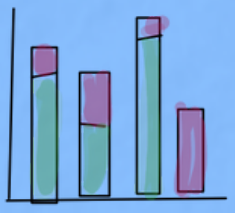
FREE

DATA SCIENCE

FULL ARCHIVE

200+ Python & Data Science Tips



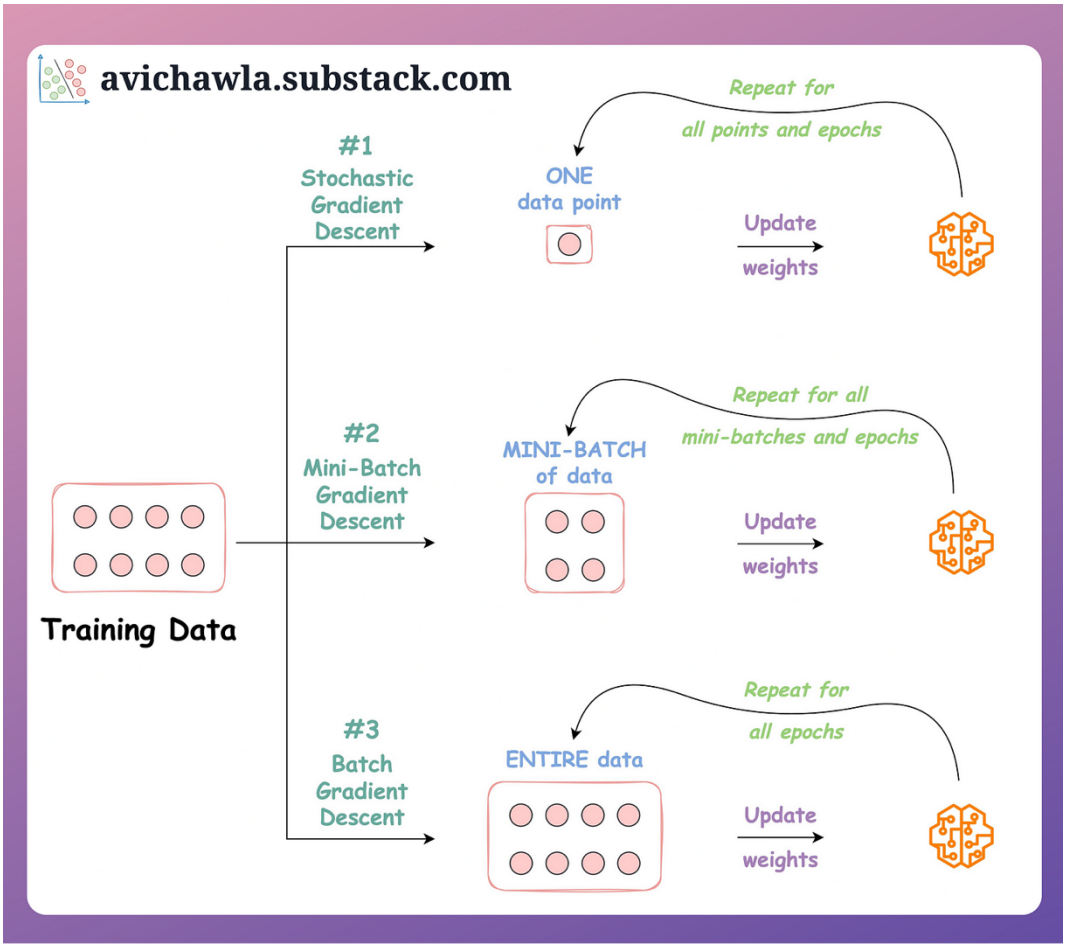



Daily Dose of Data Science

avichawla.substack.com

Table of Contents

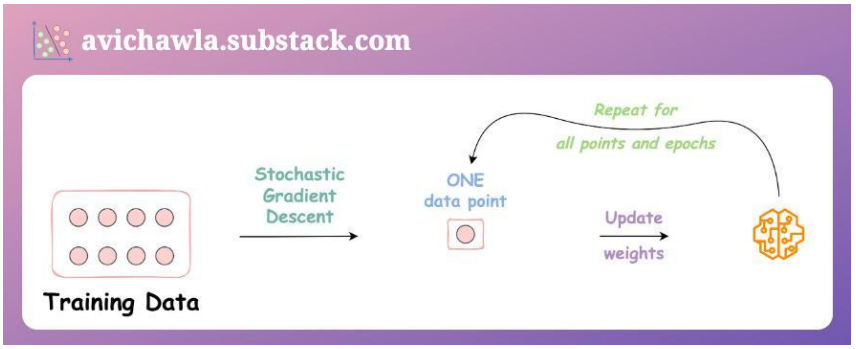
A Visual Guide to Stochastic, Mini-batch, and Batch Gradient Descent



Gradient descent is a widely used optimization algorithm for training machine learning models.

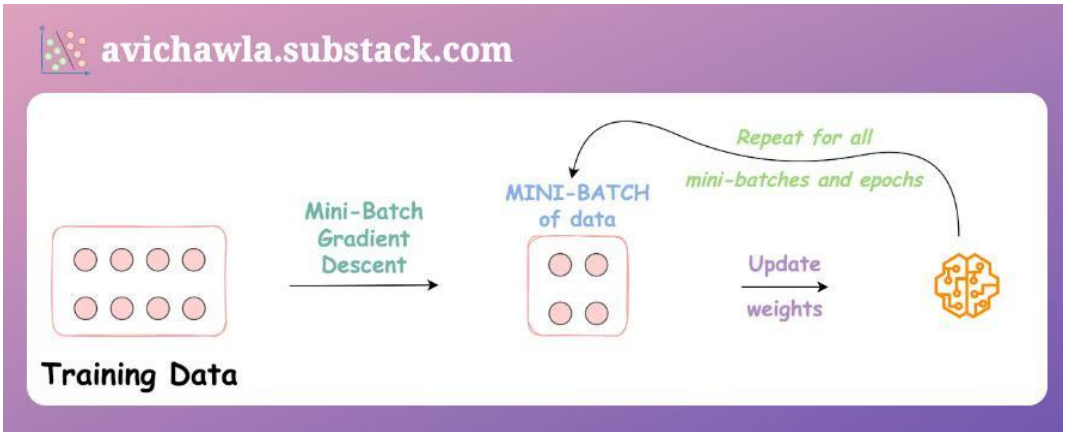
Stochastic, mini-batch, and batch gradient descent are three different variations of gradient descent, and they are distinguished by the number of data points used to update the model weights at each iteration.

🔷 Stochastic gradient descent: Update network weights using one data point at a time.



- **Advantages:**
  - Easier to fit in memory.
  - Can converge faster on large datasets and can help avoid local minima due to oscillations.
- **Disadvantages:**
  - Noisy steps can lead to slower convergence and require more tuning of hyperparameters.
  - Computationally expensive due to frequent updates.
  - Loses the advantage of vectorized operations.

🔷 Mini-batch gradient descent: Update network weights using a few data points at a time.

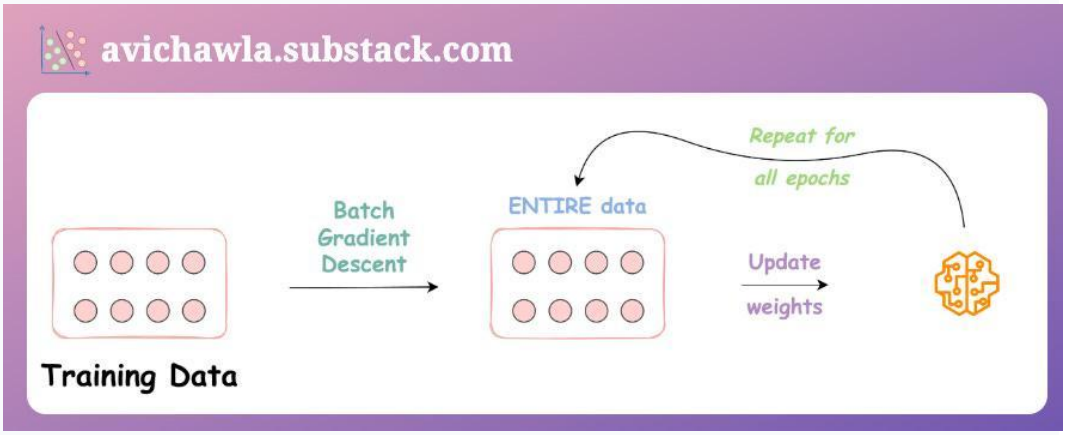


- **Advantages:**
  - More computationally efficient than batch gradient descent due to vectorization benefits.

- Less noisy updates than stochastic gradient descent.

- **Disadvantages:**
  - Requires tuning of batch size.
  - May not converge to a global minimum if the batch size is not well-tuned.

🔷 Batch gradient descent: Update network weights using the entire data at once.

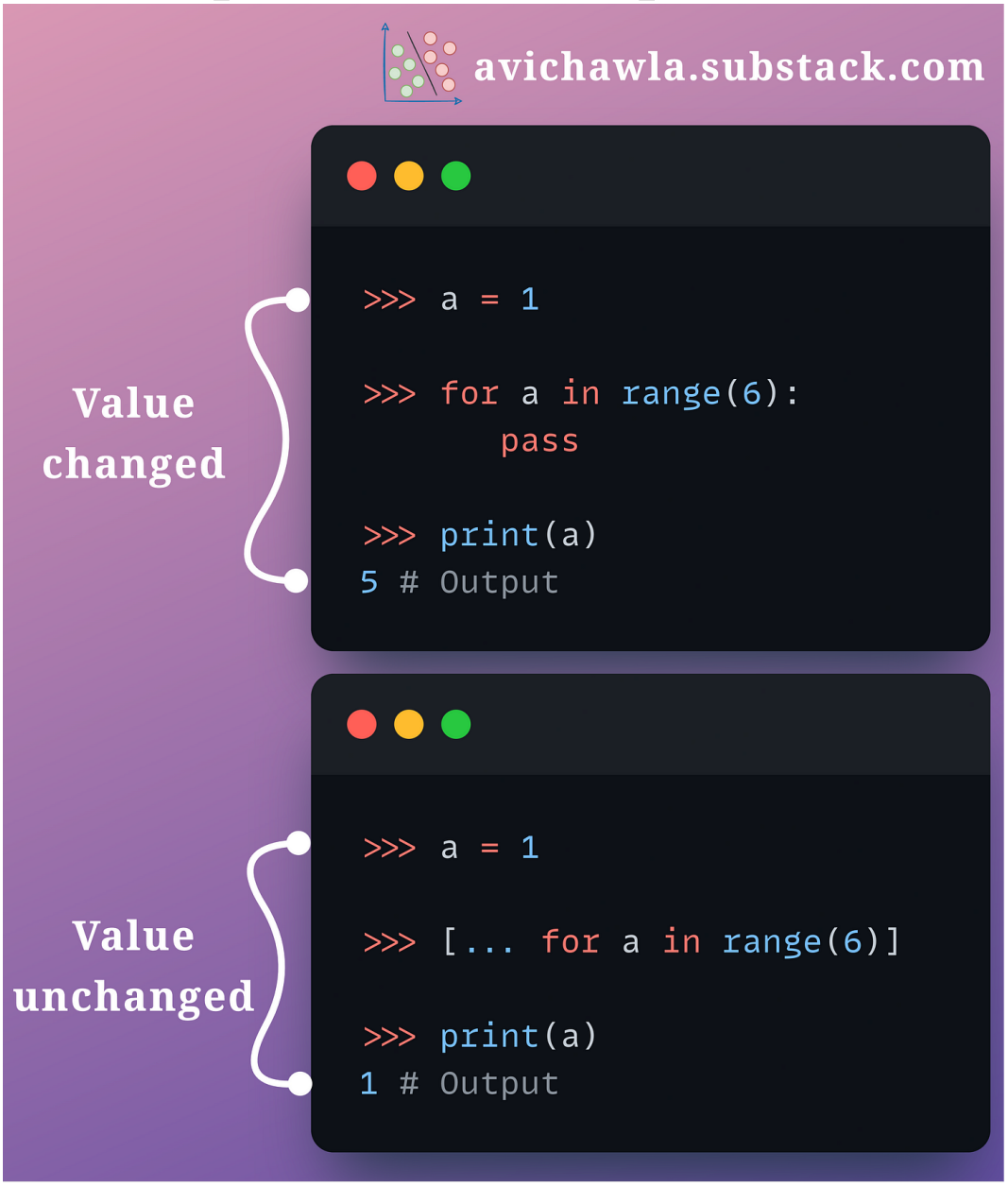


- **Advantages:**
  - Less noisy steps taken towards global minima.
  - Can benefit from vectorization.
  - Produces a more stable convergence.

- **Disadvantages:**
  - Enforces memory constraints for large datasets.
  - Computationally slow as many gradients are computed, and all weights are updated at once.

Over to you: What are some other advantages/disadvantages you can think of? Let me know :)

A Lesser-Known Difference Between For-Loops and List Comprehensions

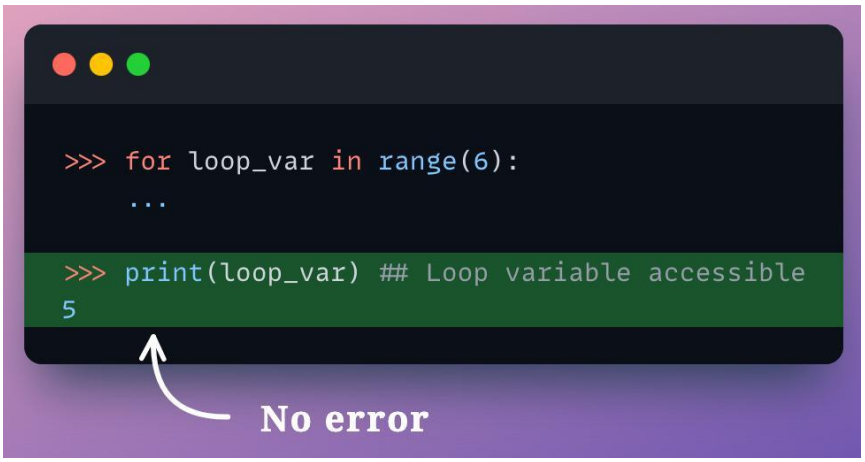


In the code above, the for-loop updated the existing variable (`a`), but list comprehension didn't. Can you guess why? Read more to know.

A loop variable is handled differently in for-loops and list comprehensions.

A for-loop leaks the loop variable into the surrounding scope. In other words, once the loop is over, you can still access the loop variable.

We can verify this below:



In the main snippet above, as the loop variable (`a`) already existed, it was overwritten in each iteration.

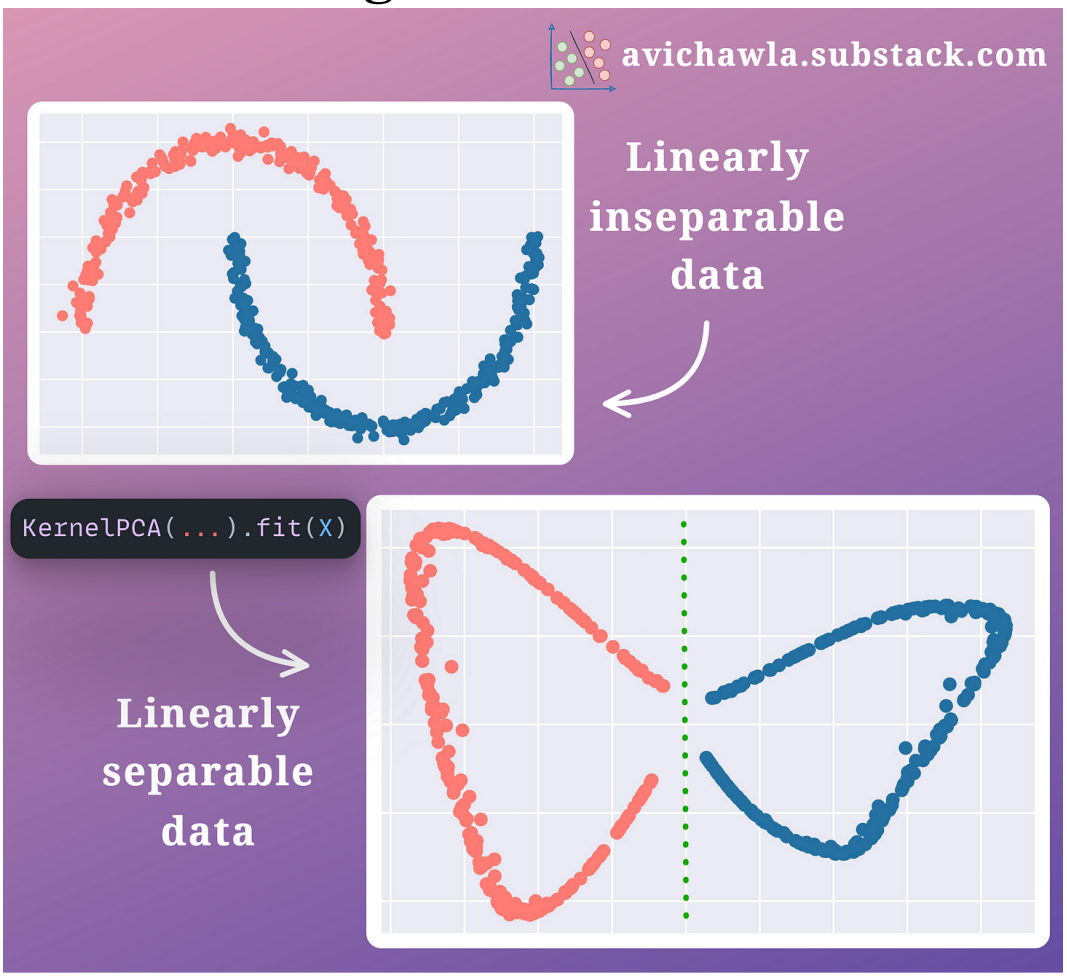
But a list comprehension does not work this way. Instead, the loop variable always remains local to the list comprehension. It is never leaked outside.

We can verify this below:

That is why the existing variable (`a`), which was also used inside the list comprehension, remained unchanged. The list comprehension defined the loop variable (`a`) local to its scope.

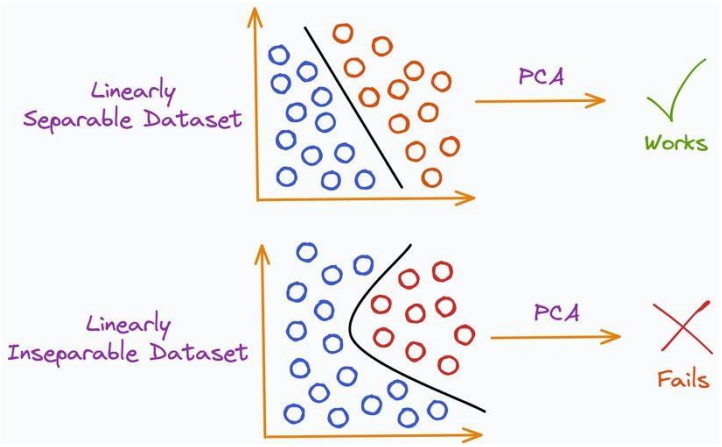
Over to you: What are some other differences that you know of between for-loops and list comprehension?

The Limitation of PCA Which Many Folks Often Ignore



Imagine you have a classification dataset. If you use PCA to reduce dimensions, it is inherently assumed that your data is linearly separable.

But it may not be the case always. Thus, PCA will fail in such cases.



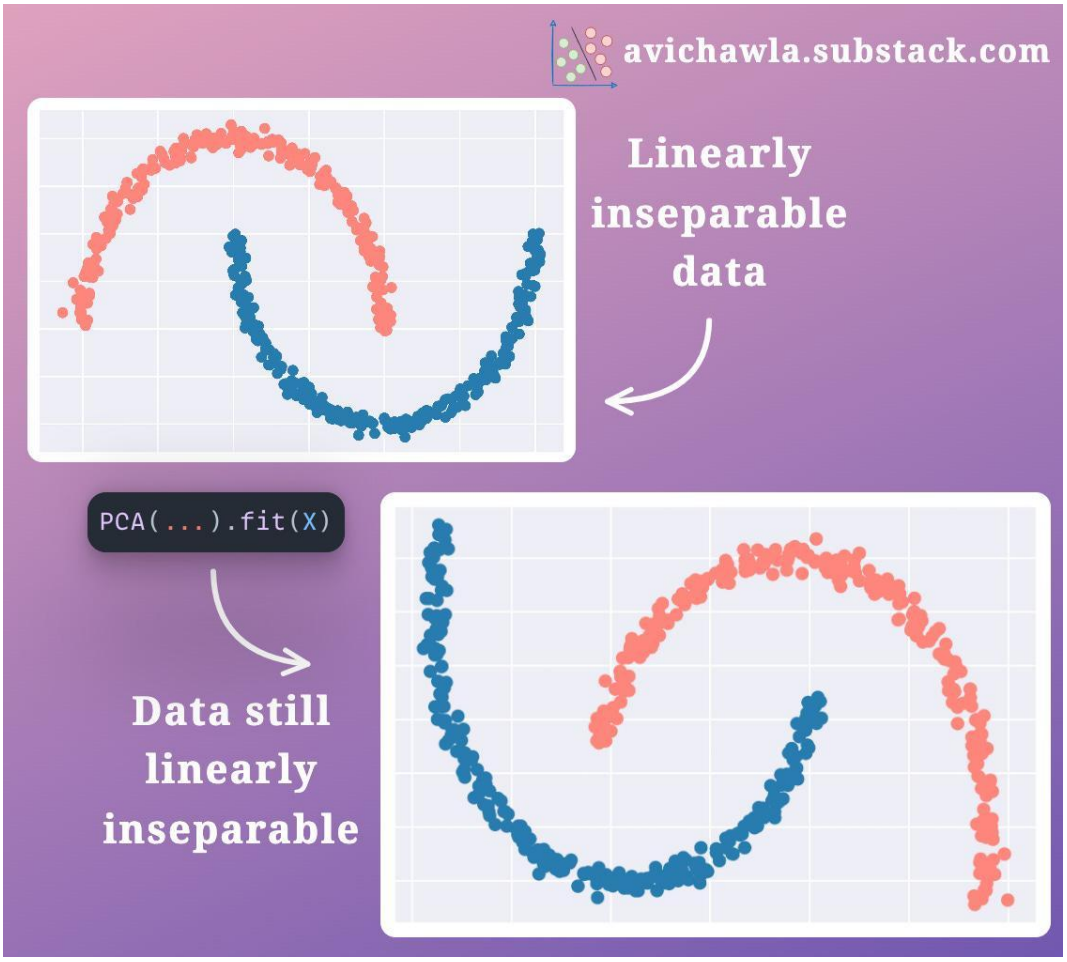
If you wish to read how PCA works, I would highly recommend reading one of my previous posts: A Visual and Overly Simplified Guide to PCA.

To resolve this, we use the kernel trick (or the KernelPCA). The idea is to:

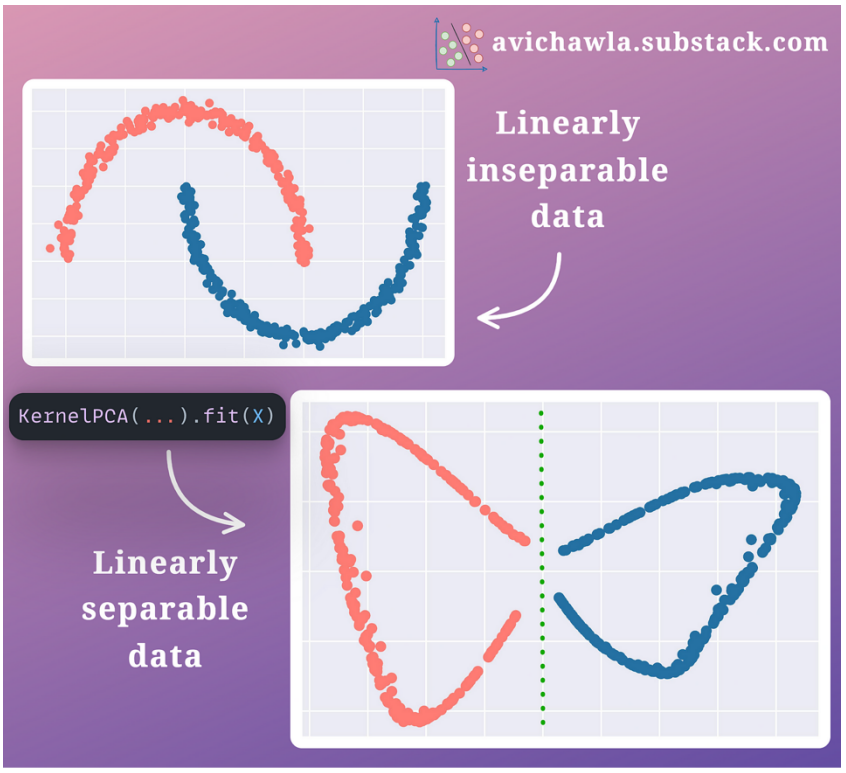
Project the data to another space using a kernel function, where the data becomes linearly separable.

Apply the standard PCA algorithm to the transformed data.

For instance, in the image below, the original data is linearly inseparable. Using PCA directly does not produce any desirable results.



But as mentioned above, KernelPCA first transforms the data to a linearly separable space and then applies PCA, resulting in a linearly separable dataset.



Sklearn provides a KernelPCA wrapper, supporting many popularly used kernel functions. You can find more details here: Sklearn Docs.

Having said that, it is also worth noting that the run-time of PCA is cubic in relation to the number of dimensions of the data.

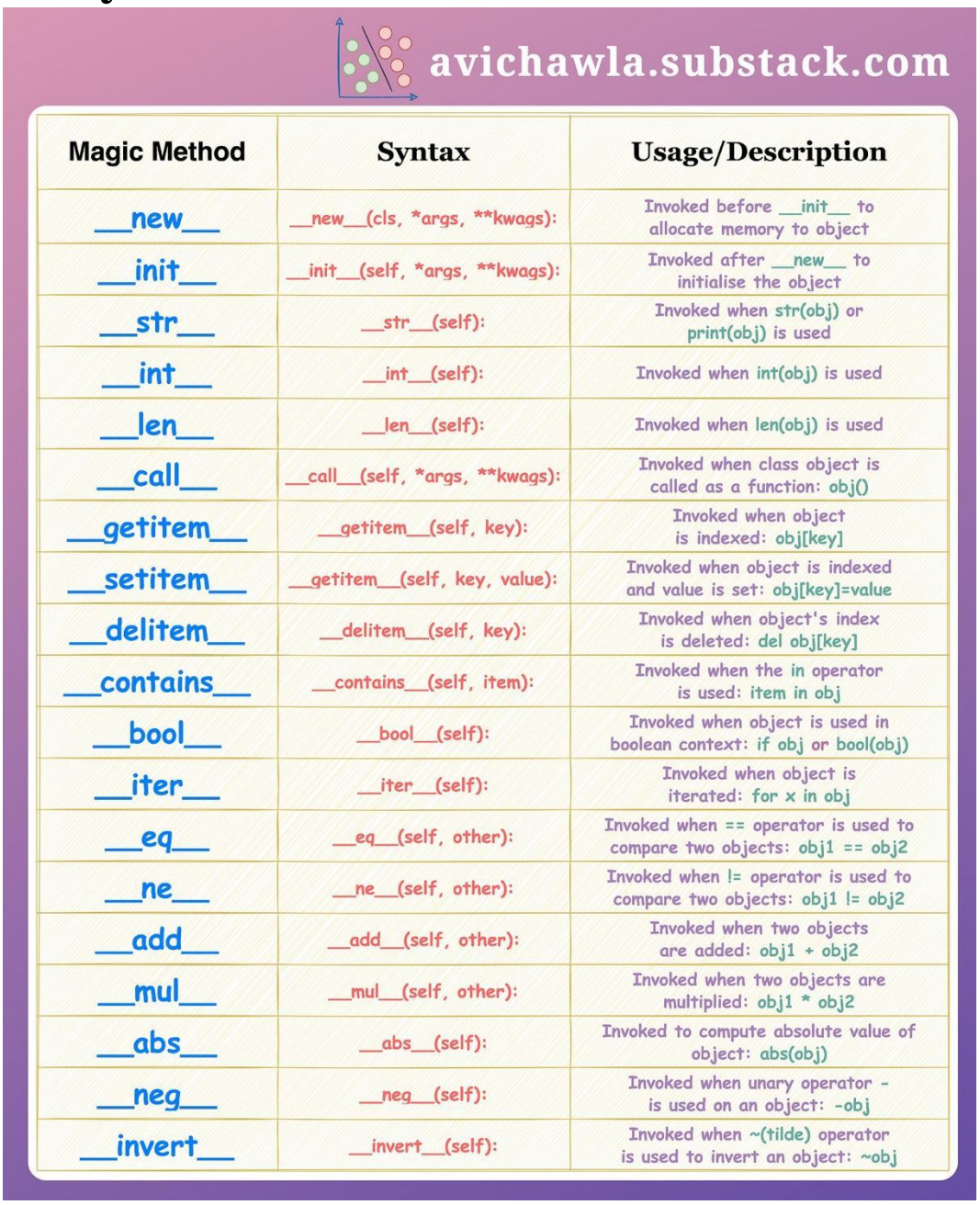
$$Runtime : O(nd^2 + d^3)$$

$$d : dimensions$$

$$n : samples$$

When we use a KernelPCA, typically, the original data (in `n` dimensions) is projected to a new higher dimensional space (in `m` dimensions; `m>n`). Therefore, it increases the overall run-time of PCA.

Magic Methods: An Underrated Gem of Python OOP



Magic Method	Syntax	Usage/Description
__new__	__new__(cls, *args, **kwags):	Invoked before __init__ to allocate memory to object
__init__	__init__(self, *args, **kwags):	Invoked after __new__ to initialise the object
__str__	__str__(self):	Invoked when str(obj) or print(obj) is used
__int__	__int__(self):	Invoked when int(obj) is used
__len__	__len__(self):	Invoked when len(obj) is used
__call__	__call__(self, *args, **kwags):	Invoked when class object is called as a function: obj()
__getitem__	__getitem__(self, key):	Invoked when object is indexed: obj[key]
__setitem__	__getitem__(self, key, value):	Invoked when object is indexed and value is set: obj[key]=value
__delitem__	__delitem__(self, key):	Invoked when object's index is deleted: del obj[key]
__contains__	__contains__(self, item):	Invoked when the in operator is used: item in obj
__bool__	__bool__(self):	Invoked when object is used in boolean context: if obj or bool(obj)
__iter__	__iter__(self):	Invoked when object is iterated: for x in obj
__eq__	__eq__(self, other):	Invoked when == operator is used to compare two objects: obj1 == obj2
__ne__	__ne__(self, other):	Invoked when != operator is used to compare two objects: obj1 != obj2
__add__	__add__(self, other):	Invoked when two objects are added: obj1 + obj2
__mul__	__mul__(self, other):	Invoked when two objects are multiplied: obj1 * obj2
__abs__	__abs__(self):	Invoked to compute absolute value of object: abs(obj)
__neg__	__neg__(self):	Invoked when unary operator - is used on an object: -obj
__invert__	__invert__(self):	Invoked when ~(tilde) operator is used to invert an object: ~obj

Magic Methods (also called **dunder methods**) are special methods defined inside a Python class' implementation.

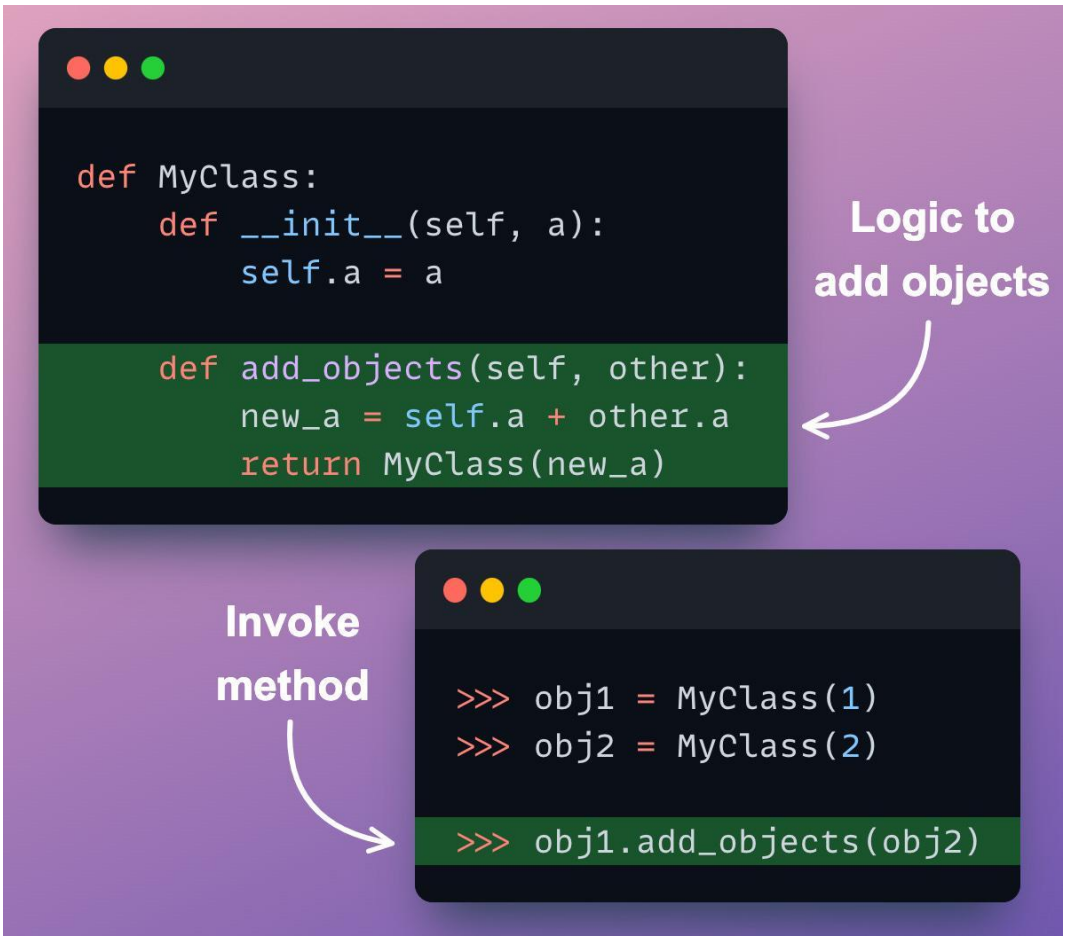
*On a side note, the word "Dunder" is short for **Double Underscore**.*

They are prefixed and suffixed with double underscores, such as `__len__`, `__str__`, and many more.

Magic Methods offer immense flexibility to define the behavior of class objects in certain scenarios.

For instance, say we want to define a custom behavior for adding two objects of our class (`obj1 + obj2`).

An obvious and straightforward to do this is by defining a method, say `add_objects()`, and passing the two objects as its argument.



While the above approach will work, invoking a method explicitly for adding two objects isn't as elegant as using the `+` operator:

This is where magic methods come in. In the above example, implementing the `__add__` magic method will allow you to add the two objects using the `+` operator instead.

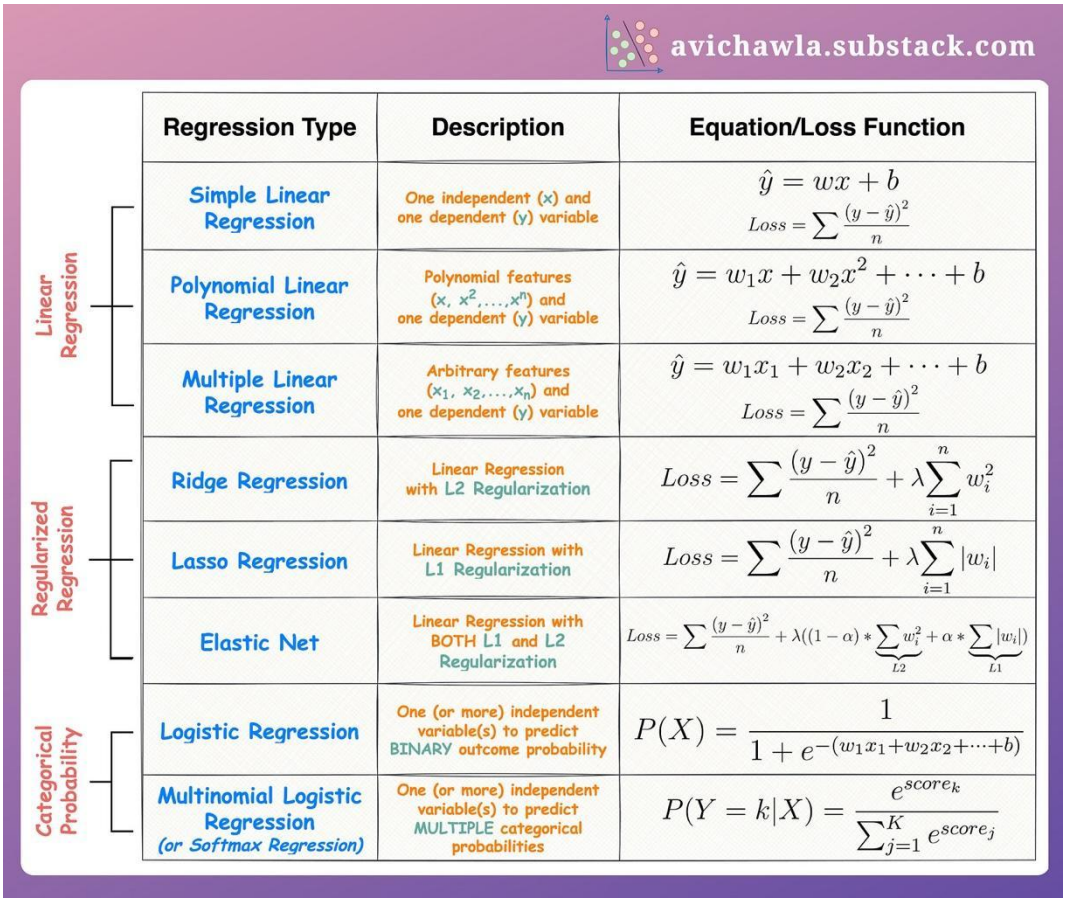
Thus, magic methods allow us to make our classes more intuitive and easier to work with.

As a result, awareness about them is extremely crucial for developing elegant, and intuitive pipelines.

The visual summarizes **~20** most commonly used magic methods in Python.

Over to you: What other magic methods will you include here? Which ones do you use the most? Let me know :)

The Taxonomy Of Regression Algorithms That Many Don't Bother To Remember



Regression algorithms allow us to model the relationship between a dependent variable and one or more independent variables.

After estimating the parameters of a regression model, we can gain insight into how changes in one variable affect another.

Being widely used in data science, an awareness of their various forms is crucial to precisely convey which algorithm you are using.

Here are eight of the most standard regression algorithms described in a single line:

Linear Regression

- **Simple linear regression:** One independent (x) and one dependent (y) variable.

- **Polynomial Linear Regression:** Polynomial features and one dependent (y) variable.
- **Multiple Linear Regression:** Arbitrary features and one dependent (y) variable.

Regularized Regression

- **Lasso Regression:** Linear Regression with L1 Regularization.
- **Ridge Regression:** Linear Regression with L2 Regularization.
- **Elastic Net:** Linear Regression with BOTH L1 and L2 Regularization.

Categorical Probability Prediction

- **Logistic Regression:** Predict binary outcome probability.
- **Multinomial Logistic Regression (or Softmax Regression):** Predict multiple categorical probabilities.

Over to you: What other regressions algorithms will you include here?

A Highly Overlooked Approach To Analysing Pandas DataFrames

Instead of previewing raw DataFrames, styling can make data analysis much easier and faster. Here's how.

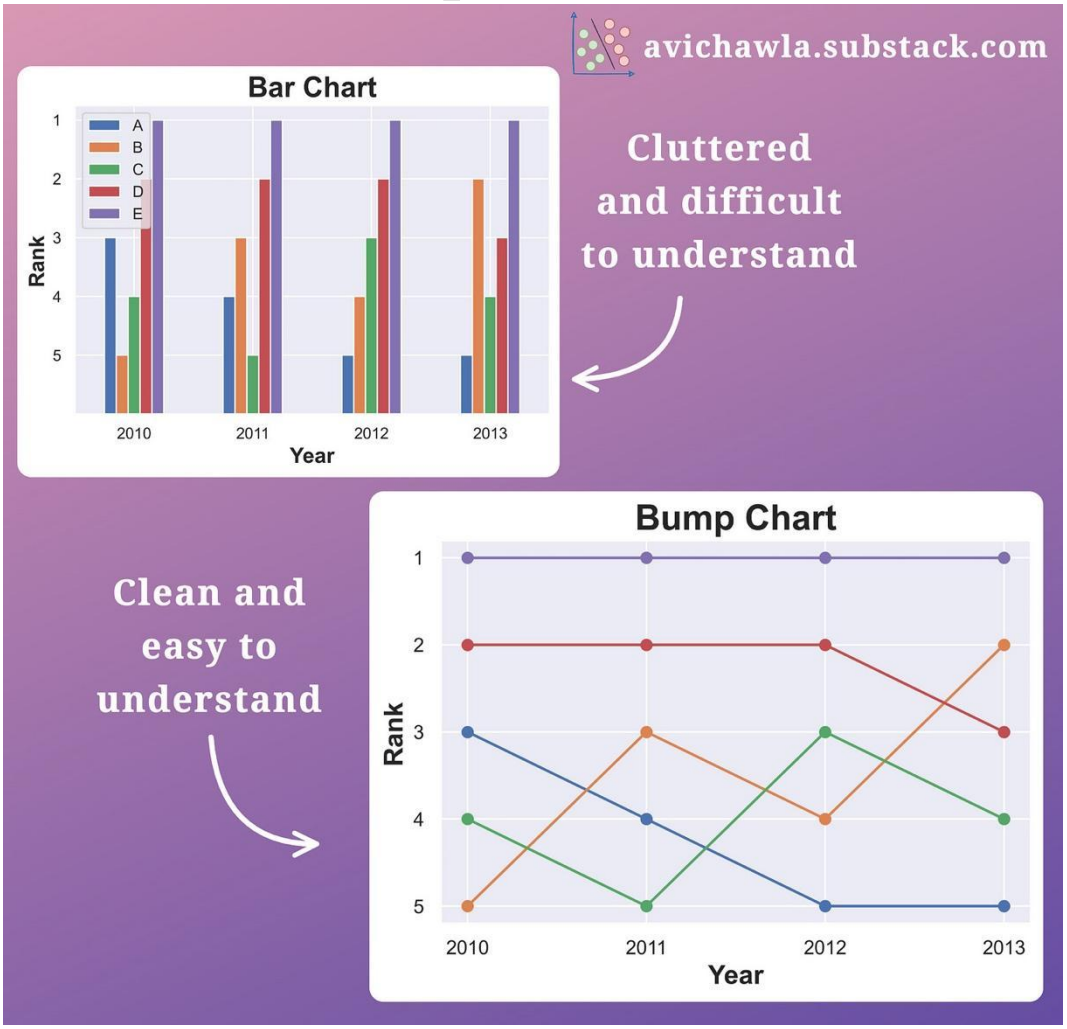
Jupyter is a web-based IDE. Anything you print is rendered using HTML and CSS.

This means you can style your output in many different ways.

To style Pandas DataFrames, use its Styling API (**df.style**). As a result, the DataFrame is rendered with the specified styling.

Read more here: Documentation.

Visualise The Change In Rank Over Time With Bump Charts



When visualizing the change in rank over time, using a bar chart may not be appropriate. Instead, try Bump Charts.

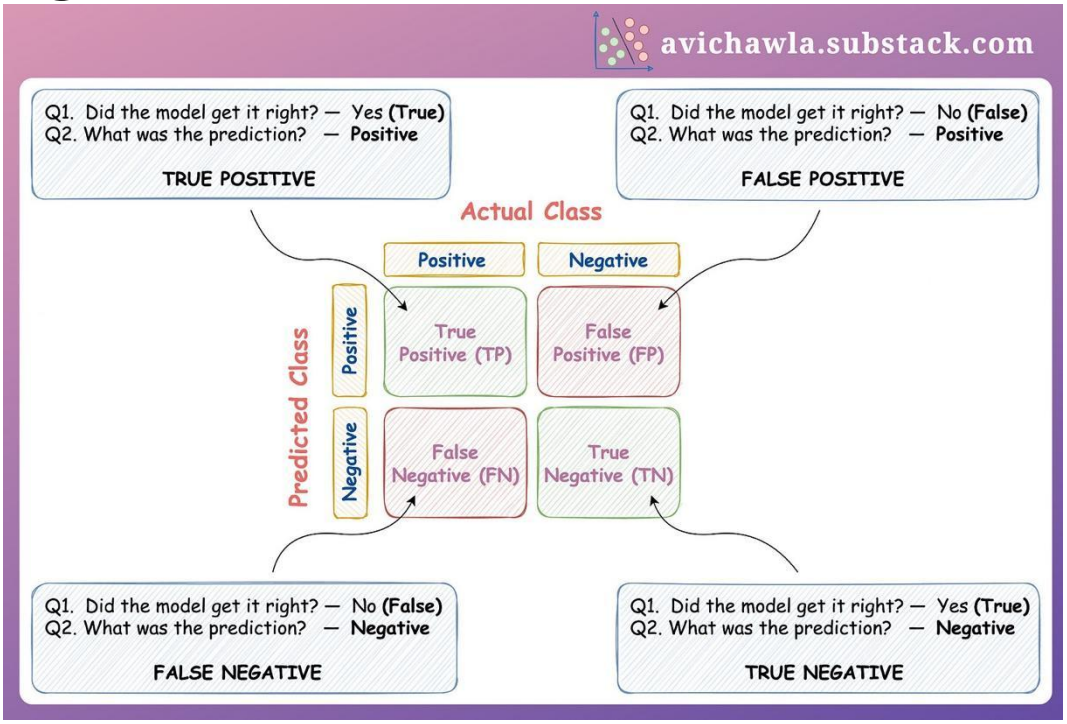
They are specifically used to visualize the rank of different items over time.

In contrast to the commonly used bar chart, they are clear, elegant, and easy to comprehend.

Over to you: What are some other bar chart alternatives to try in such cases? Let me know :)

Find the code for creating a bump chart in Python here: Notebook.

Use This Simple Technique To Never Struggle With TP, TN, FP and FN Again



Do you often struggle to label model predictions as TP, TN, FP and FN, and comprehend them? If yes, here's a simple guide to help you out.

When labeling any prediction, ask yourself two questions:

Did the model get it right? **Answer: yes (or True) / no (or False).**

What was the predicted class? **Answer: Positive/Negative.**

Next, combine the above two answers to get the final label.

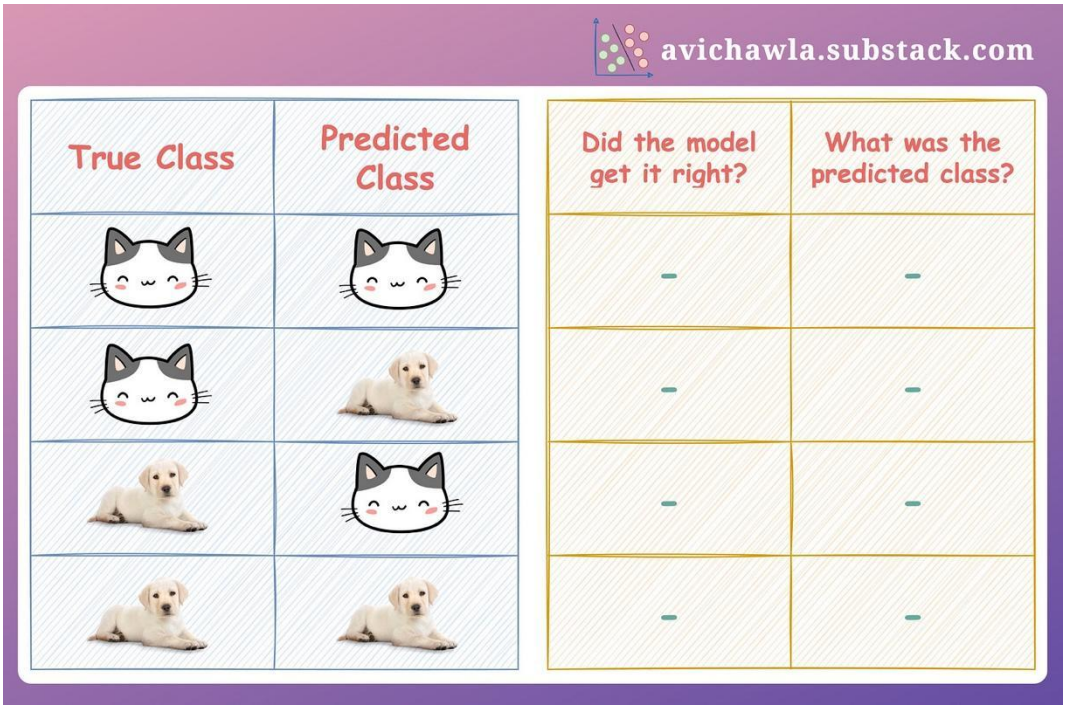
For instance, say the actual and predicted class were positive.

Did the model get it right? The answer is **yes (or TRUE)**.

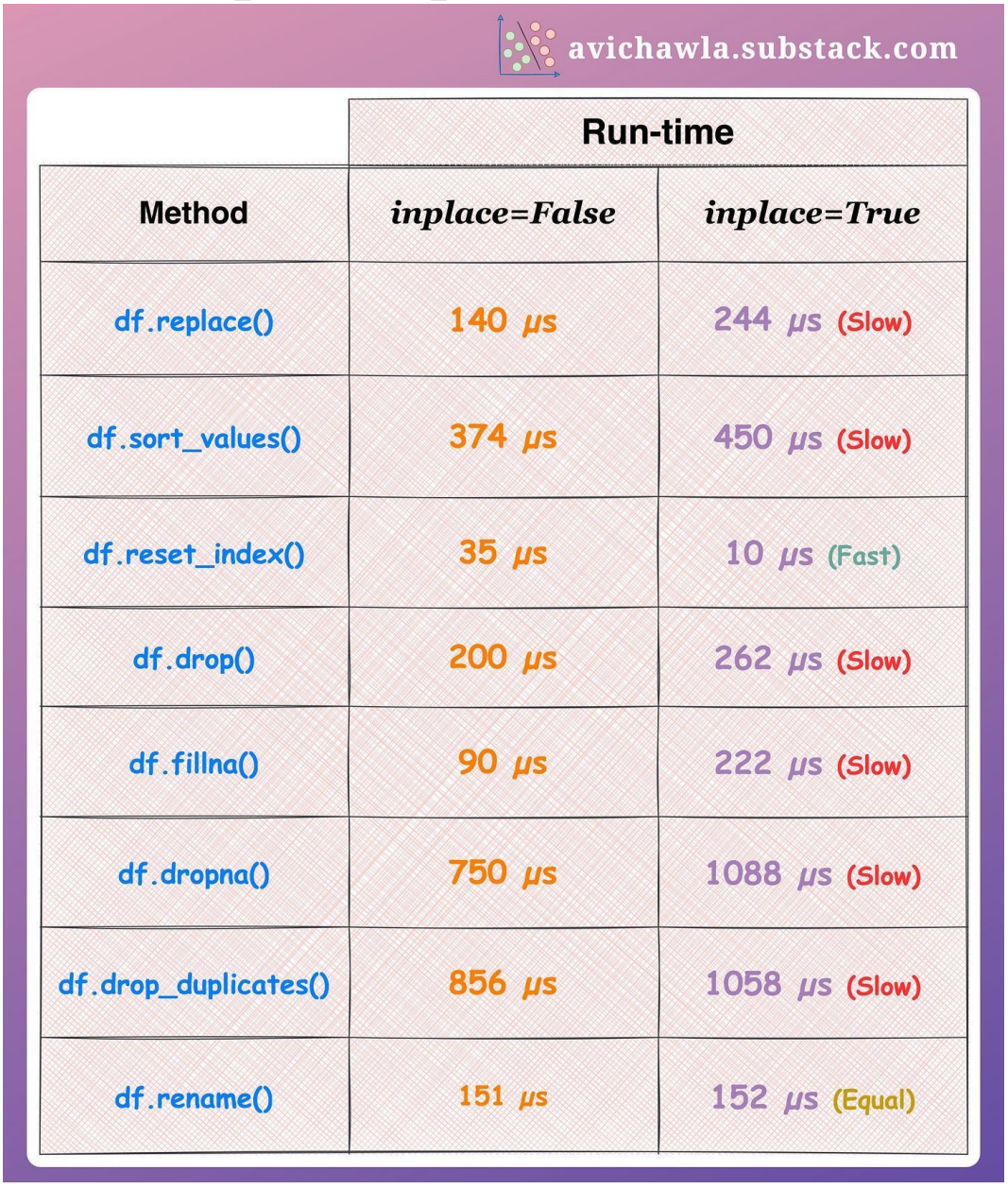
What was the predicted class? The answer is **POSITIVE**.

Final label: **TRUE POSITIVE**.

As an exercise, try labeling the following predictions. Consider the “Cat” class as “Positive” and the “Dog” class as “Negative”.


avichawla.substack.com
True Class
Predicted Class
Did the model get it right?
What was the predicted class?
-
-
-
-
-
-
-
-
-
-

The Most Common Misconception About Inplace Operations in Pandas



	Run-time	
Method	*inplace=False*	*inplace=True*
df.replace()	140 µs	244 µs (Slow)
df.sort_values()	374 µs	450 µs (Slow)
df.reset_index()	35 µs	10 µs (Fast)
df.drop()	200 µs	262 µs (Slow)
df.fillna()	90 µs	222 µs (Slow)
df.dropna()	750 µs	1088 µs (Slow)
df.drop_duplicates()	856 µs	1058 µs (Slow)
df.rename()	151 µs	152 µs (Equal)

Pandas users often modify a DataFrame inplace expecting better performance. Yet, it may not always be efficient. Here's why.

The image compares the run-time of inplace and non-in-place operations. In most cases, inplace operations are slow.

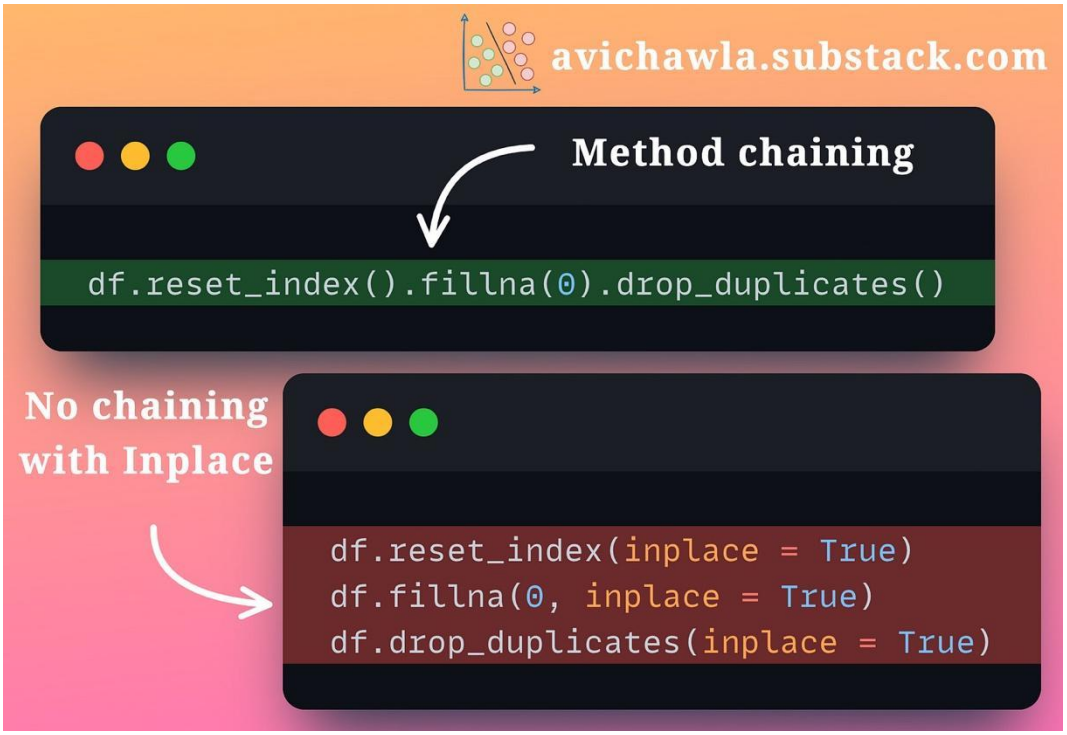
Why?

Contrary to common belief, most inplace operations DO NOT prevent the creation of a new copy. It is just that inplace assigns the copy back to the same address.

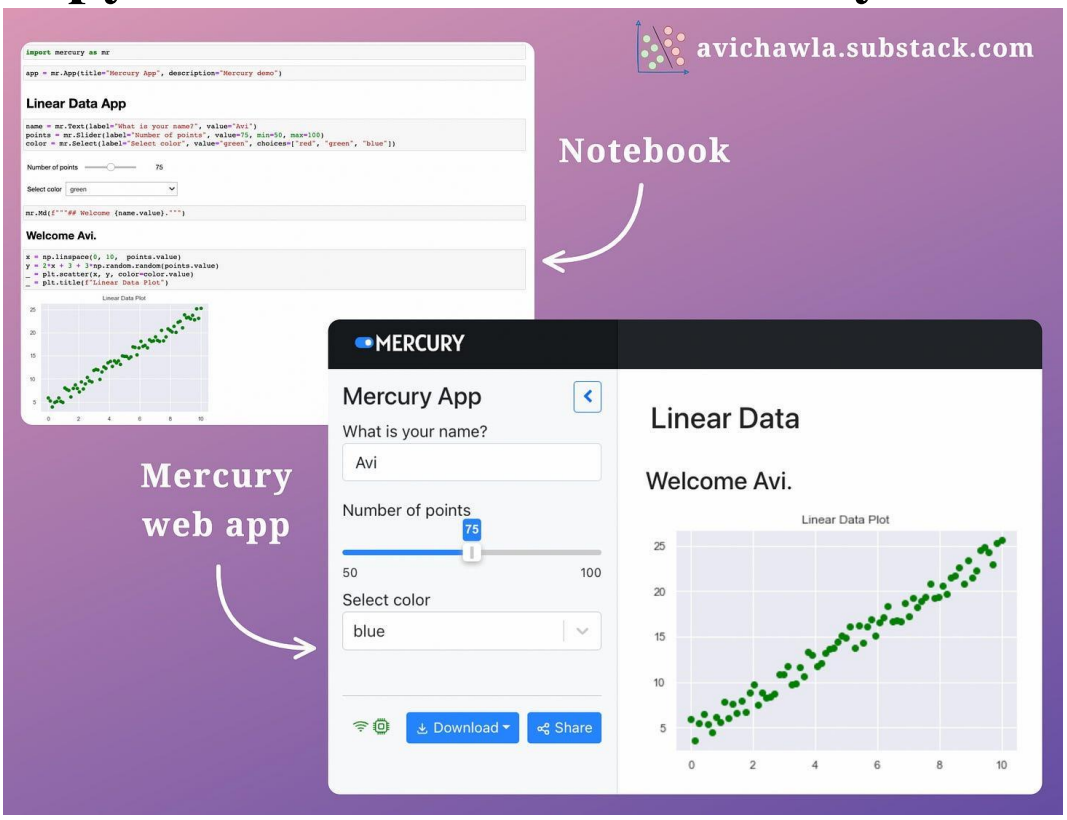
But during this assignment, Pandas performs some extra checks (SettingWithCopy) to ensure that the DataFrame is being modified correctly. This, at times, can be an expensive operation.

Yet, in general, there is no guarantee that an inplace operation is faster.

What's more, inplace operations do not allow chaining multiple operations, such as this:



Build Elegant Web Apps Right From Jupyter Notebook with Mercury

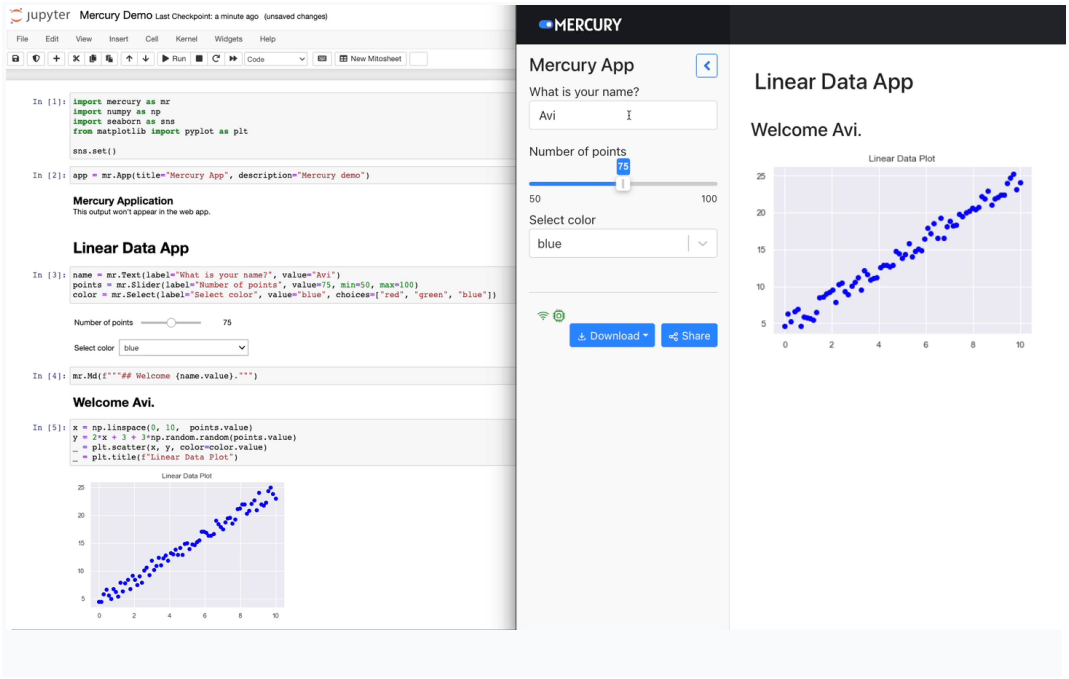


Exploring and sharing data insights using Jupyter is quite common for data folks. Yet, an interactive app is better for those who don't care about your code and are interested in your results.

While creating presentations is possible, it can be time-consuming. Also, one has to leave the comfort of a Jupyter Notebook.

Instead, try Mercury. It's an open-source tool that converts your jupyter notebook to a web app in no time. Thus, you can create the web app without leaving the notebook.

A quick demo is shown below:



What's more, all updates to the Jupyter Notebook are instantly reflected in the Mercury app.

In contrast to the widely-adopted streamlit, web apps created with Mercury can be:

Exported as PDF/HTML.

Showcased as a live presentation.

Secured with authentication to restrict access.

Become A Bilingual Data Scientist With These Pandas to SQL Translations

Operation	Pandas	SQL
Read CSV	pd.read_csv(file)	LOAD DATA INFILE 'data.csv' INTO TABLE table FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n' IGNORE 1 ROWS;
Print first 10 (or k) rows	df.head(10)	SELECT * FROM table LIMIT 10;
Dimensions	df.shape	SELECT count(*) FROM table;
Datatype	df.dtyes	DESCRIBE table;
Filter Data	df[df.column>10]	SELECT * FROM table where column>10;
Select column(s)	df.column	SELECT column FROM table;
Sort	df.sort_values("column")	SELECT * FROM table ORDER BY column;
Fill NaN	df.column.fillna(0)	UPDATE table SET column=0 WHERE column IS NULL;
Join	pd.merge(df1, df2, on ="col", how = "inner")	SELECT * FROM table1 JOIN table2 ON (table1.col = table2.col);
Concatenate	pd.concat((df1, df2))	SELECT * FROM table1 UNION ALL table2;
Group	df.groupby("column"). agg_col.mean()	SELECT column, avg(agg_col) FROM table GROUP BY column;
Unique values	df.column.unique()	SELECT DISTINCT column FROM table;
Rename column	df.rename(columns = {"old_name": "new_name"})	ALTER TABLE table RENAME COLUMN old_name TO new_name;
Delete column	df.drop(columns = ["column"])	ALTER TABLE table DROP COLUMN column;

SQL and Pandas are both powerful tools for data scientists to work with data.

Together, SQL and Pandas can be used to clean, transform, and analyze large datasets, and to create complex data pipelines and models.

Thus, proficiency in both frameworks can be extremely valuable to data scientists.

This visual depicts a few common operations in Pandas and their corresponding translations in SQL.

I have a detailed blog on Pandas to SQL translations with many more examples. Read it here: Pandas to SQL blog.

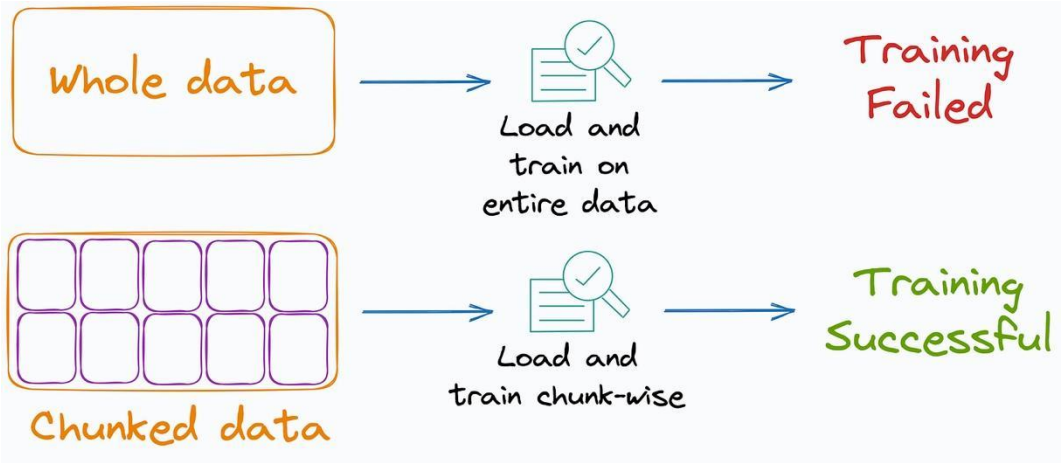
Over to you: What other Pandas to SQL translations will you include here?

A Lesser-Known Feature of Sklearn To Train Models on Large Datasets

It is difficult to train models with sklearn when you have plenty of data. This may often raise memory errors as the entire data is loaded in memory. But here's what can help.

Sklearn implements the **partial_fit** API for various algorithms, which offers incremental learning.

As the name suggests, the model can learn incrementally from a mini-batch of instances. This prevents limited memory constraints as only a few instances are loaded in memory at once.



As shown in the main image, `clf.fit(X, y)` takes the entire data, and thus, it may raise memory errors. But, loading chunks of data and invoking the `clf.partial_fit()` method prevents this and offers seamless training.

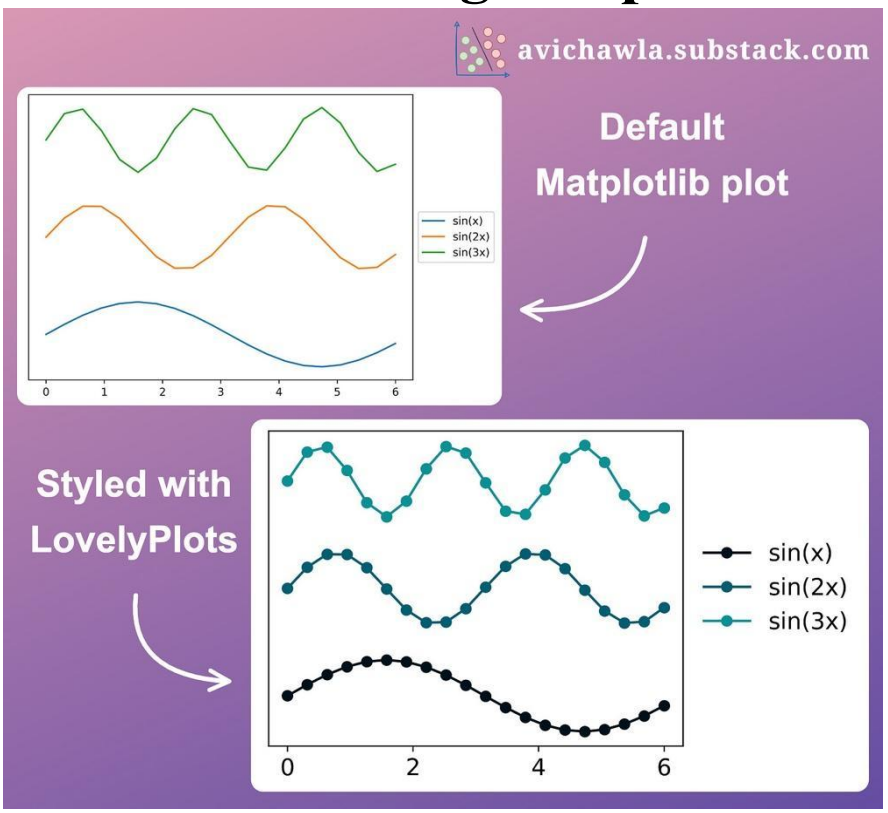
Also, remember that while using the **partial_fit** API, a mini-batch may not have instances of all classes (especially the first mini-batch). Thus, the model will be unable to cope with new/unseen classes in subsequent mini-batches. Therefore, you should pass a list of all possible classes in the `classes` parameter.

Having said that, it is also worth noting that not all sklearn estimators implement the **partial_fit** API. Here's the list:

- **Classification**
  - `sklearn.naive_bayes.MultinomialNB`
  - `sklearn.naive_bayes.BernoulliNB`
  - `sklearn.linear_model.Perceptron`
  - `sklearn.linear_model.SGDClassifier`
  - `sklearn.linear_model.PassiveAggressiveClassifier`
- **Regression**
  - `sklearn.linear_model.SGDRegressor`
  - `sklearn.linear_model.PassiveAggressiveRegressor`
- **Clustering**
  - `sklearn.cluster.MiniBatchKMeans`
- **Decomposition / feature Extraction**
  - `sklearn.decomposition.MiniBatchDictionaryLearning`
  - `sklearn.cluster.MiniBatchKMeans`

Yet, it is surely worth exploring to see if you can benefit from it :)

A Simple One-Liner to Create Professional Looking Matplotlib Plots



The default styling of matplotlib plots appears pretty basic at times. Here's how you can make them appealing.

To create professional-looking plots for presentations, reports, or scientific papers, try LovelyPlots.

It provides many style sheets to improve their default appearance, by simply adding just one line of code.

To install LovelyPlots, run the following command:

```
pip install LovelyPlots
```

Next, import the matplotlib library, and change the style as follows: (You don’t have to import LovelyPlots anywhere)

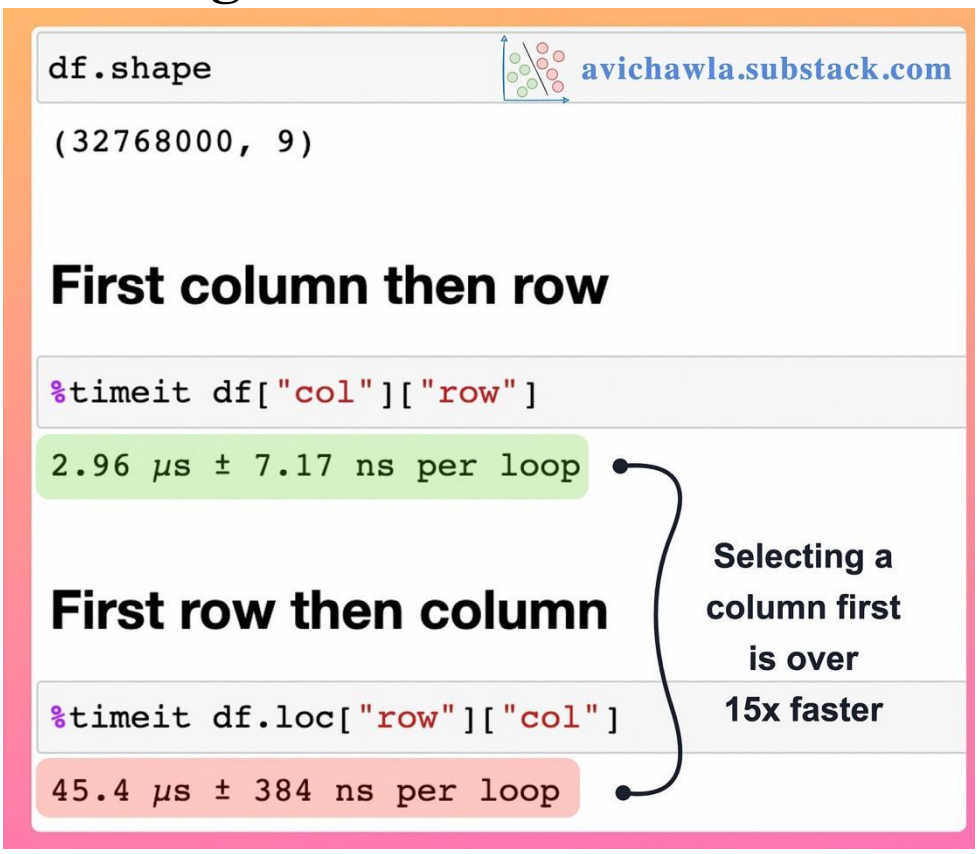
```
import matplotlib.pyplot as plt

plt.style.use(style) ## change to the style provided by LovelyPlots
```

Print the list of all possible styles as follows:

```
plt.style.available
```

Get Started: LovelyPlots Repository.

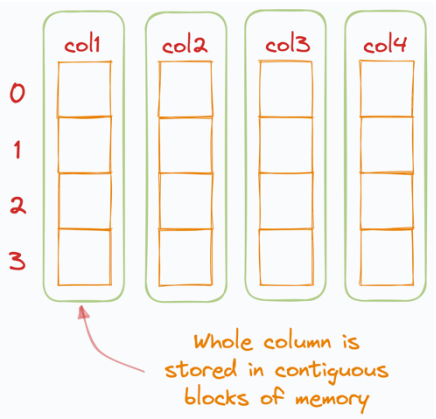
Avoid This Costly Mistake When Indexing A DataFrame



When indexing a dataframe, choosing whether to select a column first or slice a row first is pretty important from a run-time perspective.

As shown above, selecting the column first is over **15 times** faster than slicing the row first. Why?

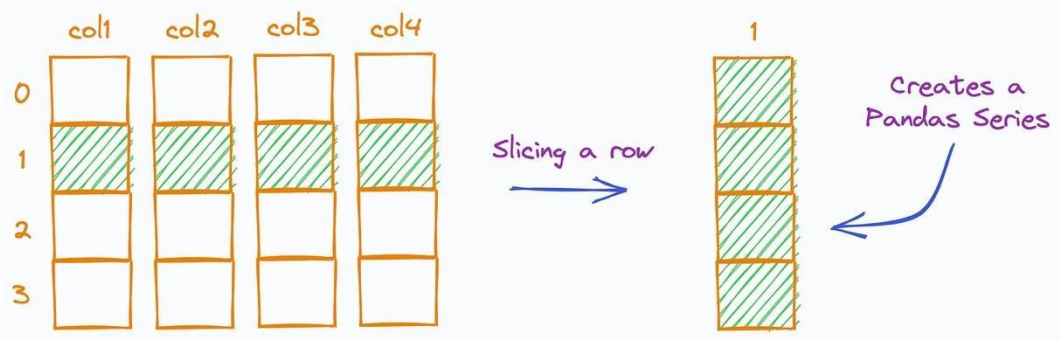
As I have talked before, Pandas DataFrame is a column-major data structure. Thus, consecutive elements in a column are stored next to each other in memory.



As processors are efficient with contiguous blocks of memory, accessing a column is much faster than accessing a row (read more about this in one of my previous posts here).

But when you slice a row first, each row is retrieved by accessing non-contiguous blocks of memory, thereby making it slow.

Also, once all the elements of a row are gathered, Pandas converts them to a Series, which is another overhead.



We can verify this conversion below:

Instead, when you select a column first, elements are retrieved by accessing contiguous blocks of memory, which is way faster. Also, a column is inherently a Pandas Series. Thus, there is no conversion overhead involved like above.

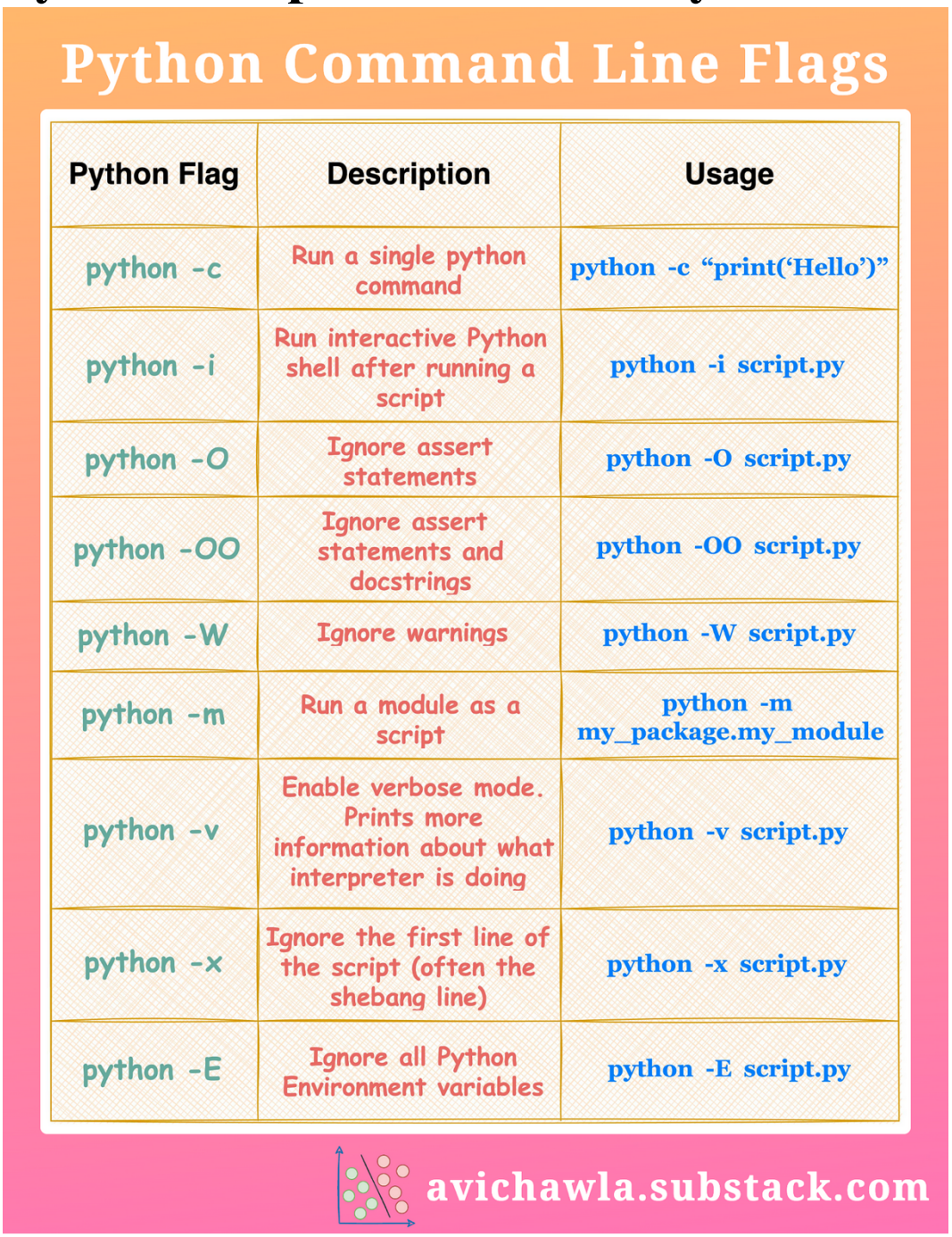
Overall, by accessing the column first, we avoid accessing non-contiguous memory access, which does happen when we access the row first.

This makes selecting the column first faster than slicing a row first in indexing operations.

If you are confused about what selecting, indexing, slicing, and filtering mean, here's what you should read next:

https://avichawla.substack.com/p/are-you-sure-you-are-using-the-correct.

9 Command Line Flags To Run Python Scripts More Flexibly



Python Flag	Description	Usage
python -c	Run a single python command	python -c "print('Hello')"
python -i	Run interactive Python shell after running a script	python -i script.py
python -O	Ignore assert statements	python -O script.py
python -OO	Ignore assert statements and docstrings	python -OO script.py
python -W	Ignore warnings	python -W script.py
python -m	Run a module as a script	python -m my_package.my_module
python -v	Enable verbose mode. Prints more information about what interpreter is doing	python -v script.py
python -x	Ignore the first line of the script (often the shebang line)	python -x script.py
python -E	Ignore all Python Environment variables	python -E script.py

When invoking a Python script, you can specify various options/flags. They are used to modify the behavior of the Python interpreter when it runs a script or module.

Here are 9 of the most commonly used options:

🔷 **python -c**: Run a single Python command. Useful for running simple one-liners or testing code snippets.

🔷 **python -i**: Run the script as usual and enter the interactive mode instead of terminating the program. Useful for debugging as you can interact with objects created during the program.

🔷 **python -O**: Ignore assert statements (This is alphabet 'O'). Useful for optimizing code by removing debugging code.

🔷 **python -OO**: Ignore assert statements and discard docstrings. Useful for further optimizing code by removing documentation strings.

🔷 **python -W**: Ignore all warnings. Useful for turning off warnings temporarily and focusing on development.

🔷 **python -m**: Run a module as a script.

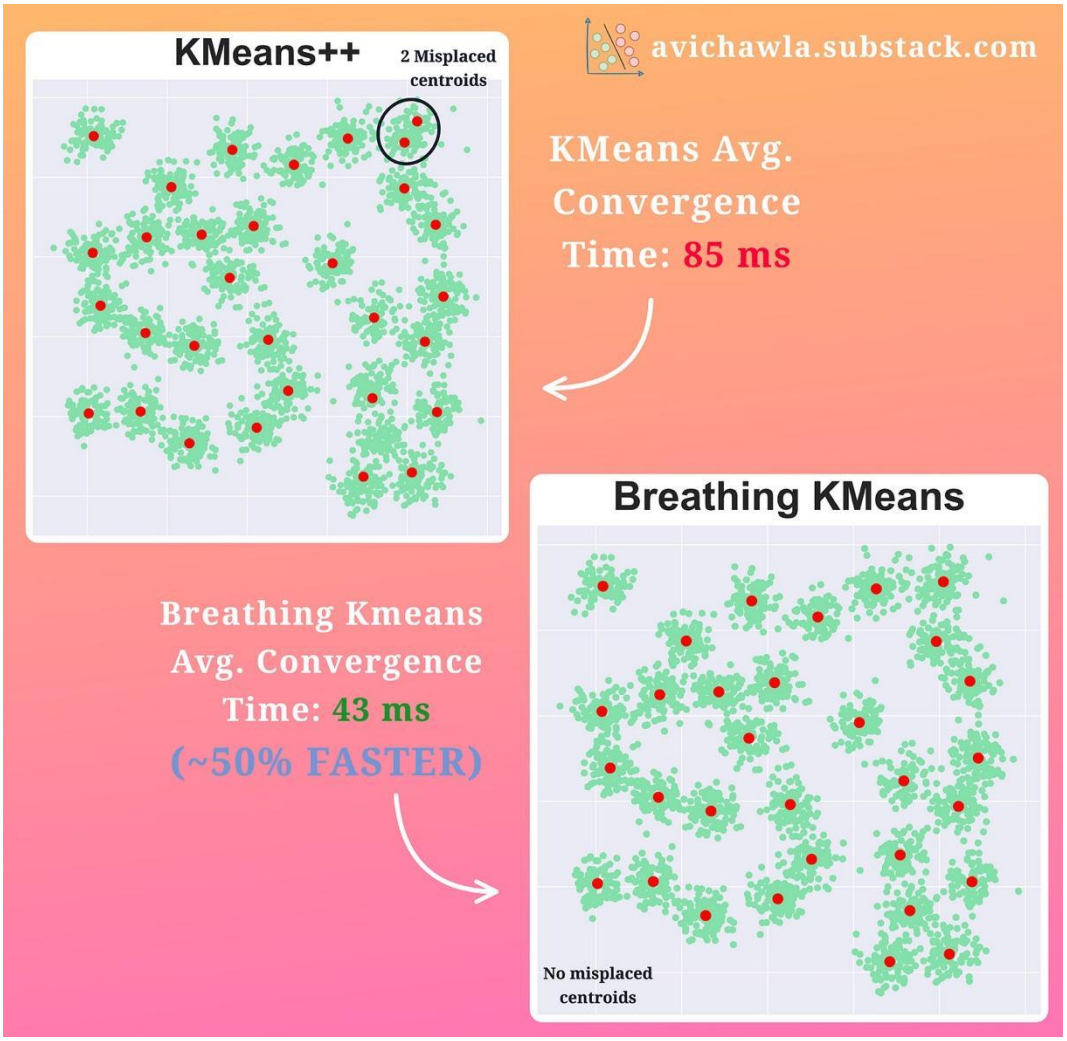
🔷 **python -v**: Enter verbose mode. Useful for printing extra information during program execution.

🔷 **python -x**: Skip the first line. Useful for removing shebang lines or other comments at the start of a script.

🔷 **python -E**: ignore all Python environment variables. Useful for ensuring a consistent program behavior by ignoring environment variables that may affect program execution.

Which ones have I missed? Let me know :)

Breathing KMeans: A Better and Faster Alternative to KMeans



The performance of KMeans is entirely dependent on the centroid initialization step. Thus, obtaining inaccurate clusters is highly likely.

While KMeans++ offers smarter centroid initialization, it does not always guarantee accurate convergence (read how KMeans++ works in my previous post). This is especially true when the number of clusters is high. Here, repeating the algorithm may help. But it introduces an unnecessary overhead in run-time.

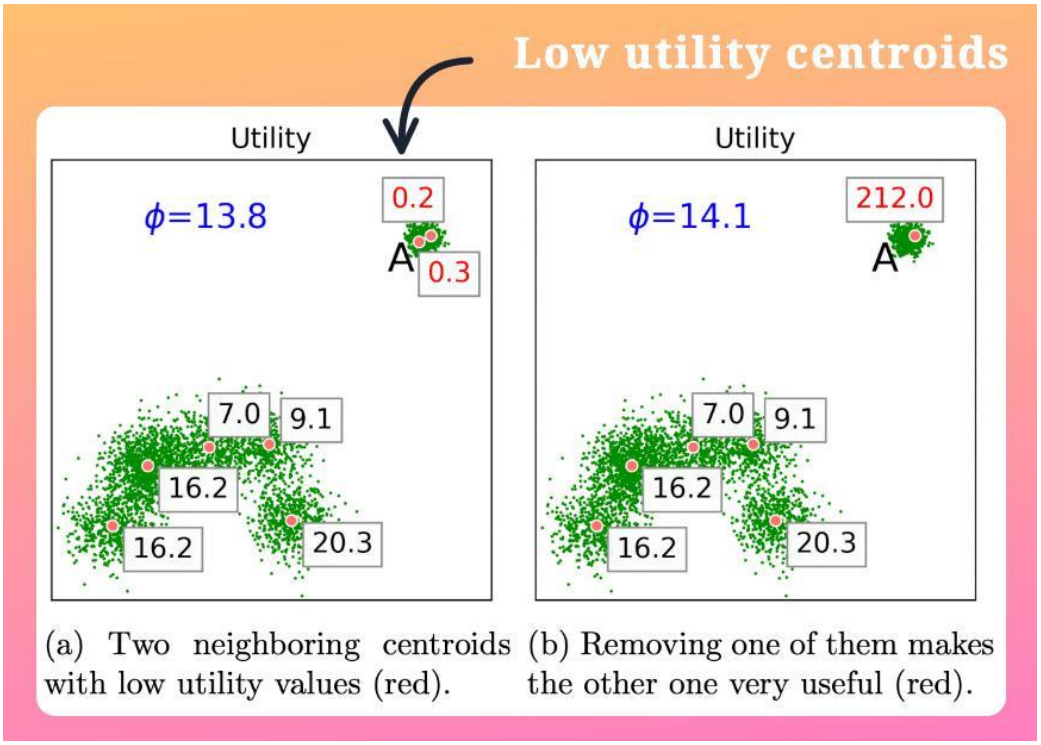
Instead, Breathing KMeans is a better alternative here. Here's how it works:

- **Step 1:** Initialise `k` centroids and run KMeans without repeating. In other words, don't re-run it with different initializations. Just run it once.
- **Step 2 — Breathe in step:** Add `m` new centroids and run KMeans with (`k+m`) centroids without repeating.

- **Step 3 — Breathe out step:** Remove `m` centroids from existing (`k+m`) centroids. Run KMeans with the remaining `k` centroids without repeating.
- **Step 4:** Decrease `m` by `1`.
- **Step 5:** Repeat Steps 2 to 4 until `m=0`.

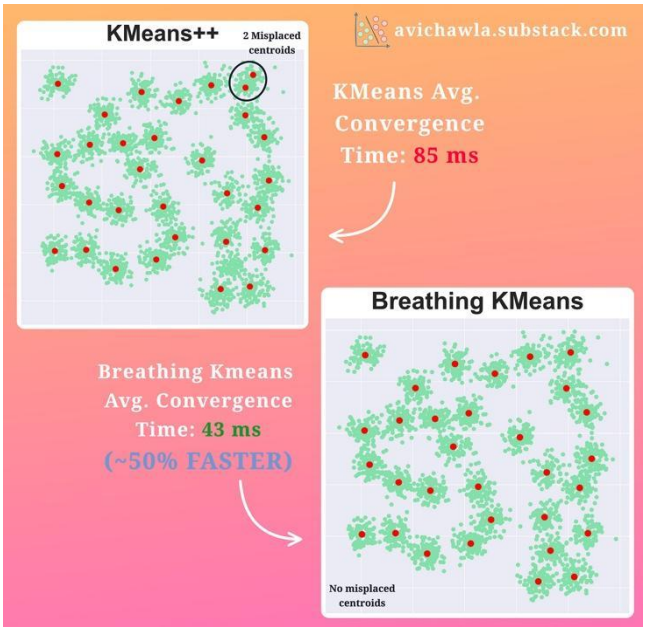
Breathe in step inserts new centroids close to the centroids with the largest errors. A centroid's error is the sum of the squared distance of points under that centroid.

Breathe out step removes centroids with low utility. A centroid's utility is proportional to its distance from other centroids. The intuition is that if two centroids are pretty close, they are likely falling in the same cluster. Thus, both will be assigned a low utility value, as demonstrated below.



With these repeated breathing cycles, Breathing KMeans provides a faster and better solution than KMeans. In each cycle, new centroids are added at "good" locations, and centroids with low utility are removed.

In the figure below, KMeans++ produced two misplaced centroids.



However, Breathing KMeans accurately clustered the data, with a `50%` improvement in run-time.

You can use Breathing KMeans by installing its open-source library, **bkmeans**, as follows:

```
pip install bkmeans
```

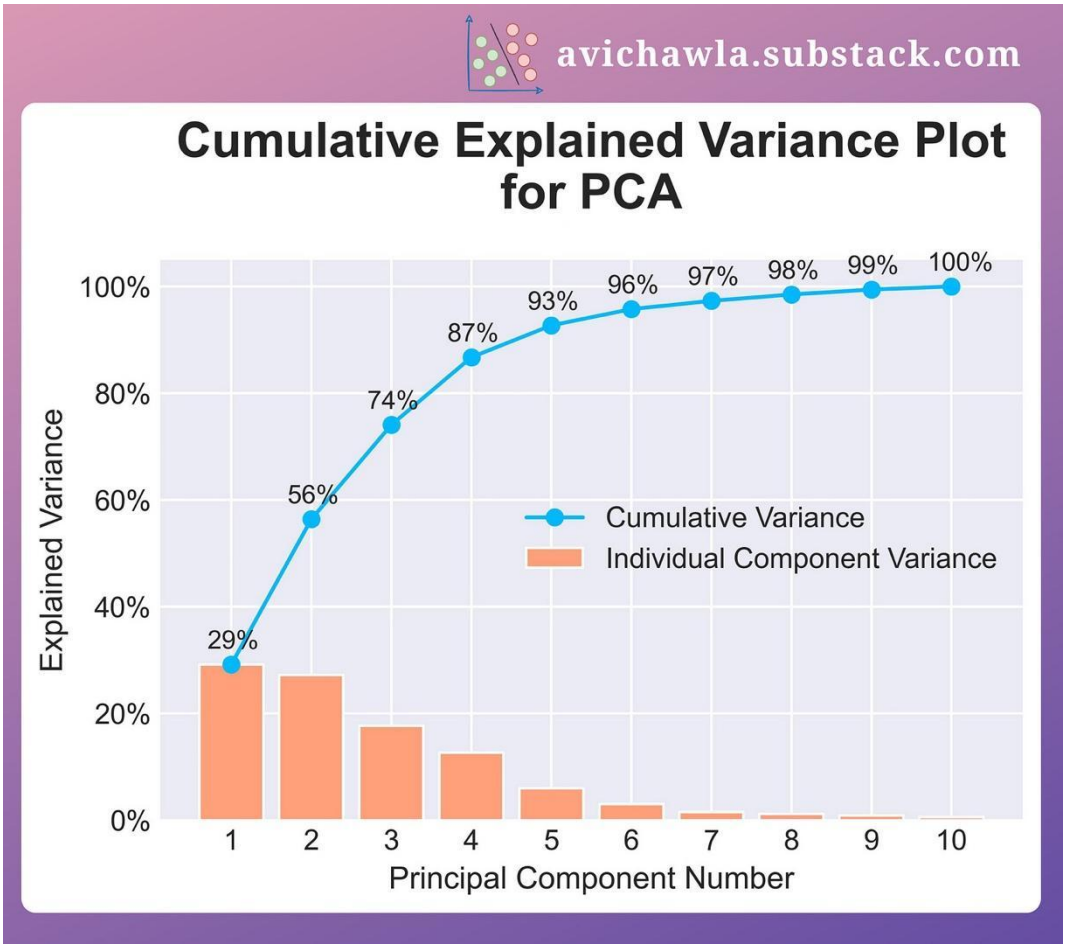
Next, import the library and run the clustering algorithm:

```
import numpy as np
from bkmeans import BKMeans

# generate random data set
X=np.random.rand(1000,2)

# create BKMeans instance
bkm = BKMeans(n_clusters=100)

# run the algorithm
bkm.fit(X)
```

In fact, the BKMeans class inherits from the KMeans class of sklearn. So you can specify other parameters and use any of the other methods on the `BKMeans` object as needed.

More details about Breathing KMeans: GitHub | Paper.

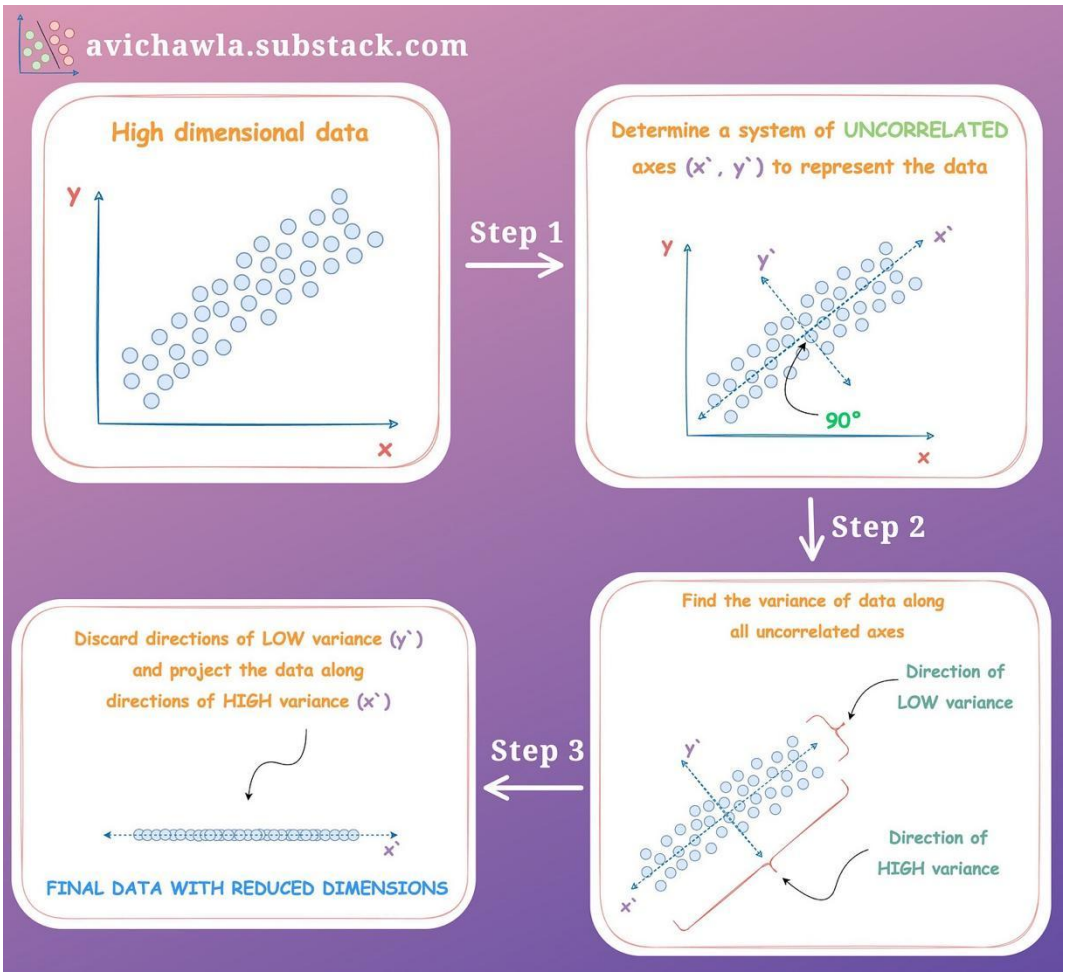
How Many Dimensions Should You Reduce Your Data To When Using PCA?



When using PCA, it can be difficult to determine the number of components to keep. Yet, here's a plot that can immensely help.

Note: If you don't know how PCA works, feel free to read my detailed post: A Visual Guide to PCA.

Still, here's a quick step-by-step refresher. Feel free to skip this part if you remember my PCA post.



Step 1. Take a high-dimensional dataset (**(x, y)** in the above figure) and represent it with uncorrelated axes (**(x`, y`)** in the above figure). Why uncorrelated?

This is to ensure that data has zero correlation along its dimensions and each new dimension represents its individual variance.

For instance, as data represented along **(x, y)** is correlated, the variance along **x** is influenced by the spread of data along **y**.

Instead, if we represent data along **(x`, y`)**, the variance along **x`** is not influenced by the spread of data along **y`**.

The above space is determined using eigenvectors.

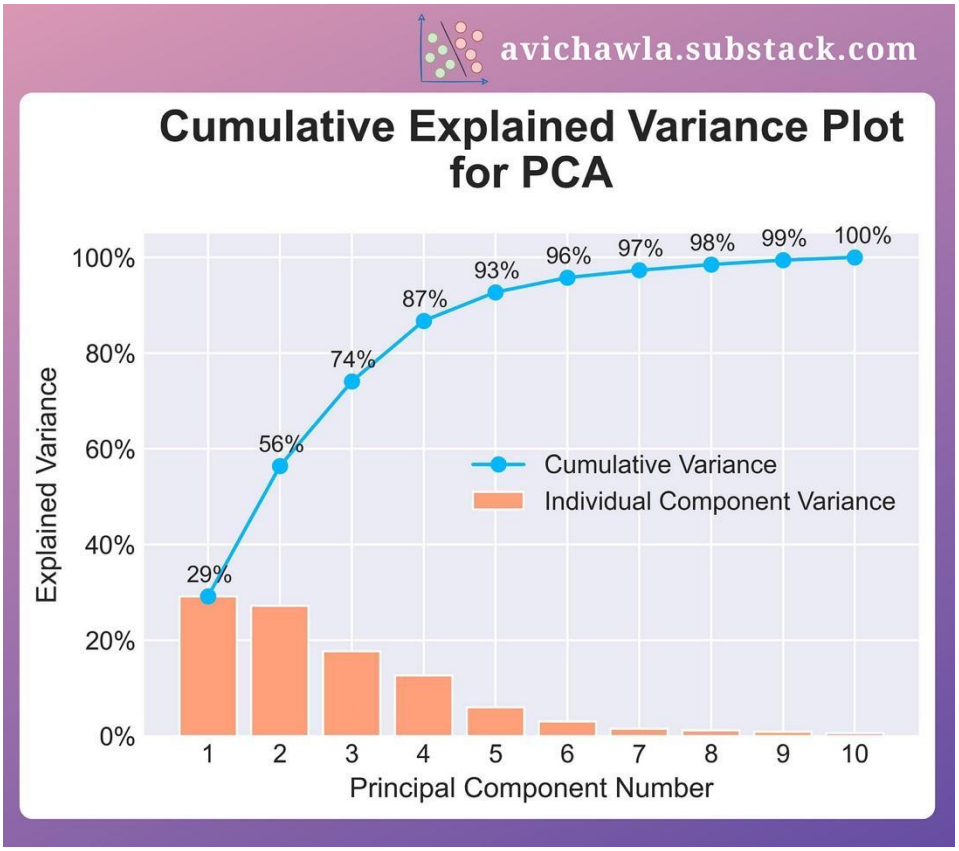
Step 2. Find the variance along all uncorrelated axes **(x`, y`)**. The eigenvalue corresponding to each eigenvector denotes the variance.

Step 3. Discard the axes with low variance. How many dimensions to discard (or keep) is a hyperparameter, which we will discuss below. Project the data along the retained axes.

When reducing dimensions, the purpose is to retain enough variance of the original data.

As each principal component explains some amount of variance, cumulatively plotting the component-wise variance can help identify which components have the most variance.

This is called a cumulative explained variance plot.

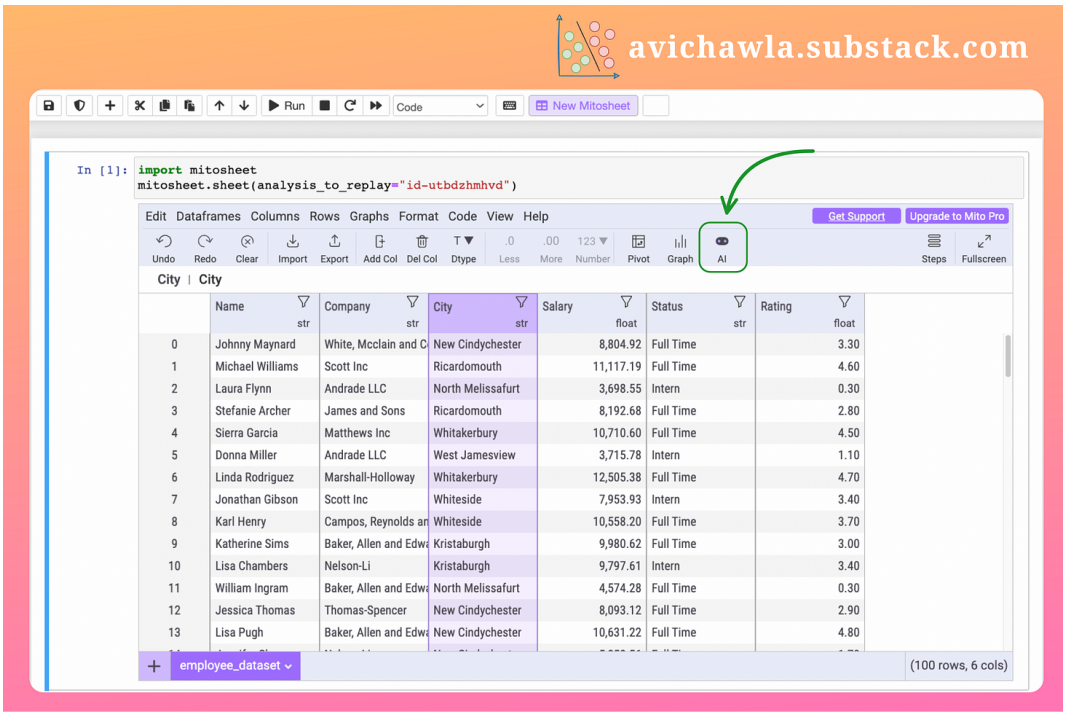


For instance, say we intend to retain **~85%** of the data variance. The above plot clearly depicts that reducing the data to four components will do that.

Also, as expected, all ten components together represent 100% variance of the data.

Creating this plot is pretty simple in Python. **Find the code here: PCA-CEV Plot.**

🚀 Mito Just Got Supercharged With AI!



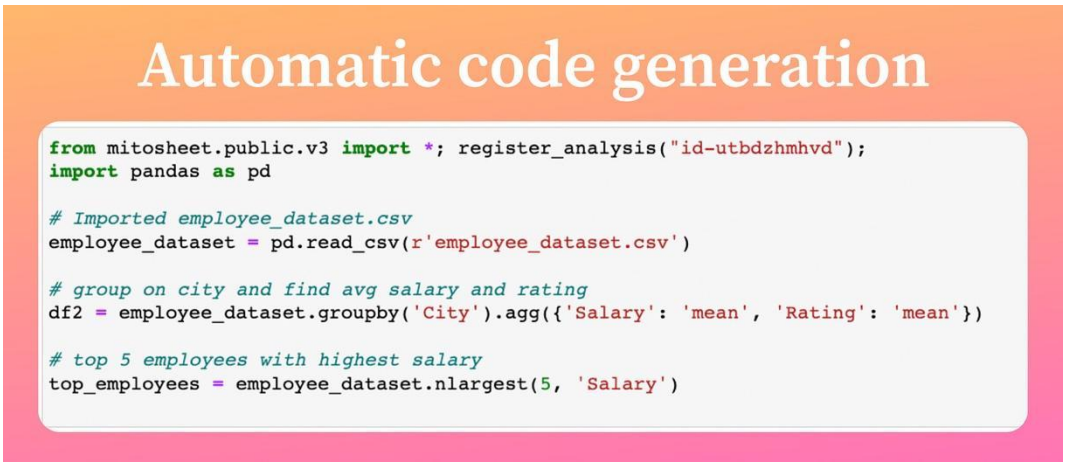
Personally, I am a big fan of no-code data analysis tools. They are extremely useful in eliminating repetitive code across projects—thereby boosting productivity.

Yet, most no-code tools are often limited in terms of the functionality they support. Thus, flexibility is usually a big challenge while using them.

Mito is an incredible open-source tool that allows you to analyze your data within a spreadsheet interface in Jupyter without writing any code.

What's more, Mito recently supercharged its spreadsheet interface with AI. As a result, you can now analyze data in a notebook with text prompts.

One of the coolest things about using Mito is that each edit in the spreadsheet automatically generates an equivalent Python code. This makes it convenient to reproduce the analysis later.



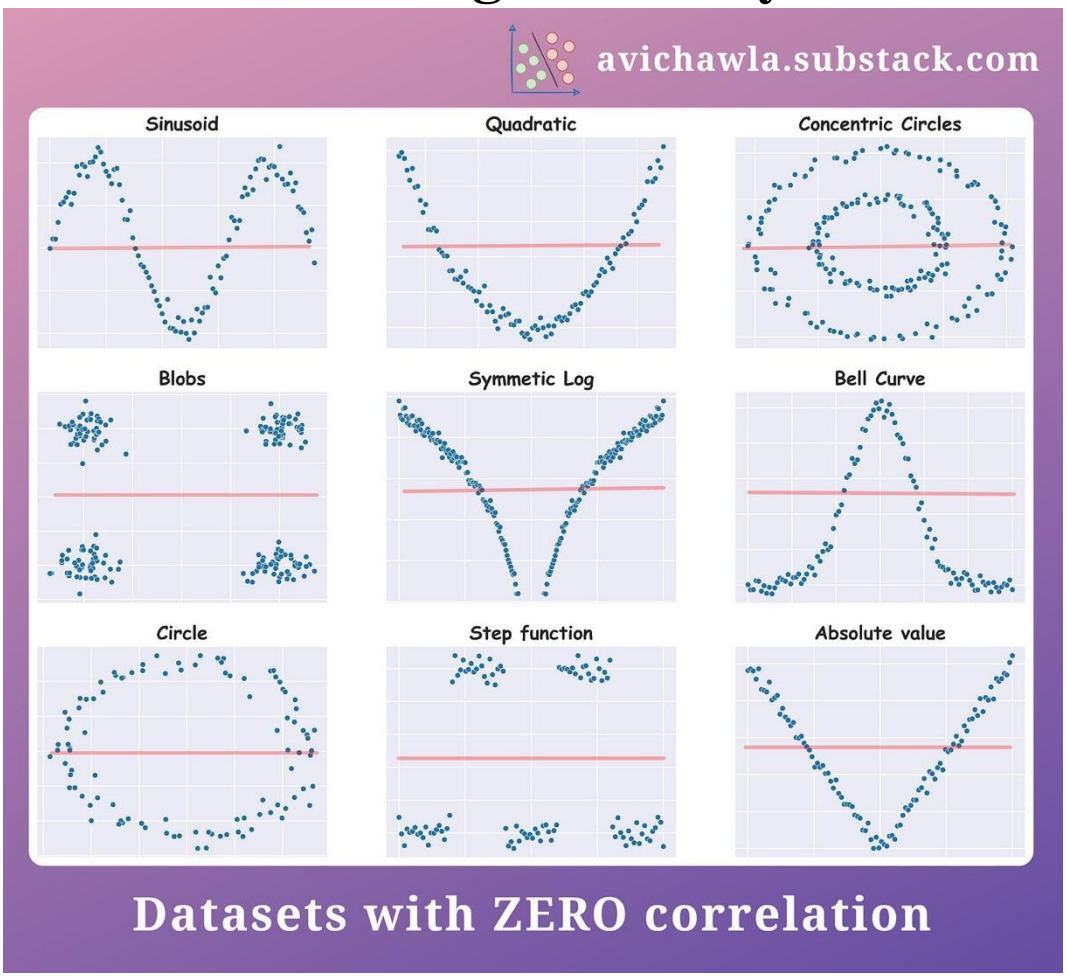
You can install Mito using pip as follows:

```
python -m pip install mitosheet
```

Next, to activate it in Jupyter, run the following two commands:

```
python -m jupyter nbextension install --py --user mitosheet
python -m jupyter nbextension enable --py --user mitosheet
```

Be Cautious Before Drawing Any Conclusions Using Summary Statistics



While analyzing data, one may be tempted to draw conclusions solely based on its statistics. Yet, the actual data might be conveying a totally different story.

Here's a visual depicting nine datasets with approx. zero correlation between the two variables. But the summary statistic (Pearson correlation in this case) gives no clue about what's inside the data.

What's more, data statistics could be heavily driven by outliers or other artifacts. I covered this in a previous post here.

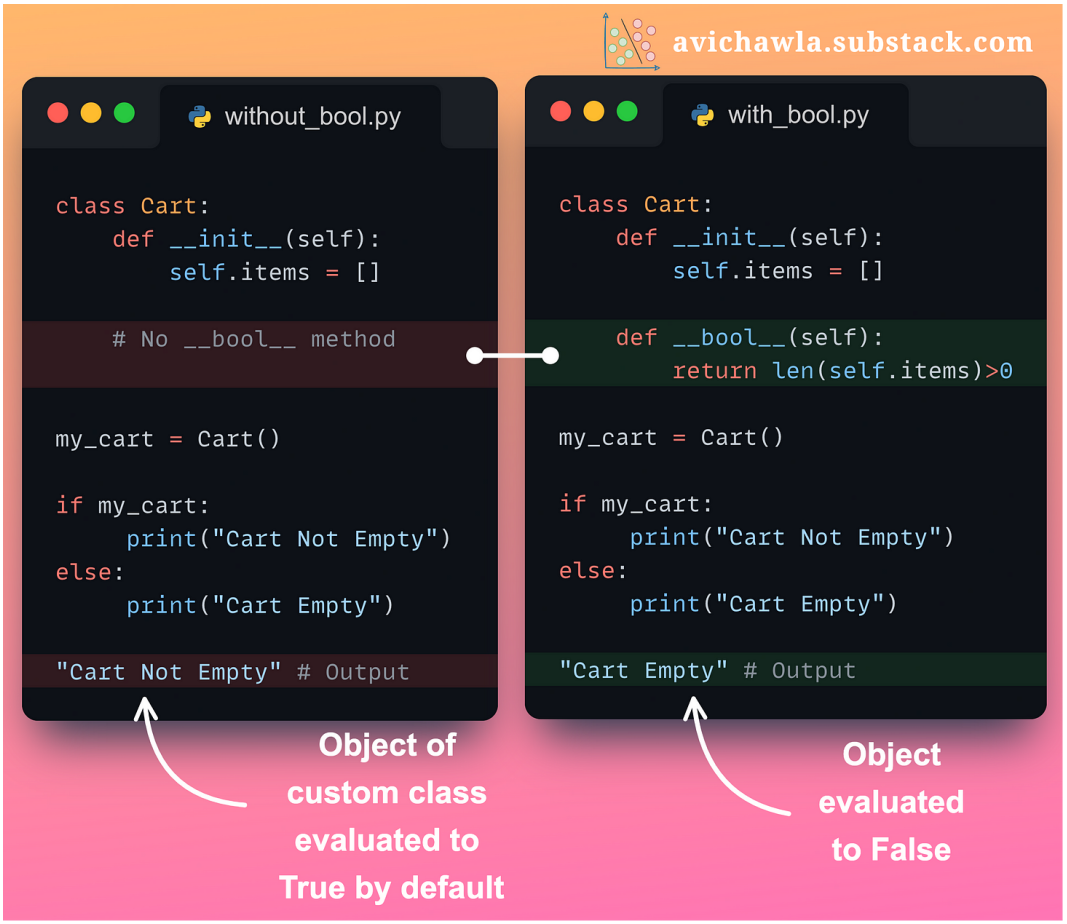
Thus, the importance of looking at the data cannot be stressed enough. It saves you from drawing wrong conclusions, which you could have made otherwise by looking at the statistics alone.

For instance, in the sinusoidal dataset above, Pearson correlation may make you believe that there is no association between the two variables. However, remember

that it is only quantifying the extent of a linear relationship between them. Read more about this in another one of my previous posts here.

Thus, if there's any other non-linear relationship (quadratic, sinusoid, exponential, etc.), it will fail to measure that.

Use Custom Python Objects In A Boolean Context



In a boolean context, Python always evaluates the objects of a custom class to True. But this may not be desired in all cases. Here's how you can override this behavior.

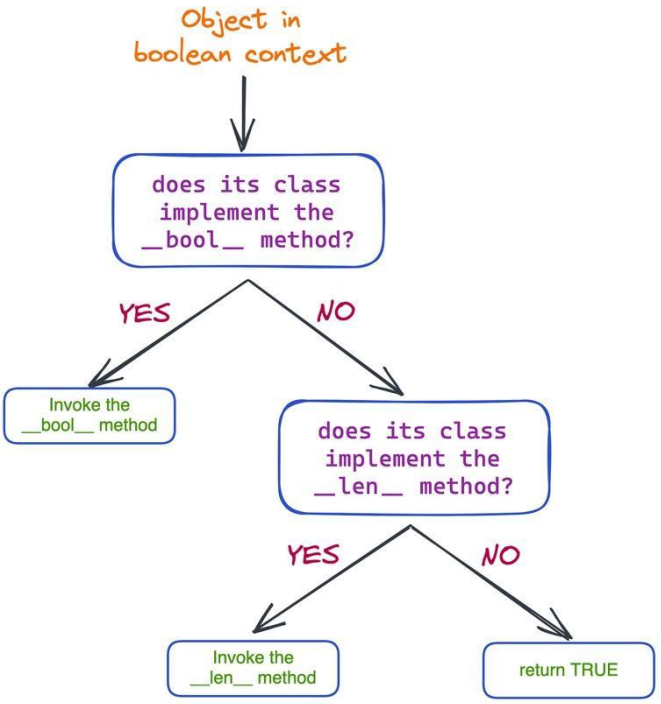
The **__bool__** dunder method is used to define the behavior of an object when used in a boolean context. As a result, you can specify explicit conditions to determine the truthiness of an object.

This allows you to use class objects in a more flexible and intuitive way.

As demonstrated above, without the **__bool__** method (*without_bool.py*), the object evaluates to True. But implementing the **__bool__** method lets us override this default behavior (*with_bool.py*).

Some additional good-to-know details

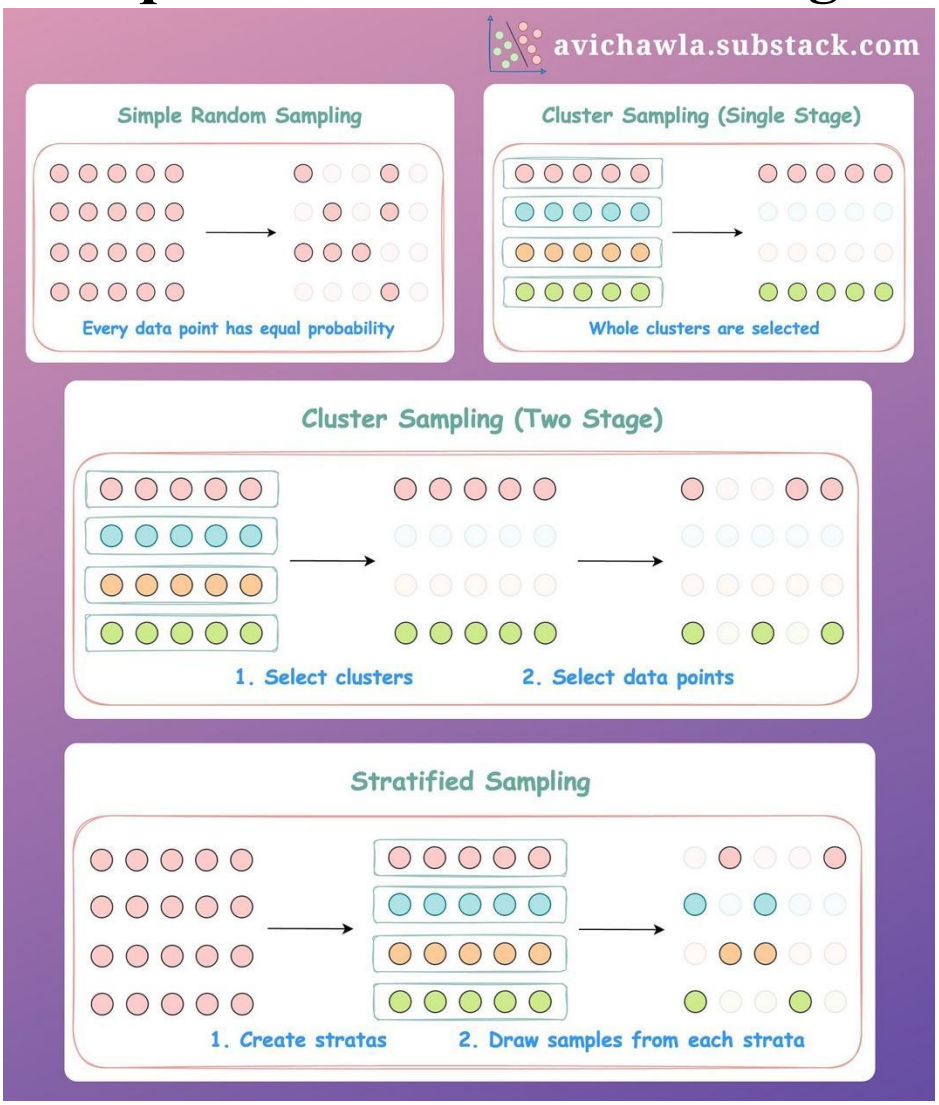
When we use ANY object (be it instantiated from a custom or an in-built class) in a boolean context, here's what Python does:



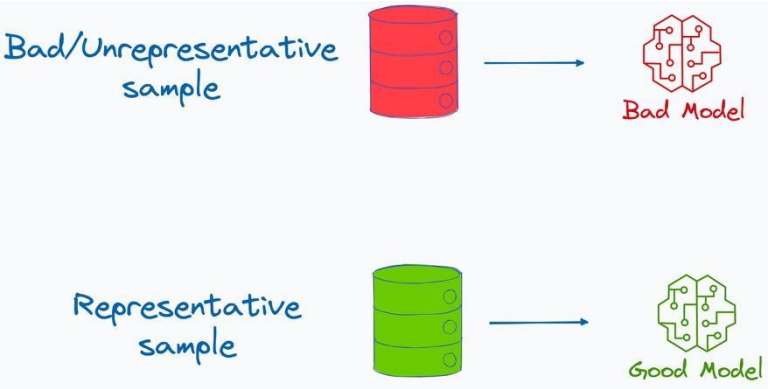
First, Python checks for the **__bool__** method in its class implementation. If found, it is invoked. If not, Python checks for the **__len__** method. If found, **__len__** is invoked. Otherwise, Python returns True.

This explains the default behavior of objects instantiated from a custom class. As the *Cart* class implemented neither the **__bool__** method nor the **__len__** method, the *cart* object was evaluated to True.

A Visual Guide To Sampling Techniques in Machine Learning



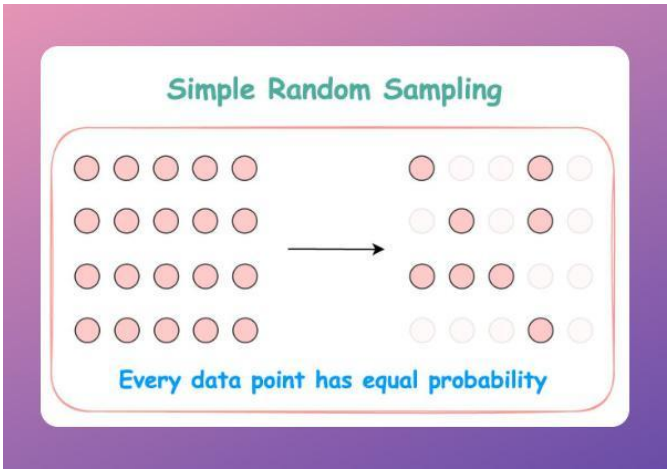
When you are dealing with large amounts of data, it is often preferred to draw a relatively smaller sample and train a model. But any mistakes can adversely affect the accuracy of your model.



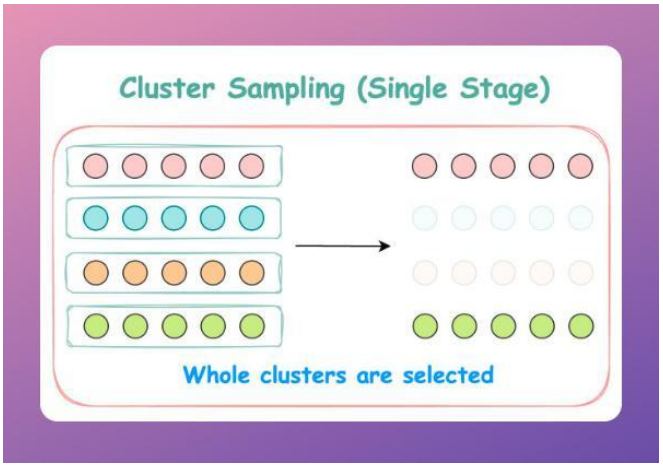
This makes sampling a critical aspect of training ML models.

Here are a few popularly used techniques that one should know about:

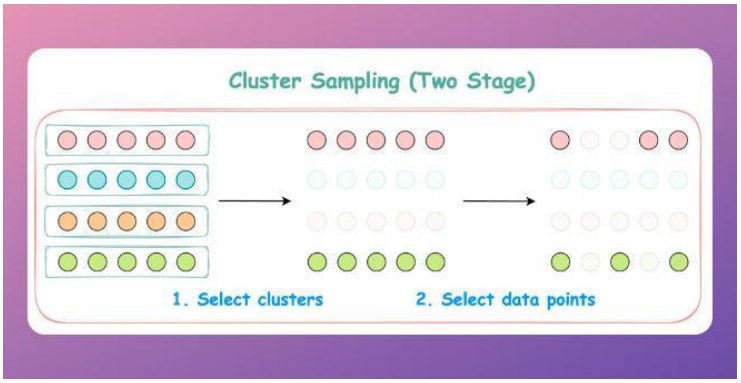
- **Simple random sampling:** Every data point has an equal probability of being selected in the sample.



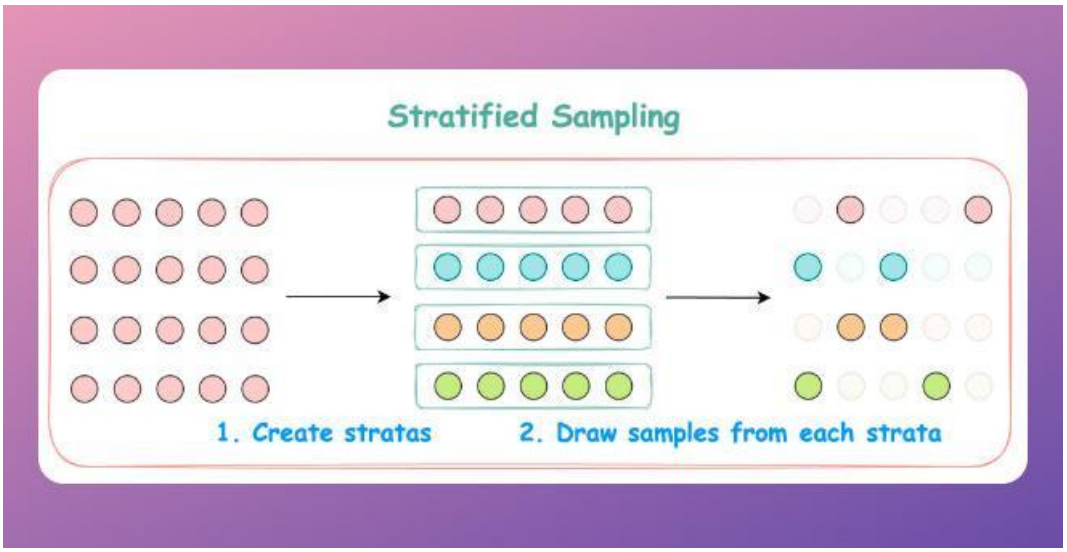
- **Cluster sampling (single-stage):** Divide the data into clusters and select a few entire clusters.



🔹 **Cluster sampling (two-stage):** Divide the data into clusters, select a few clusters, and choose points from them randomly.

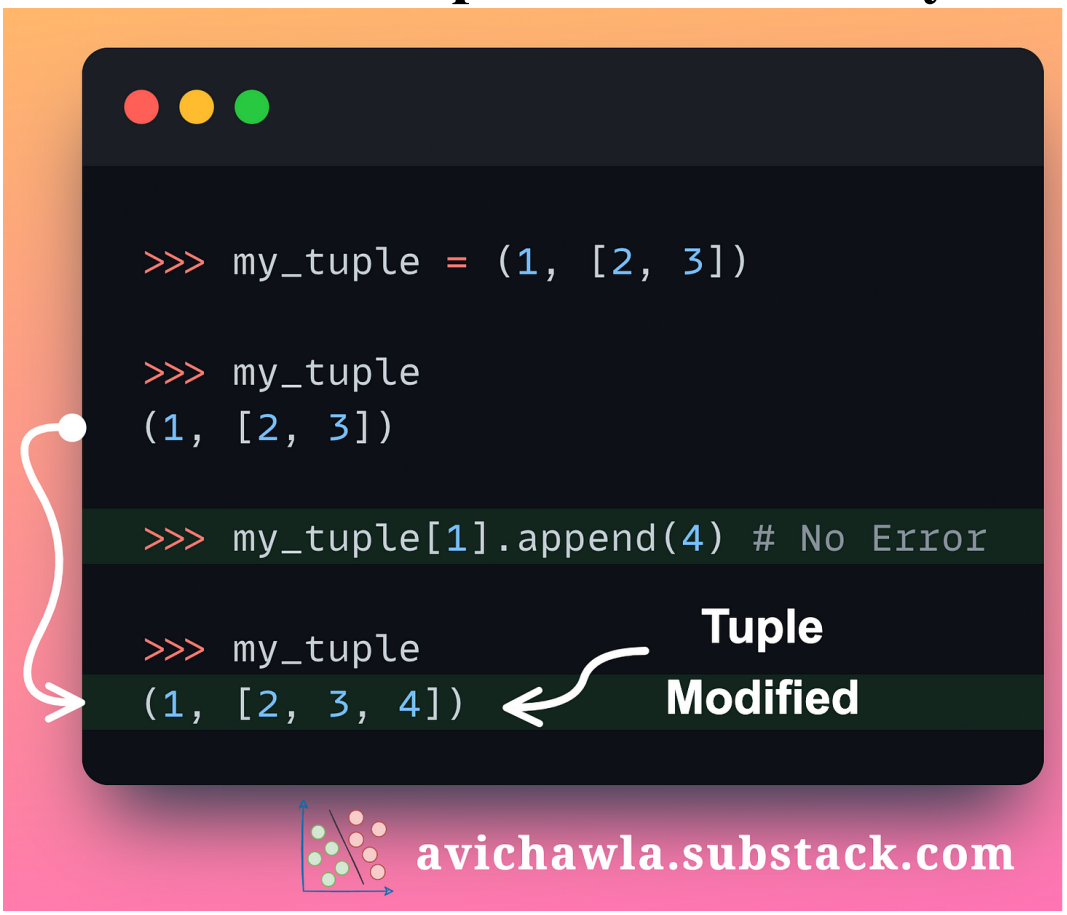


🔹 **Stratified sampling:** Divide the data points into homogenous groups (based on age, gender, etc.), and select points randomly.



What are some other sampling techniques that you commonly resort to?

You Were Probably Given Incomplete Info About A Tuple's Immutability



When we say tuples are immutable, many Python programmers think that the values inside a tuple cannot change. But this is not true.

The immutability of a tuple is solely restricted to the identity of objects it holds, not their value.

In other words, say a tuple has two objects with IDs **1** and **2**. Immutability says that the collection of IDs referenced by the tuple (and their order) can never change.

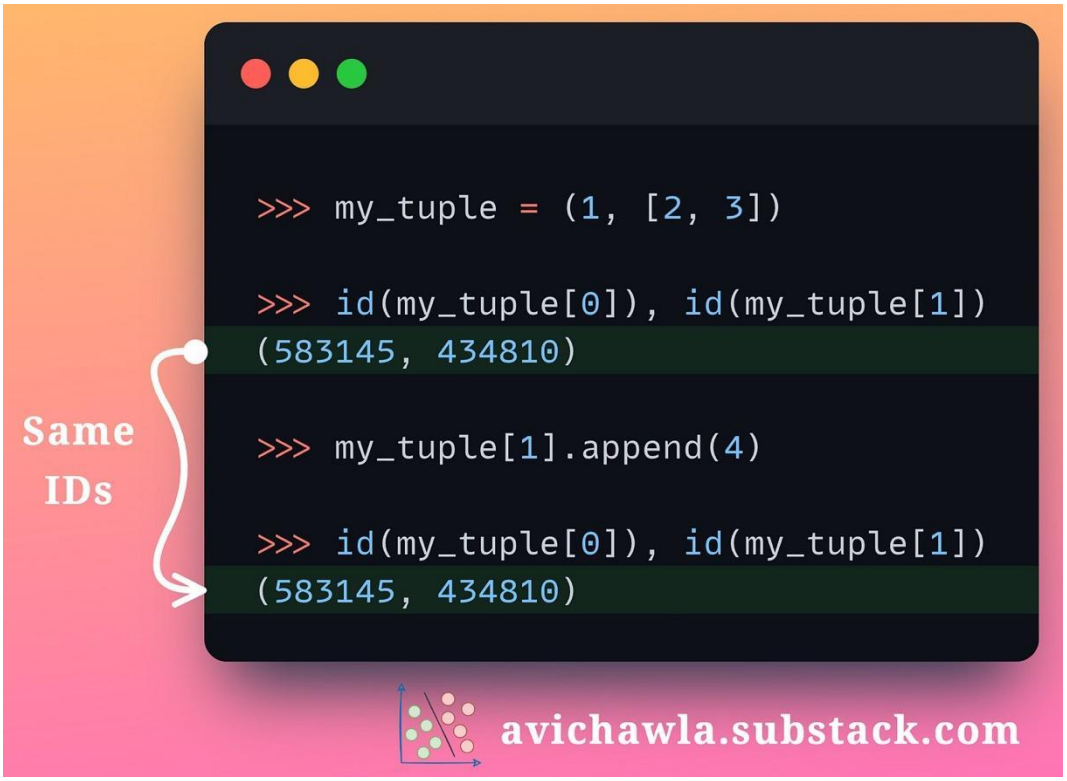
Yet, there is **NO** such restriction that the individual objects with IDs **1** and **2** cannot be modified.

Thus, if the elements inside the tuple are mutable objects, you can indeed modify them.

And as long as the collection of IDs remains the same, the immutability of a tuple is not violated.

This explains the demonstration above. As `append` is an inplace operation, the collection of IDs didn't change. Thus, Python didn't raise an error.

We can also verify this by printing the collection of object IDs referenced inside the tuple before and after the append operation:



As shown above, the IDs pre and post append are the same. Thus, immutability isn’t violated.

A Simple Trick That Significantly Improves The Quality of Matplotlib Plots

Matplotlib plots often appear dull and blurry, especially when scaled or zoomed. Yet, here's a simple trick to significantly improve their quality.

Matplotlib plots are rendered as an image by default. Thus, any scaling/zooming drastically distorts their quality.

Instead, always render your plot as a scalable vector graphic (SVG). As the name suggests, they can be scaled without compromising the plot's quality.

As demonstrated in the image above, the plot rendered as SVG clearly outshines and is noticeably sharper than the default plot.

The following code lets you change the render format to SVG. If the difference is not apparent in the image above, I would recommend trying it yourself and noticing the difference.

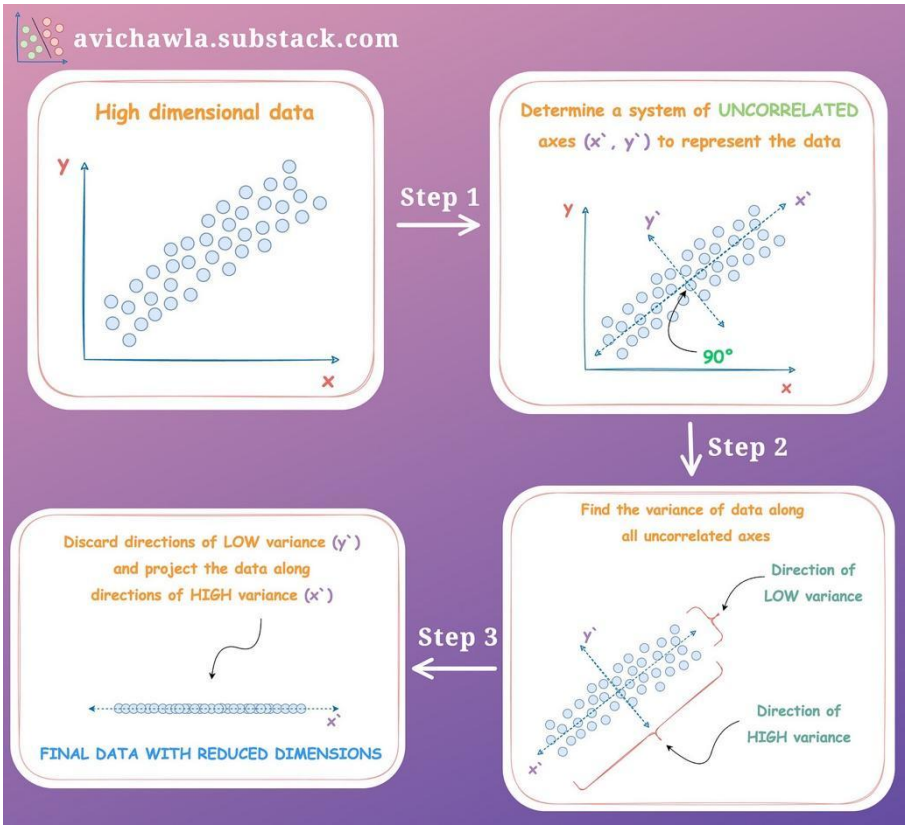
```
from matplotlib_inline.backend_inline import set_matplotlib_formats
set_matplotlib_formats('svg')
```

Alternatively, you can also use the following code:

```
%config InlineBackend.figure_format = 'svg'
```

P.S. If there's a chance that you don't know what is being depicted in the bar plot above, check out this YouTube video by Numberphile.

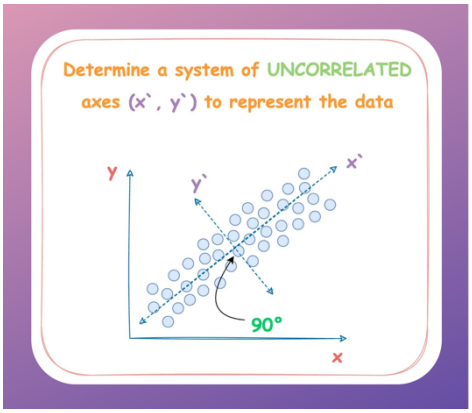
A Visual and Overly Simplified Guide to PCA



Many folks often struggle to understand the core essence of principal component analysis (PCA), which is widely used for dimensionality reduction. Here's a simplified visual guide depicting what goes under the hood.

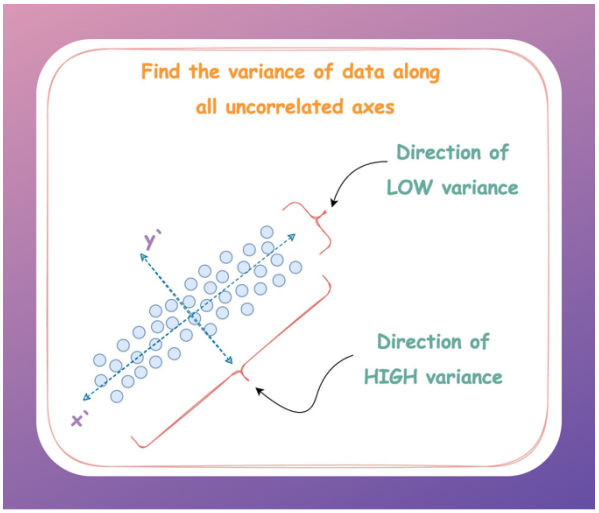
In a gist, while reducing the dimensions, the aim is to retain as much variation in data as possible.

To begin with, as the data may have correlated features, the first step is to determine a new coordinate system with orthogonal axes. This is a space where all dimensions are uncorrelated.



The above space is determined using the data's eigenvectors.

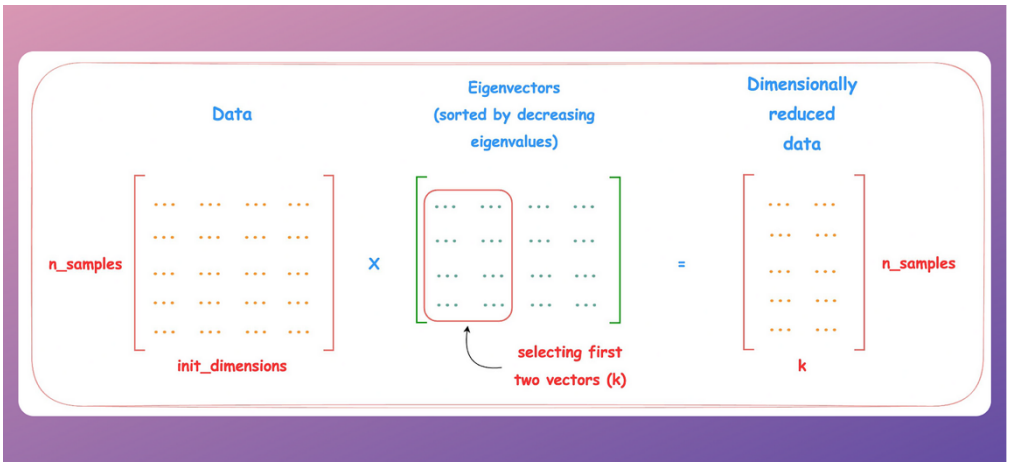
Next, we find the variance of our data along these uncorrelated axes. The variance is represented by the corresponding eigenvalues.



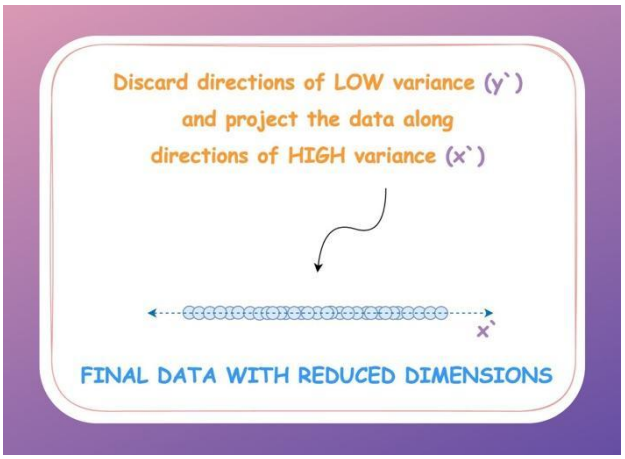
Next, we decide the number of dimensions we want our data to have post-reduction (a hyperparameter), say two. As our aim is to retain as much variance as possible, we select two eigenvectors with the highest eigenvalues.

Why highest, you may ask? As mentioned above, the variance along an eigenvector is represented by its eigenvalue. Thus, selecting the top two eigenvalues ensures we retain the maximum variance of the overall data.

Lastly, the data is transformed using a simple matrix multiplication with the top two vectors, as shown below:



After reducing the dimension of the 2D dataset used above, we get the following.



This is how PCA works. I hope this algorithm will never feel daunting again :)

Supercharge Your Jupyter Kernel With ipyflow

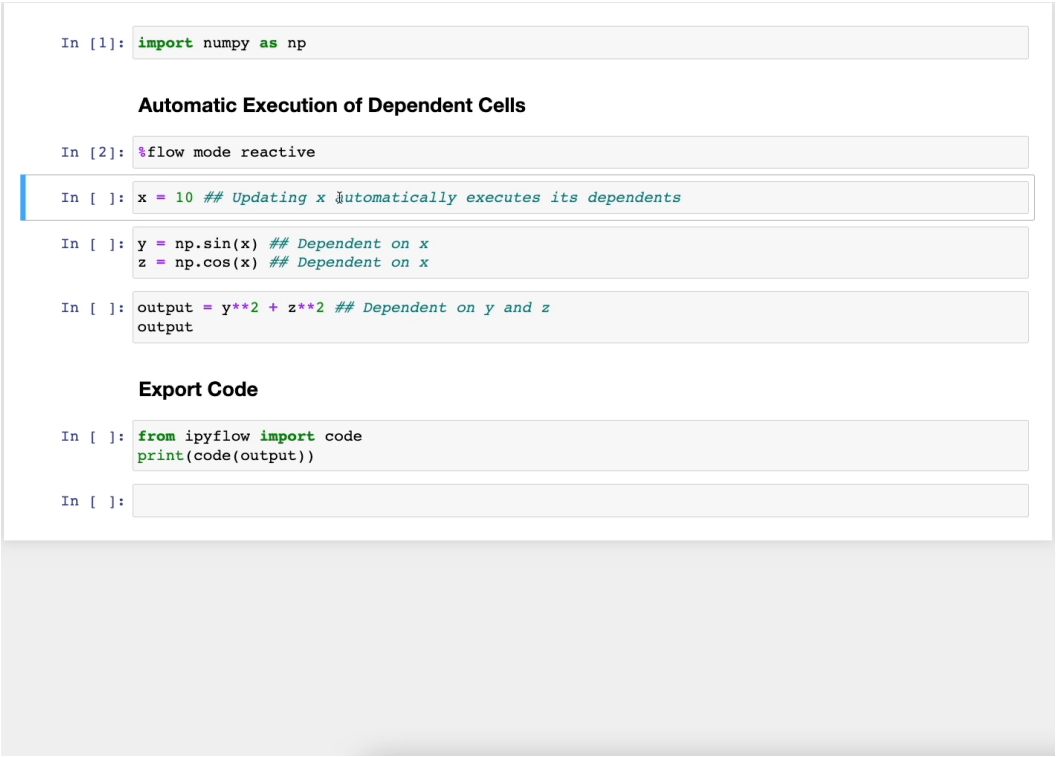
This is a pretty cool Jupyter hack I learned recently.

While using Jupyter, you must have noticed that when you update a variable, all its dependent cells have to be manually re-executed.

Also, at times, isn't it difficult to determine the exact sequence of cell executions that generated an output?

This is tedious and can get time-consuming if the sequence of dependent cells is long.

To resolve this, try **ipyflow**. It is a supercharged kernel for Jupyter, which tracks the relationship between cells and variables.

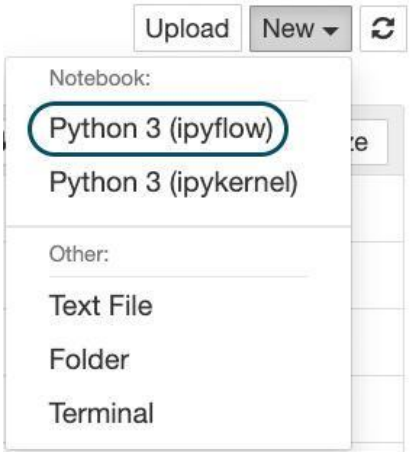


Thus, at any point, you can obtain the corresponding code to reconstruct any symbol.

What's more, its magic command enables an automatic recursive re-execution of dependent cells if a variable is updated.

As shown in the demo above, updating the variable `X` automatically triggers its dependent cells.

Do note that **ipyflow** offers a different kernel from the default kernel in Jupyter. Thus, once you install **ipyflow**, select the following kernel while launching a new notebook:

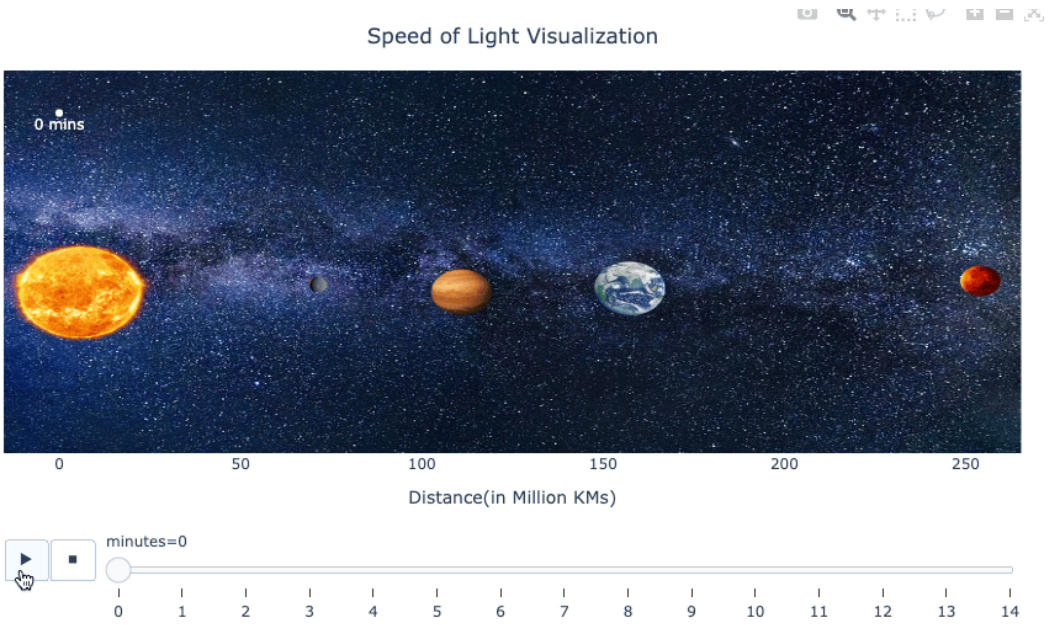


Find more details here: ipyflow.

A Lesser-known Feature of Creating Plots with Plotly

Plotly is pretty diverse when it comes to creating different types of charts. While many folks prefer it for interactivity, you can also use it to create animated plots.

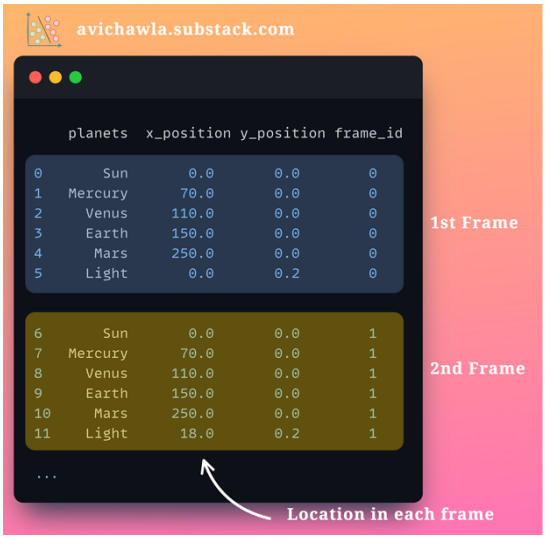
Here's an animated visualization depicting the time taken by light to reach different planets after leaving the Sun.



Several functions in Plotly support animations using the **animation_frame** and **animation_group** parameters.

The core idea behind creating an animated plot relies on plotting the data one frame at a time.

For instance, consider we have organized the data frame-by-frame, as shown below:

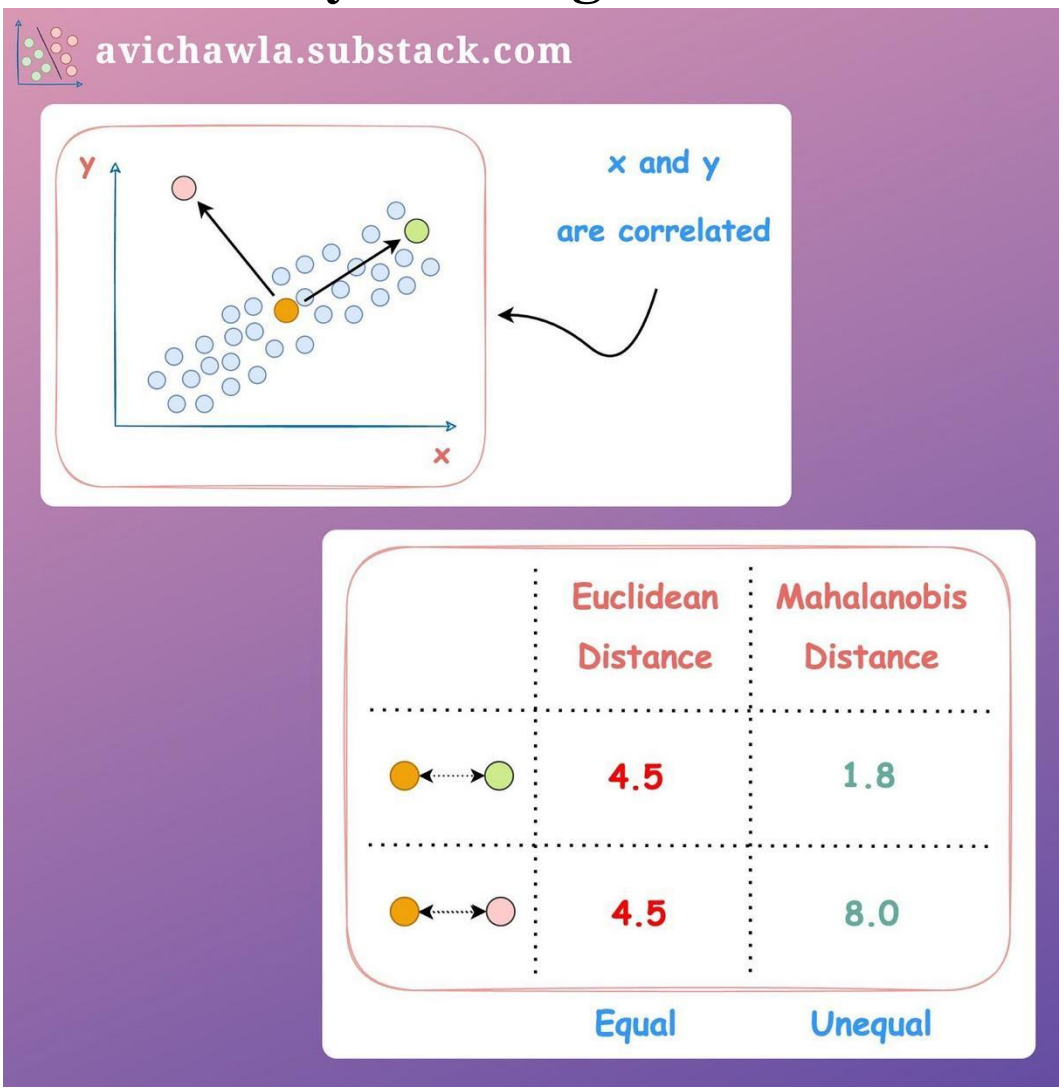


Now, if we invoke the scatter method with the **animation_frame** argument, it will plot the data frame-by-frame, giving rise to an animation.

In the above function call, the data corresponding to **frame_id=0** will be plotted first. This will be replaced by the data with **frame_id=1** in the next frame, and so on.

Find the code for this post here: GitHub.

The Limitation Of Euclidean Distance Which Many Often Ignore



Euclidean distance is a commonly used distance metric. Yet, its limitations often make it inapplicable in many data situations.

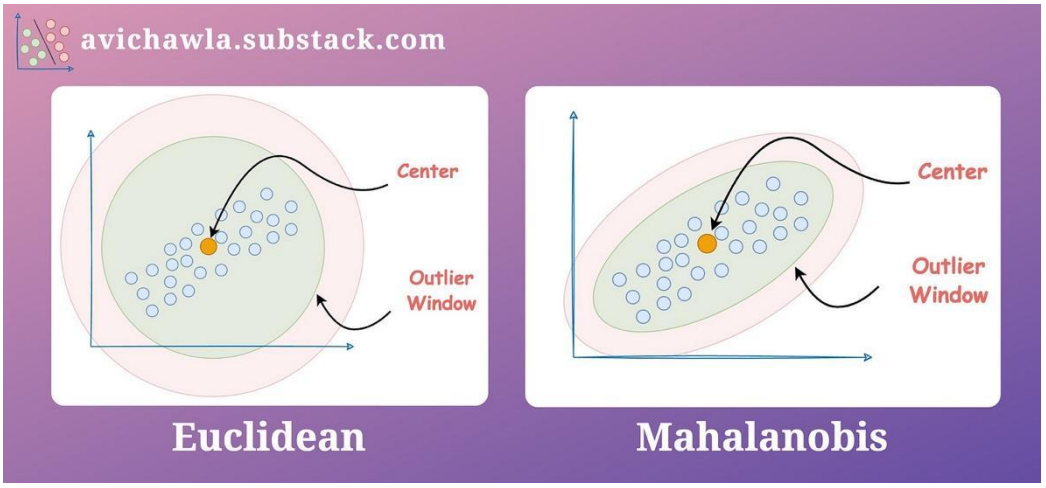
Euclidean distance assumes independent axes, and the data is somewhat spherically distributed. But when the dimensions are correlated, euclidean may produce misleading results.

Mahalanobis distance is an excellent alternative in such cases. It is a multivariate distance metric that takes into account the data distribution.

As a result, it can measure how far away a data point is from the distribution, which Euclidean cannot.

As shown in the image above, Euclidean considers pink and green points equidistant from the central point. But Mahalanobis distance considers the green point to be closer, which is indeed true, taking into account the data distribution.

Mahalanobis distance is commonly used in outlier detection tasks. As shown below, while Euclidean forms a circular boundary for outliers, Mahalanobis, instead, considers the distribution—producing a more practical boundary.



Essentially, Mahalanobis distance allows the data to construct a coordinate system for itself, in which the axes are independent and orthogonal.

Computationally, it works as follows:

- **Step 1:** Transform the columns into uncorrelated variables.
- **Step 2:** Scale the new variables to make their variance equal to 1.
- **Step 3:** Find the Euclidean distance in this new coordinate system, where the data has a unit variance.

So eventually, we do reach Euclidean. However, to use Euclidean, we first transform the data to ensure it obeys the assumptions.

Mathematically, it is calculated as follows:

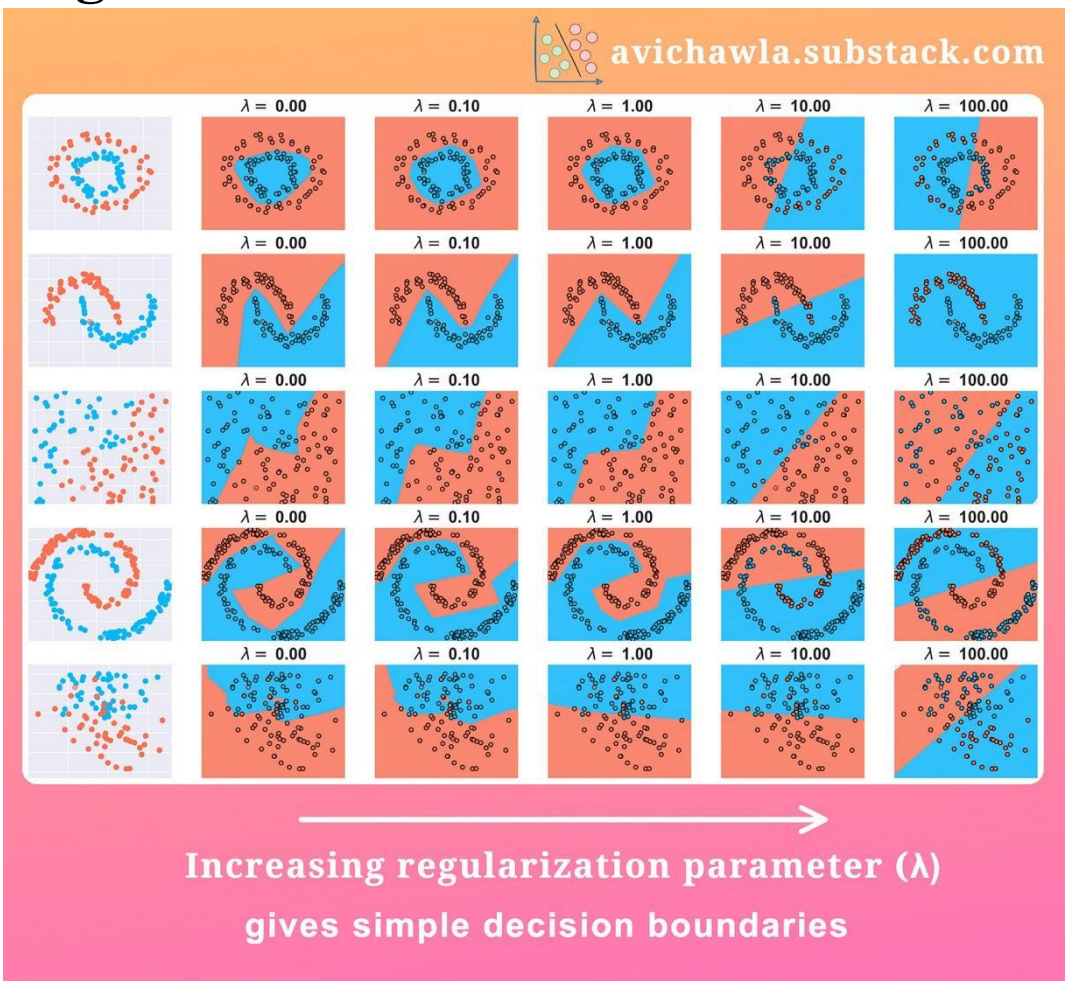
$$D^2 = (x - \mu)^T \cdot C^{-1} \cdot (x - \mu)$$

- x: rows of your dataset (Shape: n_samples*n_dimensions).
- μ: mean of individual dimensions (Shape: 1*n_dimensions).
- C^-1: Inverse of the covariance matrix (Shape: n_dimensions*n_dimensions).

- D^2: Square of the Mahalanobis distance (Shape: n_samples*n_samples).

Find more info here: Scipy docs.

Visualising The Impact Of Regularisation Parameter



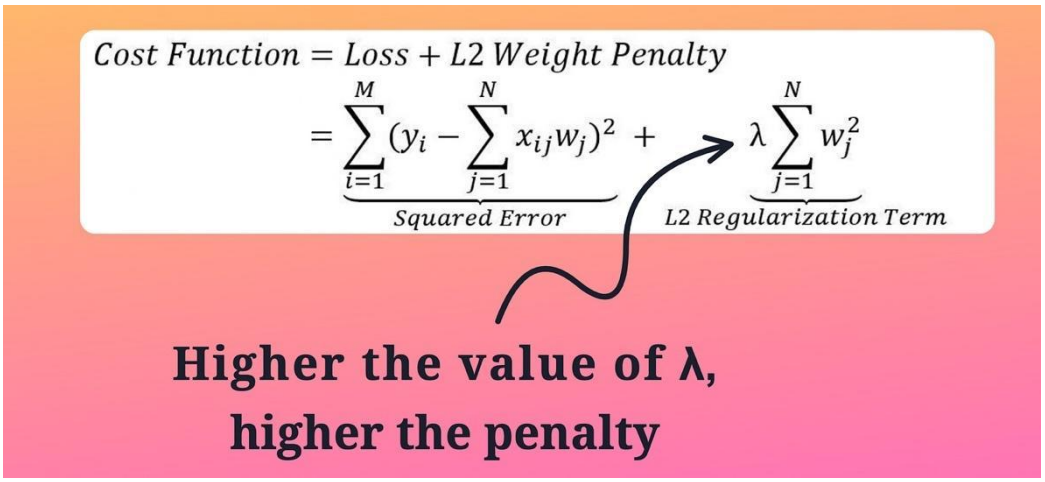
Regularization is commonly used to prevent overfitting. The above visual depicts the decision boundary obtained on various datasets by varying the regularization parameter.

As shown, increasing the parameter results in a decision boundary with fewer curvatures. Similarly, decreasing the parameter produces a more complicated decision boundary.

But have you ever wondered what goes on behind the scenes? Why does increasing the parameter force simpler decision boundaries?

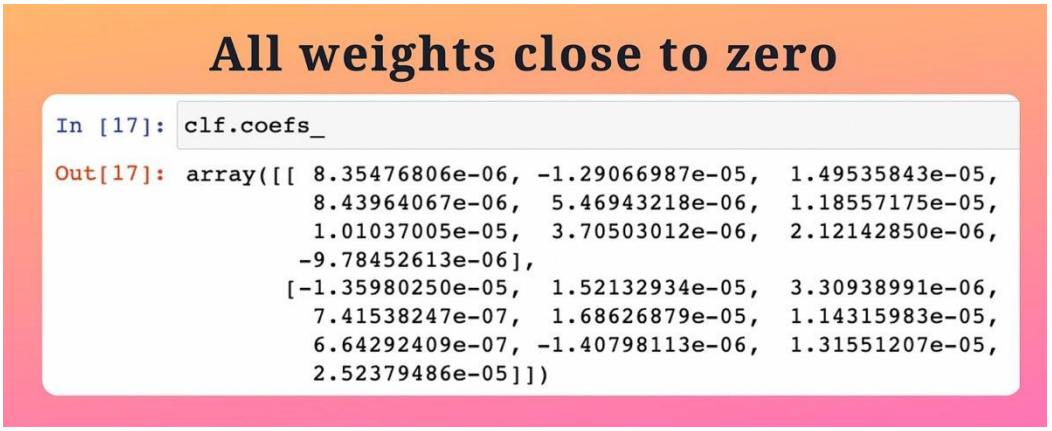
To understand that, consider the cost function equation below (this is for regression though, but the idea stays the same for classification).

It is clear that the cost increases linearly with the parameter **λ.**



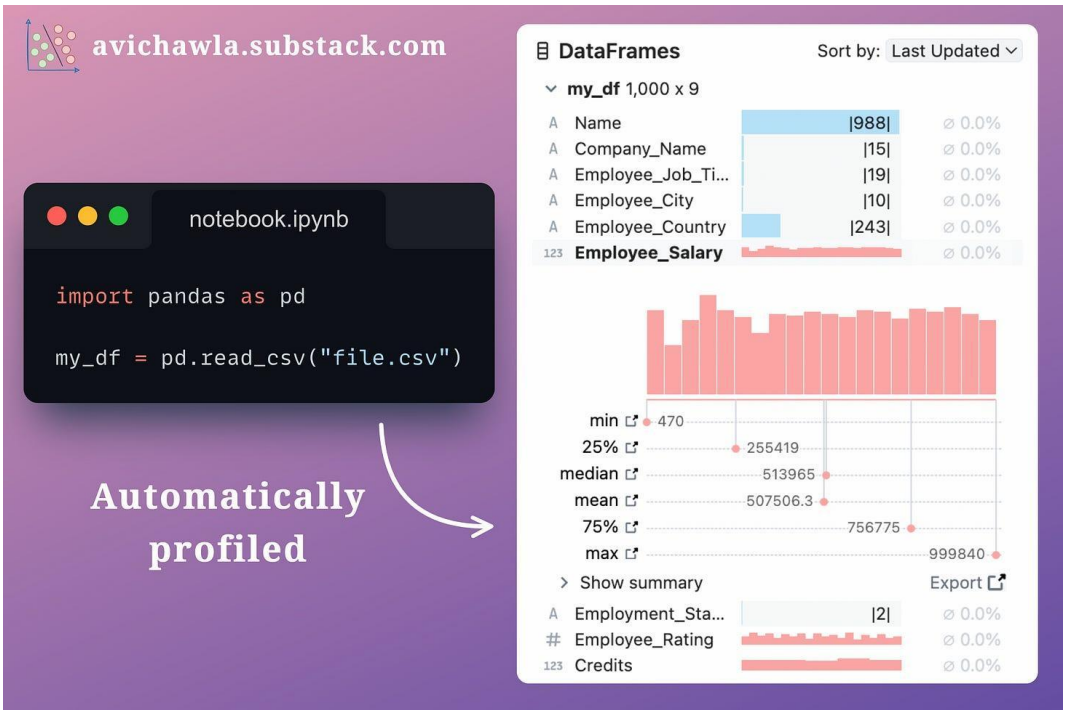
Now, if the parameter is too high, the penalty becomes higher too. Thus, to minimize its impact on the overall cost function, the network is forced to approach weights that are closer to zero.

This becomes evident if we print the final weights for one of the models, say one at the bottom right (last dataset, last model).



Having smaller weights effectively nullifies many neurons, producing a much simpler network. This prevents many complex transformations, that could have happened otherwise.

AutoProfiler: Automatically Profile Your DataFrame As You Work

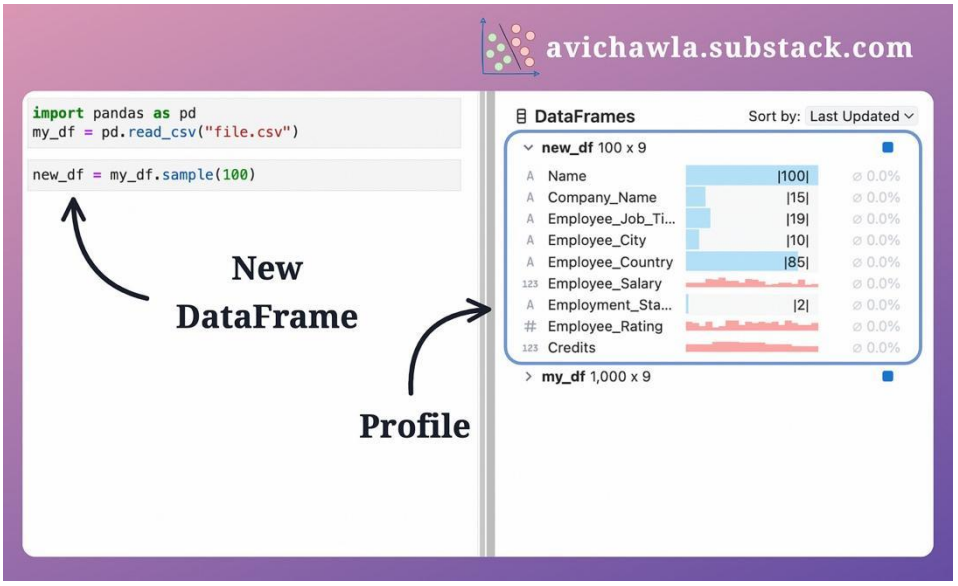


Pandas AutoProfiler: Automatically profile Pandas DataFrames at each execution, without any code.

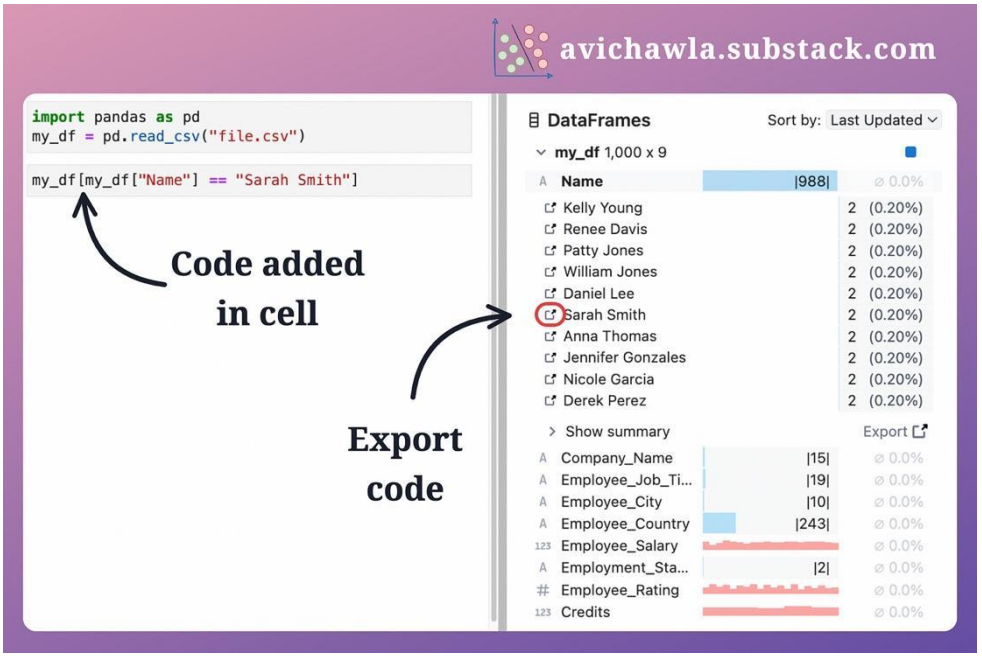
AutoProfiler is an open-source dataframe analysis tool in jupyter. It reads your notebook and automatically profiles every dataframe in your memory as you change them.

In other words, if you modify an existing dataframe, AutoProfiler will automatically update its corresponding profiling.

Also, if you create a new dataframe (say from an existing dataframe), AutoProfiler will automatically profile that as well, as shown below:

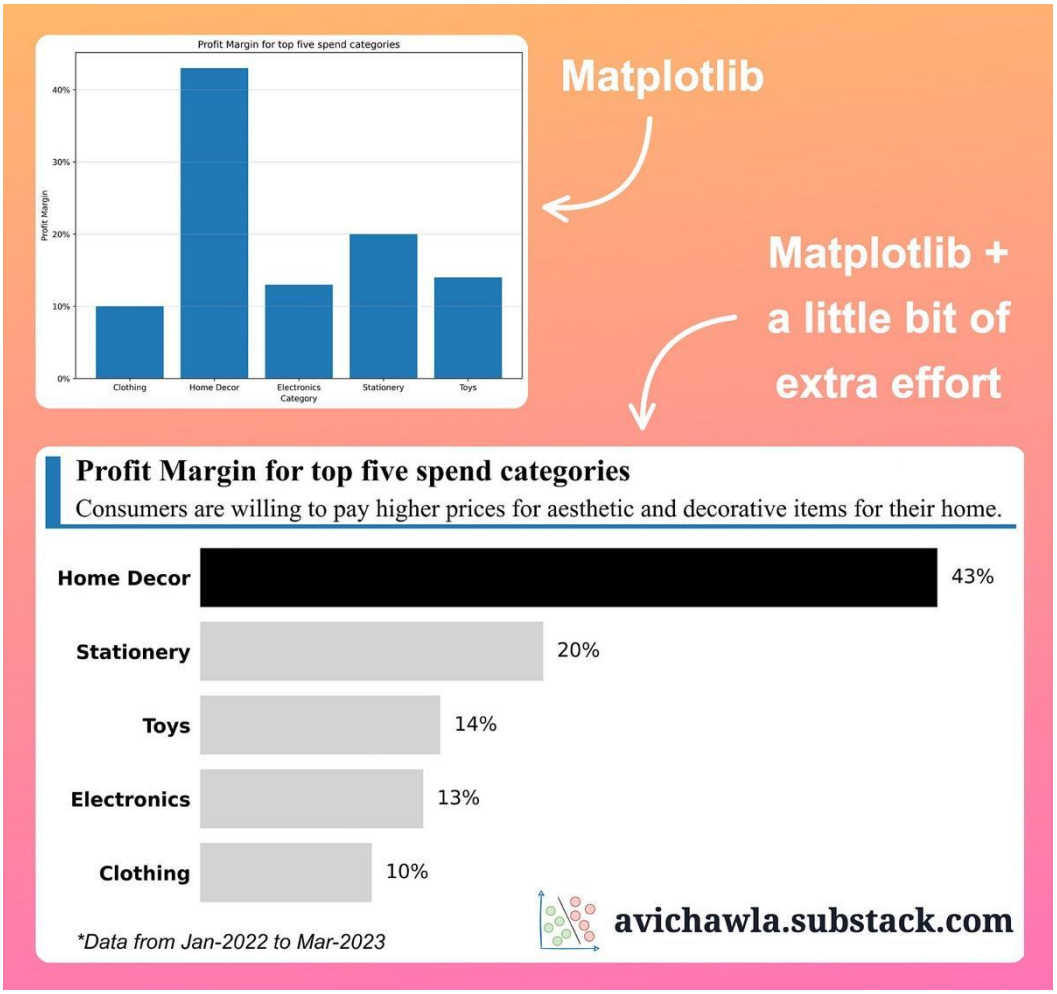


Profiling info includes column distribution, summary stats, null stats, and many more. Moreover, you can also generate the corresponding code, with its export feature.



Find more info here: GitHub Repo.

A Little Bit Of Extra Effort Can Hugely Transform Your Storytelling Skills



Matplotlib is pretty underrated when it comes to creating professional-looking plots. Yet, it is totally capable of doing so.

For instance, consider the two plots below.

Yes, both were created using matplotlib. But a bit of formatting makes the second plot much more informative, appealing, and easy to follow.

The title and subtitle significantly aid the story. Also, the footnote offers extra important information, which is nowhere to be seen in the basic plot.

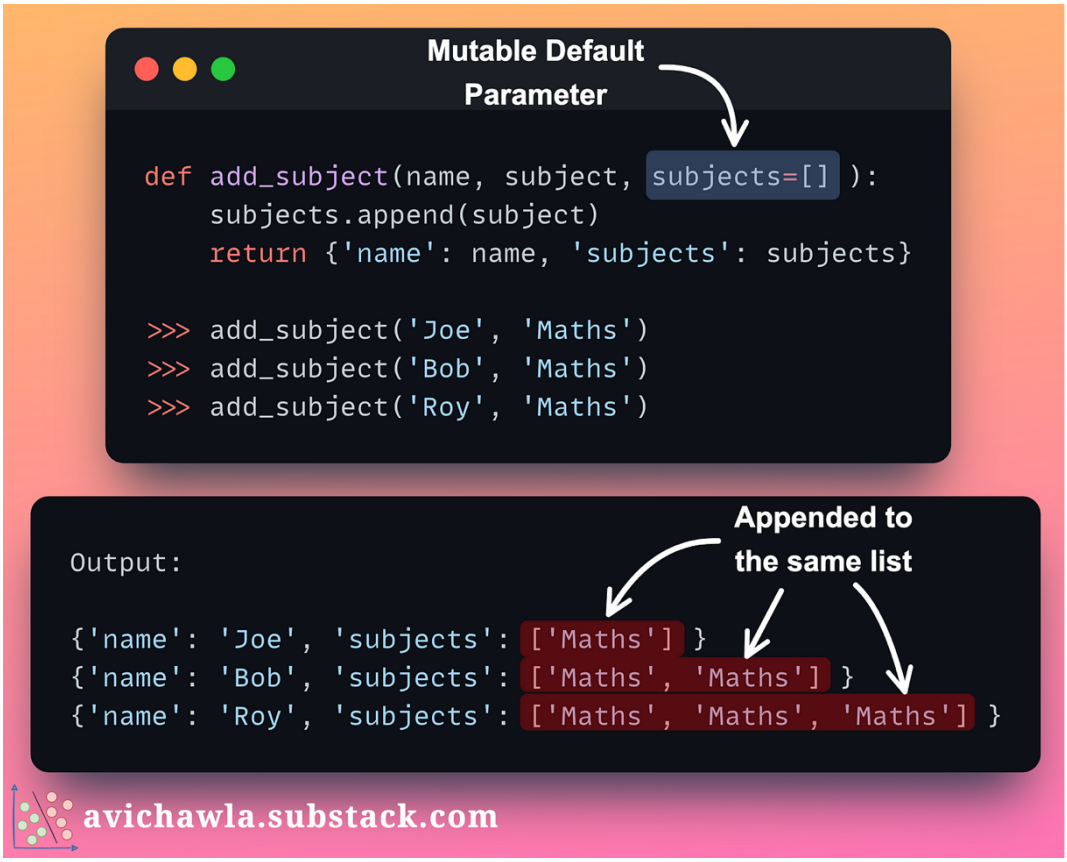
Lastly, the bold bar immediately draws the viewer's attention and conveys the category's importance.

So what's the message here?

Towards being a good data storyteller, ensure that your plot demands minimal effort from the viewer. Thus, don't shy away from putting in that extra effort. This is especially true for professional environments.

At times, it may be also good to ensure that your visualizations convey the right story, even if they are viewed in your absence.

A Nasty Hidden Feature of Python That Many Programmers Aren't Aware Of



Mutability in Python is possibly one of the most misunderstood and overlooked concepts. The above image demonstrates an example that many Python programmers (especially new ones) struggle to understand.

Can you figure it out? If not, let's understand it.

The default parameters of a function are evaluated right at the time the function is defined. In other words, they are not evaluated each time the function is called (like in C++).

Thus, as soon as a function is defined, the function object stores the default parameters in its `__defaults__` attribute. We can verify this below:

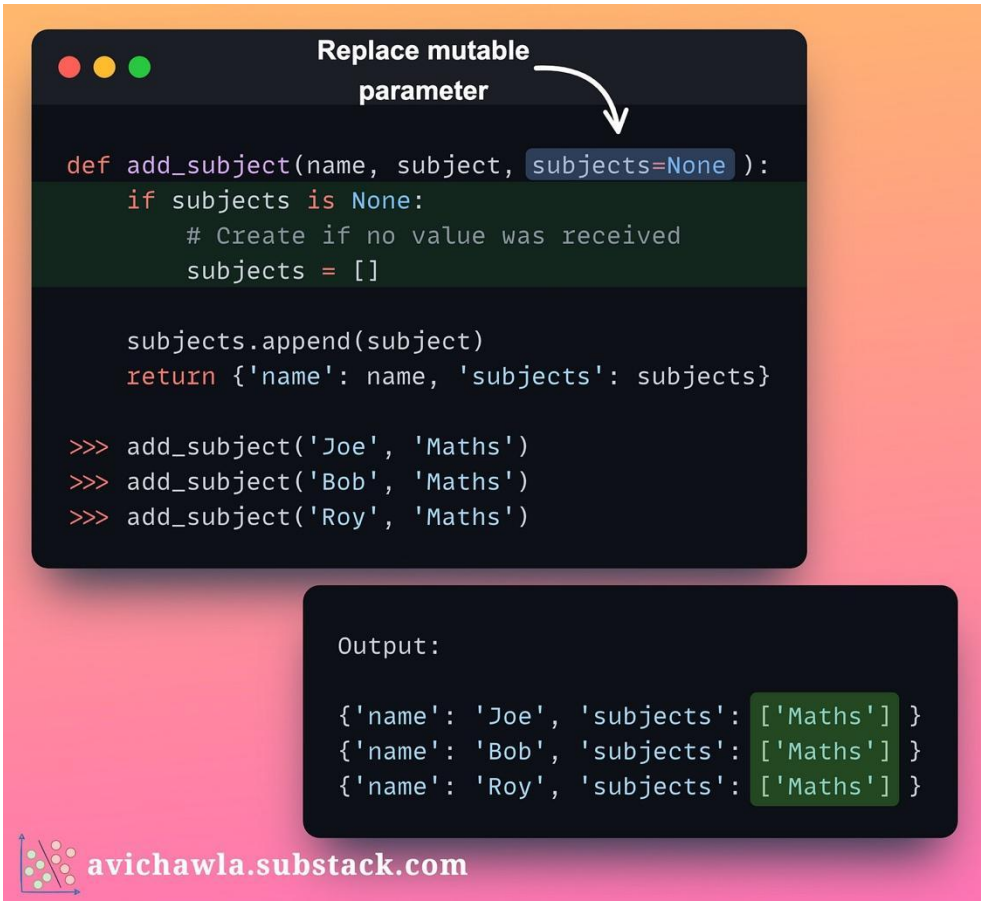
Thus, if you specify a mutable default parameter in a function and mutate it, you unknowingly and unintentionally modify the parameter for all future calls to that function.

This is shown in the demonstration below. Instead of creating a new list at each function call, Python appends the element to the same copy.

So what can we do to avoid this?

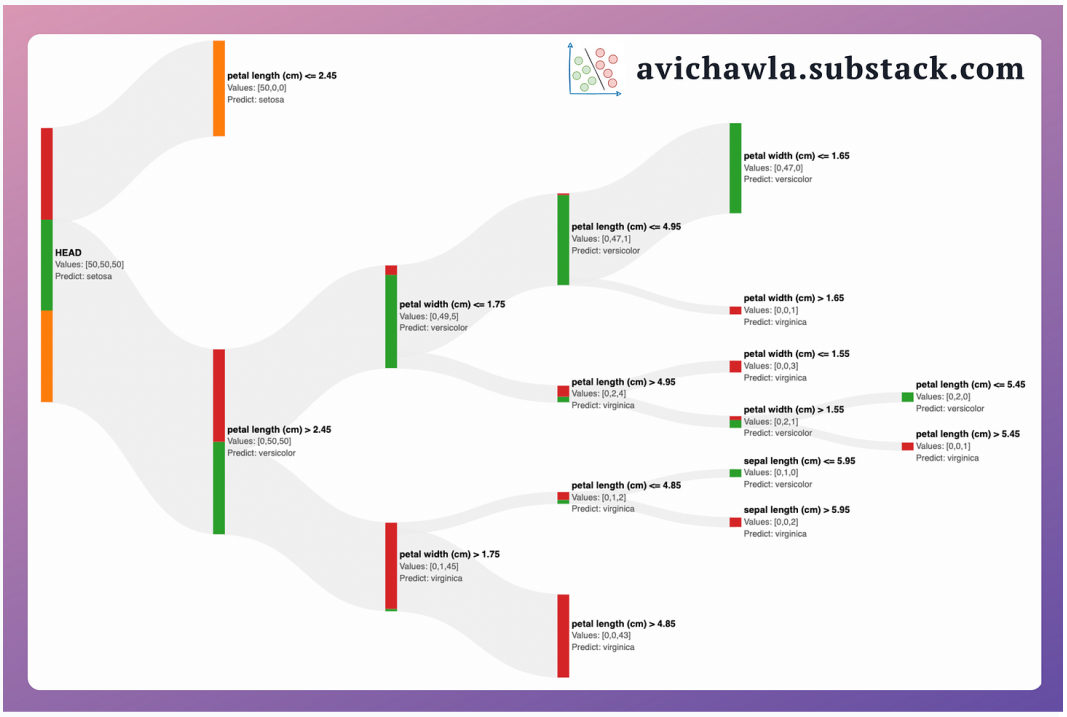
Instead of specifying a mutable default parameter in a function's definition, replace them with None. If the function does not receive a corresponding value during the function call, create the mutable object inside the function.

This is demonstrated below:



As shown above, we create a new list if the function didn't receive any value when it was called. This lets you avoid the unexpected behavior of mutating the same object.

Interactively Visualise A Decision Tree With A Sankey Diagram



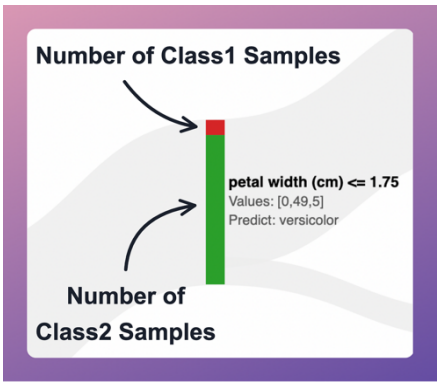
In one of my earlier posts, I explained why sklearn's decision trees always overfit the data with its default parameters (read here if you wish to recall).

To avoid this, it is always recommended to specify appropriate hyperparameter values. This includes the max depth of the tree, min samples in leaf nodes, etc.

But determining these hyperparameter values is often done using trial-and-error, which can be a bit tedious and time-consuming.

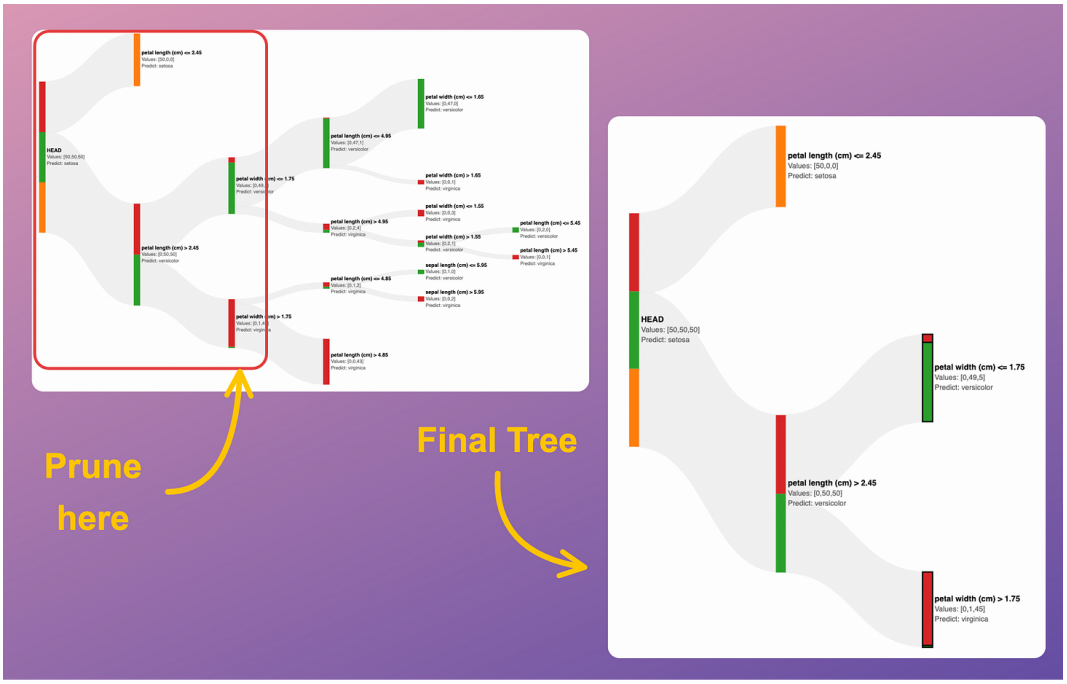
The Sankey diagram above allows you to interactively visualize the predictions of a decision tree at each node.

Also, the number of data points from each class is size-encoded on all nodes, as shown below.



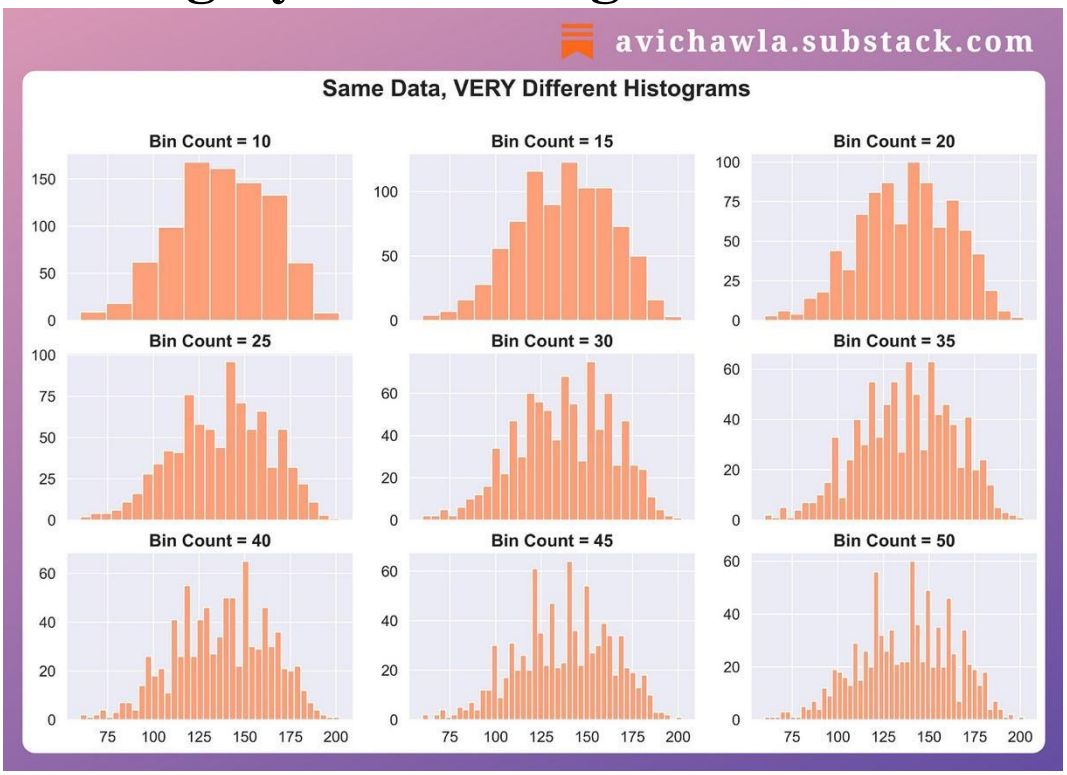
This immediately gives an estimate of the impurity of the node. Based on this, you can visually decide to prune the tree.

For instance, in the full decision tree shown below, pruning the tree at a depth of two appears to be reasonable.



Once you have obtained a rough estimate for these hyperparameter values, you can train a new decision tree. Next, measure its performance on new data to know if the decision tree is generalizing or not.

Use Histograms With Caution. They Are Highly Misleading!



Histograms are commonly used for data visualization. But, they can be misleading at times. Here's why.

Histograms divide the data into small bins and represent the frequency of each bin.

Thus, the choice of the number of bins you begin with can significantly impact its shape.

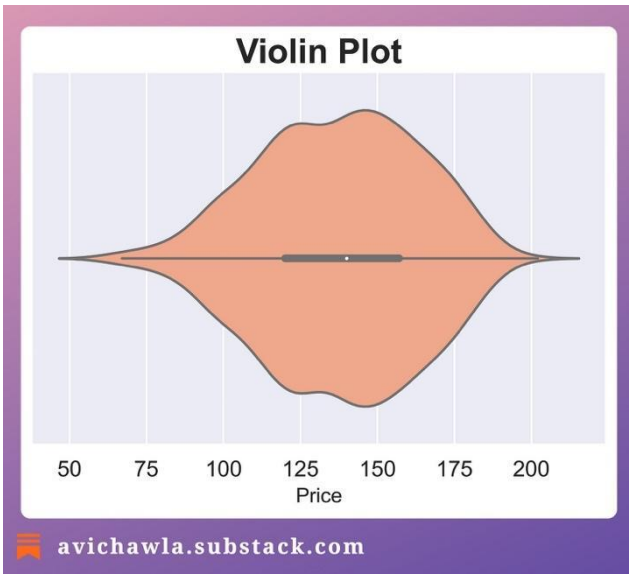
The figure above depicts the histograms obtained on the same data, but by altering the number of bins. Each histogram conveys a different story, even though the underlying data is the same.

This, at times, can be misleading and may lead you to draw the wrong conclusions.

The takeaway is NOT that histograms should not be used. Instead, look at the underlying distribution too. Here, a violin plot and a KDE plot can help.

Violin plot

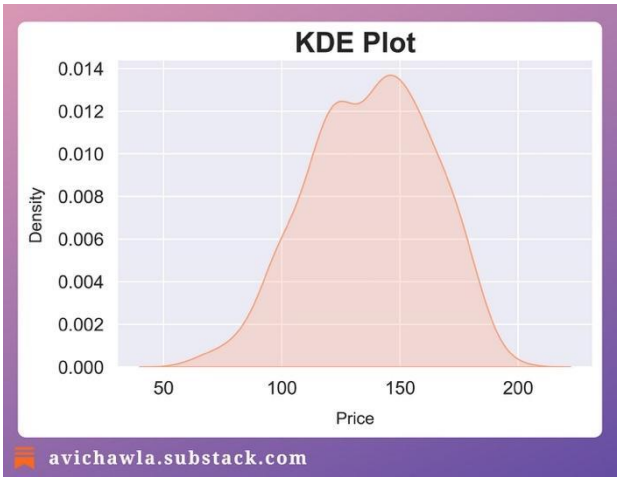
Similar to box plots, Violin plots also show the distribution of data based on quartiles. However, it also adds a kernel density estimation to display the density of data at different values.



This provides a more detailed view of the distribution, particularly in areas with higher density.

KDE plot

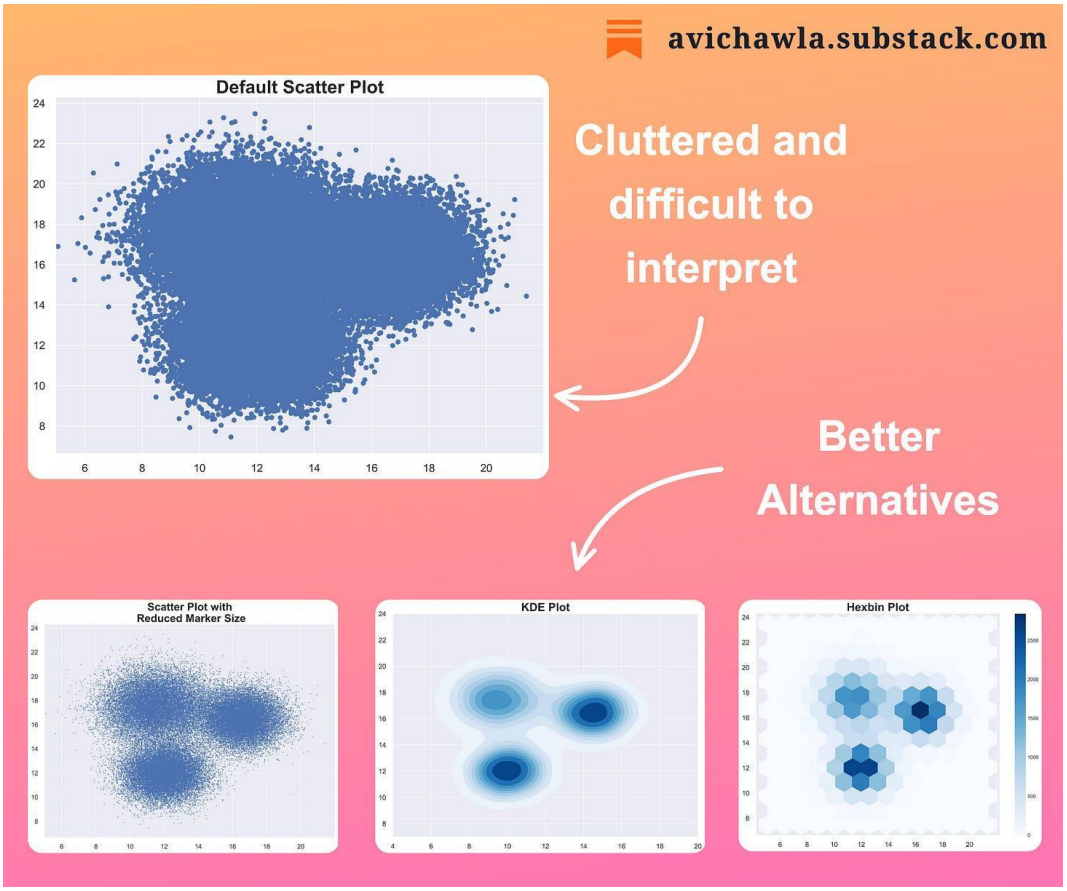
KDE plots use a smooth curve to represent the data distribution, without the need for binning, as shown below:



As a departing note, always remember that whenever you condense a dataset, you run the risk of losing important information.

Thus, be mindful of any limitations (and assumptions) of the visualizations you use. Also, consider using multiple methods to ensure that you are seeing the whole picture.

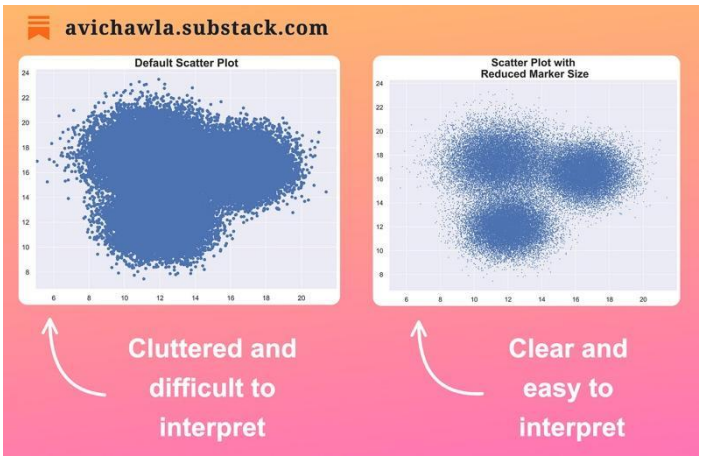
Three Simple Ways To (Instantly) Make Your Scatter Plots Clutter Free



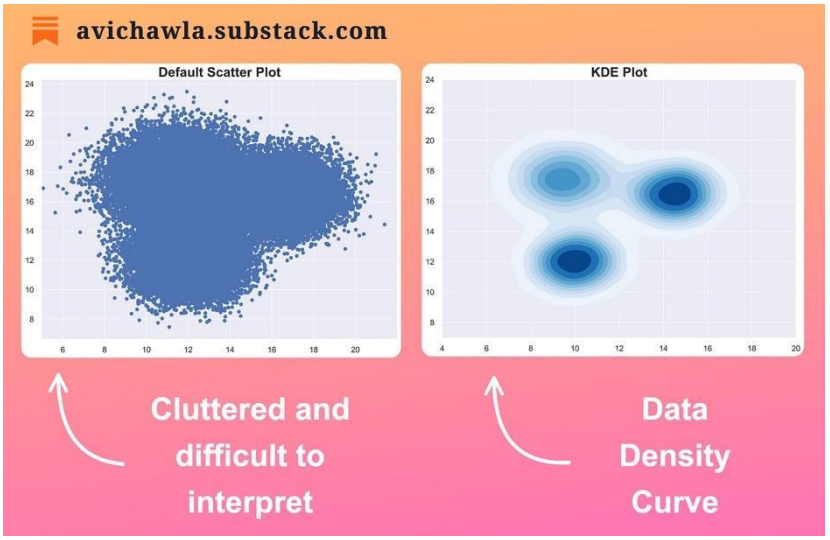
Scatter plots are commonly used in data visualization tasks. But when you have many data points, they often get too dense to interpret.

Here are a few techniques (and alternatives) you can use to make your data more interpretable in such cases.

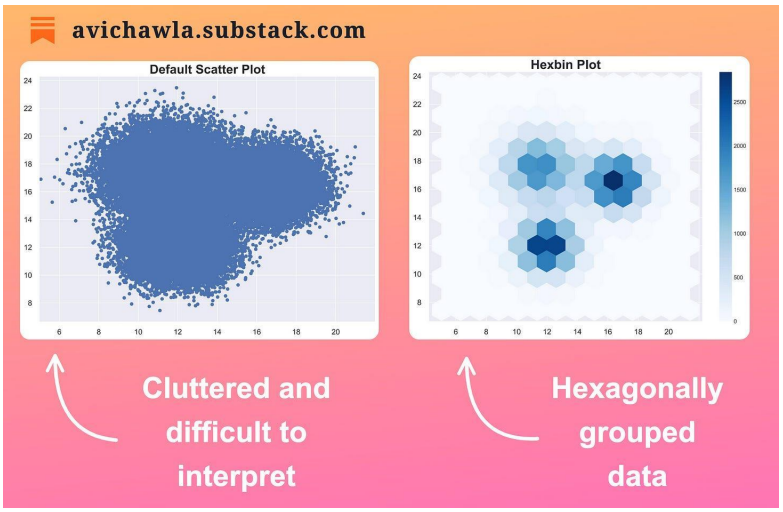
One of the simplest yet effective ways could be to reduce the marker size. This, at times, can instantly offer better clarity over the default plot.



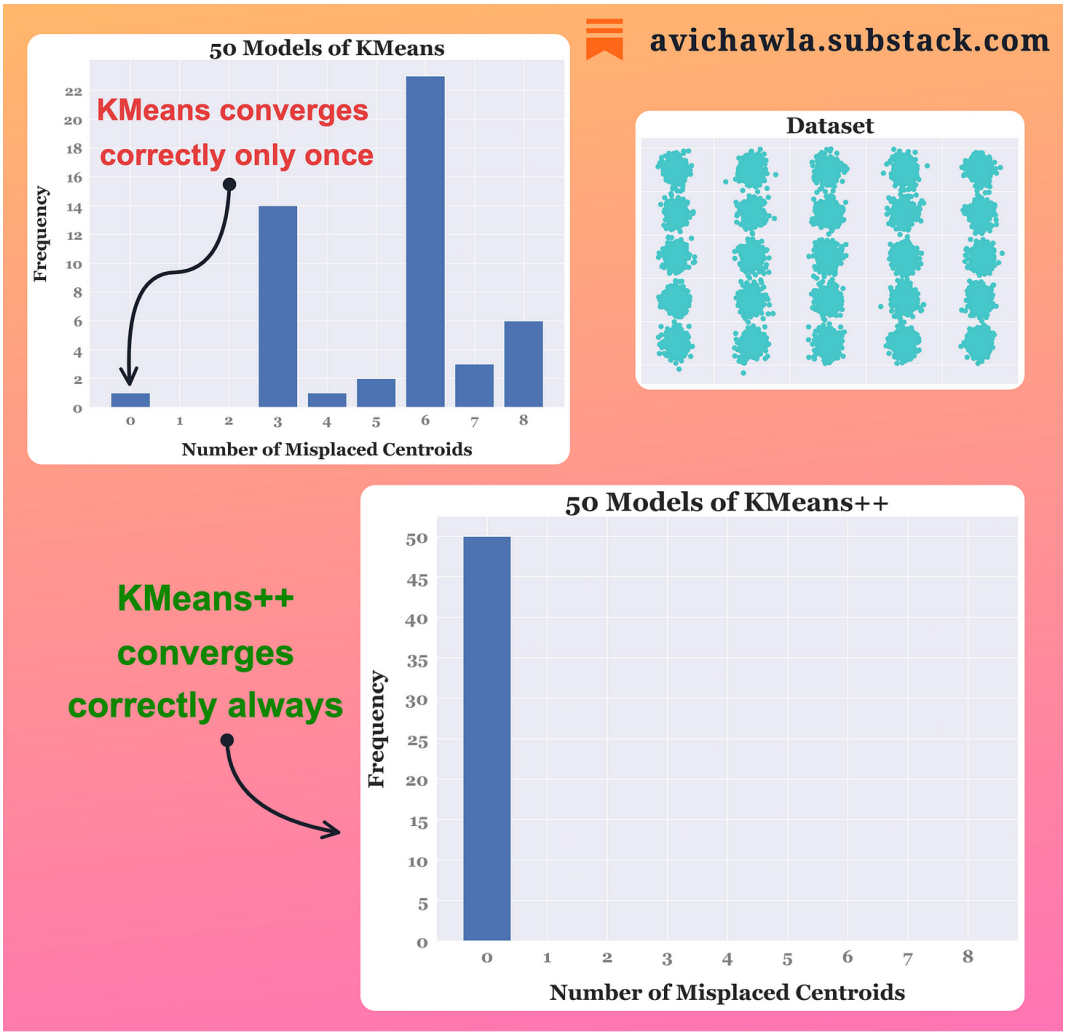
Next, as an alternative to a scatter plot, you can use a density plot, which depicts the data distribution. This makes it easier to identify regions of high and low density, which may not be evident from a scatter plot.



Lastly, another better alternative can be a hexbin plot. It bins the chart into hexagonal regions and assigns a color intensity based on the number of points in that area.


avichawla.substack.com
Default Scatter Plot
Hexbin Plot
Cluttered and difficult to interpret
Hexagonally grouped data

A (Highly) Important Point to Consider Before You Use KMeans Next Time



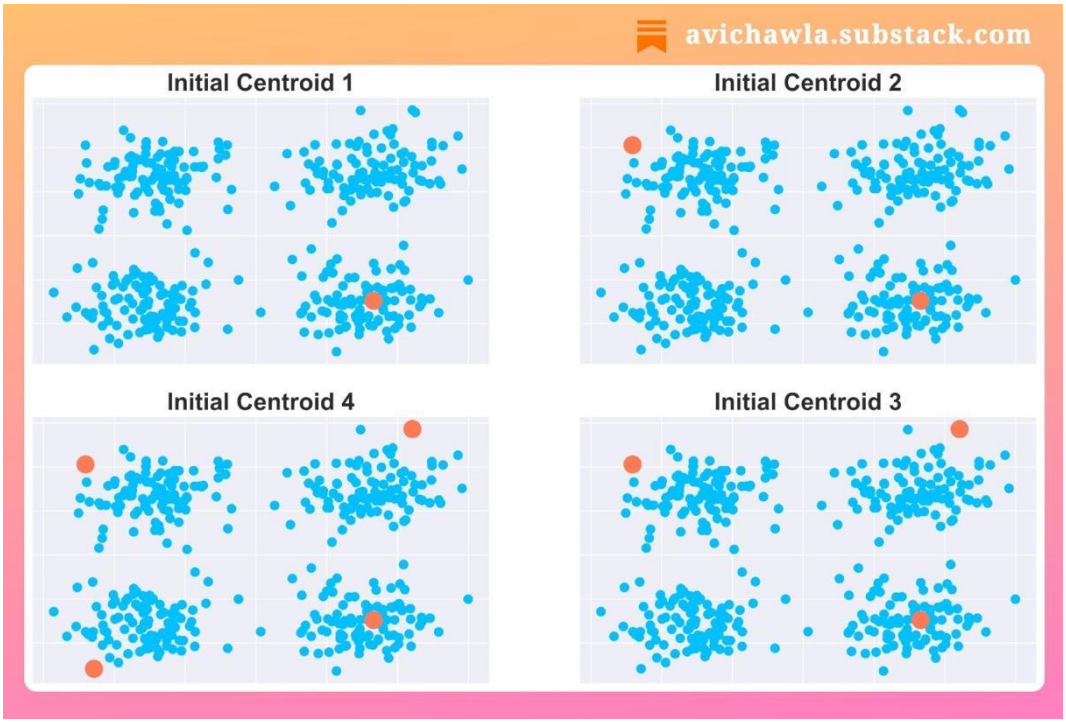
The most important yet often overlooked step of KMeans is its centroid initialization. Here's something to consider before you use it next time.

KMeans selects the initial centroids randomly. As a result, it fails to converge at times. This requires us to repeat clustering several times with different initialization.

Yet, repeated clustering may not guarantee that you will soon end up with the correct clusters. This is especially true when you have many centroids to begin with.

Instead, KMeans++ takes a smarter approach to initialize centroids.

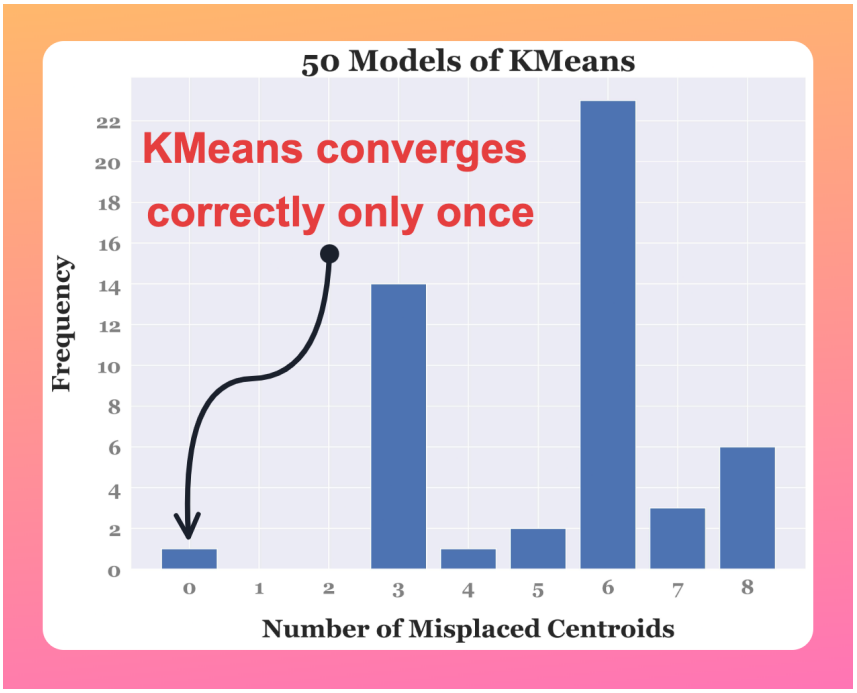
The first centroid is selected randomly. But the next centroid is chosen based on the distance from the first centroid.



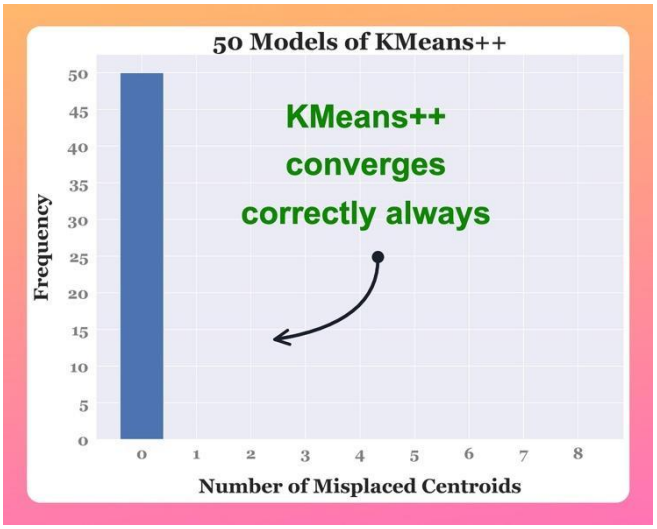
In other words, a point that is away from the first centroid is more likely to be selected as an initial centroid. This way, all the initial centroids are likely to lie in different clusters already, and the algorithm may converge faster and more accurately.

The impact is evident from the bar plots shown below. They depict the frequency of the number of misplaced centroids obtained (analyzed manually) after training 50 different models with KMeans and KMeans++.

On the given dataset, out of the 50 models, KMeans only produced zero misplaced centroids once, which is a success rate of just **2%**.



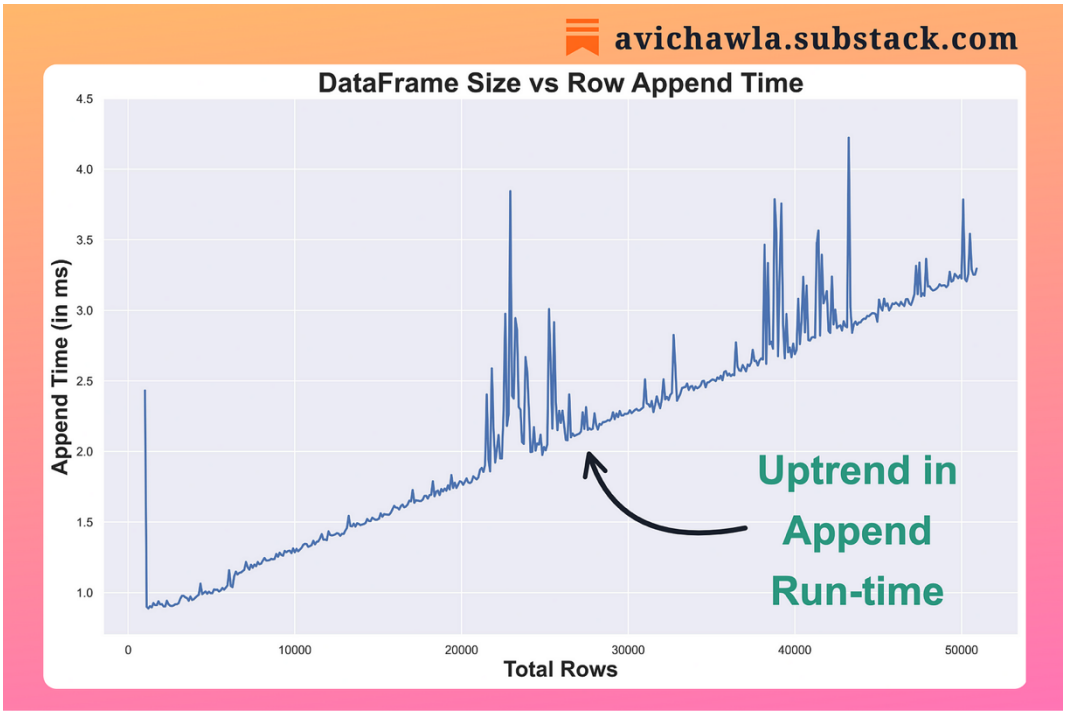
In contrast, KMeans++ never produced any misplaced centroids.



Luckily, if you are using sklearn, you don't need to worry about the initialization step. This is because sklearn, by default, resorts to the KMeans++ approach.

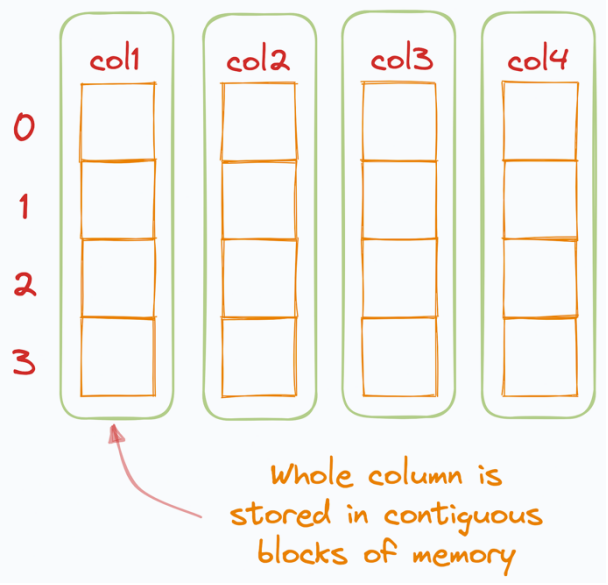
However, if you have a custom implementation, do give it a thought.

Why You Should Avoid Appending Rows To A DataFrame



As we append more and more rows to a Pandas DataFrame, the append run-time keeps increasing. Here's why.

A DataFrame is a column-major data structure. Thus, consecutive elements in a column are stored next to each other in memory.



As new rows are added, Pandas always wants to preserve its column-major form.

But while adding new rows, there may not be enough space to accommodate them while also preserving the column-major structure.

In such a case, existing data is moved to a new memory location, where Pandas finds a contiguous block of memory.

Thus, as the size grows, memory reallocation gets more frequent, and the run time keeps increasing.

The reason for spikes in this graph may be because a column taking higher memory was moved to a new location at this point, thereby taking more time to reallocate, or many columns were shifted at once.

So what can we do to mitigate this?

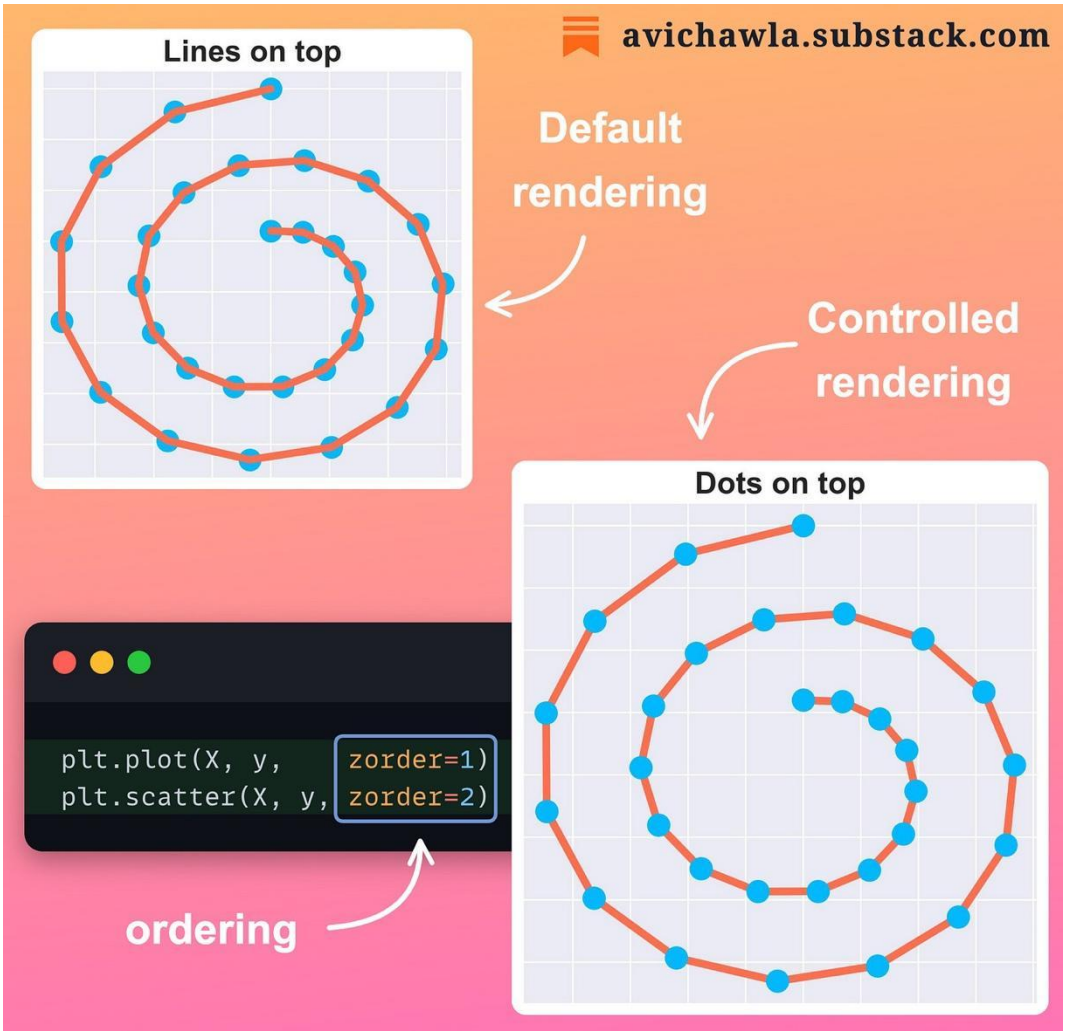
The increase in run-time solely arises because Pandas is trying to maintain its column-major structure.

Thus, if you intend to grow a dataframe (row-wise) this frequently, it is better to first convert the dataframe to another data structure, a dictionary or a numpy array, for instance.

Carry out the append operations here, and when you are done, convert it back to a dataframe.

P.S. Adding new columns is not a problem. This is because this operation does not conflict with other columns.

Matplotlib Has Numerous Hidden Gems. Here's One of Them.



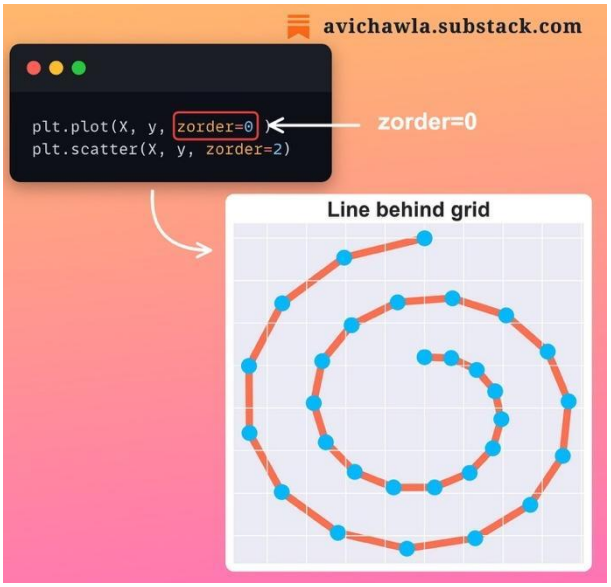
One of the best yet underrated and underutilized potentials of matplotlib is customizability. Here's a pretty interesting thing you can do with it.

By default, matplotlib renders different types of elements (also called artists), like plots, legend, texts, etc., in a specific order.

But this ordering may not be desirable in all cases, especially when there are overlapping elements in a plot, or the default rendering is hiding some crucial details.

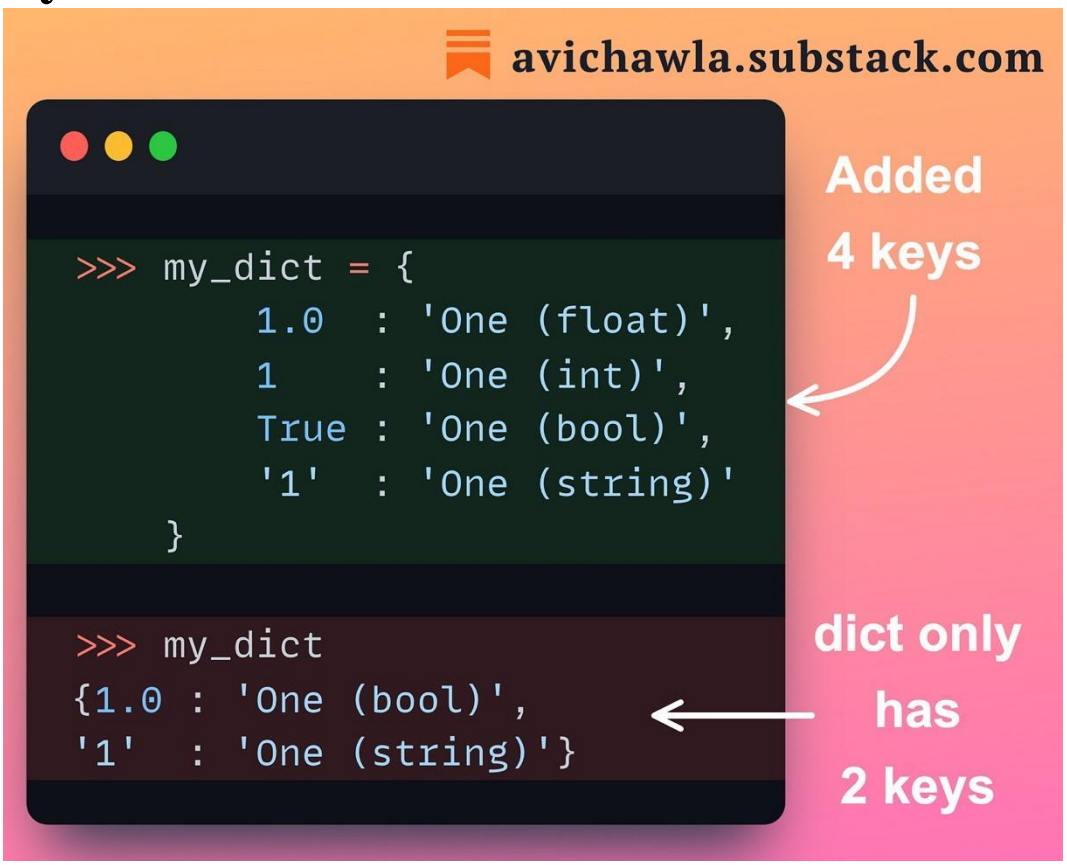
With the `zorder` parameter, you can control this rendering order. As a result, plots with higher `zorder` value appear closer to the viewer and are drawn on top of artists with lower `zorder` values.

Lastly, in the above demonstration, if we specify `zorder=0` for the line plot, we notice that it goes behind the grid lines.



You can find more details about `zorder` here: Matplotlib docs.

A Counterintuitive Thing About Python Dictionaries

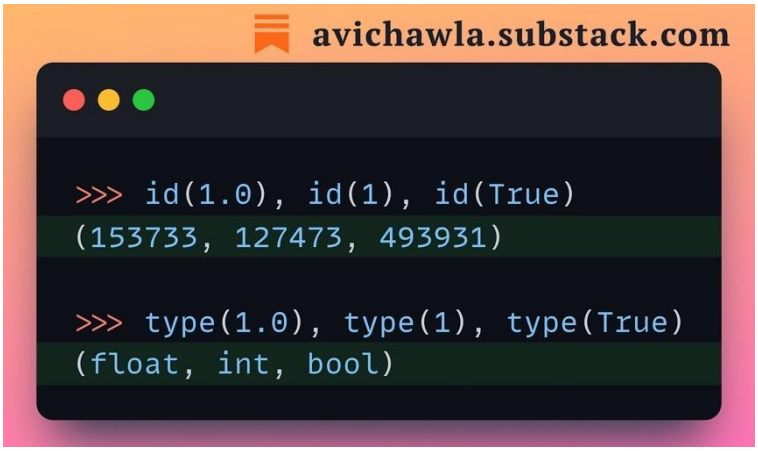


Despite adding 4 distinct keys to a Python dictionary, can you tell why it only preserves two of them?

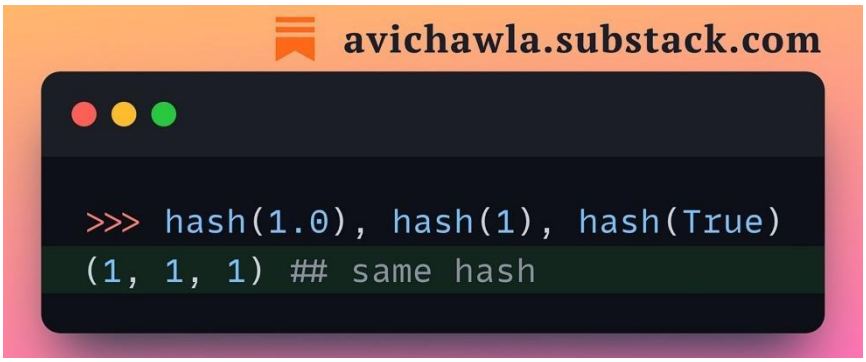
Here's why.

In Python, dictionaries find a key based on the equivalence of hash (computed using hash()), but not identity (computed using id()).

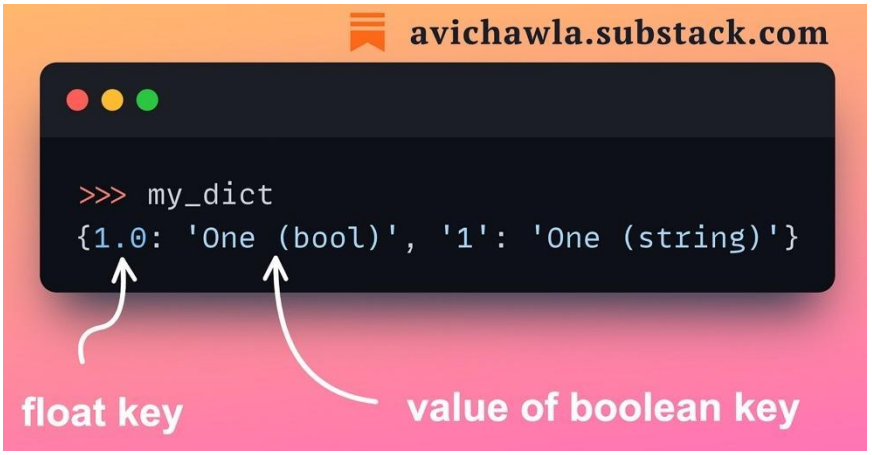
In this case, there's no doubt that 1.0, 1, and True inherently have different datatypes and are also different objects. This is shown below:



Yet, as they share the same hash value, the dictionary considers them as the same keys.



But did you notice that in the demonstration, the final key is 1.0, while the value corresponds to the key True.



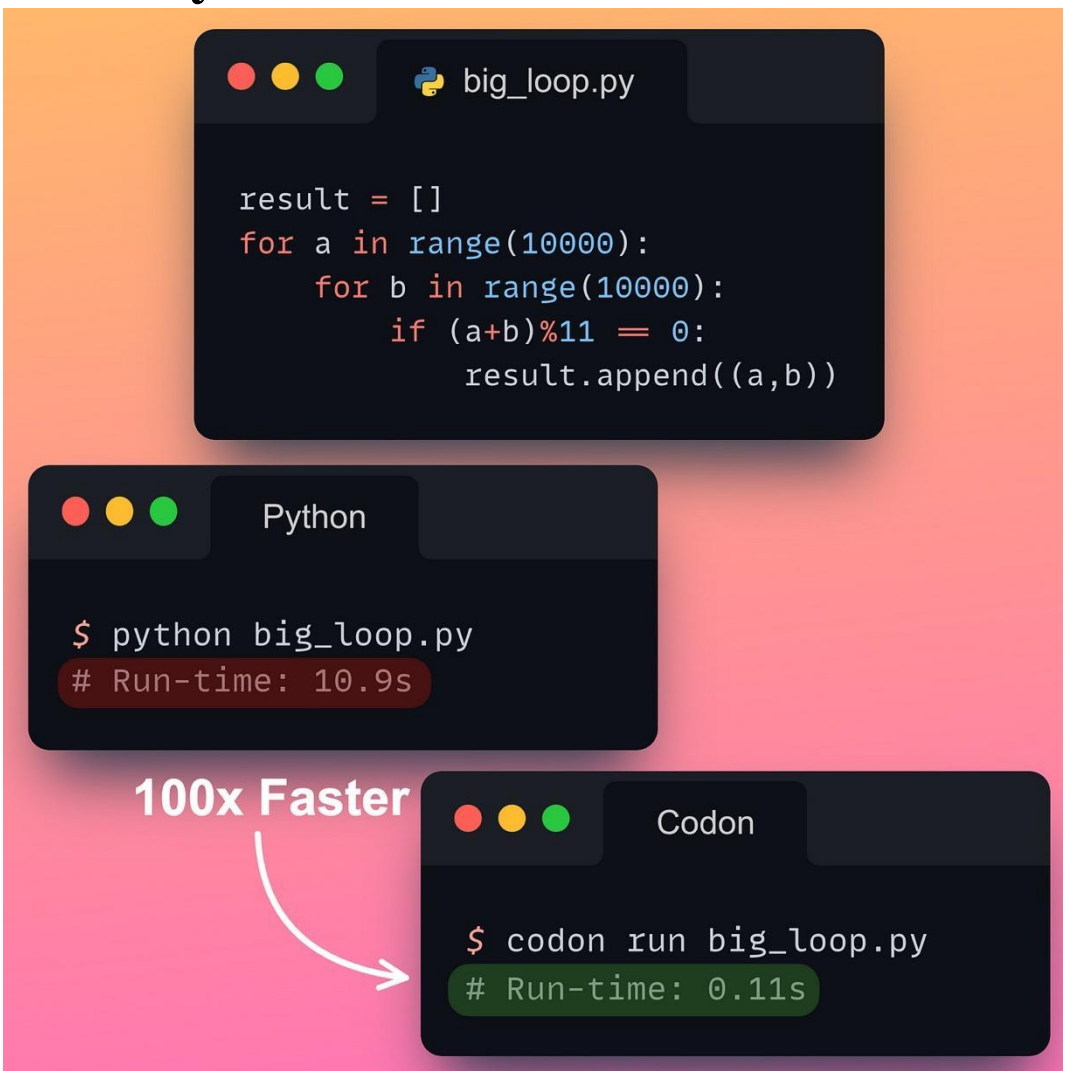
This is because, at first, `1.0` is added as a key and its value is `'One (float)'`. Next, while adding the key `1`, python recognizes it as an equivalence of the hash value.

Thus, the value corresponding to `1.0` is overwritten by `'One (int)'`, while the key (`1.0`) is kept as is.

Finally, while adding `True`, another hash equivalence is encountered with an existing key of `1.0`. Yet again, the value corresponding to `1.0`, which was updated to `'One (int)'` in the previous step, is overwritten by `'One (bool)'`.

I am sure you may have already guessed why the string key '`1`' is retained.

Probably The Fastest Way To Execute Your Python Code



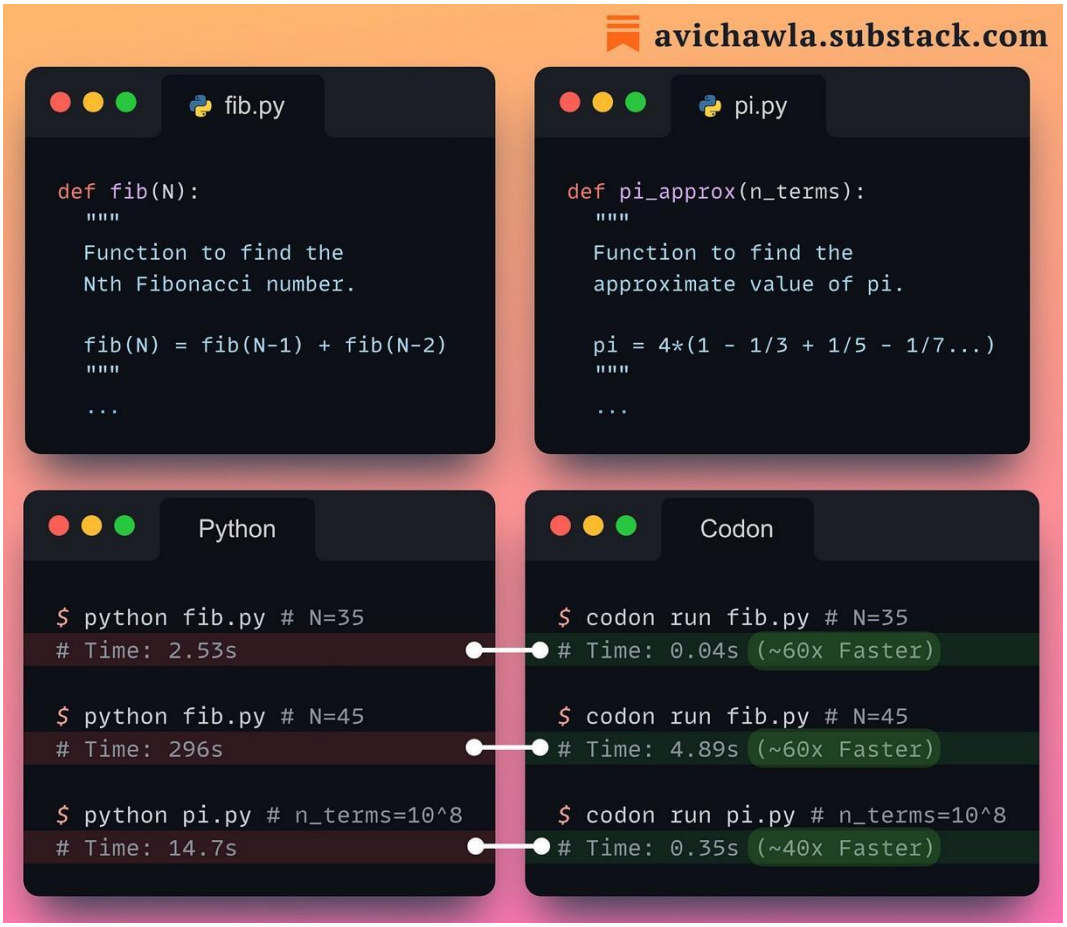
Many Python programmers are often frustrated with Python's run-time. Here's how you can make your code blazingly fast by changing just one line.

Codon is an open-source, high-performance Python compiler. In contrast to being an interpreter, it compiles your python code to fast machine code.

Thus, post compilation, your code runs at native machine code speed. As a result, typical speedups are often of the order **50x** or more.

According to the official docs, if you know Python, you already know 99% of Codon. There are very minute differences between the two, which you can read here: Codon docs.

Find some more benchmarking results between Python and Codon below:

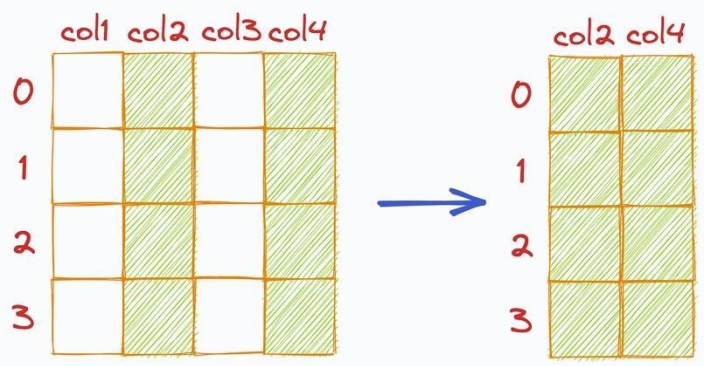

avichawla.substack.com
fib.py
def fib(N):
"""
Function to find the
Nth Fibonacci number.
fib(N) = fib(N-1) + fib(N-2)
"""
...
pi.py
def pi_approx(n_terms):
"""
Function to find the
approximate value of pi.
pi = 4*(1 - 1/3 + 1/5 - 1/7...)
"""
...
Python
$ python fib.py # N=35
# Time: 2.53s
$ python fib.py # N=45
# Time: 296s
$ python pi.py # n_terms=10^8
# Time: 14.7s
Codon
$ codon run fib.py # N=35
# Time: 0.04s (~60x Faster)
$ codon run fib.py # N=45
# Time: 4.89s (~60x Faster)
$ codon run pi.py # n_terms=10^8
# Time: 0.35s (~40x Faster)

Are You Sure You Are Using The Correct Pandas Terminologies?

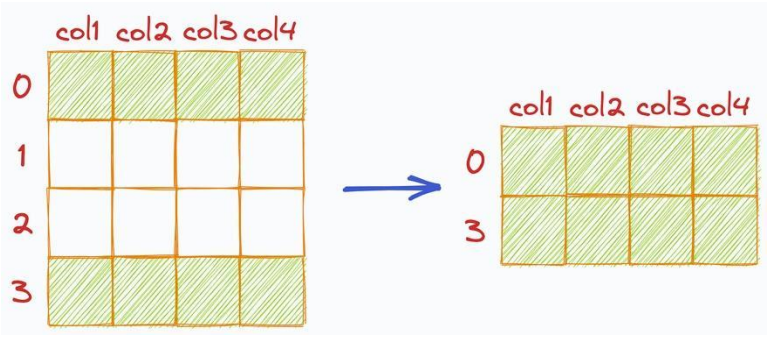
Many Pandas users use the dataframe subsetting terminologies incorrectly. So let's spend a minute to get it straight.

SUBSETTING means extracting value(s) from a dataframe. This can be done in four ways:

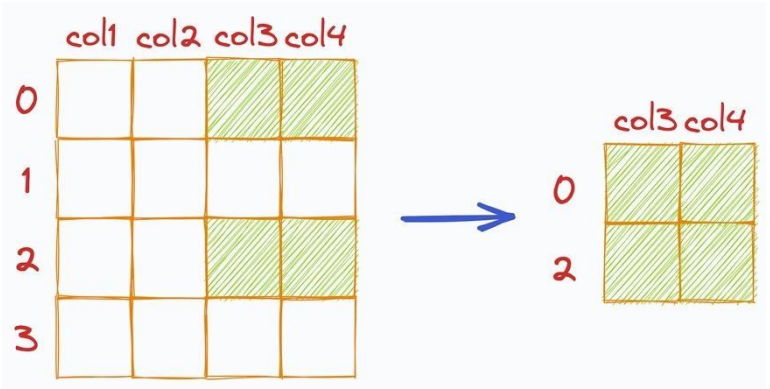
1) We call it **SELECTING** when we extract one or more of its **COLUMNS** based on index location or name. The output contains some columns and all rows.



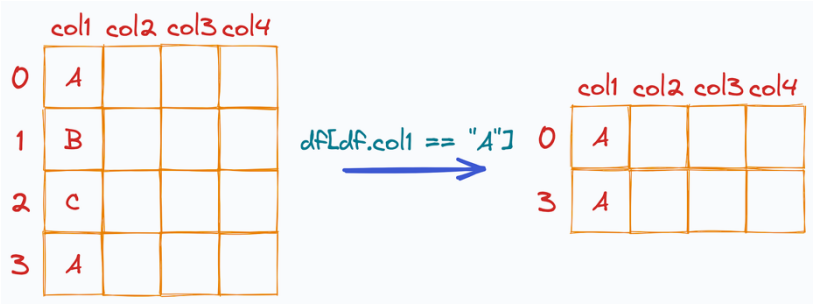
2) We call it **SLICING** when we extract one or more of its **ROWS** based on index location or name. The output contains some rows and all columns.



3) We call it **INDEXING** when we extract both **ROWS** and **COLUMNS** based on index location or name.



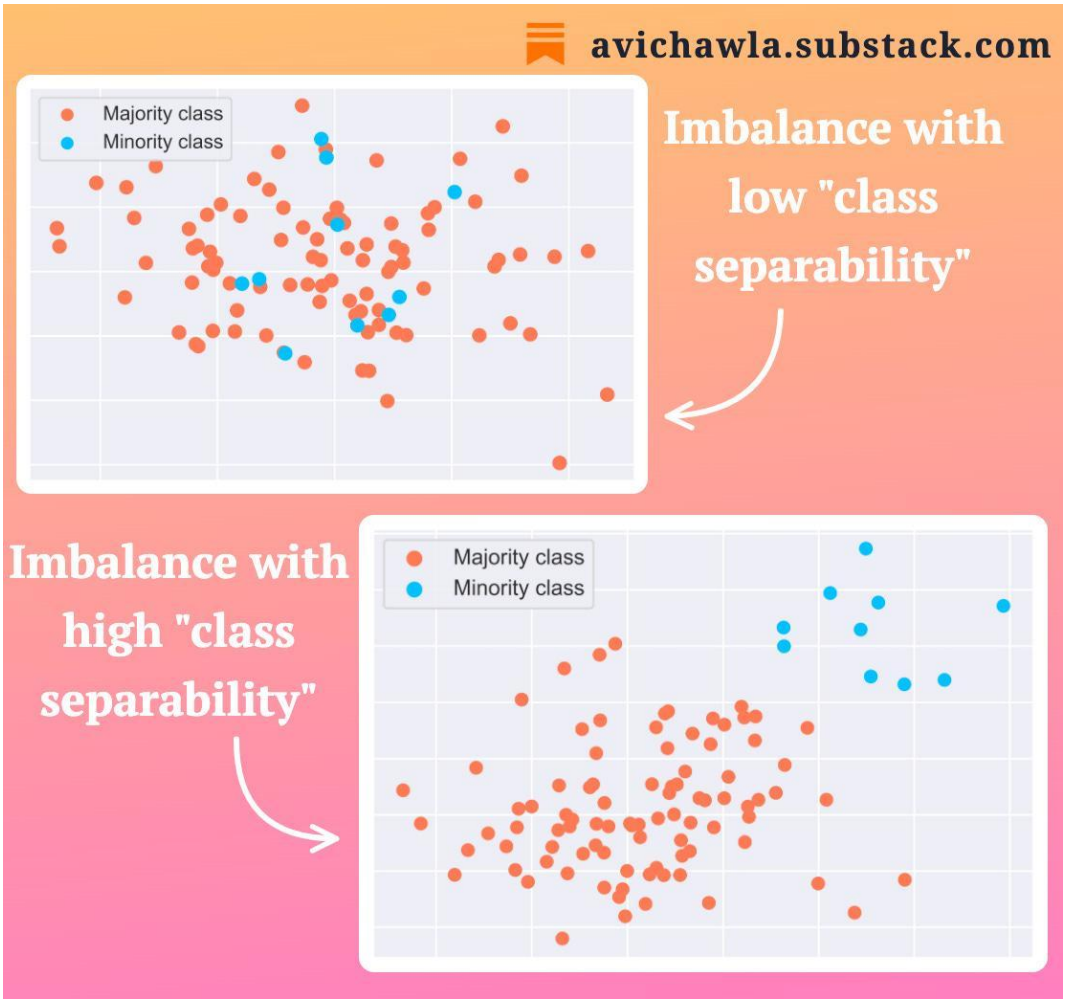
4) We call it **FILTERING** when we extract **ROWS** and **COLUMNS** based on conditions.



Of course, there are many other ways you can perform these four operations.

Here's a comprehensive Pandas guide I prepared once: Pandas Map. Please refer to the "DF Subset" branch to read about various subsetting methods :)

Is Class Imbalance Always A Big Problem To Deal With?

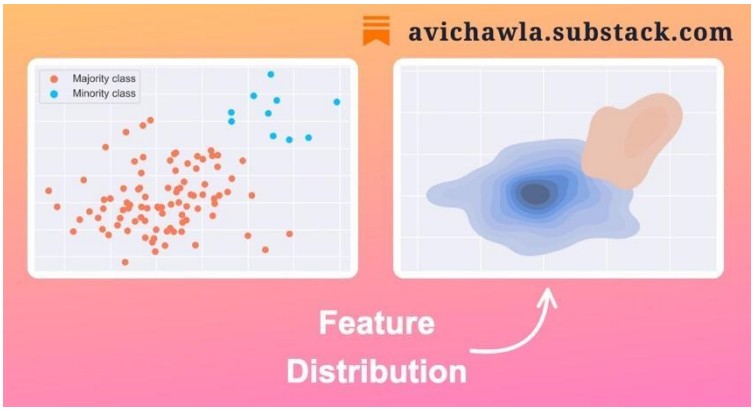


Addressing class imbalance is often a challenge in ML. Yet, it may not always cause a problem. Here's why.

One key factor in determining the impact of imbalance is **class separability**.

As the name suggests, it measures the degree to which two or more classes can be distinguished or separated from each other based on their feature values.

When classes are highly separable, there is little overlap between their feature distributions (as shown below). This makes it easier for a classifier to correctly identify the class of a new instance.

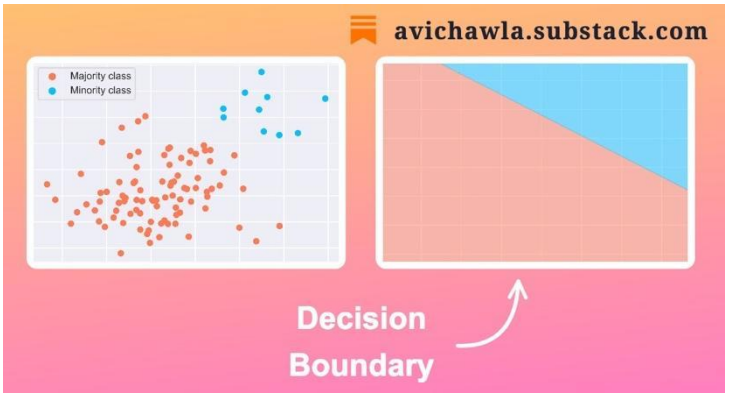


Thus, despite imbalance, even if your data has a high degree of class separability, imbalance may not be a problem per se.

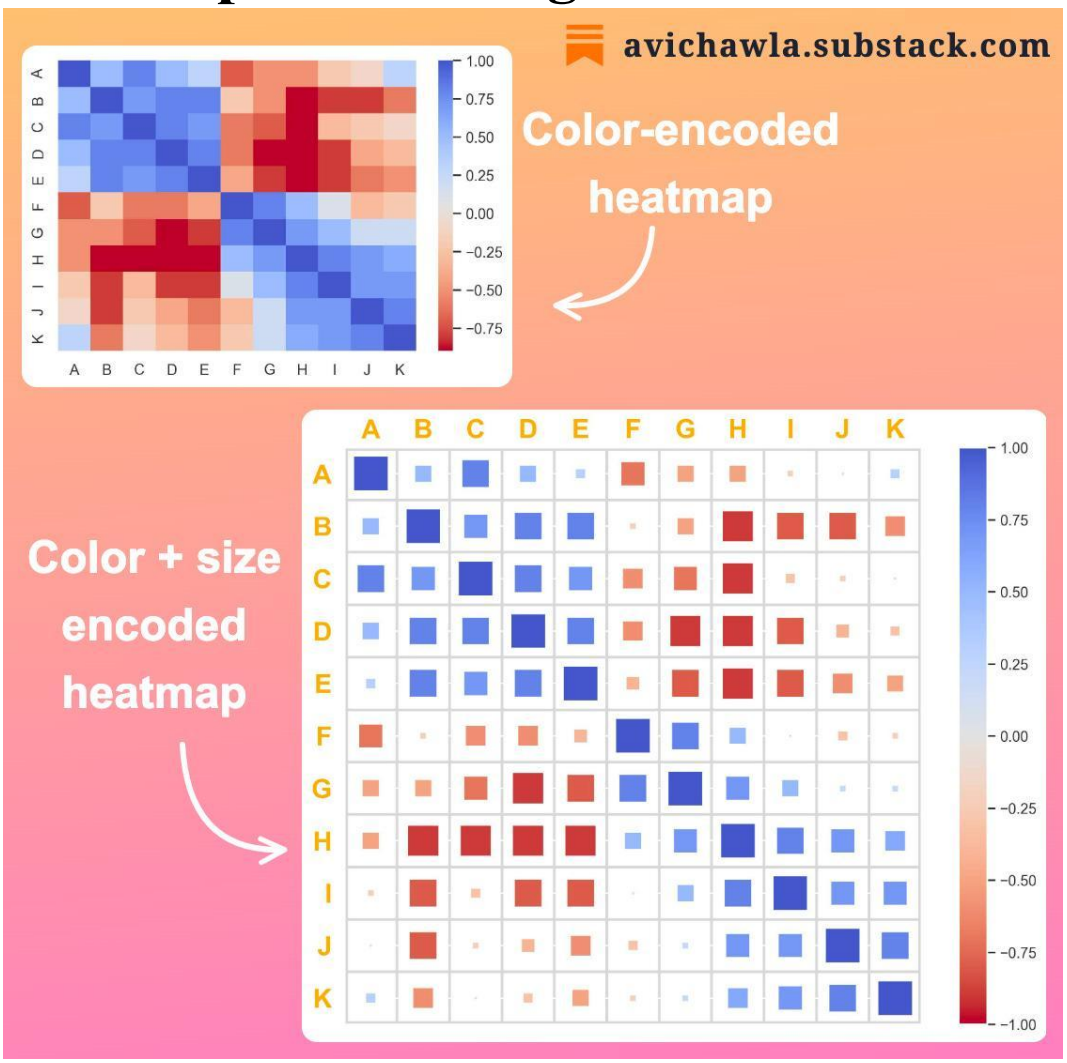
To conclude, consider estimating the class separability before jumping to any sophisticated modeling steps.

This can be done visually or by evaluating imbalance-specific metrics on simple models.

The figure below depicts the decision boundary learned by a logistic regression model on the class-separable dataset.



A Simple Trick That Will Make Heatmaps More Elegant



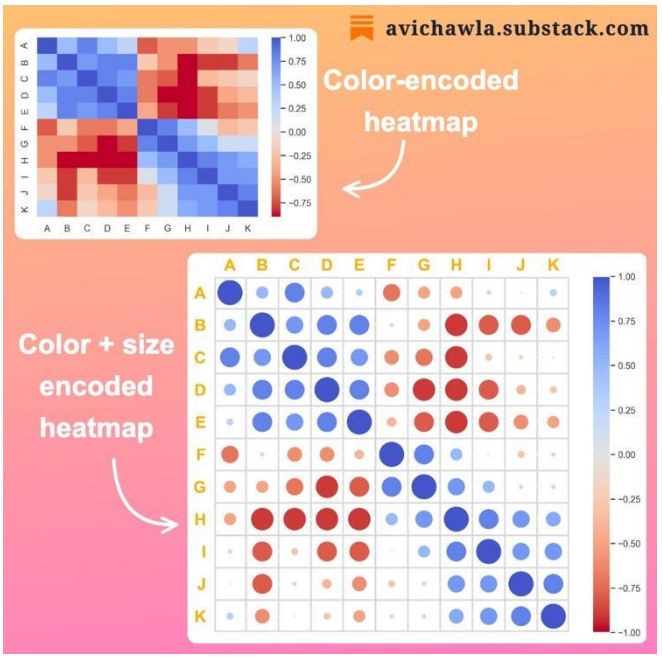
Heatmaps often make data analysis much easier. Yet, they can be further enriched with a simple modification.

A traditional heatmap represents the values using a color scale. Yet, mapping the cell color to numbers is still challenging.

Embedding a size component can be extremely helpful in such cases. In essence, the bigger the size, the higher the absolute value.

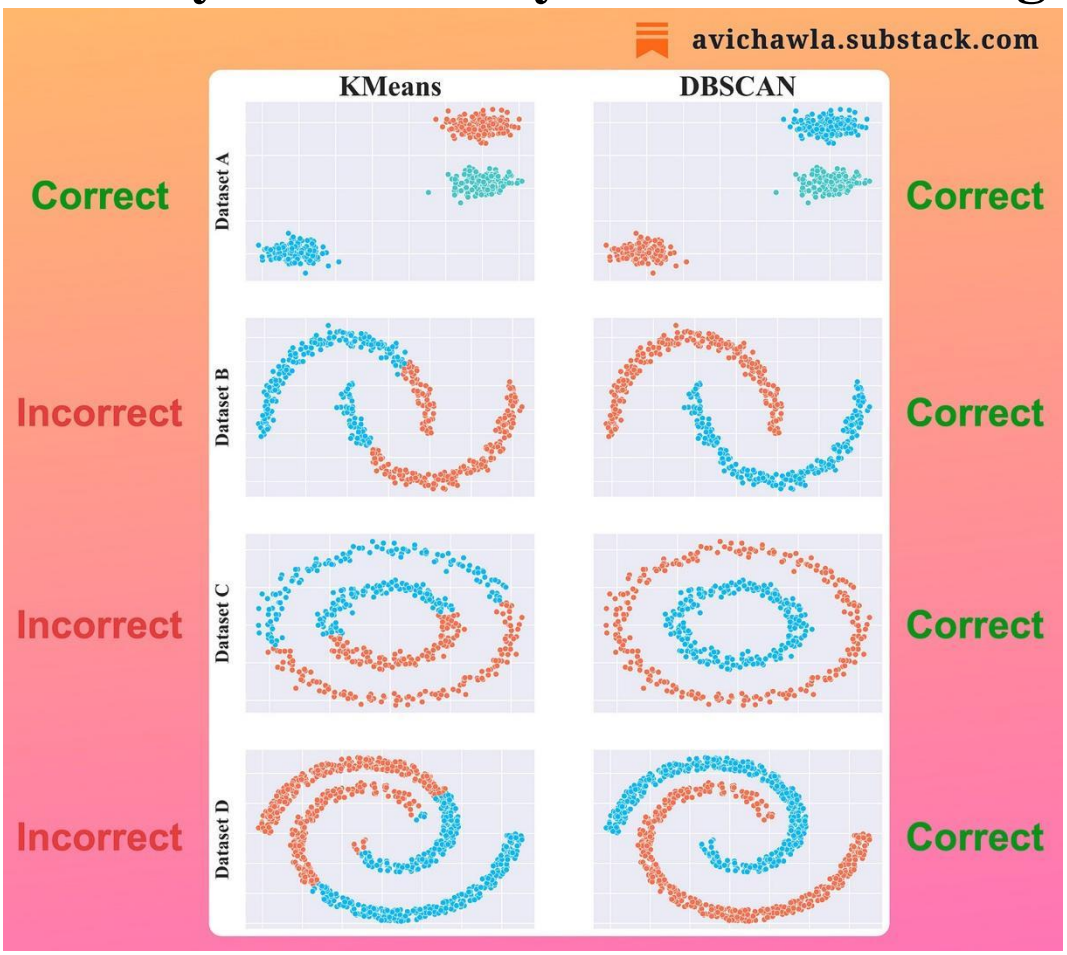
This is especially useful to make heatmaps cleaner, as many values nearer to zero will immediately shrink.

In fact, you can represent the size with any other shape. Below, I created the same heatmap using a circle instead:



Find the code for this post here: GitHub.

A Visual Comparison Between Locality and Density-based Clustering



The utility of KMeans is limited to datasets with spherical clusters. Thus, any variation is likely to produce incorrect clustering.

Density-based clustering algorithms, such as DBSCAN, can be a better alternative in such cases.

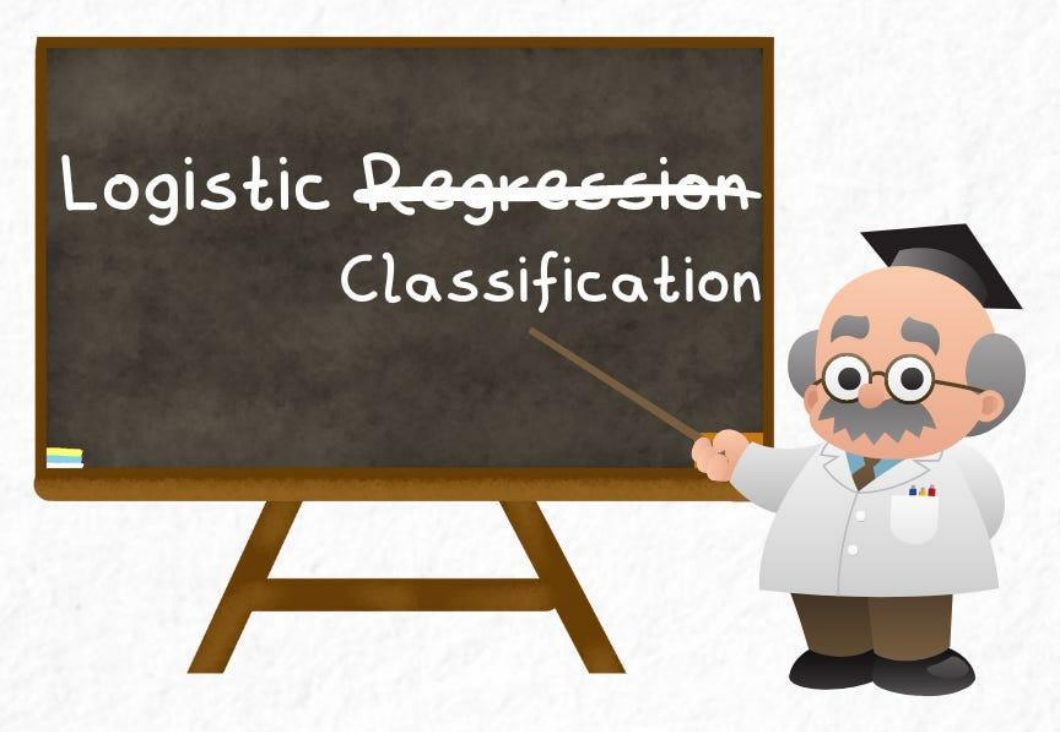
They cluster data points based on density, making them robust to datasets of varying shapes and sizes.

The image depicts a comparison of KMeans vs. DBSCAN on multiple datasets.

As shown, KMeans only works well when the dataset has spherical clusters. But in all other cases, it fails to produce correct clusters.

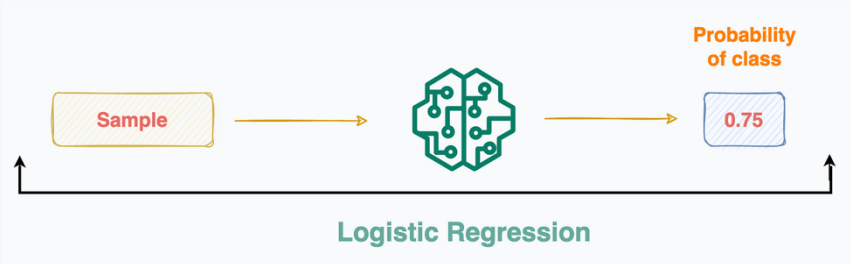
Find more here: Sklearn Guide.

Why Don't We Call It Logistic Classification Instead?

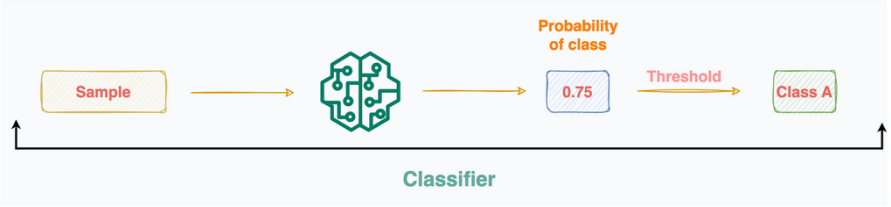


Have you ever wondered why logistic regression is called "regression" when we only use it for classification tasks? Why not call it "logistic classification" instead? Here's why.

Most of us interpret logistic regression as a classification algorithm. However, it is a regression algorithm by nature. This is because it predicts a continuous outcome, which is the probability of a class.

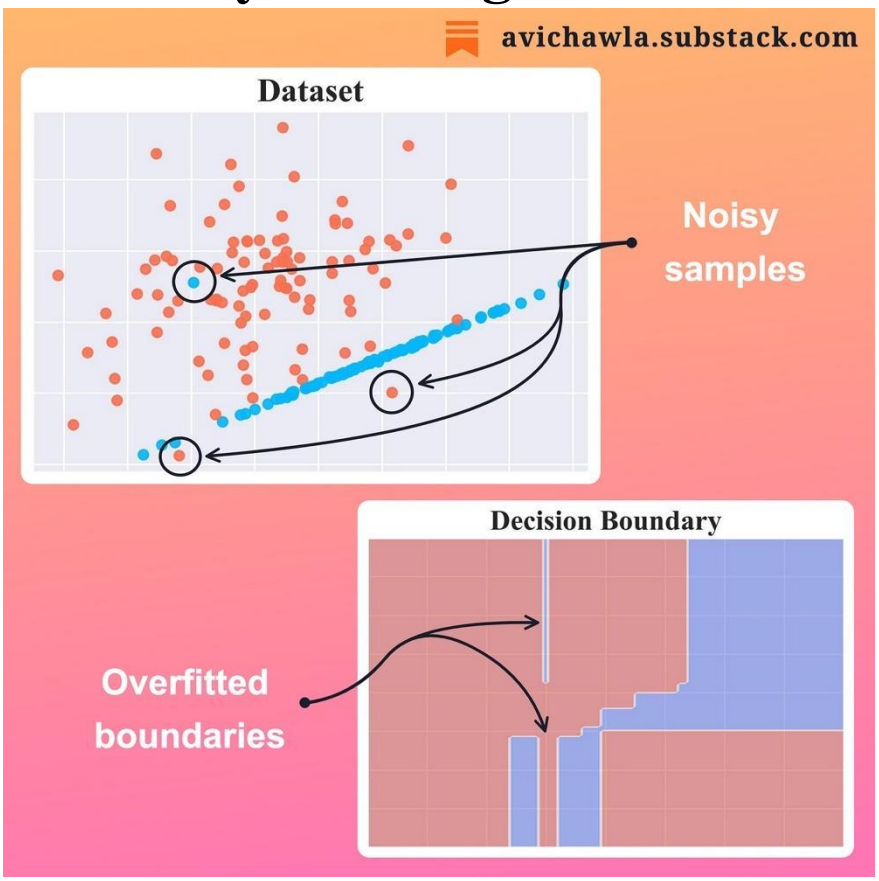


It is only when we apply those thresholds and change the interpretation of its output that the whole pipeline becomes a classifier.



Yet, intrinsically, it is never the algorithm performing the classification. The algorithm always adheres to regression. Instead, it is that extra step of applying probability thresholds that classifies a sample.

A Typical Thing About Decision Trees Which Many Often Ignore



Although decision trees are simple and intuitive, they always need a bit of extra caution. Here's what you should always remember while training them.

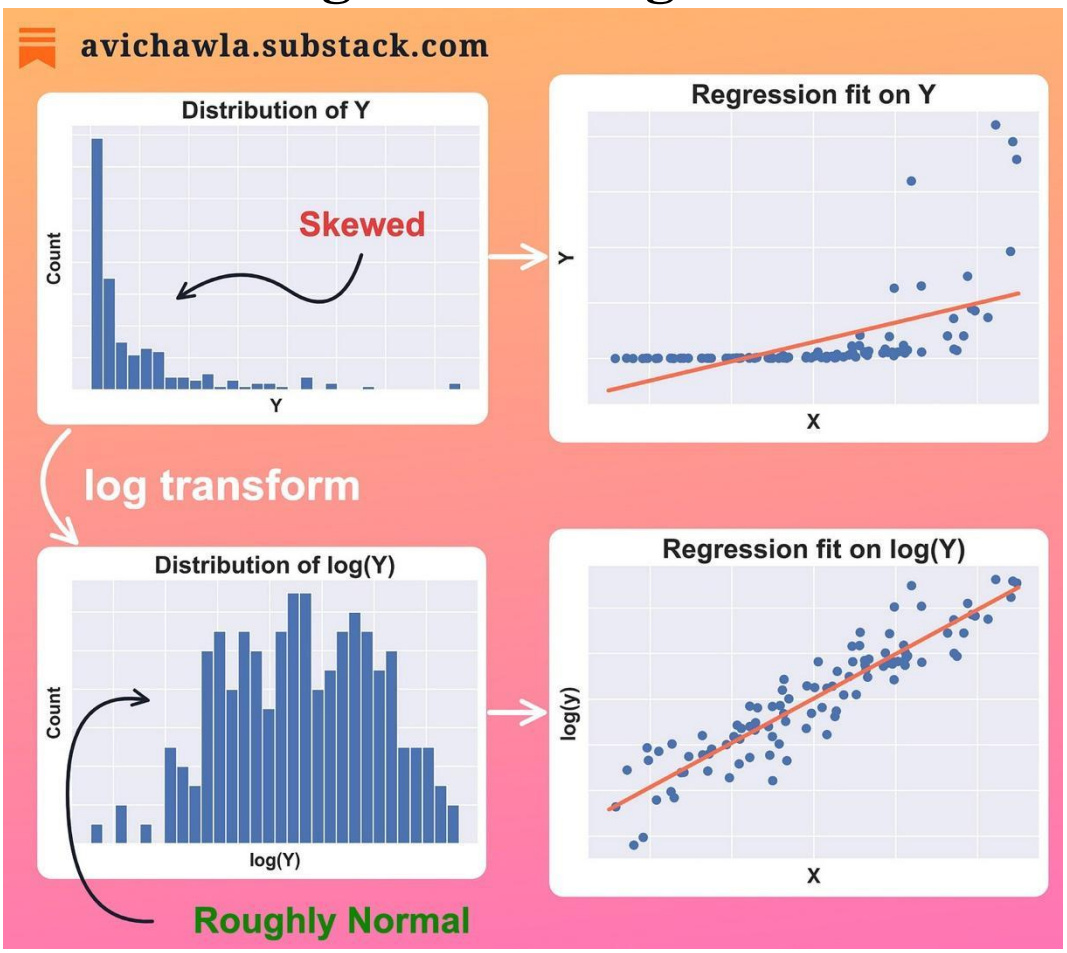
In sklearn's implementation, by default, a decision tree is allowed to grow until all leaves are pure. This leads to overfitting as the model attempts to classify every sample in the training set.

There are various techniques to avoid this, such as pruning and ensembling. Also, make sure that you tune hyperparameters if you use sklearn's implementation.

This was a gentle reminder as many of us often tend to use sklearn’s implementations in their default configuration.

It is always a good practice to know what a default implementation is hiding underneath.

Always Validate Your Output Variable Before Using Linear Regression



The effectiveness of a linear regression model largely depends on how well our data satisfies the algorithm's underlying assumptions.

Linear regression inherently assumes that the residuals (actual-prediction) follow a normal distribution. One way this assumption may get violated is when your output is skewed.

As a result, it will produce an incorrect regression fit.

But the good thing is that it can be corrected. One common way to make the output symmetric before fitting a model is to apply a log transform.

It removes the skewness by evenly spreading out the data, making it look somewhat normal.

One thing to note is that if the output has negative values, a log transform will raise an error. In such cases, one can apply translation transformation first on the output, followed by the log.

A Counterintuitive Fact About Python Functions

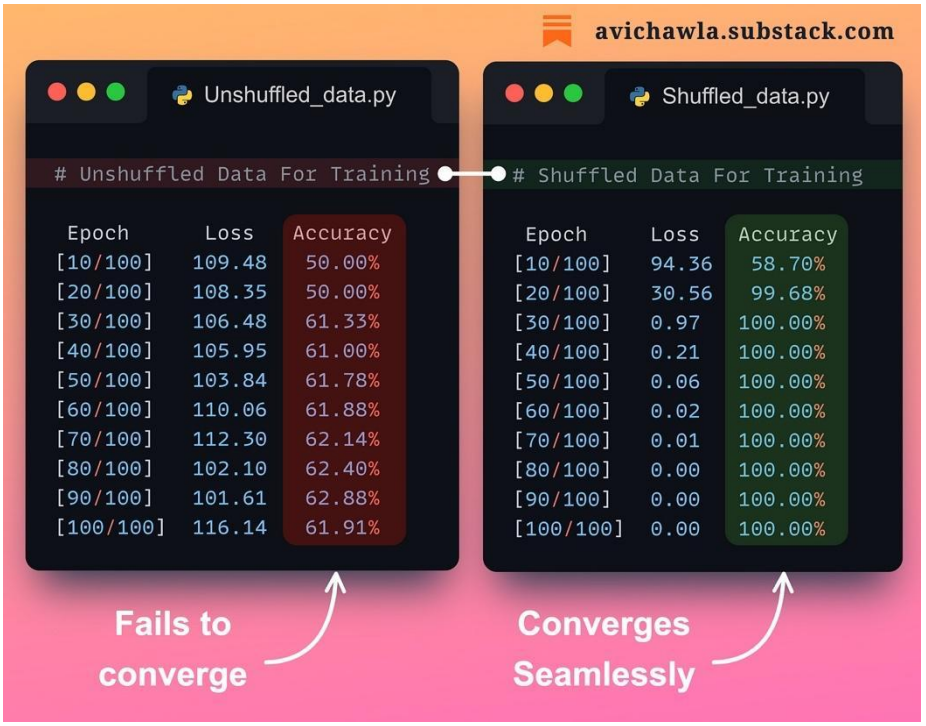
Everything in python is an object instantiated from some class. This also includes functions, but accepting this fact often feels counterintuitive at first.

Here are a few ways to verify that python functions are indeed objects.

The friction typically arises due to one's acquaintance with other programming languages like C++ and Java, which work very differently.

However, python is purely an object-oriented programming (OOP) language. You are always using OOP, probably without even realizing it.

Why Is It Important To Shuffle Your Dataset Before Training An ML Model



ML models may fail to converge for many reasons. Here's one of them which many folks often overlook.

If your data is ordered by labels, this could negatively impact the model's convergence and accuracy. This is a mistake that can typically go unnoticed.

In the above demonstration, I trained two neural nets on the same data. Both networks had the same initial weights, learning rate, and other settings.

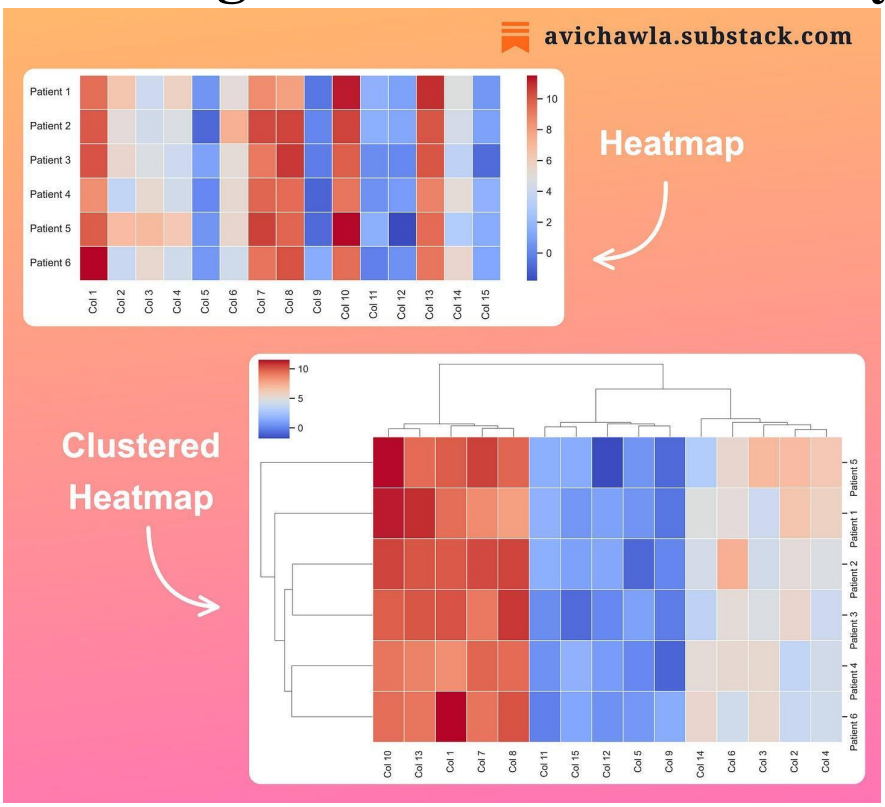
However, in one of them, the data was ordered by labels, while in another, it was randomly shuffled.

As shown, the model receiving a label-ordered dataset fails to converge. However, shuffling the dataset allows the network to learn from a more representative data sample in each batch. This leads to better generalization and performance.

In general, it's a good practice to shuffle the dataset before training. This prevents the model from identifying any label-specific yet non-existing patterns.

In fact, it is also recommended to alter batch-specific data in every epoch.

The Limitations Of Heatmap That Are Slowing Down Your Data Analysis



Heatmaps often make data analysis much easier. Yet, they do have some limitations.

A traditional heatmap does not group rows (and features). Instead, its orientation is the same as the input. This makes it difficult to visually determine the similarity between rows (and features).

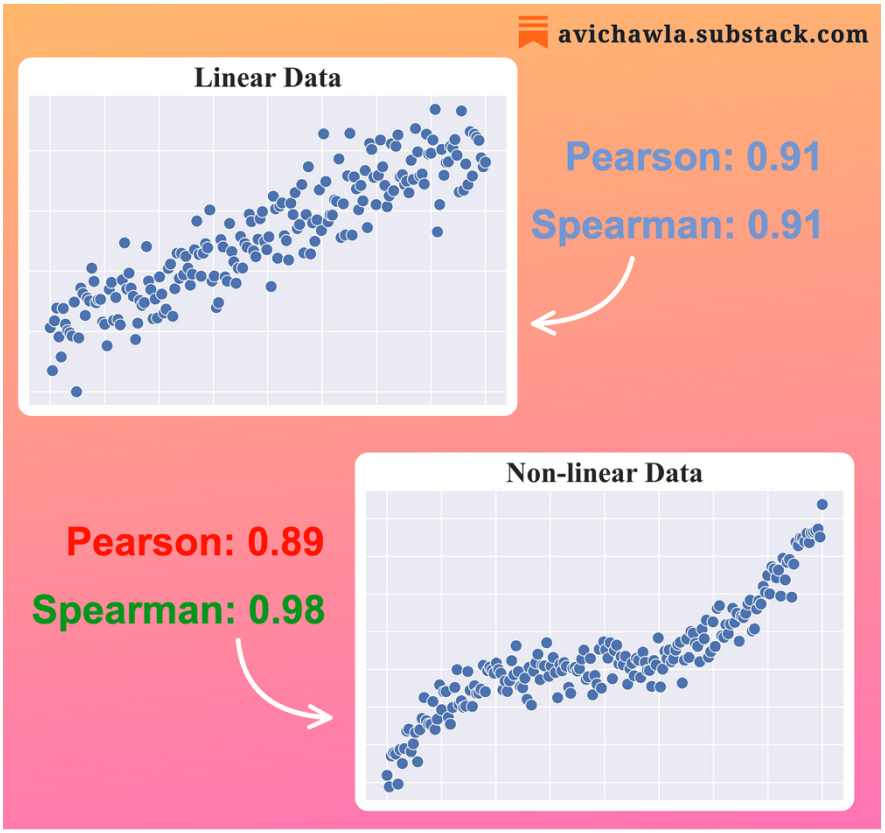
Clustered heatmaps can be a better choice in such cases. It clusters the rows and features together to help you make better sense of the data.

They can be especially useful when dealing with large datasets. While a traditional heatmap will be visually daunting to look at.

However, the groups in a clustered heatmap make it easier to visualize similarities and identify which rows (and features) go with one another.

To create a clustered heatmap, you can use the **sns.clustermap**() method from Seaborn. More info here: Seaborn docs.

The Limitation Of Pearson Correlation Which Many Often Ignore



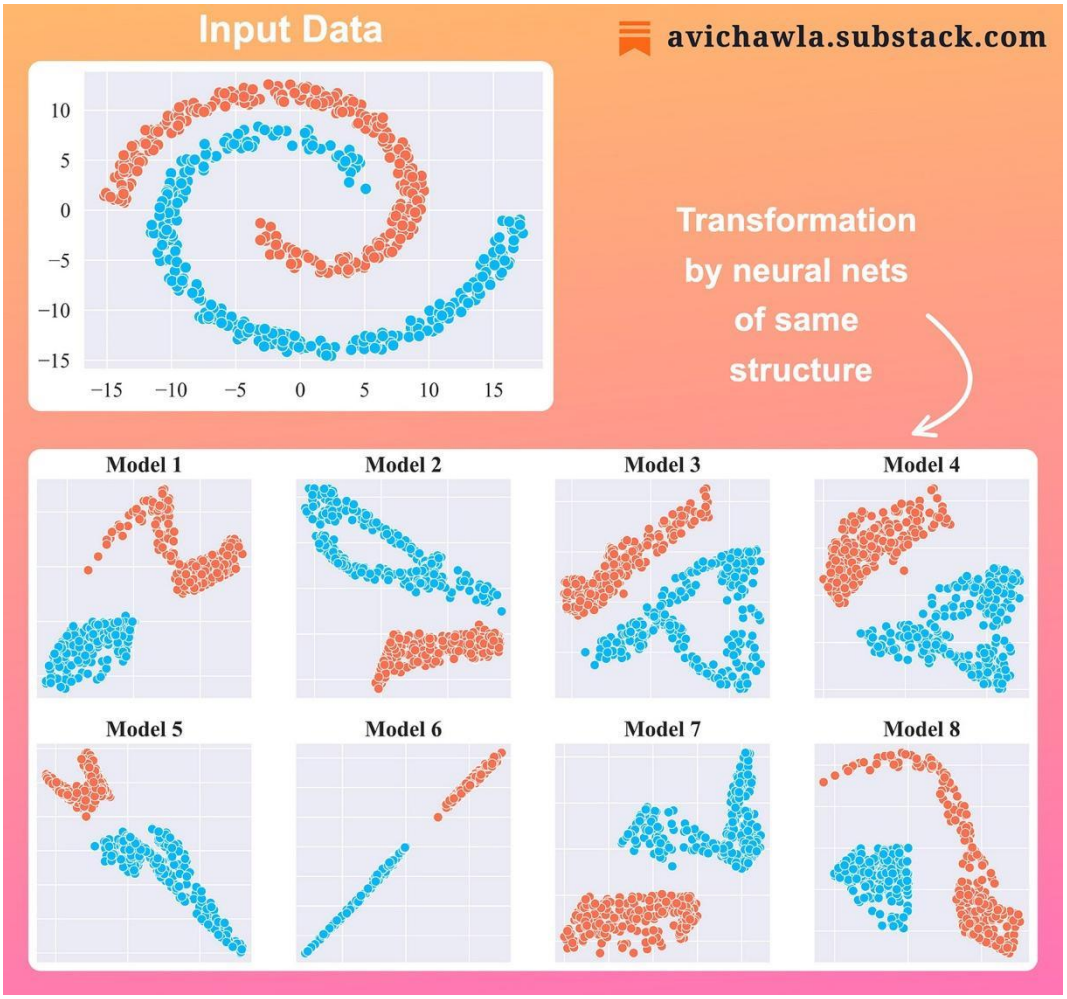
Pearson correlation is commonly used to determine the association between two continuous variables. But many often ignore its assumption.

Pearson correlation primarily measures the LINEAR relationship between two variables. As a result, even if two variables have a non-linear but monotonic relationship, Pearson will penalize that.

One great alternative is the Spearman correlation. It primarily assesses the monotonicity between two variables, which may be linear or non-linear.

What's more, Spearman correlation is also useful in situations when your data is ranked or ordinal.

Why Are We Typically Advised To Set Seeds for Random Generators?



From time to time, we advised to set seeds for random numbers before training an ML model. Here's why.

The weight initialization of a model is done randomly. Thus, any repeated experiment never generates the same set of numbers. This can hinder the reproducibility of your model.

As shown above, the same input data gets transformed in many ways by different neural networks of the same structure.

Thus, before training any model, always ensure that you set seeds so that your experiment is reproducible later.

An Underrated Technique To Improve Your Data Visualizations

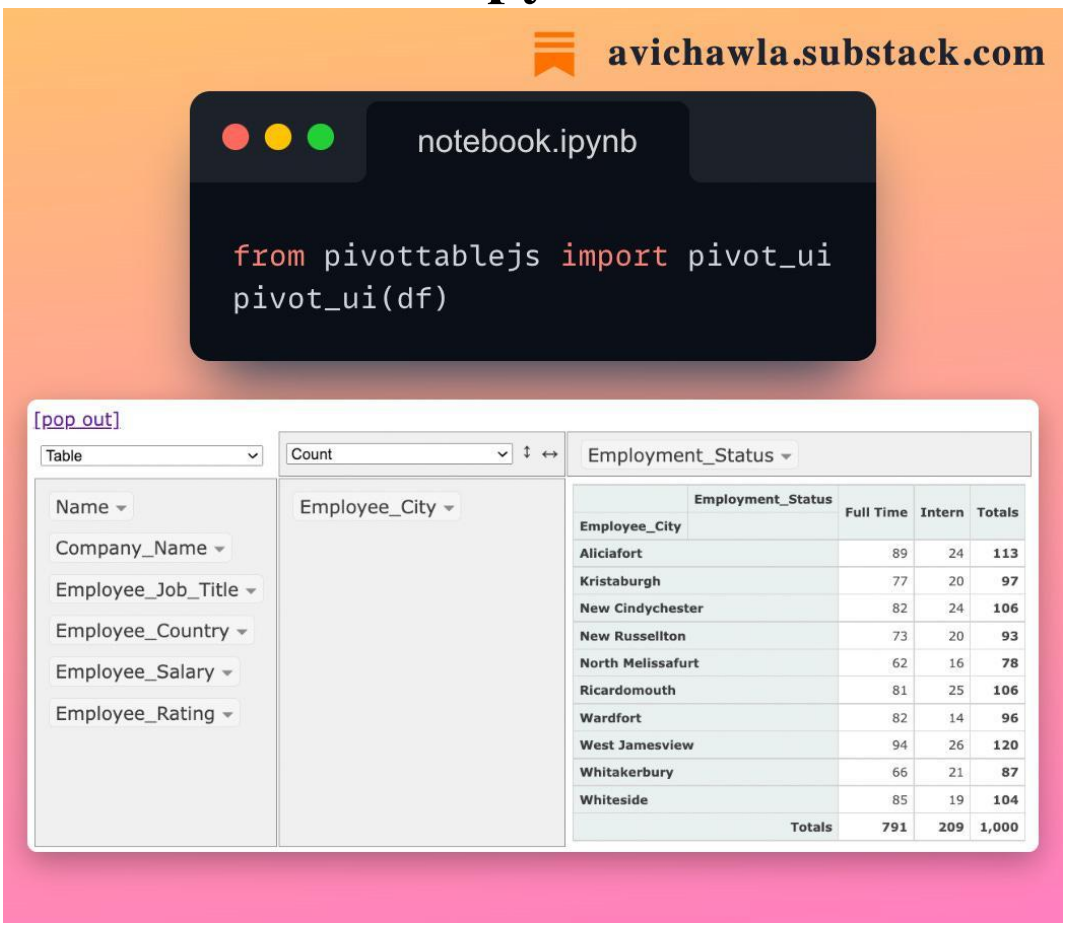
At times, ensuring that your plot conveys the right message may require you to provide additional context. Yet, augmenting extra plots may clutter your whole visualization.

One great way to provide extra info is by adding text annotations to a plot.

In matplotlib, you can use **annotate**(). It adds explanatory texts to your plot, which lets you guide a viewer's attention to specific areas and aid their understanding.

Find more info here: Matplotlib docs.

A No-Code Tool to Create Charts and Pivot Tables in Jupyter



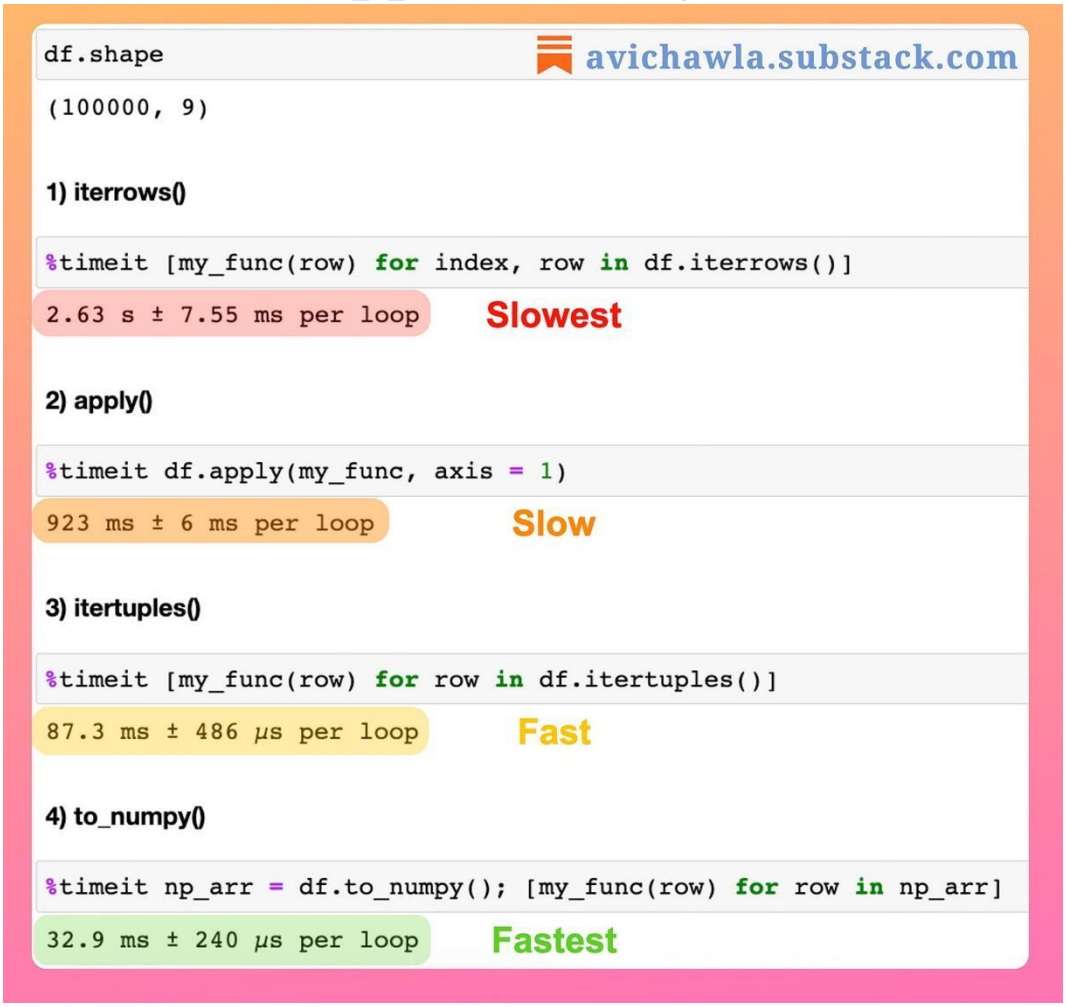
Here's a quick and easy way to create pivot tables, charts, and group data without writing any code.

PivotTableJS is a drag-n-drop tool for creating pivot tables and interactive charts in Jupyter. What's more, you can also augment pivot tables with heatmaps for enhanced analysis.

Find more info here: PivotTableJS.

Watch a video version of this post for enhanced understanding: Video.

If You Are Not Able To Code A Vectorized Approach, Try This.



Although we should never iterate over a dataframe and prefer vectorized code, what if we are not able to come up with a vectorized solution?

In my yesterday's post on why iterating a dataframe is costly, someone posed a pretty genuine question. They asked: "*Let's just say you are forced to iterate. What will be the best way to do so?*"

Firstly, understand that the primary reason behind the slowness of iteration is due to the way a dataframe is stored in memory. (If you wish to recap this, read yesterday's post here.)

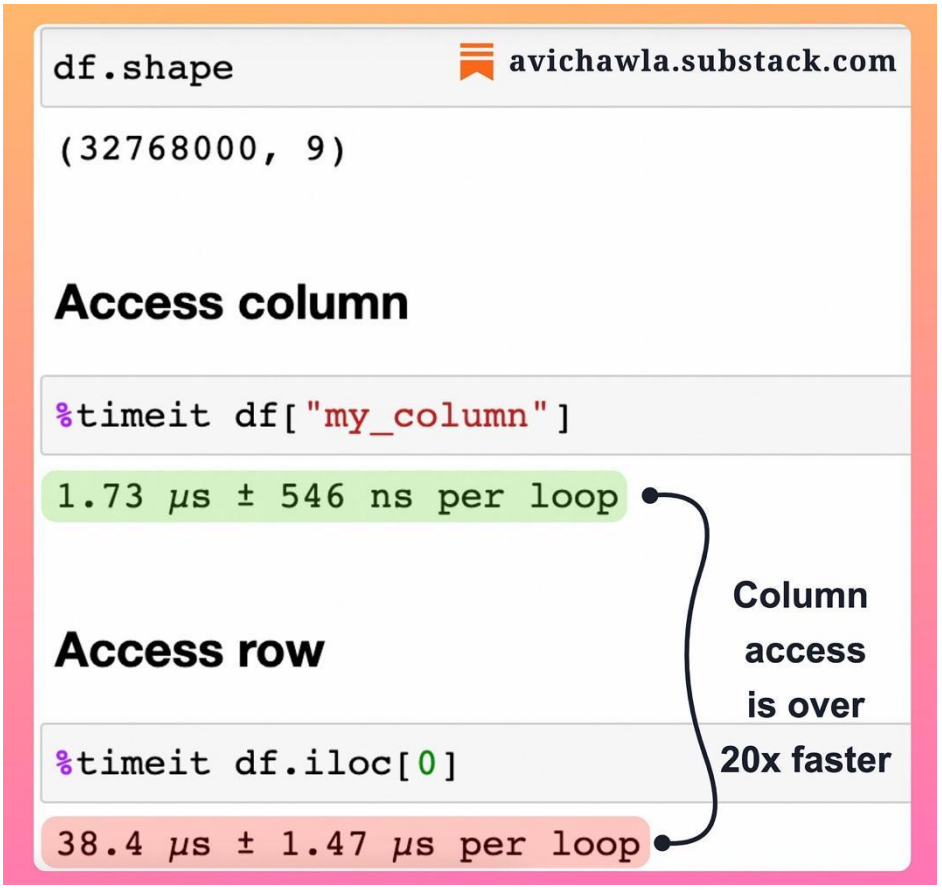
Being a column-major data structure, retrieving its rows requires accessing non-contiguous blocks of memory. This increases the run-time drastically.

Yet, if you wish to perform only row-based operations, a quick fix is to convert the dataframe to a NumPy array.

NumPy is faster here because, by default, it stores data in a row-major manner. Thus, its rows are retrieved by accessing contiguous blocks of memory, making it efficient over iterating a dataframe.

That being said, do note that the best way is to write vectorized code always. Use the Pandas-to-NumPy approach only when you are truly struggling with writing vectorized code.

Why Are We Typically Advised To Never Iterate Over A DataFrame?



From time to time, we are advised to avoid iterating on a Pandas DataFrame. But what is the exact reason behind this? Let me explain.

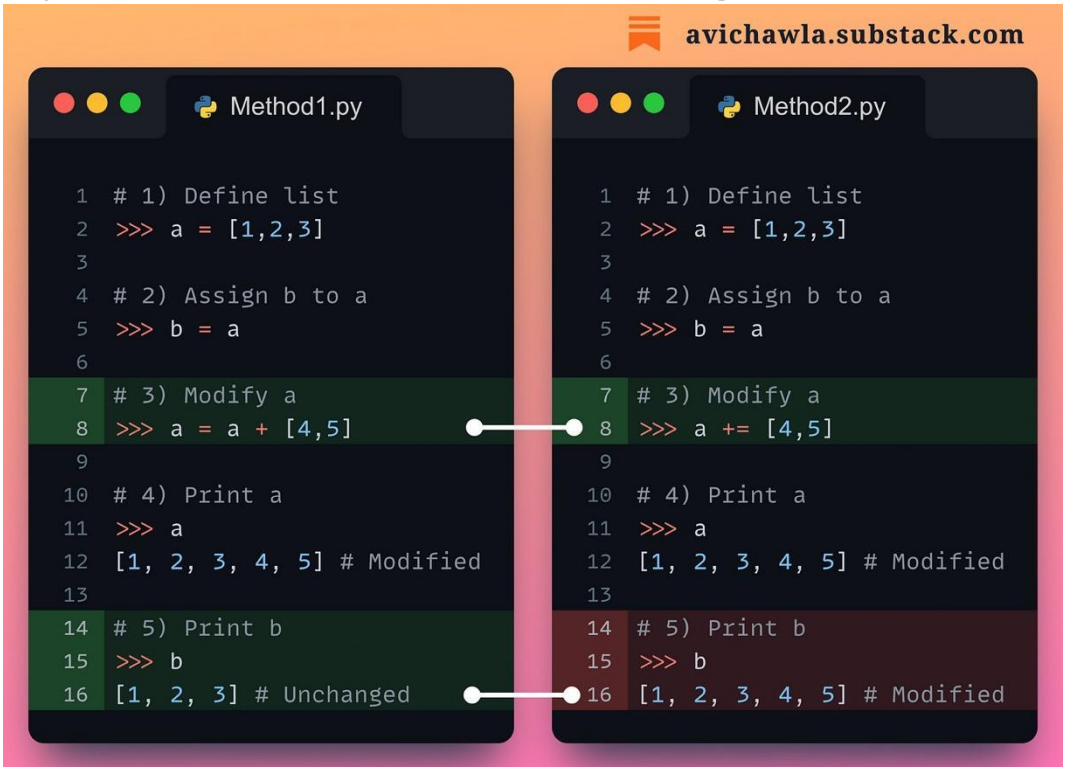
A DataFrame is a column-major data structure. Thus, consecutive elements in a column are stored next to each other in memory.

As processors are efficient with contiguous blocks of memory, retrieving a column is much faster than a row.

But while iterating, as each row is retrieved by accessing non-contiguous blocks of memory, the run-time increases drastically.

In the image above, retrieving over 32M elements of a column was still over **20x faster** than fetching just nine elements stored in a row.

Manipulating Mutable Objects In Python Can Get Confusing At Times



Did you know that with mutable objects, "**a** +=" and "**a** = **a** +" work differently in Python? Here's why.

Let's consider a list, for instance.

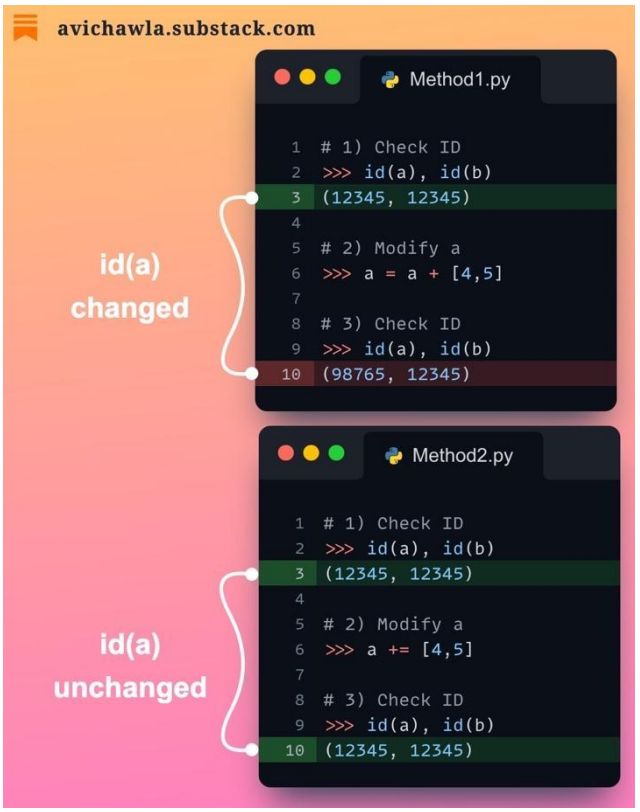
When we use the `=` operator, Python creates a new object in memory and assigns it to the variable.

Thus, all the other variables still reference the previous memory location, which was never updated. This is shown in `Method1.py` above.

But with the `+=` operator, changes are enforced in-place. This means that Python does not create a new object and the same memory location is updated.

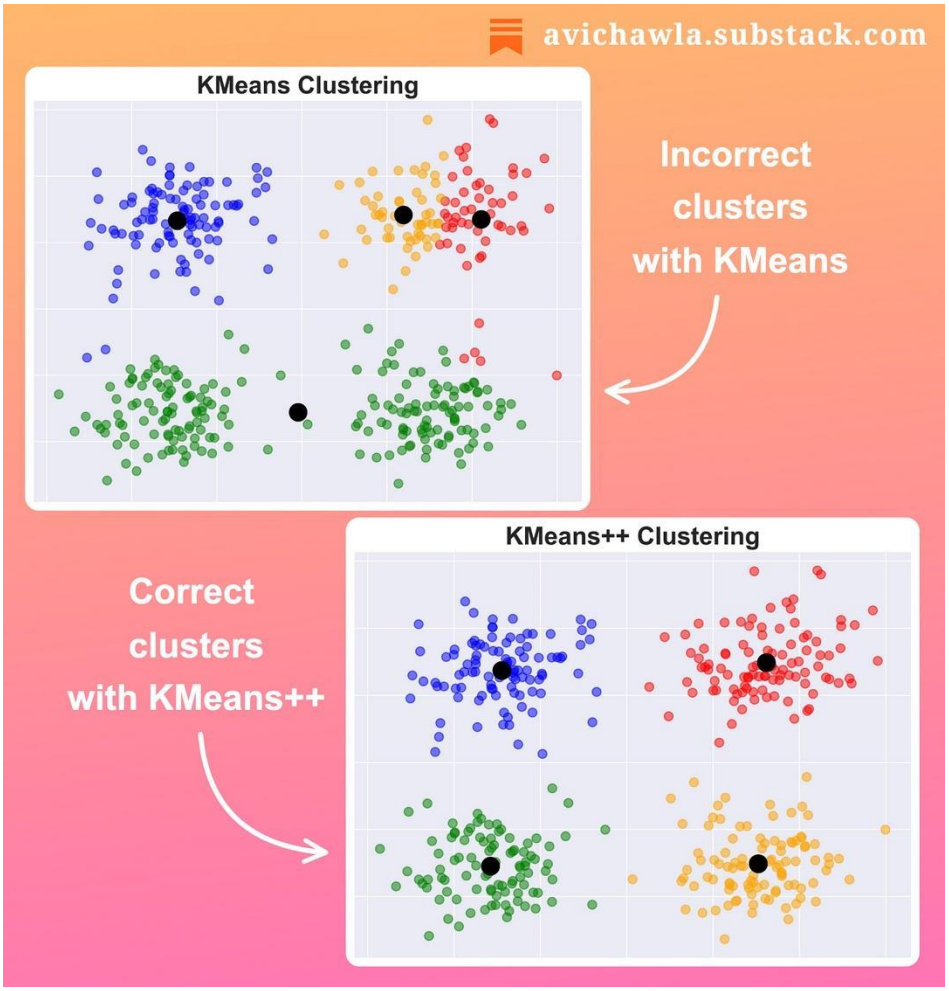
Thus, changes are visible through all other variables that reference the same location. This is shown in `Method2.py` above.

We can also verify this by comparing the **id**() pre-assignment and post-assignment.



With "**a** = **a** +", the **id** gets changed, indicating that Python created a new object. However, with "**a** +=", **id** stays the same. This indicates that the same memory location was updated.

This Small Tweak Can Significantly Boost The Run-time of KMeans



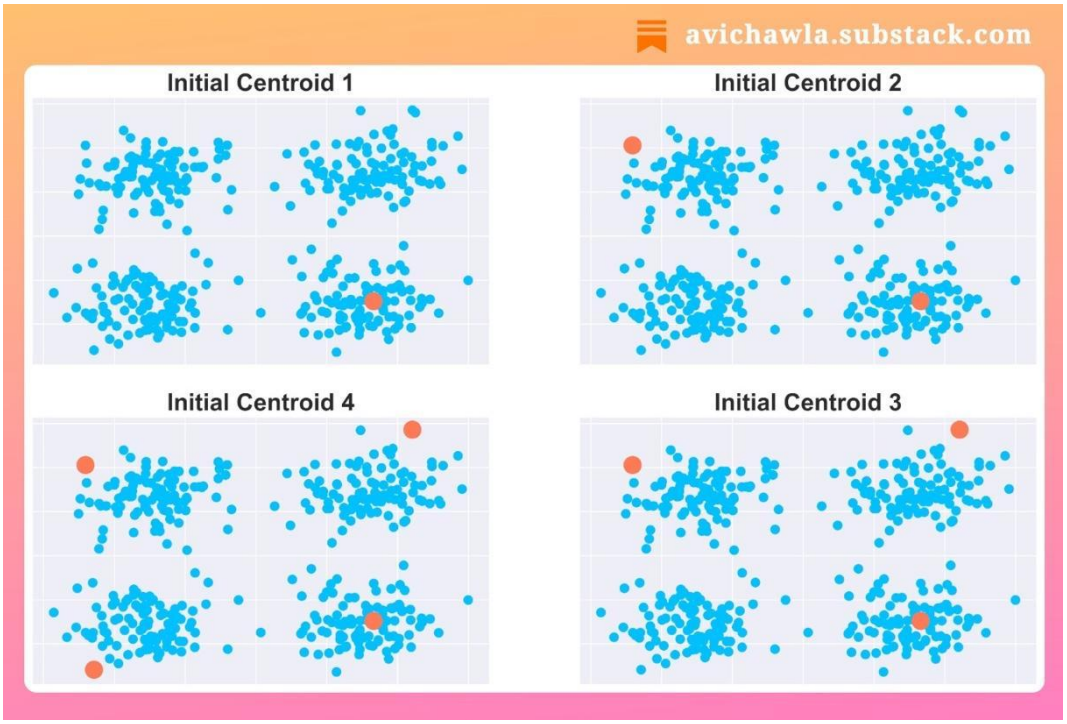
KMeans is a popular but high-run-time clustering algorithm. Here's how a small tweak can significantly improve its run time.

KMeans selects the initial centroids randomly. As a result, it fails to converge at times. This requires us to repeat clustering several times with different initialization.

Instead, KMeans++ takes a smarter approach to initialize centroids. The first centroid is selected randomly. But the next centroid is chosen based on the distance from the first centroid.

In other words, a point that is away from the first centroid is more likely to be selected as an initial centroid. This way, all the initial centroids are likely to lie in different clusters already and the algorithm may converge faster.

The illustration below shows the centroid initialization of KMeans++:



Most Python Programmers Don't Know This About Python OOP

Most python programmers misunderstand the **__init__**() method. They think that it creates a new object. But that is not true.

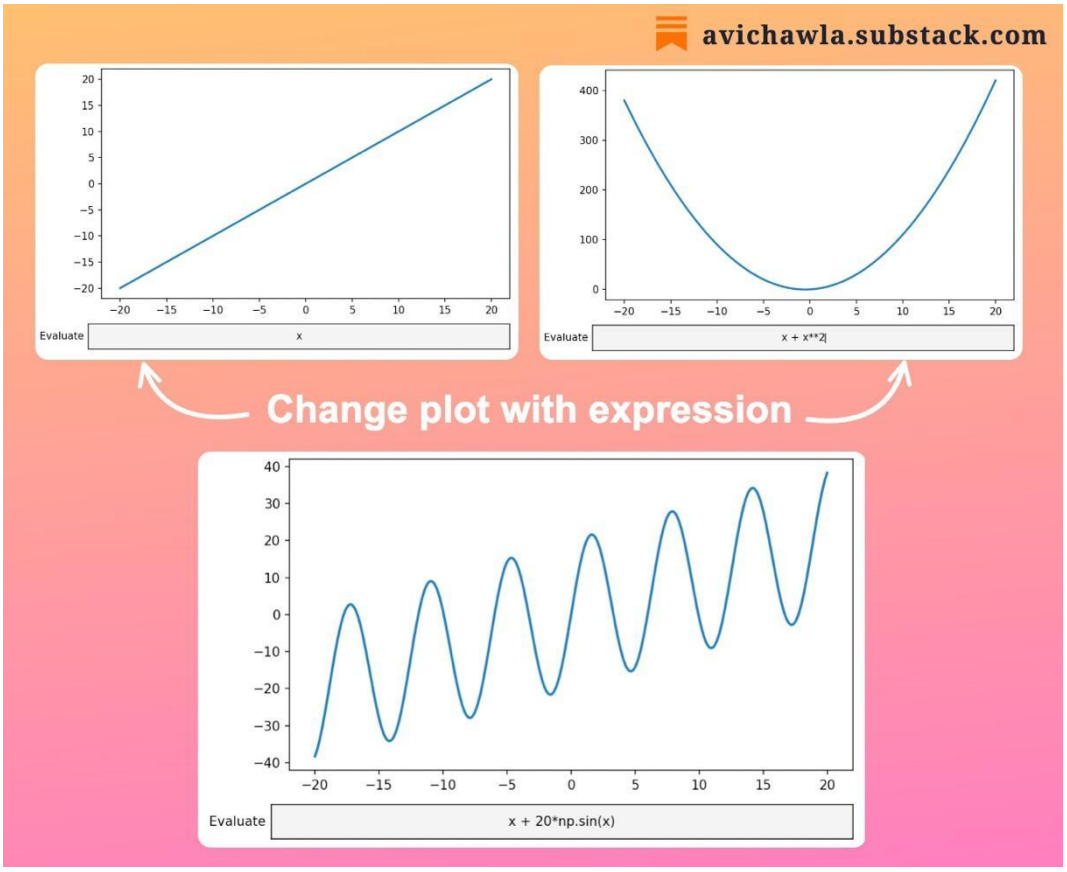
When we create an object, it is not the **__init__**() method that allocates memory to it. As the name suggests, **__init__**() only assigns value to an object's attributes.

Instead, Python invokes the **__new__**() method first to create a new object and allocate memory to it. But how is that useful, you may wonder? There are many reasons.

For instance, by implementing the **__new__**() method, you can apply data checks. This ensures that your program allocates memory only when certain conditions are met.

Other common use cases involve defining singleton classes (classes with only one object), creating subclasses of immutable classes such as tuples, etc.

Who Said Matplotlib Cannot Create Interactive Plots?



👉 Please watch a video version of this post for better understanding: Video Link.

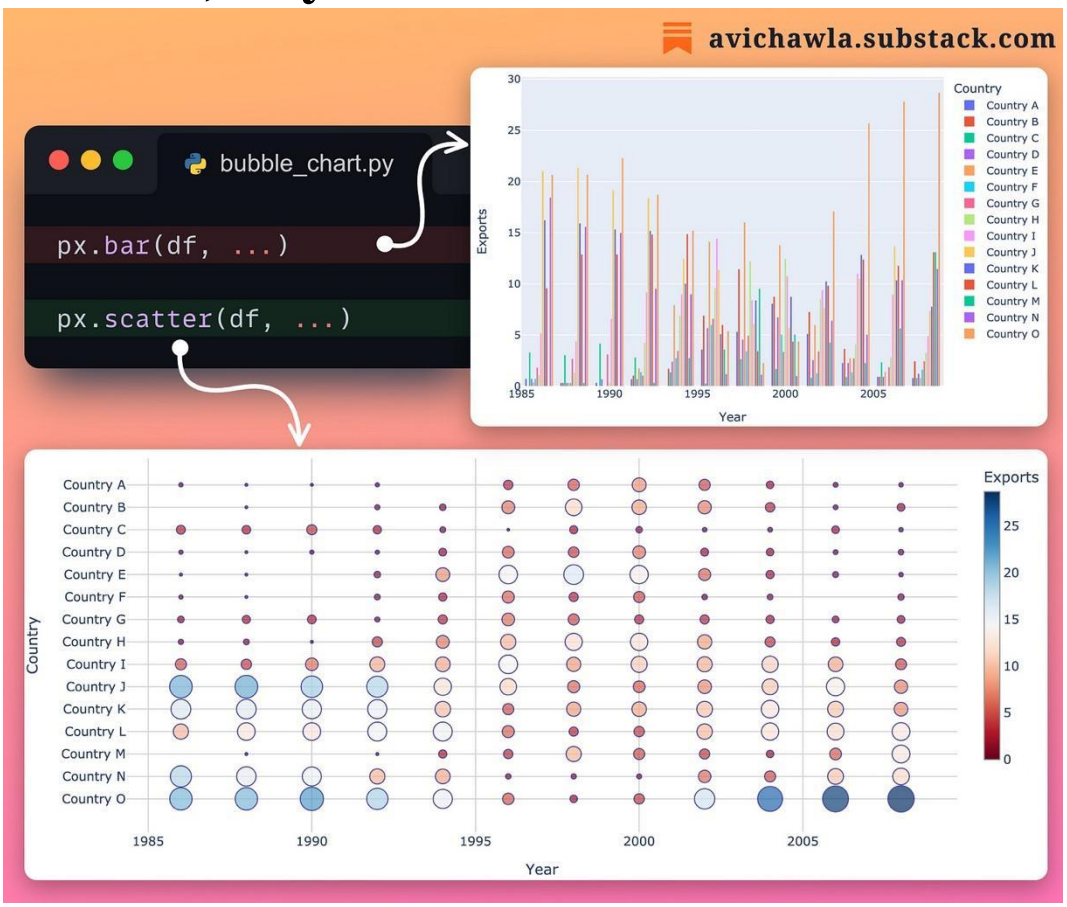
In most cases, Matplotlib is used to create static plots. But very few know that it can create interactive plots too. Here's how.

By default, Matplotlib uses the **inline** mode, which renders static plots. However, with the **%matplotlib widget** magic command, you can enable interactive backend for Matplotlib plots.

What's more, its **widgets** module offers many useful widgets. You can integrate them with your plots to make them more elegant.

Find a detailed guide here: Matplotlib widgets.

Don't Create Messy Bar Plots. Instead, Try Bubble Charts!



Bar plots often get incomprehensible and messy when we have many categories to plot.

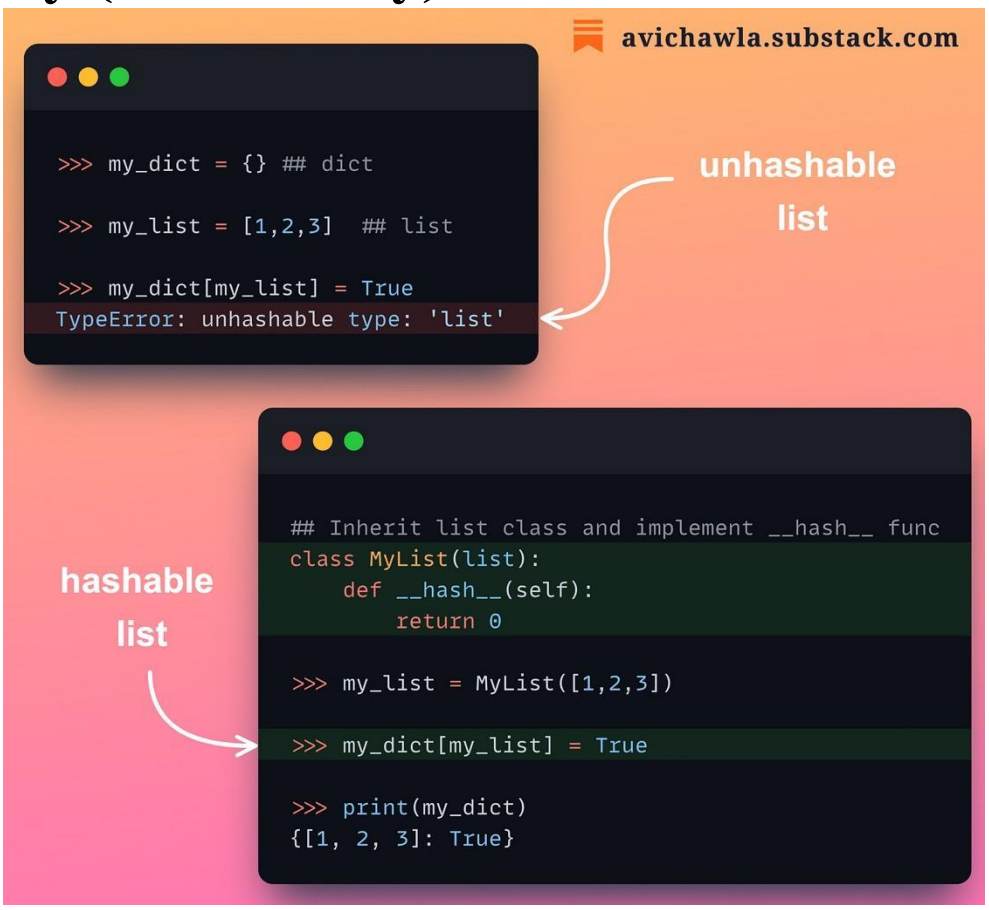
A bubble chart can be a better choice in such cases. They are like scatter plots but with one categorical and one continuous axis.

Compared to a bar plot, they are less cluttered and offer better comprehension.

Of course, the choice of plot ultimately depends on the nature of the data and the specific insights you wish to convey.

Which plot do you typically prefer in such situations?

You Can Add a List As a Dictionary's Key (Technically)!



Python raises an error whenever we add a list as a dictionary's key. But do you know the technical reason behind it? Here you go.

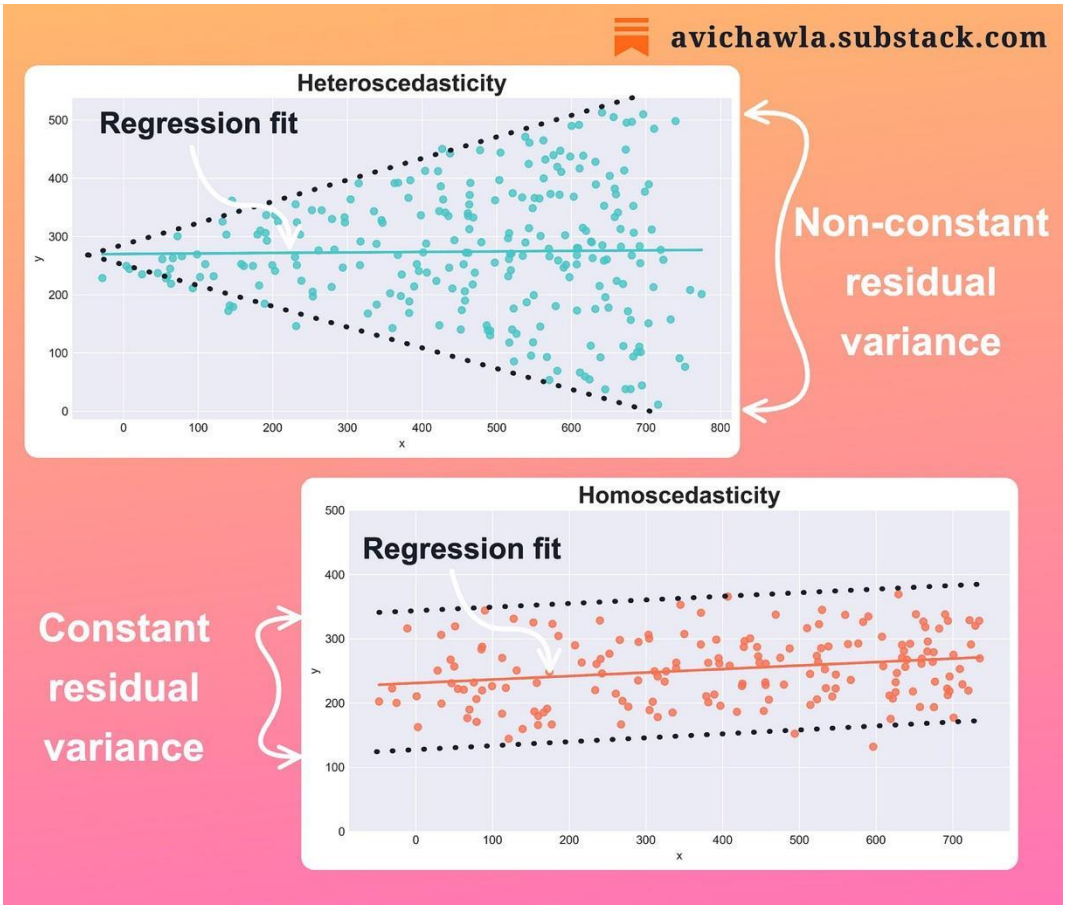
Firstly, understand that everything in Python is an object instantiated from some class. Whenever we add an object as a dict's key, Python invokes the **__hash__** function of that object's class.

While classes of int, str, tuple, frozenset, etc. implement the **__hash__** method, it is missing from the list class. That is why we cannot add a list as a dictionary's key.

Thus, technically if we extend the list class and add this method, a list can be added as a dictionary's key.

While this makes a list hashable, it isn't recommended as it can lead to unexpected behavior in your code.

Most ML Folks Often Neglect This While Using Linear Regression



The effectiveness of a linear regression model is determined by how well the data conforms to the algorithm's underlying assumptions.

One highly important, yet often neglected assumption of linear regression is homoscedasticity.

A dataset is homoscedastic if the variability of residuals (=actual-predicted) stays the same across the input range.

In contrast, a dataset is heteroscedastic if the residuals have non-constant variance.

Homoscedasticity is extremely critical for linear regression. This is because it ensures that our regression coefficients are reliable. Moreover, we can trust that the predictions will always stay within the same confidence interval.

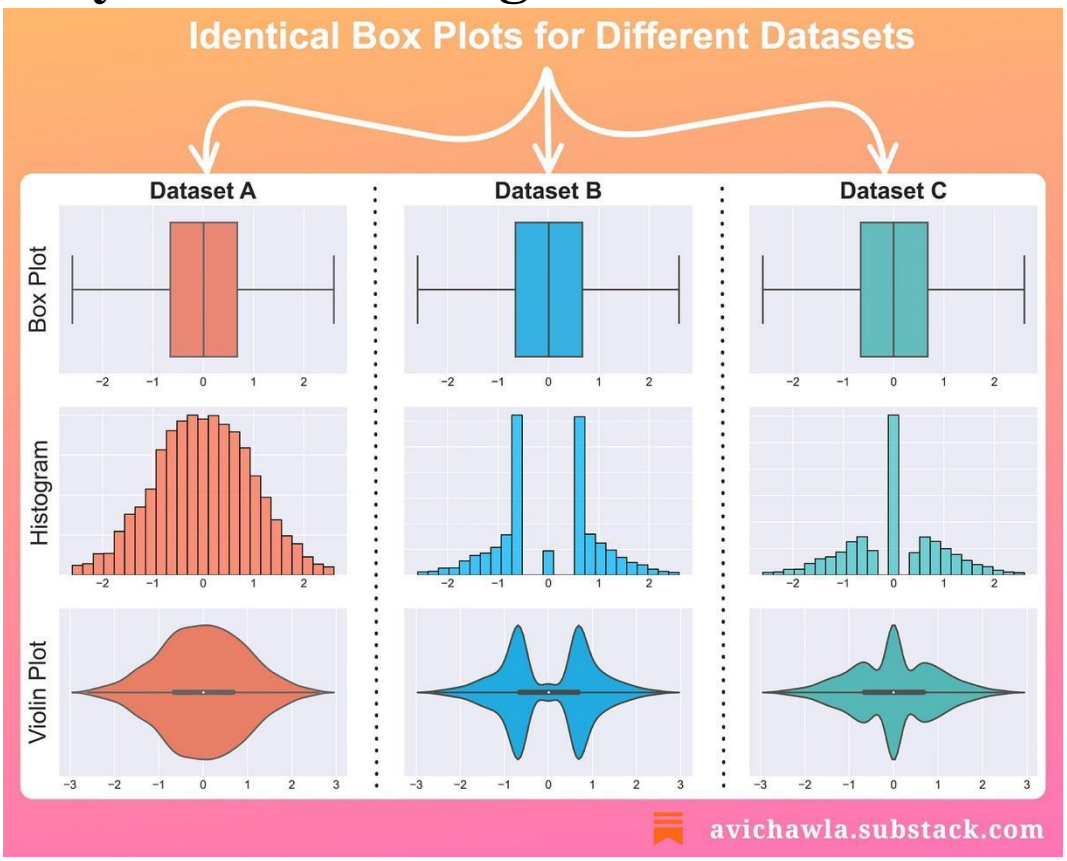
35 Hidden Python Libraries That Are Absolute Gems

I reviewed 1,000+ Python libraries and discovered these hidden gems I never knew even existed.

Here are some of them that will make you fall in love with Python and its versatility (even more).

Read this full list here:
https://avichawla.substack.com/p/35-gem-py-libs.

Use Box Plots With Caution! They May Be Misleading.



Box plots are quite common in data analysis. But they can be misleading at times. Here's why.

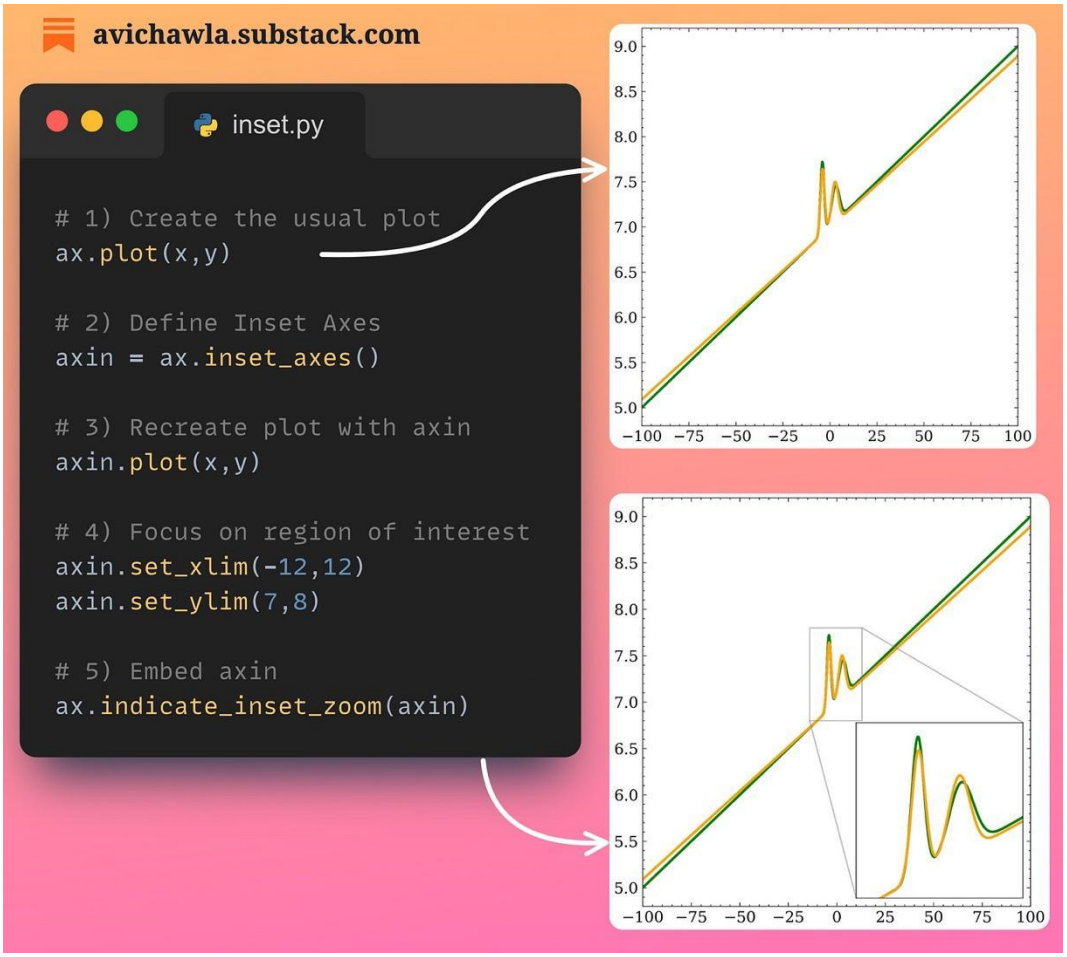
A box plot is a graphical representation of just five numbers – min, first quartile, median, third quartile, and max.

Thus, two different datasets with similar five values will produce identical box plots. This, at times, can be misleading and one may draw wrong conclusions.

The takeaway is NOT that box plots should not be used. Instead, look at the underlying distribution too. Here, histograms and violin plots can help.

Lastly, always remember that when you condense a dataset, you don't see the whole picture. You are losing essential information.

An Underrated Technique To Create Better Data Plots



While creating visualizations, there are often certain parts that are particularly important. Yet, they may not be immediately obvious to the viewer.

A good data storyteller will always ensure that the plot guides the viewer's attention to these key areas.

One great way is to zoom in on specific regions of interest in a plot. This ensures that our plot indeed communicates what we intend it to depict.

In matplotlib, you can do so using **indicate_inset_zoom**(). It adds an indicator box, that can be zoomed-in for better communication.

Find more info here: Matplotlib docs.

The Pandas DataFrame Extension Every Data Scientist Has Been Waiting For

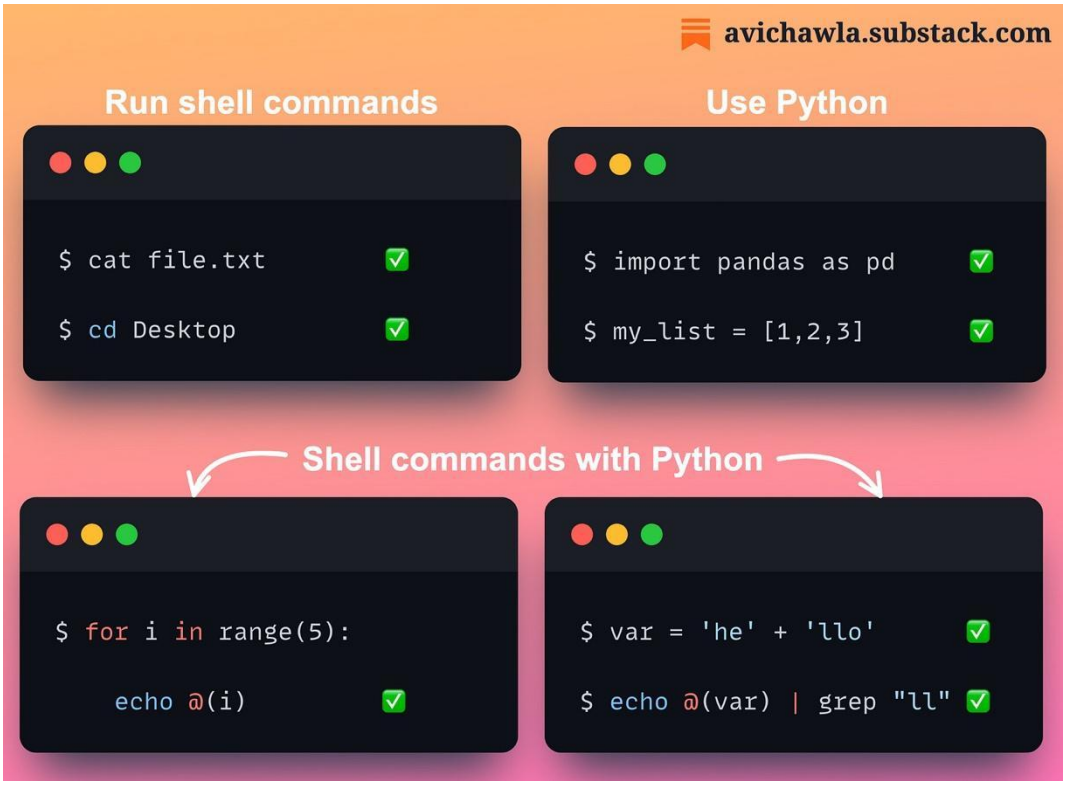
Watch a video version of this post for better understanding: Video Link.

PyGWalker is an open-source alternative to Tableau that transforms pandas dataframe into a tableau-style user interface for data exploration.

It provides a tableau-like UI in Jupyter, allowing you to analyze data faster and without code.

Find more info here: PyGWalker.

Supercharge Shell With Python Using Xonsh



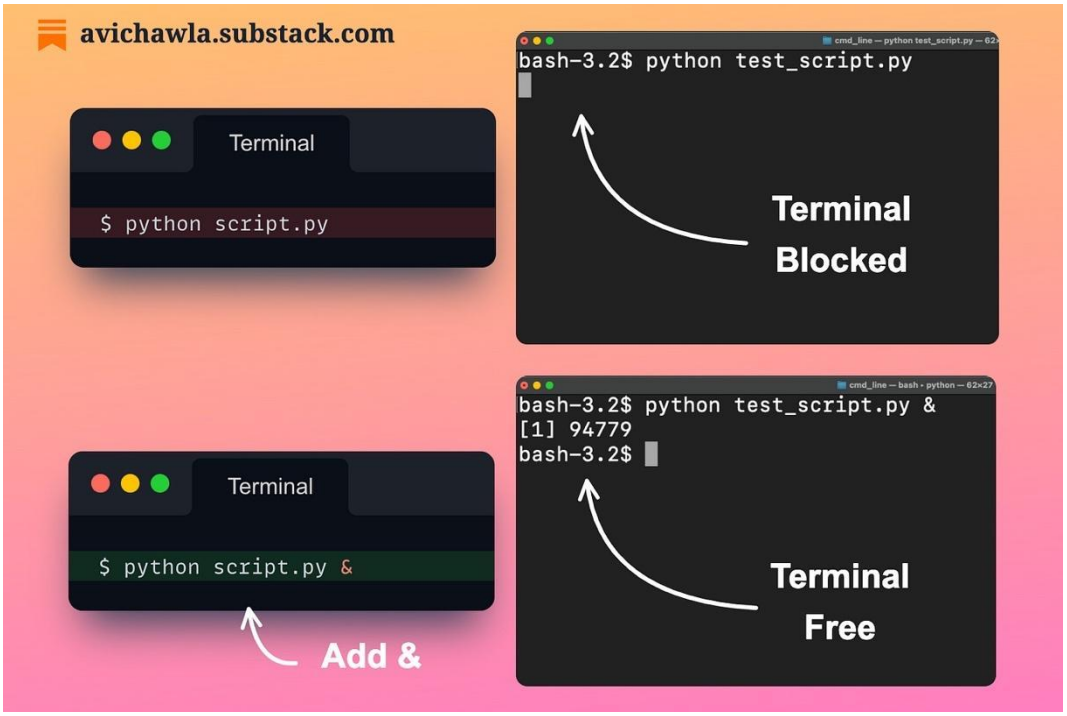
Traditional shells have a limitation for python users. At a time, users can either run shell commands or use IPython.

As a result, one has to open multiple terminals or switch back and forth between them in the same terminal.

Instead, try Xonsh. It combines the convenience of a traditional shell with the power of Python. Thus, you can use Python syntax as well as run shell commands in the same shell.

Find more info here: Xonsh.

Most Command-line Users Don't Know This Cool Trick About Using Terminals



Watch a video version of this post for better understanding: Video Link.

After running a command (or script, etc.), most command-line users open a new terminal to run other commands. But that is never required.

Here's how.

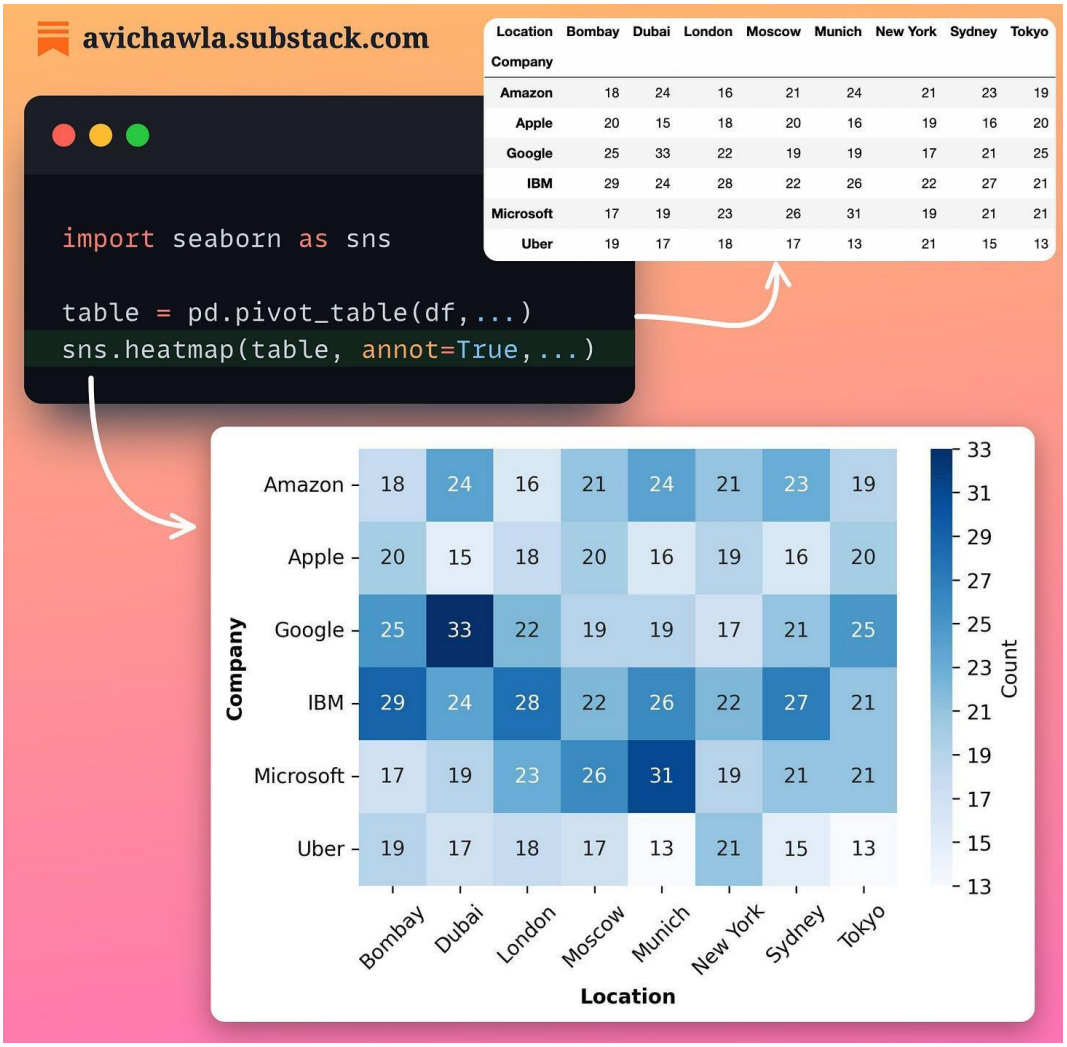
When we run a program from the command line, by default, it runs in the foreground. This means you can't use the terminal until the program has been completed.

However, if you add '&' at the end of the command, the program will run in the background and instantly free the terminal.

This way, you can use the same terminal to run another command.

To bring the program back to the foreground, use the 'fg' command.

A Simple Trick to Make The Most Out of Pivot Tables in Pandas

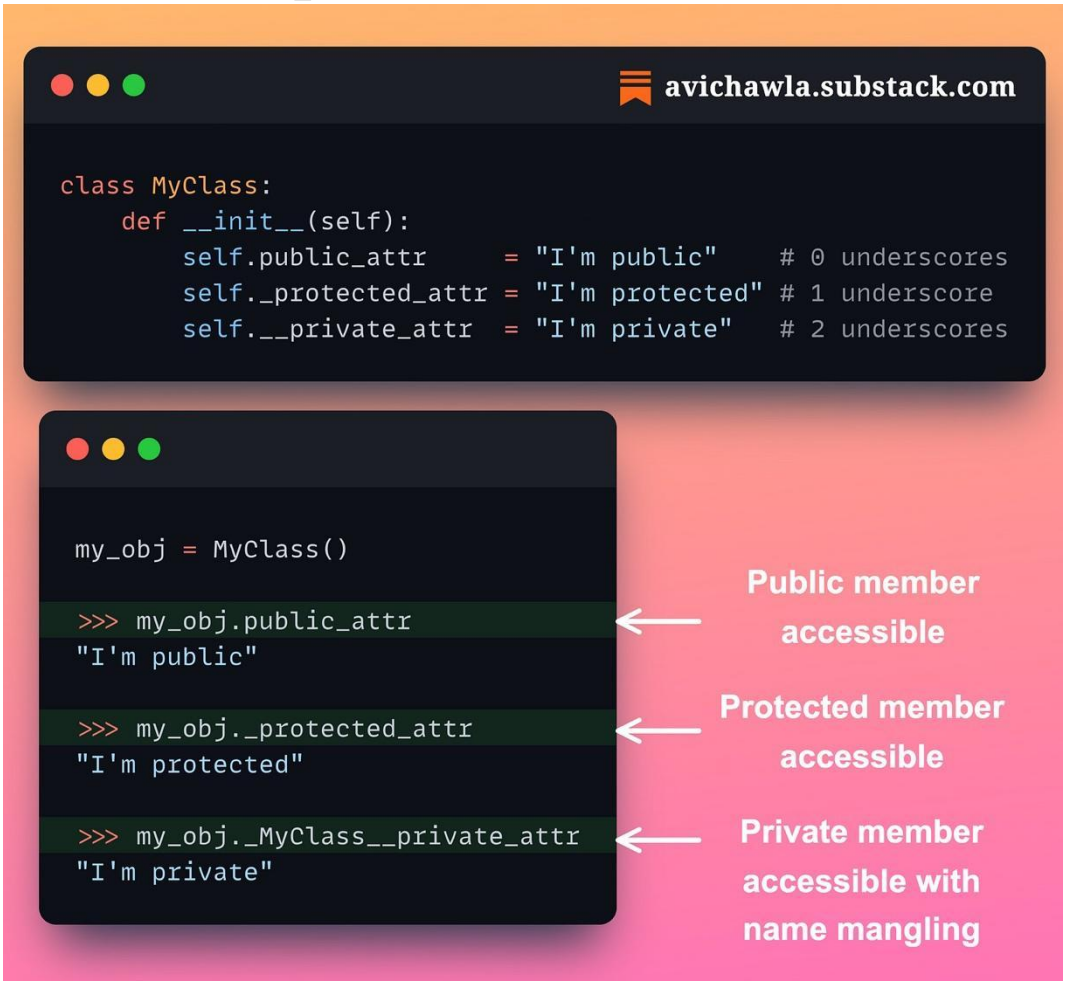


Company	Bombay	Dubai	London	Moscow	Munich	New York	Sydney	Tokyo
Amazon	18	24	16	21	24	21	23	19
Apple	20	15	18	20	16	19	16	20
Google	25	33	22	19	19	17	21	25
IBM	29	24	28	22	26	22	27	21
Microsoft	17	19	23	26	31	19	21	21
Uber	19	17	18	17	13	21	15	13

Pivot tables are pretty common for data exploration. Yet, analyzing raw figures is tedious and challenging. What's more, one may miss out on some crucial insights about the data.

Instead, enrich your pivot tables with heatmaps. The color encodings make it easier to analyze the data and determine patterns.

Why Python Does Not Offer True OOP Encapsulation



Using access modifiers (public, protected, and private) is fundamental to encapsulation in OOP. Yet, Python, in some way, fails to deliver true encapsulation.

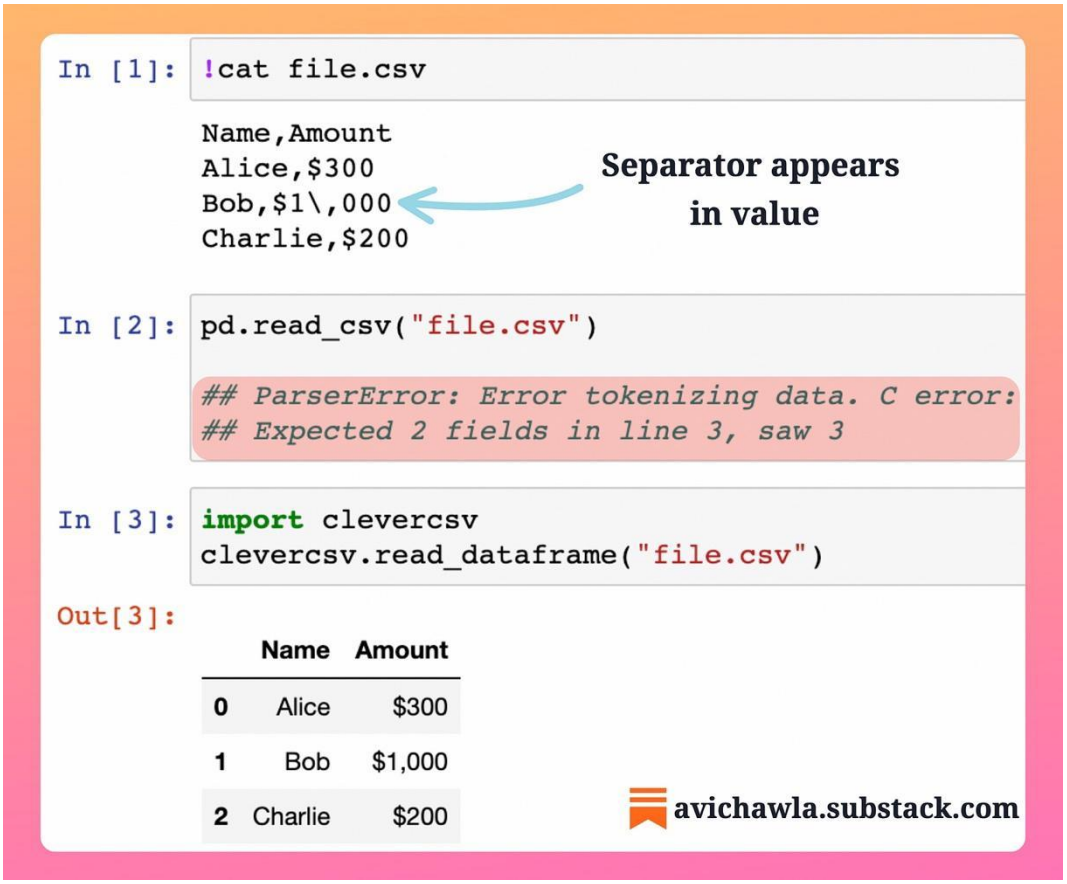
By definition, a public member is accessible everywhere. A private member can only be accessed inside the base class. A protected member is accessible inside the base class and child class(es).

But, with Python, there are no such strict enforcements.

Thus, protected members behave exactly like public members. What's more, private members can be accessed outside the class using name mangling.

As a programmer, remember that encapsulation in Python mainly relies on conventions. Thus, it is the responsibility of the programmer to follow them.

Never Worry About Parsing Errors Again While Reading CSV with Pandas



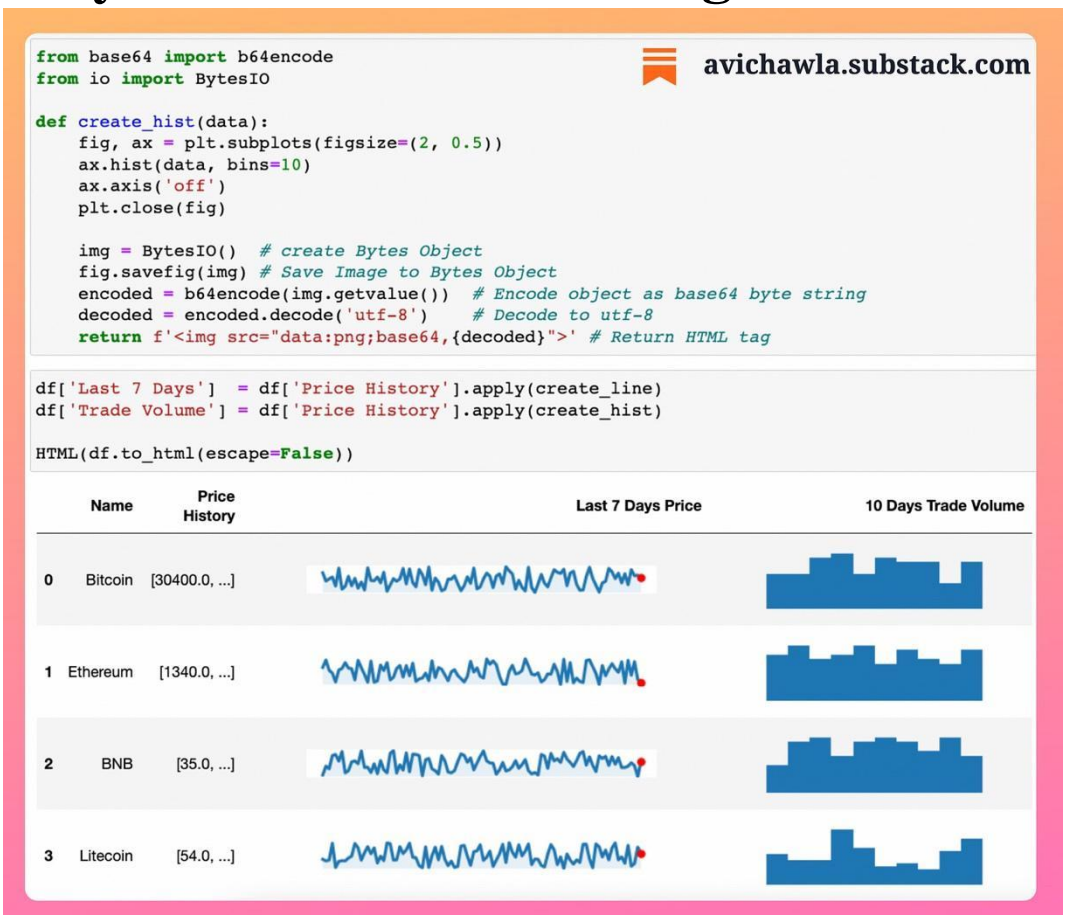
Pandas isn't smart (yet) to read messy CSV files.

Its read_csv method assumes the data source to be in a standard tabular format. Thus, any irregularity in data raises parsing errors, which may require manual intervention.

Instead, try CleverCSV. It detects the format of CSVs and makes it easier to load them, saving you tons of time.

Find more info here: CleverCSV.

An Interesting and Lesser-Known Way To Create Plots Using Pandas



Whenever you print/display a DataFrame in Jupyter, it is rendered using HTML and CSS. This allows us to format the output just like any other web page.

One interesting way is to embed inline plots which appear as a column of a dataframe.

In the above snippet, we first create a plot as we usually do. Next, we return the <img> HTML tag with its source as the plot. Lastly, we render the dataframe as HTML.

Find the code for this tip here: Notebook.

Most Python Programmers Don't Know This About Python For-loops

Often when we use a for-loop in Python, we tend not to modify the loop variable inside the loop.

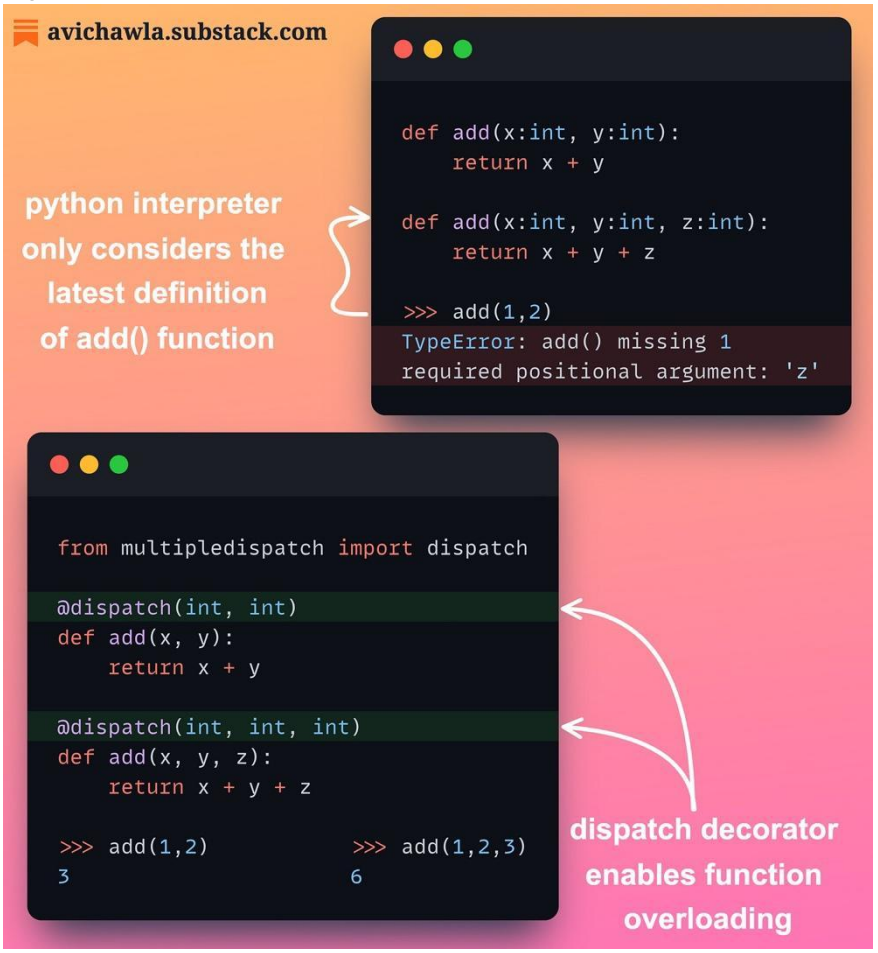
The impulse typically comes from acquaintance with other programming languages like C++ and Java.

But for-loops don't work that way in Python. Modifying the loop variable has no effect on the iteration.

This is because, before every iteration, Python unpacks the next item provided by iterable (**range**(5)) and assigns it to the loop variable (**num**).

Thus, any changes to the loop variable are replaced by the new value coming from the iterable.

How To Enable Function Overloading In Python



Python has no native support for function overloading. Yet, there's a quick solution to it.

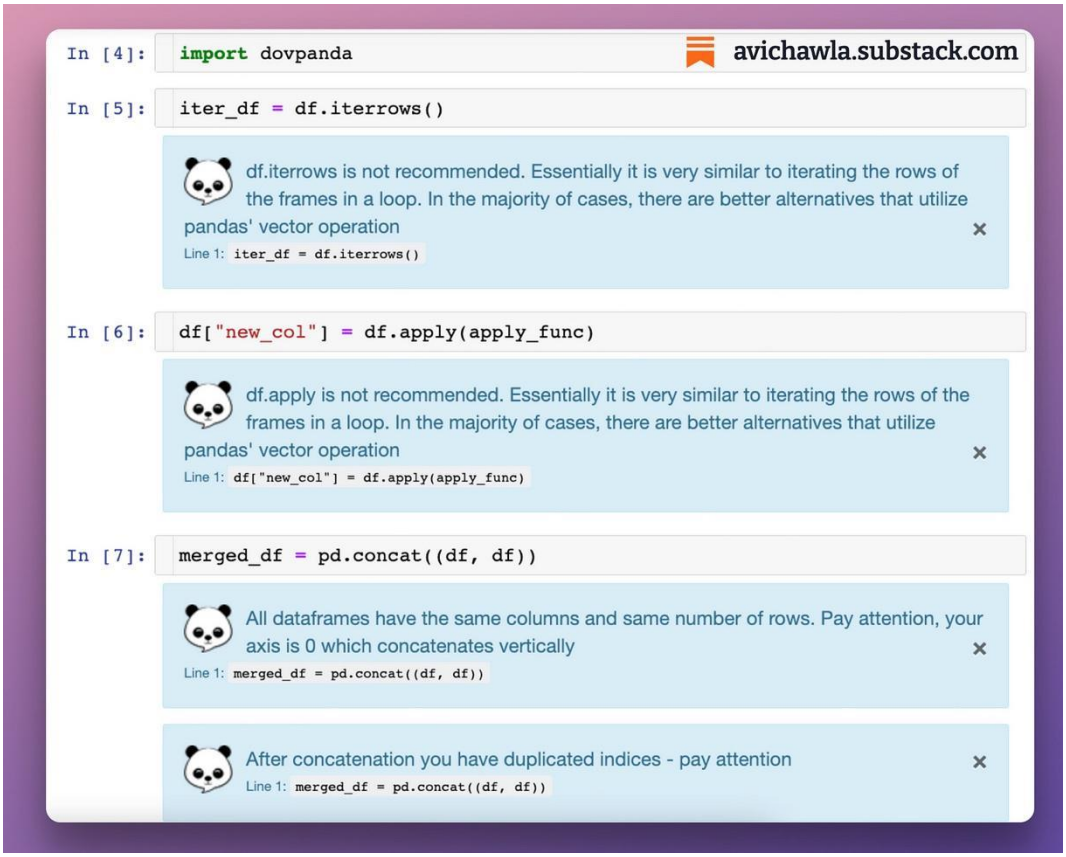
Function overloading (having multiple functions with the same name but different number/type of parameters) is one of the core ideas behind polymorphism in OOP.

But if you have many functions with the same name, python only considers the latest definition. This restricts writing polymorphic code.

Despite this limitation, the dispatch decorator allows you to leverage function overloading.

Find more info here: Multipledispatch.

Generate Helpful Hints As You Write Your Pandas Code



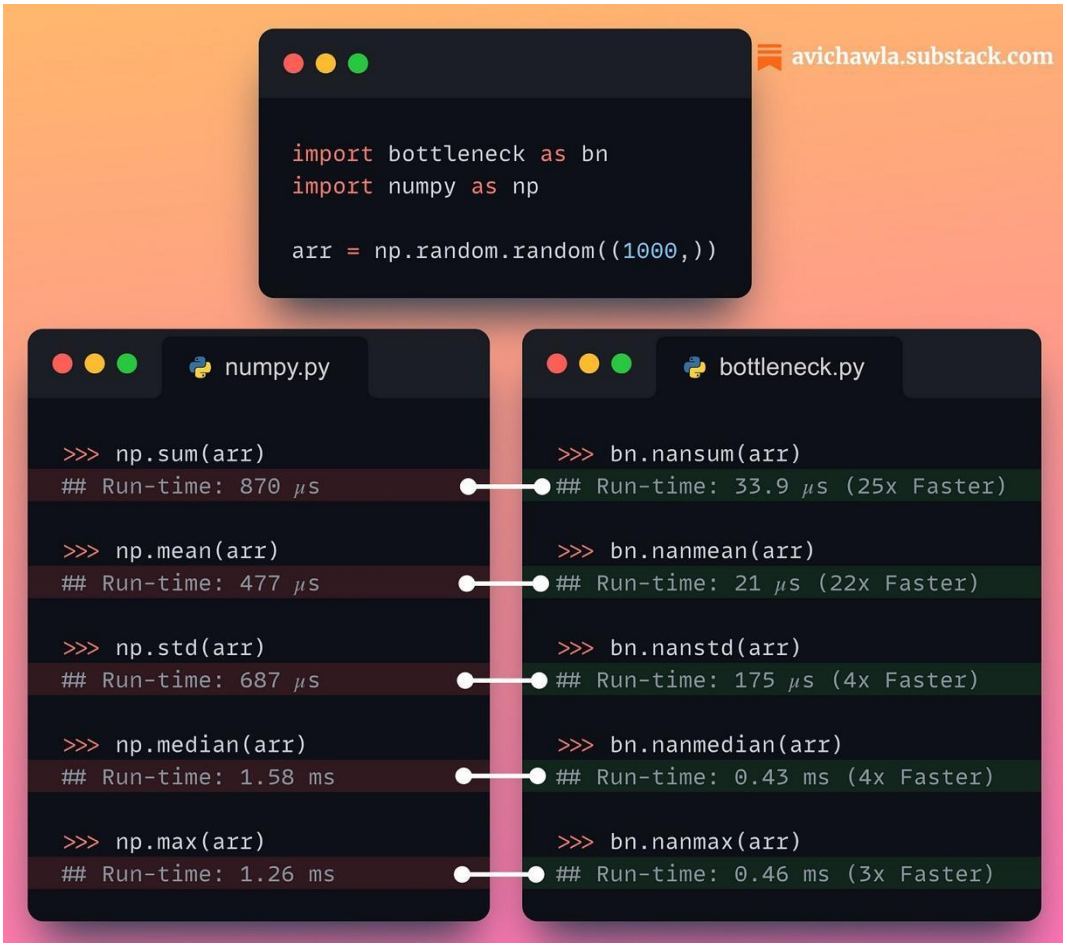
When manipulating a dataframe, at times, one may be using unoptimized methods. What's more, errors introduced into the data can easily go unnoticed.

To get hints and directions about your data/code, try Dovpanda. It works as a companion for your Pandas code. As a result, it gives suggestions/warnings about your data manipulation steps.

P.S. When you will import Dovpanda, you will likely get an error. Ignore it and proceed with using Pandas. You will still receive suggestions from Dovpanda.

Find more info here: Dovpandas.

Speedup NumPy Methods 25x With Bottleneck



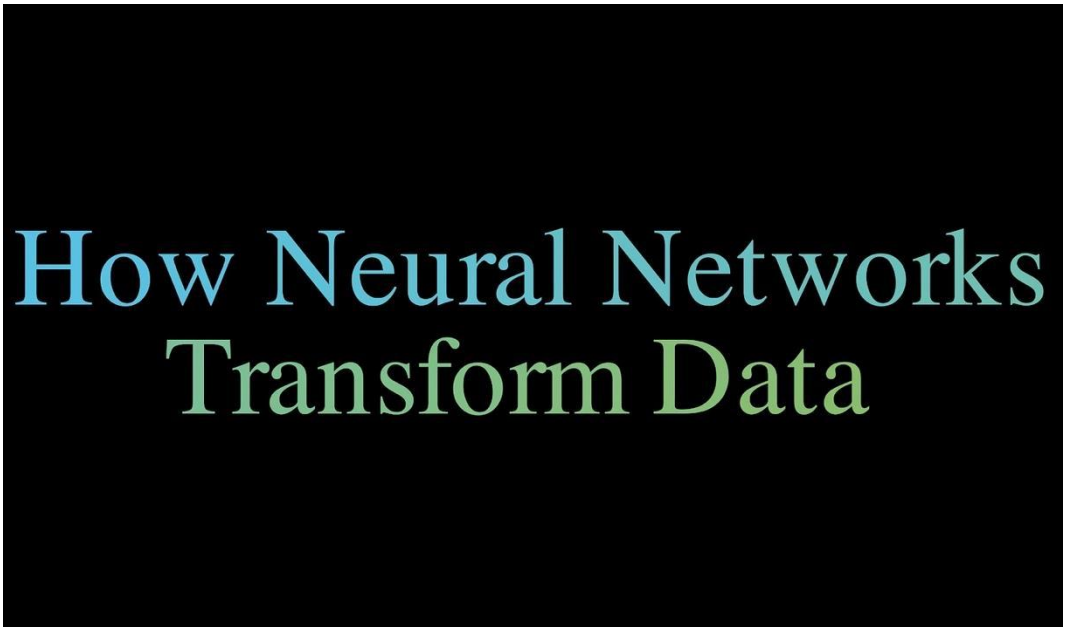
NumPy's methods are already highly optimized for performance. Yer, here's how you can further speed them up.

Bottleneck provides a suite of optimized implementations of NumPy methods.

Bottleneck is especially efficient for arrays with NaN values where performance boost can reach up to 100-120x.

Find more info here: Bottleneck.

Visualizing The Data Transformation of a Neural Network



If you struggle to comprehend how a neural network learns complex non-linear data, I have created an animation that will surely help.

Please find the video here: **Neural Network Animation.**

For linearly inseparable data, the task boils down to projecting the data to a space where it becomes linearly separable.

Now, either you could do this manually by adding relevant features that will transform your data to a linear separable form. Consider concentric circles for instance. Passing a square of (x,y) coordinates as a feature will do this job.

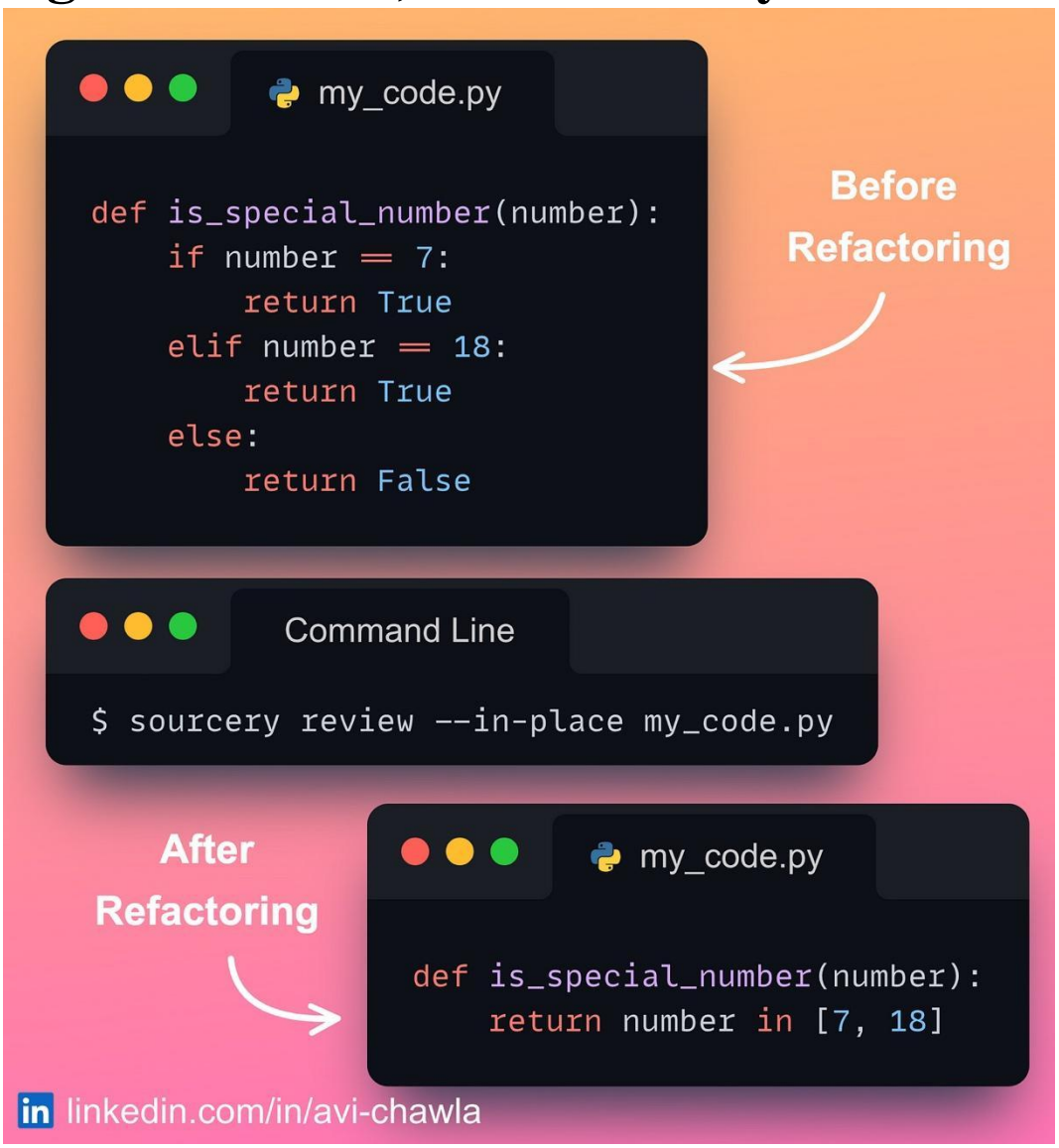
But in most cases, the transformation is unknown or complex to figure out. Thus, non-linear activation functions are considered the best bet, and a neural network is allowed to figure out this "non-linear to linear transformation" on its own.

As shown in the animation, if we tweak the neural network by adding a 2D layer right before the output, and visualize this transformation, we see that the neural network has learned to linearly separate the data. We add a layer 2D because it is easy to visualize.

This linearly separable data can be easily classified by the last layer. To put it another way, the last layer is analogous to a logistic regression model which is given a linear separable input.

The code for this visualization experiment is available here: GitHub.

Never Refactor Your Code Manually Again. Instead, Use Sourcery!



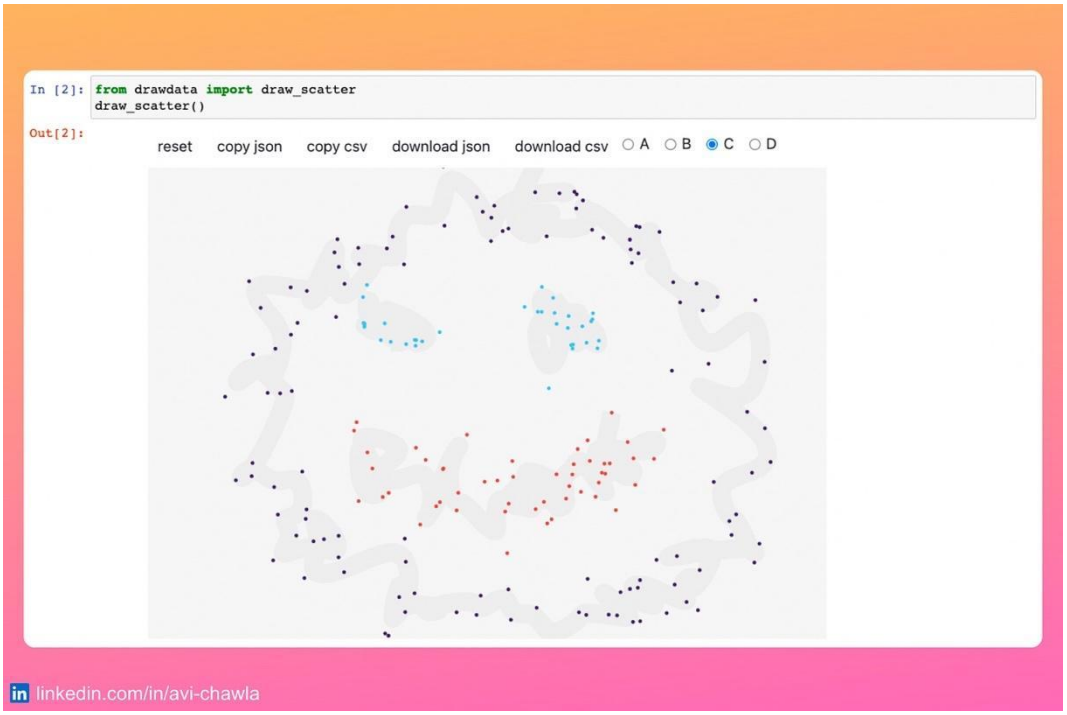
Refactoring code is an important step in pipeline development. Yet, manual refactoring takes additional time for testing as one might unknowingly introduce errors.

Instead, use Sourcery. It's an automated refactoring tool that makes your code elegant, concise, and Pythonic in no time.

With Sourcery, you can refactor code from the command line, as an IDE plugin in VS Code and PyCharm, pre-commit, etc.

Find more info here: Sourcery.

Draw The Data You Are Looking For In Seconds



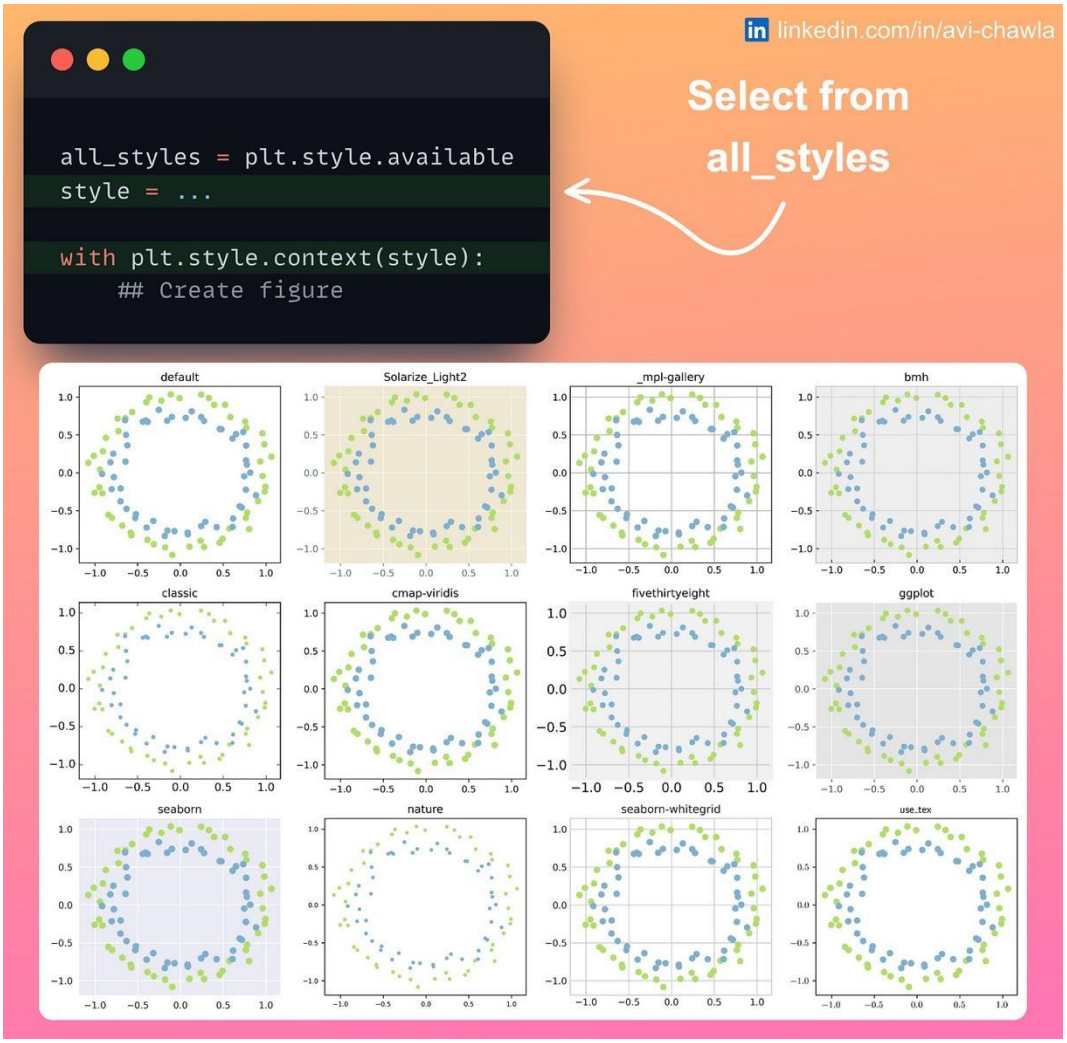
Please watch a video version of this post for better understanding: **Video Link**.

Often when you want data of some specific shape, programmatically generating it can be a tedious and time-consuming task.

Instead, use drawdata. This allows you to draw any 2D dataset in a notebook and export it. Besides a scatter plot, it can also create histogram and line plot

Find more info here: Drawdata.

Style Matplotlib Plots To Make Them More Attractive

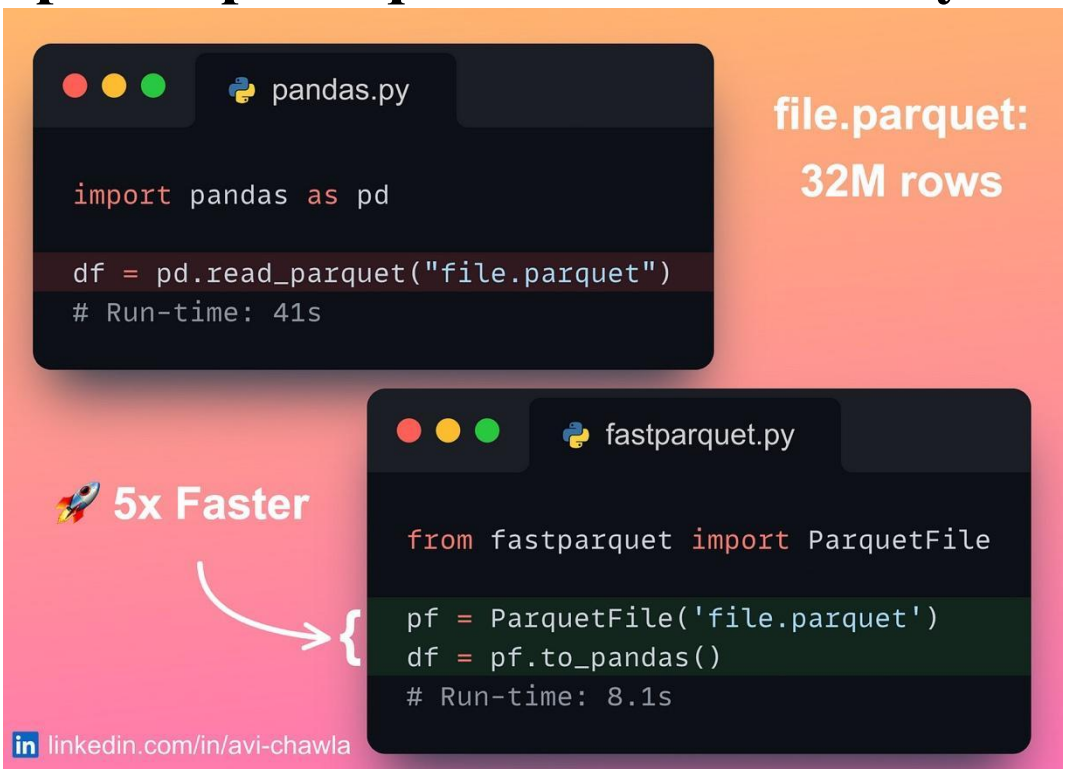


Matplotlib offers close to 50 different styles to customize the plot's appearance.

To alter the plot's style, select a style from **plt.style.available** and create the plot as you originally would.

Find more info about styling here: Docs.

Speed-up Parquet I/O of Pandas by 5x



Dataframes are often stored in parquet files and read using Pandas' **read_parquet**() method.

Rather than using Pandas, which relies on a single-core, use fastparquet. It offers immense speedups for I/O on parquet files using parallel processing.

Find more info here: Docs.

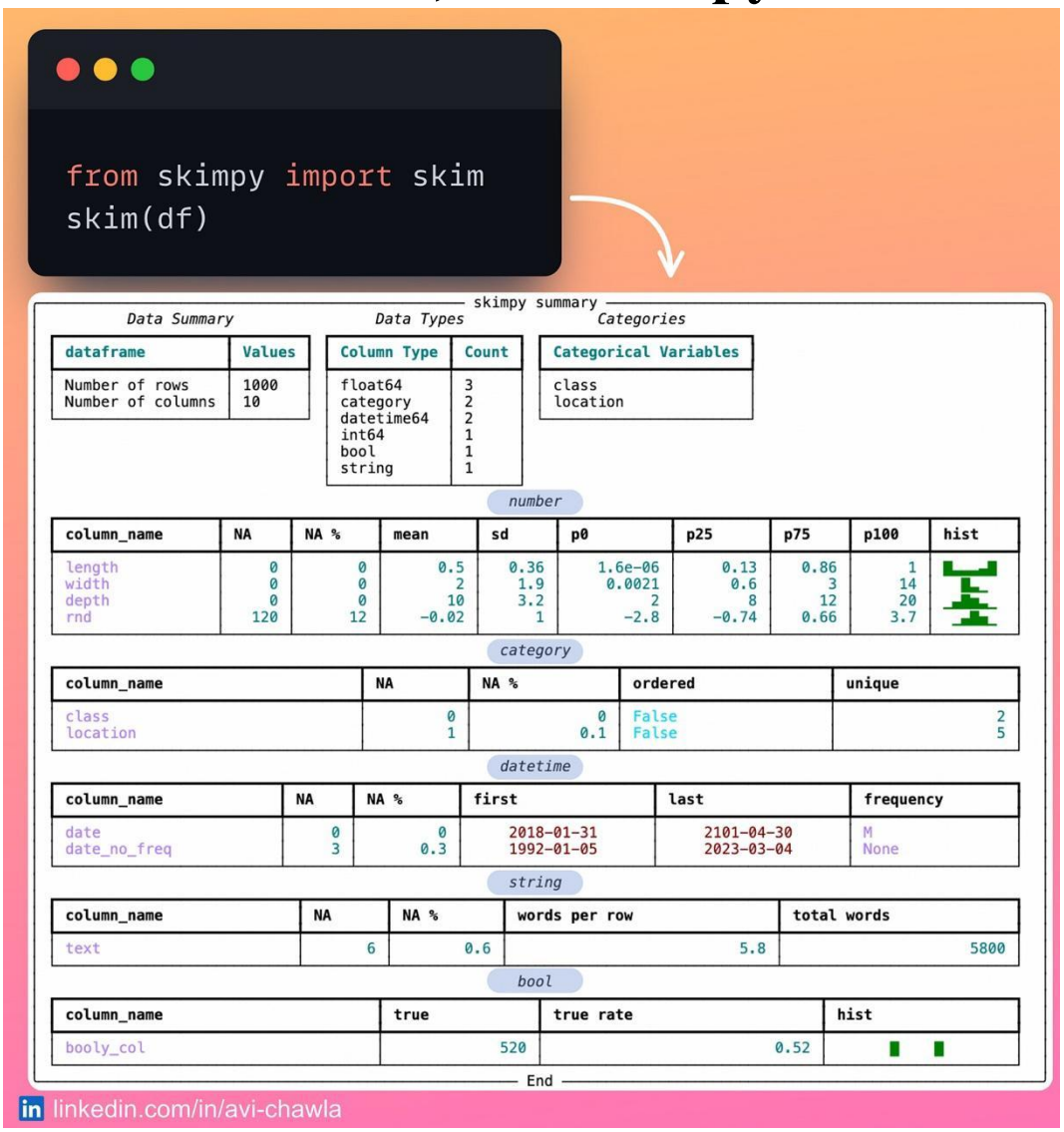
40 Open-Source Tools to Supercharge Your Pandas Workflow

Pandas receives over 3M downloads per day. But 99% of its users are not using it to its full potential.

I discovered these open-source gems that will immensely supercharge your Pandas workflow the moment you start using them.

Read this list here:
https://avichawla.substack.com/p/37-open-source-tools-to-supercharge-pandas.

Stop Using The Describe Method in Pandas. Instead, use Skimpy.



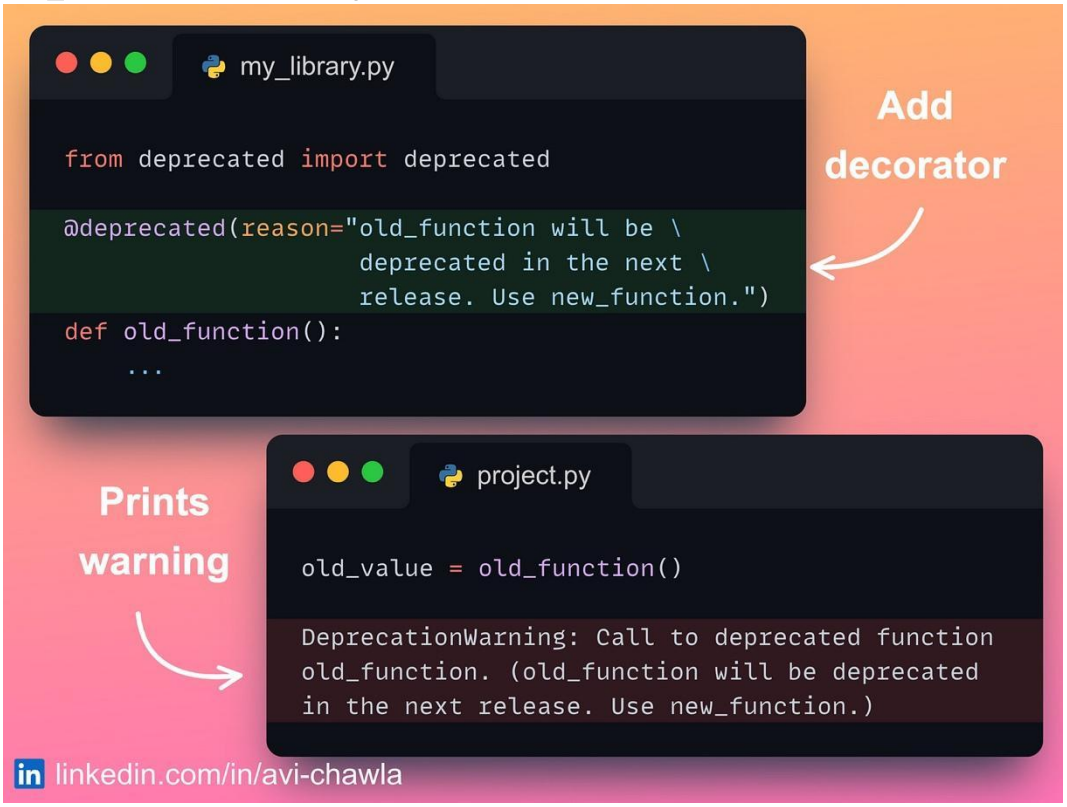
Supercharge the describe method in Pandas.

Skimpy is a lightweight tool for summarizing Pandas dataframes. In a single line of code, it generates a richer statistical summary than the describe() method.

What's more, the summary is grouped by datatypes for efficient analysis. You can use Skimpy from the command line too.

Find more info here: Docs.

The Right Way to Roll Out Library Updates in Python



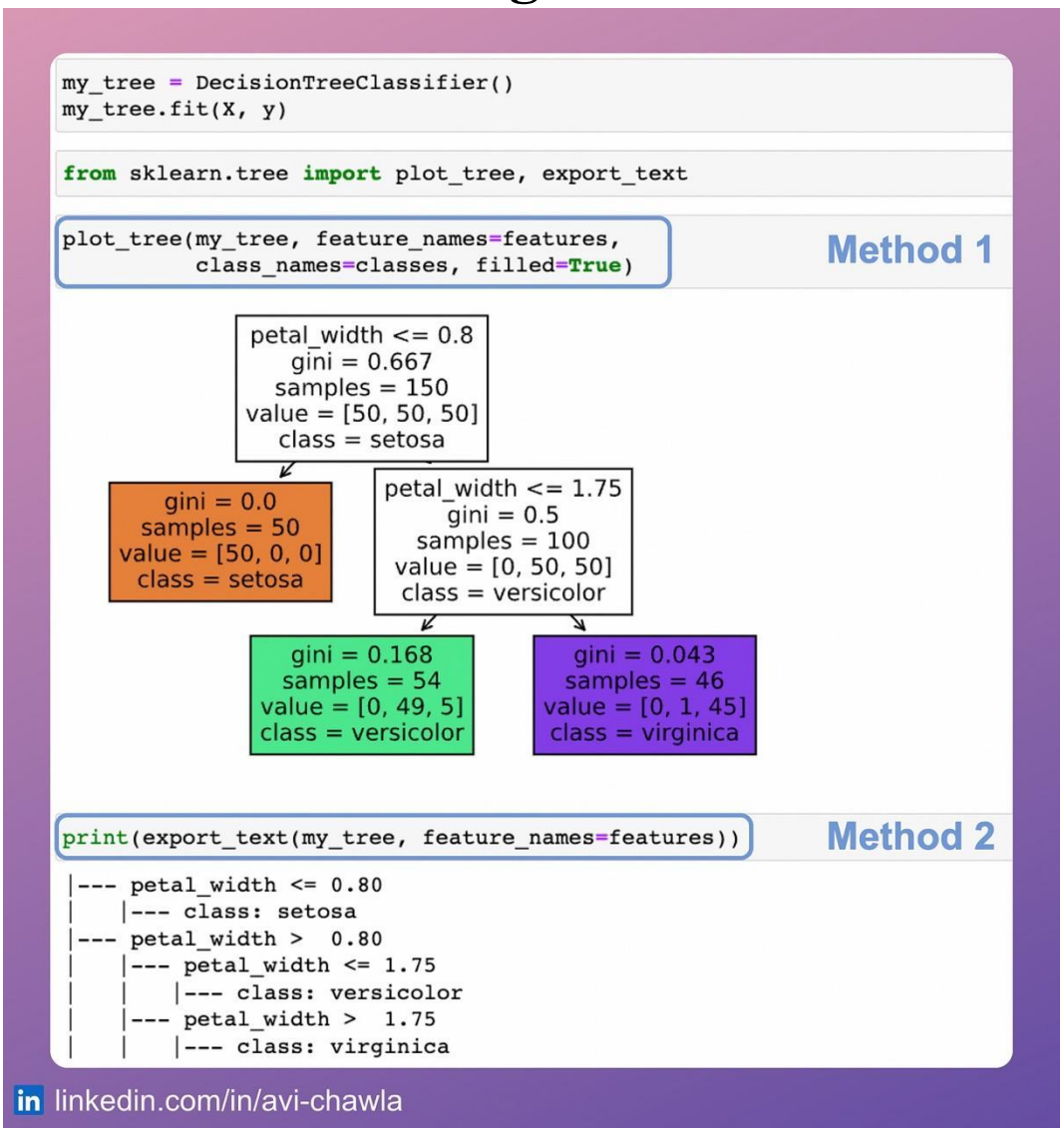
While developing a library, authors may decide to remove some functions/methods/classes. But instantly rolling the update without any prior warning isn't a good practice.

This is because many users may still be using the old methods and they may need time to update their code.

Using the **deprecated** decorator, one can convey a warning to the users about the update. This allows them to update their code before it becomes outdated.

Find more info here: GitHub.

Simple One-Liners to Preview a Decision Tree Using Sklearn



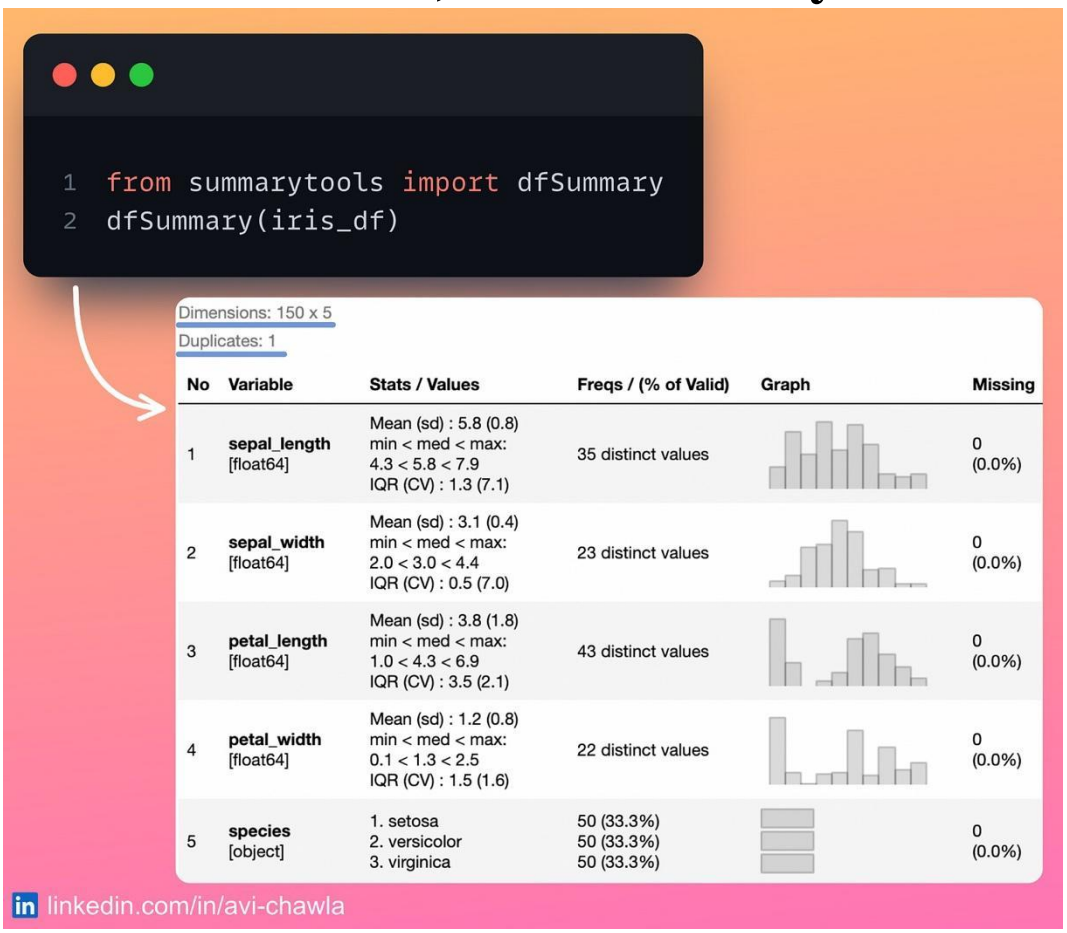
If you want to preview a decision tree, sklearn provides two simple methods to do so.

1. **plot_tree** creates a graphical representation of a decision tree.

2. **export_text** builds a text report showing the rules of a decision tree.

This is typically used to understand the rules learned by a decision tree and gaining a better understanding of the behavior of a decision tree model.

Stop Using The Describe Method in Pandas. Instead, use Summarytools.

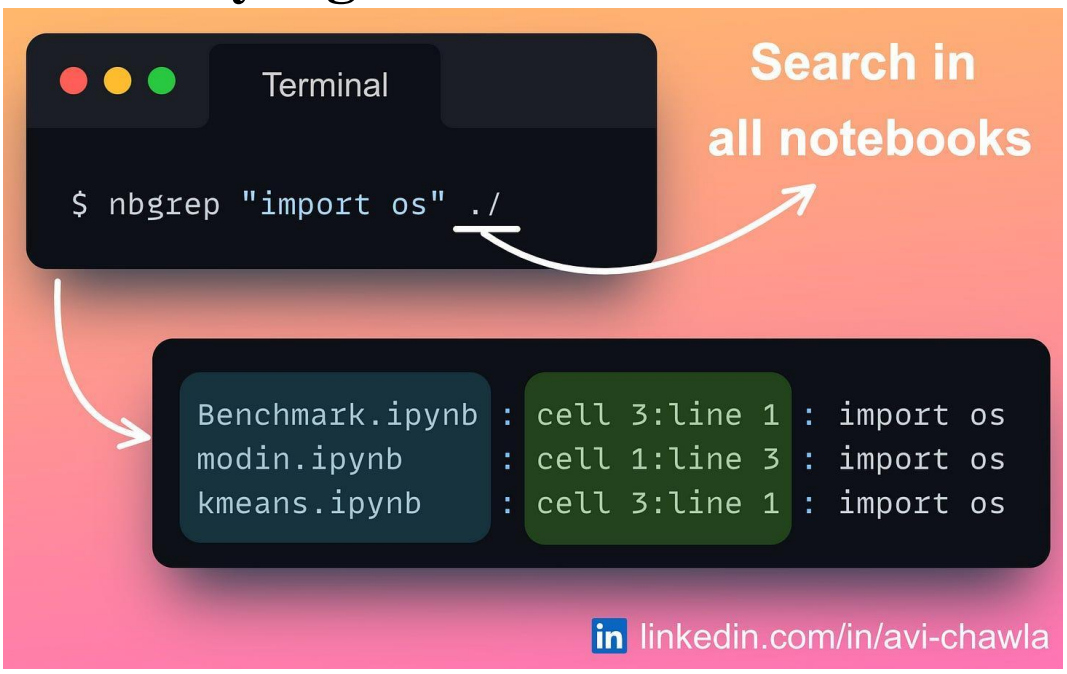


Summarytools is a simple EDA tool that gives a richer summary than **describe**() method. In a single line of code, it generates a standardized and comprehensive data summary.

The summary includes column statistics, frequency, distribution chart, and missing stats.

Find more info here: Summary Tools.

Never Search Jupyter Notebooks Manually Again To Find Your Code



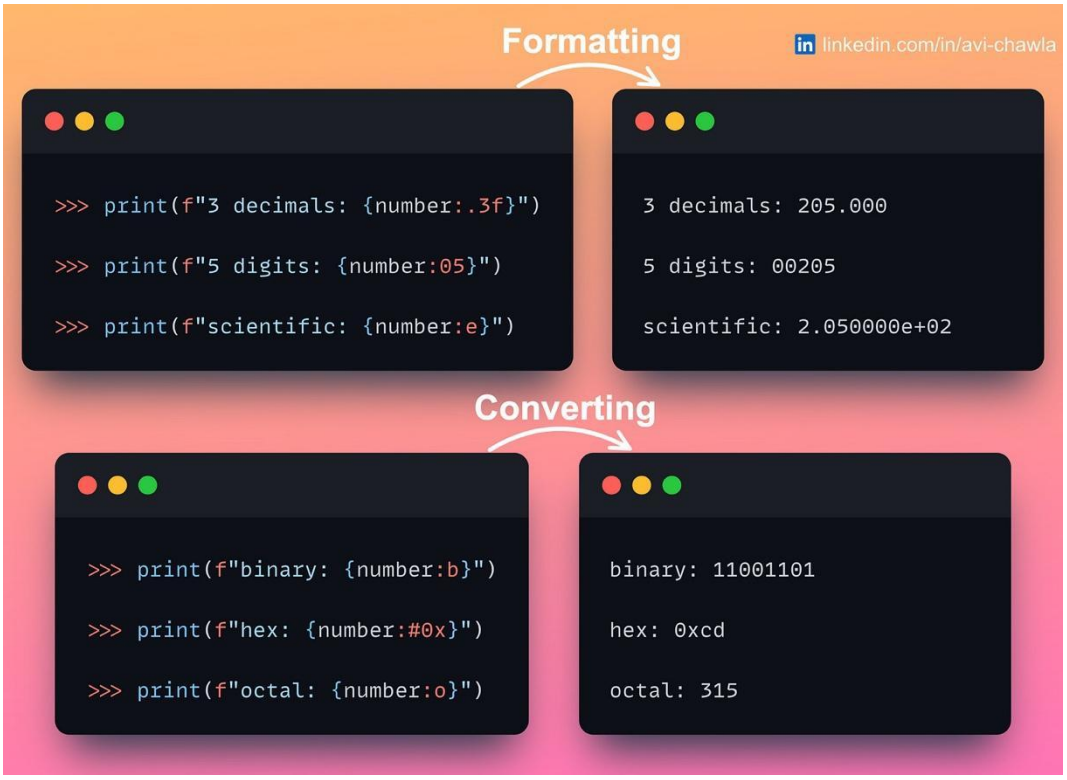
Have you ever struggled to recall the specific Jupyter notebook in which you wrote some code? Here's a quick trick to save plenty of manual work and time.

nbcommands provides a bunch of commands to interact with Jupyter from the terminal.

For instance, you can search for code, preview a few cells, merge notebooks, and many more.

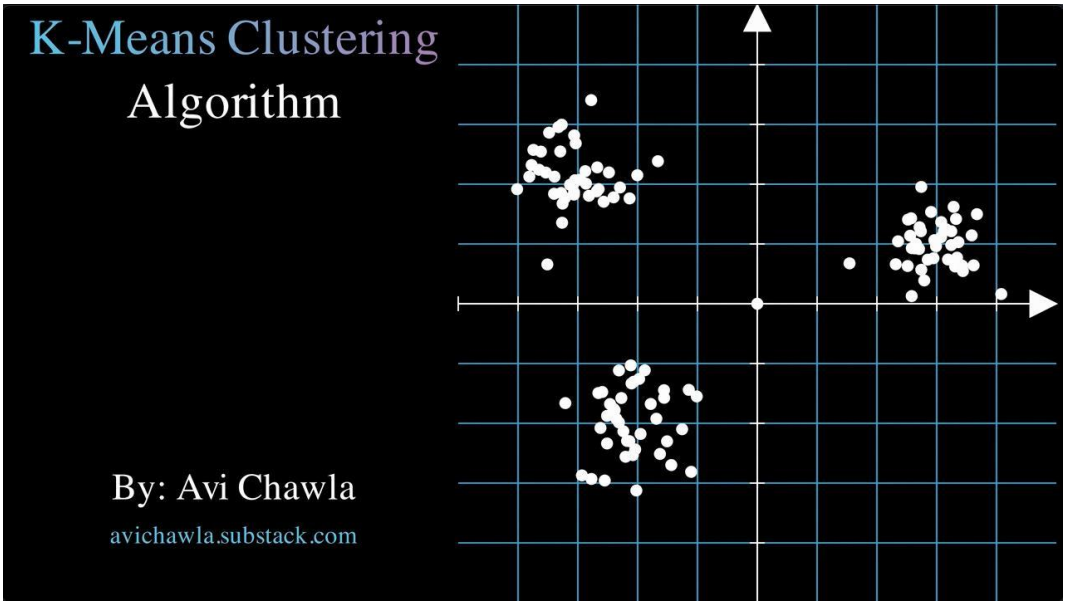
Find more info here: GitHub.

F-strings Are Much More Versatile Than You Think



Here are 6 lesser-known ways to format/convert a number using f-strings. What is your favorite f-string hack?

Is This The Best Animated Guide To KMeans Ever?



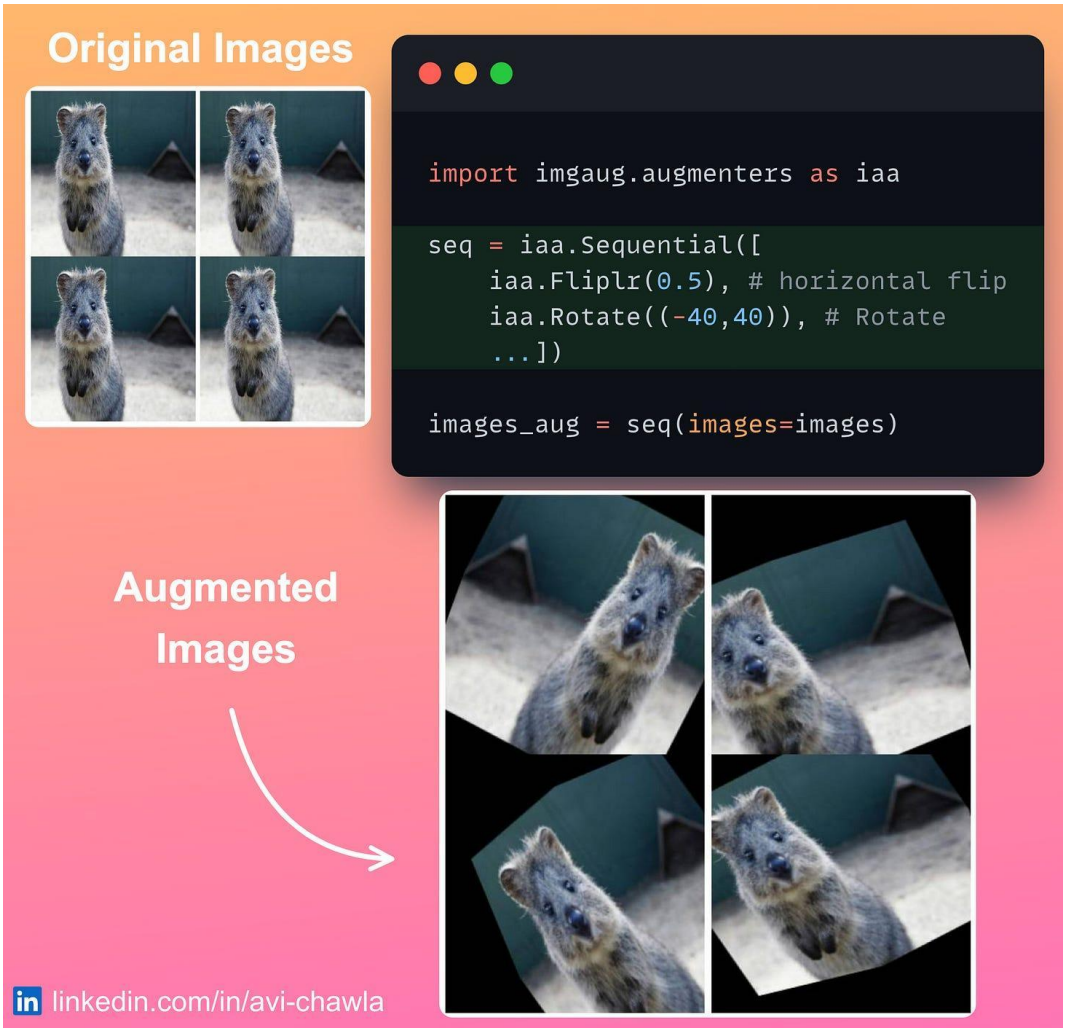
Have you ever struggled with understanding KMeans? How it works, how are the data points assigned to a centroid, or how do the centroids move?

If yes, let me help.

I created a beautiful animation using Manim to help you build an intuitive understanding of the algorithm.

Please find this video here: **Video Link.**

An Effective Yet Underrated Technique To Improve Model Performance



Robust ML models are driven by diverse training data. Here's a simple yet highly effective technique that can help you create a diverse dataset and increase model performance.

One way to increase data diversity is using data augmentation.

The idea is to create new samples by transforming the available samples. This can prevent overfitting, improve performance, and build robust models.

For images, you can use imgaug (linked in comments). It provides a variety of augmentation techniques such as flipping, rotating, scaling, adding noise to images, and many more.

Find more info: Imgaug.

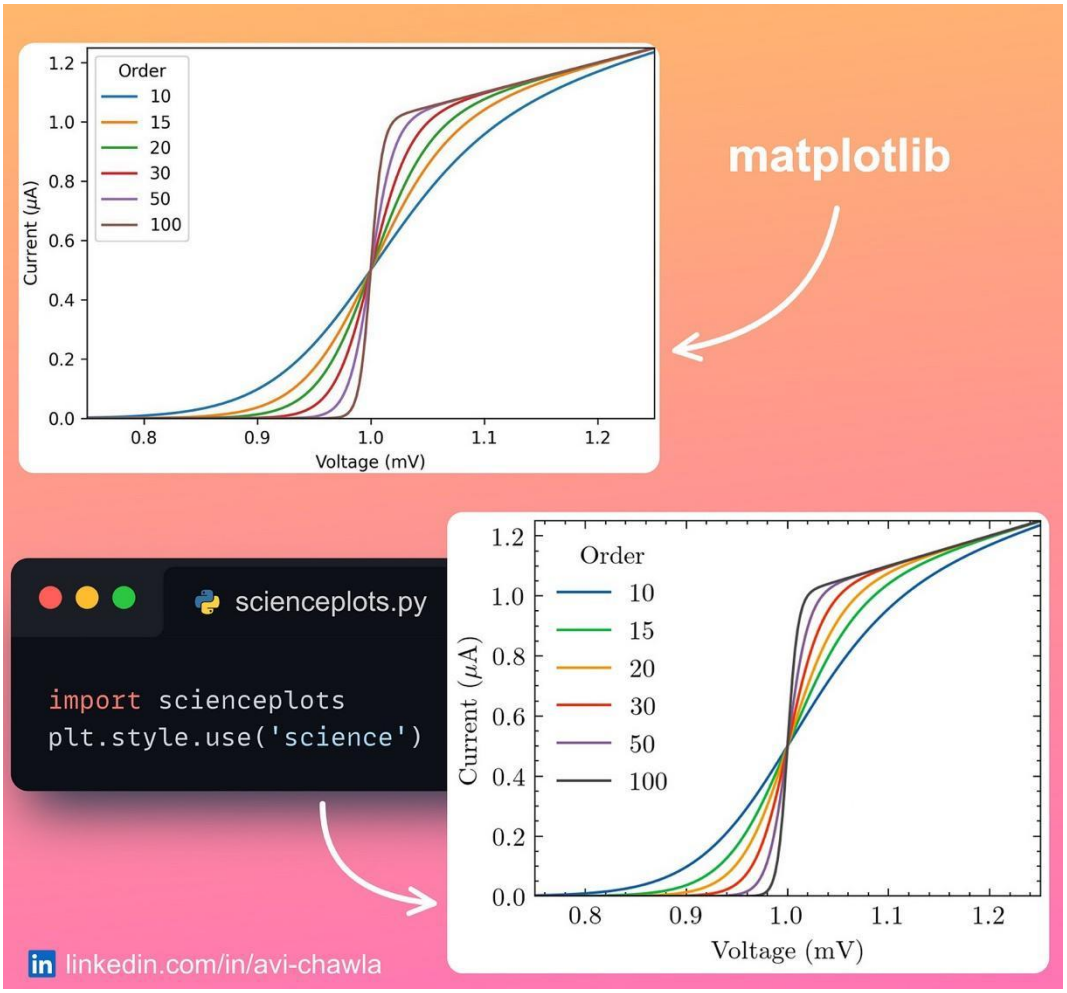
Create Data Plots Right From The Terminal

Visualizing data can get tough when you don't have access to a GUI. But here's what can help.

Bashplotlib offers a quick and easy way to make basic plots right from the terminal. Being pure python, you can quickly install it anywhere using pip and visualize your data.

Find more info here: Bashplotlib.

Make Your Matplotlib Plots More Professional



The default matplotlib plots are pretty basic in style and thus, may not be the apt choice always. Here's how you can make them appealing.

To create professional-looking and attractive plots for presentations, reports, or scientific papers, try Science Plots.

Adding just two lines of code completely transforms the plot's appearance.

Find more info here: GitHub.

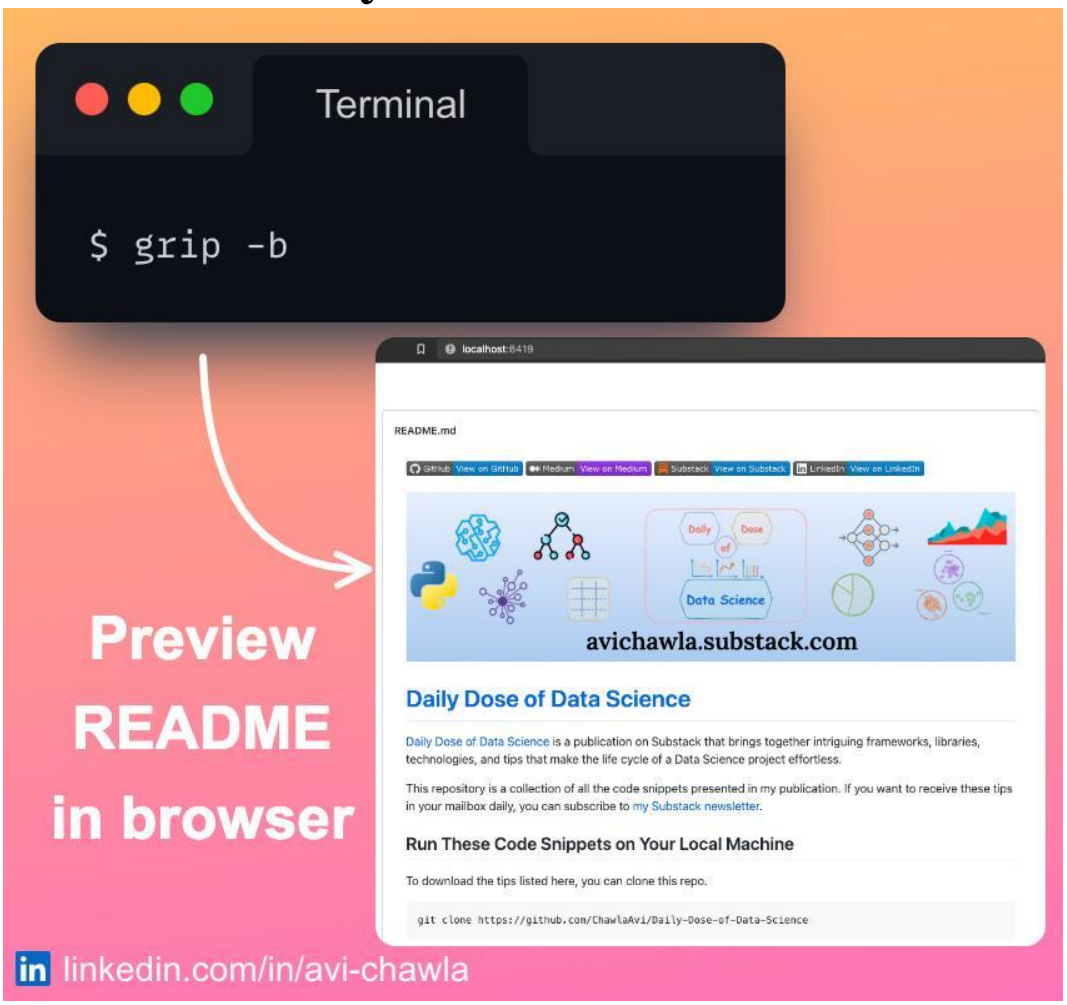
37 Hidden Python Libraries That Are Absolute Gems

I reviewed 1,000+ Python libraries and discovered these hidden gems I never knew even existed.

Here are some of them that will make you fall in love with Python' and its versatility (even more).

Read this list here: https://avichawla.substack.com/p/gem-libraries.

Preview Your README File Locally In GitHub Style



Please watch a video version for better understanding: **Video Link.**

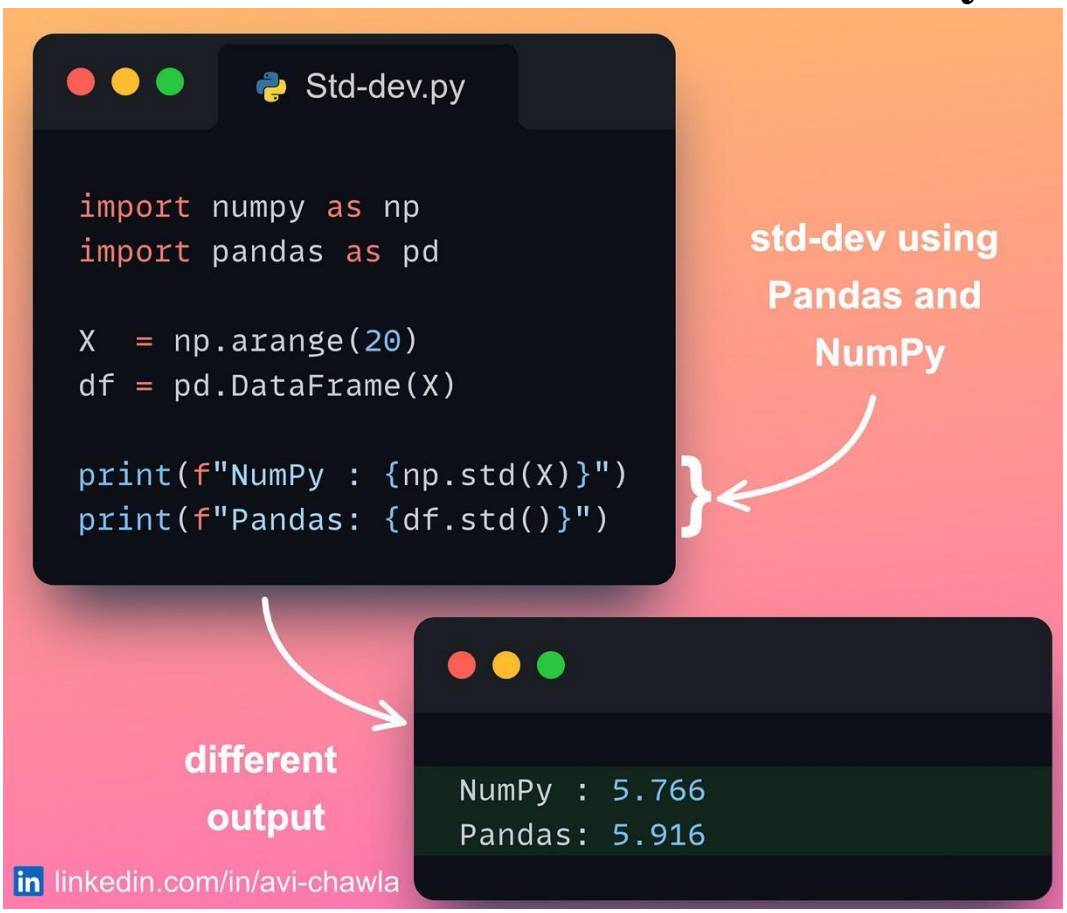
Have you ever wanted to preview a README file before committing it to GitHub? Here's how to do it.

Grip is a command-line tool that allows you to render a README file as it will appear on GitHub. This is extremely useful as sometimes one may want to preview the file before pushing it to GitHub.

What's more, editing the README instantly reflects in the browser without any page refresh.

Read more: Grip.

Pandas and NumPy Return Different Values for Standard Deviation. Why?



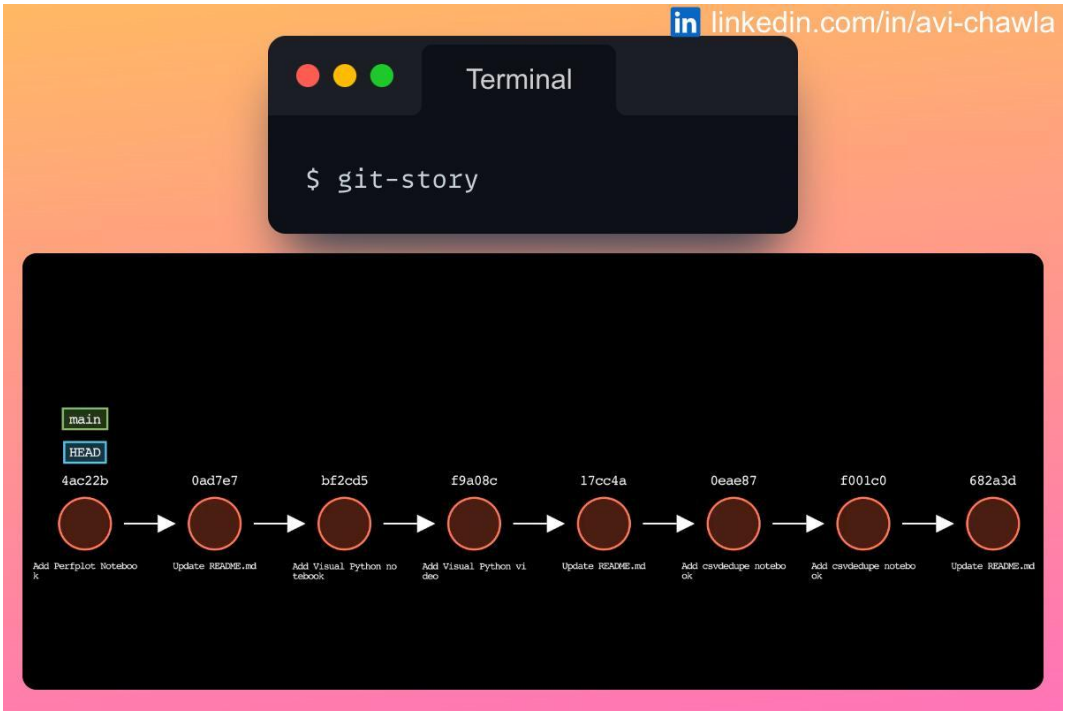
Pandas assumes that the data is a sample of the population and that the obtained result can be biased towards the sample.

Thus, to generate an unbiased estimate, it uses (n-1) as the dividing factor instead of n. In statistics, this is also known as Bessel's correction.

NumPy, however, does not make any such correction.

Find more info here: Bessel's correction.

Visualize Commit History of Git Repo With Beautiful Animations



As the size of your project grows, it can get difficult to comprehend the Git tree.

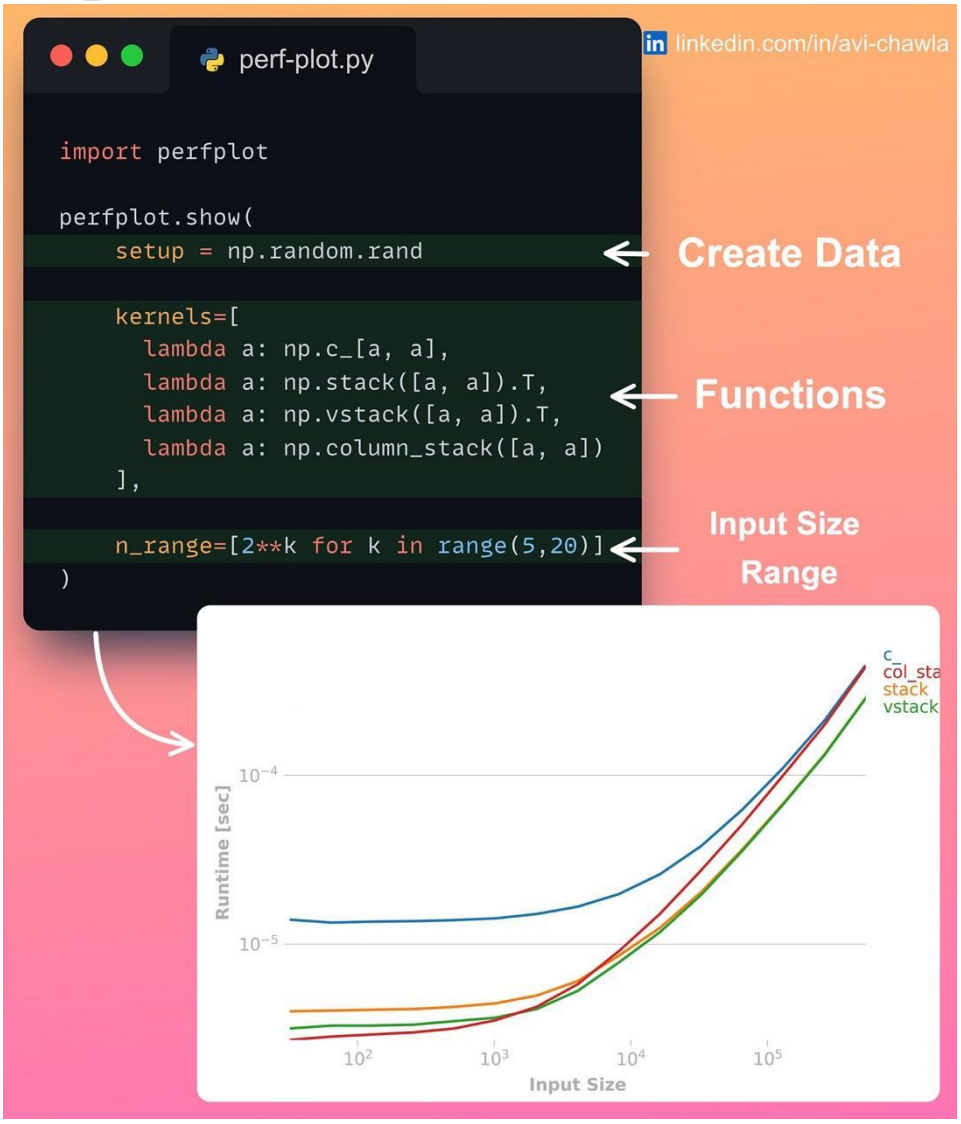
Git-story is a command line tool to create elegant animations for your git repository.

It generates a video that depicts the commits, branches, merges, HEAD commit, and many more. Find more info in the comments.

Please watch a video version of this post here: **Video.**

Read more: Git-story.

Perfplot: Measure, Visualize and Compare Run-time With Ease



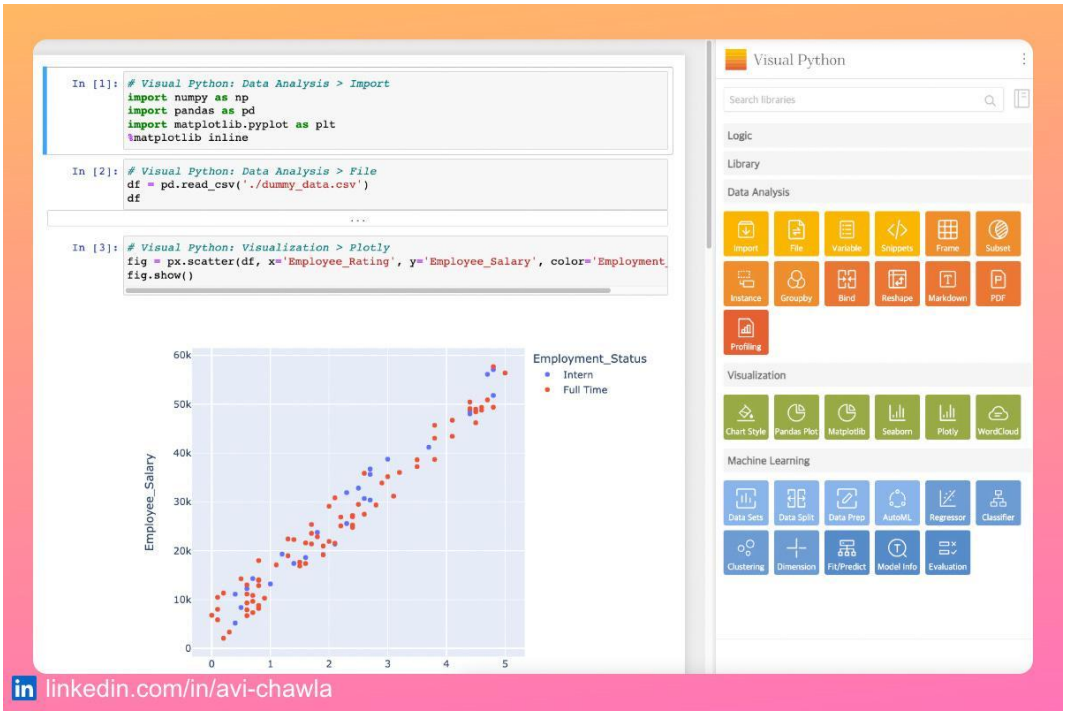
Here's an elegant way to measure the run-time of various Python functions.

Perfplot is a tool designed for quick run-time comparisons of many functions/algorithms.

It extends Python's timeit package and allows you to quickly visualize the run-time in a clear and informative way.

Find more info: Perfplot.

This GUI Tool Can Possibly Save You Hours Of Manual Work



Please watch a video version of this post for better understanding: **Link**.

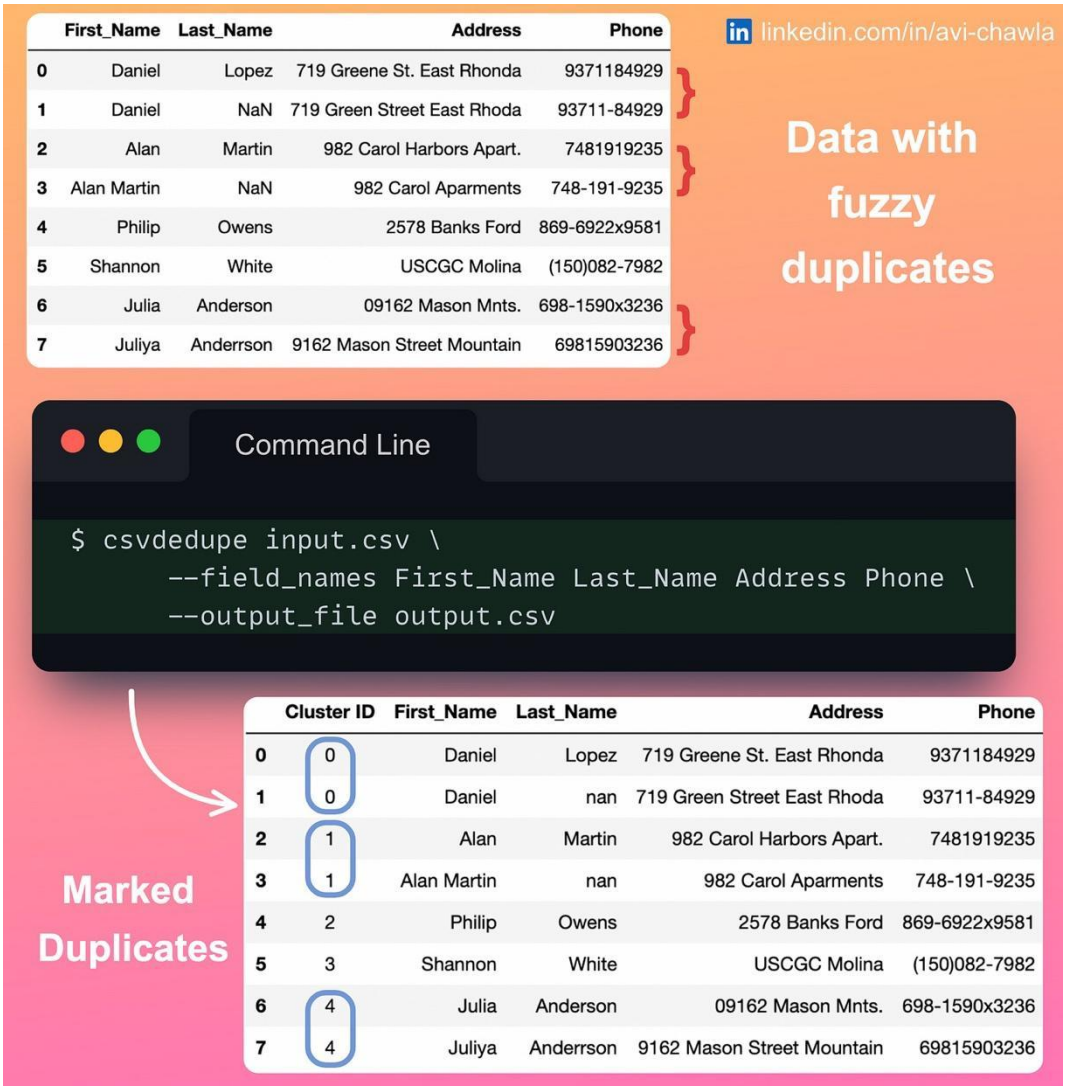
This is indeed one of the coolest and most useful Jupyter notebook-based data science tools.

Visual Python is a GUI-based python code generator. Using this, you can easily eliminate writing code for many repetitive tasks. This includes importing libraries, I/O, Pandas operations, plotting, etc.

Moreover, with the click of a couple of buttons, you can import the code for many ML-based utilities. This covers sklearn models, evaluation metrics, data splitting functions, and many more.

Read more: Visual Python.

How Would You Identify Fuzzy Duplicates In A Data With Million Records?



	First_Name	Last_Name	Address	Phone
0	Daniel	Lopez	719 Greene St. East Rhonda	9371184929
1	Daniel	NaN	719 Green Street East Rhoda	93711-84929
2	Alan	Martin	982 Carol Harbors Apart.	7481919235
3	Alan Martin	NaN	982 Carol Aparments	748-191-9235
4	Philip	Owens	2578 Banks Ford	869-6922x9581
5	Shannon	White	USCGC Molina	(150)082-7982
6	Julia	Anderson	09162 Mason Mnts.	698-1590x3236
7	Juliya	Anderrson	9162 Mason Street Mountain	69815903236

	Cluster ID	First_Name	Last_Name	Address	Phone
0	0	Daniel	Lopez	719 Greene St. East Rhonda	9371184929
1	0	Daniel	nan	719 Green Street East Rhoda	93711-84929
2	1	Alan	Martin	982 Carol Harbors Apart.	7481919235
3	1	Alan Martin	nan	982 Carol Aparments	748-191-9235
4	2	Philip	Owens	2578 Banks Ford	869-6922x9581
5	3	Shannon	White	USCGC Molina	(150)082-7982
6	4	Julia	Anderson	09162 Mason Mnts.	698-1590x3236
7	4	Juliya	Anderrson	9162 Mason Street Mountain	69815903236

Imagine you have over a million records with fuzzy duplicates. How would you identify potential duplicates?

The naive approach of comparing every pair of records is infeasible in such cases. That's over 10^12 comparisons (n^2). Assuming a speed of 10,000 comparisons per second, it will take roughly 3 years to complete.

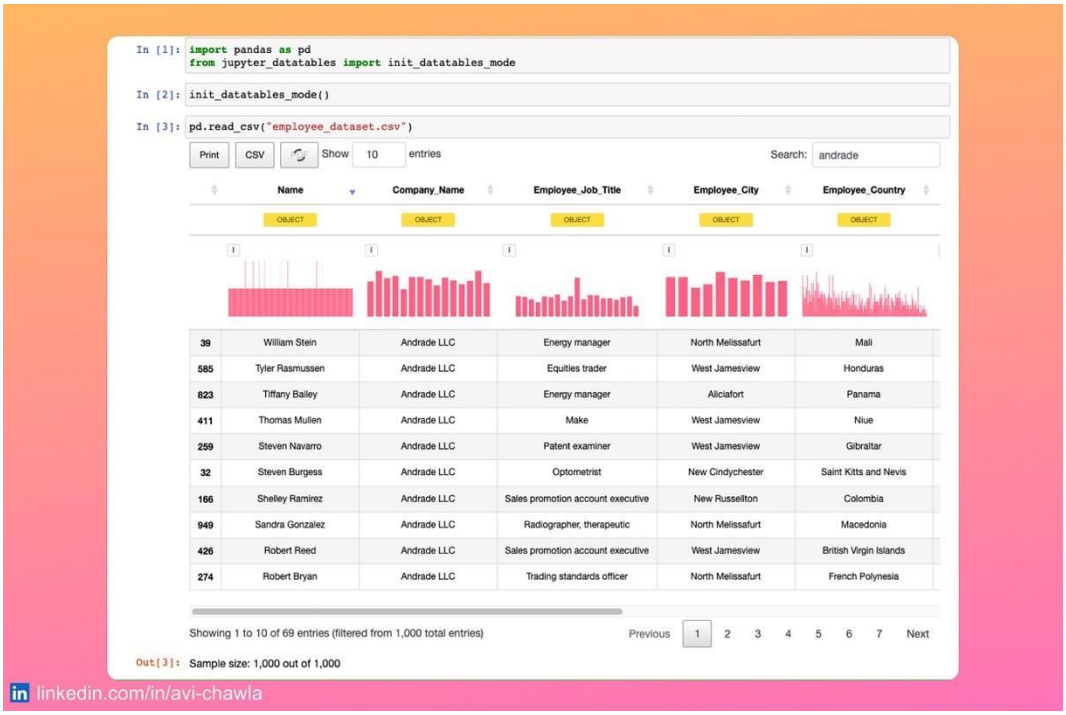
The csvdedupe tool (linked in comments) solves this by cleverly reducing the comparisons. For instance, comparing the name “Daniel” to “Philip” or “Shannon” to “Julia” makes no sense. They are guaranteed to be distinct records.

Thus, it groups the data into smaller buckets based on rules. One rule could be to group all records with the same first three letters in the name.

This way, it drastically reduces the number of comparisons with great accuracy.

Read more: csvdedupe.

Stop Previewing Raw DataFrames. Instead, Use DataTables.



After loading any dataframe in Jupyter, we preview it. But it hardly tells anything about the data.

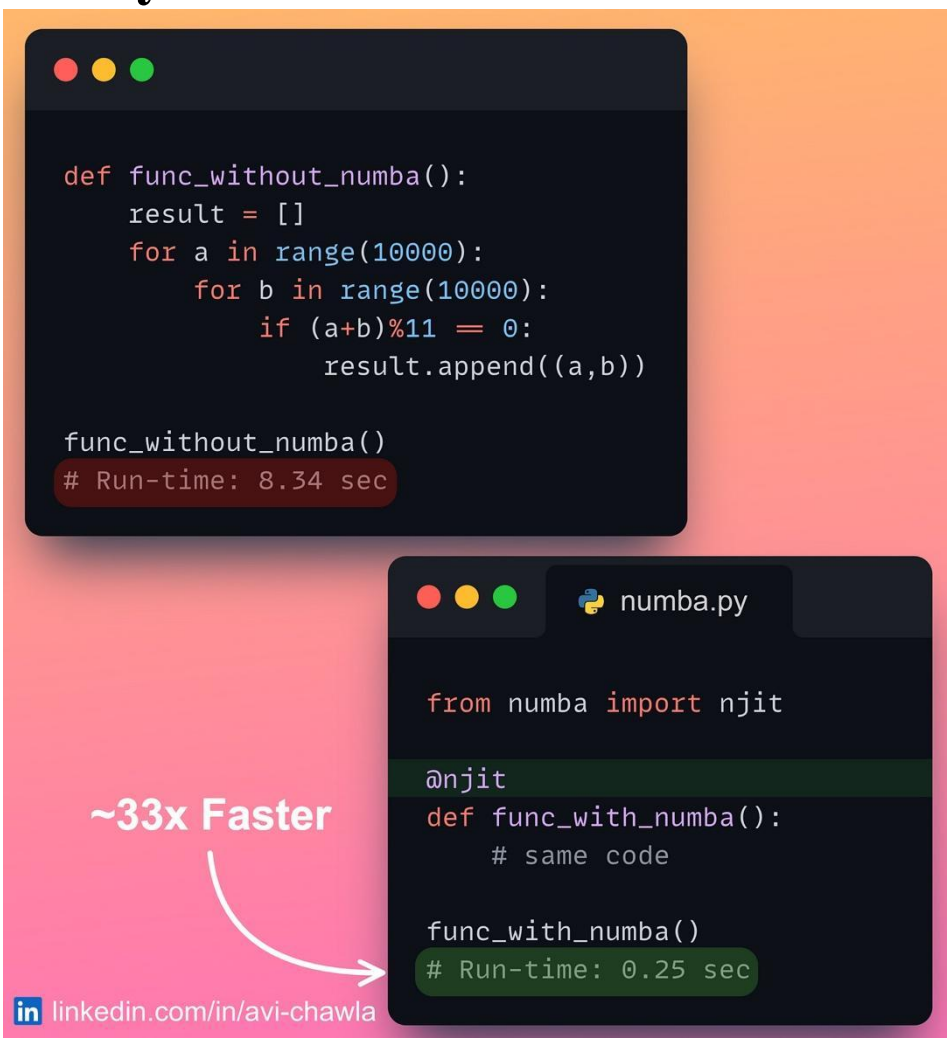
One has to dig deeper by analyzing it, which involves simple yet repetitive code.

Instead, use Jupyter-DataTables.

It supercharges the default preview of a DataFrame with many common operations. This includes sorting, filtering, exporting, plotting column distribution, printing data types, and pagination.

Please view a video version here for better understanding: **Post Link**.

🚀 A Single Line That Will Make Your Python Code Faster



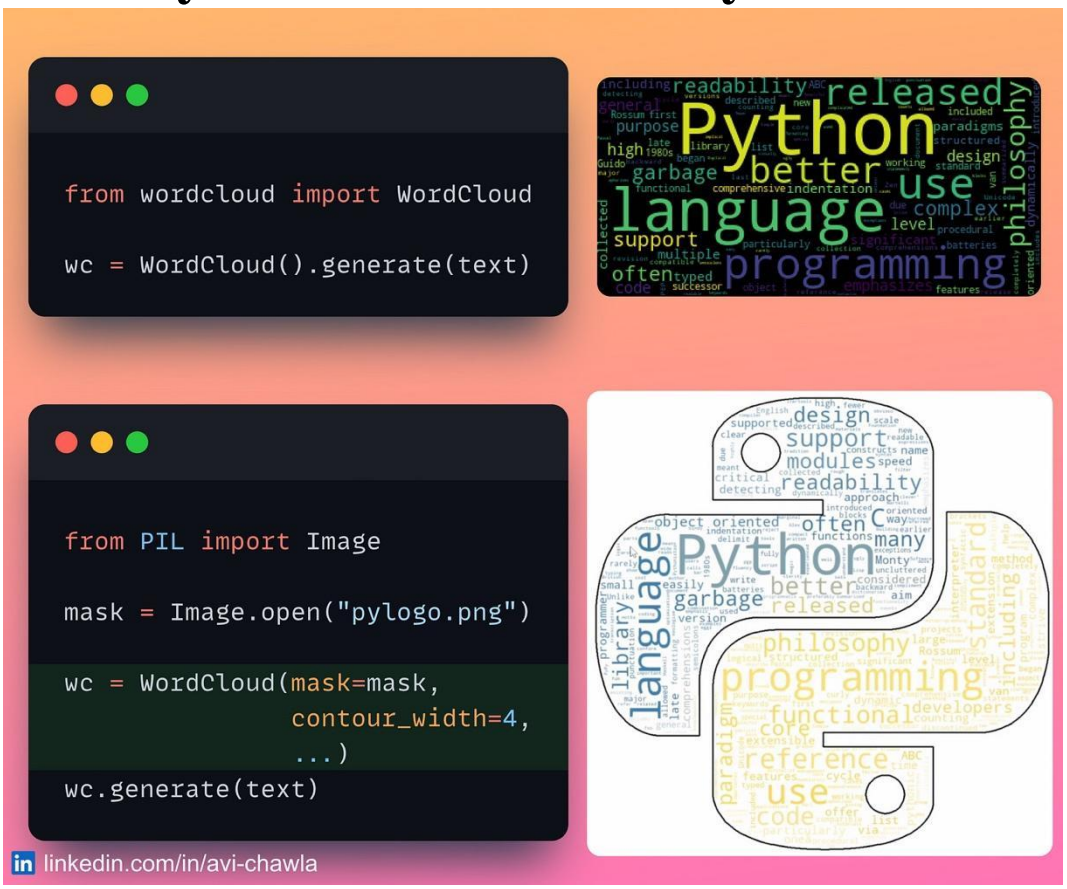
If you are frustrated with Python's run-time, here's how a single line can make your code blazingly fast.

Numba is a just-in-time (JIT) compiler for Python. This means that it takes your existing python code and generates a fast machine code (at run-time).

Thus, post compilation, your code runs at native machine code speed. Numba works best on code that uses NumPy arrays and functions, and loops.

Get Started: Numba Guide.

Prettify Word Clouds In Python

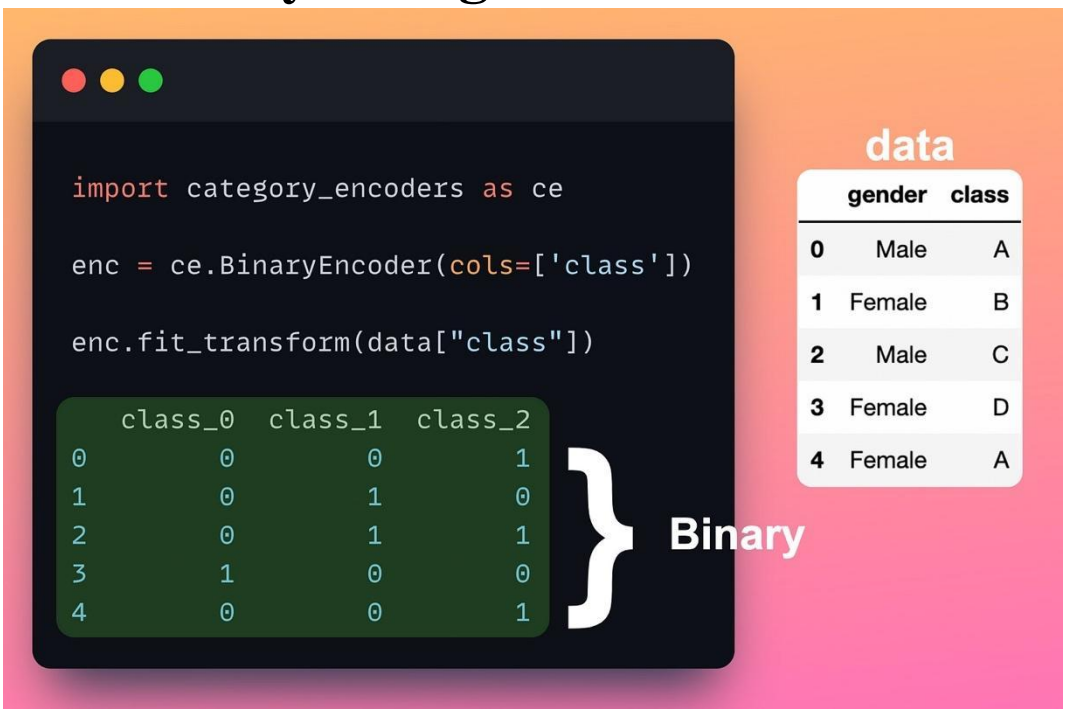


If you use word clouds often, here's a quick way to make them prettier.

In Python, you can easily alter the shape and color of a word cloud. By supplying a mask image, the resultant world cloud will take its shape and appear fancier.

Find more info here: Notebook Link.

How to Encode Categorical Features With Many Categories?



We often encode categorical columns with one-hot encoding. But the feature matrix becomes sparse and unmanageable with many categories.

The category-encoders library provides a suite of encoders specifically for categorical variables. This makes it effortless to experiment with various encoding techniques.

For instance, I used its binary encoder above to represent a categorical column in binary format.

Read more: Documentation.

Calendar Map As A Richer Alternative to Line Plot

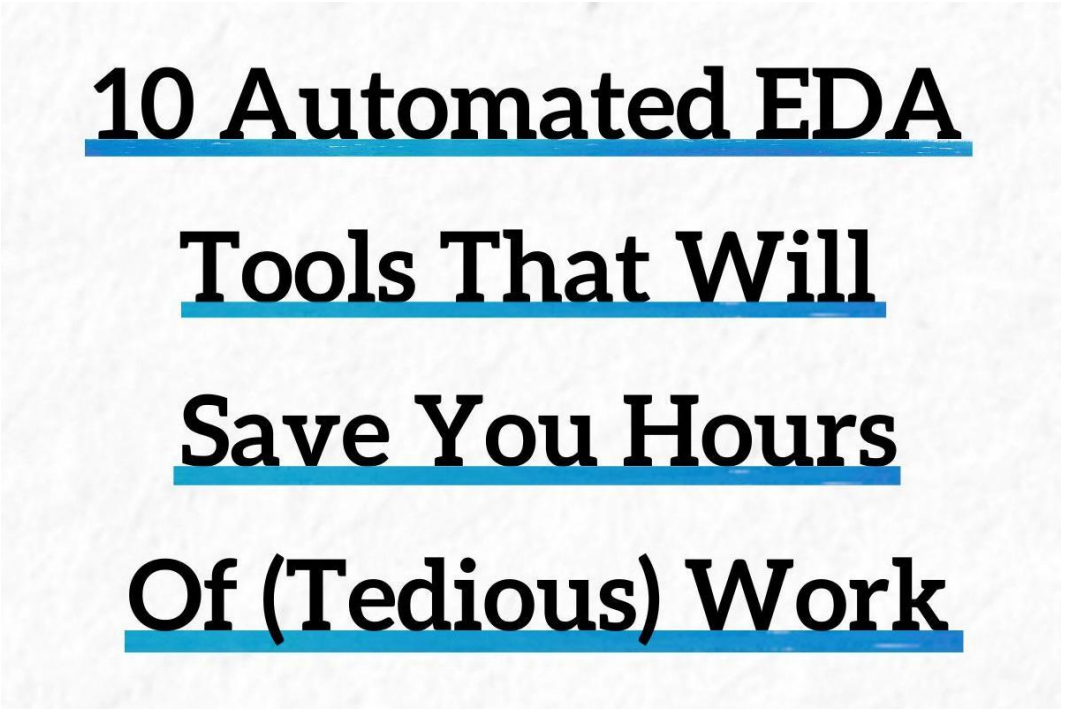
Ever seen one of those calendar heat maps? Here's how you can create one in two lines of Python code.

A calendar map offers an elegant way to visualize daily data. At times, they are better at depicting weekly/monthly seasonality in data instead of line plots. For instance, imagine creating a line plot for “Work Group Messages” above.

To create one, you can use "plotly_calplot". Its input should be a DataFrame. A row represents the value corresponding to a date.

Read more: Plotly Calplot.

10 Automated EDA Tools That Will Save You Hours Of (Tedious) Work

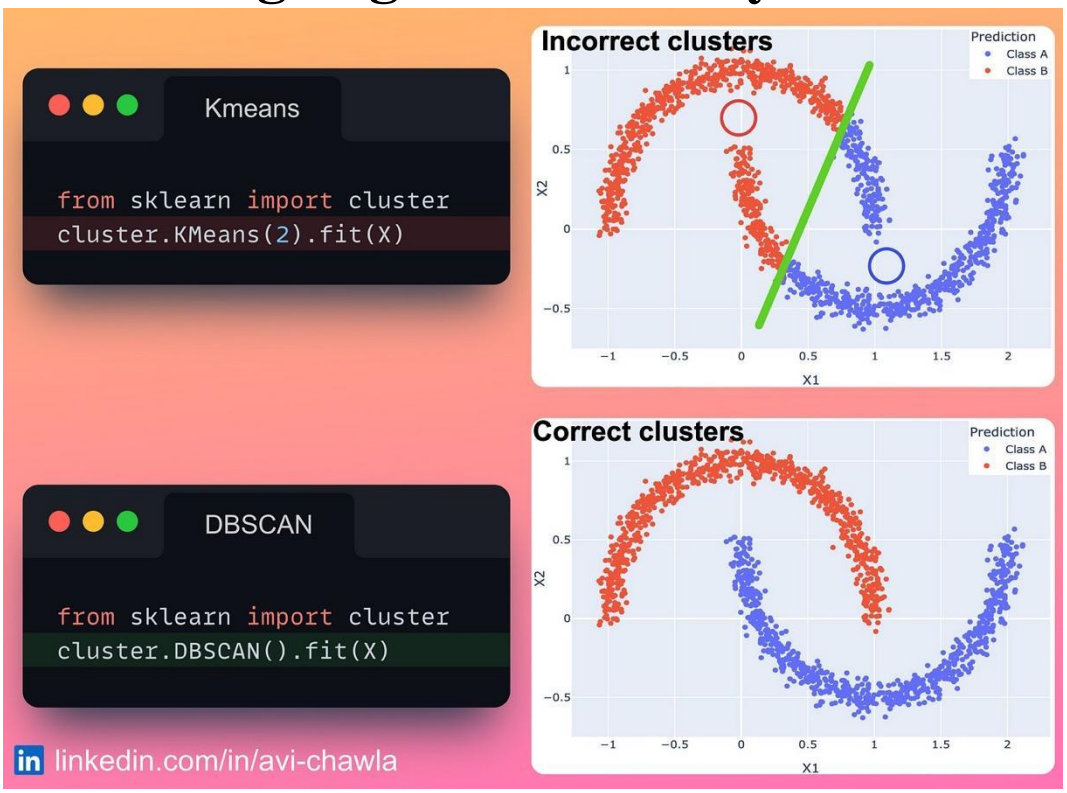


Most steps in a data analysis task stay the same across projects. Yet, manually digging into the data is tedious and time-consuming, which inhibits productivity.

Here are 10 EDA tools that automate these repetitive steps and profile your data in seconds.

Please find this full document in my LinkedIn post: Post Link.

Why KMeans May Not Be The Apt Clustering Algorithm Always



KMeans is a popular clustering algorithm. Yet, its limitations make it inapplicable in many cases.

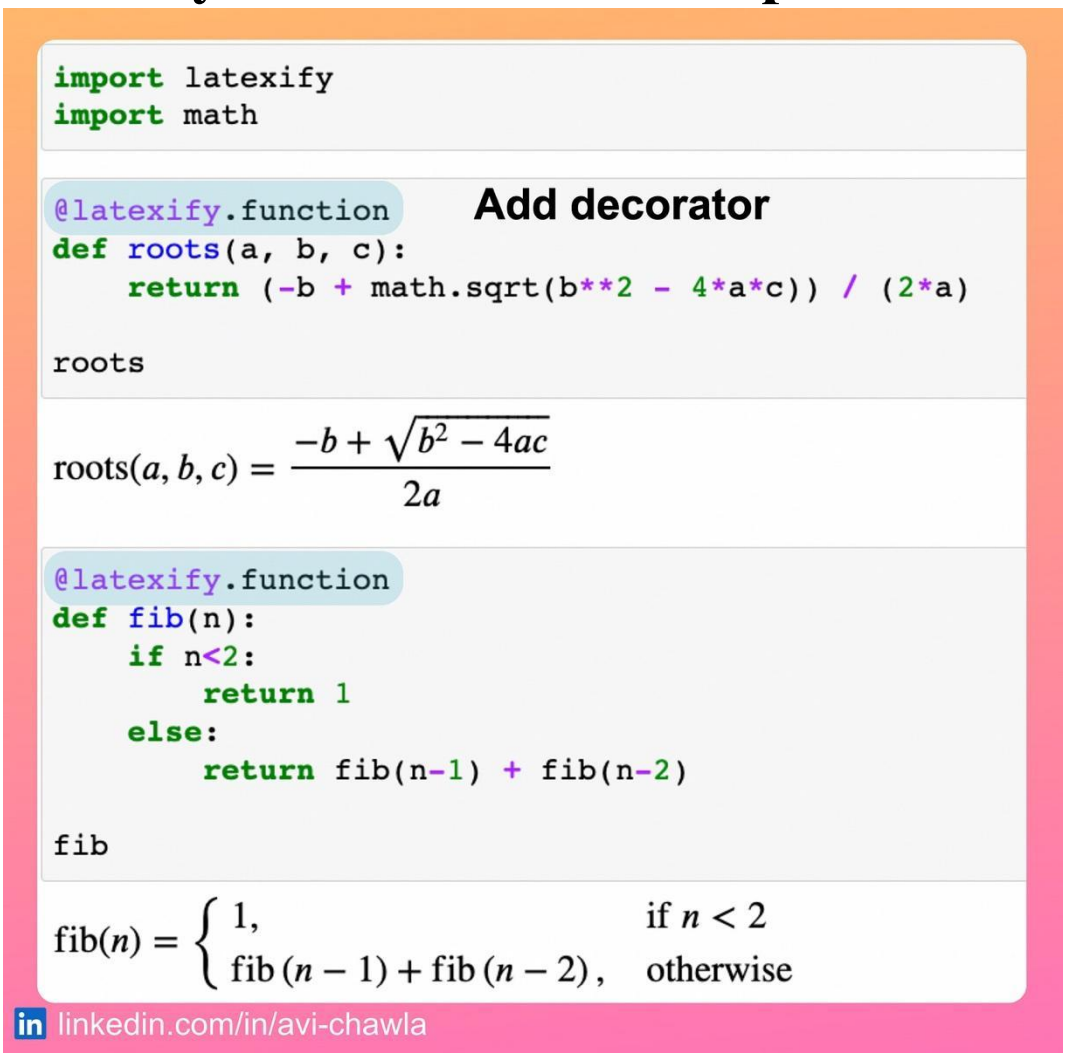
For instance, KMeans clusters the points purely based on locality from centroids. Thus, it can create wrong clusters when data points have arbitrary shapes.

Among the many possible alternatives is DBSCAN, which is a density-based clustering algorithm. Thus, it can identify clusters of arbitrary shape and size.

This makes it robust to data with non-spherical clusters and varying densities. Find more info in the comments.

Find more here: Sklearn Guide.

Converting Python To LaTeX Has Possibly Never Been So Simple

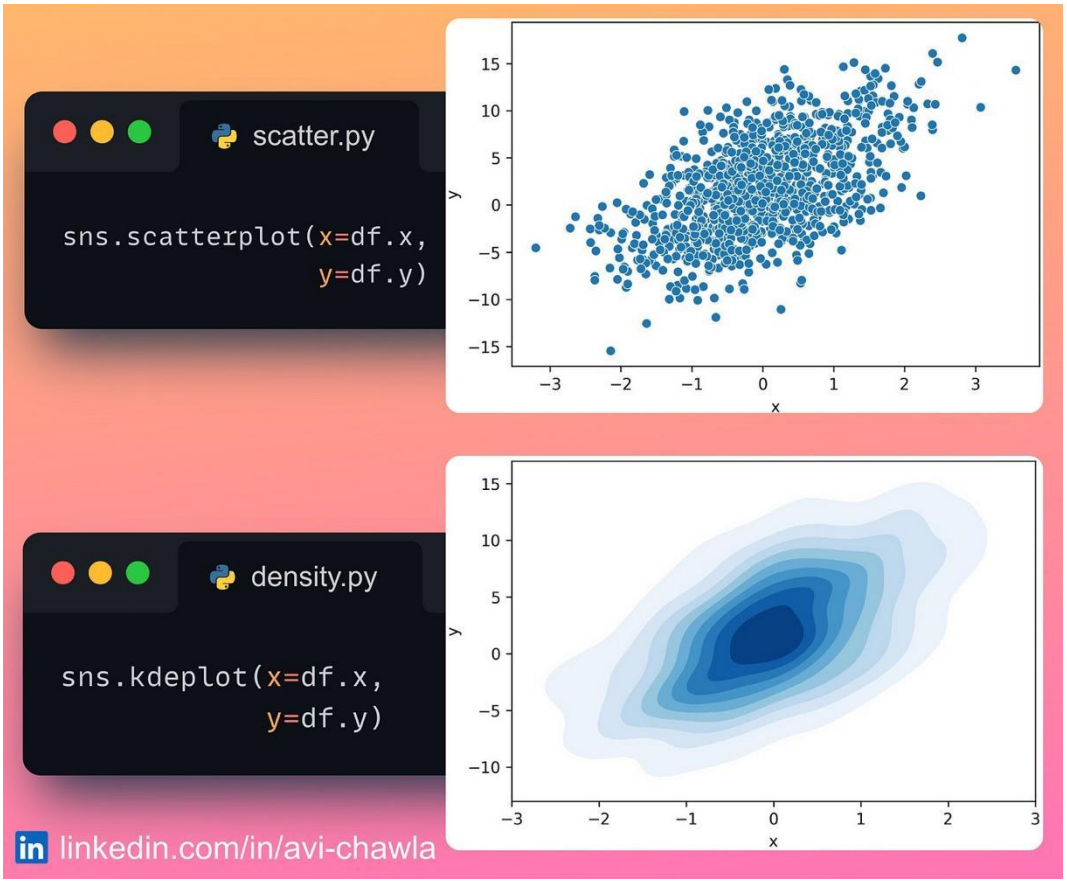


If you want to display python code and its output as LaTeX, try **latexify_py**. With this, you can print python code as a LaTeX expression and make your code more interpretable.

What's more, it can also generate LaTeX code for python code. This saves plenty of time and effort of manually writing the expressions in LaTeX.

Find more info here: Repository.

Density Plot As A Richer Alternative to Scatter Plot



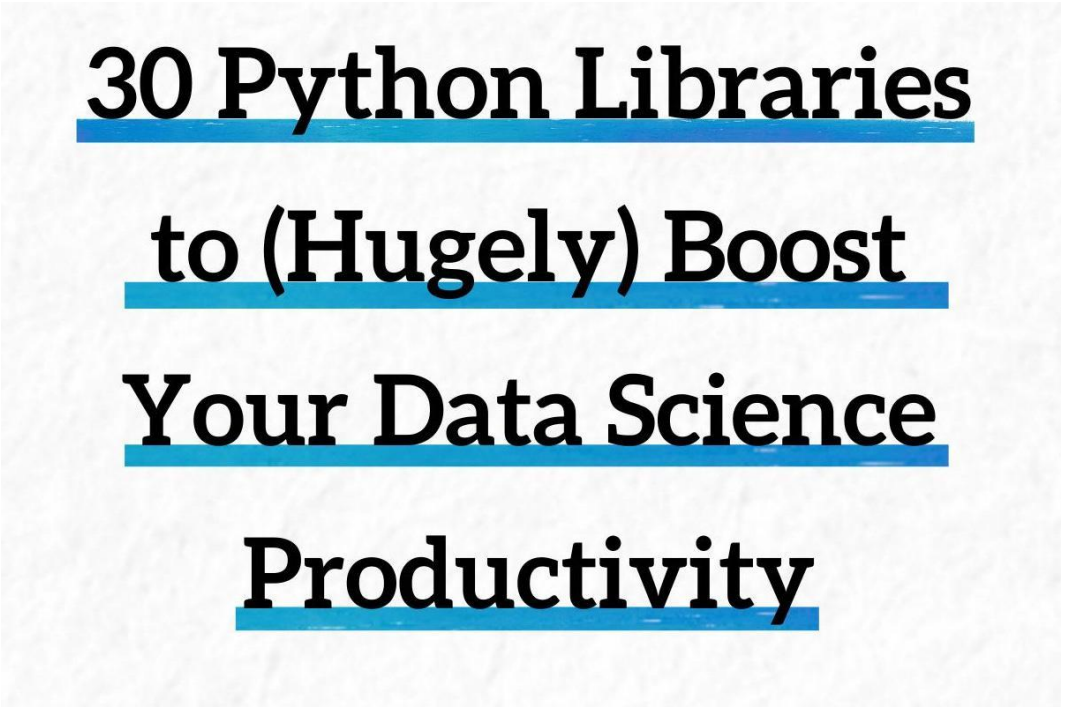
Scatter plots are extremely useful for visualizing two sets of numerical variables. But when you have, say, thousands of data points, scatter plots can get too dense to interpret.

A density plot can be a good choice in such cases. It depicts the distribution of points using colors (or contours). This makes it easy to identify regions of high and low density.

Moreover, it can easily reveal clusters of data points that might not be obvious in a scatter plot.

Read more: Docs.

30 Python Libraries to (Hugely) Boost Your Data Science Productivity

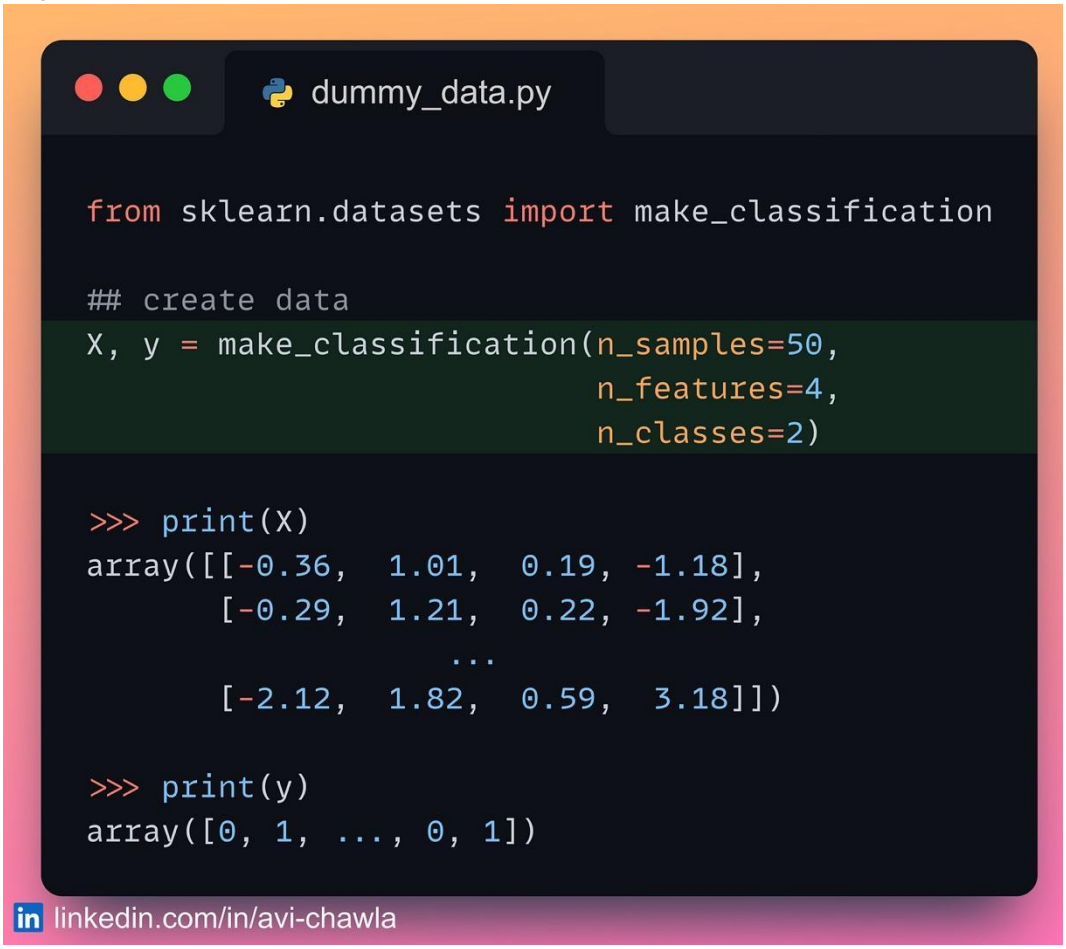


Here's a collection of 30 essential open-source data science libraries. Each has its own use case and enormous potential to skyrocket your data science skills.

I would love to know the ones you use.

Please find this full document in my LinkedIn post: **Post Link**.

Sklearn One-liner to Generate Synthetic Data



Often for testing/building a data pipeline, we may need some dummy data.

With Sklearn, you can easily create a dummy dataset for regression, classification, and clustering tasks.

More info here: Sklearn Docs.

Label Your Data With The Click Of A Button

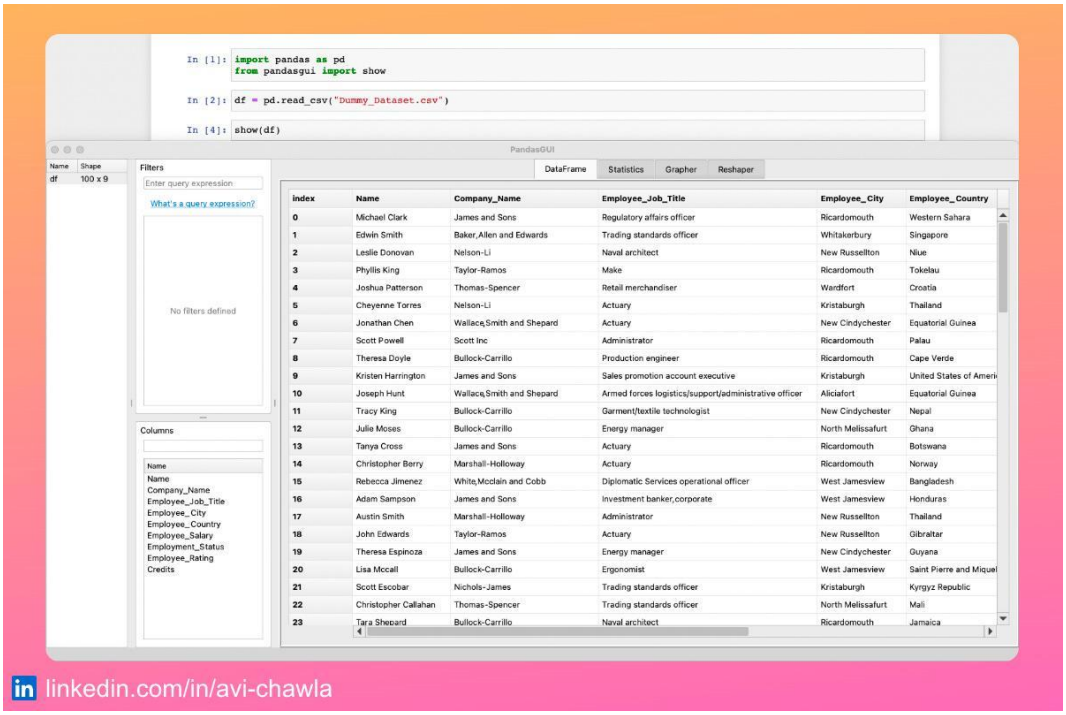
Often with unlabeled data, one may have to spend some time annotating/labeling it.

To do this quickly in a jupyter notebook, use **ipyannotate**. With this, you can annotate your data by simply clicking the corresponding button.

Read more: ipyannotate.

Watch a video version of this post on LinkedIn: Post Link.

Analyze A Pandas DataFrame Without Code

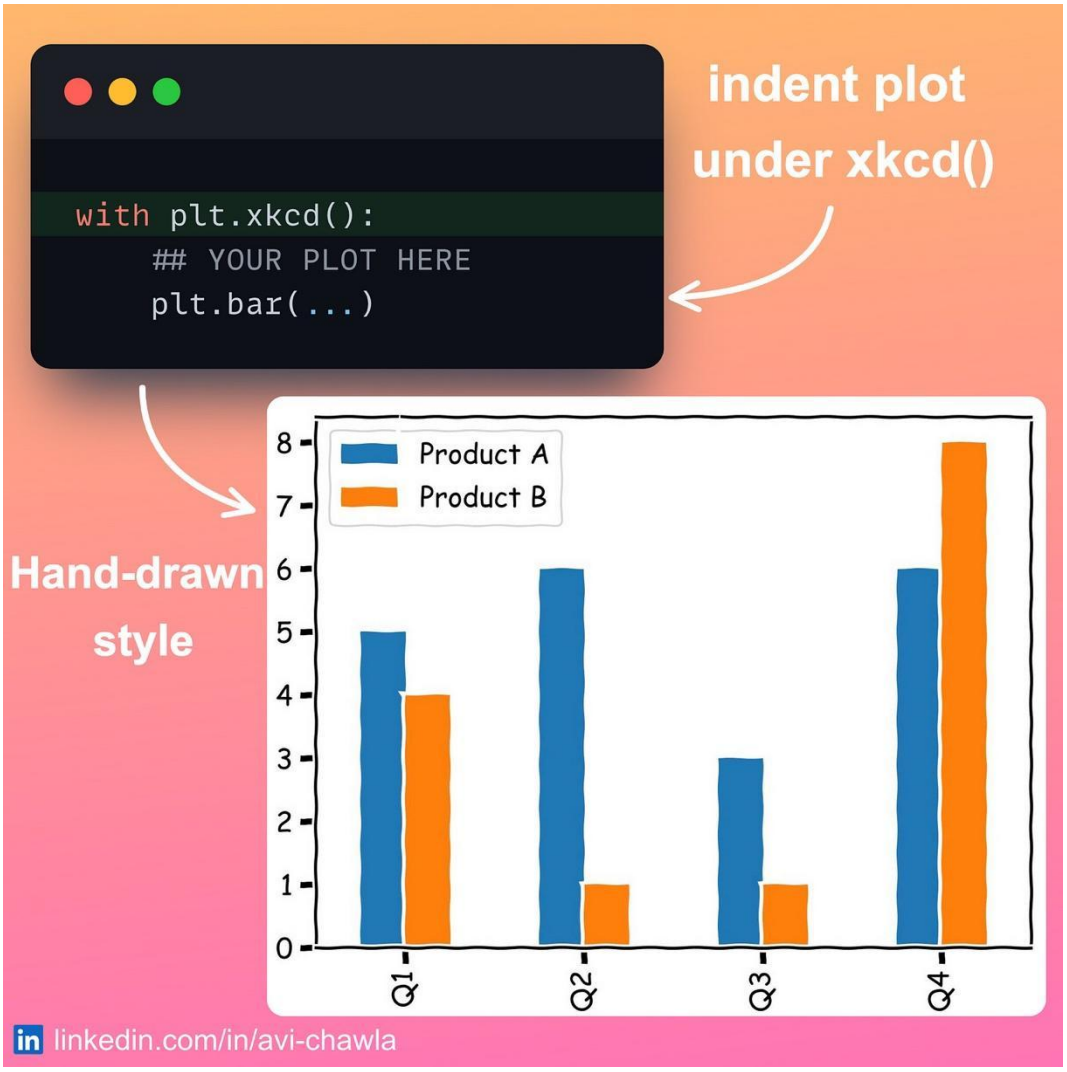


If you want to analyze your dataframe in a GUI-based application, try Pandas GUI. It provides an elegant GUI for viewing, filtering, sorting, describing tabular datasets, etc.

What's more, using its intuitive drag-and-drop functionality, you can easily create a variety of plots and export them as code.

Watch a video version of this post on LinkedIn: Post Link.

Python One-Liner To Create Sketchy Hand-drawn Plots



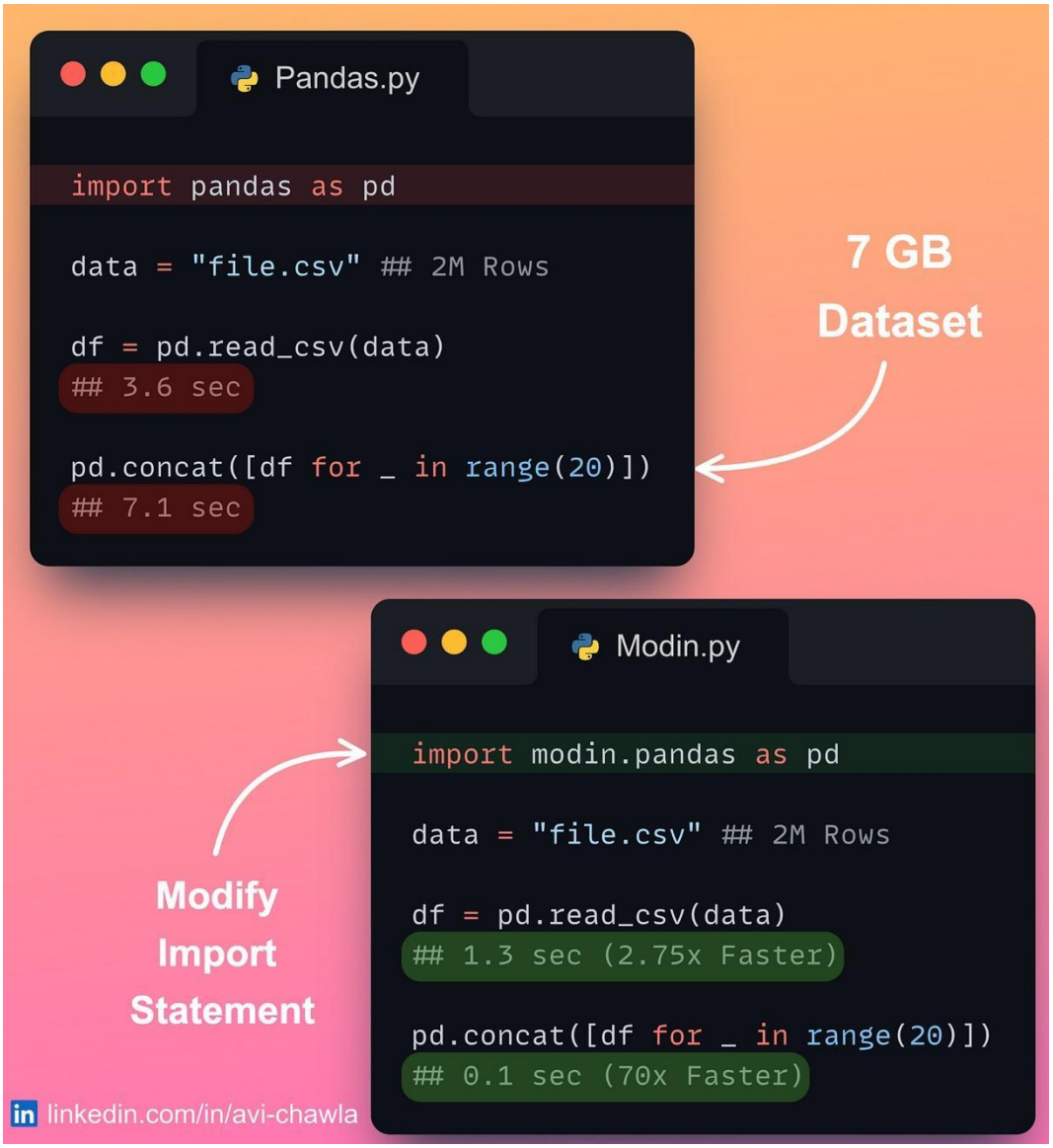
xkcd comic is known for its informal and humorous style, as well as its stick figures and simple drawings.

Creating such visually appealing hand-drawn plots is pretty simple using matplotlib. Just indent the code in a **plt.xkcd**() context to display them in comic style.

Do note that this style is just used to improve the aesthetics of a plot through hand-drawn effects. However, it is not recommended for formal presentations, publications, etc.

Read more: Docs.

70x Faster Pandas By Changing Just One Line of Code

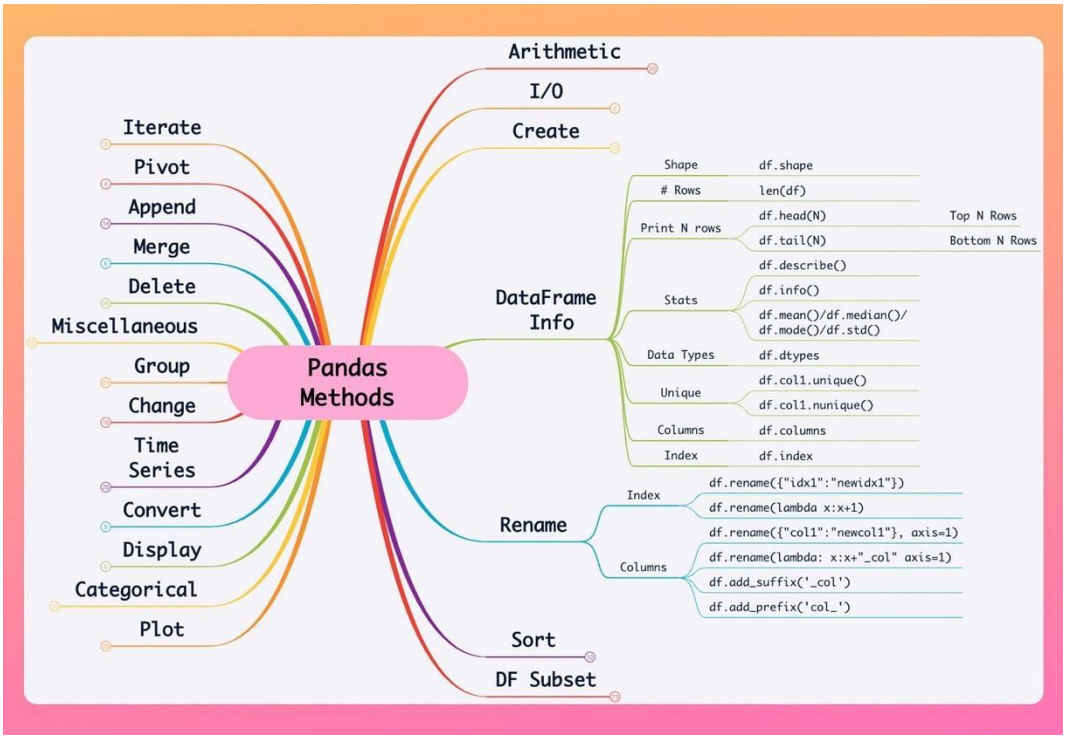


It is challenging to work on large datasets in Pandas. This, at times, requires plenty of optimization and can get tedious as the dataset grows further.

Instead, try Modin. It delivers instant improvements with no extra effort. Change the import statement and use it like the Pandas API, with significant speedups. Find more info in the comments.

Read more: Modin Guide.

An Interactive Guide To Master Pandas In One Go



Here's a mind map illustrating Pandas Methods on one page. How many do you know :)

- Load/Save
- DataFrame info
- Filter
- Merge
- Time-series
- Plot
- and many more, in a single map.

Find the full diagram here: Pandas Mind Map.

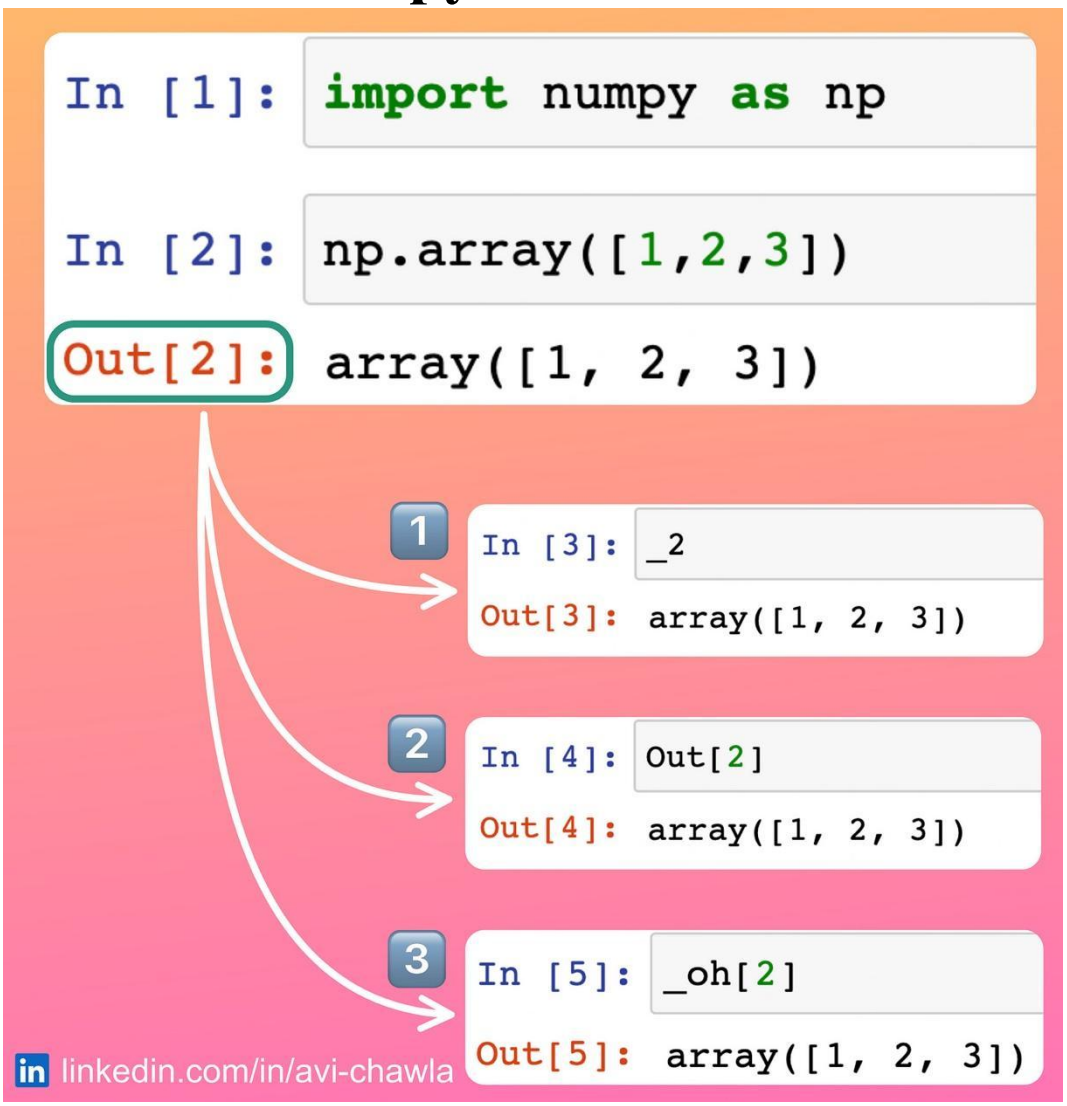
Make Dot Notation More Powerful in Python

Dot notation offers a simple and elegant way to access and modify the attributes of an instance.

Yet, it is a good programming practice to use the getter and setter method for such purposes. This is because it offers more control over how attributes are accessed/changed.

To leverage both in Python, use the **@property** decorator. As a result, you can use the dot notation and still have explicit control over how attributes are accessed/set.

The Coolest Jupyter Notebook Hack

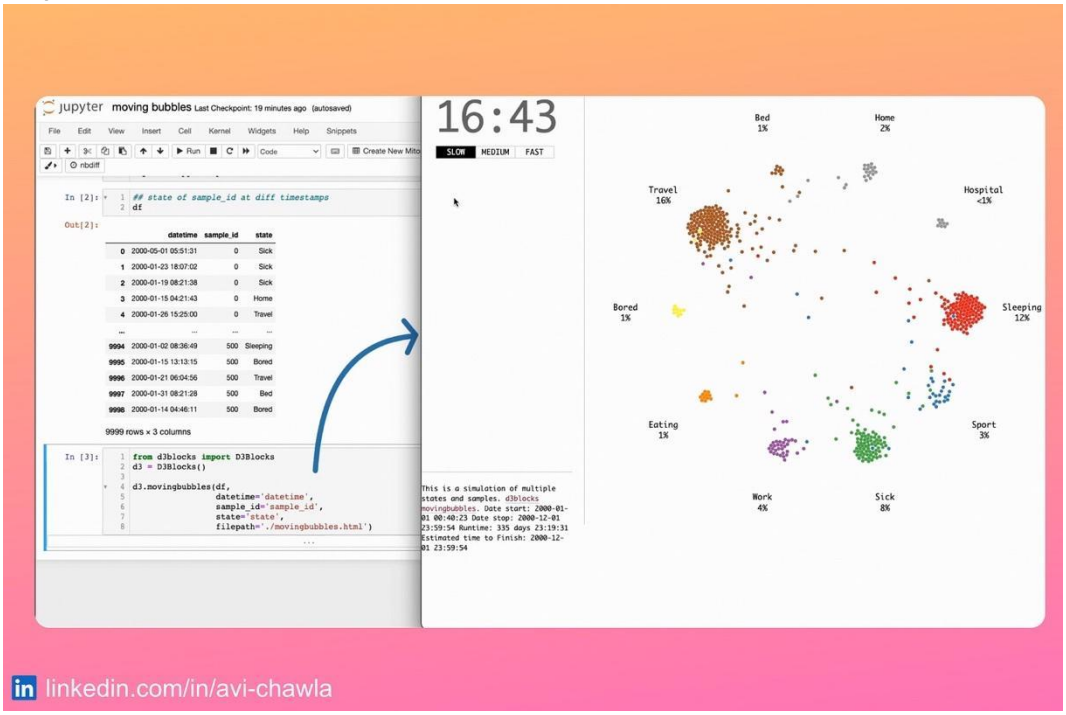


Have you ever forgotten to assign the results to a variable in Jupyter? Rather than recomputing the result by rerunning the cell, here are three ways to retrieve the output.

1) Use the underscore followed by the output-cell-index.

2/3) Use the **Out** or **_oh** dict and specify the output-cell-index as the key.

Create a Moving Bubbles Chart in Python

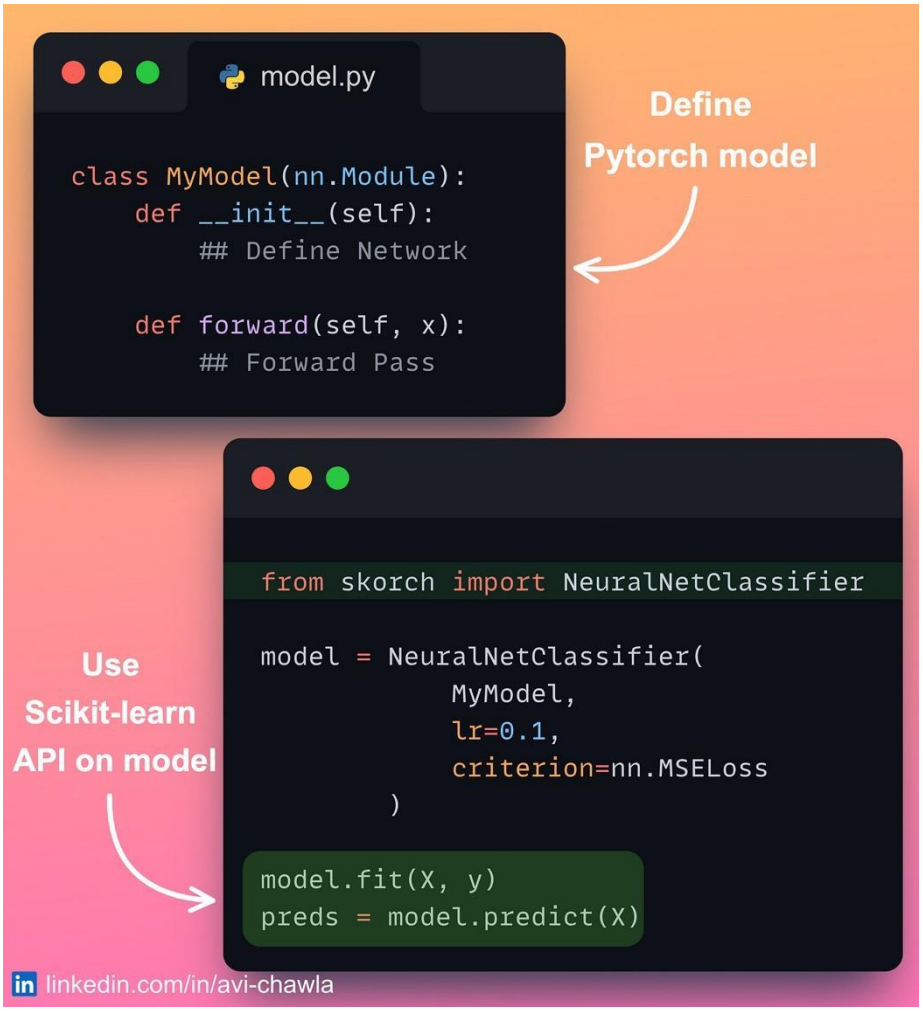


Ever seen one of those moving points charts? Here's how you can create one in Python in just three lines of code.

A Moving Bubbles chart is an elegant way to depict the movements of entities across time. Using this, we can easily determine when clusters appear in our data and at what state(s).

To create one, you can use "d3blocks". Its input should be a DataFrame. A row represents the state of a sample at a particular timestamp.

Skorch: Use Scikit-learn API on PyTorch Models



skorch is a high-level library for PyTorch that provides full Scikit-learn compatibility. In other words, it combines the power of PyTorch with the elegance of sklearn.

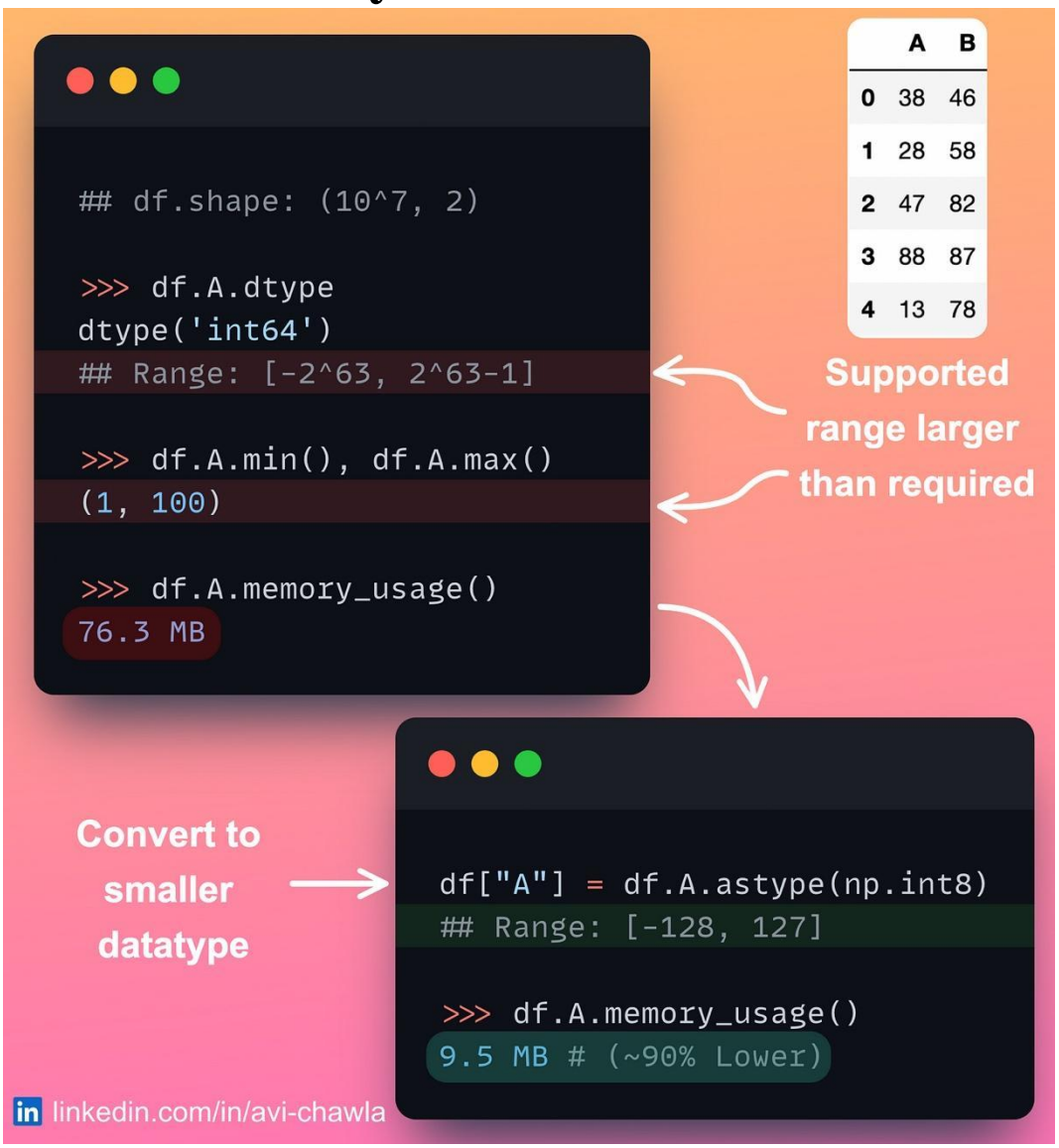
Thus, you can train PyTorch models in a way similar to Scikit-learn, using functions such as fit, predict, score, etc.

Using skorch, you can also put a PyTorch model in the sklearn pipeline, and many more.

Overall, it aims at being as flexible as PyTorch while having a clean interface as sklearn.

Read more: Documentation.

Reduce Memory Usage Of A Pandas DataFrame By 90%

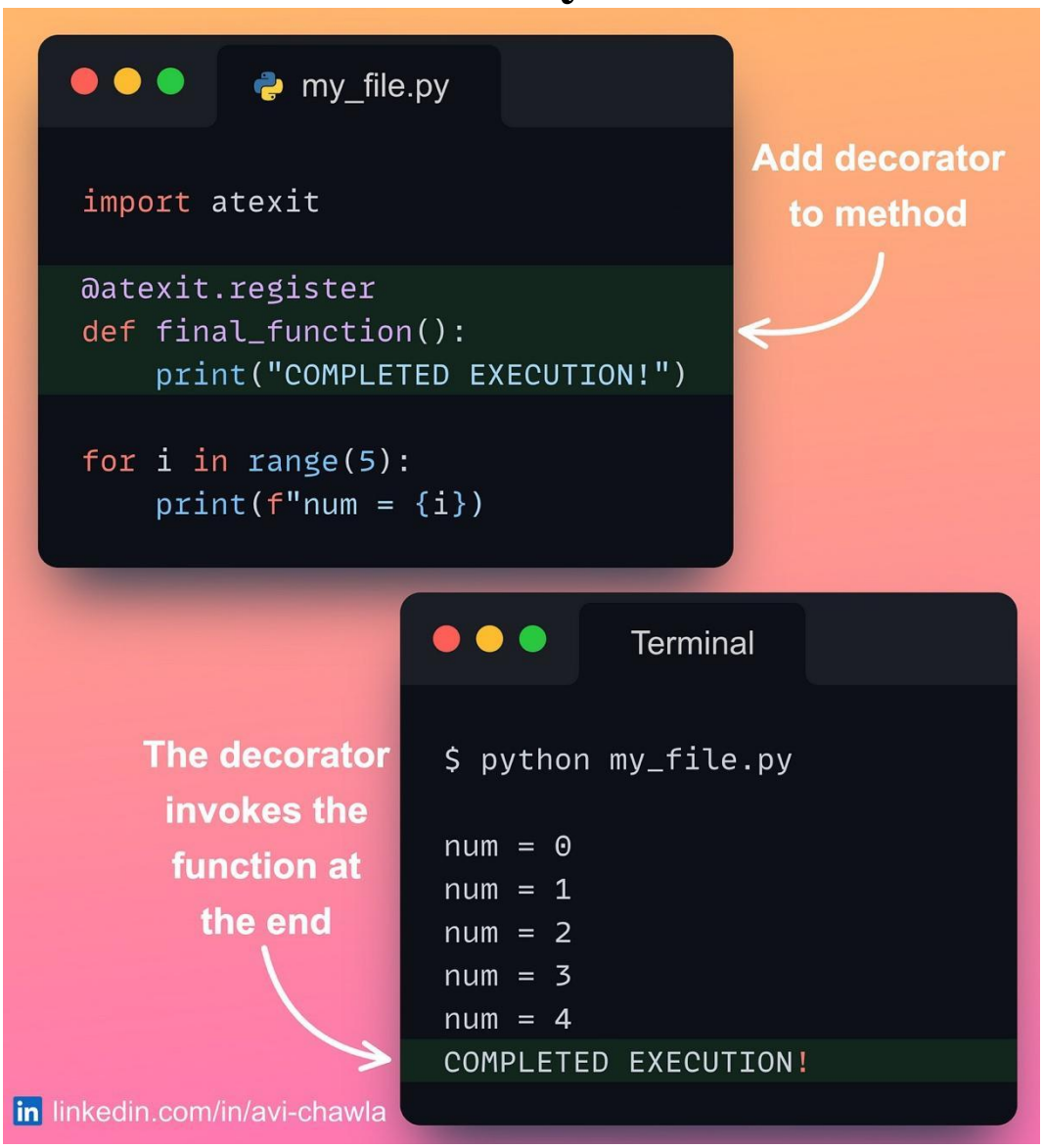


By default, Pandas always assigns the highest memory datatype to its columns. For instance, an integer-valued column always gets the int64 datatype, irrespective of its range.

To reduce memory usage, represent it using an optimized datatype, which is enough to span the range of values in your columns.

Read this blog for more info. It details many techniques to optimize the memory usage of a Pandas DataFrame.

An Elegant Way To Perform Shutdown Tasks in Python



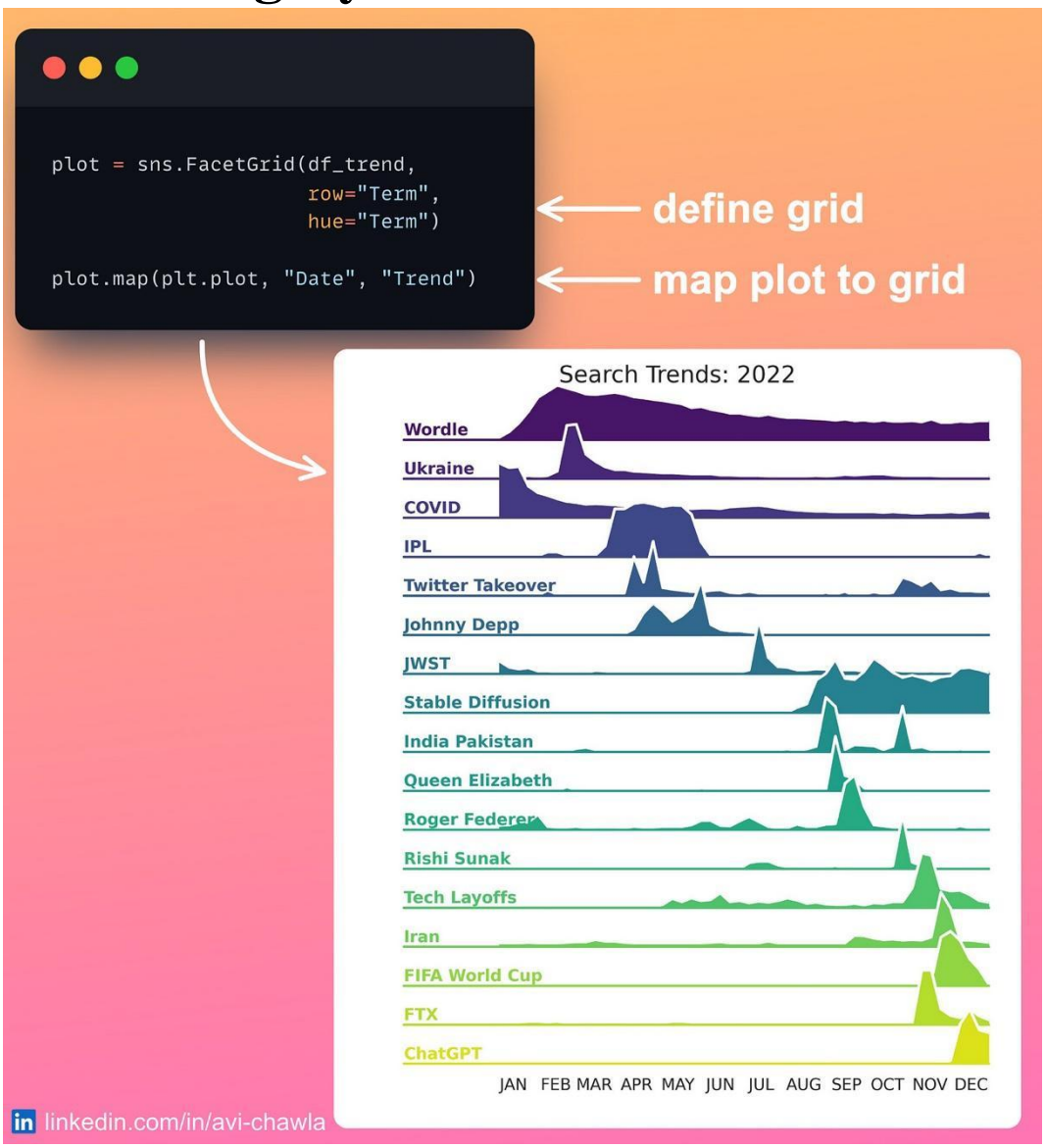
Often towards the end of a program's execution, we run a few basic tasks such as saving objects, printing logs, etc.

To invoke a method right before the interpreter is shutting down, decorate it with the **@atexit.register** decorator.

The good thing is that it works even if the program gets terminated unexpectedly. Thus, you can use this method to save the state of the program or print any necessary details before it stops.

Read more: Documentation.

Visualizing Google Search Trends of 2022 using Python



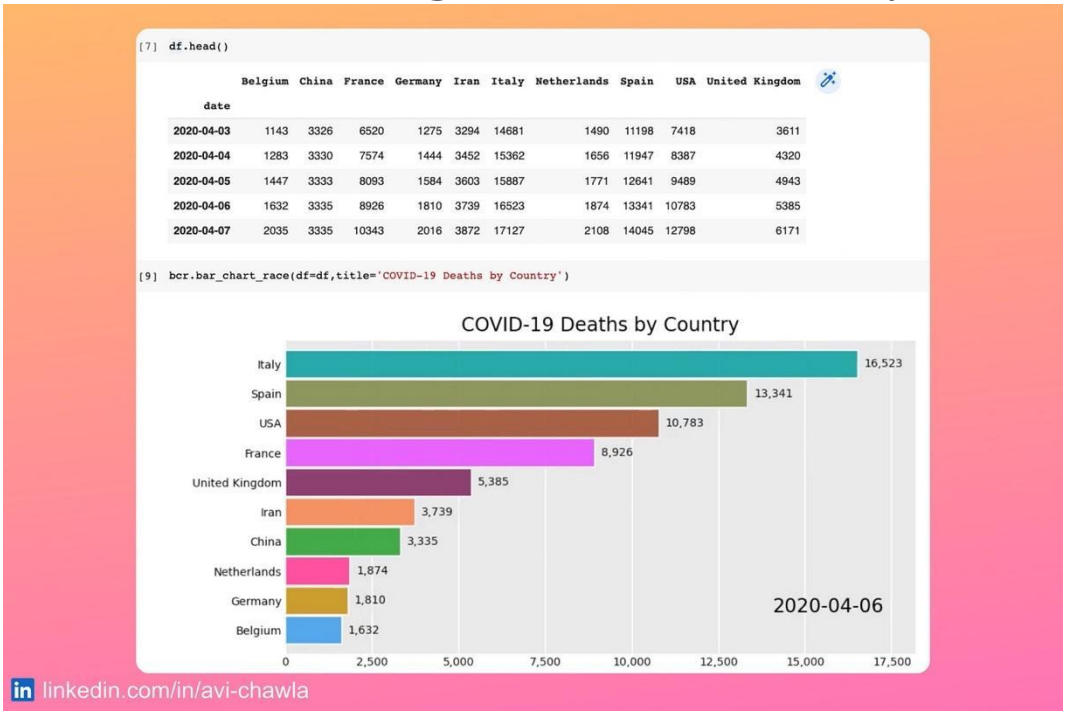
If your data has many groups, visualizing their distribution together can create cluttered plots. This makes it difficult to visualize the underlying patterns.

Instead, consider plotting the distribution across individual groups using FacetGrid. This allows you to compare the distributions of multiple groups side by side and see how they vary.

As shown above, a FacetGrid allows us to clearly see how different search terms trended across 2022.

P.S. I used the year-in-search-trends repository to fetch the trend data.

Create A Racing Bar Chart In Python



Ever seen one of those racing bar charts? Here's how you can create one in Python in just two lines of code.

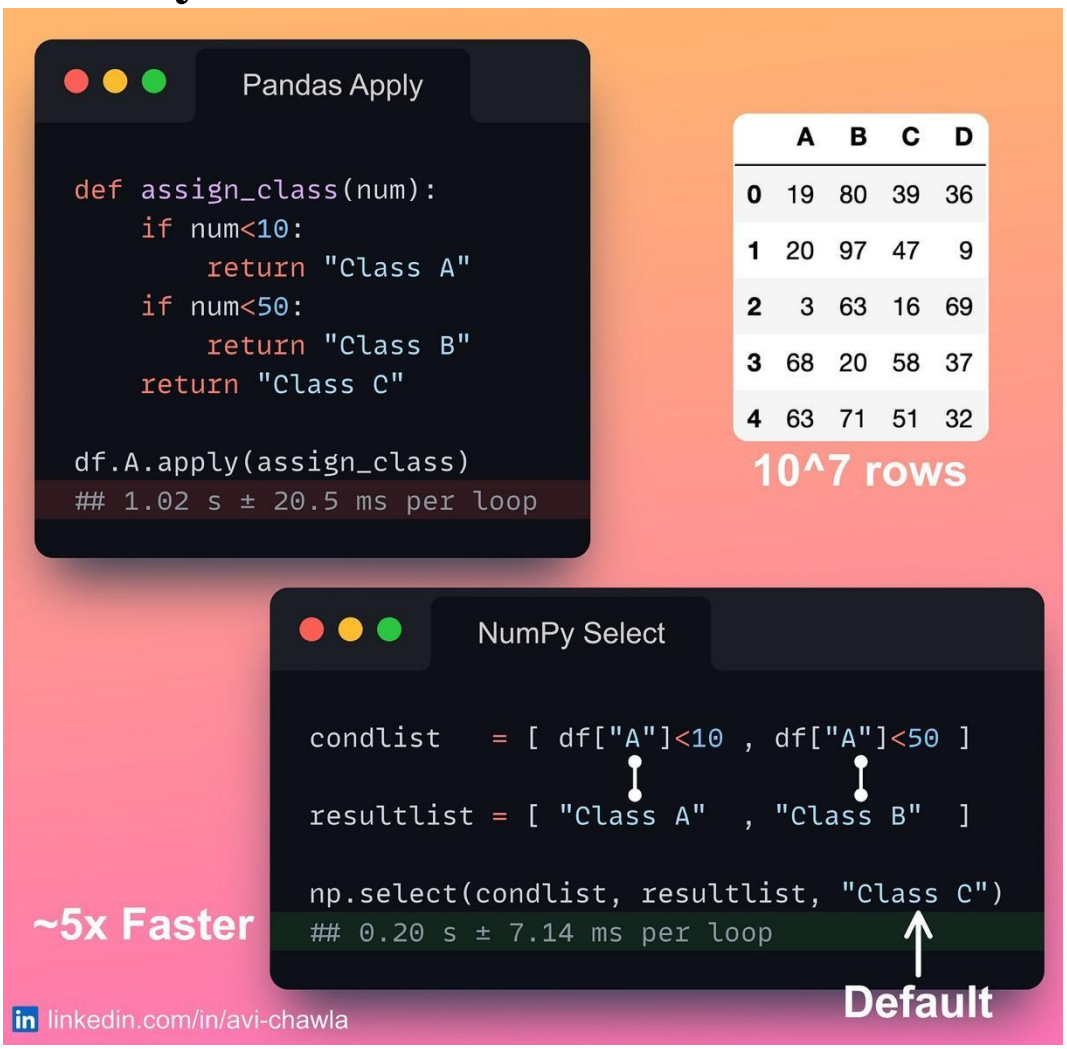
A racing bar chart is typically used to depict the progress of multiple values over time.

To create one, you can use the "**bar-chart-race**" library.

Its input should be a Pandas DataFrame where every row represents a single timestamp. The column holds the corresponding values for a particular category.

Read more: Documentation.

Speed-up Pandas Apply 5x with NumPy



While creating conditional columns in Pandas, we tend to use the **apply**() method almost all the time.

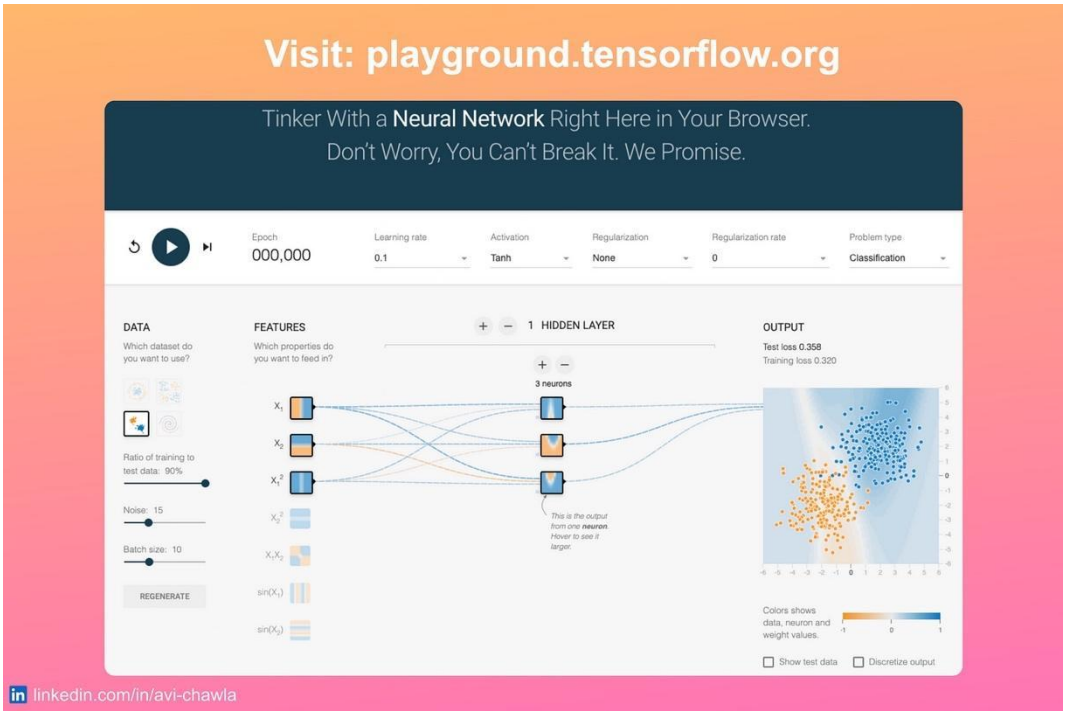
However, **apply**() in Pandas is nothing but a glorified for-loop. As a result, it misses the whole point of vectorization.

Instead, you should use the **np.select**() method to create conditional columns. It does the same job but is extremely fast.

The conditions and the corresponding results are passed as the first two arguments. The last argument is the default result.

Read more here: NumPy docs.

A No-Code Online Tool To Explore and Understand Neural Networks



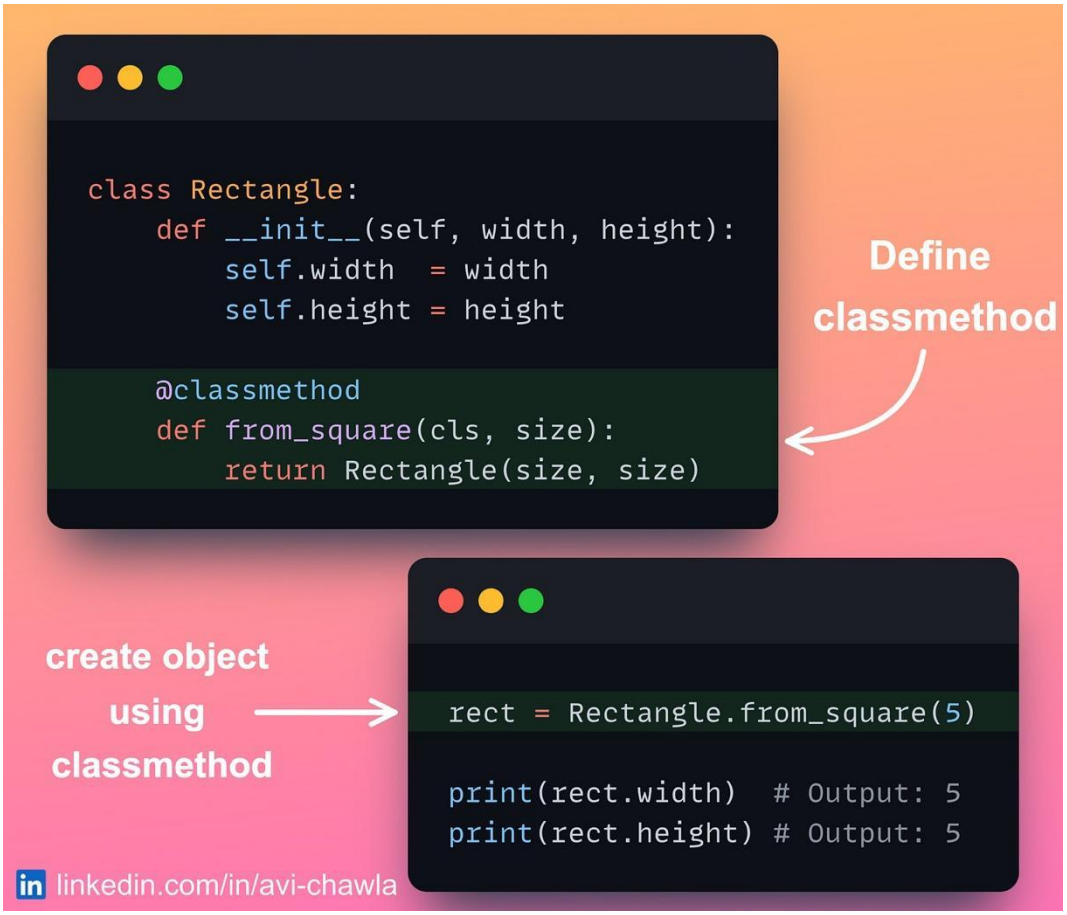
Neural networks can be intimidating for beginners. Also, experimenting programmatically does not provide enough intuitive understanding about them.

Instead, try TensorFlow Playground. Its elegant UI allows you to build, train and visualize neural networks without any code.

With a few clicks, one can see how neural networks work and how different hyperparameters affect their performance. This makes it especially useful for beginners.

Try here: Tensorflow Playground.

What Are Class Methods and When To Use Them?



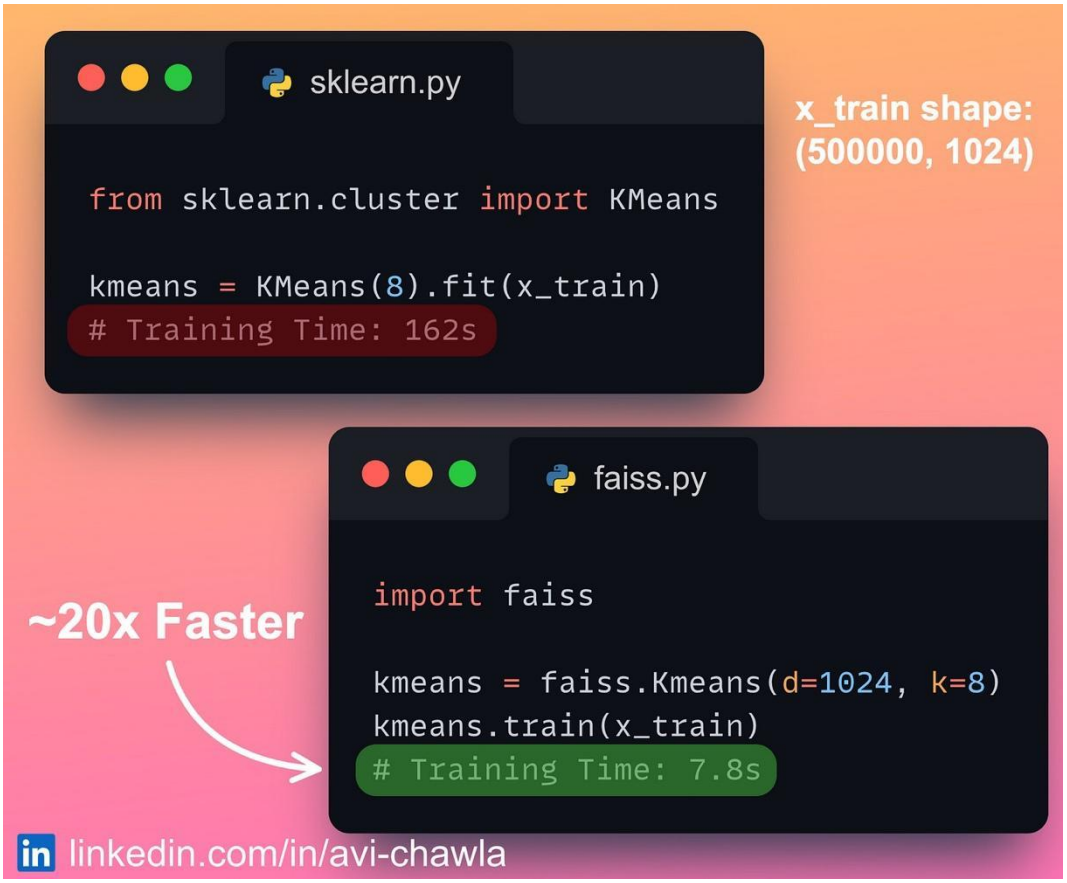
Class methods, as the name suggests, are bound to the class and not the instances of a class. They are especially useful for providing an alternative interface for creating instances.

Moreover, they can be also used to define utility functions that are related to the class rather than its instances.

For instance, one can define a class method that returns a list of all instances of the class. Another use could be to calculate a class-level statistic based on the instances.

To define a class method in Python, use the **@classmethod** decorator. As a result, this method can be called directly using the name of the class.

Make Sklearn KMeans 20x times faster



The KMeans algorithm is commonly used to cluster unlabeled data. But with large datasets, scikit-learn takes plenty of time to train and predict.

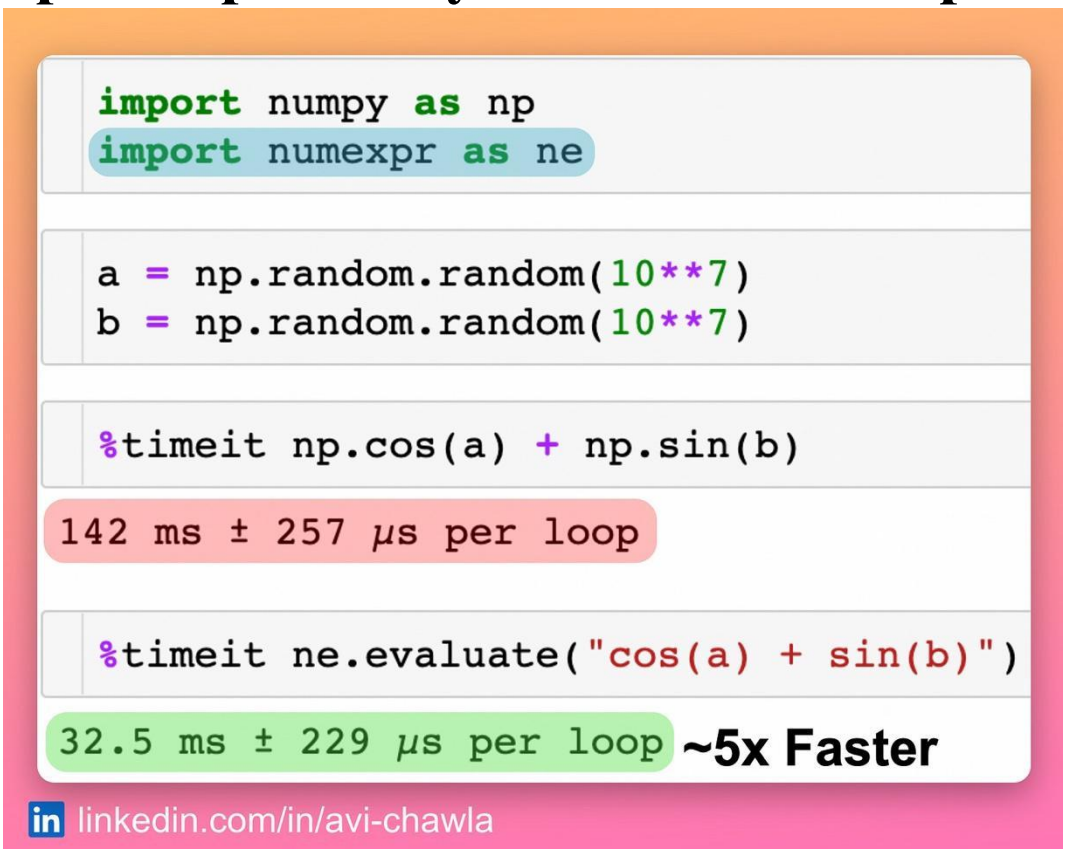
To speed-up KMeans, use Faiss by Facebook AI Research. It provides faster nearest-neighbor search and clustering.

Faiss uses "Inverted Index", an optimized data structure to store and index the data points. This makes performing clustering extremely efficient.

Additionally, Faiss provides parallelization and GPU support, which further improves the performance of its clustering algorithms.

Read more: GitHub.

Speed-up NumPy 20x with Numexpr



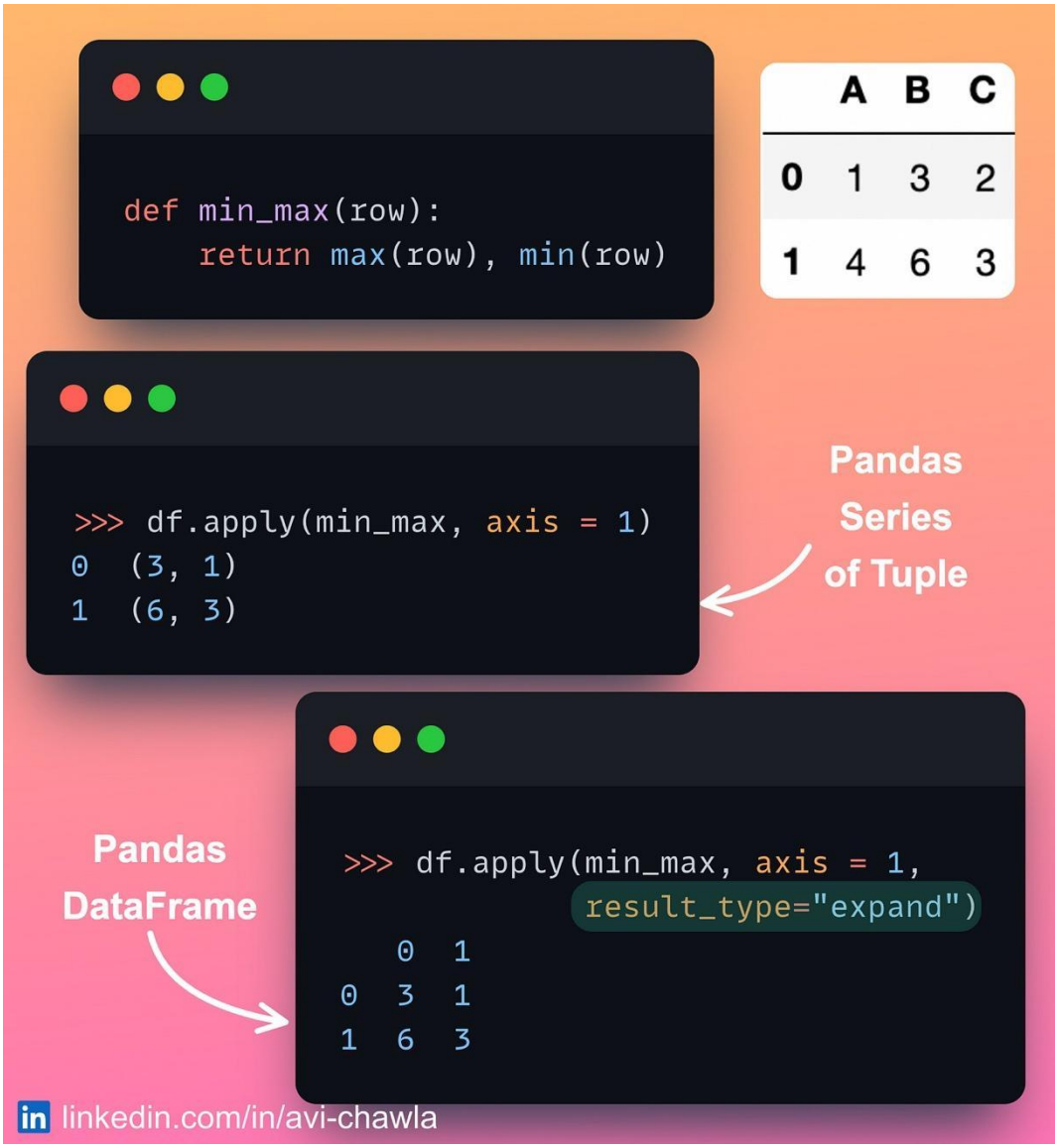
Numpy already offers fast and optimized vectorized operations. Yet, it does not support parallelism. This provides further scope for improving the run-time of NumPy.

To do so, use Numexpr. It allows you to speed up numerical computations with multi-threading and just-in-time compilation.

Depending upon the complexity of the expression, the speed-ups can range from 0.95x and 20x. Typically, it is expected to be 2x-5x.

Read more: Documentation.

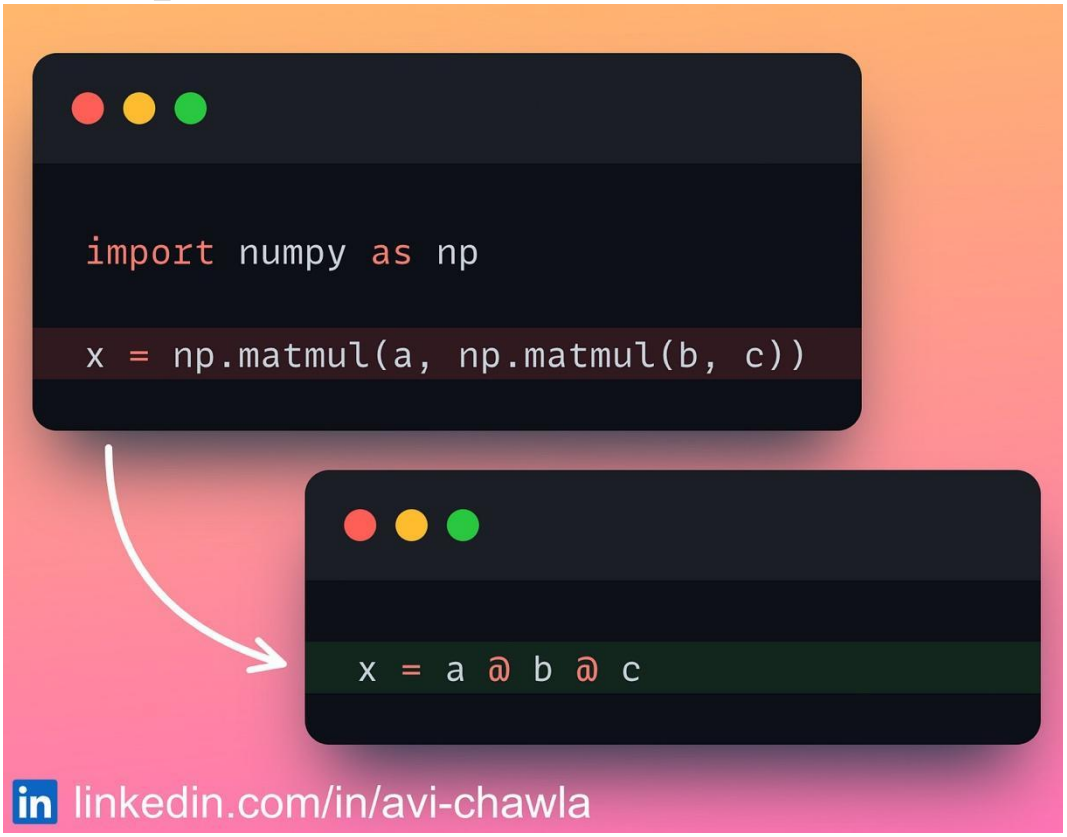
A Lesser-Known Feature of Apply Method In Pandas



After applying a method on a DataFrame, we often return multiple values as a tuple. This requires additional steps to project it back as separate columns.

Instead, with the result_type argument, you can control the shape and output type. As desired, the output can be either a DataFrame or a Series.

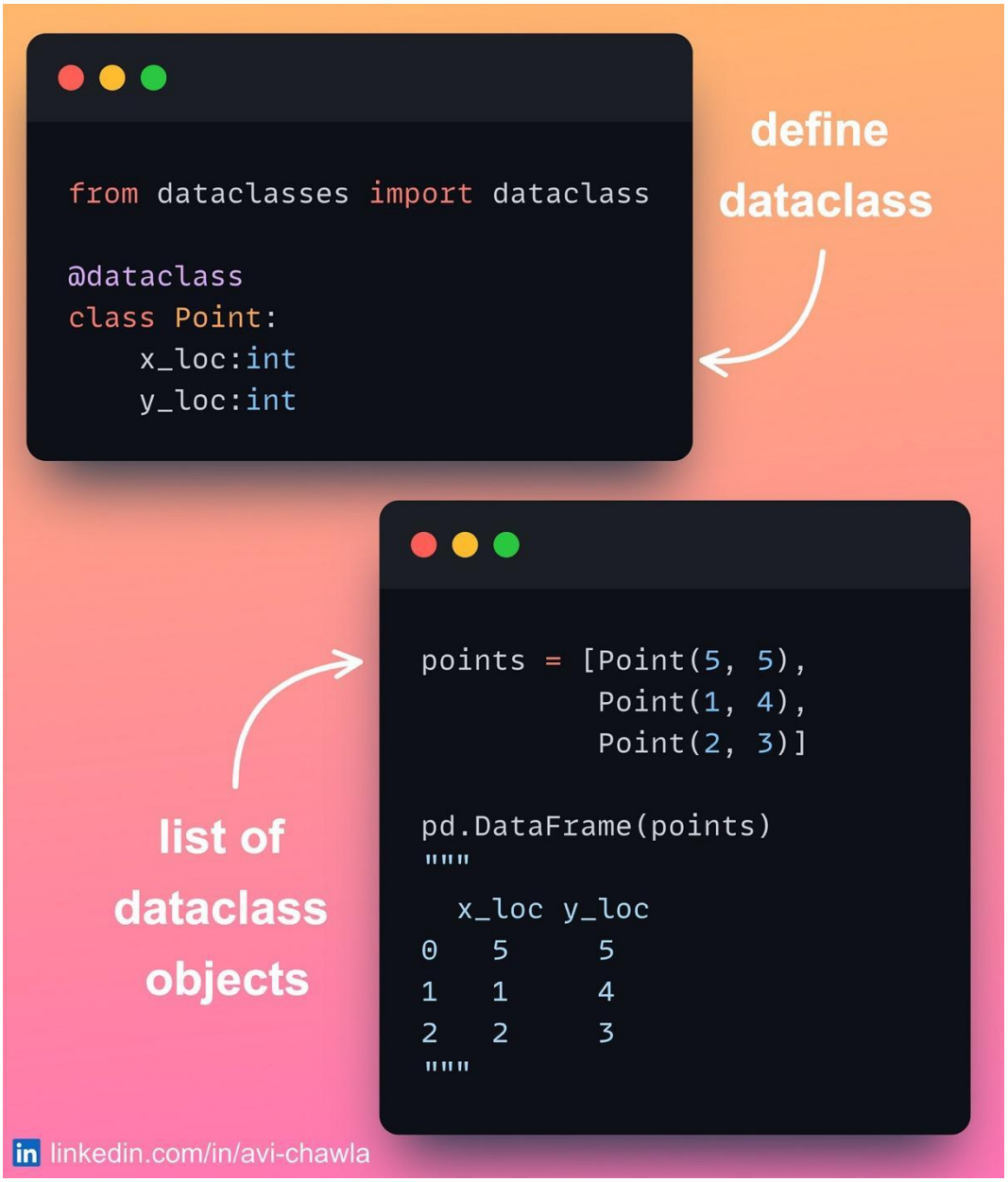
An Elegant Way To Perform Matrix Multiplication



Matrix multiplication is a common operation in machine learning. Yet, chaining repeated multiplications using **matmul** function makes the code cluttered and unreadable.

If you are using NumPy, you can instead use the @ operator to do the same.

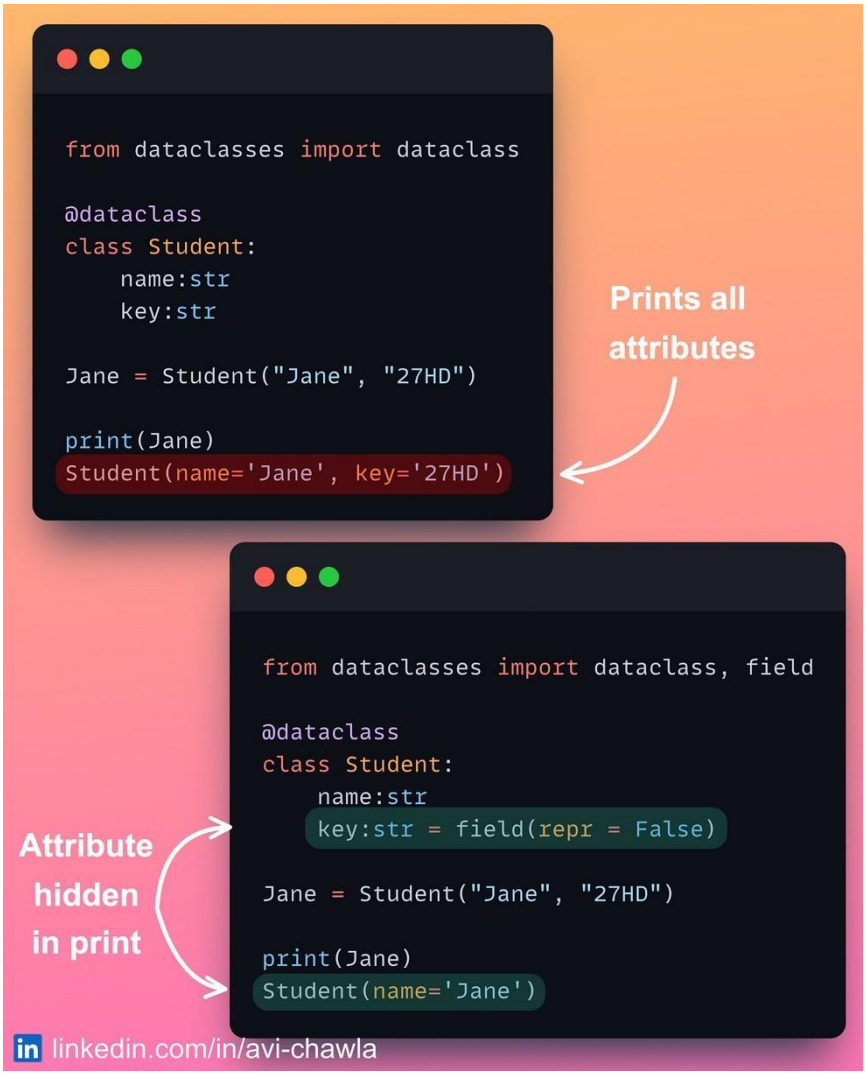
Create Pandas DataFrame from Dataclass



A Pandas DataFrame is often created from a Python list, dictionary, by reading files, etc. However, did you know you can also create a DataFrame from a Dataclass?

The image demonstrates how you can create a DataFrame from a list of dataclass objects.

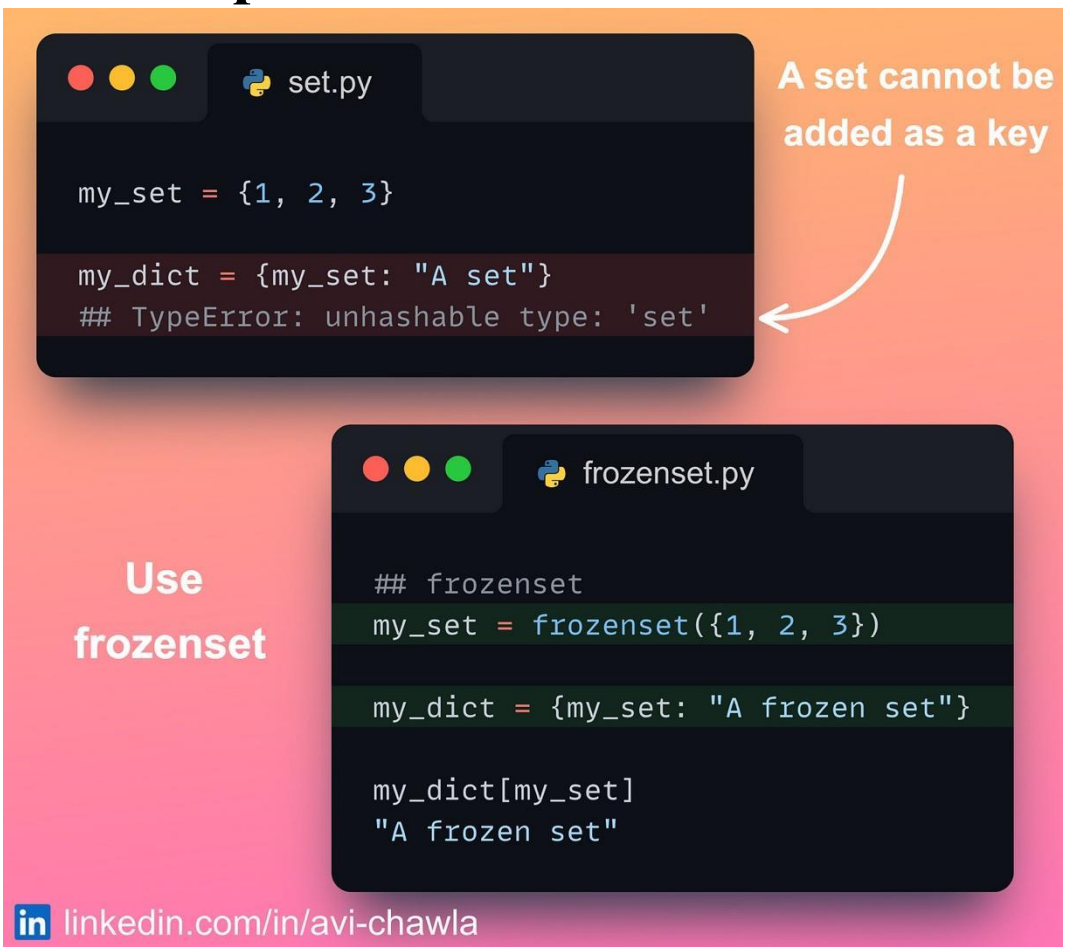
Hide Attributes While Printing A Dataclass Object



By default, a dataclass prints all the attributes of an object declared during its initialization.

But if you want to hide some specific attributes, declare **repr=False** in its field, as shown above.

List : Tuple :: Set : ?

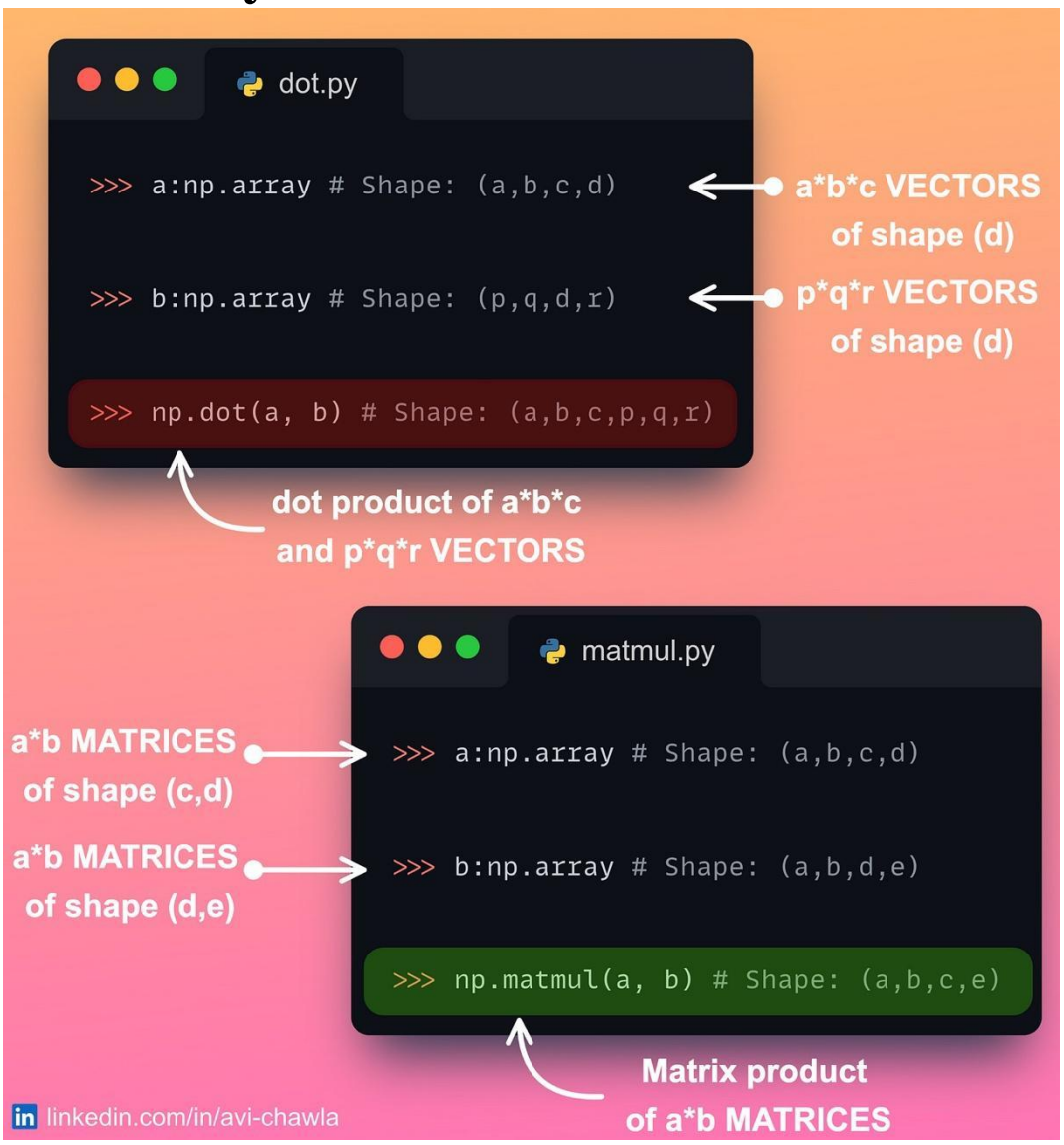


Dictionaries in Python require their keys to be immutable. As a result, a set cannot be used as keys as it is mutable.

Yet, if you want to use a set, consider declaring it as a frozenset.

It is an immutable set, meaning its elements cannot be changed after it is created. Therefore, they can be safely used as a dictionary's key.

Difference Between Dot and Matmul in NumPy

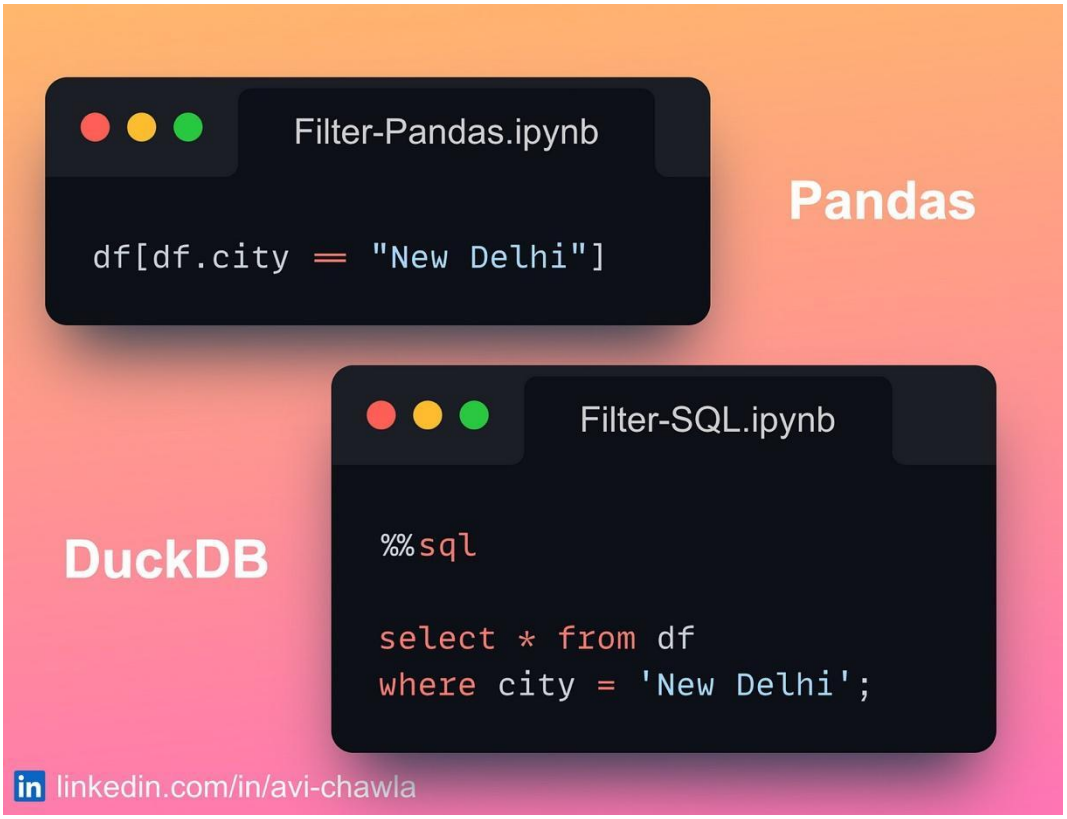


The **np.matmul**() and **np.dot**() methods produce the same output for 2D (and 1D) arrays. This makes many believe that they are the same and can be used interchangeably, but that is not true.

The **np.dot**() method revolves around individual vectors (or 1D arrays). Thus, it computes the dot product of ALL vector pairs in the two inputs.

The **np.matmul**() method, as the name suggests, is meant for matrices. Thus, it computes the matrix multiplication of corresponding matrices in the two inputs.

Run SQL in Jupyter To Analyze A Pandas DataFrame

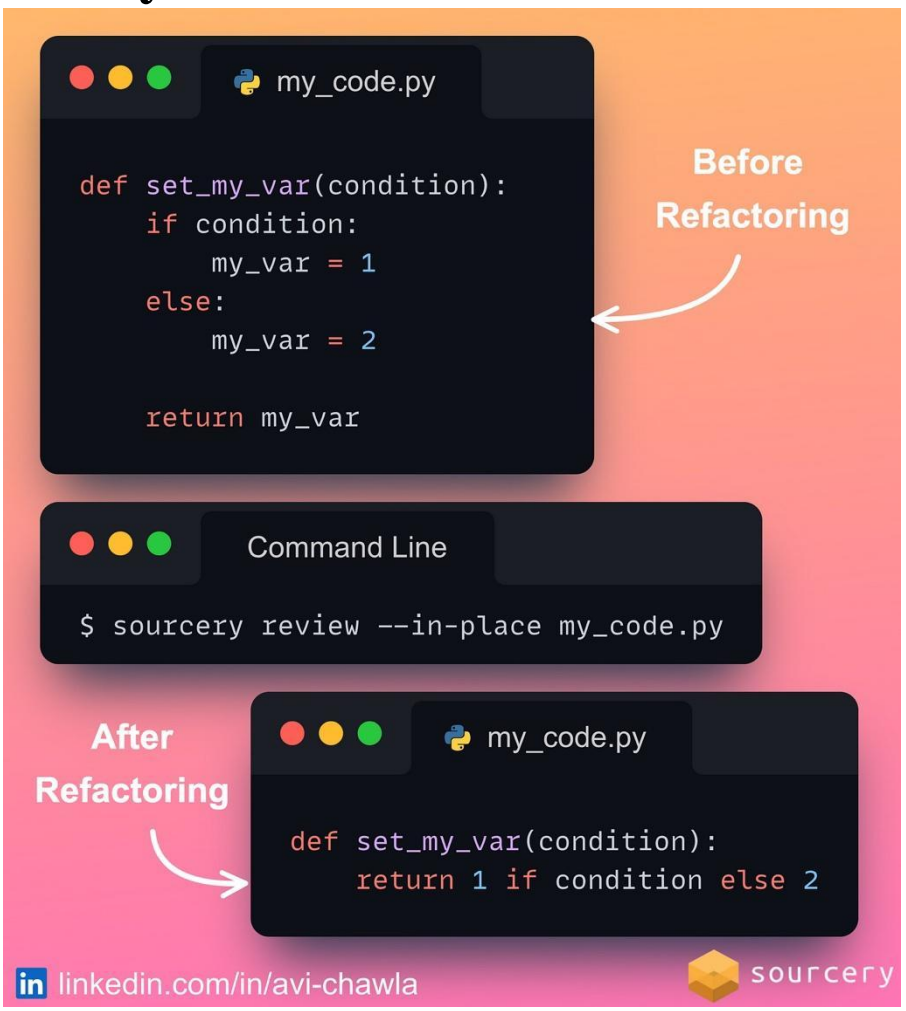


Pandas already provides a wide range of functionalities to analyze tabular data. Yet, there might be situations when one feels comfortable using SQL over Python.

Using DuckDB, you can analyze a Pandas DataFrame with SQL syntax in Jupyter, without any significant run-time difference.

Read the guide here to get started: Docs.

Automated Code Refactoring With Sourcery



Refactoring codebase is an important yet time-consuming task. Moreover, at times, one might unknowingly introduce errors during refactoring.

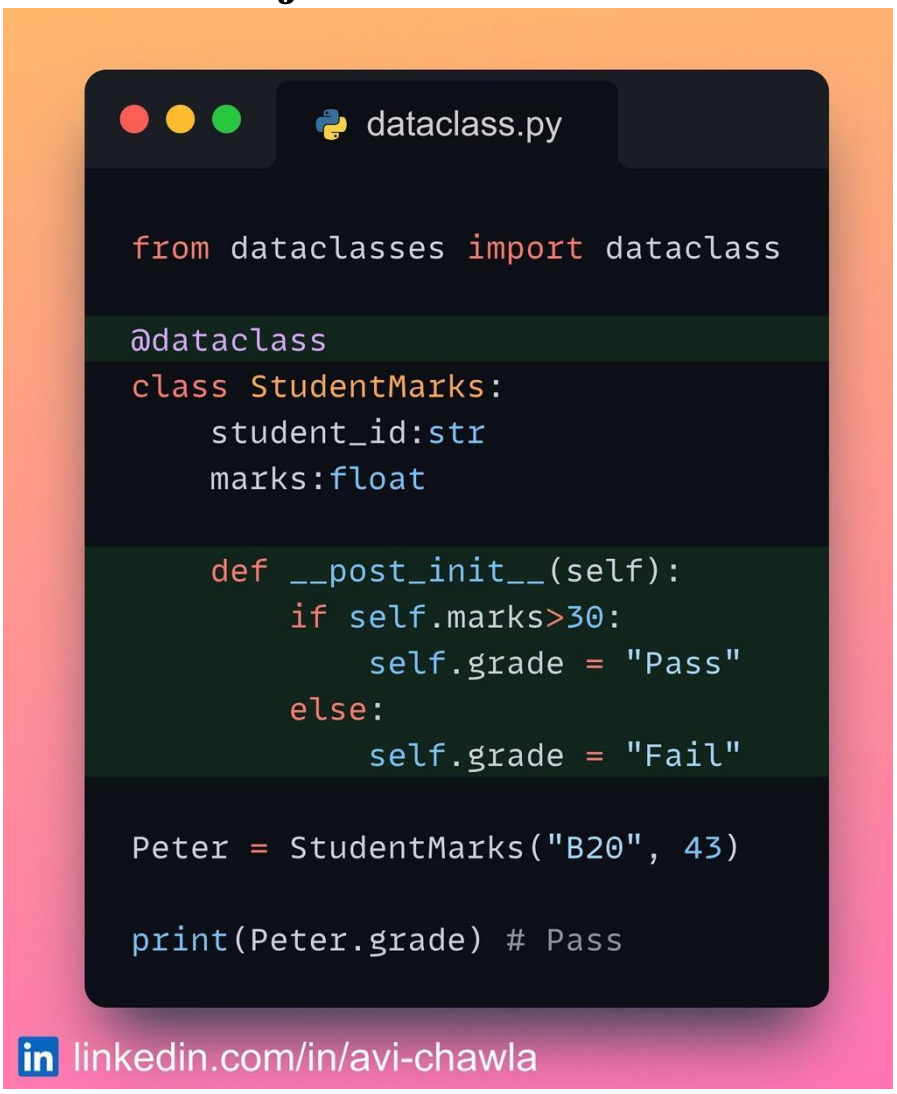
This takes additional time for testing and gets tedious with more refactoring, especially when the codebase is big.

Rather than following this approach, use Sourcery. It's an automated refactoring tool that makes your code elegant, concise, and Pythonic in no time.

With Sourcery, you can refactor code in many ways. For instance, you can refactor scripts through the command line, as an IDE plugin in VS Code and PyCharm, etc.

Read my full blog on Sourcery here: Medium.

__Post_init__: Add Attributes To A Dataclass Object Post Initialization

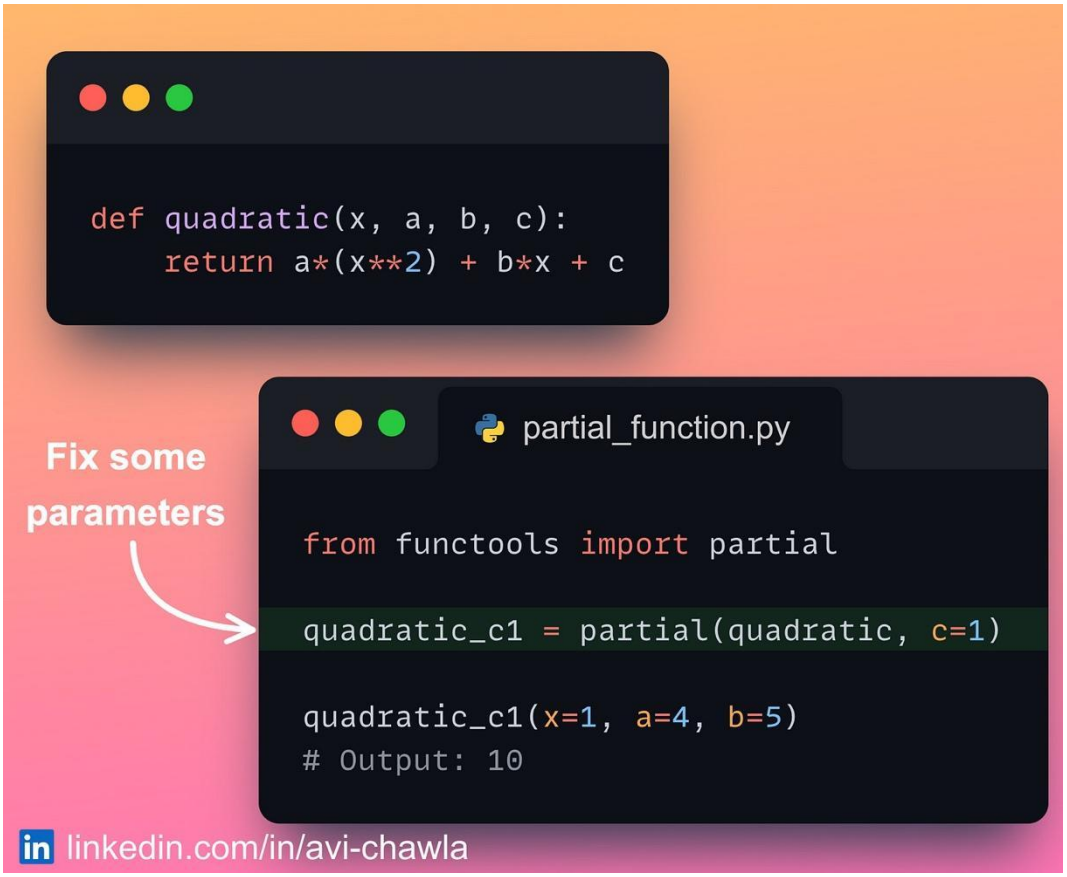


After initializing a class object, we often create derived attributes from existing variables.

To do this in dataclasses, you can use the **__post_init__** method. As the name suggests, this method is invoked right after the **__init__** method.

This is useful if you need to perform additional setups on your dataclass instance.

Simplify Your Functions With Partial Functions

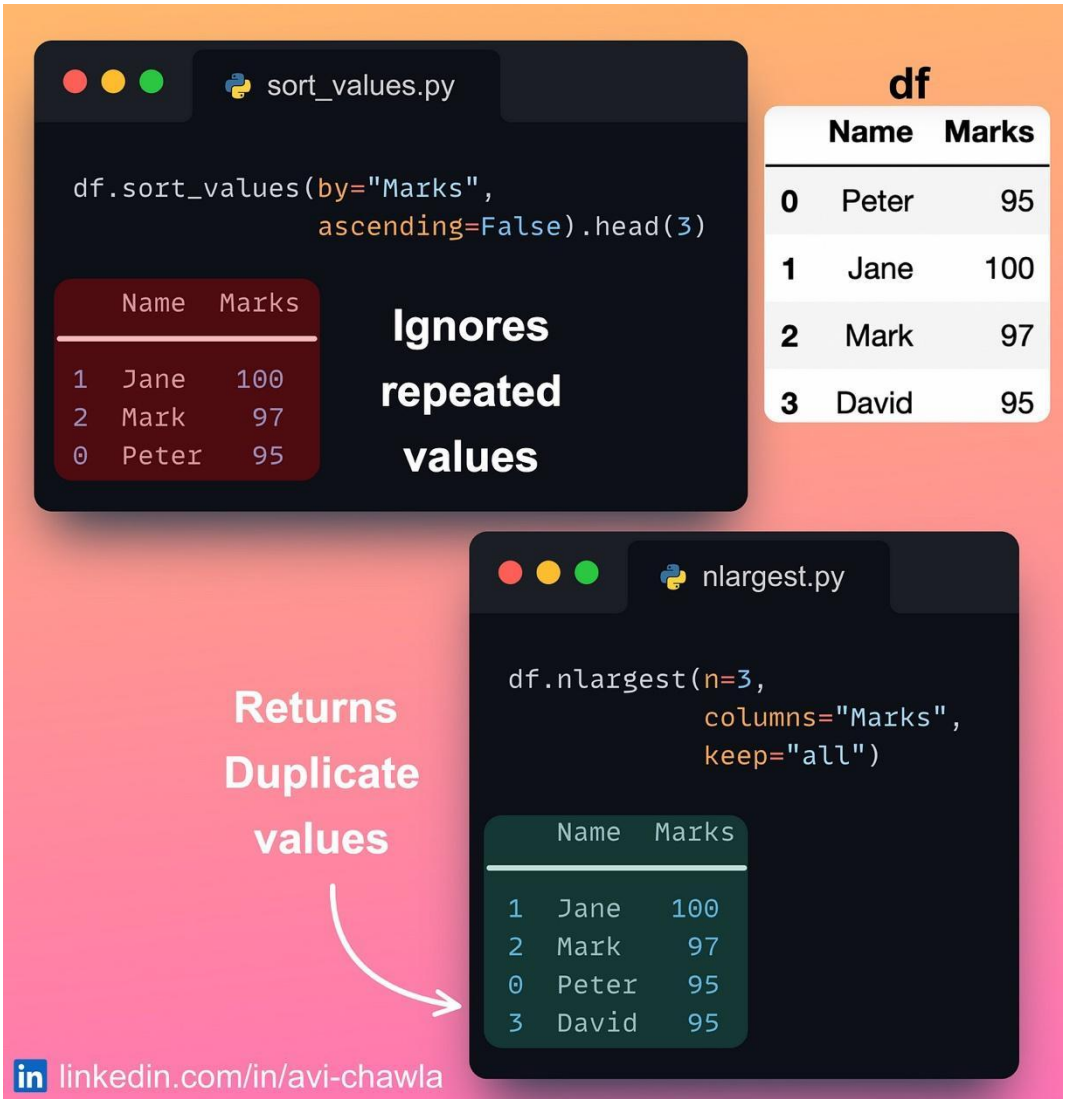


When your function takes many arguments, it can be a good idea to simplify it by using partial functions.

They let you create a new version of the function with some of the arguments fixed to specific values.

This can be useful for simplifying your code and making it more readable and concise. Moreover, it also helps you avoid repeating yourself while invoking functions.

When You Should Not Use the head() Method In Pandas

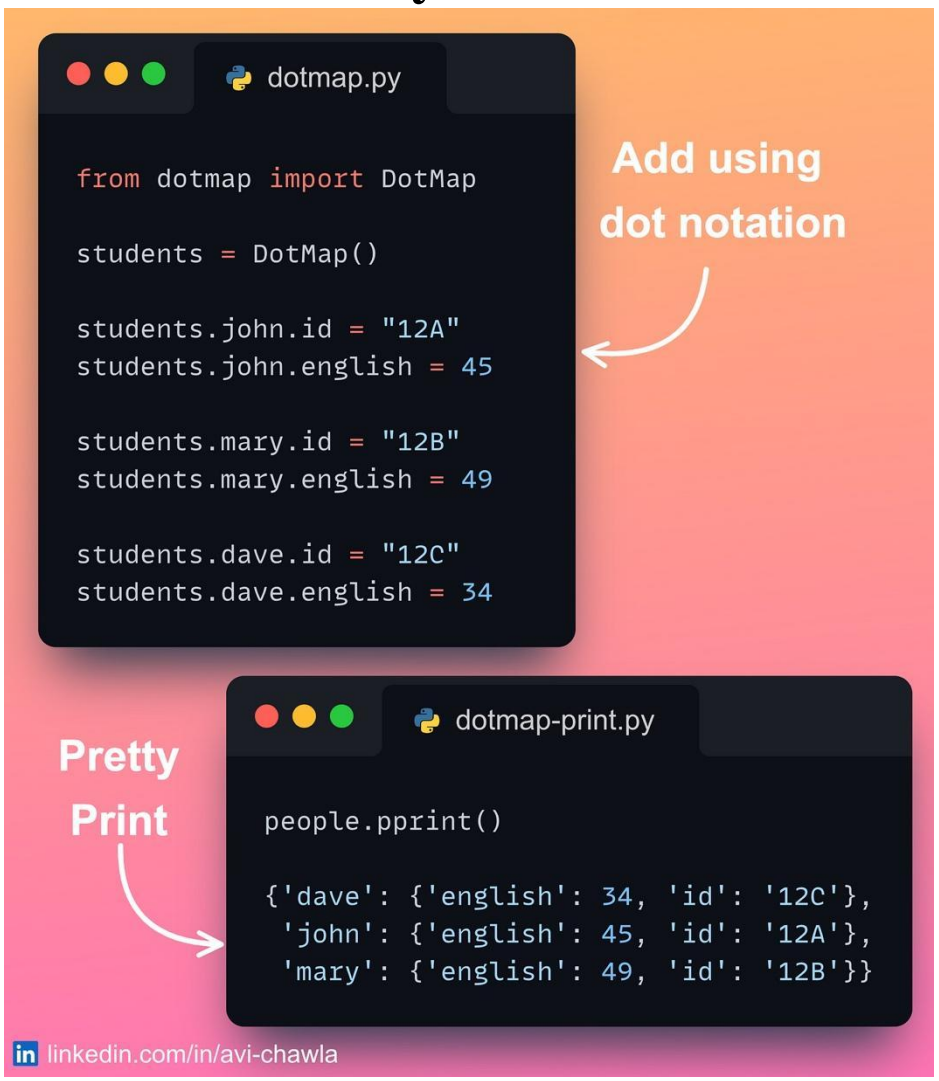


One often retrieves the top **k** rows of a sorted Pandas DataFrame by using **head**() method. However, there's a flaw in this approach.

If your data has repeated values, **head**() will not consider that and just return the first **k** rows.

If you want to consider repeated values, use **nlargest** (or **nsmallest**) instead. Here, you can specify the desired behavior for duplicate values using the **keep** parameter.

DotMap: A Better Alternative to Python Dictionary



Python dictionaries are great, but they have many limitations.

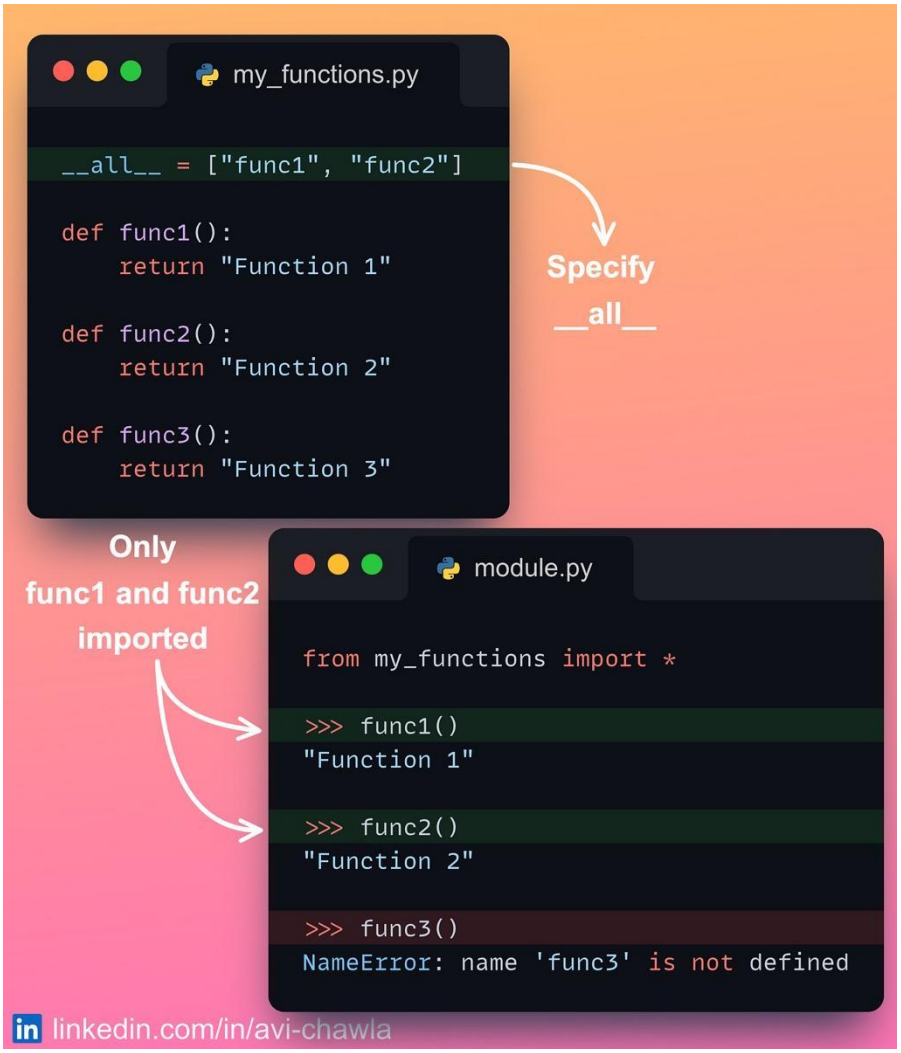
It is difficult to create dynamic hierarchical data. Also, they don't offer the widely adopted dot notation to access values.

Instead, use DotMap. It behaves like a Python dictionary but also addresses the above limitations.

What's more, it also has a built-in pretty print method to display it as a dict/JSON for debugging large objects.

Read more: GitHub.

Prevent Wild Imports With __all__ in Python

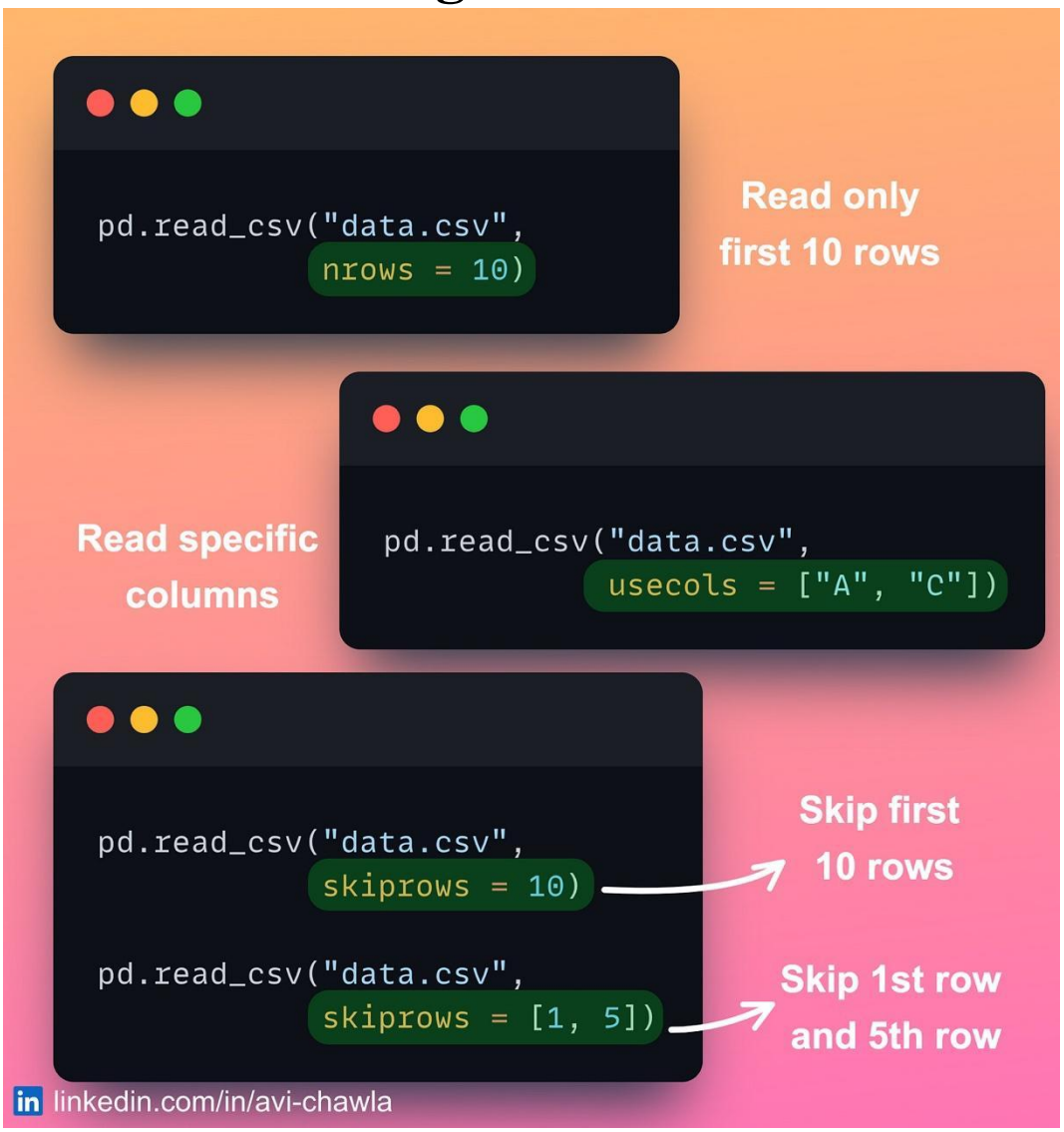


Wild imports (**from module import** *) are considered a bad programming practice. Yet, here's how you can prevent it if someone irresponsibly does that while using your code.

In your module, you can define the importable functions/classes/variables in **__all__**. As a result, whenever someone will do a wild import, Python will only import the symbols specified here.

This can be also useful to convey what symbols in your module are intended to be private.

Three Lesser-known Tips For Reading a CSV File Using Pandas



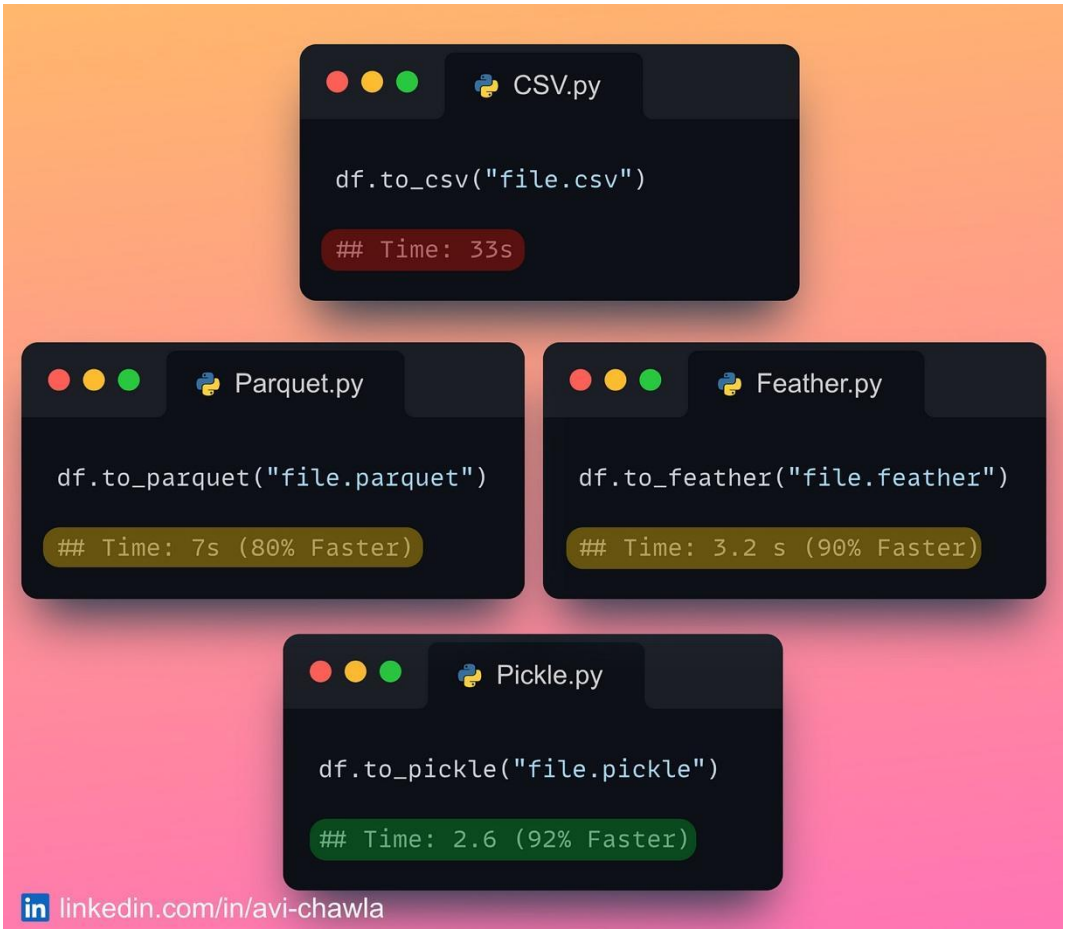
Here are three extremely useful yet lesser-known tips for reading a CSV file with Pandas:

1. If you want to read only the first few rows of the file, specify the **nrows** parameter.

2. To load a few specific columns, specify the **usecols** parameter.

3. If you want to skip some rows while reading, pass the **skiprows** parameter.

The Best File Format To Store A Pandas DataFrame



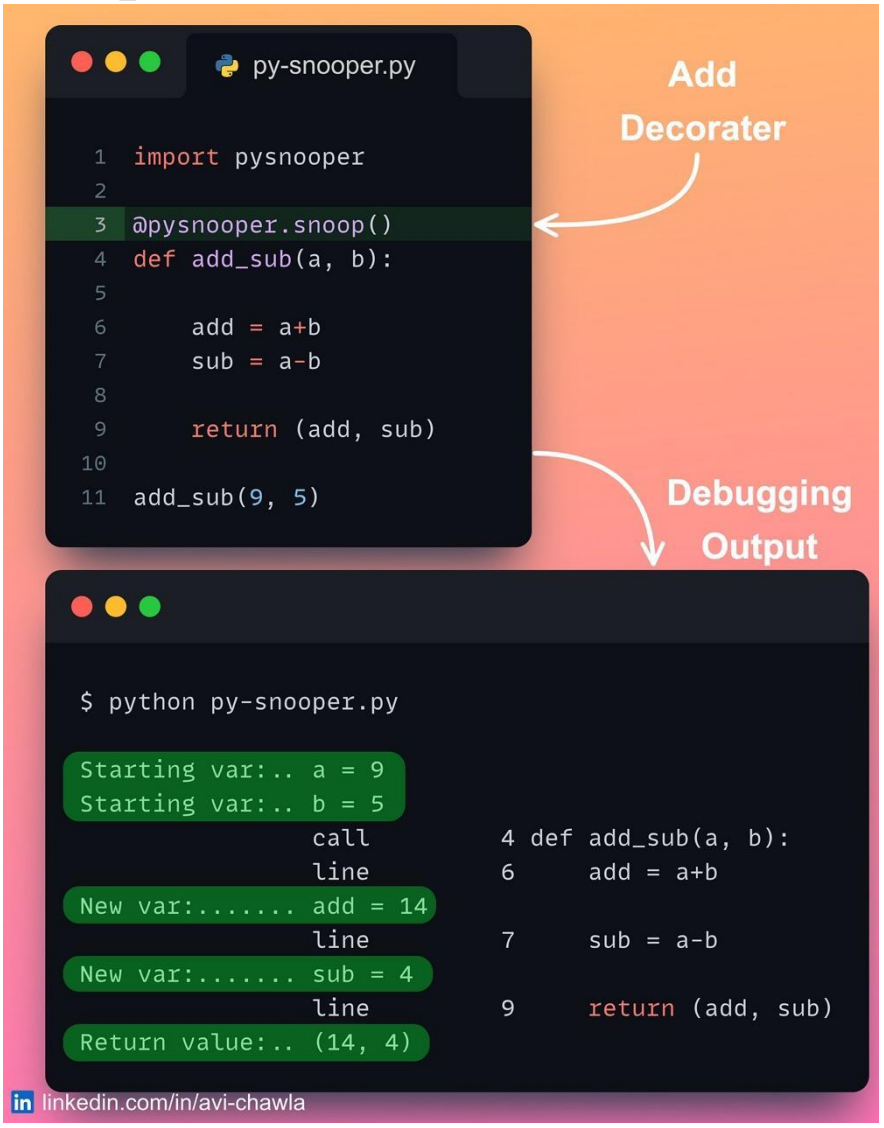
In the image above, you can find the run-time comparison of storing a Pandas DataFrame in various file formats.

Although CSVs are a widely adopted format, it is the slowest format in this list.

Thus, CSVs should be avoided unless you want to open the data outside Python (in Excel, for instance).

Read more in my blog: Medium.

Debugging Made Easy With PySnooper

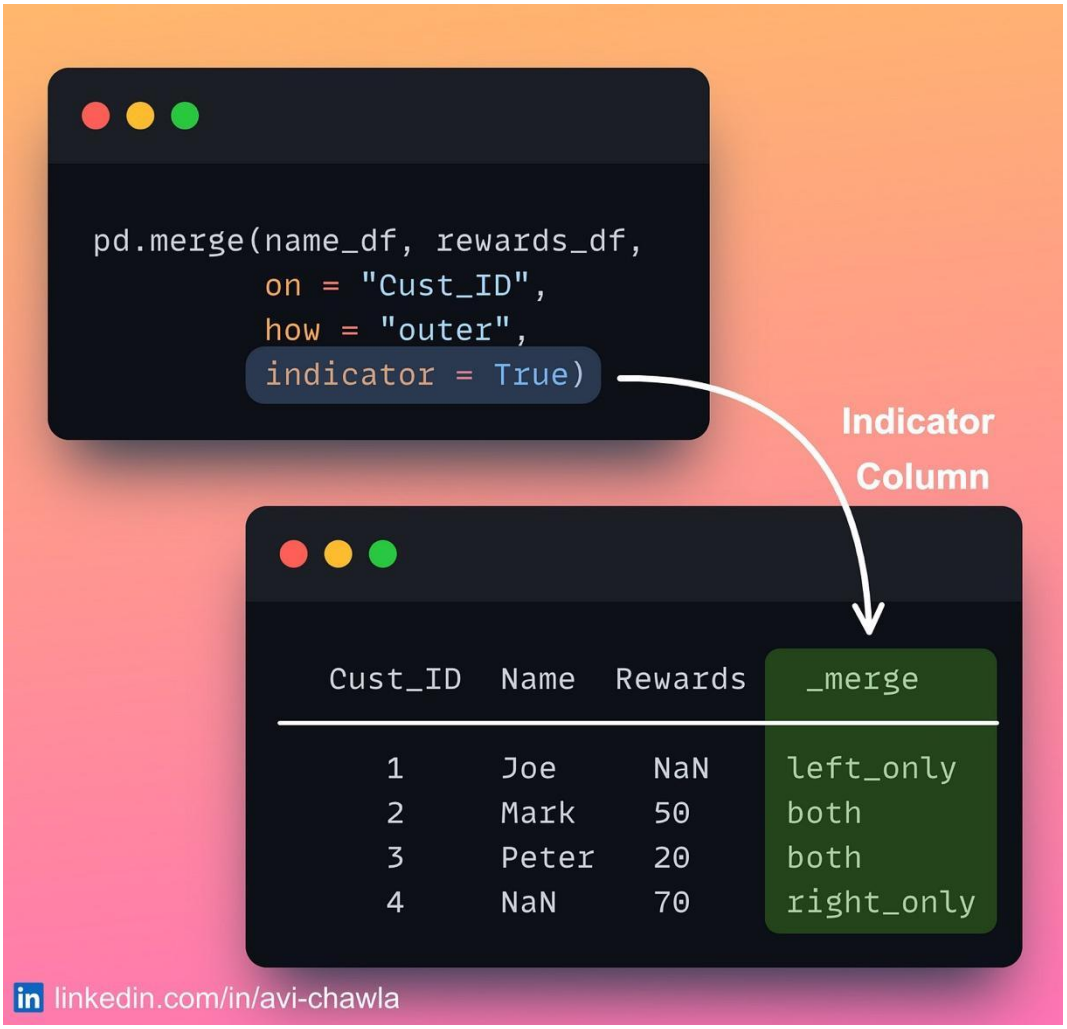


Rather than using many print statements to debug your python code, try PySnooper.

With just a single line of code, you can easily track the variables at each step of your code's execution.

Read more: Repository.

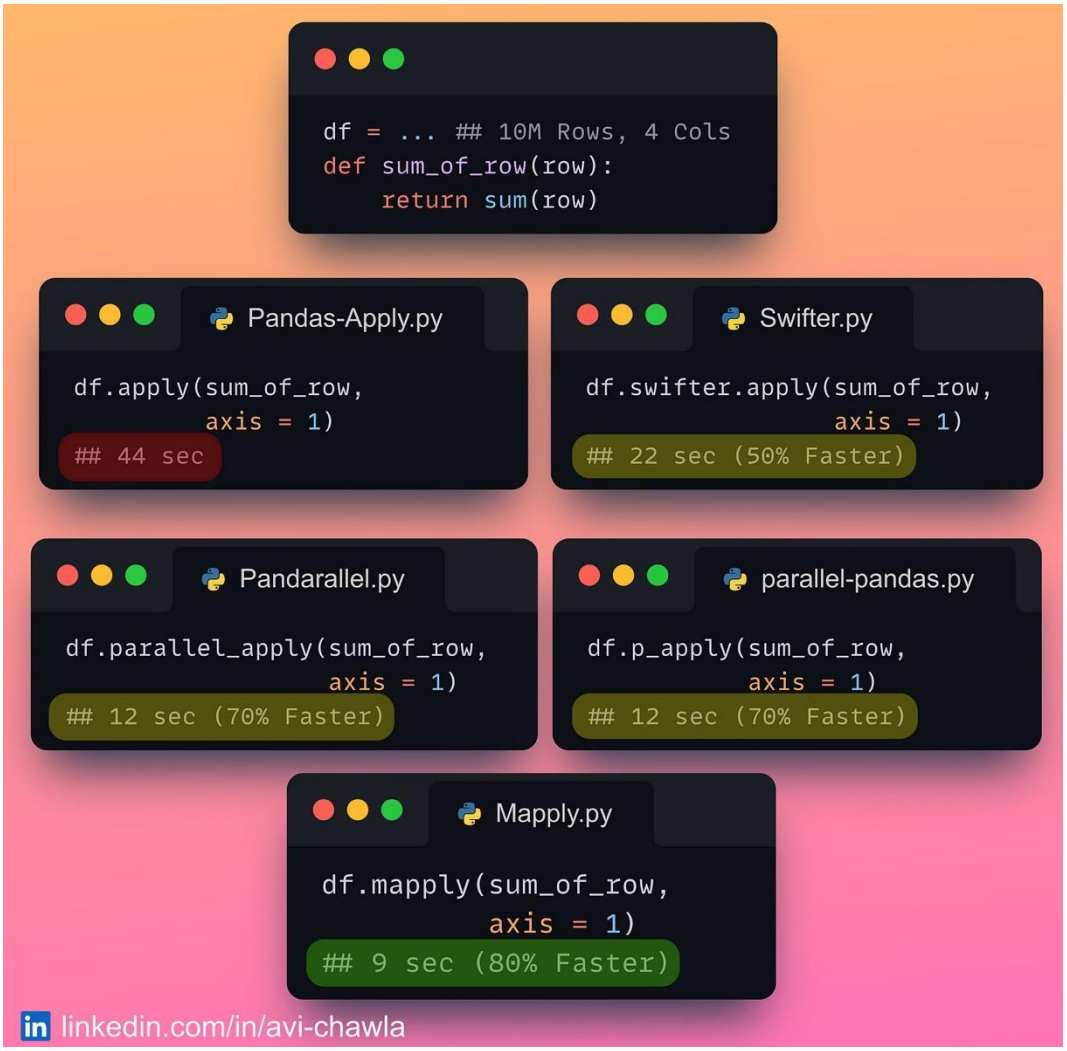
Lesser-Known Feature of the Merge Method in Pandas



While merging DataFrames in Pandas, keeping track of the source of each row in the output can be extremely useful.

You can do this using the **indicator** argument of the **merge**() method. As a result, it augments an additional column in the merged output, which tells the source of each row.

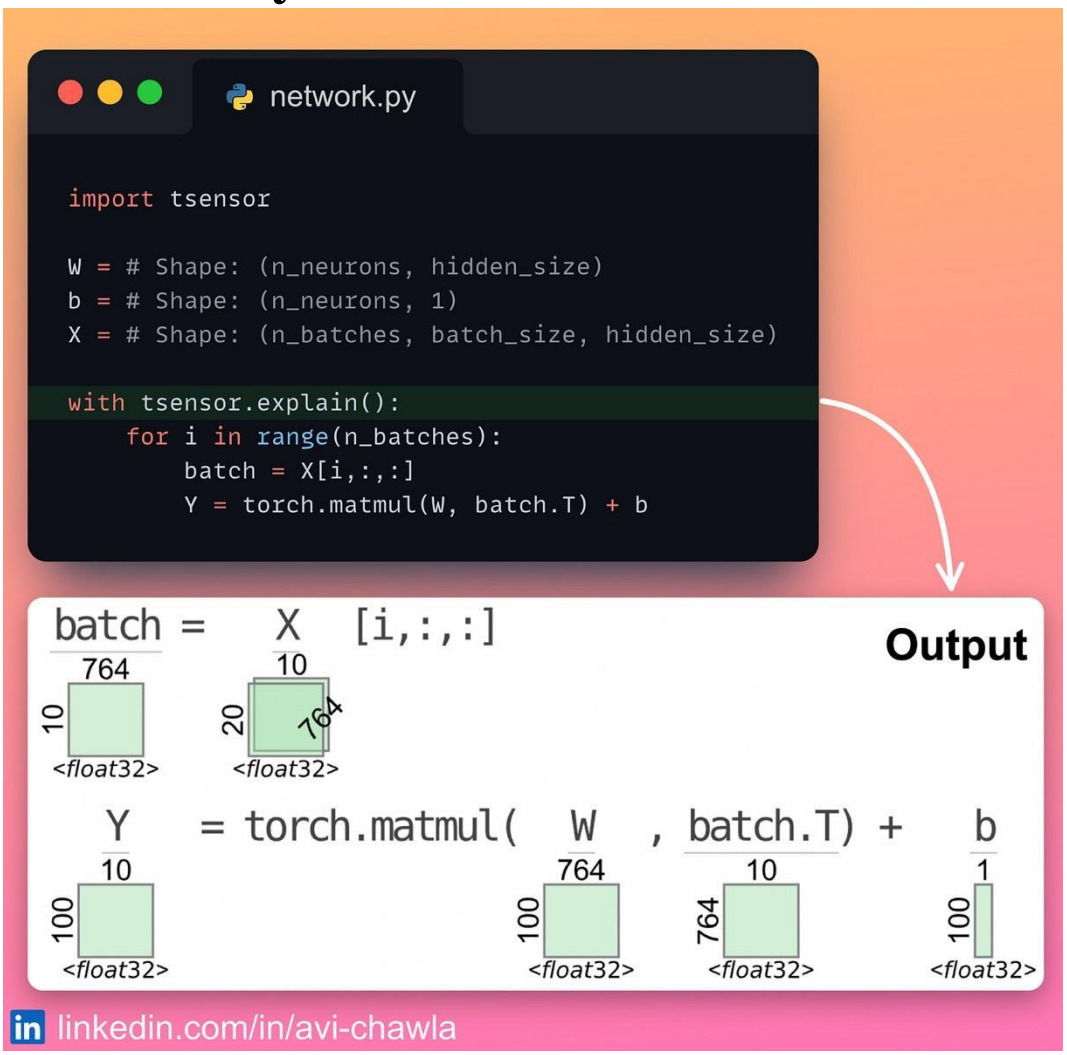
The Best Way to Use Apply() in Pandas



The image above shows a run-time comparison of popular open-source libraries that provide parallelization support for Pandas.

You can find the links to these libraries here. Also, if you know any other similar libraries built on top of Pandas, do post them in the comments or reply to this email.

Deep Learning Network Debugging Made Easy



Aligning the shape of tensors (or vectors/matrices) in a network can be challenging at times.

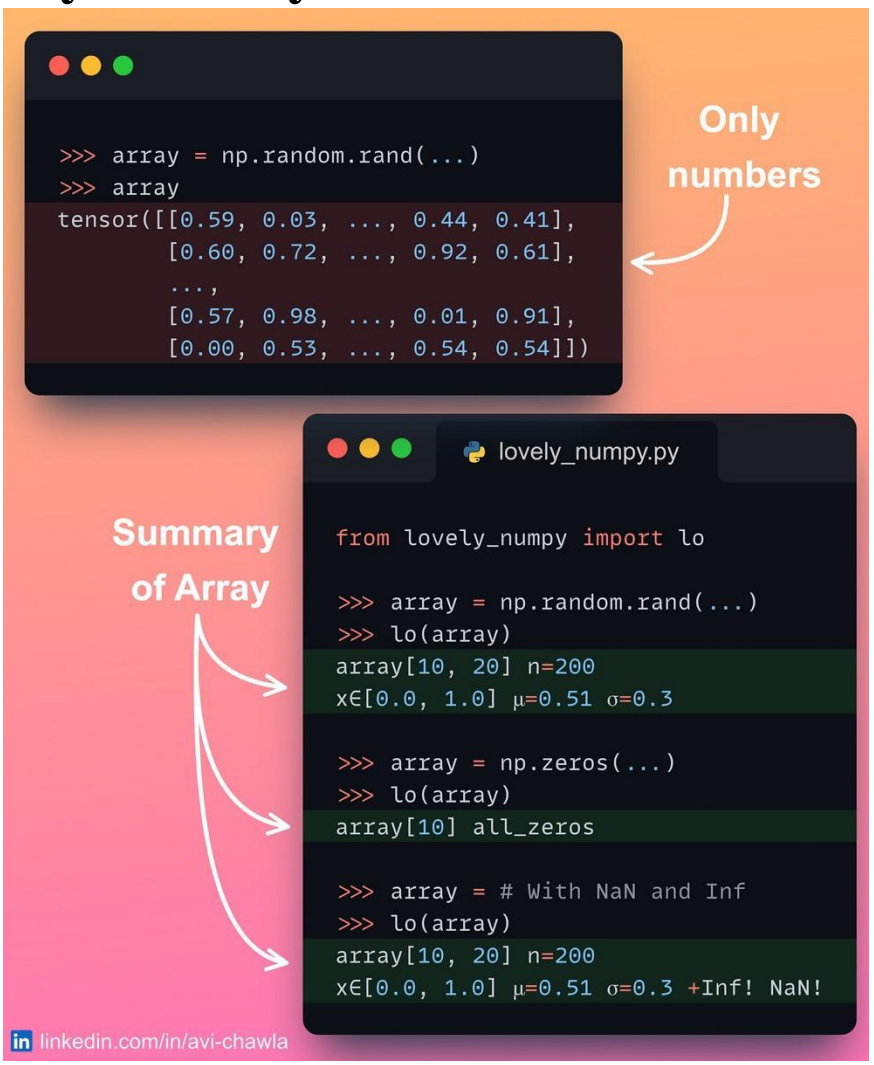
As the network grows, it is common to lose track of dimensionalities in a complex expression.

Instead of explicitly printing tensor shapes to debug, use **TensorSensor**. It generates an elegant visualization for each statement executed within its block. This makes dimensionality tracking effortless and quick.

In case of errors, it augments default error messages with more helpful details. This further speeds up the debugging process.

Read more: Documentation

Don't Print NumPy Arrays! Use Lovely-NumPy Instead.



We often print raw numpy arrays during debugging. But this approach is not very useful. This is because printing does not convey much information about the data it holds, especially when the array is large.

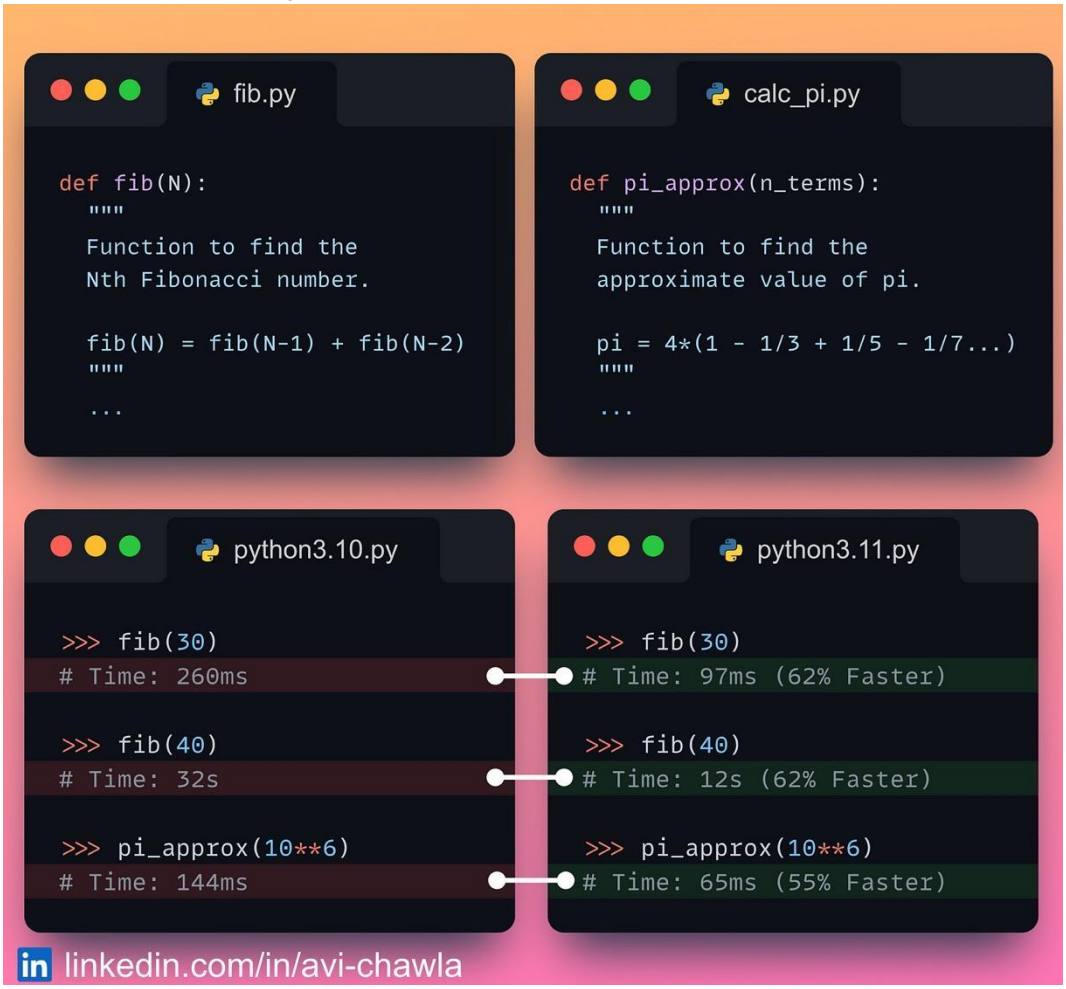
Instead, use **lovely-numpy**. Rather than viewing raw arrays, it prints a summary of the array. This includes its shape, distribution, mean, standard deviation, etc.

It also shows if the numpy array has NaNs and Inf values, whether it is filled with zeros, and many more.

P.S. If you work with tensors, then you can use **lovely-tensors**.

Read more: Documentation.

Performance Comparison of Python 3.11 and Python 3.10



Python 3.11 was released recently, and as per the official release, it is expected to be 10-60% faster than Python 3.10.

I ran a few basic benchmarking experiments to verify the performance boost. Indeed, Python 3.11 is much faster.

Although one might be tempted to upgrade asap, there are a few things you should know. Read more here.

View Documentation in Jupyter Notebook

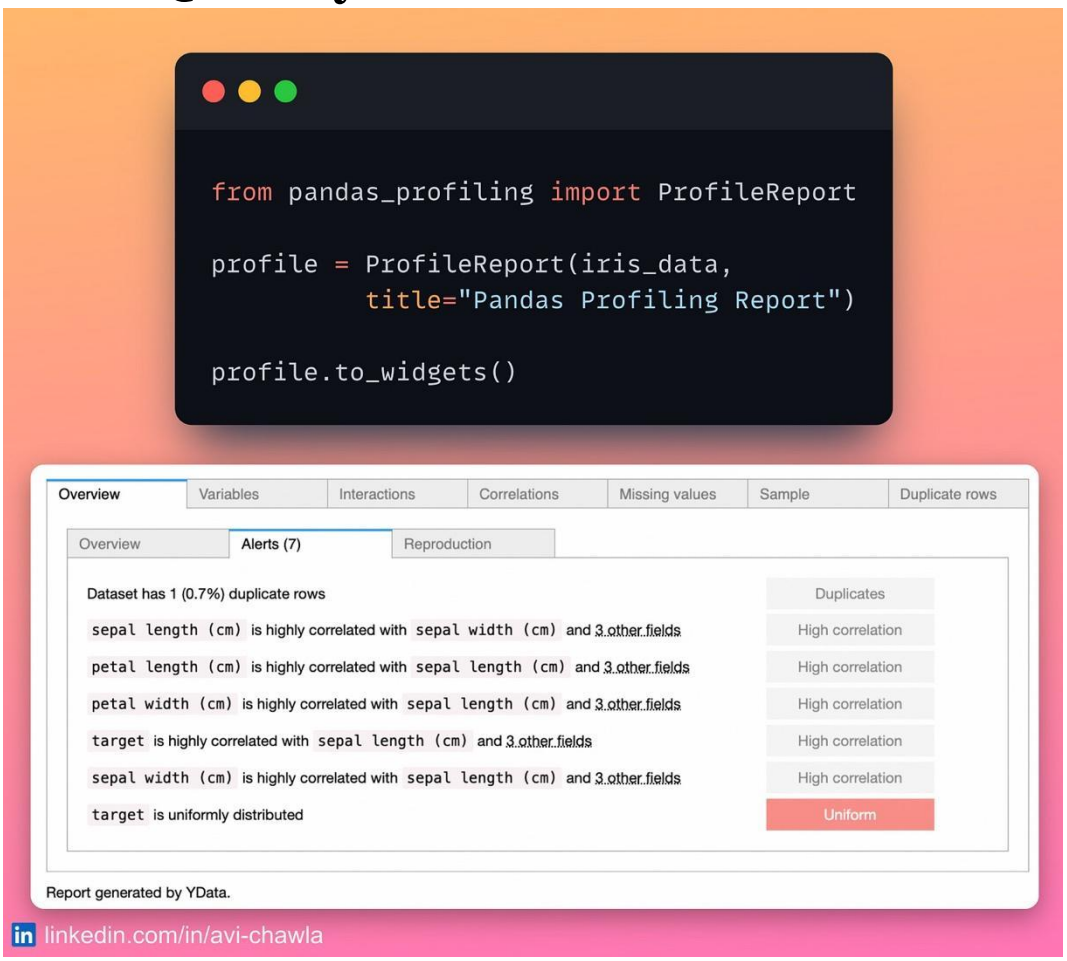
While working in Jupyter, it is common to forget the parameters of a function and visit the official docs (or Stackoverflow). However, you can view the documentation in the notebook itself.

Pressing **Shift-Tab** opens the documentation panel. This is extremely useful and saves time as one does not have to open the official docs every single time.

This feature also works for your custom functions.

View a video version of this post on LinkedIn: Post Link.

A No-code Tool To Understand Your Data Quickly



The preliminary steps of any typical EDA task are often the same. Yet, across projects, we tend to write the same code to carry out these tasks. This gets repetitive and time-consuming.

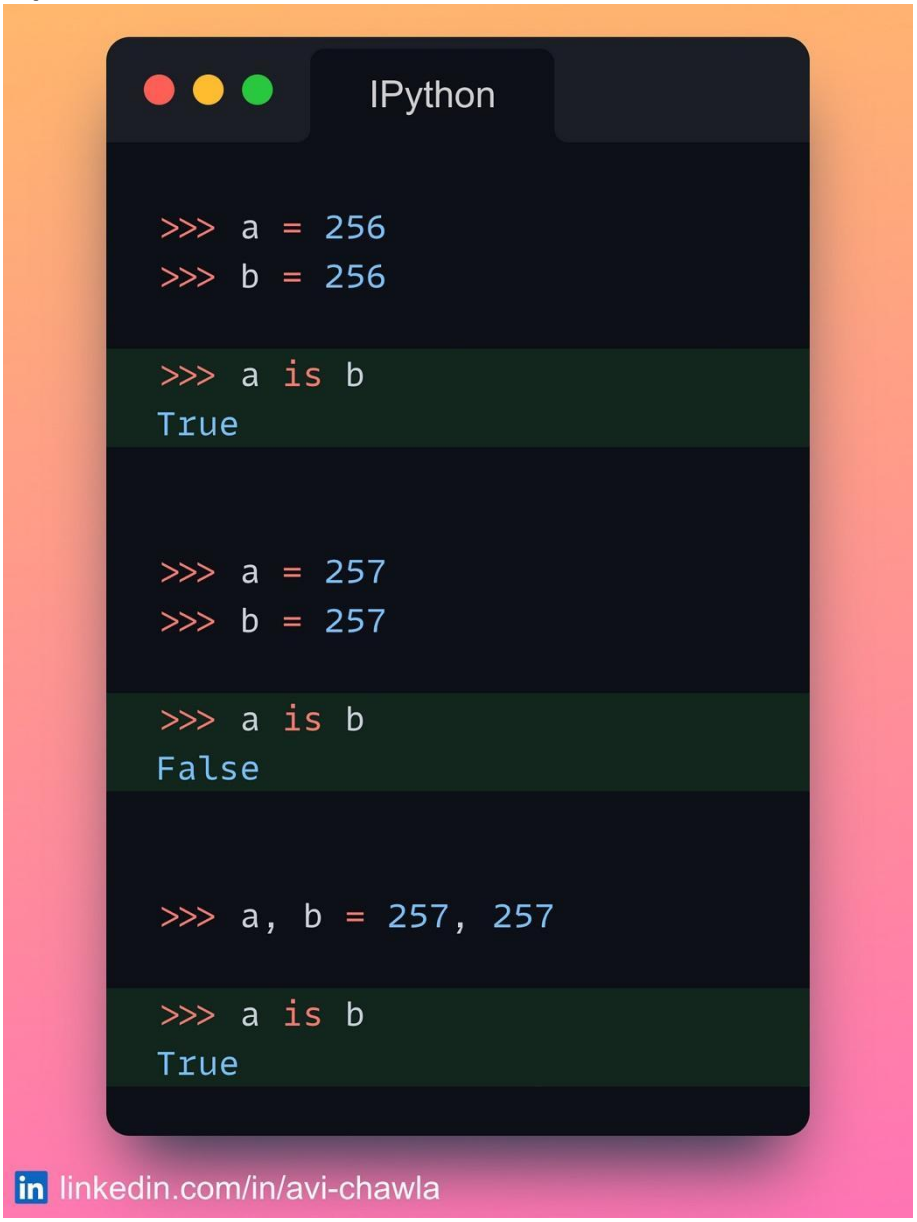
Instead, use **pandas-profiling**. It automatically generates a standardized report for data understanding in no time. Its intuitive UI makes this effortless and quick.

The report includes the dimension of the data, missing value stats, and column data types. What's more, it also shows the data distribution, the interaction and correlation between variables, etc.

Lastly, the report also includes alerts, which can be extremely useful during analysis/modeling.

Read more: Documentation.

Why 256 is 256 But 257 is not 257?



Comparing python objects can be tricky at times. Can you figure out what is going on in the above code example? Answer below:

When we run Python, it pre-loads a global list of integers in the range [-5, 256]. Every time an integer is referenced in this range, Python does not create a new object. Instead, it uses the cached version.

This is done for optimization purposes. It was considered that these numbers are used a lot by programmers. Therefore, it would make sense to have them ready at startup.

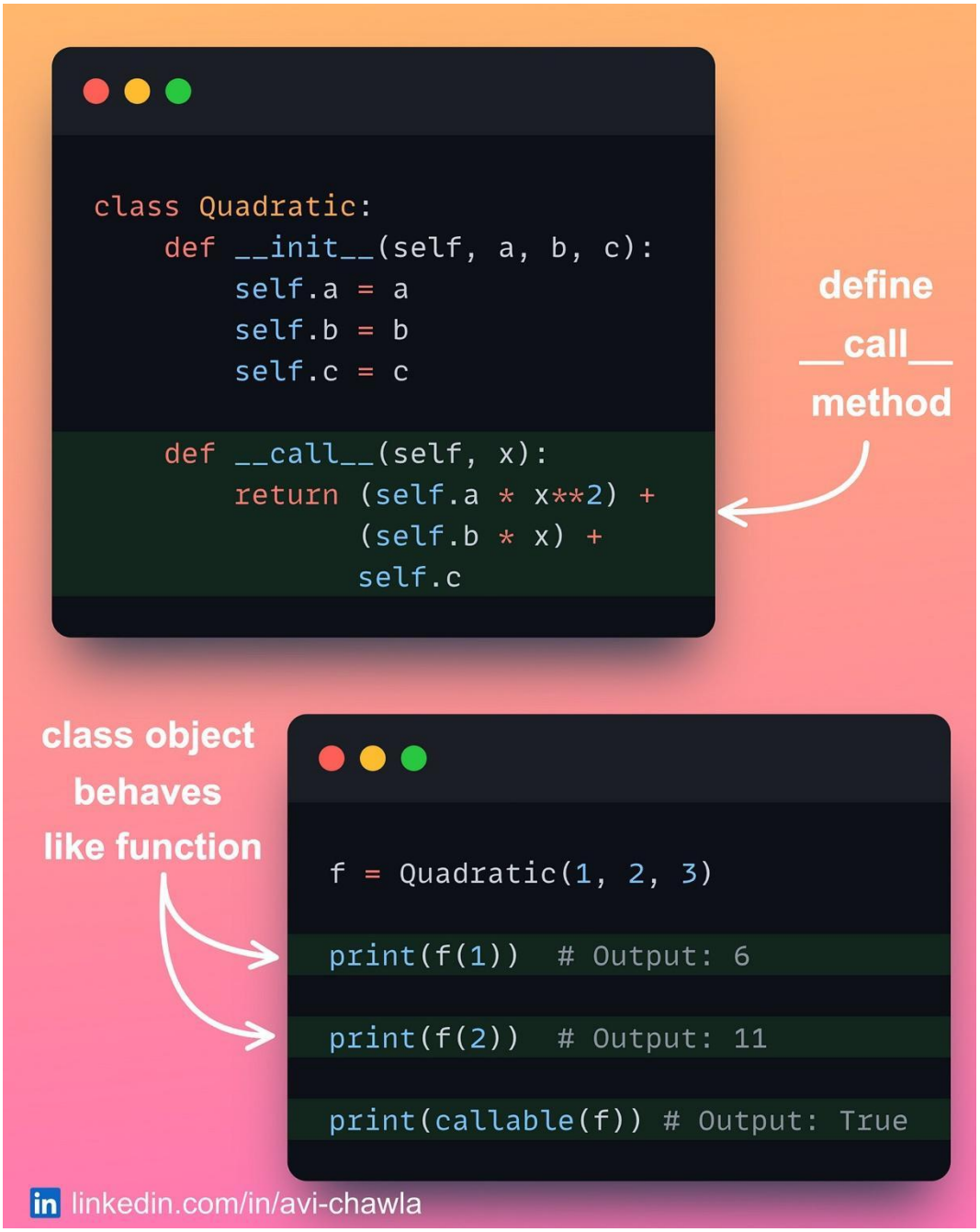
However, referencing any integer beyond 256 (or before -5) will create a new object every time.

In the last example, when a and b are set to 257 in the same line, the Python interpreter creates a new object. Then it references the second variable with the same object.

Share this post on LinkedIn: Post Link.

The below image should give you a better understanding:

Make a Class Object Behave Like a Function



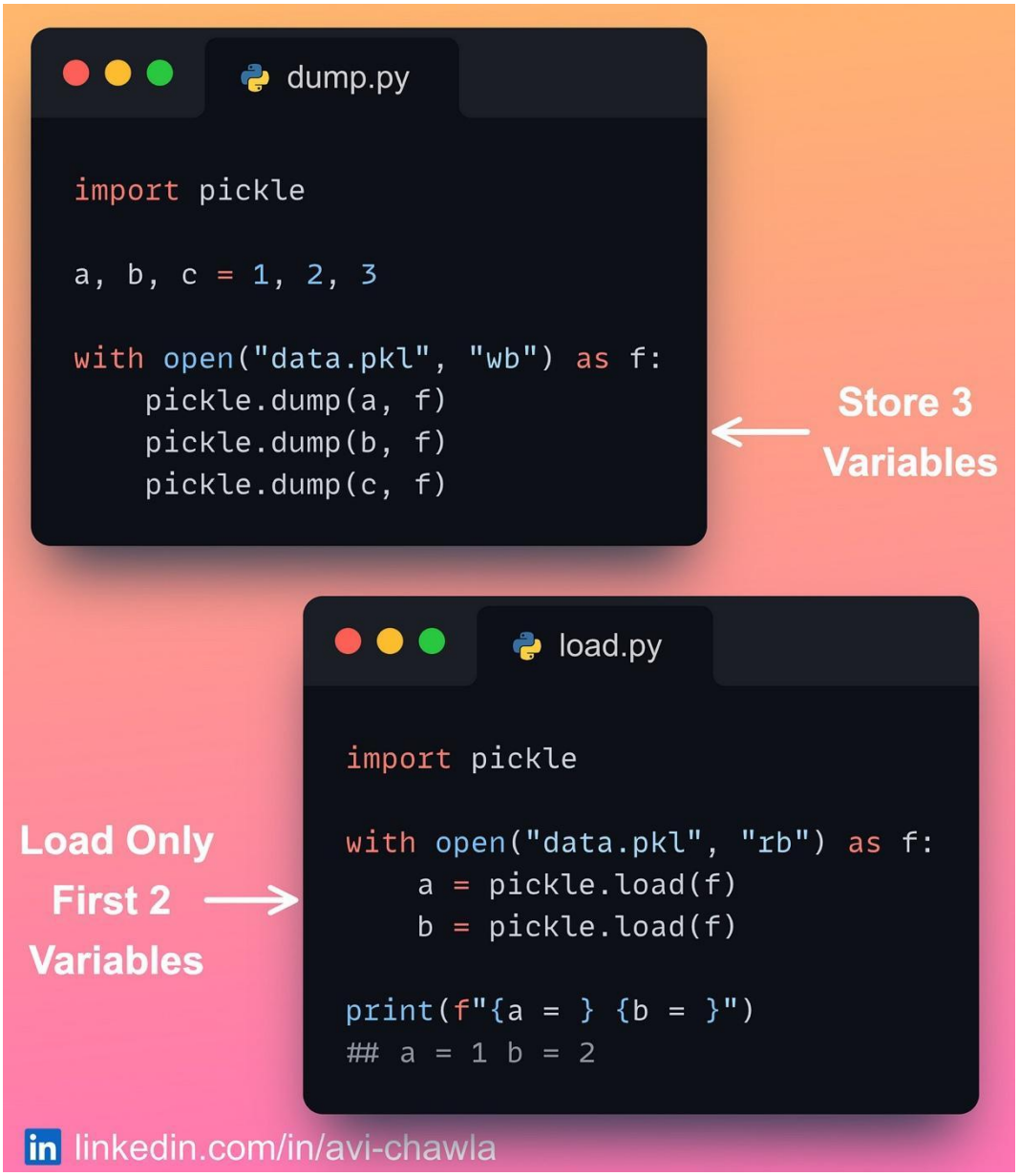
If you want to make a class object callable, i.e., behave like a function, you can do so by defining the **__call__** method.

This method allows you to define the behavior of the object when it is invoked like a function.

This can have many advantages. For instance, it allows us to implement objects that can be used in a flexible and intuitive way. What's more, the familiar function-call syntax, at times, can make your code more readable.

Lastly, it allows you to use a class object in contexts where a callable is expected. Using a class as a decorator, for instance.

Lesser-known feature of Pickle Files



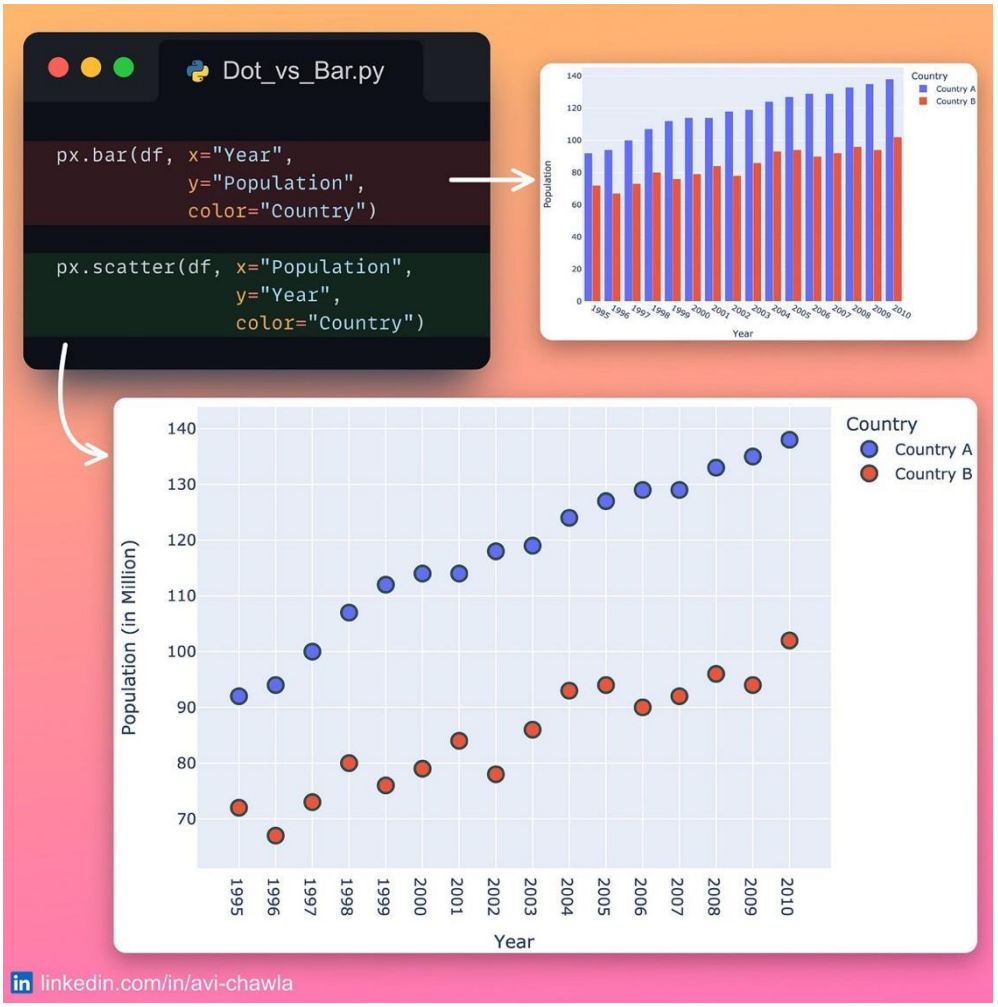
Pickles are widely used to dump data objects to disk. But folks often dump just a single object into a pickle file. Moreover, one creates multiple pickles to store multiple objects.

However, did you know that you can store as many objects as you want within a single pickle file? What's more, when reloading, it is not necessary to load all the objects.

Just make sure to dump the objects within the same context manager (using **with**).

Of course, one solution is to store the objects together as a tuple. But while reloading, the entire tuple will be loaded. This may not be desired in some cases.

Dot Plot: A Potential Alternative to Bar Plot



Bar plots are extremely useful for visualizing categorical variables against a continuous value. But when you have many categories to depict, they can get too dense to interpret.

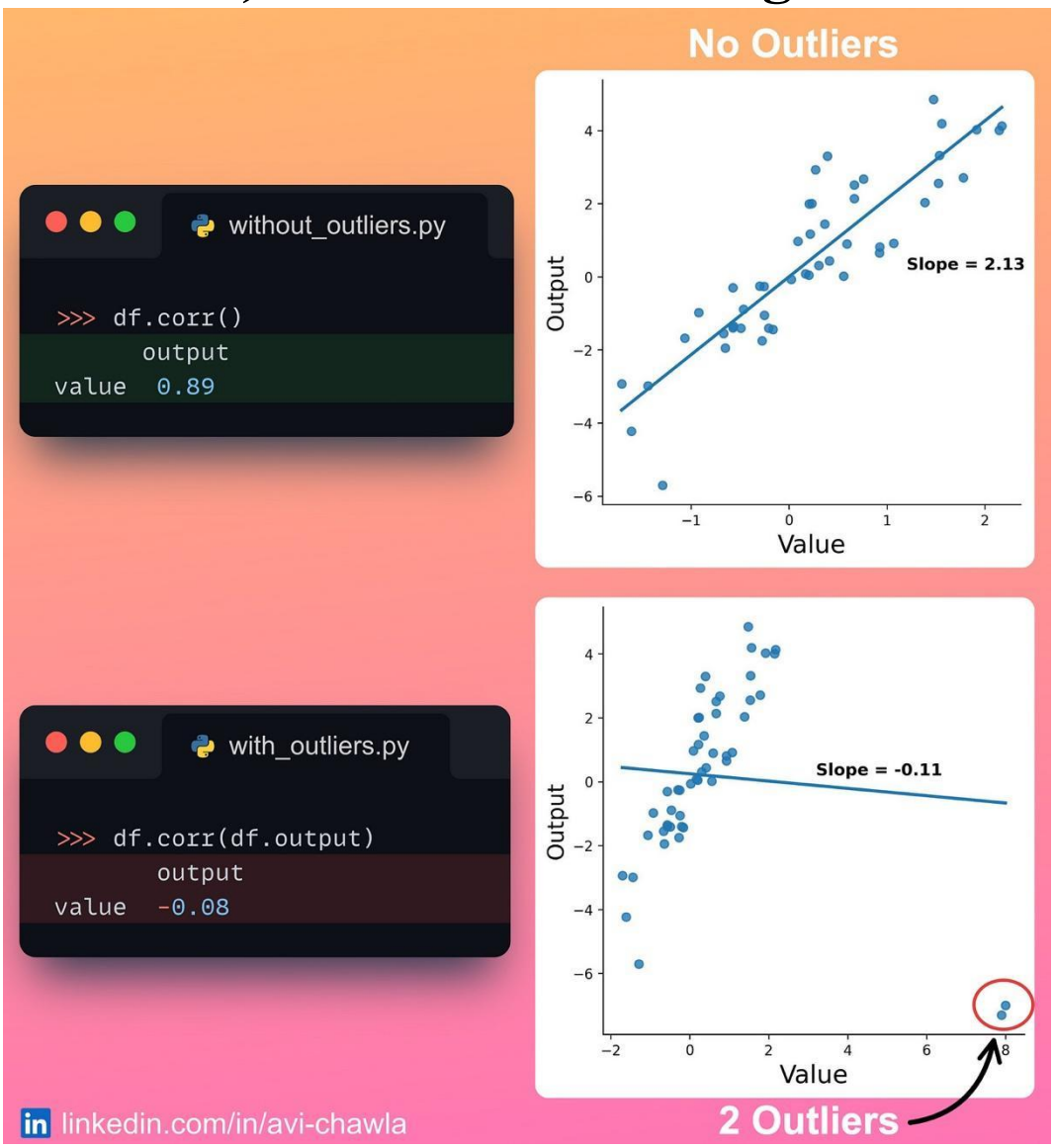
In a bar plot with many bars, we're often not paying attention to the individual bar lengths. Instead, we mostly consider the individual endpoints that denote the total value.

A Dot plot can be a better choice in such cases. They are like scatter plots but with one categorical and one continuous axis.

Compared to a bar plot, they are less cluttered and offer better comprehension. This is especially true in cases where we have many categories and/or multiple categorical columns to depict in a plot.

Read more: Documentation.

Why Correlation (and Other Statistics) Can Be Misleading.



Correlation is often used to determine the association between two continuous variables. But it has a major flaw that often gets unnoticed.

Folks often draw conclusions using a correlation matrix without even looking at the data. However, the obtained statistics could be heavily driven by outliers or other artifacts.

This is demonstrated in the plots above. The addition of just two outliers changed the correlation and the regression line drastically.

Thus, looking at the data and understanding its underlying characteristics can save from drawing wrong conclusions. Statistics are important, but they can be highly misleading at times.

Supercharge value_counts() Method in Pandas With Sidetable

The **value_counts**() method is commonly used to analyze categorical columns, but it has many limitations.

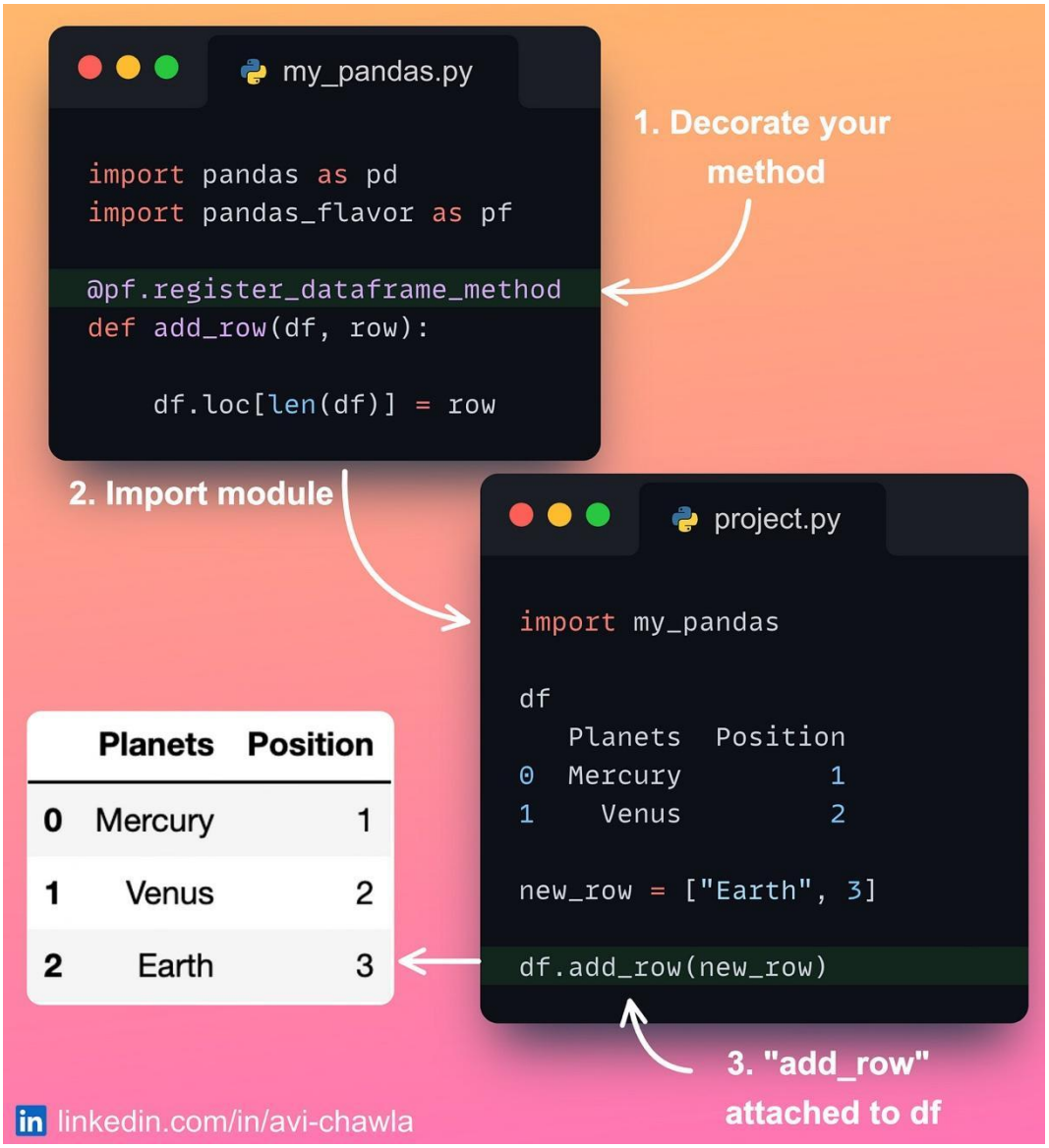
For instance, if one wants to view the percentage, cumulative count, etc., in one place, things do get a bit tedious. This requires more code and is time-consuming.

Instead, use **sidetable**. Consider it as a supercharged version of **value_counts**(). As shown below, the **freq**() method from sidetable provides a more useful summary than **value_counts**().

Additionally, sidetable can aggregate multiple columns too. You can also provide threshold points to merge data into a single bucket. What's more, it can print missing data stats, pretty print values, etc.

Read more: GitHub.

Write Your Own Flavor Of Pandas



If you want to attach a custom functionality to a Pandas DataFrame (or series) object, use "pandas-flavor".

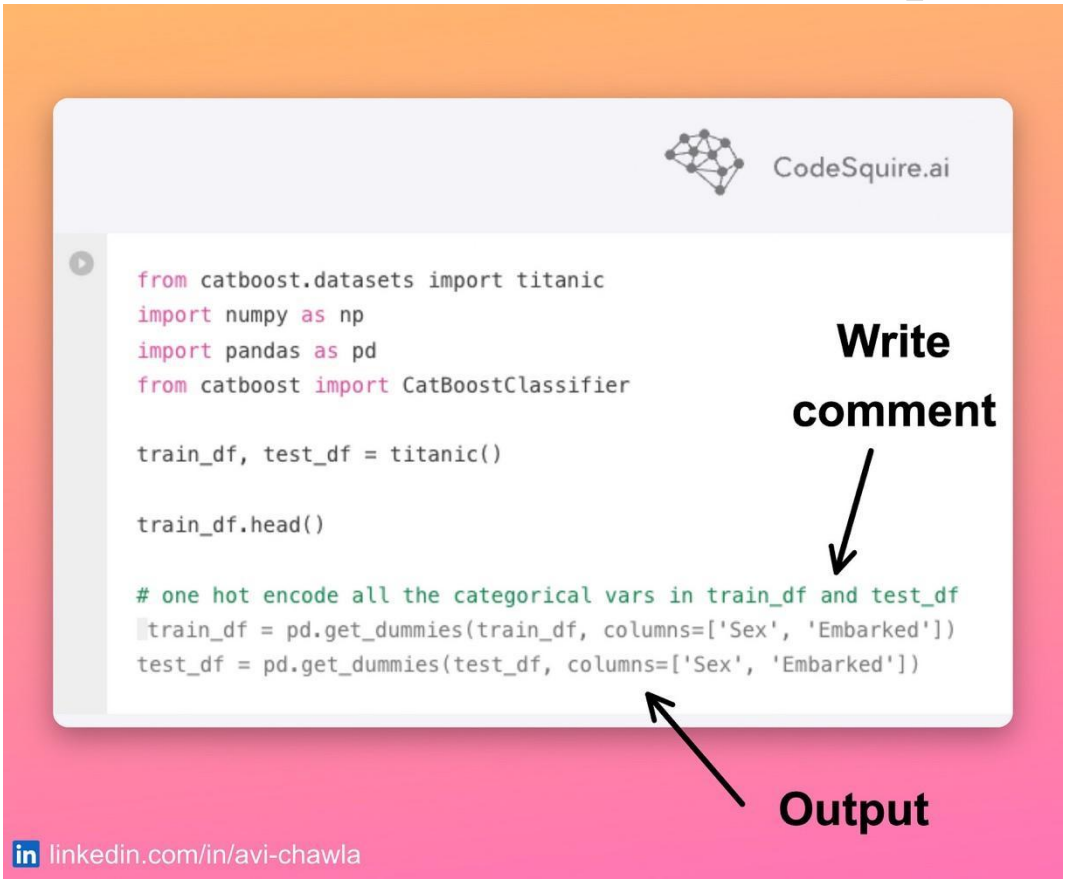
Its decorators allow you to add methods directly to the Pandas' object.

This is especially useful if you are building an open-source project involving Pandas. After installing your library, others can access your library's methods using the dataframe object.

P.S. This is how we see **df.progress_apply**() from **tqdm**, **df.parallel_apply**() from **Pandarallel**, and many more.

Read more: Documentation.

CodeSquire: The AI Coding Assistant You Should Use Over GitHub Copilot



Coding Assistants like GitHub Copilot are revolutionary as they offer many advantages. Yet, Copilot has limited utility for data professionals. This is because it's incompatible with web-based IDEs (Jupyter/Colab).

Moreover, in data science, the subsequent exploratory steps are determined by previous outputs. But Copilot does not consider that (and even markdown cells) to drive its code suggestions.

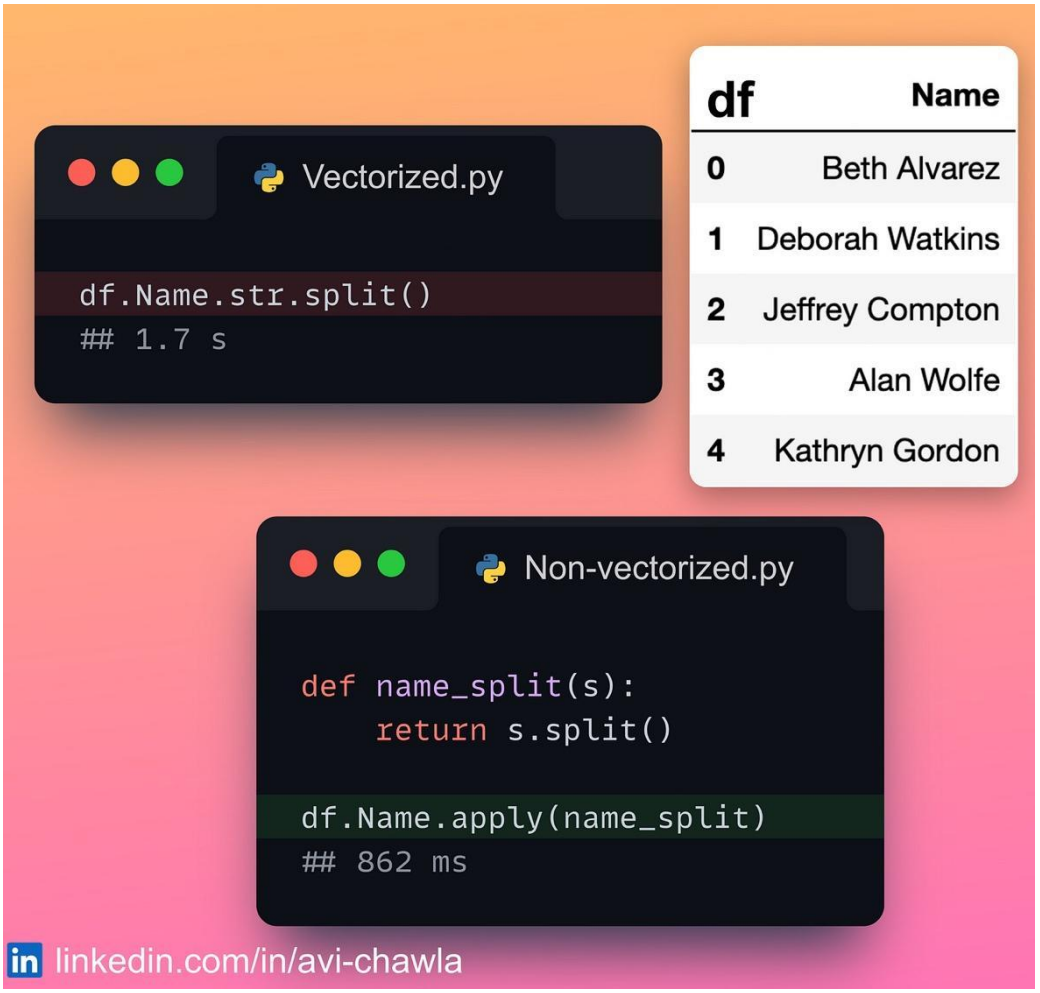
CodeSquire is an incredible AI coding assistant that addresses the limitations of Copilot. The good thing is that it has been designed specifically for data scientists, engineers, and analysts.

Besides seamless code generation, it can generate SQL queries from text and explain code. You can leverage AI-driven code generation by simply installing a browser extension.

Read more: CodeSquire.

Watch a video version of this post on LinkedIn: Post Link.

Vectorization Does Not Always Guarantee Better Performance



Vectorization is well-adopted for improving run-time performance. In a nutshell, it lets you operate data in batches instead of processing a single value at a time.

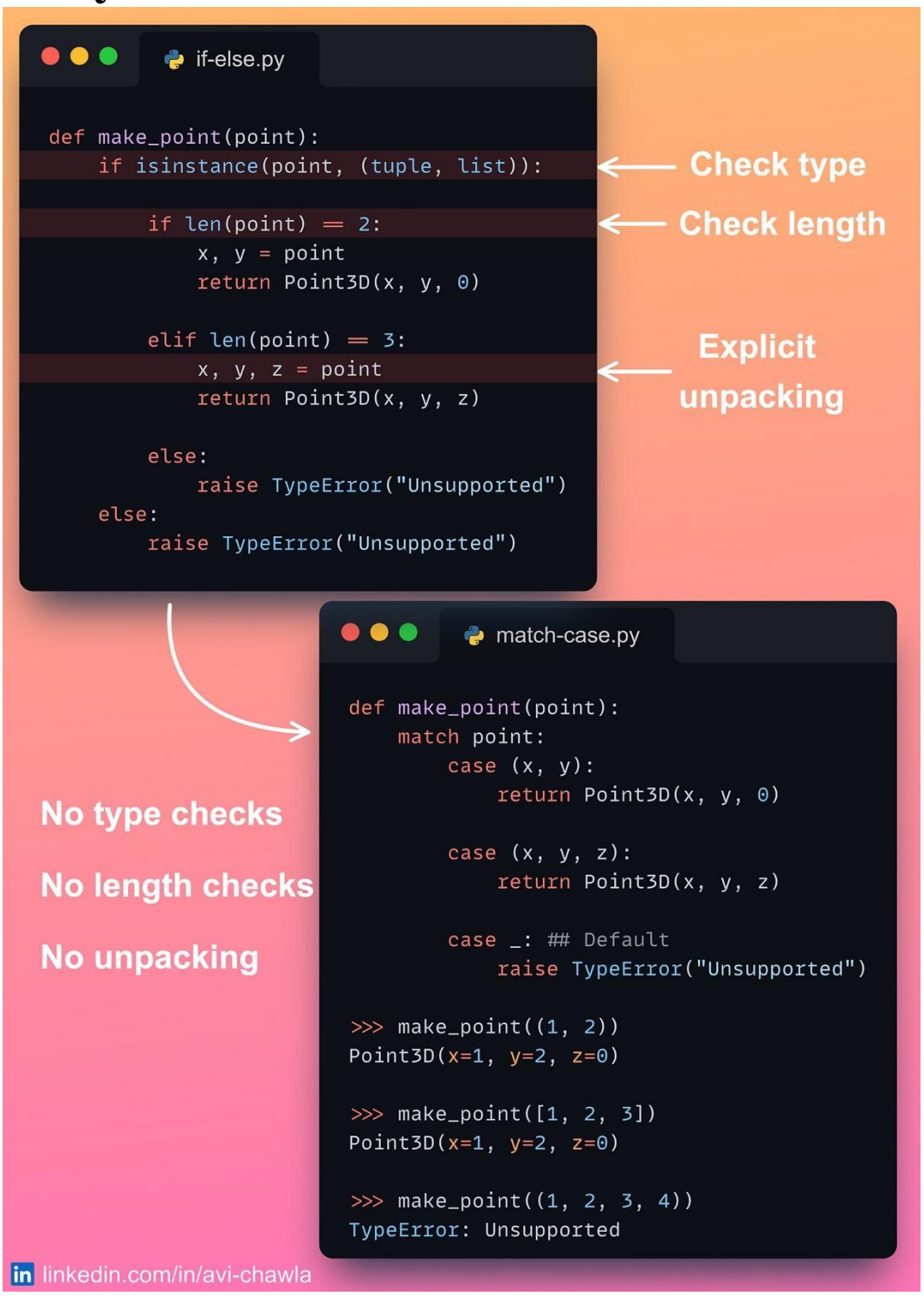
Although vectorization is extremely effective, you should know that it does not always guarantee performance gains. Moreover, vectorization is also associated with memory overheads.

As demonstrated above, the non-vectorized code provides better performance than the vectorized version.

P.S. **apply**() is also a for-loop.

Further reading: Here.

In Defense of Match-case Statements in Python



I recently came across a post on **match-case** in Python. In a gist, starting Python 3.10, you can use **match-case** statements to mimic the behavior of **if-else**.

Many responses on that post suggested that **if-else** offers higher elegance and readability. Here's an example in defense of **match-case**.

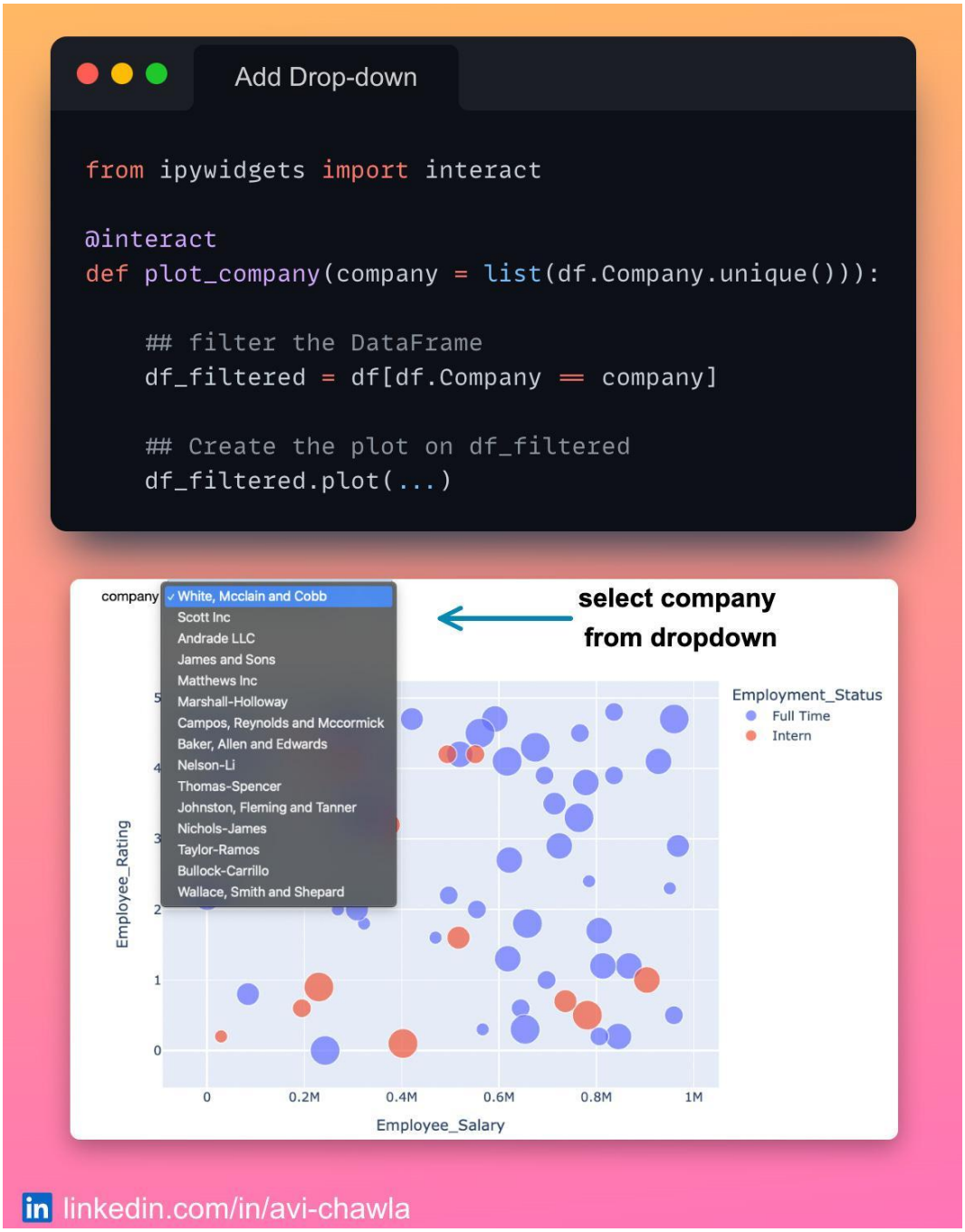
While if-else is traditionally accepted, it also comes with many downsides. For instance, many-a-times, one has to write complex chains of nested if-else statements. This includes multiple calls to **len**(), **isinstance**() methods, etc.

Furthermore, with **if-else**, one has to explicitly destructure the data to extract values. This makes your code inelegant and messy.

Match-case, on the other hand, offers Structural Pattern Matching which makes this simple and concise. In the example above, match-case automatically handles type-matching, length check, and variable unpacking.

Read more here: Python Docs.

Enrich Your Notebook With Interactive Controls



While using Jupyter, we often re-run the same cell repeatedly after changing the input slightly. This is time-consuming and also makes your data exploration tasks tedious and unorganized.

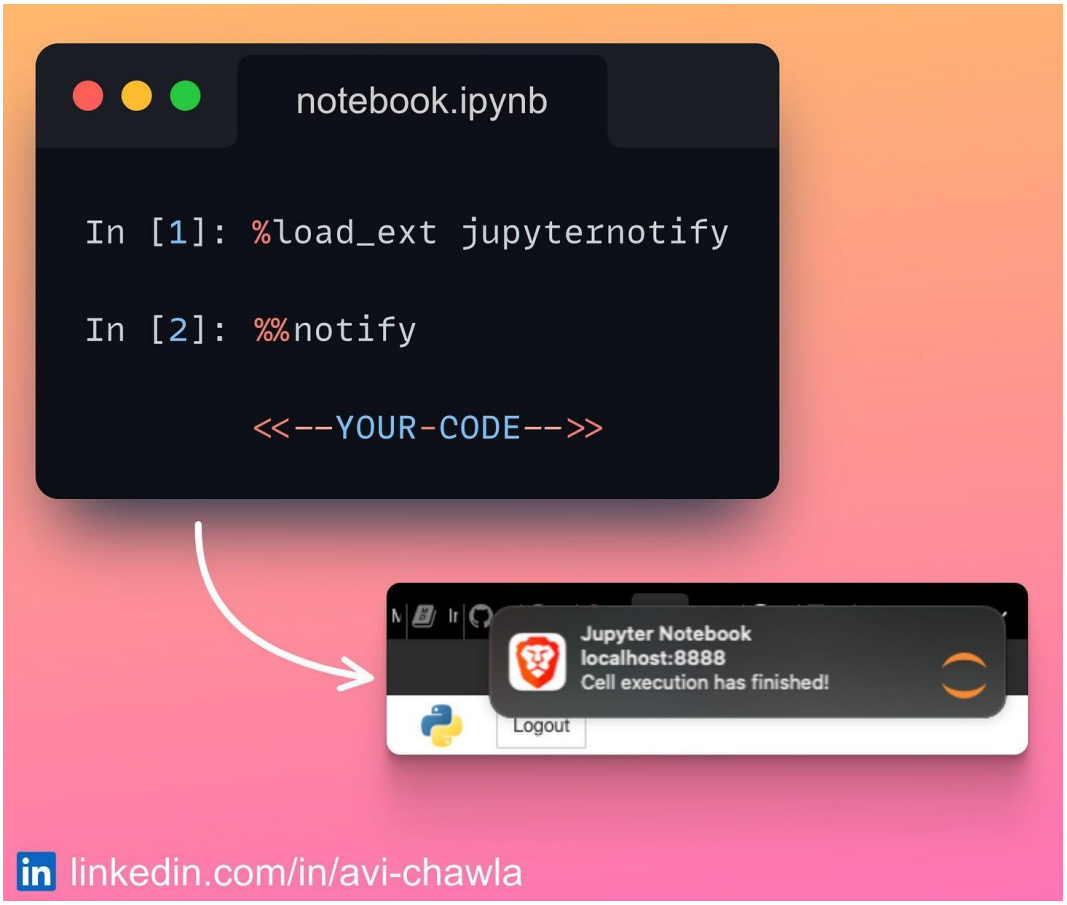
Instead, pivot towards building interactive controls in your notebook. This allows you to alter the inputs without needing to rewrite and re-run your code.

In Jupyter, you can do this using the **IPywidgets** module. Embedding interactive controls is as simple as using a decorator.

As a result, it provides you with interactive controls such as dropdowns and sliders. This saves you from tons of repetitive coding and makes your notebook organized.

Watch a video version of this post on LinkedIn: Post Link.

Get Notified When Jupyter Cell Has Executed



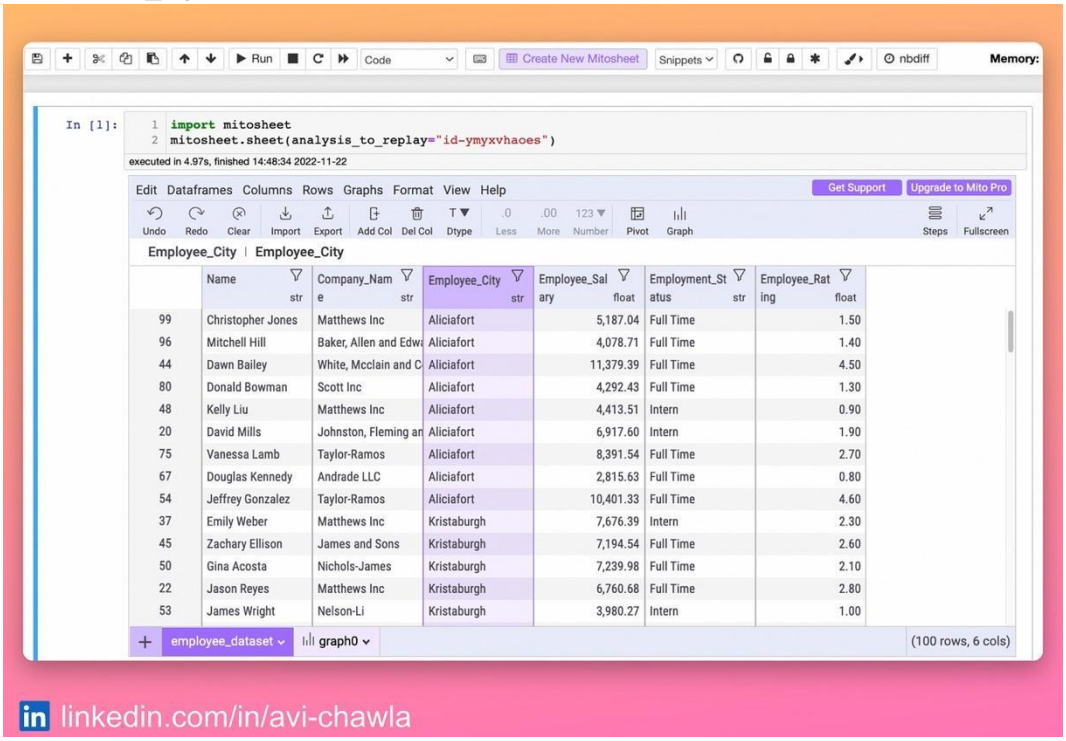
After running some code in a Jupyter cell, we often navigate away to do some other work in the meantime.

Here, one has to repeatedly get back to the Jupyter tab to check whether the cell has been executed or not.

To avoid this, you can use the %%**notify** magic command from the **jupyternotify** extension. As the name suggests, it notifies the user upon completion (both successful and unsuccessful) of a jupyter cell via a browser notification. Clicking on the notification takes you back to the jupyter tab.

Read more: GitHub.

Data Analysis Using No-Code Pandas In Jupyter



The Pandas API provides a wide range of functionalities to analyze tabular datasets.

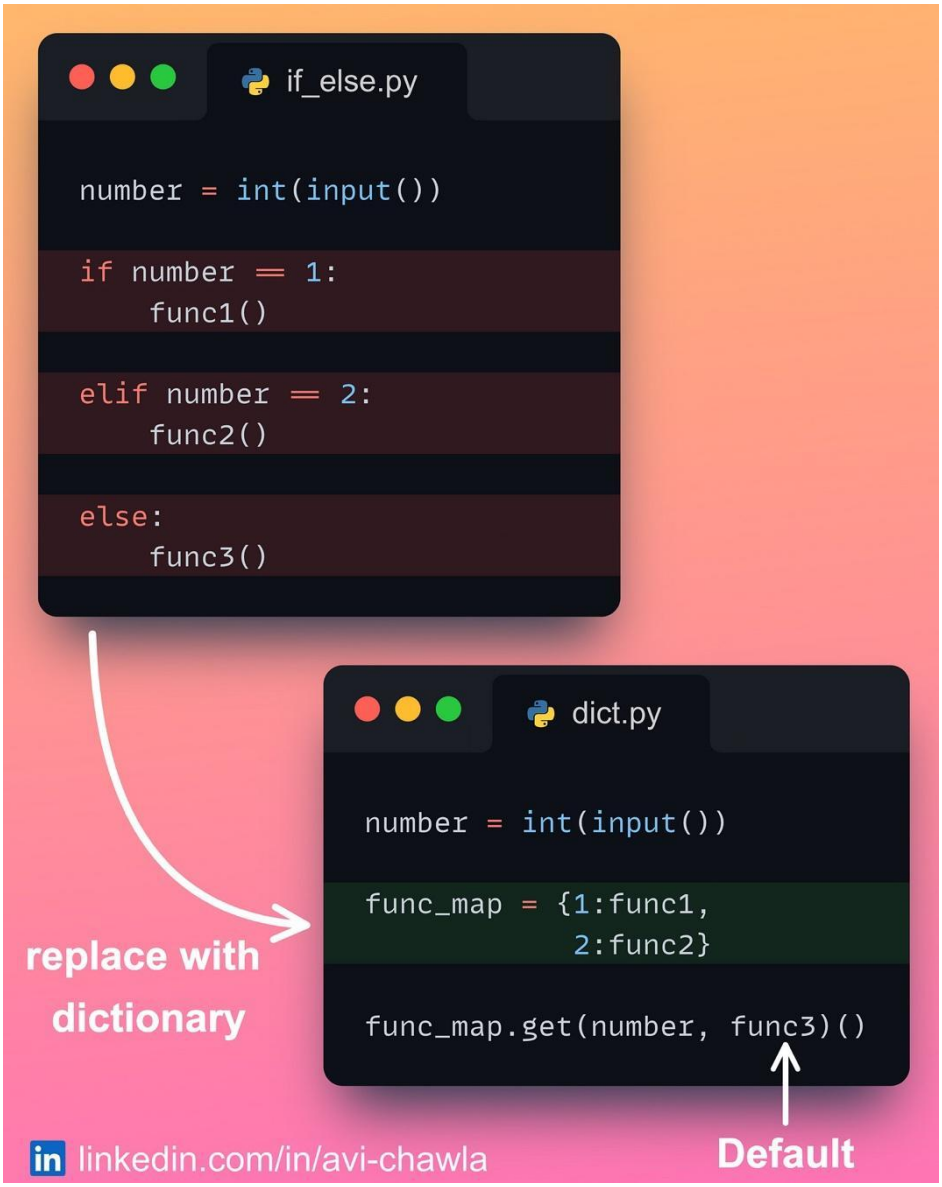
Yet, across projects, we often use the same methods over and over to analyze our data. This quickly gets repetitive and time-consuming.

To avoid this, use Mito. It's an incredible tool that allows you to analyze your data within a spreadsheet interface in Jupyter, without writing any code.

The coolest thing about Mito is that each edit in the spreadsheet automatically generates an equivalent Python code. This makes it extremely convenient to reproduce the analysis later.

Read more: Documentation.

Using Dictionaries In Place of If-conditions



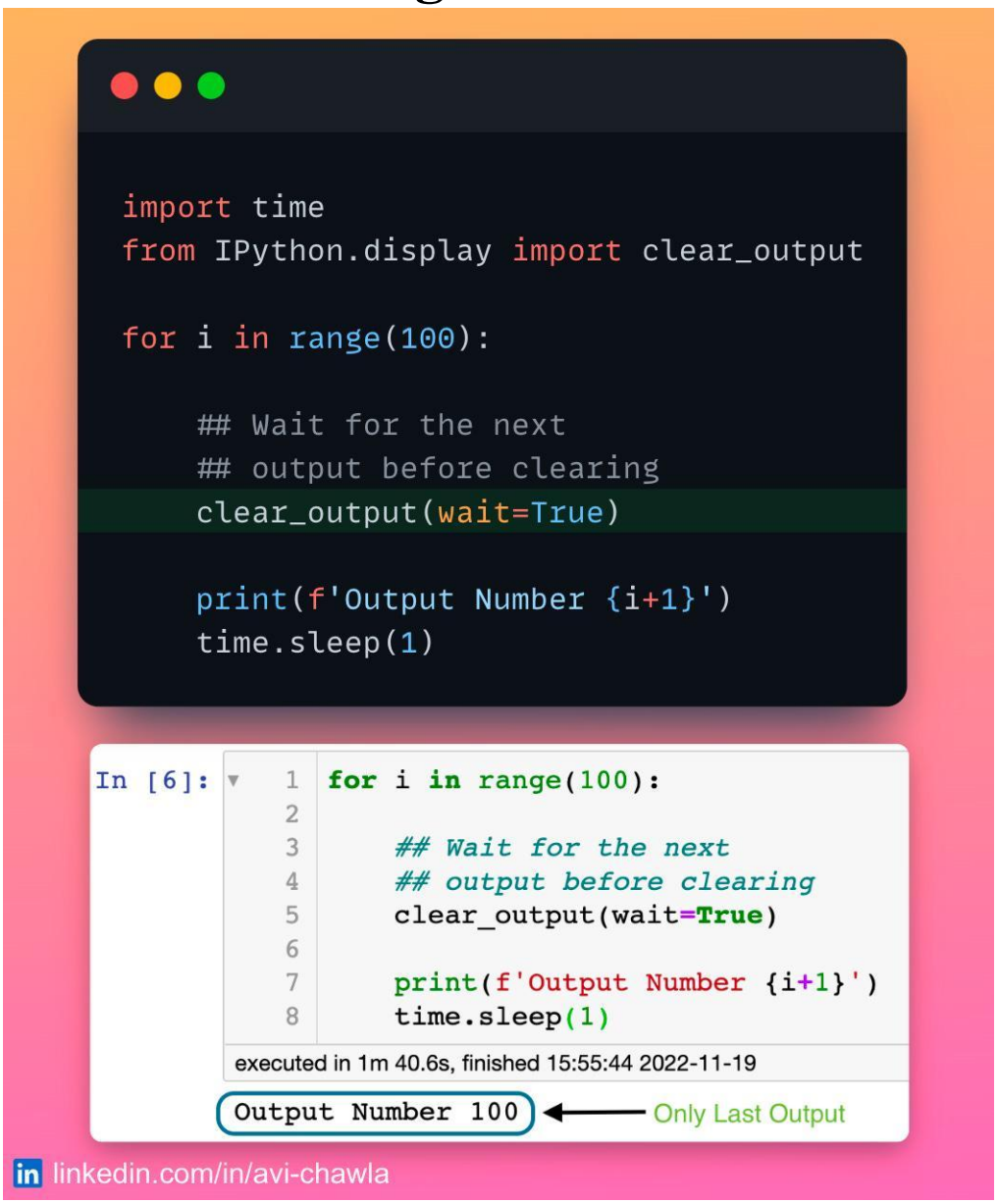
Dictionaries are mainly used as a data structure in Python for maintaining key-value pairs.

However, there's another special use case that dictionaries can handle. This is — Eliminating IF conditions from your code.

Consider the code snippet above. Here, corresponding to an input value, we invoke a specific function. The traditional way requires you to hard-code every case.

But with a dictionary, you can directly retrieve the corresponding function by providing it with the key. This makes your code concise and elegant.

Clear Cell Output In Jupyter Notebook During Run-time



While using Jupyter, we often print many details to track the code's progress.

However, it gets frustrating when the output panel has accumulated a bunch of details, but we are only interested in the most recent output. Moreover, scrolling to the bottom of the output each time can be annoying too.

To clear the output of the cell, you can use the **clear_output** method from the **IPython** package. When invoked, it will remove the current output of the cell, after which you can print the latest details.

A Hidden Feature of Describe Method In Pandas

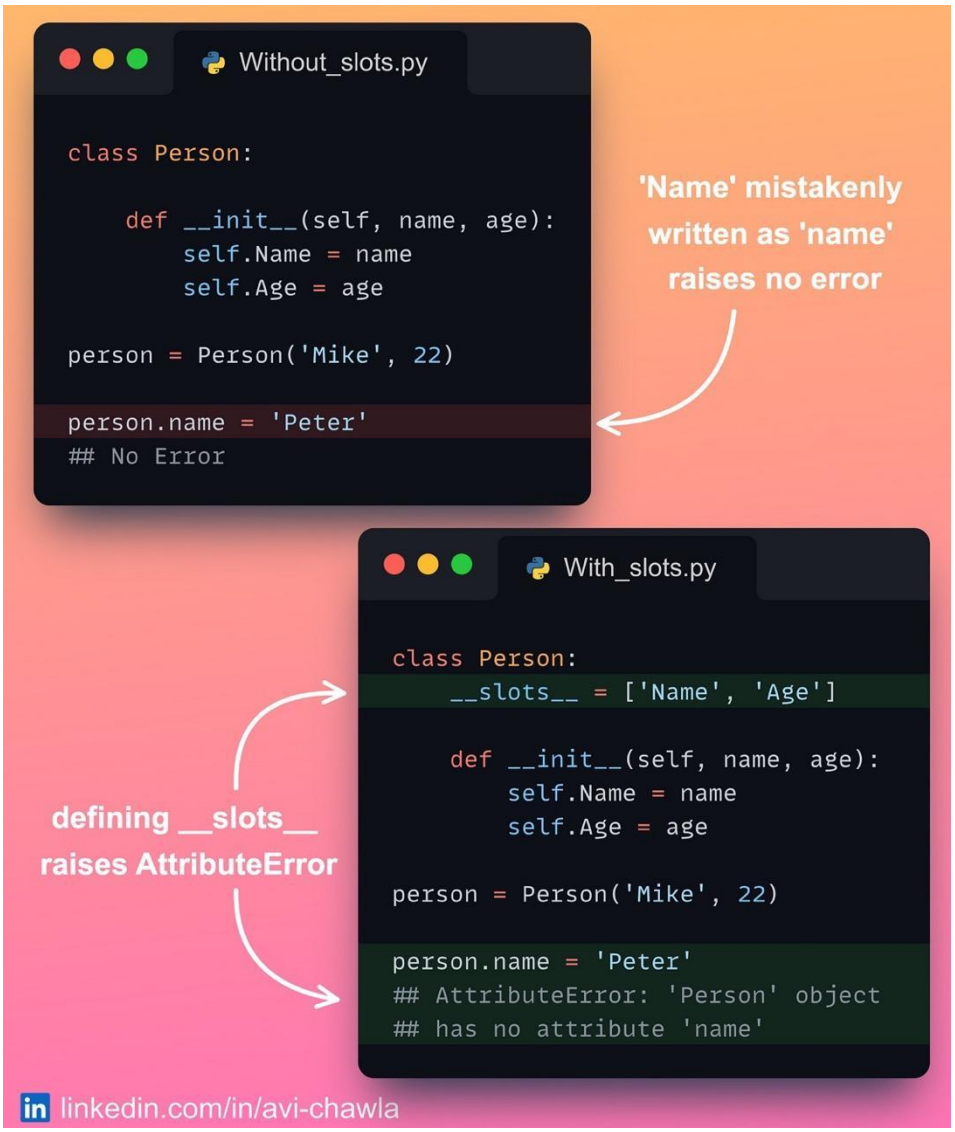
The **describe**() method in Pandas is commonly used to print descriptive statistics about the data.

But have you ever noticed that its output is always limited to numerical columns? Of course, details like mean, median, std. dev., etc. hold no meaning for non-numeric columns, so the results make total sense.

However, **describe**() can also provide a quick summary of non-numeric columns. You can do this by specifying **include**="**all**." As a result, it will return the number of unique elements, the top element with its frequency.

Read more: Documentation.

Use Slotted Class To Improve Your Python Code



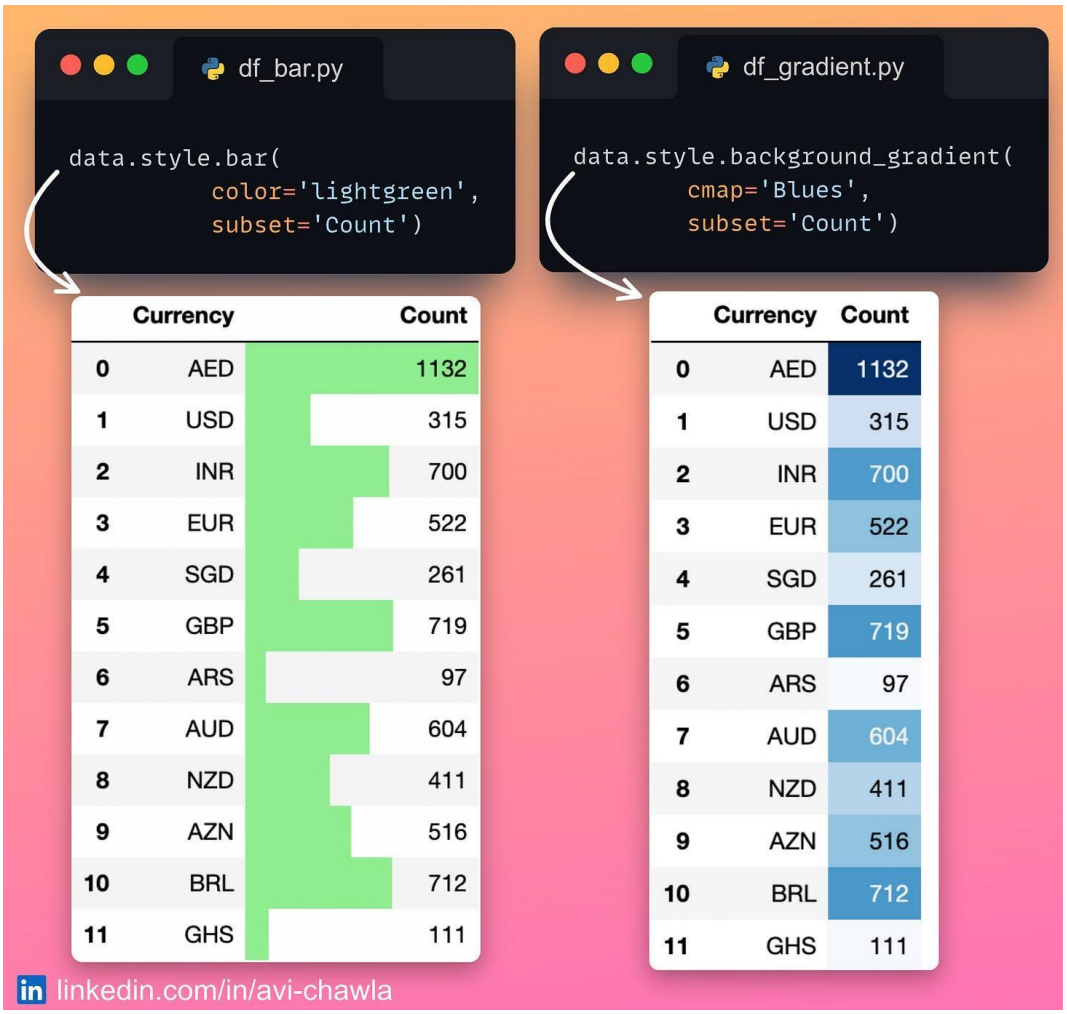
If you want to fix the attributes a class can hold, consider defining it as a slotted class.

While defining classes, **__slots__** allows you to explicitly specify the class attributes. This means you cannot randomly add new attributes to a slotted class object. This offers many advantages.

For instance, slotted classes are memory efficient and they provide faster access to class attributes. What's more, it also helps you avoid common typos. This, at times, can be a costly mistake that can go unnoticed.

Read more: StackOverflow.

Stop Analysing Raw Tables. Use Styling Instead!



Jupyter is a web-based IDE. Thus, whenever you print/display a DataFrame in Jupyter, it is rendered using HTML and CSS.

This means you can style your output in many different ways.

To do so, use the Styling API of Pandas. Here, you can make many different modifications to a DataFrame's styler object (**df.style**). As a result, the DataFrame will be displayed with the specified styling.

Styling makes these tables visually appealing. Moreover, it allows for better comprehensibility of data than viewing raw tables.

Read more here: Documentation.

Explore CSV Data Right From The Terminal

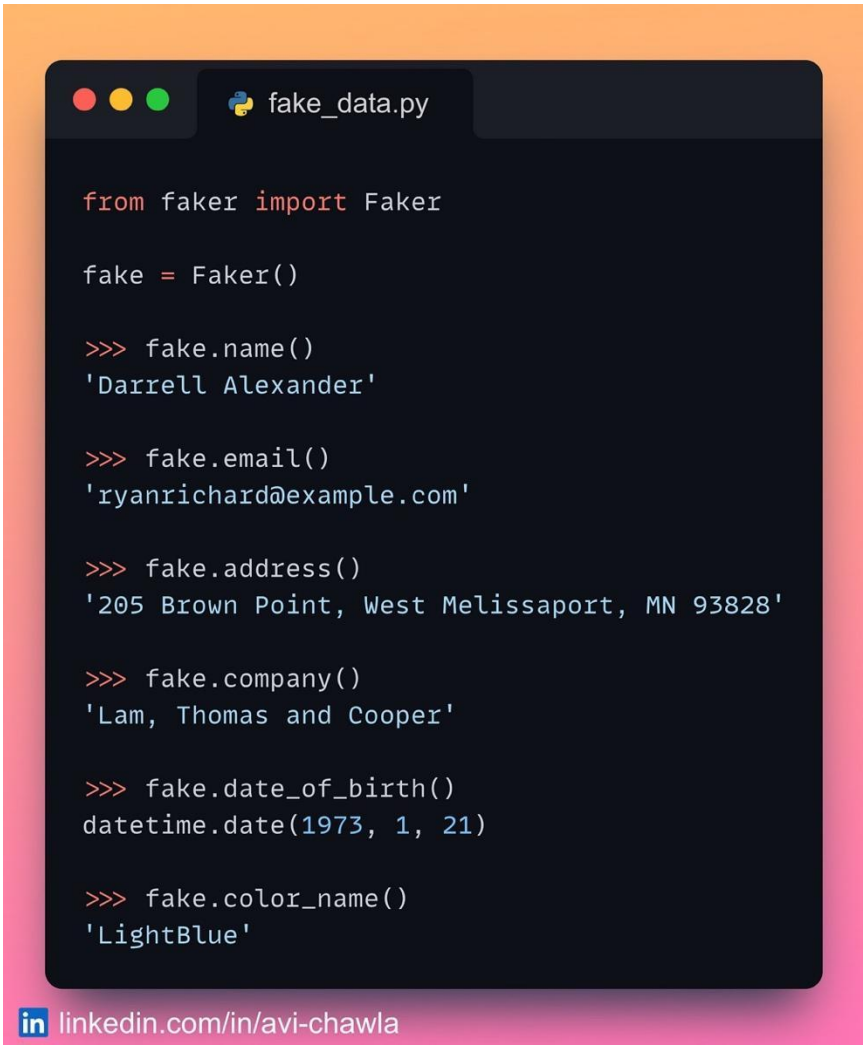
If you want to quickly explore some CSV data, you may not always need to run a Jupyter session.

Rather, with "**csvkit**", you can do it from the terminal itself. As the name suggests, it provides a bunch of command-line tools to facilitate data analysis tasks.

These include converting Excel to CSV, viewing column names, data statistics, and querying using SQL. Moreover, you can also perform popular Pandas functions such as sorting, merging, and slicing.

Read more: Documentation.

Generate Your Own Fake Data In Seconds



Usually, for executing/testing a pipeline, we need to provide it with some dummy data.

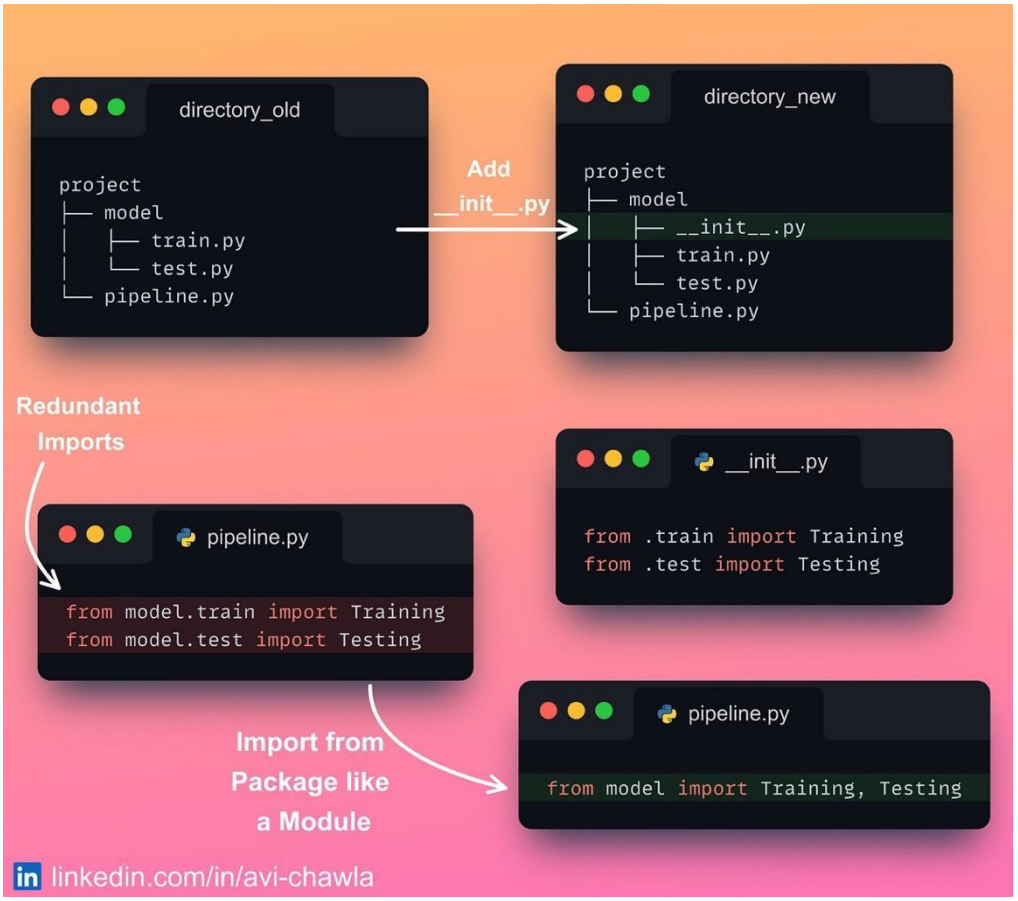
Although using Python's "**random**" library, one can generate random strings, floats, and integers. Yet, being random, it does not output any meaningful data such as people's names, city names, emails, etc.

Here, looking for open-source datasets can get time-consuming. Moreover, it's possible that the dataset you find does not fit pretty well into your requirements.

The **Faker** module in Python is a perfect solution to this. Faker allows you to generate highly customized fake (yet meaningful) data quickly. What's more, you can also generate data specific to a demographic.

Read more here: Documentation.

Import Your Python Package as a Module



A python module is a single python file (.**py**). An organized collection of such python files is called a python package.

While developing large projects, it is a good practice to define an **__init__.py** file inside a package.

Consider **train.py** has a **Training** class and **test.py** has a **Testing** class.

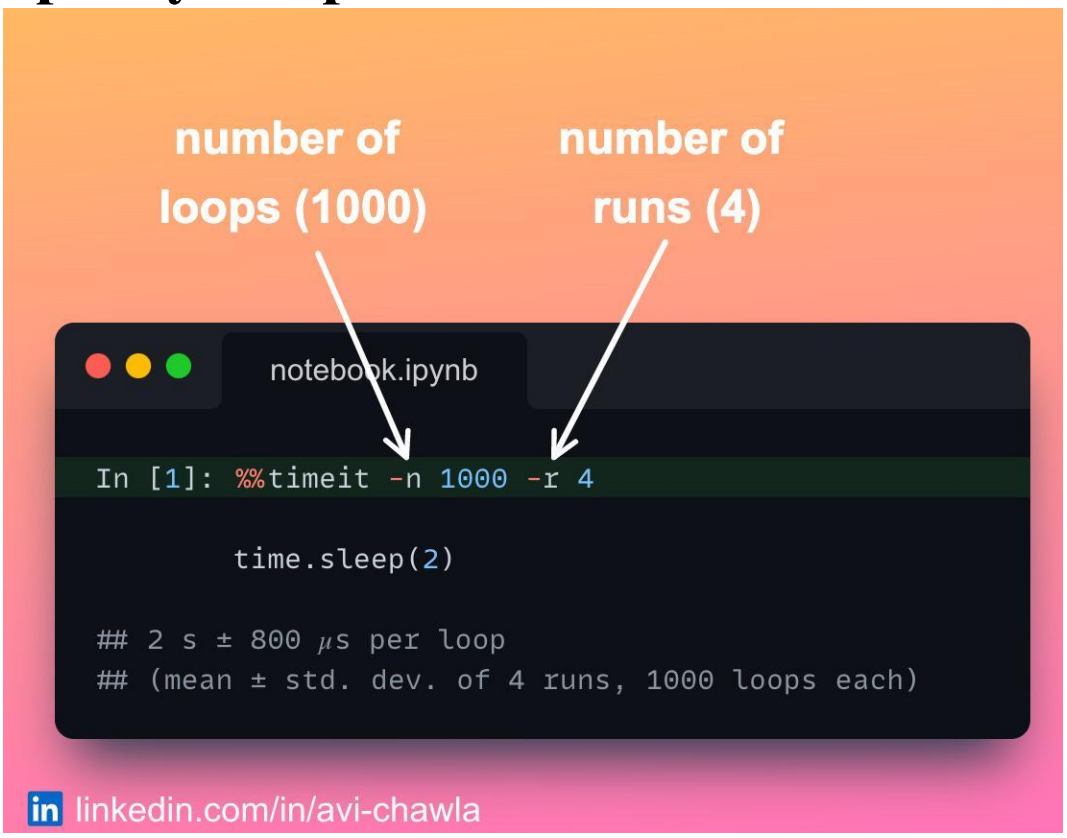
Without **__init__.py**, one has to explicitly import them from specific python files. As a result, it is redundant to write the two import statements.

With **__init__.py**, you can group python files into a single importable module. In other words, it provides a mechanism to treat the whole package as a python module.

This saves you from writing redundant import statements and makes your code cleaner in the calling script.

Read more in this blog: Blog Link.

Specify Loops and Runs In %%timeit

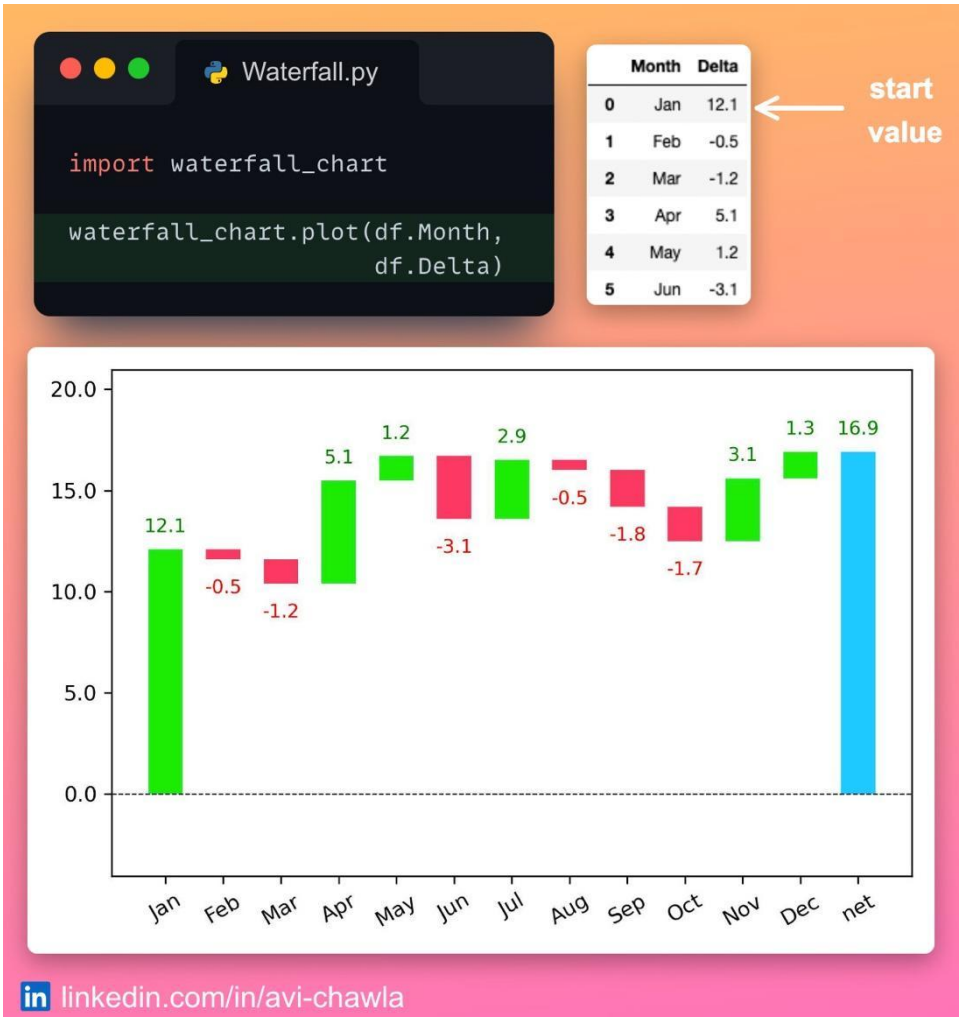


We commonly use the %**timeit** (or %%**timeit**) magic command to measure the execution time of our code.

Here, **timeit** limits the number of runs depending on how long the script takes to execute. This is why you get to see a different number of loops (and runs) across different pieces of code.

However, if you want to explicitly define the number of loops and runs, use the -**n** and -**r** options. Use -**n** to specify the loops and -**r** for the number the runs.

Waterfall Charts: A Better Alternative to Line/Bar Plot



If you want to visualize a value over some period, a line (or bar) plot may not always be an apt choice.

A line-plot (or bar-plot) depicts the actual values in the chart. Thus, sometimes, it can get difficult to visually estimate the scale of incremental changes.

Instead, you can use a waterfall chart, which elegantly depicts these rolling differences.

To create one, you can use **waterfall_chart** in Python. Here, the start and final values are represented by the first and last bars. Also, the marginal changes are automatically color-coded, making them easier to interpret.

Read more here: GitHub.

Hexbin Plots As A Richer Alternative to Scatter Plots

Scatter plots are extremely useful for visualizing two sets of numerical variables. But when you have, say, thousands of data points, scatter plots can get too dense to interpret.

Hexbins can be a good choice in such cases. As the name suggests, they bin the area of a chart into hexagonal regions. Each region is assigned a color intensity based on the method of aggregation used (the number of points, for instance).

Hexbins are especially useful for understanding the spread of data. It is often considered an elegant alternative to a scatter plot. Moreover, binning makes it easier to identify data clusters and depict patterns.

Importing Modules Made Easy with Pyforest

The typical programming-related stuff in data science begins by importing relevant modules.

However, across notebooks/projects, the modules one imports are mostly the same. Thus, the task of importing all the individual libraries is kinda repetitive.

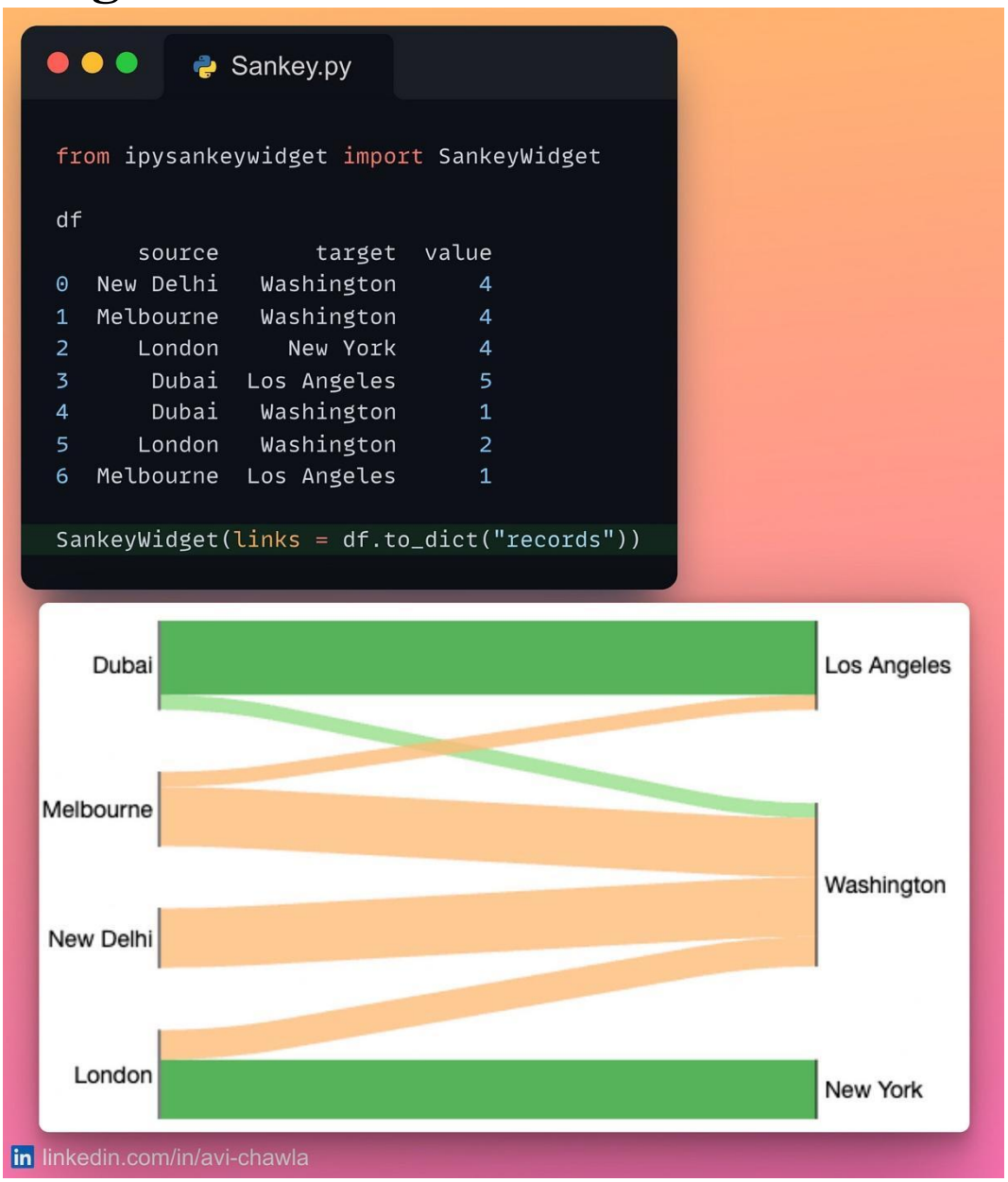
With **pyforest**, you can use the common Python libraries without explicitly importing them. A good thing is that it imports all the libraries with their standard conventions. For instance, **pandas** is imported with the **pd** alias.

With that, you should also note that it is a good practice to keep Pyforest limited to prototyping stages. This is because once you say, develop and open-source your pipeline, other users may face some difficulties understanding it.

But if you are up for some casual experimentation, why not use it instead of manually writing all the imports?

Read more: GitHub.

Analyse Flow Data With Sankey Diagrams



Many tabular data analysis tasks can be interpreted as a flow between the source and a target.

Here, manually analyzing tabular reports/data to draw insights is typically not the right approach.

Instead, Flow diagrams serve as a great alternative in such cases.

Being visually appealing, they immensely assist you in drawing crucial insights from your data, which you may find challenging to infer by looking at the data manually.

For instance, from the diagram above, one can quickly infer that:

1. Washington hosts flights from all origins.

2. New York only receives passengers from London.

3. Majority of flights in Los Angeles come from Dubai.

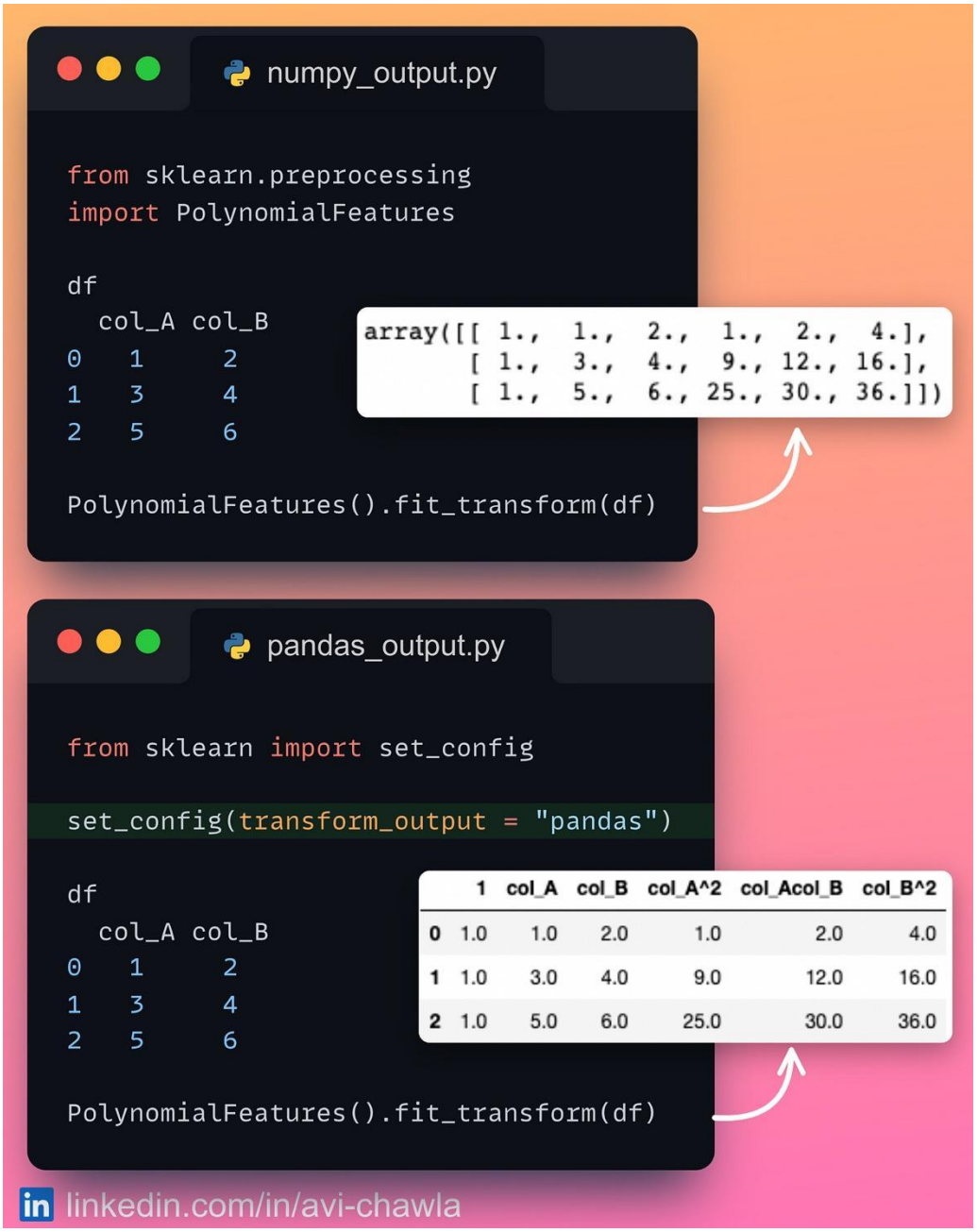
4. All flights from New Delhi go to Washington.

Now imagine doing that by just looking at the tabular data. Not only will it be time-consuming, but there are chances that you may miss out on a few insights.

To generate a flow diagram, you can use floWeaver. It helps you to visualize flow data using Sankey diagrams.

Read more here: Documentation.

Feature Tracking Made Simple In Sklearn Transformers



Recently, **scikit-learn** announced the release of one of the most awaited improvements. In a gist, sklearn can now be configured to output Pandas DataFrames.

Until now, Sklearn's transformers were configured to accept a Pandas DataFrame as input. But they always returned a NumPy array as an output. As a result, the output had to be manually projected back to a Pandas DataFrame. This, at times, made it difficult to track and assign names to the features.

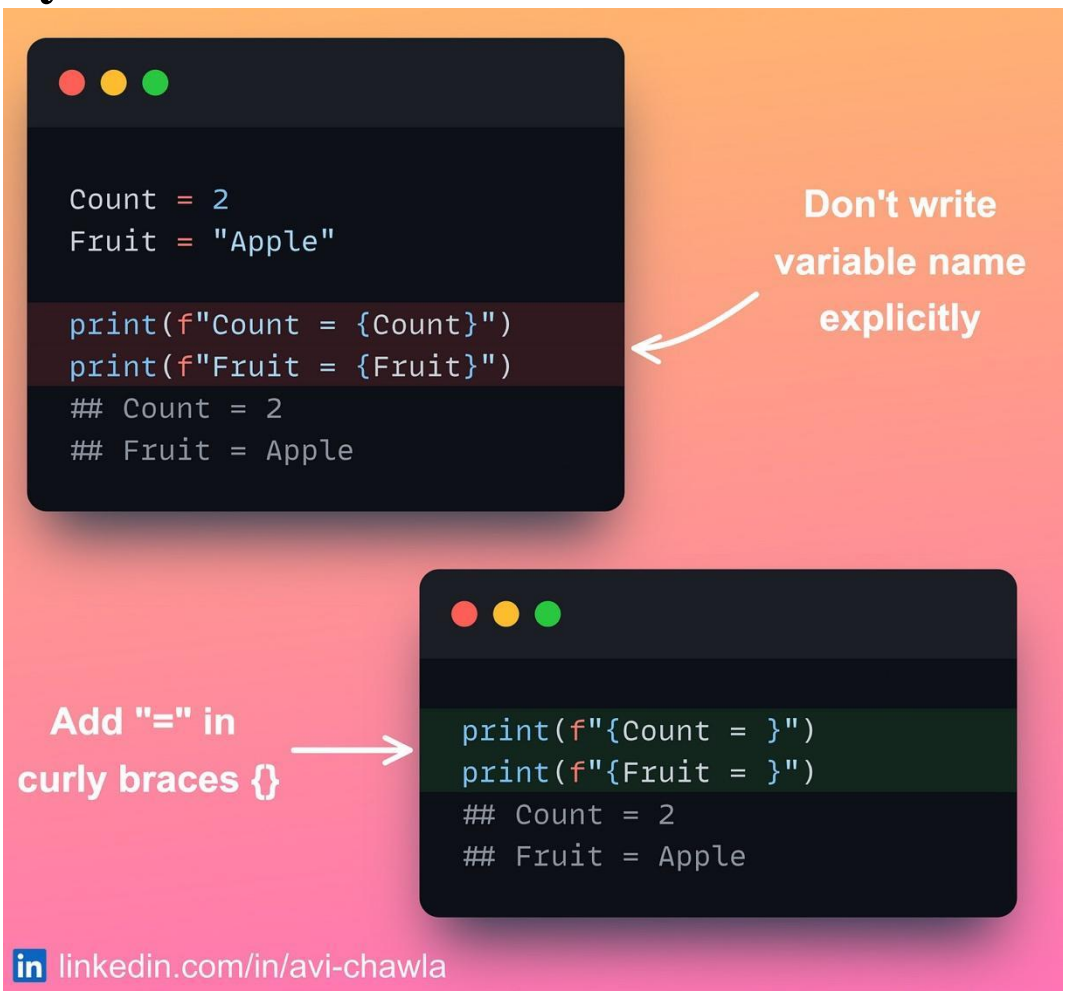
For instance, consider the snippet above.

In **numpy_output.py**, it is tricky to infer the name (or computation) of a column by looking at the NumPy array.

However, in the upcoming release, the transformer can return a Pandas DataFrame (**pandas_output.py**). This makes tracking feature names incredibly simple.

Read more: Release page.

Lesser-known Feature of f-strings in Python



While debugging, one often explicitly prints the name of the variable with its value to enhance code inspection.

Although there's nothing wrong with this approach, it makes your print statements messy and lengthy.

f-strings in Python offer an elegant solution for this.

To print the name of the variable, you can add an equals sign (=) in the curly braces after the variable. This will print the name of the variable along with its value but it is concise and clean.

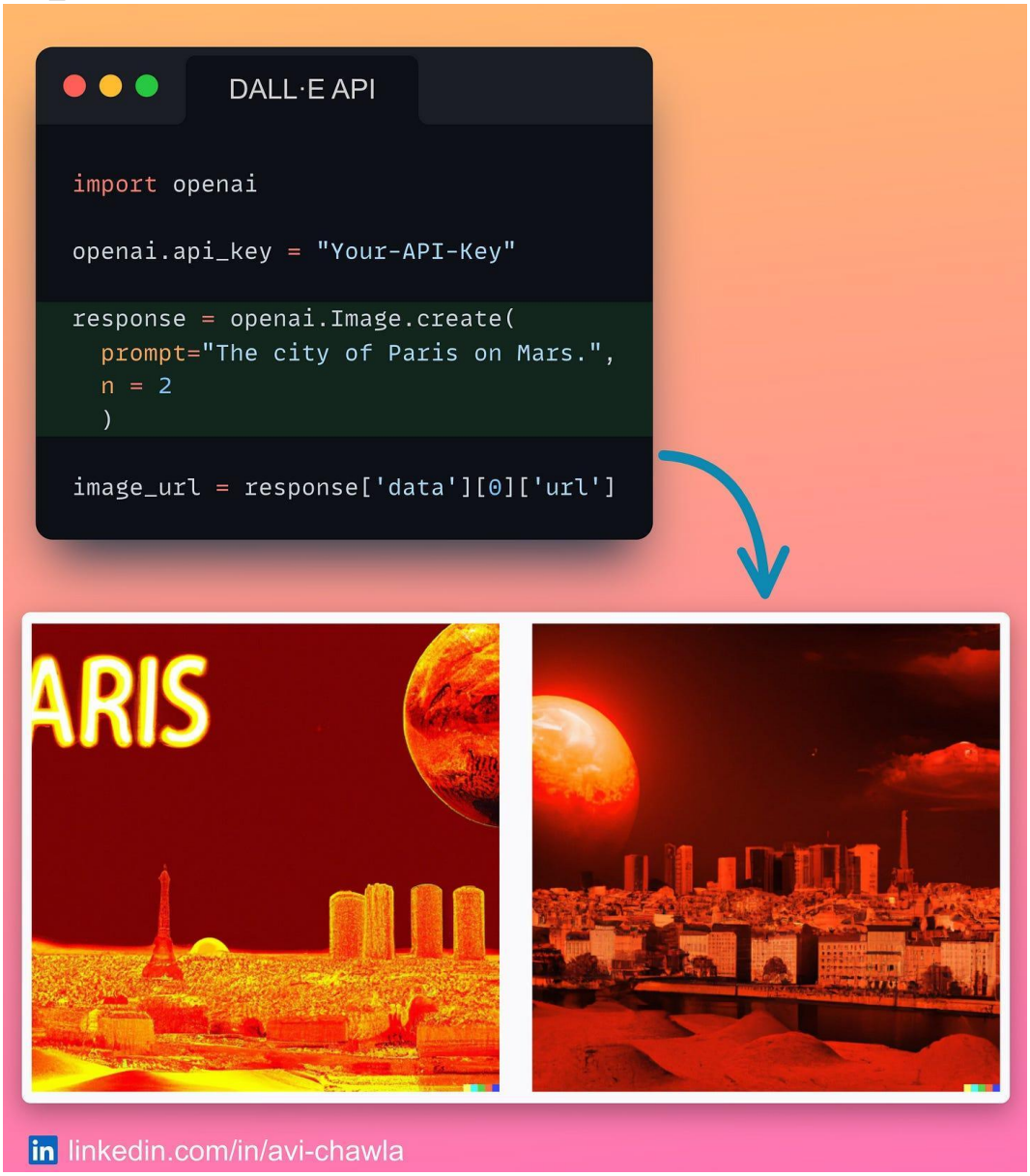
Don't Use time.time() To Measure Execution Time

The **time**() method from the time library is frequently used to measure the execution time.

However, **time**() is not meant for timing your code. Rather, its actual purpose is to tell the current time. This, at many times, compromises the accuracy of measuring the exact run time.

The correct approach is to use **perf_counter**(), which deals with relative time. Thus, it is considered the most accurate way to time your code.

Now You Can Use DALL·E With OpenAI API

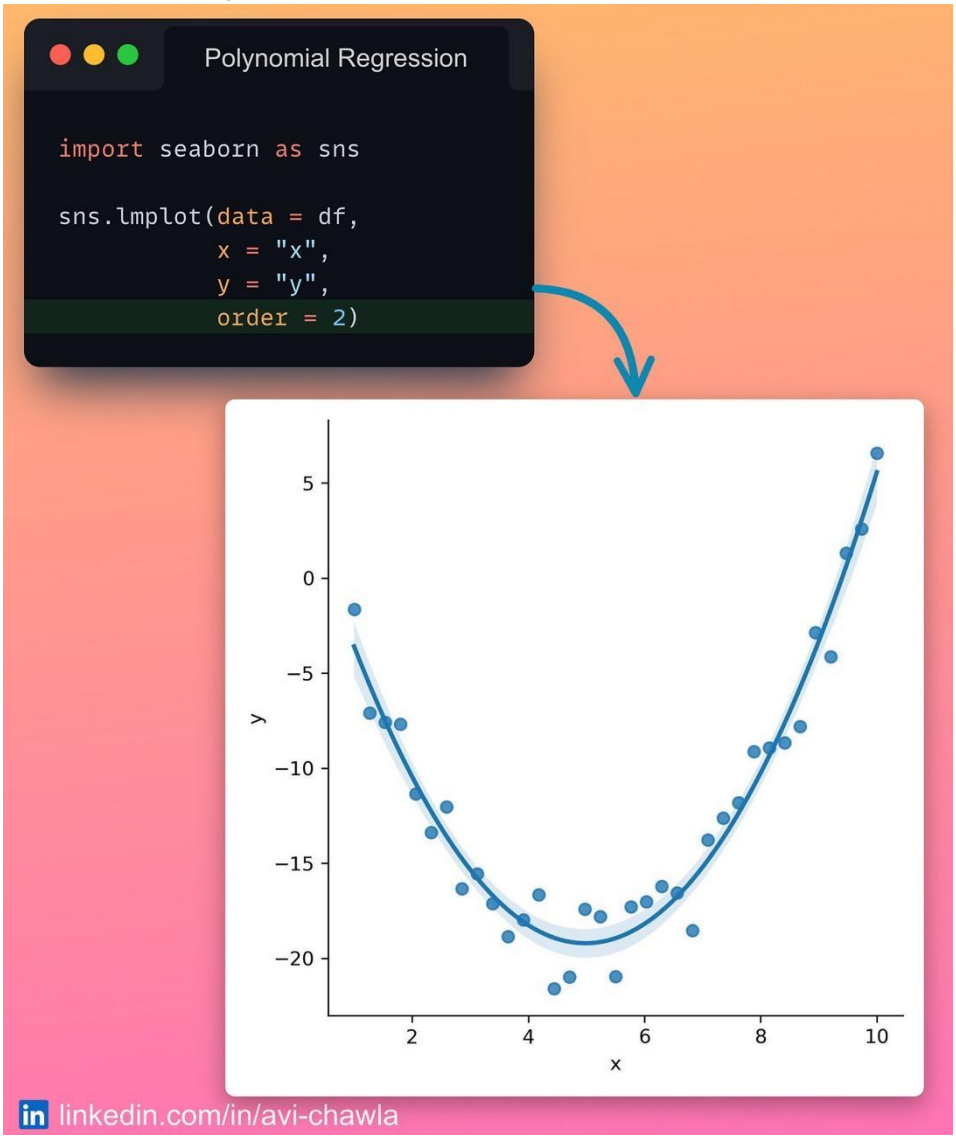


DALL·E is now accessible using the OpenAI API.

OpenAI recently made a big announcement. In a gist, developers can now integrate OpenAI's popular text-to-image model DALL·E into their apps using OpenAI API.

To achieve this, first, specify your API key (obtained after signup). Next, pass a text prompt to generate the corresponding image.

Polynomial Linear Regression Plot Made Easy With Seaborn



While creating scatter plots, one is often interested in displaying a linear regression (simple or polynomial) fit on the data points.

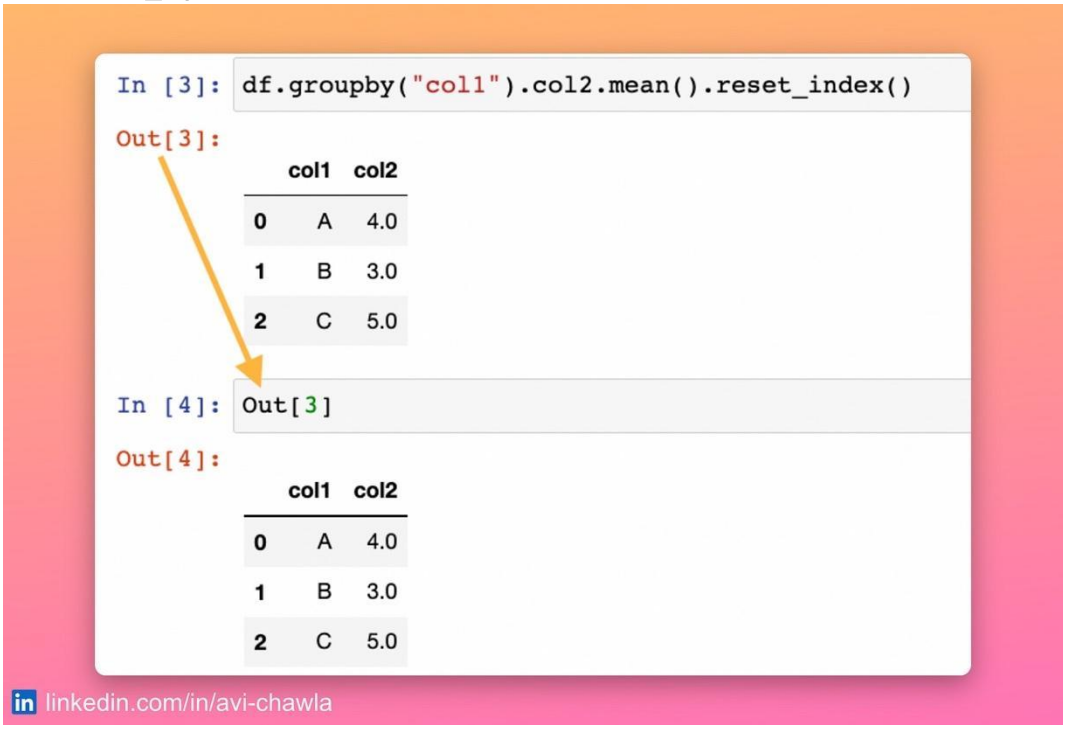
Here, training a model and manually embedding it in the plot can be a tedious job to do.

Instead, with Seaborn's **lmplot**(), you can add a regression fit to a plot, without explicitly training a model.

Specify the degree of the polynomial as the "**order**" parameter. Seaborn will add the corresponding regression fit on the scatter plot.

Read more here: Seaborn Docs.

Retrieve Previously Computed Output In Jupyter Notebook



This is indeed one of the coolest things I have learned about Jupyter Notebooks recently.

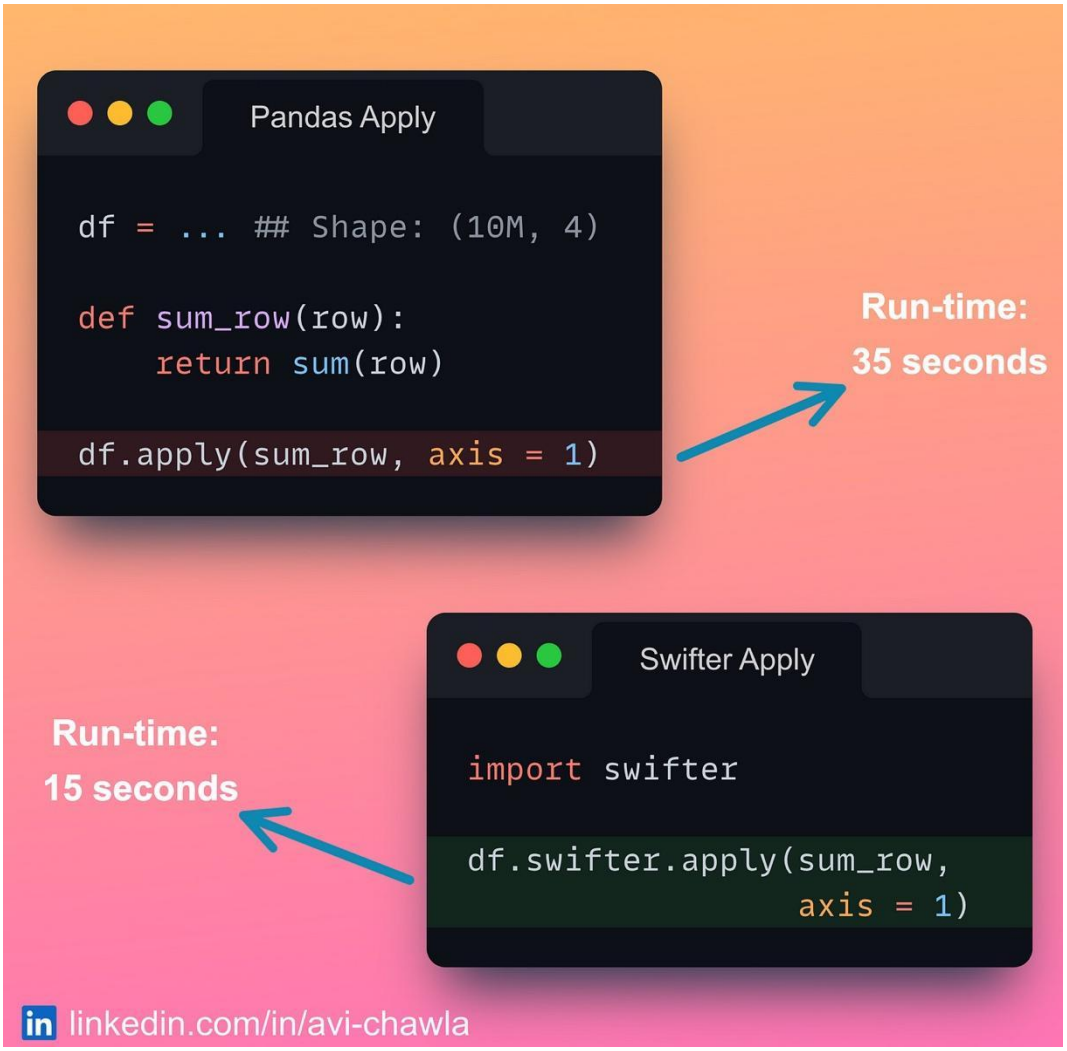
Have you ever been in a situation where you forgot to assign the results obtained after some computation to a variable? Left with no choice, one has to unwillingly recompute the result and assign it to a variable for further use.

Thankfully, you don't have to do that anymore!

IPython provides a dictionary "**Out**", which you can use to retrieve a cell's output. All you need to do is specify the cell number as the dictionary's key, which will return the corresponding output. Isn't that cool?

View a video version of this post on LinkedIn: Post Link.

Parallelize Pandas Apply() With Swifter



The Pandas library has no inherent support to parallelize its operations. Thus, it always adheres to a single-core computation, even when other cores are idle.

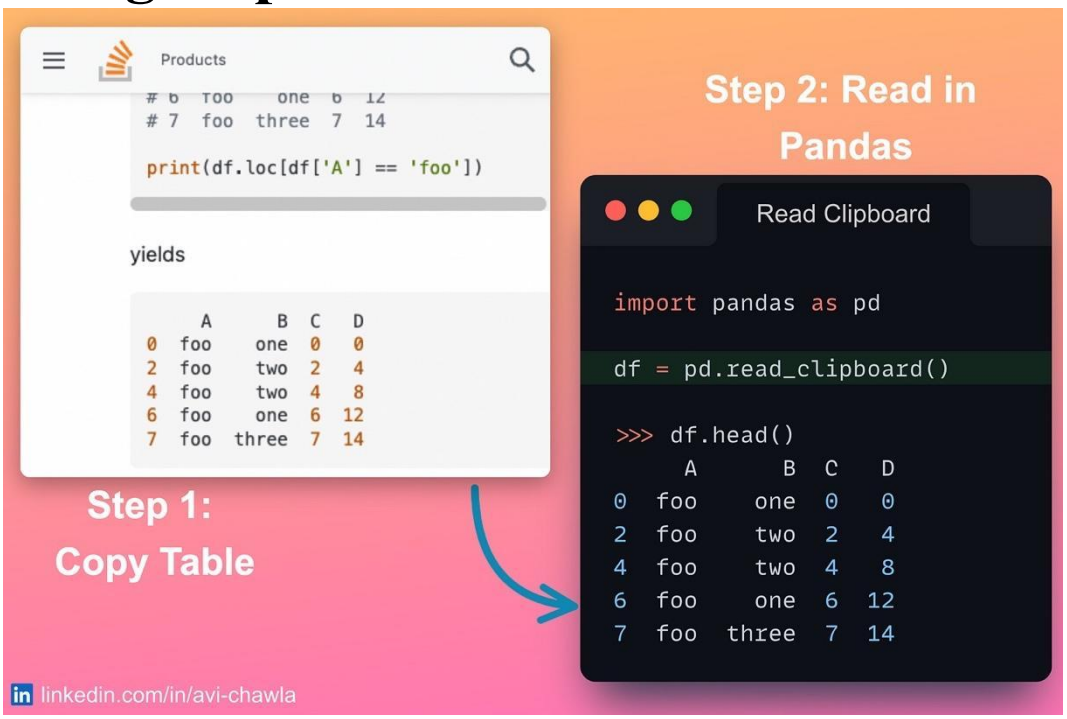
Things get even worse when we use **apply**(). In Pandas, **apply**() is nothing but a glorified for-loop. As a result, it cannot even take advantage of vectorization.

A quick solution to parallelize **apply**() is to use **swifter** instead.

Swifter allows you to apply any function to a Pandas DataFrame in a parallelized manner. As a result, it provides considerable performance gains while preserving the old syntax. All you have to do is use **df.swifter.apply** instead of **df.apply**.

Read more here: Swifter Docs.

Create DataFrame Hassle-free By Using Clipboard



Many Pandas users think that a DataFrame can ONLY be loaded from disk. However, this is not true.

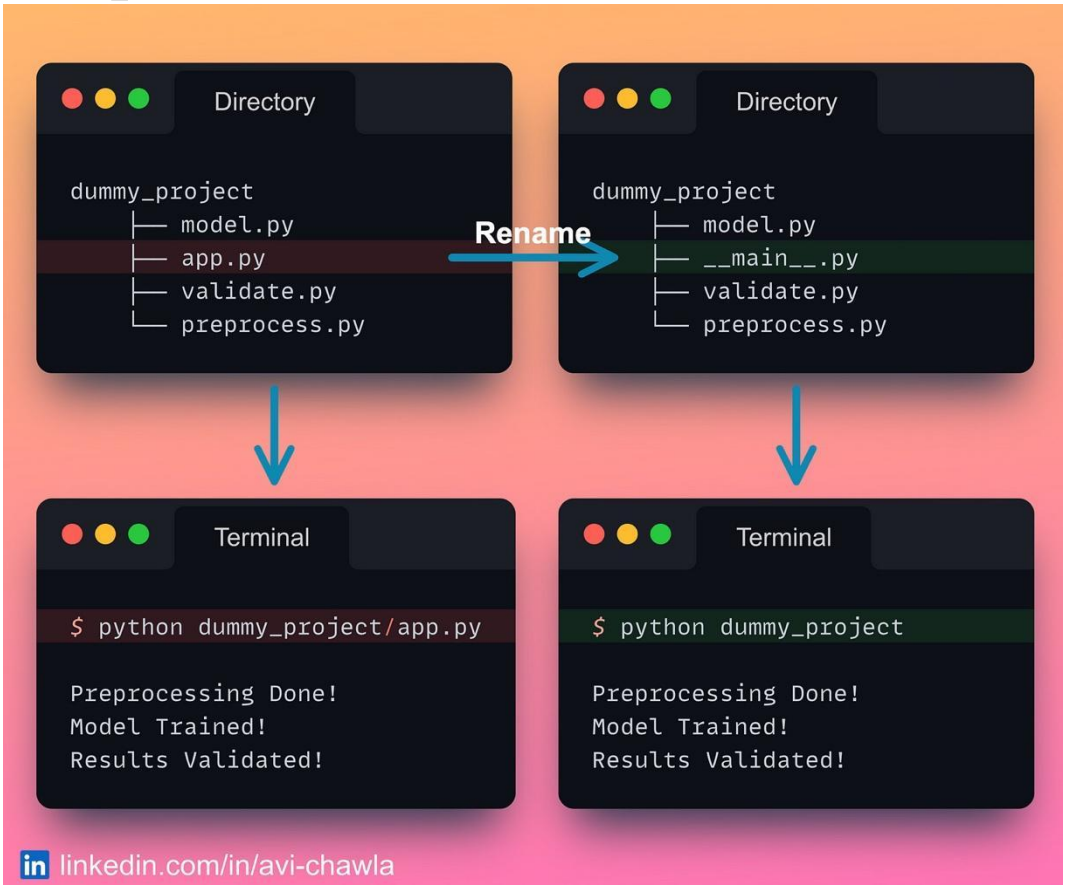
Imagine one wants to create a DataFrame from tabular data printed on a website. Here, they are most likely to be tempted to copy the contents to a CSV and read it using Pandas' **read_csv**() method. But this is not an ideal approach here.

Instead, with the **read_clipboard()** method, you can eliminate the CSV step altogether.

This method allows you to create a DataFrame from tabular data stored in a clipboard buffer. Thus, you just need to copy the data and invoke the method to create a DataFrame. This is an elegant approach that saves plenty of time.

Read more here: Pandas Docs.

Run Python Project Directory As A Script

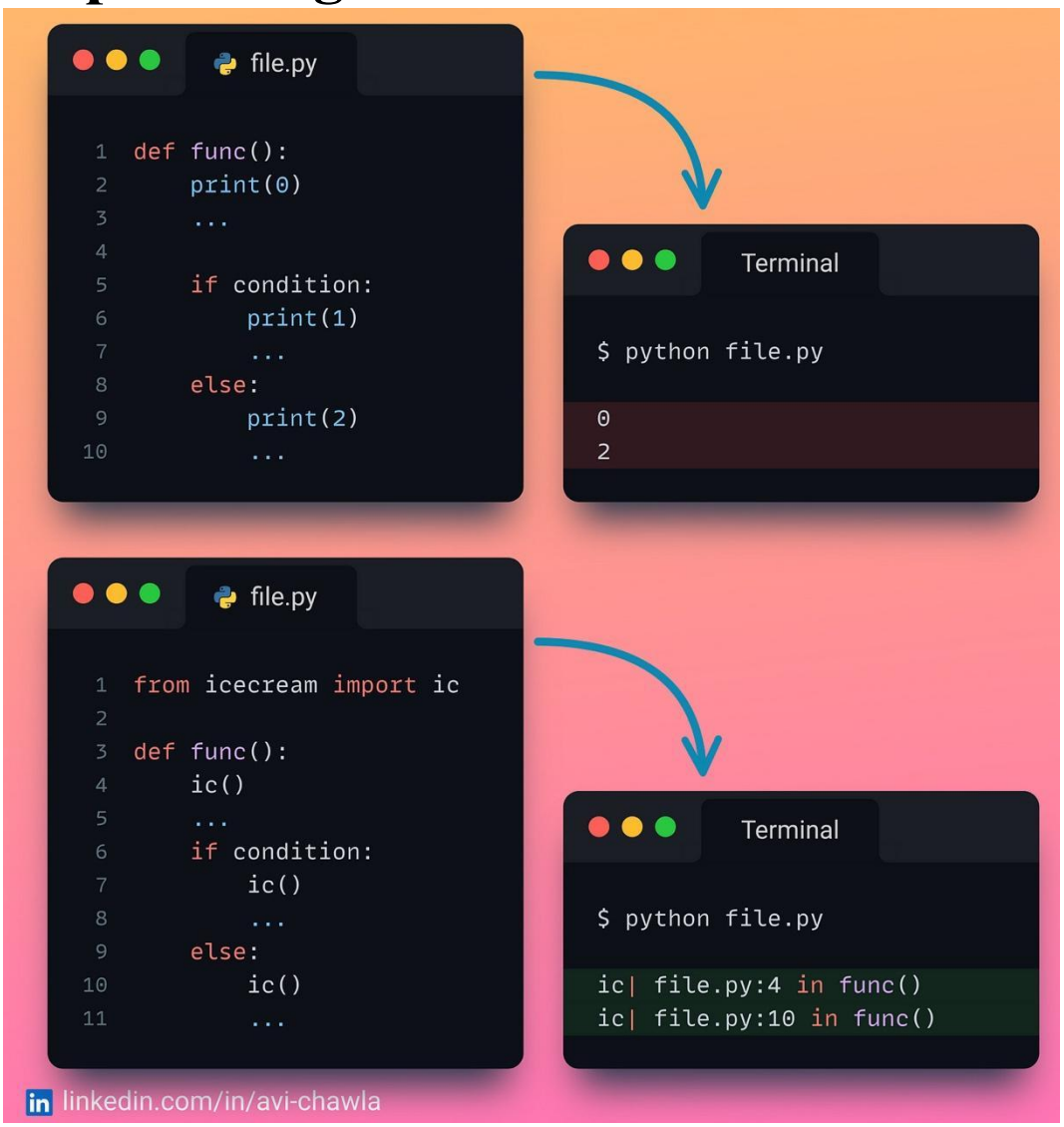


A Python script is executed when we run a .**py** file. In large projects with many files, there's often a source (or base) Python file we begin our program from.

To make things simpler, you can instead rename this base file to **__main__.py**. As a result, you can execute the whole pipeline by running the parent directory itself.

This is concise and also makes it slightly easier for other users to use your project.

Inspect Program Flow with IceCream



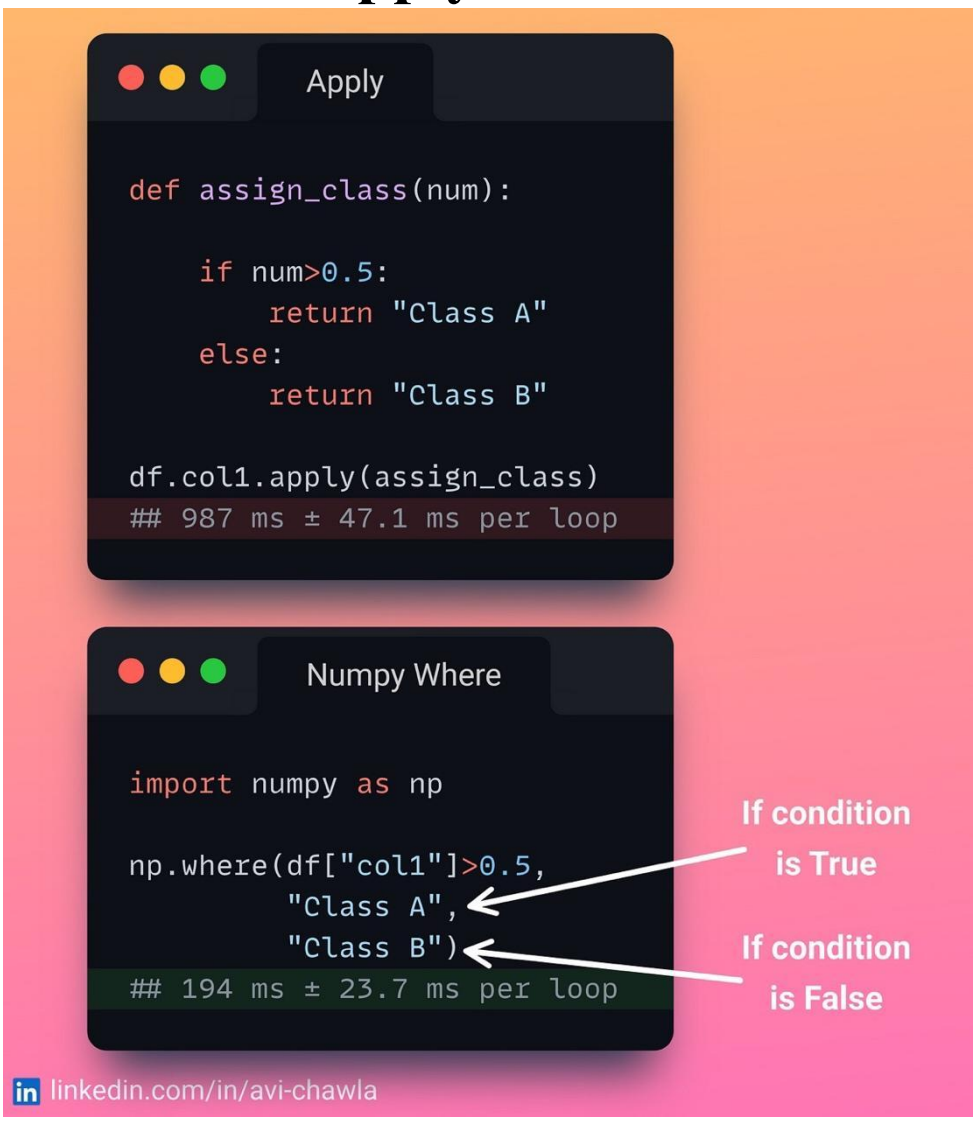
While debugging, one often writes many **print**() statements to inspect the program's flow. This is especially true when we have many IF conditions.

Using empty **ic**() statements from the IceCream library can be a better alternative here. It outputs many additional details that help in inspecting the flow of the program.

This includes the line number, the name of the function, the file name, etc.

Read more in my Medium Blog: Link.

Don't Create Conditional Columns in Pandas with Apply



While creating conditional columns in Pandas, we tend to use the **apply**() method almost all the time.

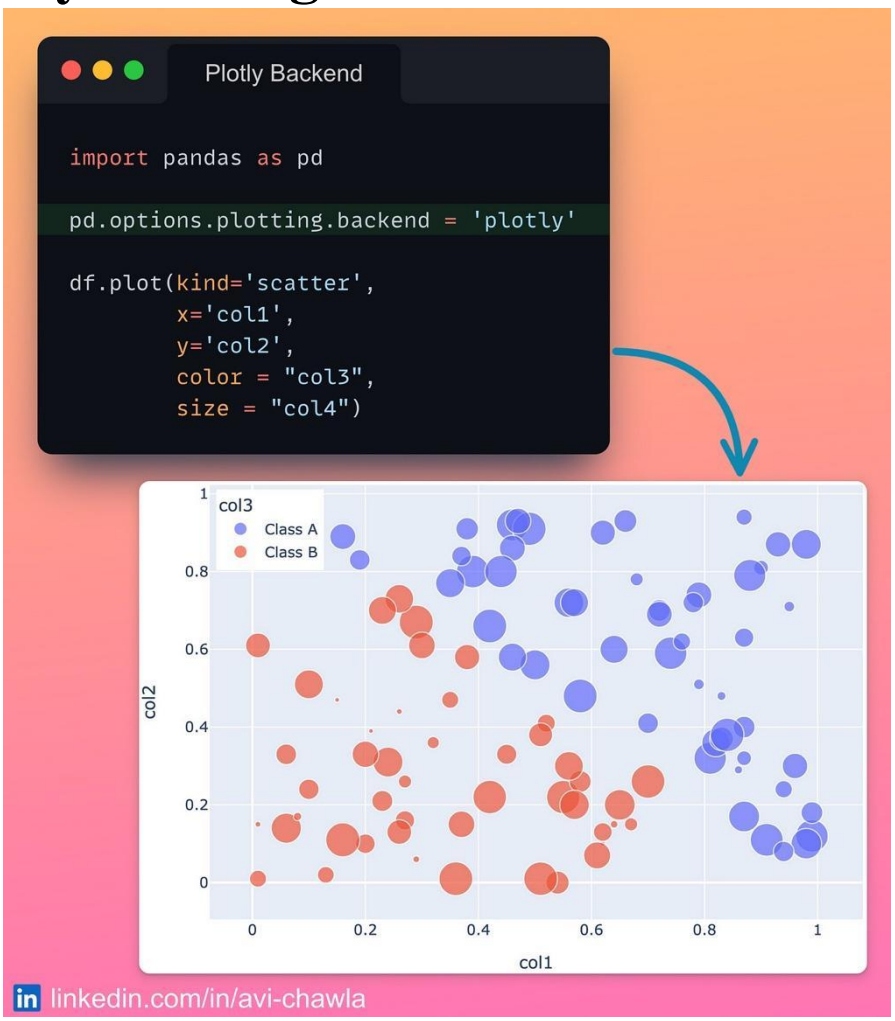
However, **apply**() in Pandas is nothing but a glorified for-loop. As a result, it misses the whole point of vectorization.

Instead, you should use the **np**.**where**() method to create conditional columns. It does the same job but is extremely fast.

The condition is passed as the first argument. This is followed by the result if the condition evaluates to True (second argument) and False (third argument).

Read more here: NumPy docs.

Pretty Plotting With Pandas



Matplotlib is the default plotting API of Pandas. This means you can create a Matplotlib plot in Pandas, without even importing it.

Despite that, these plots have always been basic and not so visually appealing. Plotly, with its pretty and interactive plots, is often considered a suitable alternative. But familiarising yourself with a whole new library and its syntax can be time-consuming.

Thankfully, Pandas does allow you to change the default plotting backend. Thus, you can leverage third-party visualization libraries for plotting with Pandas. This makes it effortless to create prettier plots while almost preserving the old syntax.

Build Baseline Models Effortlessly With Sklearn

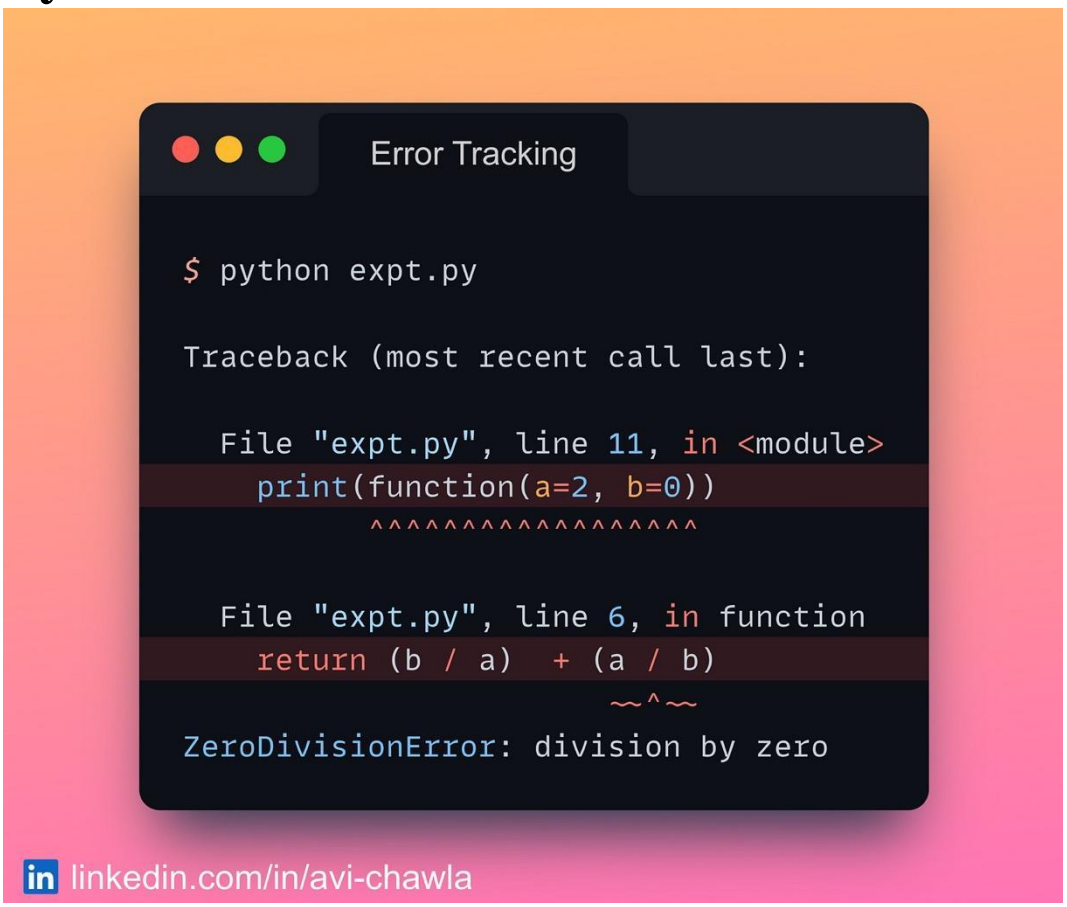
Before developing a complex ML model, it is always sensible to create a baseline first.

The baseline serves as a benchmark for the engineered model. Moreover, it ensures that the model is better than making random (or fixed) predictions. But building baselines with various strategies (random, fixed, most frequent, etc.) can be tedious.

Instead, Sklearn's **DummyClassifier**() (and **DummyRegressor**()) makes it totally effortless and straightforward. You can select the specific behavior of the baseline with the **strategy** parameter.

Read more here: Documentation.

Fine-grained Error Tracking With Python 3.11



Python 3.11 was released today, and many exciting features have been introduced.

For instance, various speed improvements have been implemented. As per the official release, Python 3.11 is, on average, 25% faster than Python 3.10. Depending on your work, it can be up to 10-60% faster.

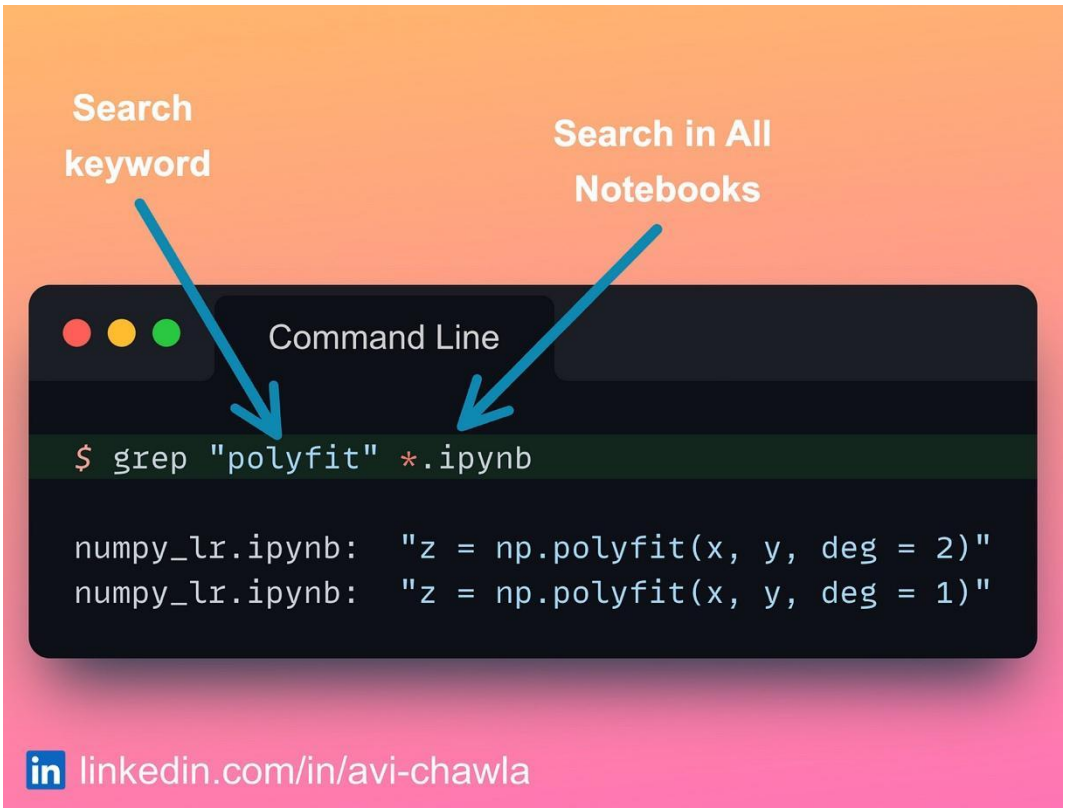
One of the coolest features is the fine-grained error locations in tracebacks.

In Python 3.10 and before, the interpreter showed the specific line that caused the error. This, at many times, caused ambiguity during debugging.

In Python 3.11, the interpreter will point to the exact location that caused the error. This will immensely help programmers during debugging.

Read more here: Official Release.

Find Your Code Hiding In Some Jupyter Notebook With Ease



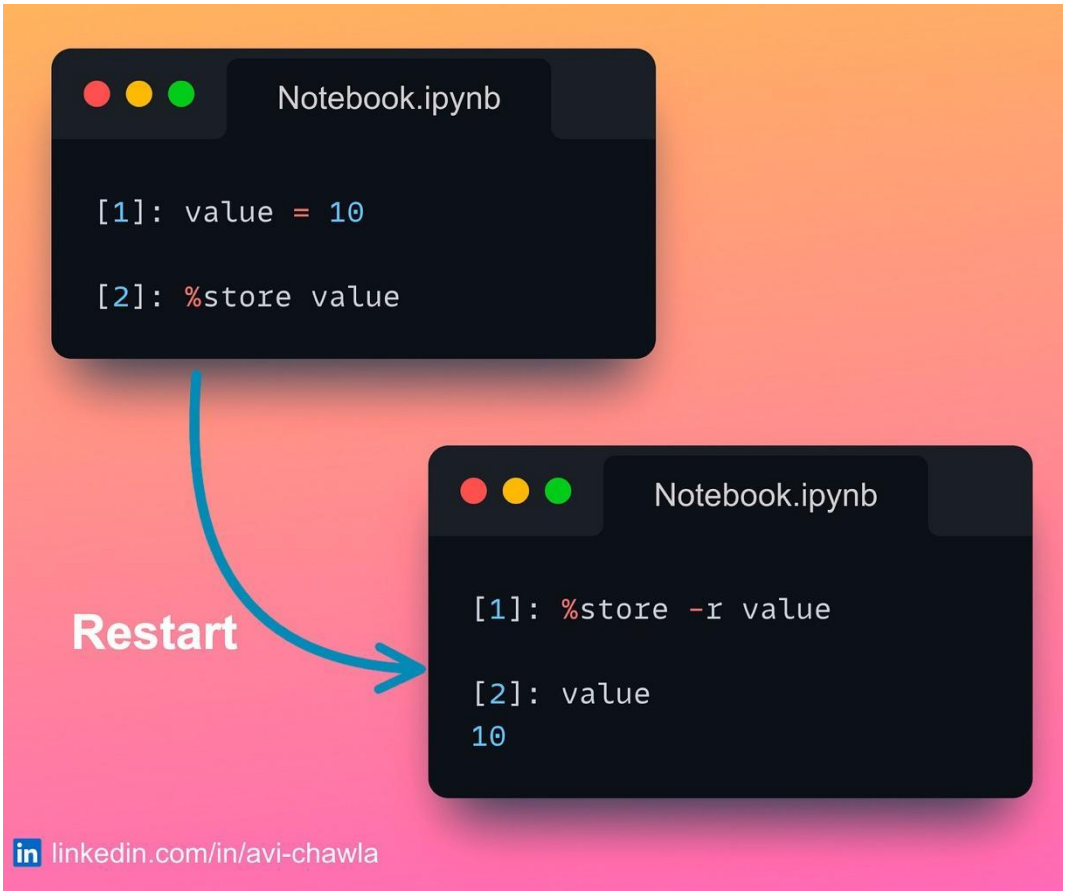
Programmers who use Jupyter often refer to their old notebooks to find a piece of code.

However, it gets tedious when they have multiple files to look for and can't recall the specific notebook of interest. The file name **Untitled1.ipynb**, ..., and **Untitled82.ipynb**, don't make it any easier.

The "**grep**" command is a much better solution to this. Very know that you can use "**grep**" in the command line to search in notebooks, as you do in other files (.txt, for instance). This saves plenty of manual work and time.

P.S. How do you find some previously written code in your notebooks (if not manually)?

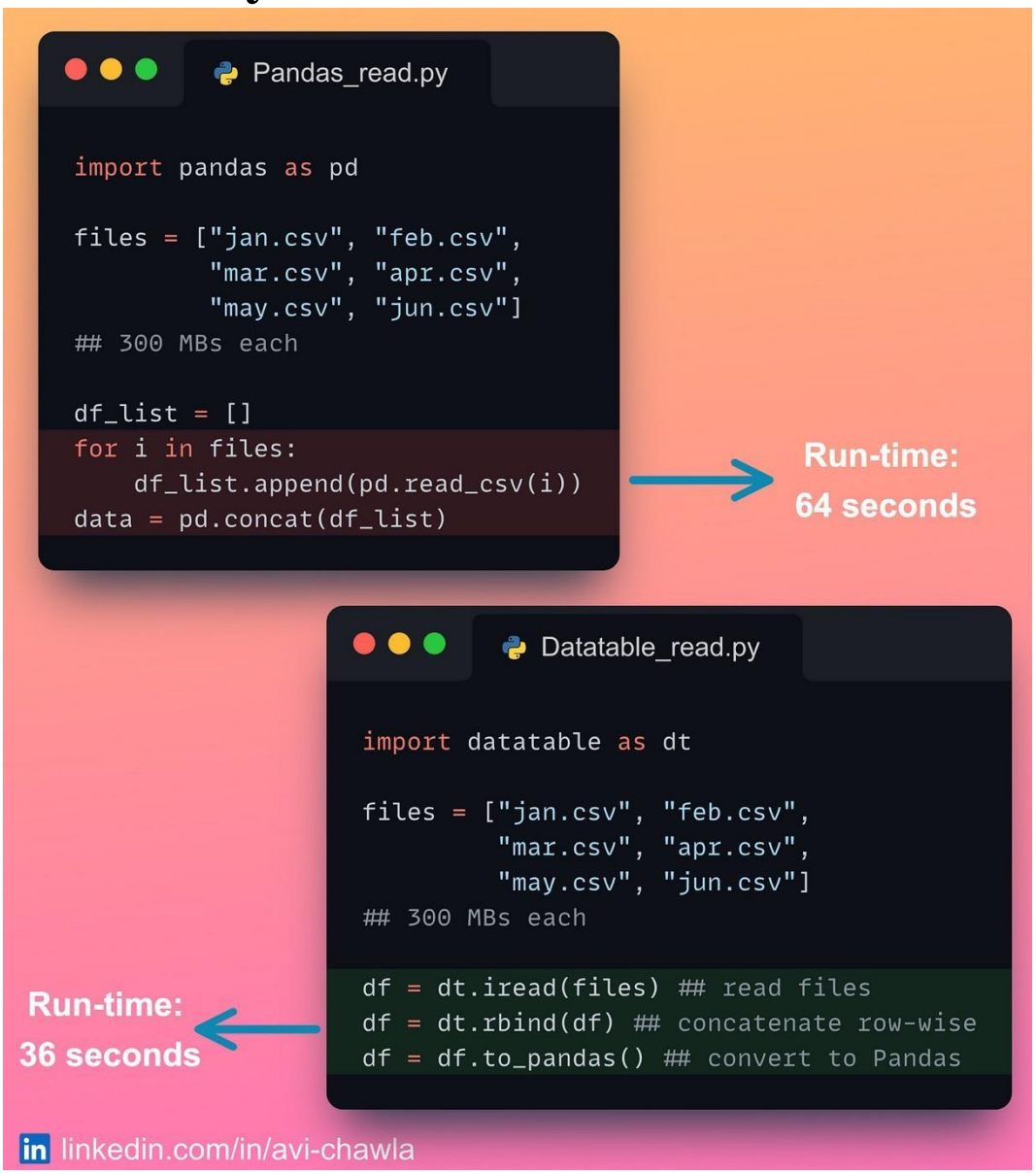
Restart the Kernel Without Losing Variables



While working in a Jupyter Notebook, you may want to restart the kernel due to several reasons. But before restarting, one often tends to dump data objects to disk to avoid recomputing them in the subsequent run.

The "store" magic command serves as an ideal solution to this. Here, you can obtain a previously computed value even after restarting your kernel. What's more, you never need to go through the hassle of dumping the object to disk.

How to Read Multiple CSV Files Efficiently



In many situations, the data is often split into multiple CSV files and transferred to the DS/ML team for use.

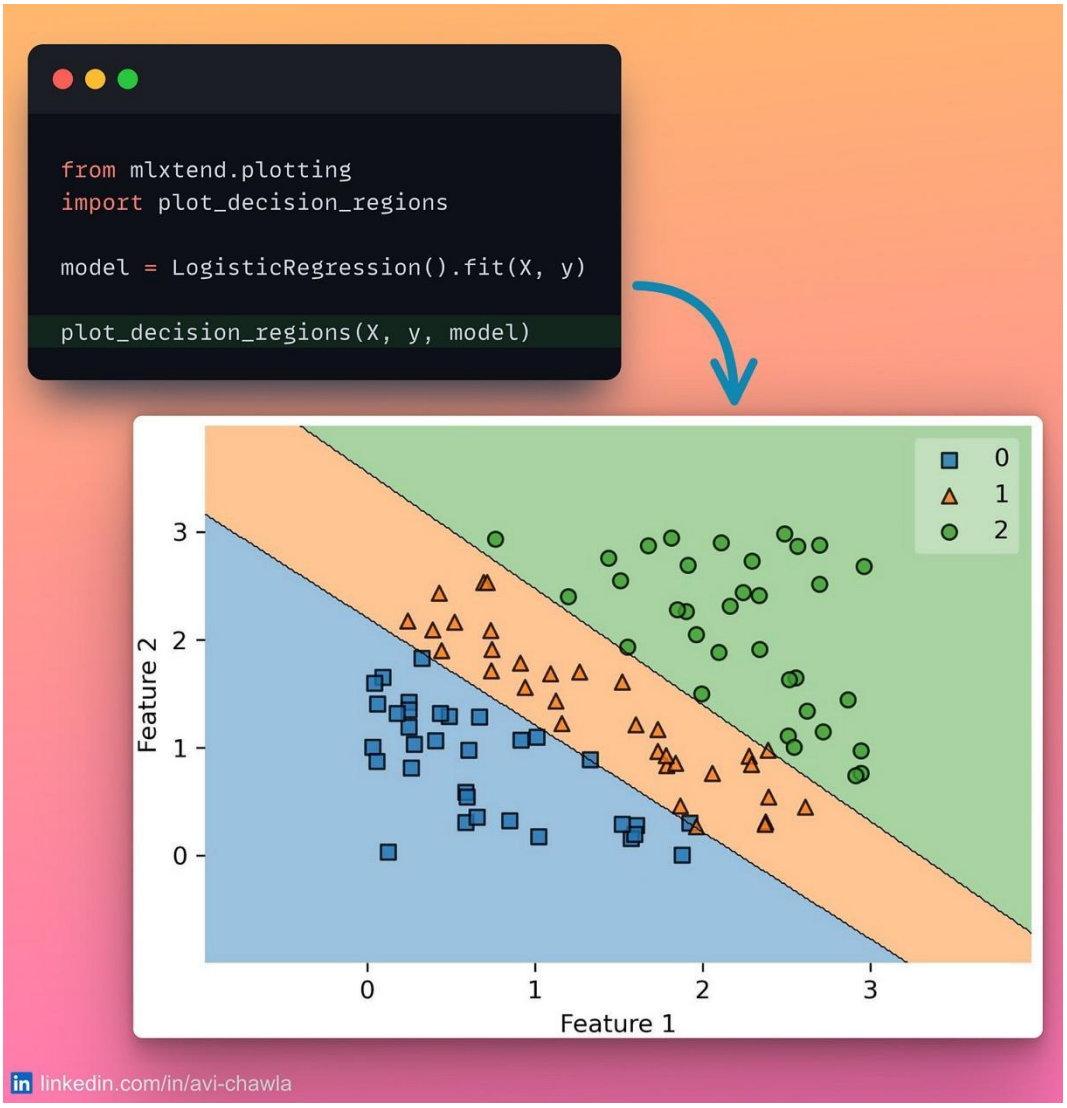
As Pandas does not support parallelization, one has to iterate over the list of files and read them one by one for further processing.

"Datatable" can provide a quick fix for this. Instead of reading them iteratively with Pandas, you can use Datatable to read a bunch of files. Being parallelized, it provides a significant performance boost as compared to Pandas.

The performance gain is not just limited to I/O but is observed in many other tabular operations as well.

Read more here: DataTable Docs.

Elegantly Plot the Decision Boundary of a Classifier

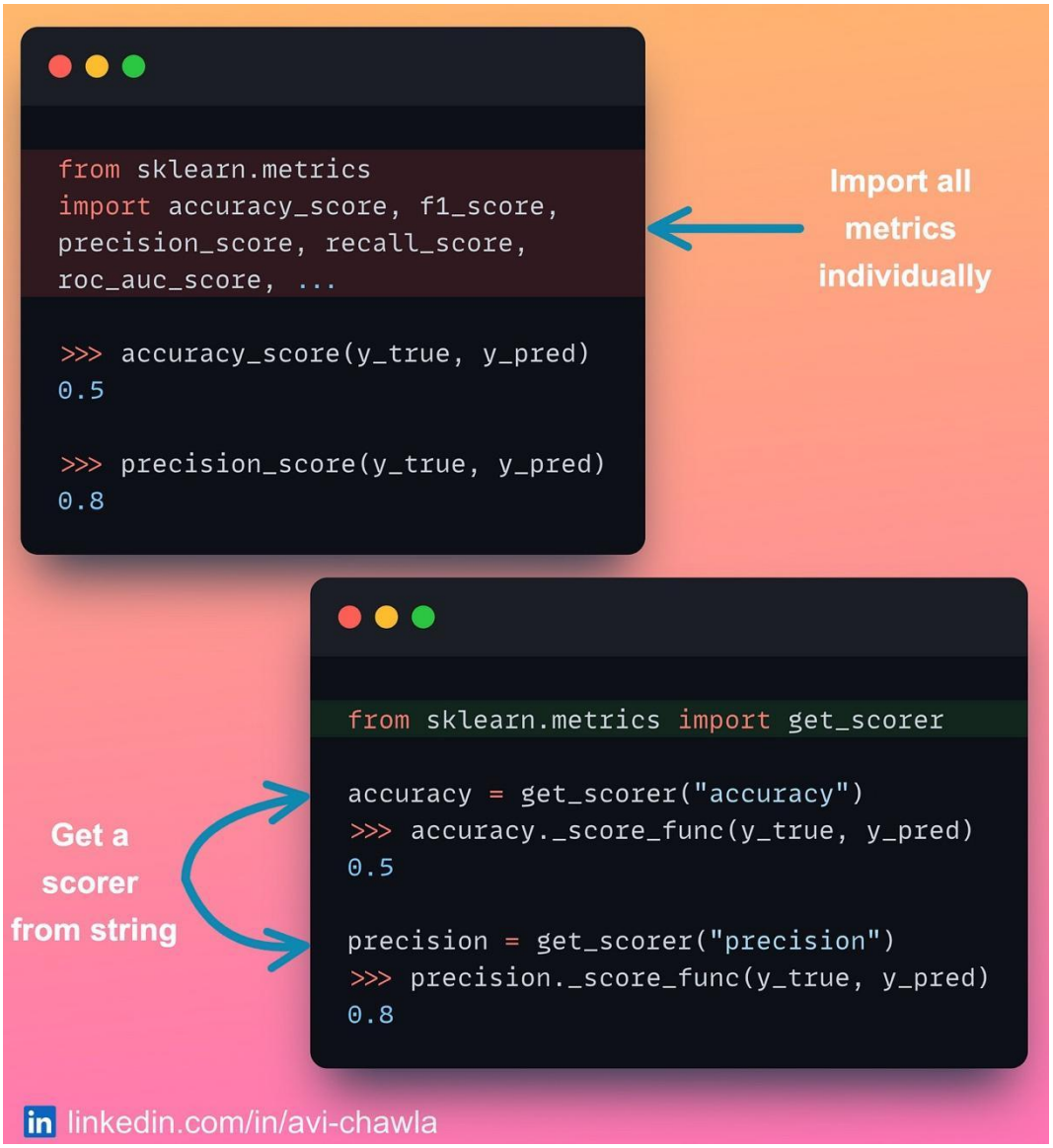


Plotting the decision boundary of a classifier can reveal many crucial insights about its performance.

Here, region-shaded plots are often considered a suitable choice for visualization purposes. But, explicitly creating one can be extremely time-consuming and complicated.

Mlxtend condenses that to a simple one-liner in Python. Here, you can plot the decision boundary of a classifier with ease, by just providing it the model and the data.

An Elegant Way to Import Metrics From Sklearn

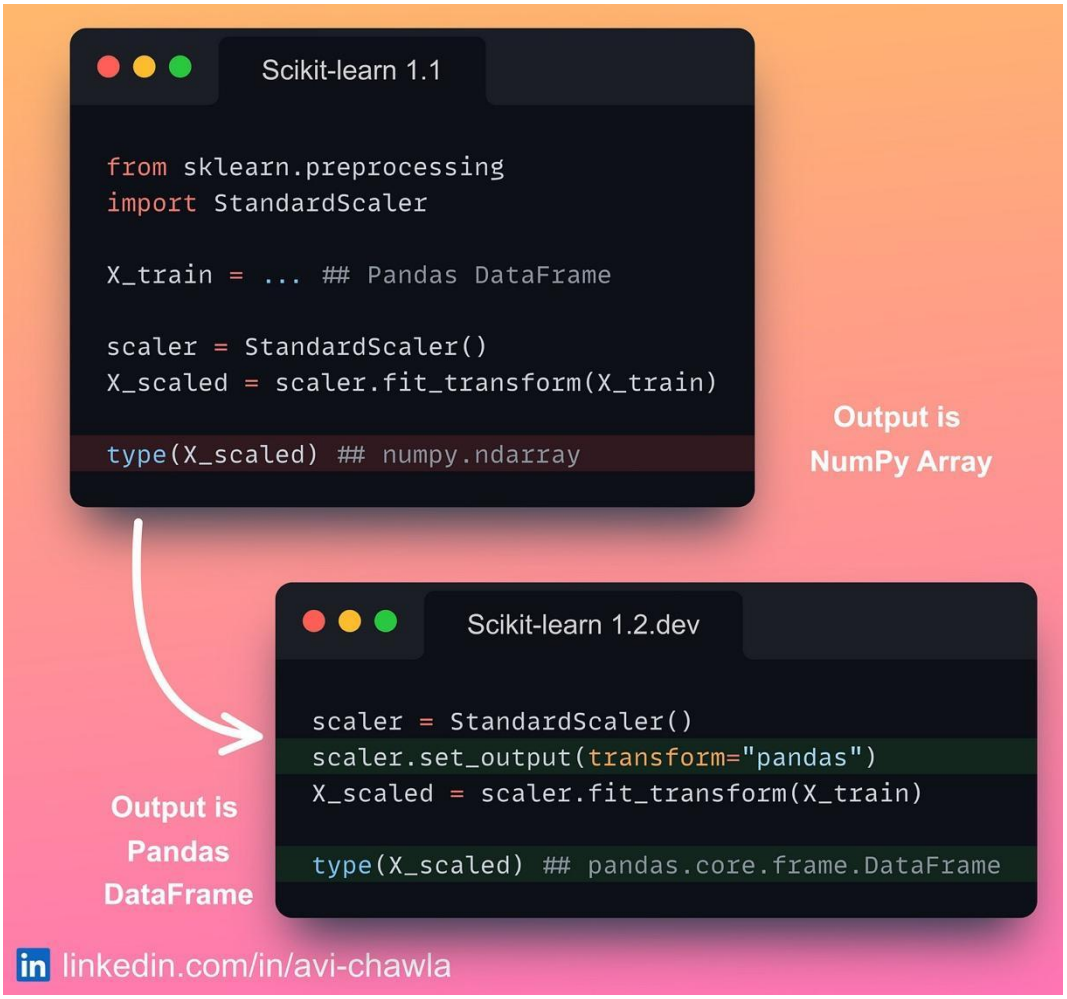


While using **scikit-learn**, one often imports multiple metrics to evaluate a model. Although there is nothing wrong with this practice, it makes the code inelegant and cluttered - with the initial few lines of the file overloaded with imports.

Instead of importing the metrics individually, you can use the **get_scorer**() method. Here, you can pass the metric's name as a string, and it returns a scorer object for you.

Read more here: Scikit-learn page.

Configure Sklearn To Output Pandas DataFrame



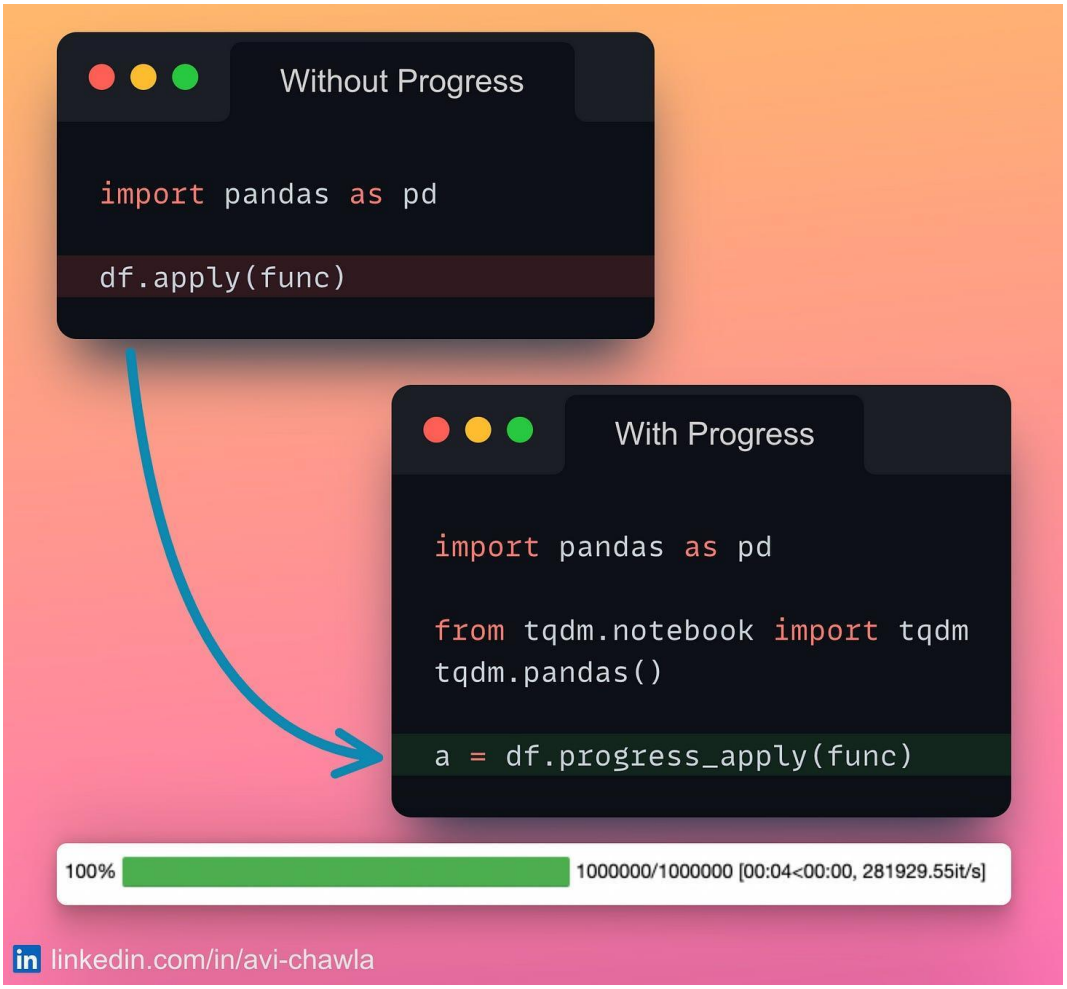
Recently, Scikit-learn announced the release of one of the most awaited improvements. In a gist, sklearn can now be configured to output Pandas DataFrames instead of NumPy arrays.

Until now, Sklearn's transformers were configured to accept a Pandas DataFrame as input. But they always returned a NumPy array as an output. As a result, the output had to be manually projected back to a Pandas DataFrame.

Now, the **set_output** API will let transformers output a Pandas DataFrame instead.

This will make running pipelines on DataFrames smoother. Moreover, it will provide better ways to track feature names.

Display Progress Bar With Apply() in Pandas



While applying a method to a DataFrame using **apply**(), we don't get to see the progress and an estimated remaining time.

To resolve this, you can instead use **progress_apply**() from **tqdm** to display a progress bar while applying a method.

Read more here: GitHub.

Modify a Function During Run-time

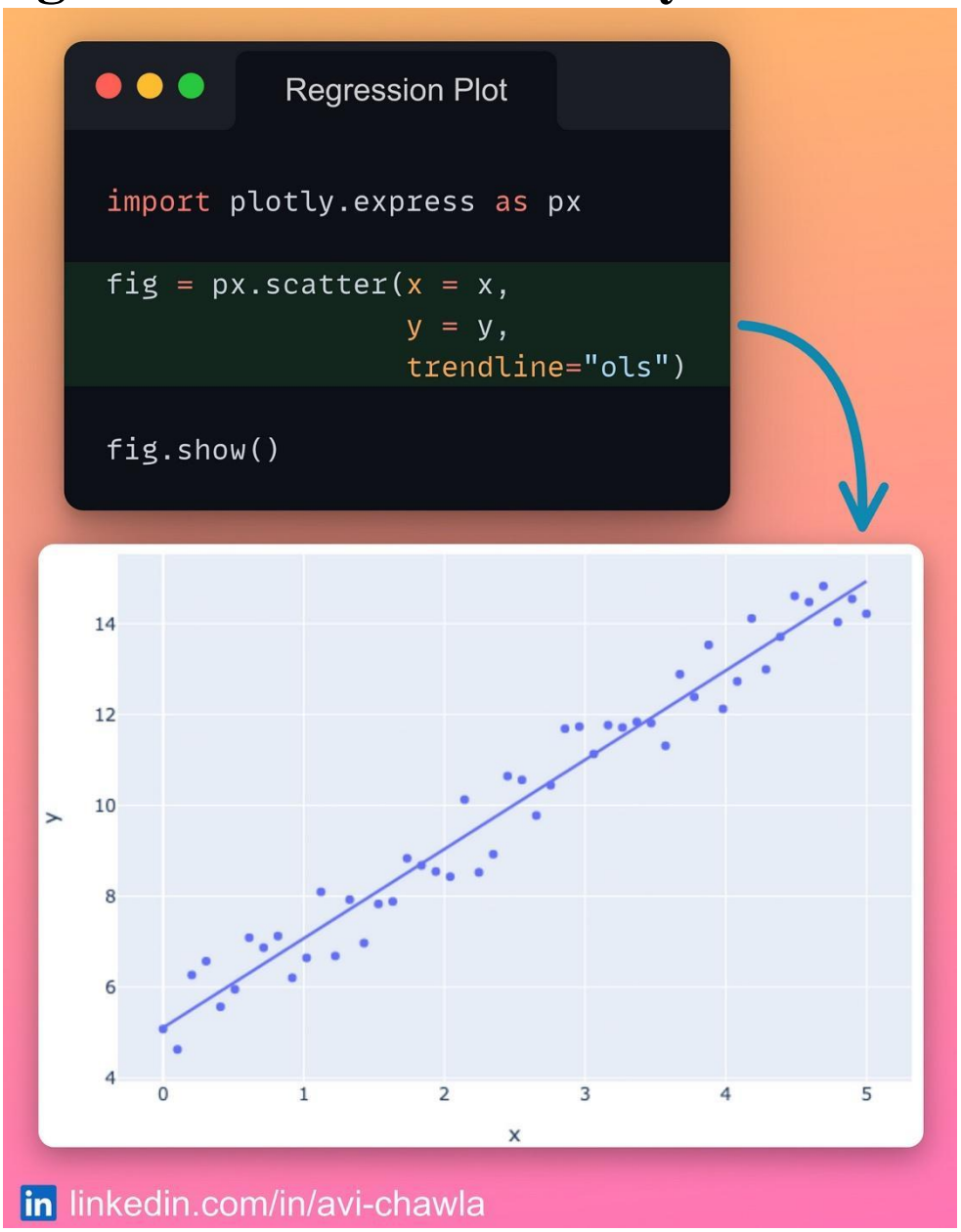
Have you ever been in a situation where you wished to add more details to an already running code?

This is typically observed in ML where one often forgets to print all the essential training details/metrics. Executing the entire code again, especially when it has been up for some time is not an ideal approach here.

If you want to modify a function during execution, decorate it with the reloading decorator (**@reloading**). As a result, Python will reload the function from the source before each execution.

Link to reloading: GitHub.

Regression Plot Made Easy with Plotly



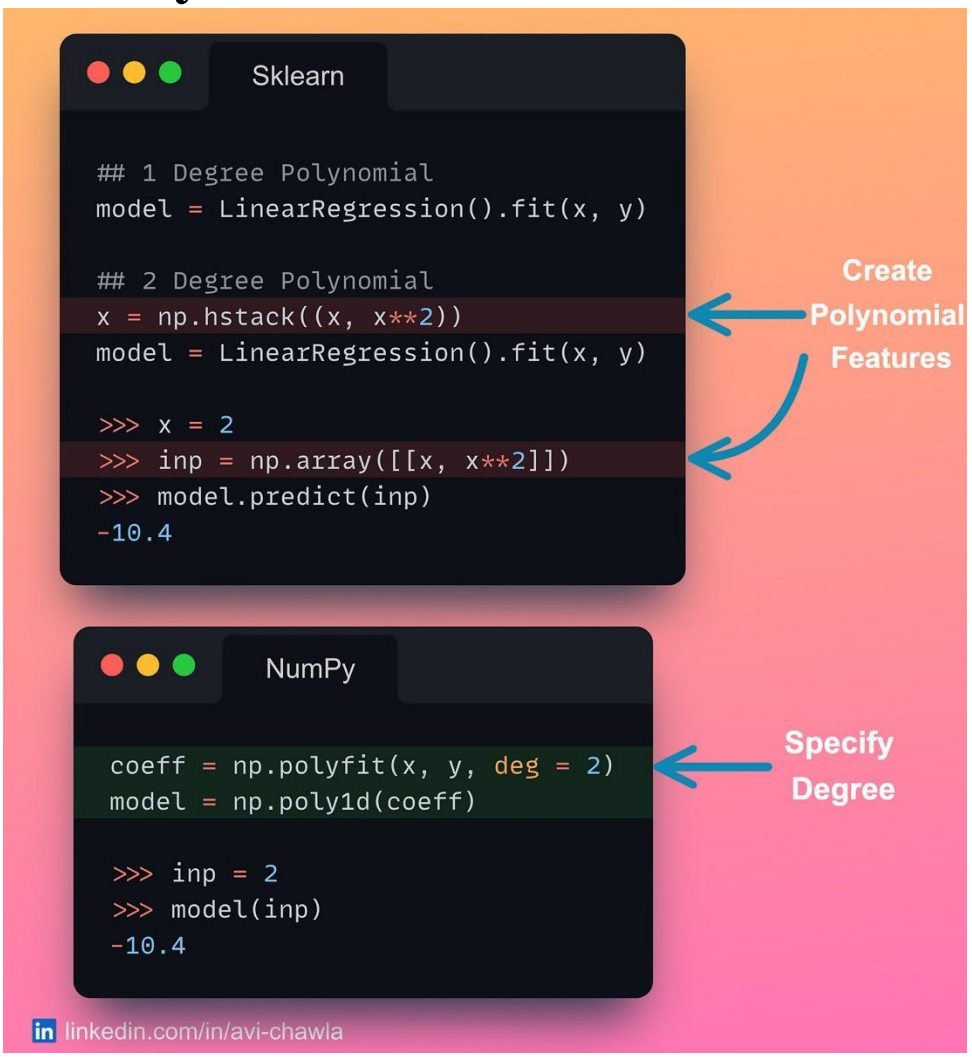
While creating scatter plots, one is often interested in displaying a simple linear regression fit on the data points.

Here, training a model and manually embedding it in the plot can be a tedious job to do.

Instead, with Plotly, you can add a regression line to a plot, without explicitly training a model.

Read more here.

Polynomial Linear Regression with NumPy



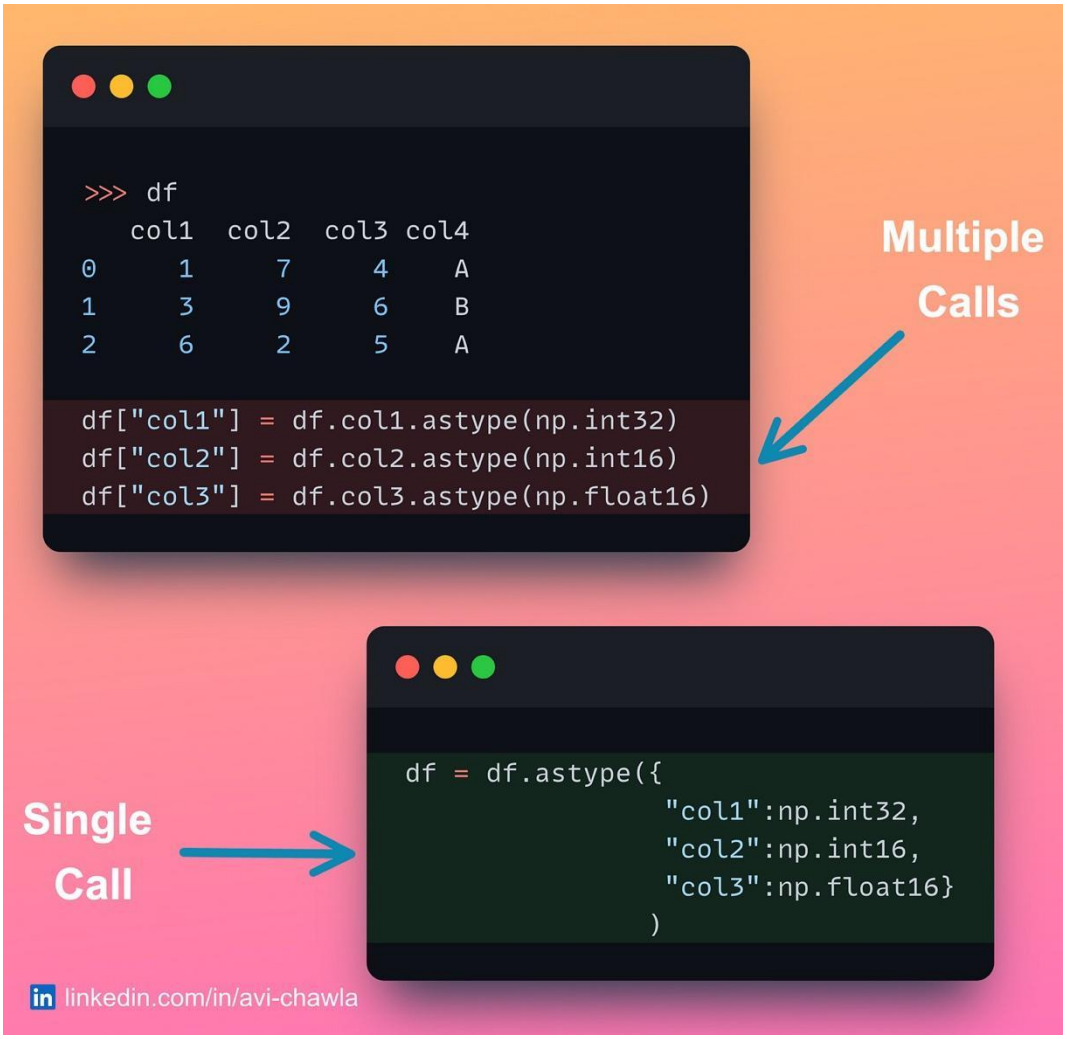
Polynomial linear regression using Sklearn is tedious as one has to explicitly code its features. This can get challenging when one has to iteratively build higher-degree polynomial models.

NumPy's **polyfit**() method is an excellent alternative to this. Here, you can specify the degree of the polynomial as a parameter. As a result, it automatically creates the corresponding polynomial features.

The downside is that you cannot add custom features such as trigonometric/logarithmic. In other words, you are restricted to only polynomial features. But if that is not your requirement, NumPy's **polyfit**() method can be a better approach.

Read more: https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html.

Alter the Datatype of Multiple Columns at Once



A common approach to alter the datatype of multiple columns is to invoke the **astype**() method individually for each column.

Although the approach works as expected, it requires multiple function calls and more code. This can be particularly challenging when you want to modify the datatype of many columns.

As a better approach, you can condense all the conversions into a single function call. This is achieved by passing a dictionary of column-to-datatype mapping, as shown below.

Datatype For Handling Missing Valued Columns in Pandas

If your data has NaN-valued columns, Pandas provides a datatype specifically for representing them - called the Sparse datatype.

This is especially handy when you are working with large data-driven projects with many missing values.

The snippet compares the memory usage of float and sparse datatype in Pandas.

Parallelize Pandas with Pandarallel

Pandas' operations do not support parallelization. As a result, it adheres to a single-core computation, even when other cores are available. This makes it inefficient and challenging, especially on large datasets.

"Pandarallel" allows you to parallelize its operations to multiple CPU cores - by changing just one line of code. Supported methods include apply(), applymap(), groupby(), map() and rolling().

Read more: GitHub.

Why you should not dump DataFrames to a CSV

The CSV file format is widely used to save Pandas DataFrames. But are you aware of its limitations? To name a few,

1. The CSV does not store the datatype information. Thus, if you modify the datatype of column(s), save it to a CSV, and load again, Pandas will not return the same datatypes.

2. Saving the DataFrame to a CSV file format isn't as optimized as other

supported formats by Pandas. These include Parquet, Pickle, etc.

Of course, if you need to view your data outside Python (Excel, for instance), you are bound to use a CSV. But if not, prefer other file formats.

Further reading: Why I Stopped Dumping DataFrames to a CSV and Why You Should Too.

Save Memory with Python Generators

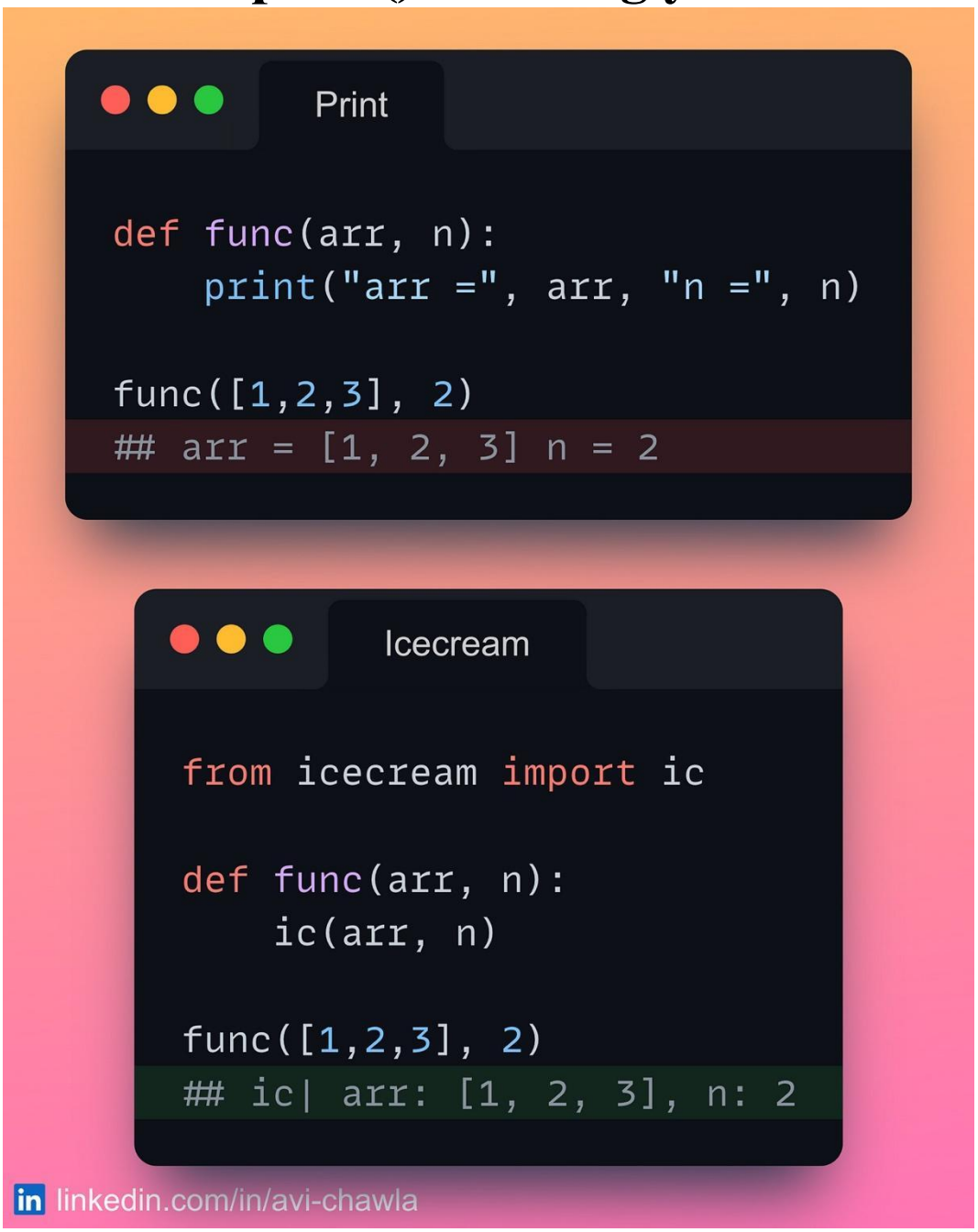
If you use large static iterables in Python, a list may not be an optimal choice, especially in memory-constrained applications.

A list stores the entire collection in memory. However, a generator computes and loads a single element at a time ONLY when it is required. This saves both memory and object creation time.

Of course, there are some limitations of generators too. For instance, you cannot use common list operations such as append(), slicing, etc.

Moreover, every time you want to reuse an element, it must be regenerated (see Generator.py: line 12).

Don't use print() to debug your code.



Debugging with print statements is a messy and inelegant approach. It is confusing to map the output to its corresponding debug statement. Moreover, it requires extra manual formatting to comprehend the output.

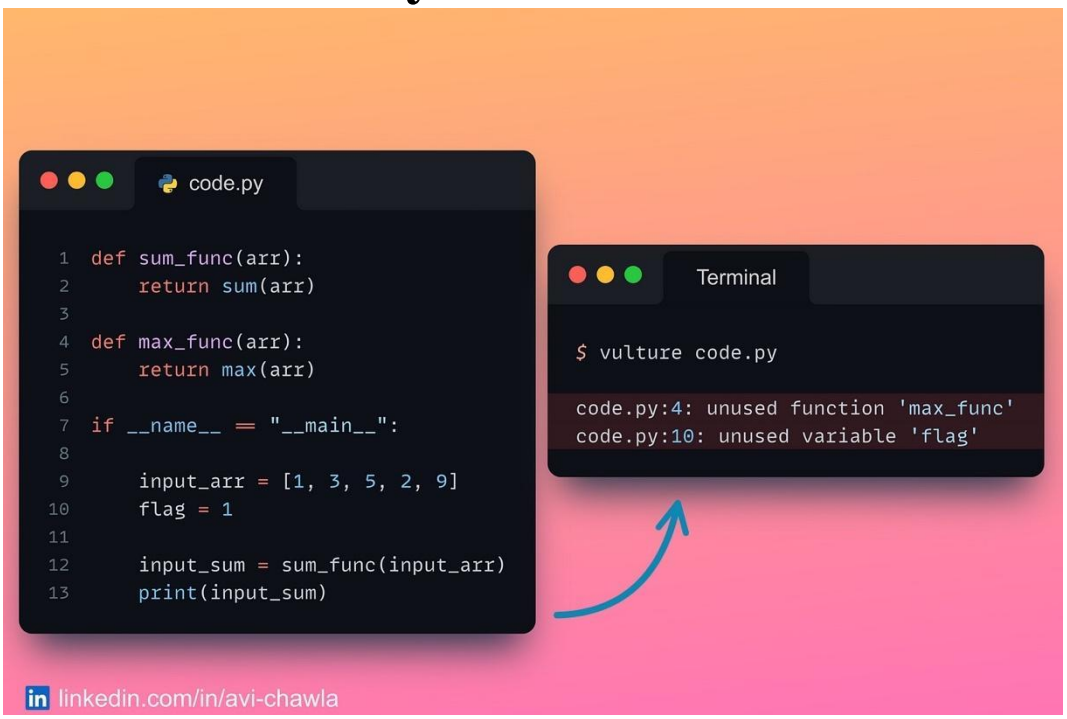
The "**icecream**" library in Python is an excellent alternative to this. It makes debugging effortless and readable, with minimal code. Features include printing expressions, variable names, function names, line numbers, filenames, and many

more.

P.S. The snippet only gives a brief demonstration. However, the actual functionalities are much more powerful and elegant as compared to debugging with print().

More about icecream here: https://github.com/gruns/icecream.

Find Unused Python Code With Ease



As the size of your codebase increases, so can the number of instances of unused code. This inhibits its readability and conciseness.

With the "vulture" module in Python, you can locate dead (unused) code in your pipeline, as shown in the snippet.

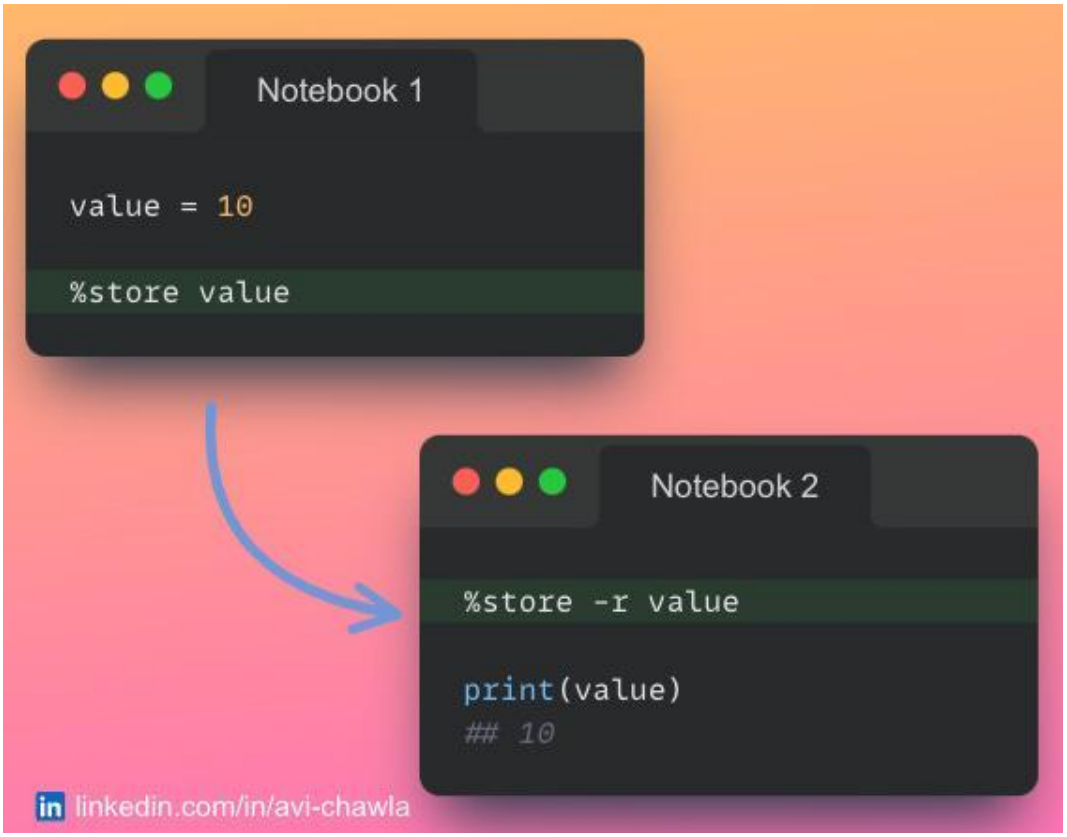
Define the Correct DataType for Categorical Columns

If your data has categorical columns, you should not represent them as int/string data type.

Rather, Pandas provides an optimized data type specifically for categorical columns. This is especially handy when you are working with large data-driven projects.

The snippet compares the memory usage of string and categorical data types in Pandas.

Transfer Variables Between Jupyter Notebooks

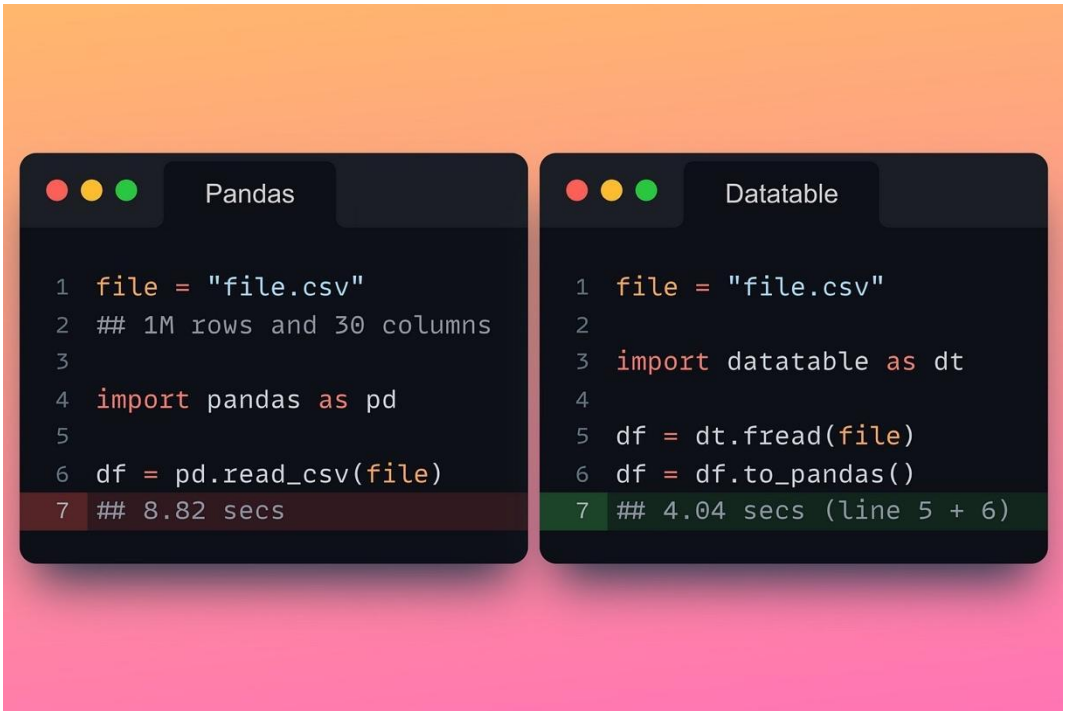


While working with multiple jupyter notebooks, you may need to share objects between them.

With the "store" magic command, you can transfer variables across notebooks without storing them on disk.

P.S. You can also restart the kernel and retrieve an old variable with "store".

Why You Should Not Read CSVs with Pandas

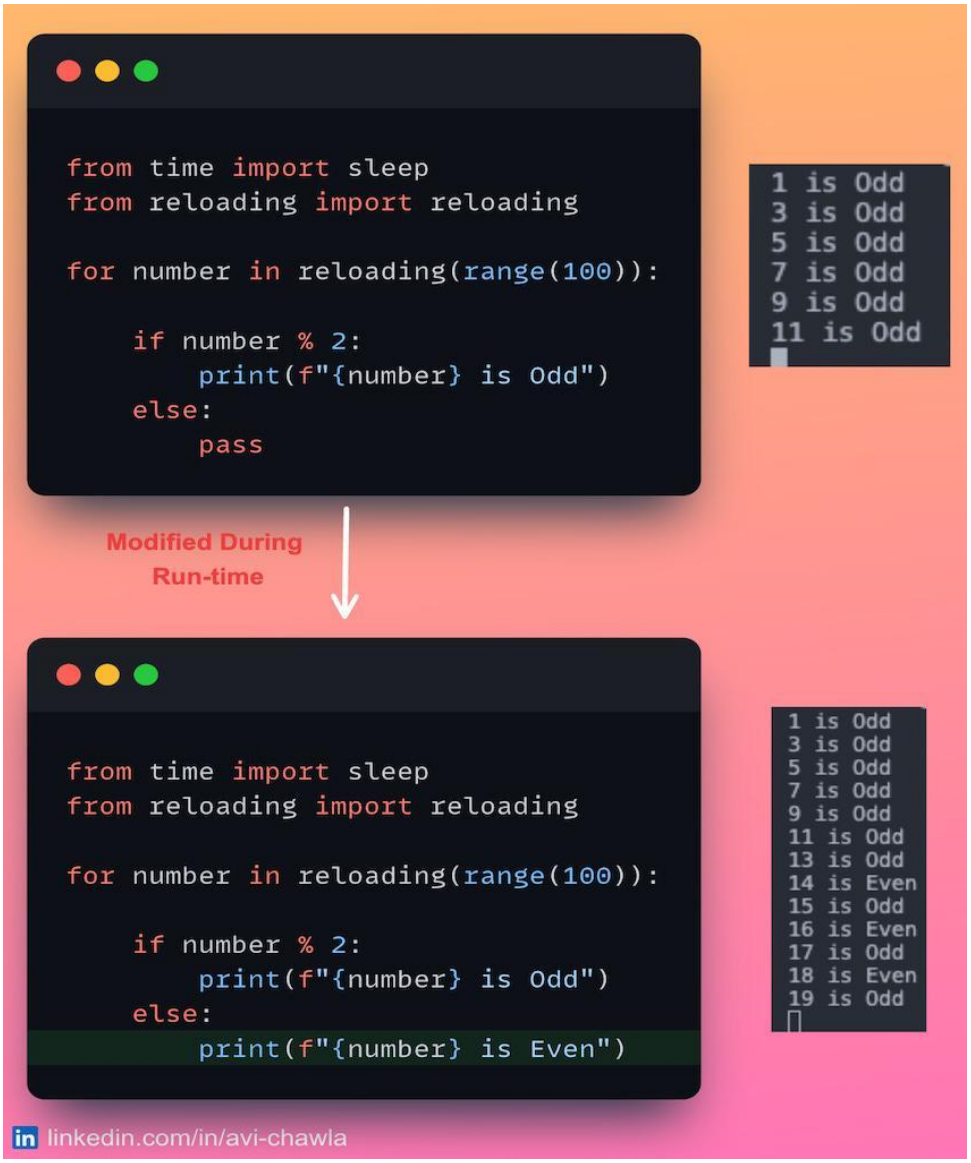


Pandas adheres to a single-core computation, which makes its operations extremely inefficient, especially on large datasets.

The "datatable" library in Python is an excellent alternative with a Pandas-like API. Its multi-threaded data processing support makes it faster than Pandas.

The snippet demonstrates the run-time comparison of creating a "Pandas DataFrame" from a CSV using Pandas and Datatable.

Modify Python Code During Run-Time

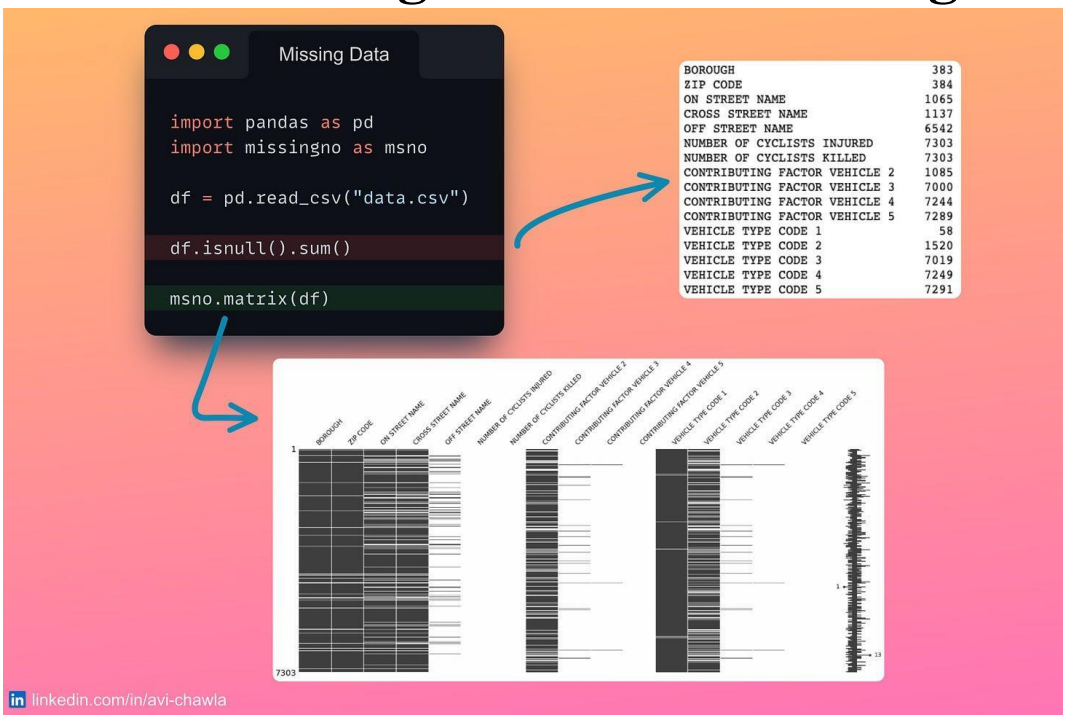


Have you ever been in a situation where you wished to add more details to an already running code (printing more details in a for-loop, for instance)?

Executing the entire code again, especially when it has been up for some time, is not the ideal approach here.

With the "reloading" library in Python, you can add more details to a running code without losing any existing progress.

Handle Missing Data With Missingno



If you want to analyze missing values in your dataset, Pandas may not be an apt choice.

Pandas' methods hide many important details about missing values. These include their location, periodicity, the correlation across columns, etc.

The "missingno" library in Python is an excellent resource for exploring missing data. It generates informative visualizations for improved data analysis.

The snippet demonstrates missing data analysis using Pandas and Missingno.