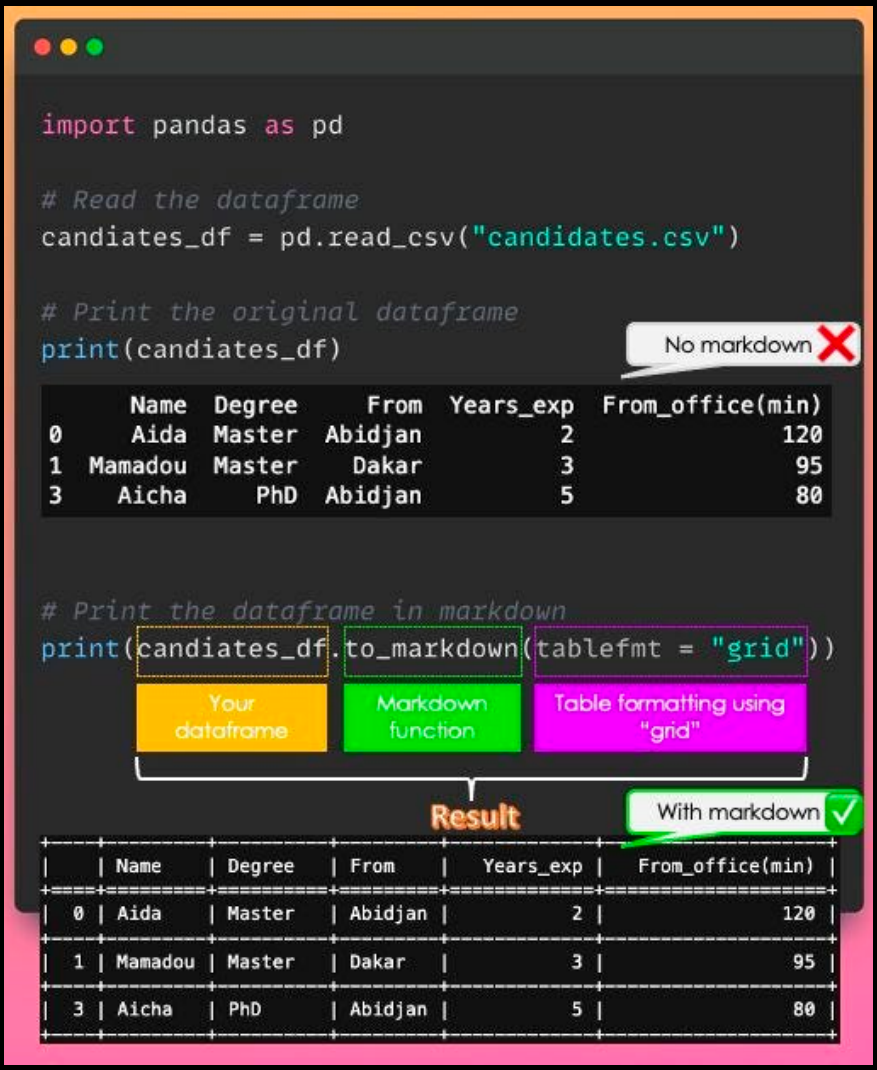
This can be helpful to quickly find some information JUST by looking at your dataframe.



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.

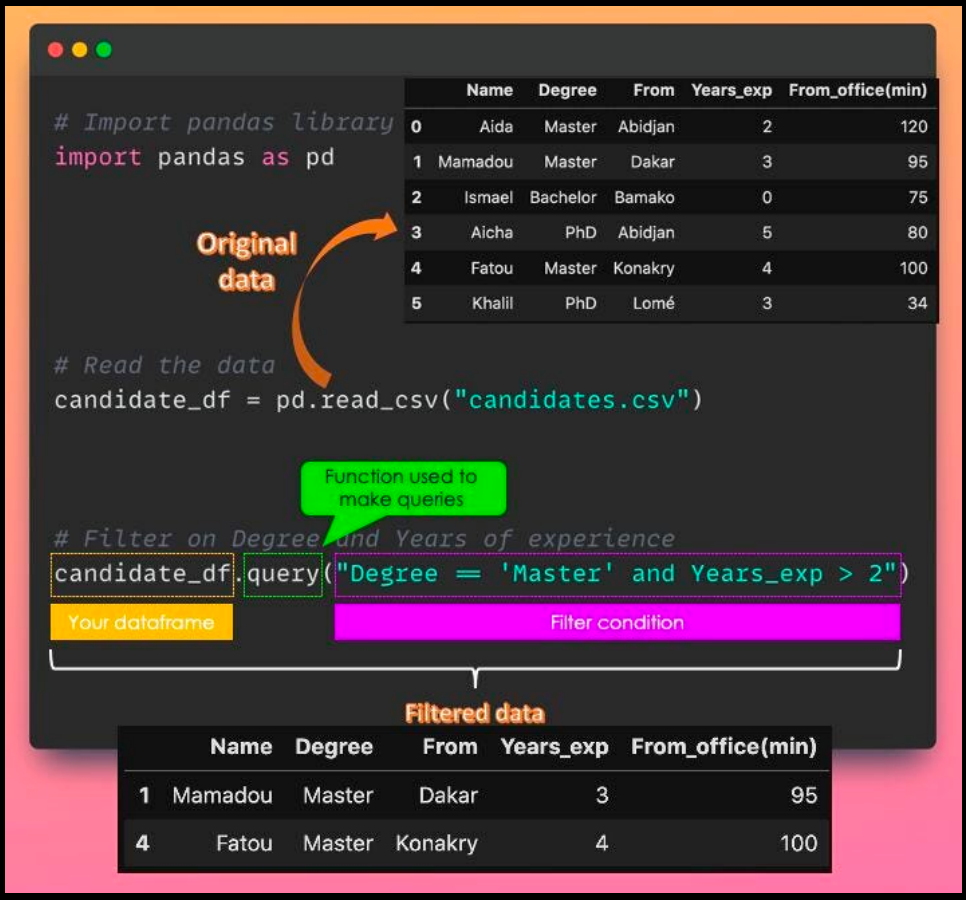


11. **SQL-like queries on dataframe**

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! 💬




```
# Import pandas library
import pandas as pd

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

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

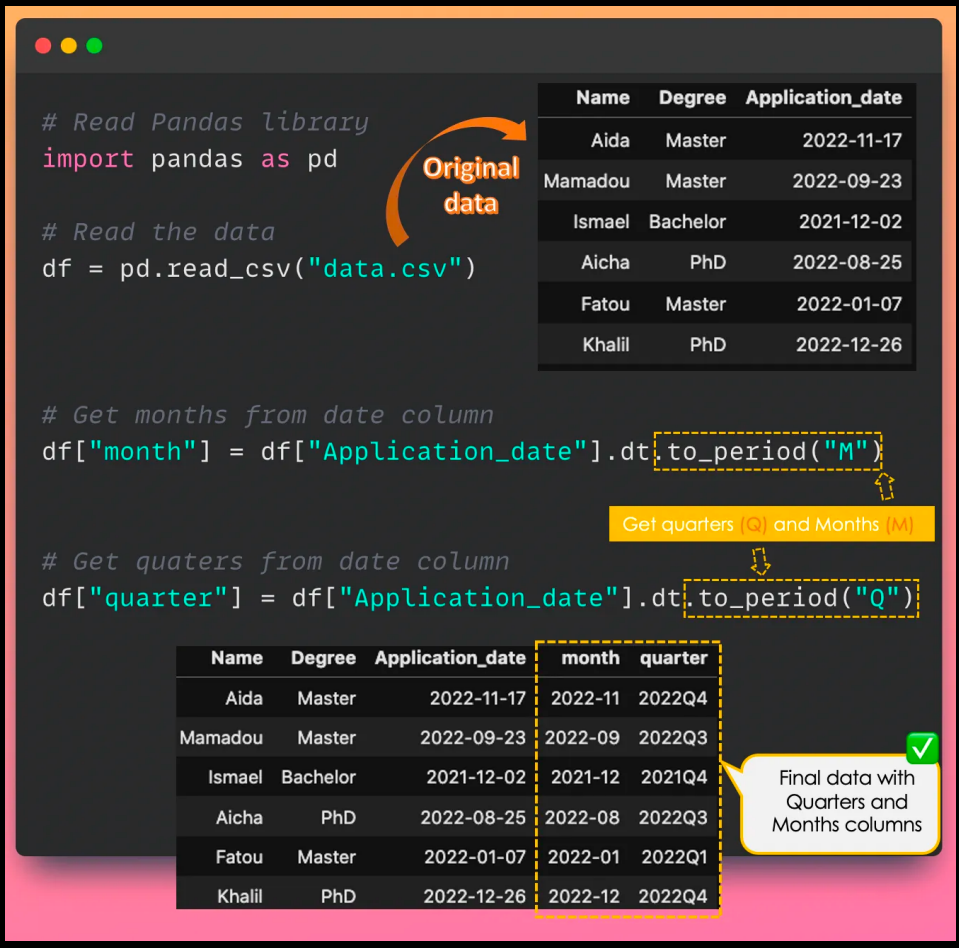
	Name	Degree	From	Years_exp	From_office(min)
0	Aida	Master	Abidjan	2	120
1	Mamadou	Master	Dakar	3	95
2	Ismael	Bachelor	Bamako	0	75
3	Aicha	PhD	Abidjan	5	80
4	Fatou	Master	Konakry	4	100
5	Khalil	PhD	Lomé	3	34

	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.




```
# Read Pandas library
import pandas as pd

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

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

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

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

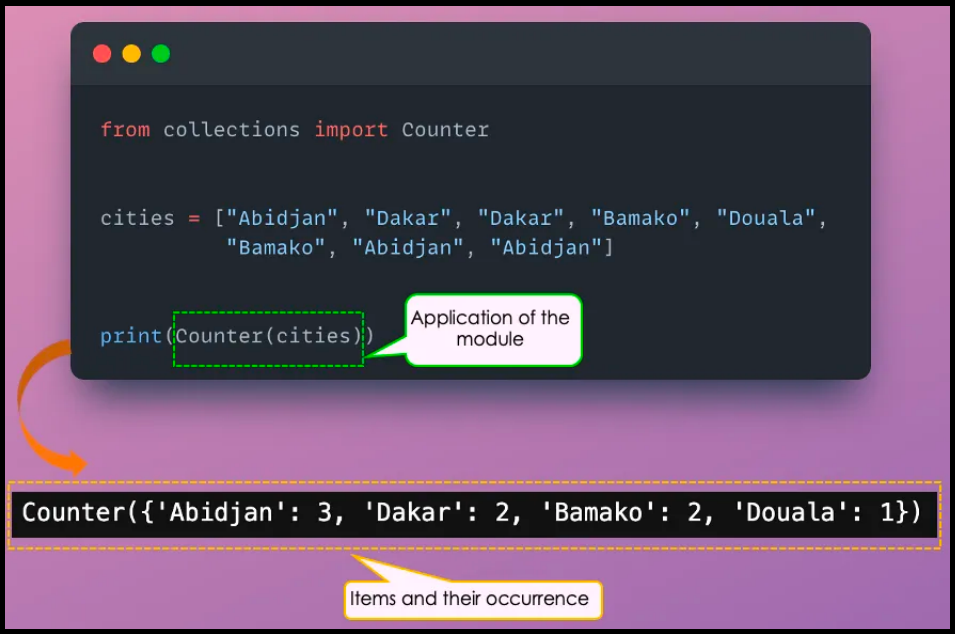
Name	Degree	Application_date	month	quarter
Aida	Master	2022-11-17	2022-11	2022Q4
Mamadou	Master	2022-09-23	2022-09	2022Q3
Ismael	Bachelor	2021-12-02	2021-12	2021Q4
Aicha	PhD	2022-08-25	2022-08	2022Q3
Fatou	Master	2022-01-07	2022-01	2022Q1
Khalil	PhD	2022-12-26	2022-12	2022Q4

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.

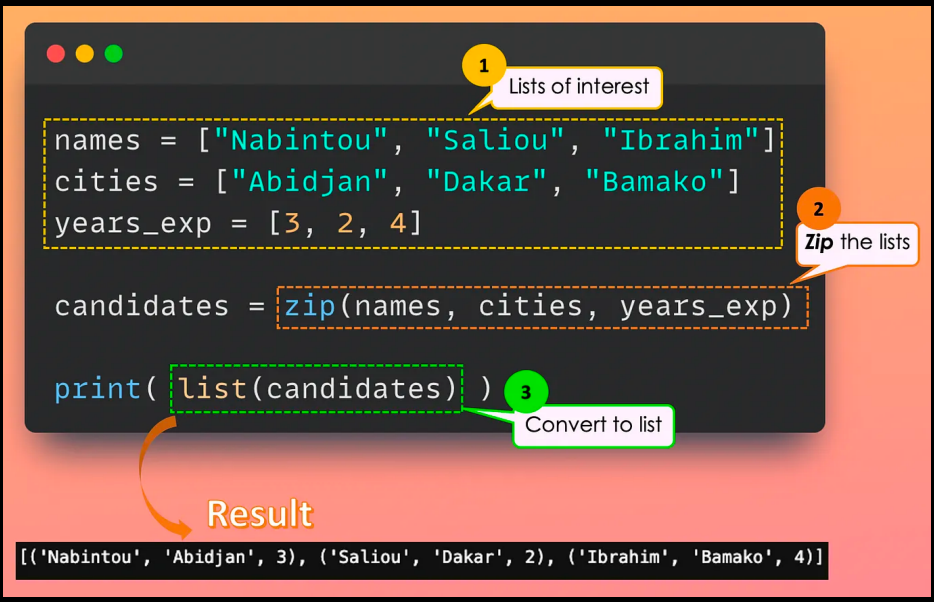


14. **Combine elements from multiple lists**

Are you trying to aggregate elements from multiple lists?

❌ Stop using **for** loops 🔁 and adopt the following approach.

✅ The Python built-in **zip**() function.



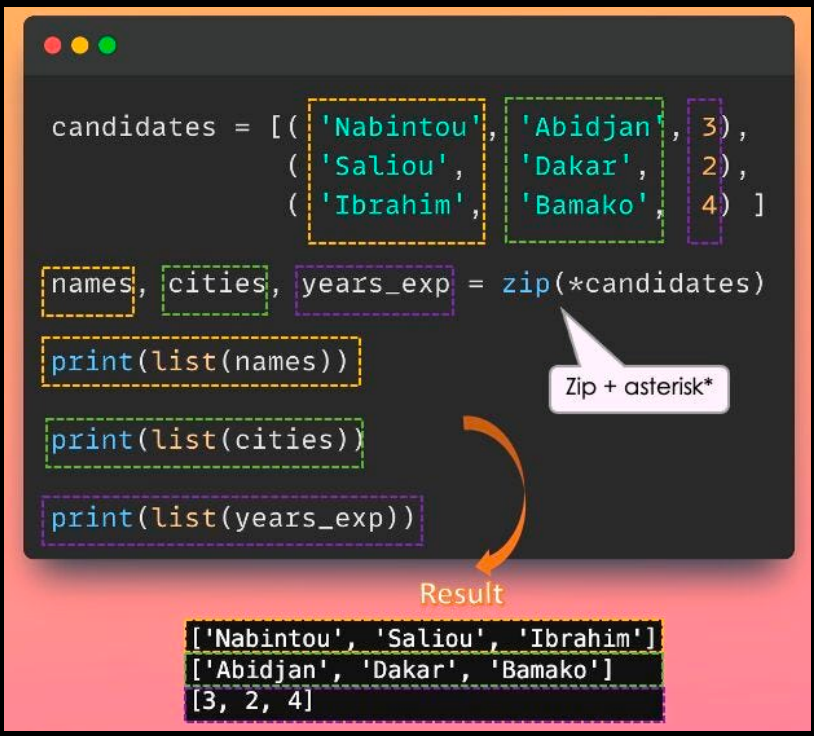
15. **Create multiple lists from aggregated elements**

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 🔁

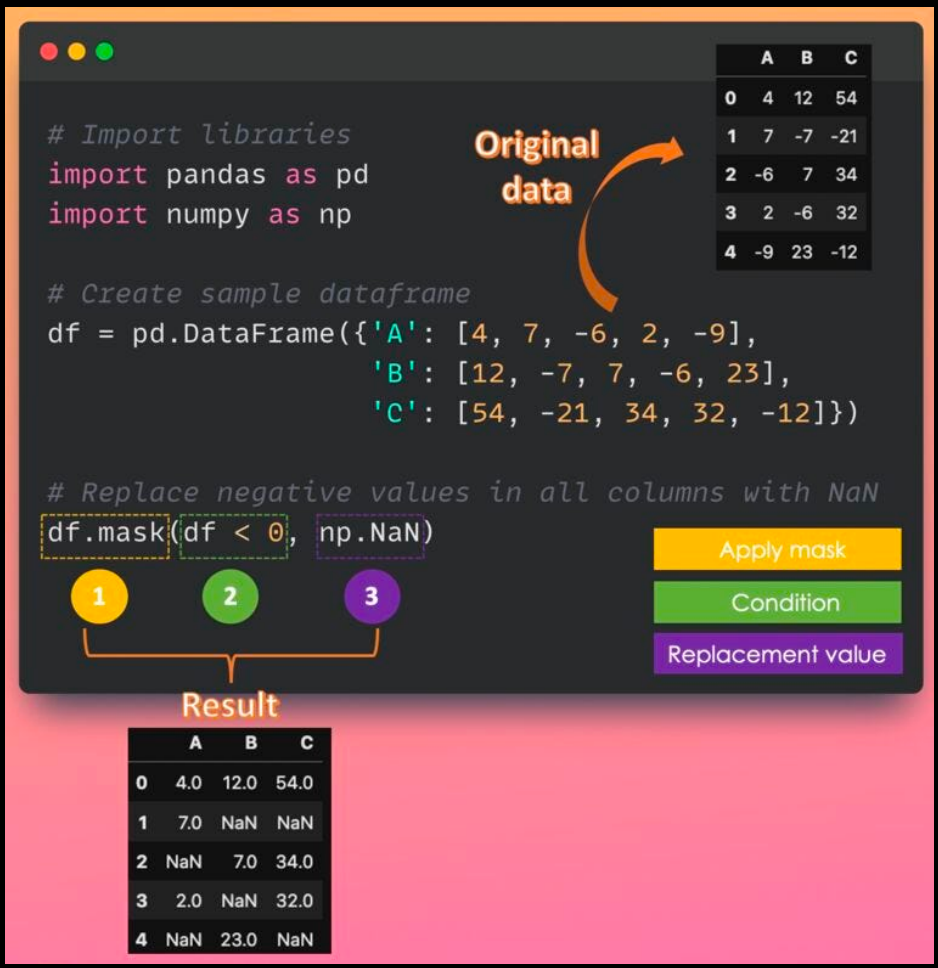
✅ Just combine the **zip**() function with **asterisk** *



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.



	A	B	C
0	4	12	54
1	7	-7	-21
2	-6	7	34
3	2	-6	32
4	-9	23	-12

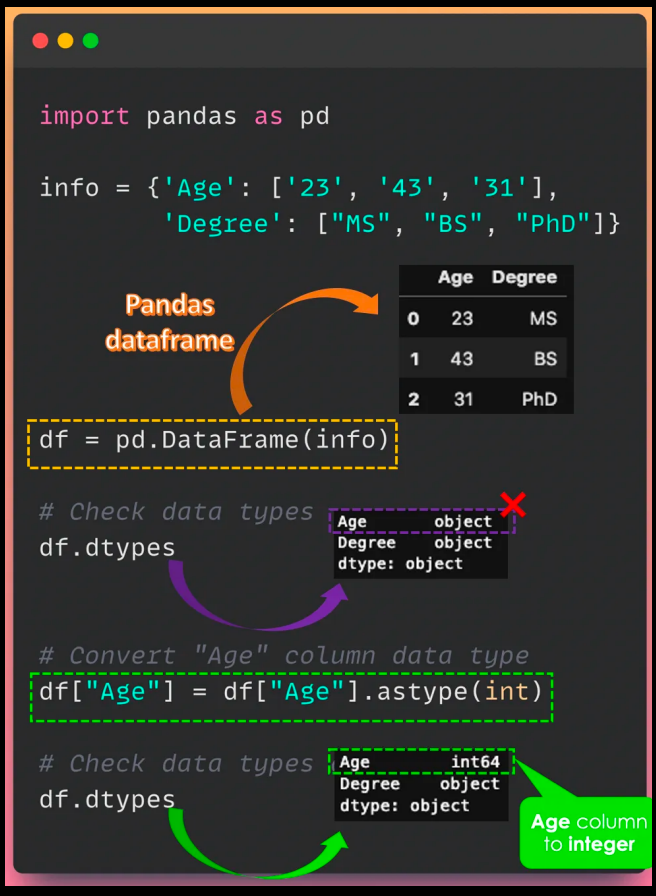
	A	B	C
0	4.0	12.0	54.0
1	7.0	NaN	NaN
2	NaN	7.0	34.0
3	2.0	NaN	32.0
4	NaN	23.0	NaN

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).



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

```python
import pandas as pd

candidates_1 = {
        'Age': ["23", "43", "31"],
        'Degree': ["MS", "BS", "PhD"]
       }

candidates_2 = {
        'Age': [23, 43, 31],
        'Degree': ["MS", "BS", "PhD"]
       }

first_candidates = pd.DataFrame(candidates_1)
second_candidates = pd.DataFrame(candidates_2)

# Check the similarity of the two datasets
first_candidates.equals(second_candidates)
```

- Same name ✅
- Same values ✅
- Different Format 🚫

	Age	Degree
0	23	MS
1	43	BS
2	31	PhD

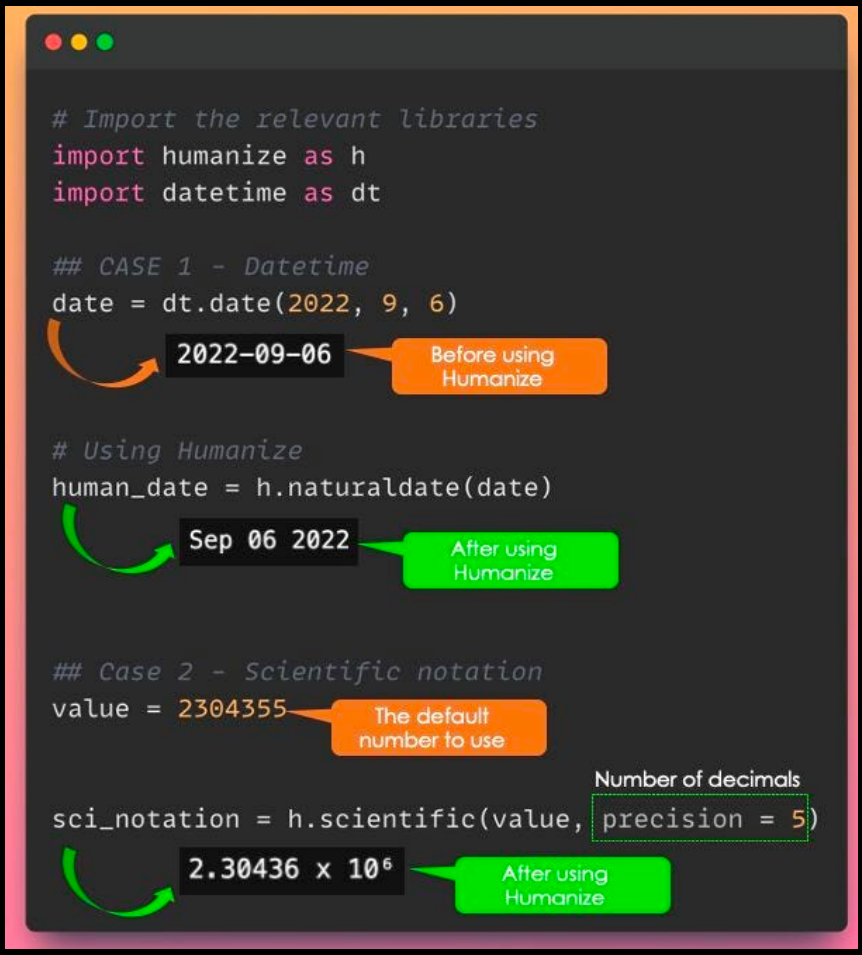
	Age	Degree
0	23	MS
1	43	BS
2	31	PhD

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.



20. **Convert natural language to numerical values**

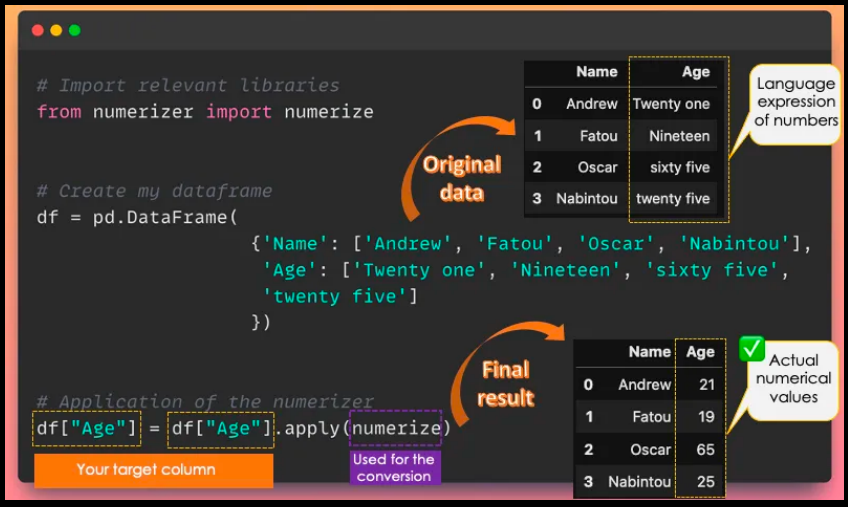
Natural language 🗣️ is everywhere 🌐, even in our DataFrames.

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.

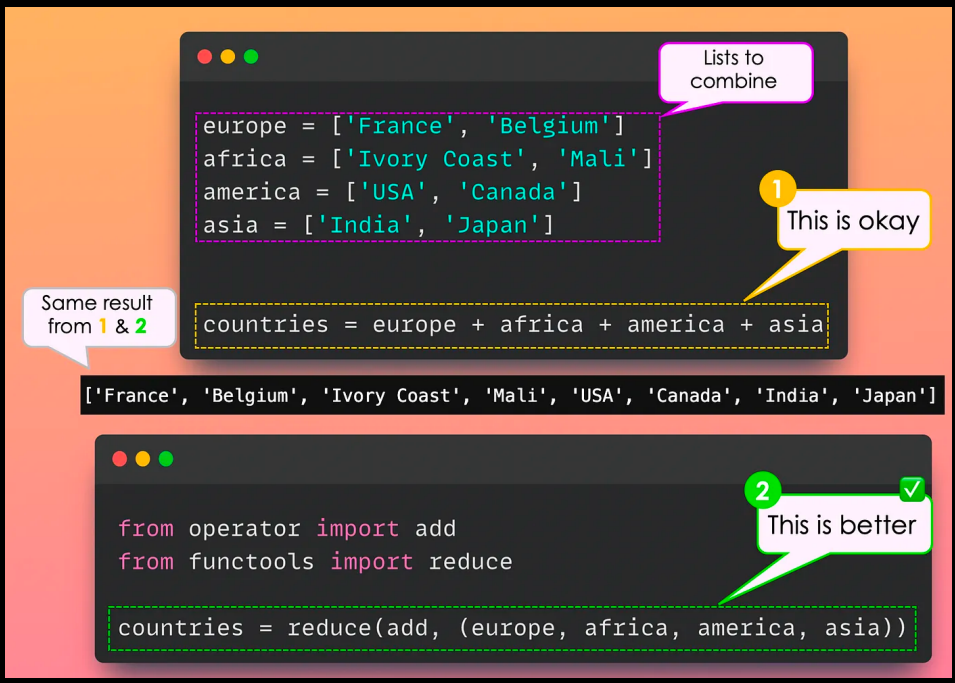


21. **Combine multiple lists**

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.



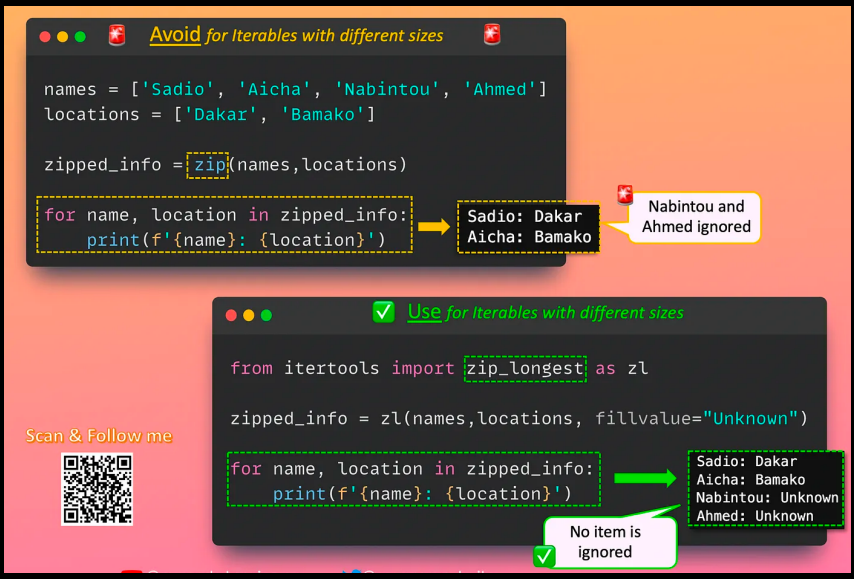
22. **Zip Iterables of different sizes**

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.



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.

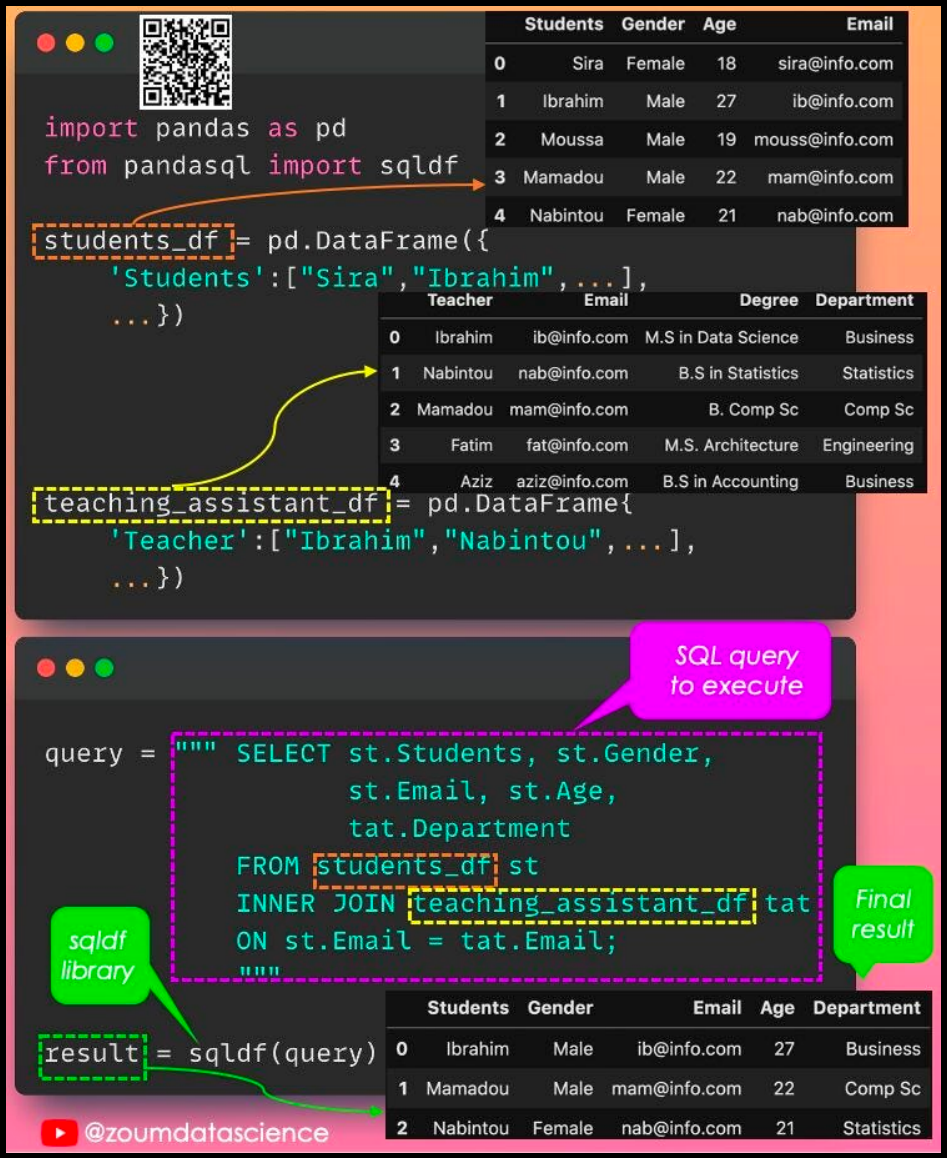
What about SQL?

Even though not everyone uses it in their daily life (because not every company has necessary 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



	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

	Teacher	Email	Degree	Department
0	Ibrahim	ib@info.com	M.S in Data Science	Business
1	Nabintou	nab@info.com	B.S in Statistics	Statistics
2	Mamadou	mam@info.com	B. Comp Sc	Comp Sc
3	Fatim	fat@info.com	M.S. Architecture	Engineering
4	Aziz	aziz@info.com	B.S in Accounting	Business

	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

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.

```python
import pandas as pd

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

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)
```

Before: Data with missing values

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

A	B	C
1	4	7
2	5	8
3	6	9

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.


Unstructured data
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 ...
import pandas as pd
import re
candidates_df = pd.DataFrame({
'Details': [
'Adja Kone: has Master in Statistics and is 23 years old',
'Issa Cisse: has Master in Economics and is 25 years old',
'Frank Kouadio: has PhD in Finance and is 30 years old',
'Fanta Traore: has PhD in Statistics and is 30 years old'
]})
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)
Pattern of desired data format
Apply pattern to extract key information
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
@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.

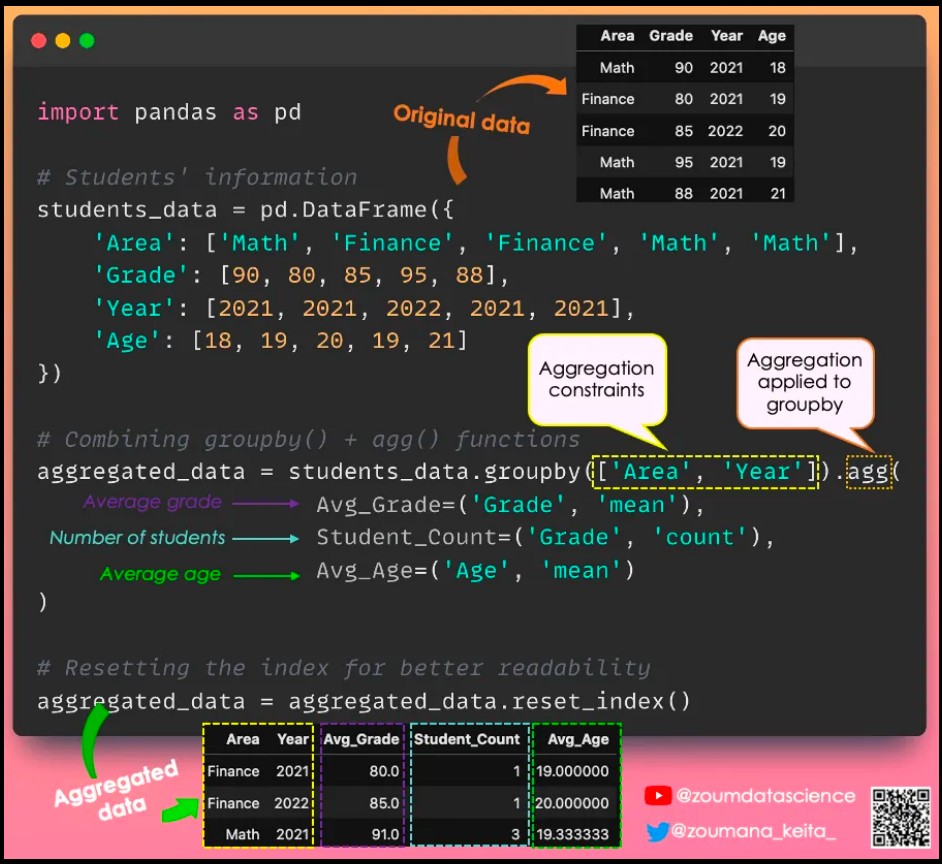
✅ You can combine **groupby**() **and agg**() **functions** from Pandas in one line of code.

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:

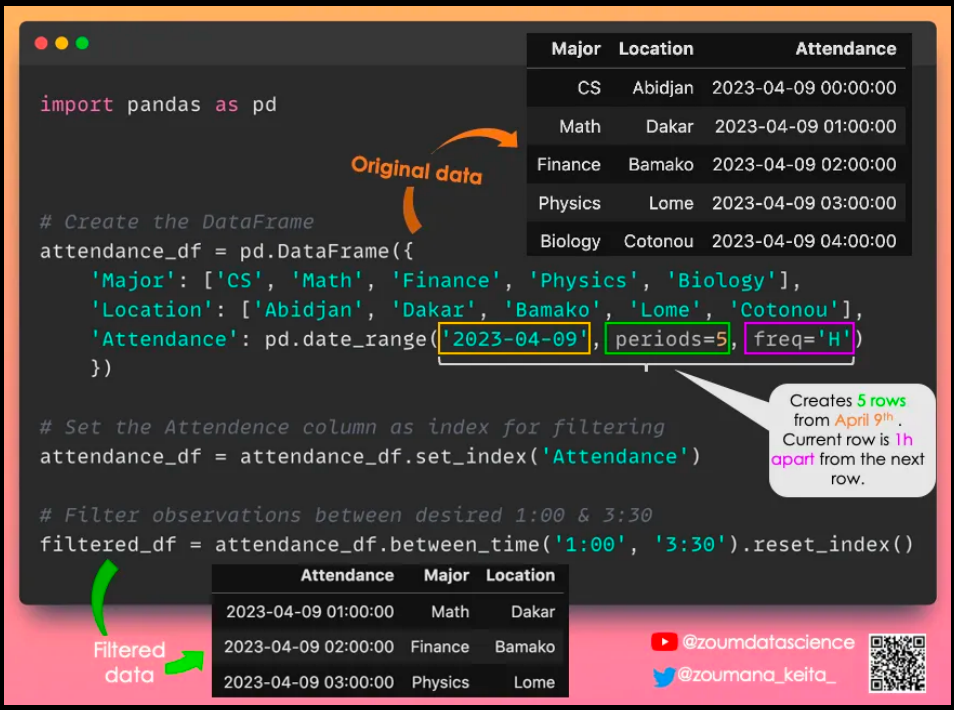
→ The number of students
→ The average grade
→ The average age


Original data
Area Grade Year Age
Math 90 2021 18
Finance 80 2021 19
Finance 85 2022 20
Math 95 2021 19
Math 88 2021 21
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]
})
Aggregation constraints
Aggregation applied to groupby
# Combining groupby() + agg() functions
aggregated_data = students_data.groupby(['Area', 'Year']).agg(
Average grade
Avg_Grade=('Grade', 'mean'),
Number of students
Student_Count=('Grade', 'count'),
Average age
Avg_Age=('Age', 'mean')
)
# Resetting the index for better readability
aggregated_data = aggregated_data.reset_index()
Aggregated data
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
@zoumdatascience
@zoumana_keita_

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



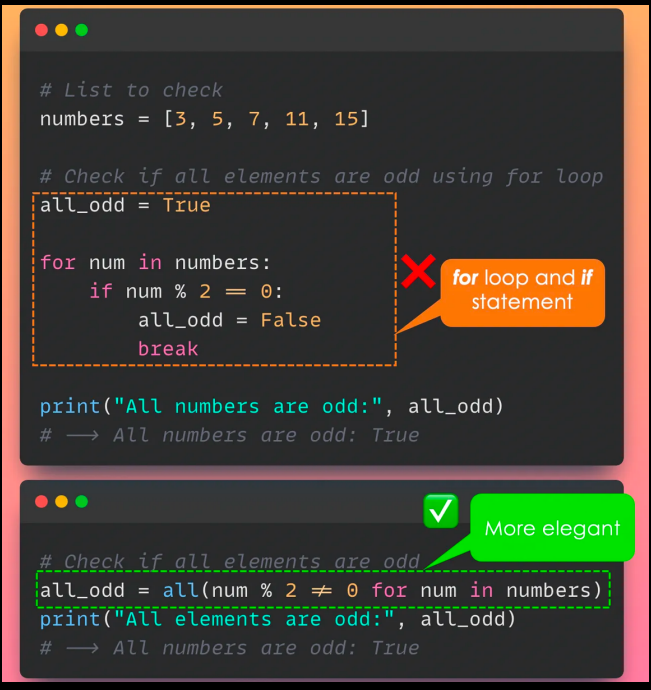
28. **Check if all elements meet a certain condition**

❌ 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:

1️⃣ Either use for loop and if statement.
OR
2️⃣ 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
```

```
# 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
```

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.

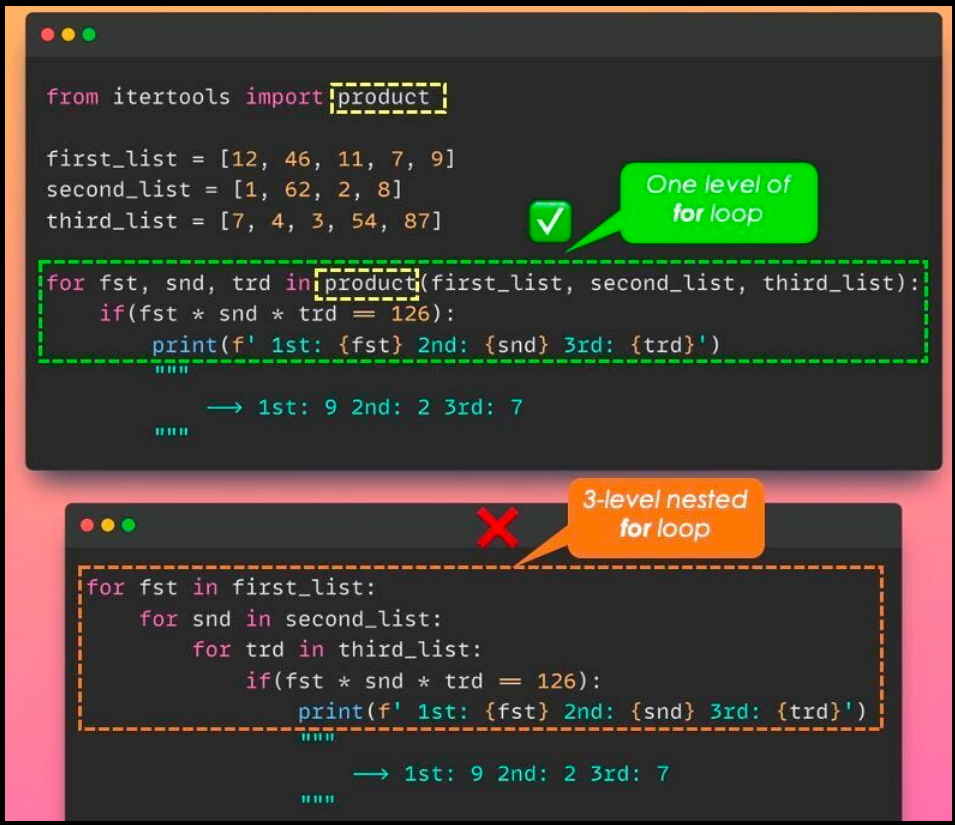
✅ Then use the any() built-in function which is more elegant than using for loop and if statement.

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
"""

3-level nested for loop
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
"""

30. **Automatically handle index in a list**

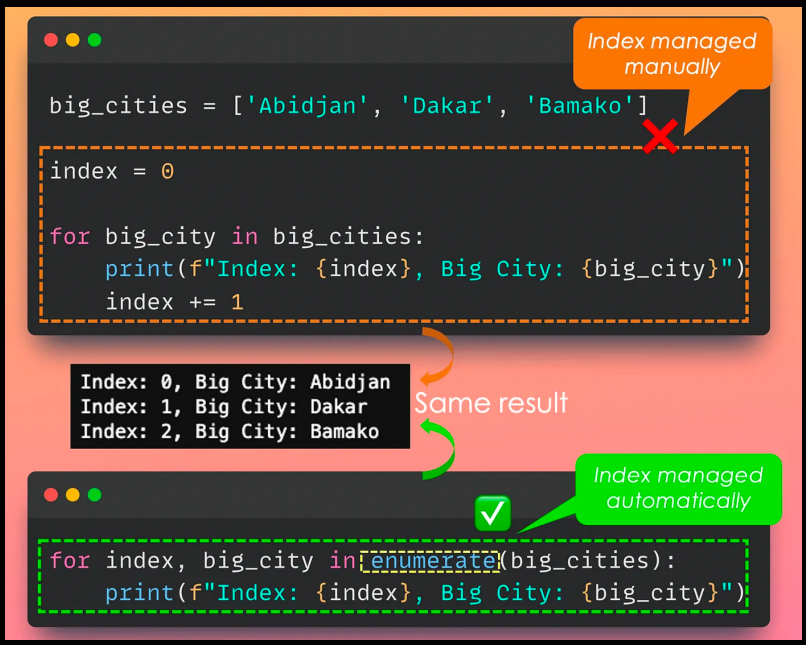
Imagine you have to access elements in a list and their indexes at the same time.

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.



Thanks for reading!

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

Newsletter 👇

ZoumDataScience.com