> DATA ARE BECOMING THE NEW RAW MATERIAL OF BUSINESS
>
> — THE ECONOMIST

NumPy and pandas – Crucial Tools for Data Scientists

Posted by Don Fox (http://www.thedataincubator.com/team.html#don fox) on February 21, 2018

This technical article was written for The Data Incubator by ***Don Fox (https://www.linkedin.com/in/don-fox-969b439a/)***, a Fellow of our 2017 Summer cohort in New York City.

When it comes to scientific computing and data science, two key python packages are NumPy and pandas. NumPy is a powerful python library that expands Python's functionality by allowing users to create multi-dimenional array objects (`ndarray`). In addition to the creation of `ndarray` objects, NumPy provides a large set of mathematical functions that can operate quickly on the entries of the `ndarray` without the need of for loops.

Using NumPy

Below is an example of the usag a d calculates the cosine for each entry.

In [23]:

```
import numpy as np

X = np.random.random((4, 2))  # create random 4x2 array
y = np.cos(X)                 # take the cosine on each entry of X

print y
print "\n The dimension of y is", y.shape
```

```
[[ 0.95819067  0.60474588]
 [ 0.78863282  0.95135038]
 [ 0.82418621  0.93289855]
 [ 0.67706351  0.83420891]]

 The dimension of y is (4, 2)
```

We can easily access entries of an array, call individual elements, and select certain rows and columns.

In [24]:

```
print y[0, :]   # select 1st row
print y[:, 1]   # select 1st column
print y[2, 1]   # select element y_12
print y[1:2, :] # select rows 2nd and 3rd row
```

```
[ 0.95819067  0.60474588]
[ 0.60474588  0.95135038  0.93289855  0.83420891]
0.932898546321
[[ 0.78863282  0.95135038]]
```

Using pandas

The pandas (PANel + DAta) Python library allows for easy and fast data analysis and manipulation tools by providing numerical tables and time series data structures called `DataFrame` and `Series` , respectively. Pandas was created to do the following:

- provide data structures that can handle both time and non-time series data
- allow mathematical operations on the data structures, ignoring the metadata of the data structures
- use relational operations like those found in programming languages like SQL (join, group by, etc.)
- handle missing data

Below is an example of the usag pa

In [25]:

```
import pandas as pd

# create data
states = ['Texas', 'Rhode Island', 'Nebraska'] # string
population = [27.86E6, 1.06E6, 1.91E6]         # float
electoral_votes = [38, 3, 5]                   # integer
is_west_of_MS = [True, False, True]            # Boolean

# create and display DataFrame
headers = ('State', 'Population', 'Electoral Votes', 'West of Mississippi')
data = (states, population, electoral_votes, is_west_of_MS)
data_dict = dict(zip(headers, data))

df1 = pd.DataFrame(data_dict)
df1
```

Out[25]:

	Electoral Votes	Population	State	West of Mississippi
0	38	27860000.0	Texas	True
1	3	1060000.0	Rhode Island	False
2	5	1910000.0	Nebraska	True

In the above code, we created a pandas `DataFrame` object, a tabular data structure that resembles a spreadsheet like those used in Excel. For those familiar with SQL, you can view a `DataFrame` as an SQL table. The `DataFrame` we created consists of four columns, each with entries of different data types (integer, float, string, and Boolean).

NumPy and pandas

Pandas is built on top of NumPy, relying on `ndarray` and its fast and efficient array based mathematical functions. For example, if we wanted to calculate the mean population across the states, we can run

In [26]:

```
print df1['Population'].mean()
```

```
10276666.6667
```

Pandas relies on NumPy data types for the entries in the `DataFrame`. Printing the types of individual entries using `iloc` shows

In [27]:

```
print type(df1['Electoral Votes'].iloc[0])
print type(df1['Population'].iloc[0])
print type(df1['West of Mississippi'].iloc[0])
```

```
<type 'numpy.int64'>
<type 'numpy.float64'>
<type 'numpy.bool_'>
```

Another example of the pandas and NumPy compatibility is if we have a `DataFrame` that is composed of purely numerical data we can apply NumPy functions. For example,

In [28]:

```
df2 = pd.DataFrame({"times": [1.0, 2.0, 3.0, 4.0], "more times": [5.0, 6.0, 7.0, 8.0]}
df2 = np.cos(df2)
df2.head()
```

Out[28]:

	more times	times
0	0.283662	0.540302
1	0.960170	-0.416147
2	0.753902	-0.989992
3	-0.145500	-0.653644

Pandas was built to ease data analysis and manipulation. Two import pandas methods are `groupby` and `apply`. The `groupby` method groups the `DataFrame` by values of a certain column and applies some aggregating function on the resulting groups. For example, if we want to determine the maximum population for states grouped by if they are either west or east of the Mississippi river, the syntax is

In [29]:

```
df1.groupby('West of Mississippi').agg('max')
```

Out[29]:

	Electoral Votes	Population	State
West of Mississippi			
False	3	1060000.0	Rhode Island
True	38	27860000.0	Texas

The `apply` method accepts a fu[illegible] s c[illegible] `es` object. This method is useful for applyin[illegible] en [illegible] in a pandas `DataFrame`. For example, we can create a `Series` object that tells us if a state's population is more than two million. The result is a `Series` object that we can append to our original `DataFrame` object.

In [30]:

```
more_than_two_million = df1['Population'].apply(lambda x: x > 2E6)  # create Series ob
df1['More than a Million'] = more_than_two_million  # append Series object to our orig
df1.head()
```

Out[30]:

	Electoral Votes	Population	State	West of Mississippi	More than a Million
0	38	27860000.0	Texas	True	True
1	3	1060000.0	Rhode Island	False	False
2	5	1910000.0	Nebraska	True	False

Accessing columns is inuitive, and returns a pandas `Series` object.

In [31]:

```
print df1['Population']
print type(df1['Population'])
```

```
0    27860000.0
1     1060000.0
2     1910000.0
Name: Population, dtype: float64
<class 'pandas.core.series.Series'>
```

A `DataFrame` is composed of multiple `Series`. The `DataFrame` class resembles a collection of NumPy arrays but with labeled axes and mixed data types across the columns. In fact, `Series` is subclass of NumPy's `ndarray`. While you can achieve the same results of certain pandas methods using NumPy, the result would require more lines of code. Pandas expands on NumPy by providing easy to use methods for data analysis to operate on the `DataFrame` and `Series` classes, which are built on NumPy's powerful `ndarray` class.

How memory is configured in NumPy

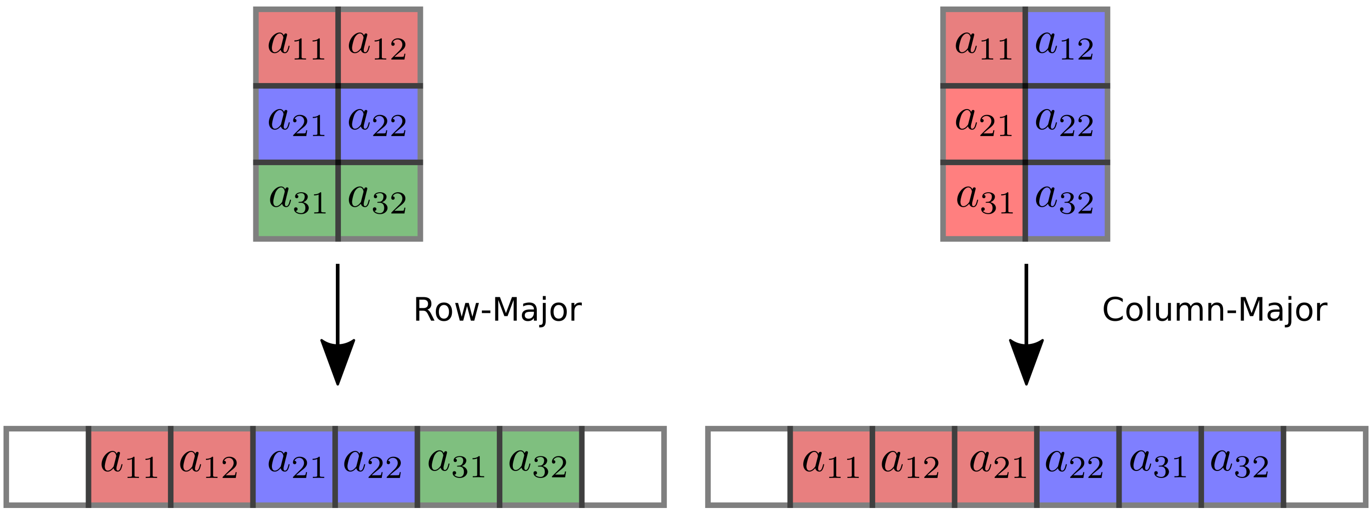
The power of NumPy comes from the `ndarray` class and how it is laid out in memory. The `ndarray` class consists of

- the data type of the entires of the array
- a pointer to a contiguous b /e ' reside
- a tuple of the array's shape

- a tuple of the array's stride

The shape refers to the dimension of the array while the stride is the number of bytes to step in a particular dimension when traversing an array in memory. With both the stride and the shape, NumPy has sufficient information to access the array's entries in memory.

By default, NumPy arranges the data in row-major order, like in C. Row-major order lays out the entries of the array by groupings of rows. An alternative is column-major ordering, as used in Fortran and MATLAB, which uses columns as the grouping. NumPy is capable of implementing both ordering schemes by passing the keyword `order` when creating an array. See the figure below for the differeneces in the schemes.



The contunguous memory layout allows NumPy to use vector processors in modern CPUs and array computations. Array computations are efficient because NumPy can loop through the entries in data properly by knowing the location in memory and the data type of the entries. NumPy can also link to established and highly optimized linear algebra libraries such as BLAS and LAPACK. As you can see, using the NumPy `ndarray` offers more efficient and fast computations over the native Python list. No wonder pandas and other Python libraries are built on top of NumPy. However, the infrastructure of the `ndarray` class must require all entries to be the same data type, something that a Python `list` class is not limited to.

Hetereogeneous data types in pandas

As mentioned earlier, the pandas `DataFrame` class can store hetereogeneous data; each column contains a `Series` object of a different data type. The `DataFrame` is stored as several blocks in memory, where each block contains the columns of the `DataFrame` that have the same data type. For example, a `DataFrame` with five columns comprised of two columns of floats, two columns of integers, and one Boolean column will be stored using three blocks.

With the data of the `DataFrame` stored using blocks grouped by data, operations within blocks are effcient, as described previously on why NumPy operations are fast. However, operations involving several blocks will not l e `Frame` object can be accessed using `.data`.

In [32]:

```
df1._data
```

Out[32]:

```
BlockManager
Items: Index([u'Electoral Votes', u'Population', u'State', u'West of Mississippi',
       u'More than a Million'],
      dtype='object')
Axis 1: RangeIndex(start=0, stop=3, step=1)
FloatBlock: slice(1, 2, 1), 1 x 3, dtype: float64
IntBlock: slice(0, 1, 1), 1 x 3, dtype: int64
BoolBlock: slice(3, 4, 1), 1 x 3, dtype: bool
ObjectBlock: slice(2, 3, 1), 1 x 3, dtype: object
BoolBlock: slice(4, 5, 1), 1 x 3, dtype: bool
```

The `DataFrame` class can allow columns with mixed data types. For these cases, the data type for the column is referred to as `object`. When the data type is `object`, the data is no longer stored in the NumPy `ndarray` format, but rather a continguous block of pointers where each pointer referrences a Python object. Thus, operations on a `DataFrame` involving `Series` of data type `object` will not be efficient.

Strings are stored in pandas as Python `object` data type. This is because strings have variable memory size. In contrast, integers and floats have a fixed byte size. However, if a `DataFrame` has columns with categorial data, encoding the entries using integers will be more memory and computational efficient. For example, a column containing entries of “small”, “medium”, and “large” can be coverted to 0, 1, and 2 and the data type of that new column is now an integer.

The importance of understanding Numpy and pandas

Through this article, we have seen

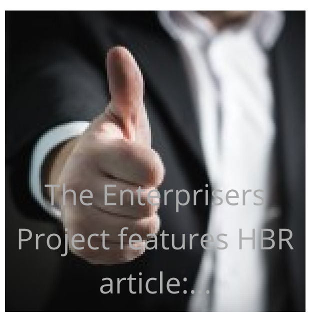
- examples of usage of NumPy and pandas
- how memory is configured in NumPy
- how pandas relies on NumPy
- how pandas deals with hetereogeneous data types

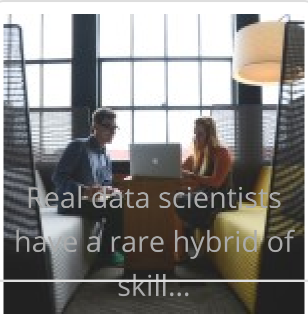
While knowing how NumPy and pandas work is not necessary to use these tools, knowing the working of these libraries and how they are related enables data scientists to effectively yield these tools. More effective use of these tools becomes more important for larger data sets and more complex analysis, where even a small improvement in terms of percentage translates to large time savings.

Visit our website to learn more about our offerings:

- Data Science Fellowship (https://www.thedataincubator.com/fellowship.html) – a free, full-time, eight-week bootcamp program for PhD and master's graduates looking to get hired as professional Data Scientists in New York City, Washington DC, San Francisco, and Boston.
- Hiring Data Scientists (https://www.thedataincubator.com/hiring.html)
- Corporate data science training (https://www.thedataincubator.com/training.html)
- **Online data science courses:** introductory part-time bootcamps – taught by our expert Data Scientists in residence, and based on our Fellowship curriculum – for busy professionals to boost their data science skills in their spare time.
  - Data Science Foundations (https://www.thedataincubator.com/foundations.html)
  - Applied Machine Learning (https://www.thedataincubator.com/machine-learning.html)
  - Data Science for Business Leaders (https://www.thedataincubator.com/business-leaders.html)
  - Artificial Intelligence with TensorFlow (https://www.thedataincubator.com/artificial-intelligence-tensorflow.html)
  - Distributed Computing with Spark (https://www.thedataincubator.com/spark.html)
  - Bitcoin and Blockchain (https://www.thedataincubator.com/bitcoin-and-blockchain.html)

Related Posts:





Subscribe to our mailing list

Enter your email

Subscribe

← Back to index (https://blog.thedataincubator.com)

Tags: #technical post (https://blog.thedataincubator.com/tag/technical-posts/) comparison (https://blog.thedataincubator.com/tag/comparison/) numpy (https://blog.thedataincubator.com/tag/numpy/) pandas (https://blog.thedataincubator.com/tag/pandas/)

> 140,000-190,000 MORE DEEP ANALYTICAL TALENT POSITIONS NEEDED
>
> — MCKINSEY GLOBAL INSTITUTE

Technology and training partners:

Follow us on social media:

(https://twitter.com/thedatainc) (https://www.facebook.com/dataincubator) (https://www.linkedin.com/company/the-data-incubator) (https://www.youtube.com/c/data

(https://www.thedataincubator.com)

Subscribe to our mailing list

Enter your email

Subscribe