numpy

August 31, 2024

[91]:
```
import numpy as np
import random
import warnings
warnings.filterwarnings("ignore")
```

Creating numpy arrays

[153]:
```
#Creating an array with random integers
arr1=np.random.randint(1,100,10)
```

[155]:
```
arr1
```

[155]:
```
array([42, 82, 30, 23, 10, 61, 51,  4, 82, 10])
```

[5]:
```
#Creating an array with random integers
arr2=np.random.randint(1,500,10)
```

[6]:
```
arr2
```

[6]:
```
array([ 37, 268, 274, 173,  60,  87,  39, 317,  30,  27])
```

Basic Operations on arrays

[8]:
```
#Return no.of dimensions(axes) of the array
arr1.ndim
```

[8]:
```
1
```

[9]:
```
#Returns the tuple of integers indicating the size of array in each dimension
arr1.shape
```

[9]:
```
(10,)
```

[10]:
```
#Returns the size of the array
arr1.size
```

[10]:
```
10
```

[11]:
```
#Returs the data type of the array
arr1.dtype
```

[11]: dtype('int32')

[12]:
```
#Return the size of each element in the array
arr1.itemsize
```

[12]: 4

[13]:
```
#return the total number of bytes for all elements in the array
arr1.nbytes
```

[13]: 40

Data manipulation

[15]:
```
#indexing
arr1[2]
```

[15]: 83

[16]:
```
arr2[4]
```

[16]: 60

[17]:
```
#Slicing
```

[18]:
```
arr1[1:4]
```

[18]: array([25, 83, 86])

[19]:
```
arr2[1:6]
```

[19]: array([268, 274, 173, 60, 87])

[20]:
```
#Creating a 2d array using replalce
arr2d=np.arange(0,20,2).reshape(2,5)
```

[21]:
```
arr2d
```

[21]:
```
array([[ 0,  2,  4,  6,  8],
       [10, 12, 14, 16, 18]])
```

[22]:
```
#indexing a 2d array the : left side represents the rows and right side␣
↪represents the columns
arr2d[:1]
```

[22]: array([[0, 2, 4, 6, 8]])

[66]:
```
arr2d1=np.arange(0,40,4).reshape(2,5)
```

[68]:
```
arr2d1
```

[68]:
```
array([[ 0,  4,  8, 12, 16],
       [20, 24, 28, 32, 36]])
```

[79]:
```
T_arr=arr2d1.T
```

Mathematical Operations

[25]:
```
#Addition
arr1+arr2
```

[25]:
```
array([ 71, 293, 357, 259, 128,  88,  53, 399,  89,  32])
```

[26]:
```
#Subtraction
arr2-arr1
```

[26]:
```
array([  3, 243, 191,  87,  -8,  86,  25, 235, -29,  22])
```

[27]:
```
#multiplication
arr1*arr2
```

[27]:
```
array([ 1258,  6700, 22742, 14878,  4080,    87,   546, 25994,  1770,
         135])
```

[28]:
```
#division
arr2/arr1
```

[28]:
```
array([ 1.08823529, 10.72      ,  3.30120482,  2.01162791,  0.88235294,
       87.        ,  2.78571429,  3.86585366,  0.50847458,  5.4       ])
```

[29]:
```
arr2 ** 2
```

[29]:
```
array([  1369,  71824,  75076,  29929,   3600,   7569,   1521, 100489,
          900,    729])
```

Matrix operation

[81]:
```
#Matrix multiplication (Dot Product)
np.dot(arr2d,T_arr)
```

[81]:
```
array([[ 240,  640],
       [ 640, 2040]])
```

[95]:
```
#Matrix Multiplication
arr2d*arr2d1
```

[95]:
```
array([[  0,   8,  32,  72, 128],
       [200, 288, 392, 512, 648]])
```

[97]:
```python
#Matrix Additio
arr2d+arr2d1
```

[97]:
```
array([[ 0,  6, 12, 18, 24],
       [30, 36, 42, 48, 54]])
```

[99]:
```python
#Matrix Subtraction
arr2d1-arr2d
```

[99]:
```
array([[ 0,  2,  4,  6,  8],
       [10, 12, 14, 16, 18]])
```

[101]:
```python
#Matrix Division
arr2d1/arr2d
```

[101]:
```
array([[nan,  2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.,  2.]])
```

Mathematical Functions in numpy arrays

[106]:
```python
n1=np.array([0,np.pi/2,np.pi])
```

[118]:
```python
#Exponential function
np.exp(n1)
```

[118]:
```
array([ 1.        ,  4.81047738, 23.14069263])
```

Trignometric Functions

[110]:
```python
np.sin(n1)
```

[110]:
```
array([0.0000000e+00, 1.0000000e+00, 1.2246468e-16])
```

[112]:
```python
np.cos(n1)
```

[112]:
```
array([ 1.000000e+00,  6.123234e-17, -1.000000e+00])
```

[114]:
```python
np.tan(n1)
```

[114]:
```
array([ 0.00000000e+00,  1.63312394e+16, -1.22464680e-16])
```

[116]:
```python
np.sqrt(n1)
```

[116]:
```
array([0.        , 1.25331414, 1.77245385])
```

Numpy Aggregation Functions

[169]:
```python
#Sum function in numpy arrays
print("Sum of all elements in the array")
print(np.sum(arr1))
```

```
#Sum along the axis
print("Sum along the 0th axis")
print(np.sum(arr2d,axis=0))
print("Sum along the 1st axis")
print(np.sum(arr2d,axis=1))
```

```
Sum of all elements in the array
395
Sum along the 0th axis
[10 14 18 22 26]
Sum along the 1st axis
[20 70]
```

[167]:
```
#Min value in the array
print("Min value of the array")
print(np.min(arr1))
#Max value of the array
print("Max value of the array")
print(np.max(arr1))
```

```
Min value of the array
4
Max value of the array
82
```

[165]:
```
#Mean of the array
print("Mean value of the array")
print(np.mean(arr1))
```

```
Mean value of the array
39.5
```

[163]:
```
#Median of the array
print("Median value of the array")
print(np.median(arr1))
```

```
Median value of the array
36.0
```

[147]:
```
#Standard Deviation of the array
print("Standard Deviation of the array")
print(np.std(arr1))
```

```
Standard Deviation of the array
31.987653868328636
```

[159]:
```
#Variance of the array
print("Variance of the array")
print(np.var(arr1))
```

```
Variance of the array
757.65
```

[161]:
```python
#Productof the array
print("Product of the array")
print(np.prod(arr1))
```

```
Product of the array
411890432
```

[171]:
```python
#Cumulative Sum of the array
print("Cumulative Sum of the array")
print(np.cumsum(arr1))
```

```
Cumulative Sum of the array
[ 42 124 154 177 187 248 299 303 385 395]
```

[173]:
```python
#Cumulative Product of the array
print("Cumulative Product of the array")
print(np.cumprod(arr1))
```

```
Cumulative Product of the array
[        42       3444     103320    2376360   23763600 1449579600
  914115568 -638505024 -817804416  411890432]
```

[179]:
```python
#Percentile of the array
print("Percentile of the array")
np.percentile(arr1,50)
```

```
Percentile of the array
```

[179]:
```
36.0
```

Data Analysis using numpy

[184]:
```python
#Generating two large datasets
data1=np.random.rand(1000000)
data2=0.1*data1 + np.random.rand(1000000)*0.1
```

[186]:
```python
data1
```

[186]:
```
array([0.31869265, 0.98678722, 0.68401622, …, 0.26870889, 0.31653988,
       0.08508693])
```

[190]:
```python
data2
```

[190]:
```
array([0.12017988, 0.12562   , 0.16340455, …, 0.04953158, 0.05296569,
       0.03282584])
```

[194]:
```
#Co-relation of the datas
co_mat=np.corrcoef(data1,data2)
co_mat[0,1]
```

[194]: 0.7070486239491695

[200]:
```
#Udentifying Outlliers
from scipy import stats

data=np.random.rand(1000000)
data[::10000]=10

z_scores=np.abs(stats.zscore(data))

outliers=np.where(z_scores>3)[0]
```

[202]:
```
outliers
```

[202]:
```
array([     0,  10000,  20000,  30000,  40000,  50000,  60000,  70000,
        80000,  90000, 100000, 110000, 120000, 130000, 140000, 150000,
       160000, 170000, 180000, 190000, 200000, 210000, 220000, 230000,
       240000, 250000, 260000, 270000, 280000, 290000, 300000, 310000,
       320000, 330000, 340000, 350000, 360000, 370000, 380000, 390000,
       400000, 410000, 420000, 430000, 440000, 450000, 460000, 470000,
       480000, 490000, 500000, 510000, 520000, 530000, 540000, 550000,
       560000, 570000, 580000, 590000, 600000, 610000, 620000, 630000,
       640000, 650000, 660000, 670000, 680000, 690000, 700000, 710000,
       720000, 730000, 740000, 750000, 760000, 770000, 780000, 790000,
       800000, 810000, 820000, 830000, 840000, 850000, 860000, 870000,
       880000, 890000, 900000, 910000, 920000, 930000, 940000, 950000,
       960000, 970000, 980000, 990000], dtype=int64)
```

[206]:
```
#Calculating Percentiles
data=np.random.randn(1000000)

percentiles=np.percentile(data,[25,50,75])
```

[208]:
```
percentiles
```

[208]:
```
array([-6.73489274e-01, -2.56438795e-04,  6.73844579e-01])
```

[210]:
```
#Normalization of the data
mean=np.mean(data)
std_dev=np.std(data)
normalized_data=(data-mean)/std_dev
```

[212]:
```
normalized_data
```

```
[212]: array([ 0.26778676,  1.33972609, -0.5302307 , …,  1.1474749 ,
        -0.0848335 , -1.34901546])
```

Applications in data Science

In this program, we demonstrated how NumPy can be effectively utilized for data analysis tasks such as finding correlations, identifying outliers, and calculating percentiles. For a data science professional, the use of NumPy is invaluable due to its efficiency, scalability, and rich set of functionalities designed for numerical and scientific computations.

Advantages of Using NumPy Over Traditional Python Data Structures Efficiency and Speed:

Vectorization: NumPy operations are vectorized, meaning they operate on entire arrays at once without the need for explicit loops. This leads to significant speed improvements, especially when working with large datasets. Optimized C Implementation: NumPy is implemented in C, which allows it to perform computations much faster than standard Python lists and loops. Memory Efficiency: NumPy arrays consume less memory compared to Python lists due to their compact storage in contiguous memory blocks, leading to better performance and reduced overhead. Broad Functionality:

NumPy provides a wide range of mathematical functions that are not available with standard Python data structures. This includes linear algebra operations, random number generation, statistical computations, and more. The ability to perform complex operations like matrix multiplication, Fourier transforms, and linear regression with just a few lines of code makes NumPy an indispensable tool. Integration with Other Libraries:

NumPy serves as the foundation for many other libraries in the Python ecosystem, such as Pandas, SciPy, Scikit-learn, TensorFlow, and PyTorch. This seamless integration makes it easier to perform data manipulation, statistical analysis, and machine learning tasks. Handling Large Datasets:

With NumPy, data scientists can efficiently process and analyze datasets that may be too large or complex for standard Python lists and loops to handle effectively. This is crucial in big data scenarios, where performance and scalability are key. Real-World Examples Where NumPy's Capabilities Are Crucial Machine Learning:

In machine learning, NumPy is often used for data preprocessing, such as normalizing datasets, calculating covariance matrices, and implementing algorithms like gradient descent. Libraries like Scikit-learn rely heavily on NumPy for numerical computations. Example: A machine learning engineer might use NumPy to normalize a dataset of images before feeding them into a neural network model. Financial Analysis:

NumPy is essential for performing quantitative financial analysis, such as calculating portfolio returns, risk metrics, and performing Monte Carlo simulations. Example: A financial analyst could use NumPy to calculate the Sharpe ratio of a portfolio or to model asset price movements using stochastic processes. Scientific Research:

Researchers in fields like physics, biology, and chemistry use NumPy to handle large datasets, perform simulations, and analyze experimental data. NumPy's ability to handle multi-dimensional arrays and its support for scientific computing make it a go-to tool for researchers. Example: A physicist might use NumPy to simulate the behavior of particles in a fluid or to analyze data from a high-energy physics experiment. Conclusion NumPy is a cornerstone of numerical computing in Python, offering powerful tools for efficient and scalable data analysis. Its advantages over traditional Python data structures, such as speed, memory efficiency, and a rich set of functions, make it an indispensable tool for data science professionals. Whether in machine learning, financial analysis, or scientific research, NumPy's capabilities enable professionals to tackle complex problems and derive meaningful insights from large datasets with ease.

```
[ ]:
```