**NumPy & Pandas**

Python is increasingly being used as a scientific language. Matrix and vector manipulations are extremely important for scientific computations. Both NumPy and Pandas have emerged to be essential libraries for any scientific computation, including machine learning, in python due to their intuitive syntax and high-performance matrix computation capabilities.

**What is NumPy?**

NumPy stands for ‘Numerical Python’ or ‘Numeric Python’. It is an open source module of Python which provides fast mathematical computation on arrays and matrices. Since, arrays and matrices are an essential part of the Machine Learning ecosystem, NumPy along with Machine Learning modules like Scikit-learn, Pandas, Matplotlib, TensorFlow, etc. complete the Python Machine Learning Ecosystem.

NumPy provides the essential multi-dimensional array-oriented computing functionalities designed for high-level mathematical functions and scientific computation. Numpy can be imported into the notebook using

>>> import numpy as np

NumPy’s main object is the homogeneous multidimensional array. It is a table with same type elements, i.e, integers or string or characters (homogeneous), usually integers. In NumPy, dimensions are called axes. The number of axes is called the rank.

There are several ways to create an array in NumPy like np.array, np.zeros, no.ones, etc. Each of them provides some flexibility.

|  |  |
| --- | --- |
| Command to create an array | Example |
| np.array | >>> a = np.array([1, 2, 3])  >>> type(a)  <type 'numpy.ndarray'>  >>> b = np.array((3, 4, 5))  >>> type(b)  <type 'numpy.ndarray'> |
| np.ones | >>> np.ones( (3,4), dtype=np.int16 )  array([[ 1,  1,  1,  1],        [ 1,  1,  1,  1],        [ 1,  1,  1,  1]]) |
| np.full | >>> np.full( (3,4), 0.11 )  array([[ 0.11,  0.11,  0.11,  0.11],    [ 0.11,  0.11,  0.11,  0.11],    [ 0.11,  0.11,  0.11,  0.11]]) |
| np.arange | >>> np.arange( 10, 30, 5 )  array([10, 15, 20, 25])  >>> np.arange( 0, 2, 0.3 )  # it accepts float arguments  array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8]) |
| np.linspace | >>> np.linspace(0, 5/3, 6)  array([0. , 0.33333333 , 0.66666667 , 1. , 1.33333333  1.66666667]) |
| np.random.rand(2,3) | >>> np.random.rand(2,3)  array([[ 0.55365951,  0.60150511,  0.36113117],        [ 0.5388662 ,  0.06929014,  0.07908068]]) |
| np.empty((2,3)) | >>> np.empty((2,3))  array([[ 0.21288689,  0.20662218,  0.78018623],        [ 0.35294004,  0.07347101,  0.54552084]]) |

Some of the important attributes of a NumPy object are:

1. **Ndim:** displays the dimension of the array
2. **Shape:** returns a tuple of integers indicating the size of the array
3. **Size:** returns the total number of elements in the NumPy array
4. **Dtype**: returns the type of elements in the array, i.e., int64, character
5. **Itemsize:** returns the size in bytes of each item
6. **Reshape**: Reshapes the NumPy array

NumPy array elements can be accessed using indexing. Below are some of the useful examples:

* A[2:5] will print items 2 to 4. Index in NumPy arrays starts from 0
* A[2::2] will print items 2 to end skipping 2 items
* A[::-1] will print the array in the reverse order
* A[1:] will print from row 1 to end

**Why NumPy and Pandas over regular Python arrays?**

In python, a vector can be represented in many ways, the simplest being a regular python list of numbers. Since Machine Learning requires lots of scientific calculations, it is much better to use NumPy’s ndarray, which provides a lot of convenient and optimized implementations of essential mathematical operations on vectors.

Vectorized operations perform faster than matrix manipulation operations performed using loops in python. For example, to carry out a 100 \* 100 matrix multiplication, vector operations using NumPy are two orders of magnitude faster than performing it using loops.

Some ways in which NumPy arrays are different from normal Python arrays are:

1. If you assign a single value to a ndarray slice, it is copied across the whole slice

|  |  |
| --- | --- |
| NumPy Array | Regular Python array |
| >>> a = np.array([1, 2, 5, 7, 8])  >>> a[1:3] = -1  >>> a  array([ 1, -1, -1,  7,  8]) | >>> b = [1, 2, 5, 7, 8]  >>> b[1:3] = -1  TypeError: can only assign an iterable |

So, it is easier to assign values to a slice of an array in a NumPy array as compared to a normal array wherein it may have to be done using loops.

1. ndarray slices are actually views on the same data buffer. If you modify it, it is going to modify the original ndarray as well.

|  |  |
| --- | --- |
| NumPy array slice | Regular python array slice |
| >>> a = np.array([1, 2, 5, 7, 8])  >>> a\_slice = a[1:5]  >>> a\_slice[1] = 1000  >>> a  array([   1,    2, 1000, 7,    8])  # Original array was modified | >>> a=[1,2,5,7,8]  >>> b=a[1:5]  >>> b[1]=3  >>> print(a)  >>> print(b)  [1, 2, 5, 7, 8]  [2, 3, 7, 8] |

If we need a copy of the NumPy array, we need to use the copy method as another\_slice = another\_slice = a[2:6].copy(). If we modify another\_slice, a remains same

1. The way multidimensional arrays are accessed using NumPy is different from how they are accessed in normal python arrays. The generic format in NumPy multi-dimensional arrays is:

Array[row\_start\_index:row\_end\_index, column\_start\_index: column\_end\_index]

NumPy arrays can also be accessed using boolean indexing. For example,

>>> a = np.arange(12).reshape(3, 4)

array([[ 0, 1, 2, 3], [ 4, 5, 6, 7], [ 8, 9, 10, 11]])

>>> rows\_on = np.array([True, False, True])

>>> a[rows\_on , : ]      # Rows 0 and 2, all columns

array([[ 0,  1,  2,  3],

      [ 8,  9, 10, 11]])

NumPy arrays are capable of performing all basic operations such as addition, subtraction, element-wise product, matrix dot product, element-wise division, element-wise modulo, element-wise exponents and conditional operations.

**What is Pandas?**

Similar to NumPy, Pandas is one of the most widely used python libraries in data science. It provides high-performance, easy to use structures and data analysis tools. Unlike NumPy library which provides objects for multi-dimensional arrays, Pandas provides in-memory 2d table object called Dataframe. It is like a spreadsheet with column names and row labels.

Hence, with 2d tables, pandas is capable of providing many additional functionalities like creating pivot tables, computing columns based on other columns and plotting graphs. Pandas can be imported into Python using:

>>> import pandas as pd

Some commonly used data structures in pandas are:

1. **Series objects**: 1D array, similar to a column in a spreadsheet
2. **DataFrame objects:** 2D table, similar to a spreadsheet
3. **Panel objects:** Dictionary of DataFrames, similar to sheet in MS Excel

Pandas Series object is created using pd.Series function. Each row is provided with an index and by defaults is assigned numerical values starting from 0. Like NumPy, Pandas also provide the basic mathematical functionalities like addition, subtraction and conditional operations and broadcasting.

Pandas dataframe object represents a spreadsheet with cell values, column names, and row index labels. Dataframe can be visualized as dictionaries of Series. Dataframe rows and columns are simple and intuitive to access. Pandas also provide SQL-like functionality to filter, sort rows based on conditions. For example,

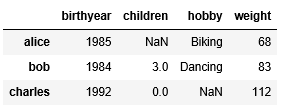
>>> people\_dict = { "weight": pd.Series([68, 83, 112],index=["alice", "bob", "charles"]),   "birthyear": pd.Series([1984, 1985, 1992], index=["bob", "alice", "charles"], name="year"),

"children": pd.Series([0, 3], index=["charles", "bob"]),

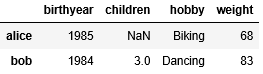
"hobby": pd.Series(["Biking", "Dancing"], index=["alice", "bob"]),}

>>> people = pd.DataFrame(people\_dict)

>>> people



>>> people[people["birthyear"] < 1990]



New columns and rows can be easily added to the dataframe. In addition to the basic functionalities, pandas dataframe can be sorted by a particular column.

Dataframes can also be easily exported and imported from CSV, Excel, JSON, HTML and SQL database. Some other essential methods that are present in dataframes are:

1. **head():** returns the top 5 rows in the dataframe object
2. **tail():** returns the bottom 5 rows in the dataframe
3. **info():** prints the summary of the dataframe
4. **describe():** gives a nice overview of the main aggregated values over each column