✨ Member-only story

# Top 10 Matrix Operations in Numpy with Examples

## Perform Linear Algebra with Python

Rukshan Pramoditha · Follow
Published in Towards Data Science
8 min read · Mar 24, 2021

Listen Share

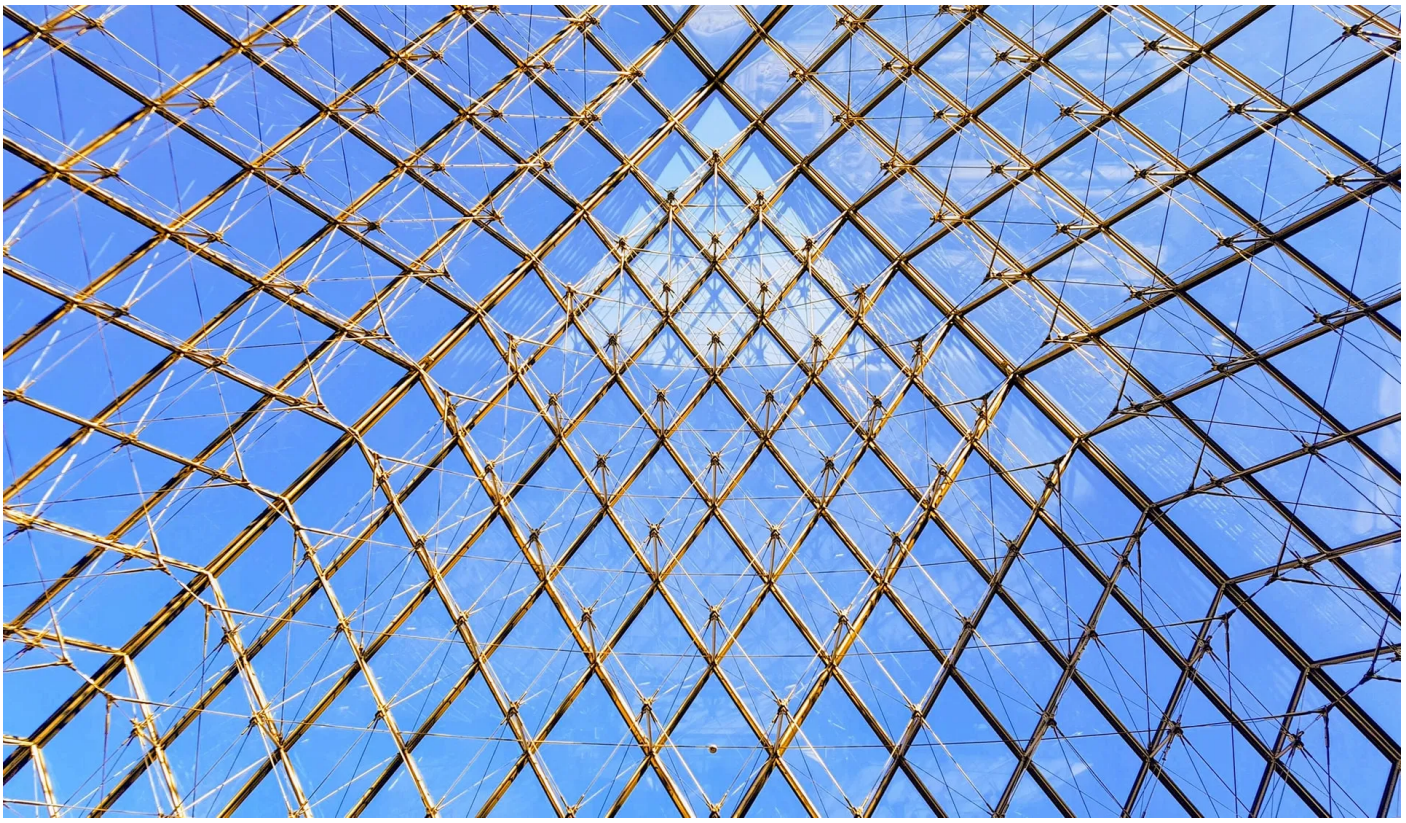


Photo by Isaiah Bekkers on Unsplash

About 30–40% of the mathematical knowledge required for Data Science and

Machine Learning comes from linear algebra. Matrix operations play a significant role in linear algebra. Today, we discuss 10 of such matrix operations with the help of the powerful numpy library. Numpy is generally used to perform numerical calculations in Python. It also has special classes and sub-packages for matrix operations. The use of vectorization allows numpy to perform matrix operations more efficiently by avoiding many for loops.

I will include the meaning, background description and code examples for each matrix operation discussing in this article. The "Key Takeaways" section at the end of this article will provide you with some more specific facts and a brief summary of matrix operations. So, make sure to read that section as well.

I will discuss each matrix operation in the following order. Here is the list of the top 10 matrix operations I have chosen for you carefully.

1. **Inner product**
2. **Dot product**
3. **Transpose**
4. **Trace**
5. **Rank**
6. **Determinant**
7. **True inverse**
8. **Pseudo inverse**
9. **Flatten**
10. **Eigenvalues and eigenvectors**

#### Prerequisites

To get the full advantage of this article, you should know the numpy basics and array creation methods. If you don't have that knowledge, read the following article written by me.

- NumPy for Data Science: Part 1 (NumPy Basics and Array Creation)

Let's get started with the first one, the ***inner product***.

### Inner product

The ***inner product*** takes two vectors of equal size and returns a single number (scalar). This is calculated by multiplying the corresponding elements in each vector and adding up all of those products. In numpy, vectors are defined as one-dimensional numpy arrays.

To get the inner product, we can use either **np.inner()** or **np.dot()**. Both give the same results. The inputs for these functions are two vectors and they should be the same size.

```
1  import numpy as np
2
3  # Vectors as 1D numpy arrays
4  a = np.array([1, 2, 3])
5  b = np.array([4, 5, 6])
6
7  print("a= ", a)
8  print("b= ", b)
9  print("\ninner:", np.inner(a, b))
10 print("dot:", np.dot(a, b))
```

**matrix_operations_in_numpy_1.py** hosted with ❤ by **GitHub** **view raw**

Wait till loading the Python code!

```
a=  [1 2 3]
b=  [4 5 6]

inner: 32
dot: 32
```

The inner product of two vectors (Image by author)

### Dot product

The ***dot product*** is defined for matrices. It is the sum of the products of the corresponding elements in the two matrices. To get the dot product, the number of columns in the first matrix should be equal to the number of rows in the second matrix.

There are two ways to create matrices in numpy. The most common one is to use the numpy ***ndarray class***. Here we create two-dimensional numpy arrays (ndarray

objects). The other one is to use the numpy ***matrix class***. Here we create matrix objects.

The dot product of both ndarray and matrix objects can be obtained using **np.dot()**.

```
import numpy as np

# Matrices as ndarray objects
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6, 7], [8, 9, 10]])
print("a", type(a))
print(a)
print("\nb", type(b))
print(b)

# Matrices as matrix objects
c = np.matrix([[1, 2], [3, 4]])
d = np.matrix([[5, 6, 7], [8, 9, 10]])
print("\nc", type(c))
print(c)
print("\nd", type(d))
print(d)
print("\ndot product of two ndarray objects")
print(np.dot(a, b))
print("\ndot product of two matrix objects")
print(np.dot(c, d))
```

matrix_operations_in_numpy_2.py hosted with ♥ by GitHub view raw

Wait till loading the Python code!

```
a <class 'numpy.ndarray'>
[[1 2]
 [3 4]]

b <class 'numpy.ndarray'>
[[ 5  6  7]
 [ 8  9 10]]

c <class 'numpy.matrix'>
[[1 2]
 [3 4]]

d <class 'numpy.matrix'>
[[ 5  6  7]
 [ 8  9 10]]

dot product of two ndarray objects
[[21 24 27]
 [47 54 61]]

dot product of two matrix objects
[[21 24 27]
 [47 54 61]]
```

The dot product of two matrices (Image by author)

When multiplying two ndarray objects using the * operator, the result is the *element-by-element multiplication*. On the other hand, when multiplying two matrix objects using the * operator, the result is the *dot (matrix) product* which is equivalent to the ***np.dot()*** as previous.

```python
import numpy as np

# Matrices as ndarray objects
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [8, 9]])
print("a", type(a))
print(a)
print("\nb", type(b))
print(b)

# Matrices as matrix objects
c = np.matrix([[1, 2], [3, 4]])
d = np.matrix([[5, 6], [8, 9]])
print("\nc", type(c))
print(c)
print("\nd", type(d))
print(d)
print("\n* operation on two ndarray objects (Elementwise)")
print(a * b)
print("\n* operation on two matrix objects (same as np.dot())")
print(c * d)
```

Wait till loading the Python code!

```
a <class 'numpy.ndarray'>
[[1 2]
 [3 4]]

b <class 'numpy.ndarray'>
[[5 6]
 [8 9]]

c <class 'numpy.matrix'>
[[1 2]
 [3 4]]

d <class 'numpy.matrix'>
[[5 6]
 [8 9]]

* operation on two ndarray objects (Elementwise)
[[ 5 12]
 [24 36]]

* operation on two matrix objects (same as np.dot())
[[21 24]
 [47 54]]
```

Different behaviour of * operator on matrix and ndarray objects (Image by author)

### Transpose

The *transpose* of a matrix is found by switching its rows with its columns. We can use **np.transpose()** function or NumPy **ndarray.transpose()** method or **ndarray.T** (a special method which does not require parentheses) to get the transpose. All give the same output.

```
import numpy as np

a = np.array([[1, 2], [3, 4], [5, 6]])
print("a = ")
print(a)

print("\nWith np.transpose(a) function")
print(np.transpose(a))

print("\nWith ndarray.transpose() method")
print(a.transpose())

print("\nWith ndarray.T short form")
print(a.T)
```

**matrix_operations_in_numpy_4.py** hosted with ❤ by **GitHub** **view raw**

Wait till loading the Python code!

```
a =
[[1 2]
 [3 4]
 [5 6]]

With np.transpose(a) function
[[1 3 5]
 [2 4 6]]

With ndarray.transpose() method
[[1 3 5]
 [2 4 6]]

With ndarray.T short form
[[1 3 5]
 [2 4 6]]
```

Getting transpose of a matrix (Image by author)

The transpose can also be applied to a vector. However, technically, a one-dimensional numpy array cannot be transposed.

```
import numpy as np

a = np.array([1, 2, 3])
print("a = ")
print(a)
print("\na.T = ")
print(a.T)
```

```
a =
[1 2 3]

a.T =
[1 2 3]
```

Image by author

If you really want to transpose a vector, it should be defined as a two-dimensional numpy array with double square brackets.

```
import numpy as np

a = np.array([[1, 2, 3]])
print("a = ")
print(a)
print("\na.T = ")
print(a.T)
```

```
a =
[[1 2 3]]

a.T =
[[1]
 [2]
 [3]]
```

Image by author

### Trace

The *trace* is the sum of diagonal elements in a square matrix. There are two methods to calculate the trace. We can simply use the **trace()** method of an ndarray object or get the diagonal elements first and then get the sum.

```
import numpy as np

a = np.array([[2, 2, 1],
              [1, 3, 1],
              [1, 2, 2]])
print("a = ")
print(a)
print("\nTrace:", a.trace())
print("Trace:", sum(a.diagonal()))
```

```
a =
[[2 2 1]
 [1 3 1]
 [1 2 2]]

Trace: 7
Trace: 7
```

Image by author

**Rank**

The ***rank*** of a matrix is the dimensions of the vector space spanned (generated) by its columns or rows. In other words, it can be defined as the maximum number of linearly independent column vectors or row vectors.

The rank of a matrix can be found using the **matrix_rank()** function which comes from the numpy ***linalg*** package.

```
import numpy as np

a = np.arange(1, 10)
a.shape = (3, 3)
print("a = ")
print(a)
rank = np.linalg.matrix_rank(a)
print("\nRank:", rank)
```

```
a =
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Rank: 2
```

Image by author

### Determinant

The ***determinant*** of a square matrix can be calculated **det()** function which also comes from the numpy ***linalg*** package. If the determinant is 0, that matrix is not invertible. It is known as a singular matrix in algebra terms.

```
import numpy as np

a = np.array([[2, 2, 1],
              [1, 3, 1],
              [1, 2, 2]])
print("a = ")
print(a)
det = np.linalg.det(a)
print("\nDeterminant:", np.round(det))
```

```
a =
[[2 2 1]
 [1 3 1]
 [1 2 2]]

Determinant: 5.0
```

Image by author

### True inverse

The ***true inverse*** of a square matrix can be found using the **inv()** function of the numpy ***linalg*** package. If the determinant of a square matrix is not 0, it has a true inverse.

```
import numpy as np

a = np.array([[2, 2, 1],
              [1, 3, 1],
              [1, 2, 2]])
print("a = ")
print(a)
det = np.linalg.det(a)
print("\nDeterminant:", np.round(det))
inv = np.linalg.inv(a)
print("\nInverse of a = ")
print(inv)
```

```
a =
[[2 2 1]
 [1 3 1]
 [1 2 2]]

Determinant: 5.0

Inverse of a =
[[ 0.8 -0.4 -0.2]
 [-0.2  0.6 -0.2]
 [-0.2 -0.4  0.8]]
```

Image by author

If you try to compute the true inverse of a singular matrix (a square matrix whose determinant is 0), you will get an error.

```
import numpy as np

a = np.array([[2, 8],
              [1, 4]])
print("a = ")
print(a)
det = np.linalg.det(a)
print("\nDeterminant:", np.round(det))
inv = np.linalg.inv(a)
print("\nInverse of a = ")
print(inv)
```

```
a =
[[2 8]
 [1 4]]

Determinant: 0.0

---------------------------------------------
LinAlgError                               Trac
<ipython-input-110-df20cbc6f696> in <module>
```

Image by author

**Pseudo inverse**

The *pseudo (not genuine) inverse* can be calculated even for a singular matrix (a square matrix whose determinant is 0) using the **pinv()** function of the numpy *linalg* package.

```
import numpy as np

a = np.array([[2, 8],
              [1, 4]])
print("a = ")
print(a)
det = np.linalg.det(a)
print("\nDeterminant:", np.round(det))
pinv = np.linalg.pinv(a)
print("\nPseudo Inverse of a = ")
print(pinv)
```

```
a =
[[2 8]
 [1 4]]

Determinant: 0.0

Pseudo Inverse of a =
[[0.02352941 0.01176471]
 [0.09411765 0.04705882]]
```

Image by author

There is no difference between true inverse and pseudo-inverse if a square matrix is non-singular (determinant is ***not*** 0).

#### Flatten

***Flatten*** is a simple method to transform a matrix into a one-dimensional numpy array. For this, we can use the **flatten()** method of an ndarray object.

```
import numpy as np

a = np.arange(1, 10)
a.shape = (3, 3)
print("a = ")
print(a)
print("\nAfter flattening")
print("------------------")
print(a.flatten())
```

```
a =
[[1 2 3]
 [4 5 6]
 [7 8 9]]

After flattening
------------------
[1 2 3 4 5 6 7 8 9]
```

Image by author

**Eigenvalues and eigenvectors**

Let **A** be an *n x n matrix*. A scalar **λ** is called an ***eigenvalue*** of **A** if there is a non-zero vector **x** satisfying the following equation.

$$Ax = \lambda x$$

Image by author

The vector **x** is called the ***eigenvector*** of **A** corresponding to **λ**.

In numpy, both eigenvalues and eigenvectors can be calculated simultaneously using the **eig()** function.

```
import numpy as np

a = np.array([[2, 2, 1],
              [1, 3, 1],
              [1, 2, 2]])
print("a = ")
print(a)
w, v = np.linalg.eig(a)
print("\nEigenvalues:")
print(w)
print("\nEigenvectors:")
print(v)
```

```
a =
[[2 2 1]
 [1 3 1]
 [1 2 2]]

Eigenvalues:
[1. 5. 1.]

Eigenvectors:
[[-0.90453403  0.57735027  0.04093652]
 [ 0.30151134  0.57735027 -0.46313831]
 [ 0.30151134  0.57735027  0.88500314]]
```

Numpy Matrix Linear Algebra Python Arrays

Image by author

The sum of eigenvalues (1+5+1=7) is equal to the trace (2+3+2=7) of the same matrix! The product of the eigenvalues (1x5x1=5) is equal to the determinant (5) of the same matrix!

Eigenvalues and eigenvectors are extremely useful in the Principal Component Analysis (PCA). In PCA, the eigenvectors of the correlation or covariance matrix r[illegible]t the principal components (the directions of maximum variance) and [illegible]ponding eigenvalues represent the amount of variation explained by eac[illegible]pal component. If you want to learn more about th[illegible] following articles written by me.



## Written by Rukshan Pramoditha



- Principal Component Analysis (PCA) with Scikit-learn
- Principal Component Analysis for Breast Cancer Data with R and Python



## Key Takeaways

Thanks to the numpy library, you can perform matrix operations easily with just one or 2 lines of code. Today, we have performed 10 matrix operations in numpy. Numpy has common functions as well as special functions dedicated to linear algebra, for example, the ***linalg*** package has some special functions dedicated to linear algebra.

In numpy, matrices and ndarrays are two different things. The best way to get familiar with them is by experimenting with the codes by yourself. It is always better to check the dimensions of matrices and ndarrays.

In Scikit-learn machine learning libraries, most of the matrix operations


This tutorial was designed and created by *Rukshan Pramoditha*, the Author of Data Science 365 Blog.

Rukshan Pramoditha in Towards Data Science

## Non-Negative Matrix Factorization (NMF) for Dimensionality Reduction in Image Data

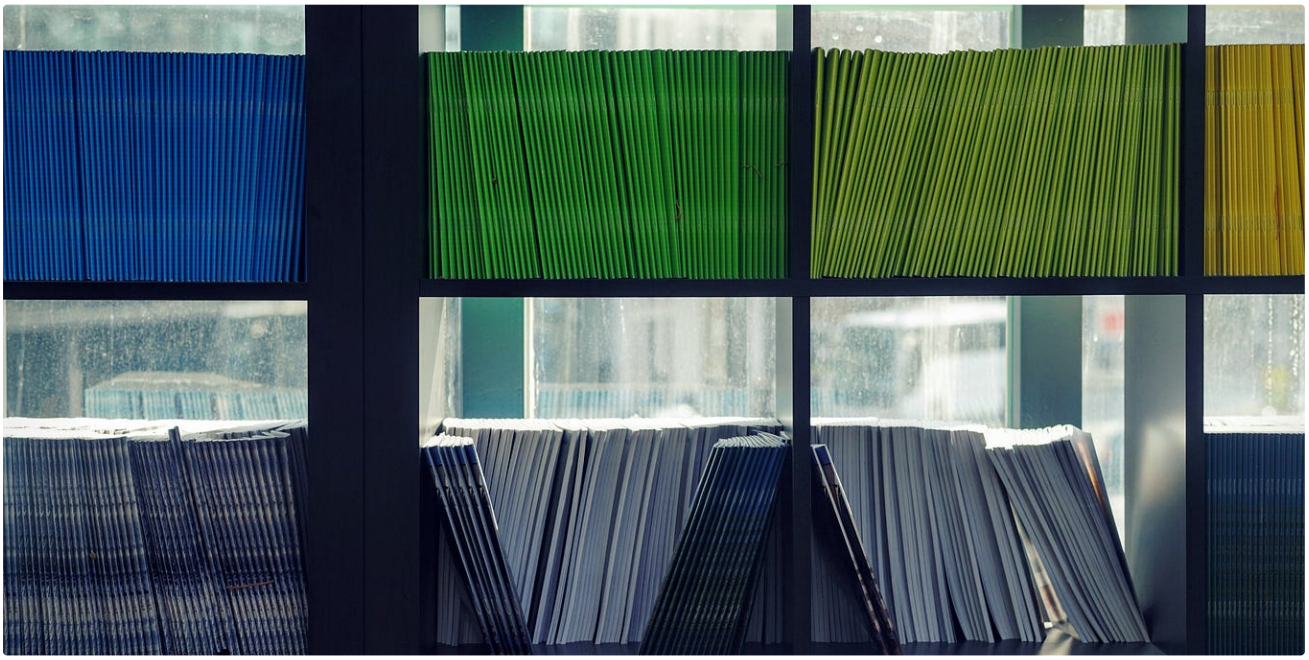
Read my stories in your inbox — Data Science Digest — https://rukshanpramoditha.medium.com

Discussing theory and implementation with Python and Scikit-learn

2021–03–24

✨ · 9 min read · May 6

193 1



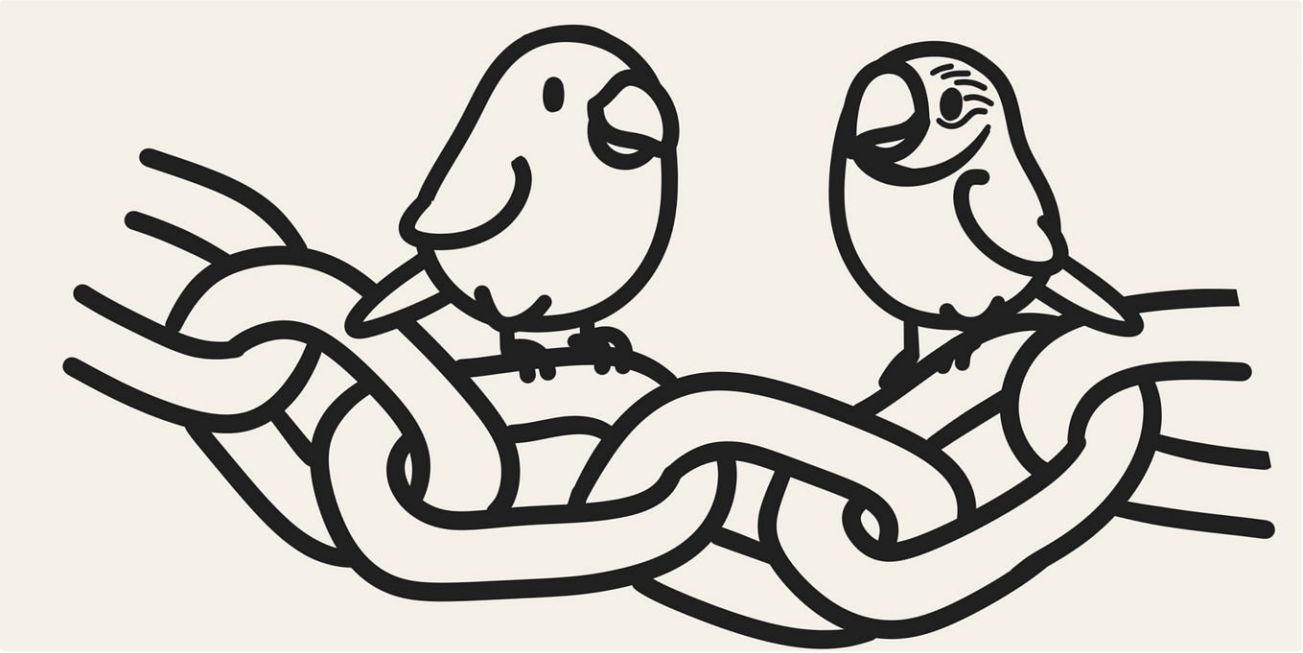
Jacob Marks, Ph.D. in Towards Data Science

## How I Turned My Company's Docs into a Searchable Database with OpenAI

And how you can do the same with your docs

15 min read · Apr 25

2.3K 30



Leonie Monigatti in Towards Data Science

## Getting Started with LangChain: A Beginner's Guide to Building LLM-Powered Applications

A LangChain tutorial to build anything with large language models in Python

· 12 min read · Apr 25

1.3K 13

Rukshan Pramoditha in Towards Data Science

## 11 Dimensionality reduction techniques you should know in 2021

Recommended from Medium

Reduce the size of your data while keeping as much of the variation as possible

✨ · 16 min read · Apr 13, 2021

Matt Chapman in Towards Data Science

Tomer Gabay in Towards Data Science

## 5 Python Tricks That Distinguish Senior Developers From Juniors

Illustrated through differences in approaches to Advent of Code puzzles

✨ · 6 min read · Jan 16

730 18

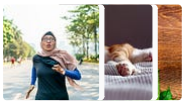
### Lists

**Staff Picks**

304 stories · 66 saves

**Stories to Help You Level-Up at Work**

19 stories · 24 saves

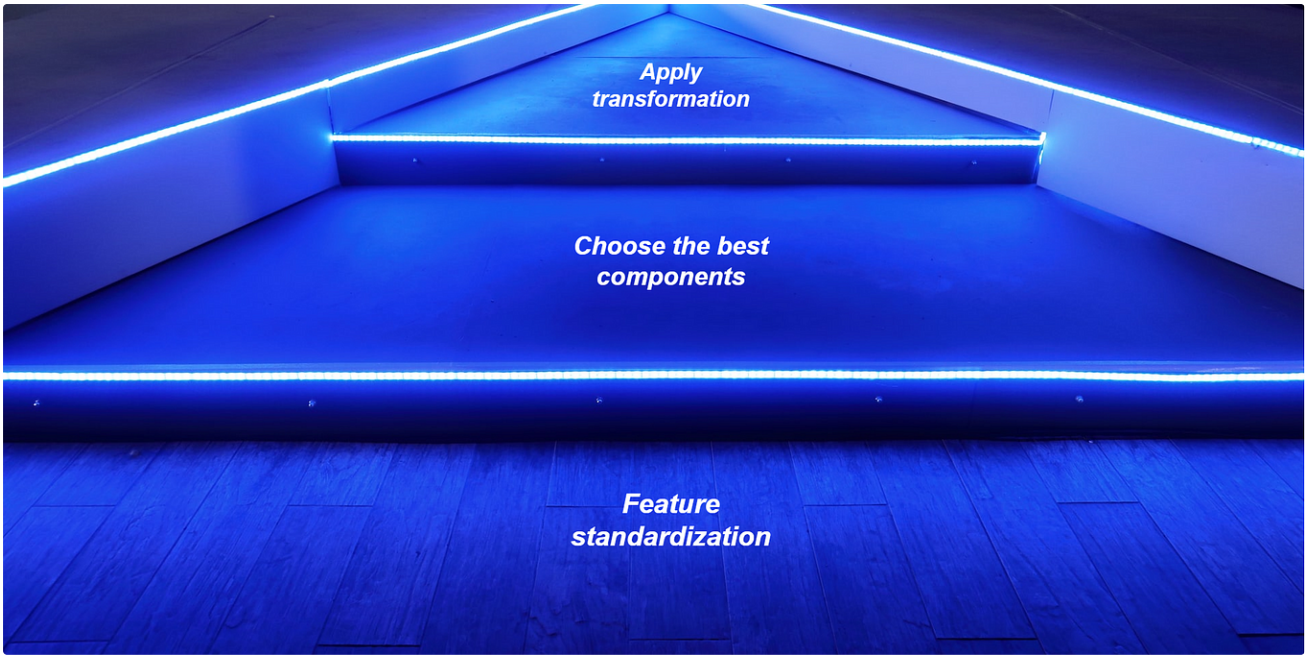


**Self-Improvement 101**

20 stories · 63 saves

**Productivity 101**

20 stories · 54 saves





Rukshan Pramoditha in Data Science 365

# 3 Easy Steps to Perform Dimensionality Reduction Using Principal Component Analysis (PCA)

Running the PCA algorithm twice is the most effective way of performing PCA

✨ · 11 min read · Jan 3

71 1

Albers Uzila in Level Up Coding

## Wanna Break into Data Science in 2023? Think Twice!

It won't be smooth sailing for you

✨ · 11 min read · Dec 23, 2022

835 14

Patrick Kalkman in ITNEXT

## Dependency Injection in Python

Building flexible and testable architectures in Python

✨ · 13 min read · Apr 14

644 5


Alexander Nguyen in Level Up Coding

## Why I Keep Failing Candidates During Google Interviews...

They don't meet the bar.

· 4 min read · Apr 13

3.7K 116

See more recommendations