

Intro to Programming for Public Policy Week 5

NumPy and Pandas Introduction

Eric Potash

April 24, 2018

NumPy arrays

Types

- ▶ So far we've used built-in python types of lists, dictionary, and sets
- ▶ NumPy provides a type called `ndarray` (n-dimensional array)

Benefits of ndarray

- ▶ ndarrays are fast and space efficiency multidimensional arrays
- ▶ They provide vectorized arithmetic operations and broadcasting capabilities
- ▶ The numpy library provides many useful mathematical functions, especially matrix operations

Limitations of ndarray

- ▶ NumPy does not provide high-level data analysis functionality
- ▶ But having an understanding of NumPy arrays and array-oriented computing will help

Create an ndarray

- Create an ndarray like so:

```
>>> import numpy as np
>>> arr = np.array([[0, 1.5, 2.0],
                    [-1.0, 3.0, 5.0]])

>>> arr
array([[ 0. ,  1.5,  2. ],
       [-1. ,  3. ,  5. ]])

>>> type(arr)
numpy.ndarray
```

Create an ndarray

- Create an ndarray like so:

```
>>> import numpy as np
>>> arr = np.array([[0, 1.5, 2.0],
                    [-1.0, 3.0, 5.0]])

>>> arr
array([[ 0. ,  1.5,  2. ],
       [-1. ,  3. ,  5. ]])
>>> type(arr)
numpy.ndarray
```

- Conveniently get its dimensions using the shape attribute:

```
>>> arr.shape
(2, 3)
```

Arithmetic with ndarray

- Can multiply by a number to scale:

```
>>> arr = np.array([[0, 1.5, 2.0],  
                    [-1.0, 3.0, 5.0]])  
  
>>> arr * 3  
array([[ 0.,  3.,  4.],  
       [-2.,  6., 10.]])
```


Arithmetic with ndarray

- ▶ Can multiply by a number to scale:

```
>>> arr = np.array([[0, 1.5, 2.0],  
                    [-1.0, 3.0, 5.0]])  
  
>>> arr * 3  
array([[ 0.,  3.,  4.],  
       [-2.,  6., 10.]])
```

- ▶ Can add two ndarrays:

```
>>> arr + arr  
array([[ 0.,  3.,  4.],  
       [-2.,  6., 10.]])
```

Arithmetic with ndarray

- ▶ Can multiply by a number to scale:

```
>>> arr = np.array([[0, 1.5, 2.0],  
                    [-1.0, 3.0, 5.0]])  
  
>>> arr * 3  
array([[ 0.,  3.,  4.],  
       [-2.,  6., 10.]])
```

- ▶ Can add two ndarrays:

```
>>> arr + arr  
array([[ 0.,  3.,  4.],  
       [-2.,  6., 10.]])
```

- ▶ Multiplication is element-wise:

```
>>> arr * arr  
array([[ 0. ,  2.25,  4.  ],  
       [ 1. ,  9. , 25. ]])
```

Indexing

On the surface, you can index an ndarray like a list:

```
>>> arr = np.array([[0, 1.5, 2.0],  
                    [-1.0, 3.0, 5.0]])  
  
>>> arr[1]  
array([-1.,  3.,  5.])  
  
>>> arr[1][1:3]  
array([ 3.,  5.])
```

Assignment

Unlike with a regular list, you can assign a value to a slice of an ndarray:

```
>>> arr = np.array([[0, 1.5, 2.0],  
                    [-1.0, 3.0, 5.0]])  
  
>>> arr[1][:] = 0  
  
>>> arr  
array([[ 0. ,  1.5,  2. ],  
       [ 0. ,  0. ,  0.]])
```

Mathematical functions

NumPy ndarrays have many builtin mathematical functions:

```
>>> arr = np.array([[0, 1.5, 2.0],  
                    [-1.0, 3.0, 5.0]])  
  
>>> arr.sum()  
3.5  
  
>>> arr.min()  
0.0  
  
>>> arr.max()  
2.0  
  
>>> arr.std()  
0.83748963509340746
```

Mathematical functions per axis

In addition to calculating these values across the entire ndarray you can calculate per axis:

```
>>> ar = np.array([[0, 1.5, 2.0],  
                  [-1.0, 3.0, 5.0]])  
  
>>> arr.sum(axis=0)  
array([ 0. ,  1.5,  2. ])  
  
>>> arr.sum(axis=1)  
array([ 3.5,  0. ])
```

Pandas

Overview

- ▶ Pandas is an important module for data analysis in python
- ▶ Pandas builds on numpy to provide data structures with:
 - ▶ Mixed types
 - ▶ Column and row names
 - ▶ Time series functionality
 - ▶ Lots of input and output formats (CSV, MS Excel, etc.)
 - ▶ Easy plotting with `matplotlib`

Data structures

There are two main data structures (types) in pandas: `Series` and `DataFrame`.

Pandas Series

Overview

A Series is a one-dimensional object (like a list or a 1d ndarray).
Additionally, a Series stores an array of data labels, called its index.

Series creation

- ▶ Create a series from a list:

```
>>> import pandas as pd
>>> s = pd.Series([3,10,5,4,-1])
>>> s
0      3
1     10
2      5
3      4
4     -1
dtype: int64
```

- ▶ Notes:

- ▶ Python represents the Series with the index on the left and the values on the right.
- ▶ Since we didn't specify an index, the default is like a list: 0 through N-1 where N is the length of the data.
- ▶ Pandas automatically assigned a datatype (dtype) of integer (int64) to this Series

Specifying an index

To specify a label for each point in the Series, called an index:

```
>>> s = pd.Series(  
    [3,10,5,4],  
    index=['Dorothy', 'Alice', 'Chris', 'Bob'])  
  
>>> s  
Dorothy      3  
Alice       10  
Chris        5  
Bob          4  
dtype: int64
```

Using an index

Now we can use the index to select from the series:

```
>>> s['Chris']  
5  
>>> s['Bob'] = -10  
>>> s[['Alice', 'Bob']]  
Alice      10  
Bob       -10  
dtype: int64
```

Selecting multiple values returns a (sub-)Series.

Operations

```
▶ >>> s*2
Dorothy      6
Alice        20
Chris        10
Bob          -20
dtype: int64
```

```
▶ >>> s >= 10
Dorothy      False
Alice        True
Chris        False
Bob          False
dtype: bool
```

```
▶ >>> s.sum()
22
```

Boolean filtering

You can subset a Series with series of the same index that has boolean values:

```
>>> s[s >= 10]  
Alice      10  
dtype: int64
```


in keyword

The `in` operator checks the index of a Series:

```
>>> 'Alice' in s
True
>>> 'Eric' in s
False
```

Alternative constructor

You can also think of a Series like a dictionary where the keys are the index. Unlike a dictionary, a Series is ordered. You can construct a series from a dictionary:

```
>>> s_data = {'Ohio': 35000, 'Texas': 71000,  
              'Oregon': 16000, 'Utah': 5000}  
>>> s2 = pd.Series(s_data)  
>>> s2  
Ohio      35000  
Oregon    16000  
Texas     71000  
Utah       5000  
dtype: int64
```

Missing values

You can also specify an index when passing a dictionary:

```
>>> s2 = pd.Series(s_data,  
                    index=['California', 'Texas', 'Utah'])  
  
>>> s2  
California      NaN  
Texas           71000.0  
Utah            5000.0  
dtype: float64
```

Missing entries have the special value NaN (not-a-number).

Working with missing values

- Use `dropna()` to get a sub-series without the missing values

```
>>> s2.dropna()  
Texas      71000.0  
Utah       5000.0  
dtype: float64
```

Working with missing values

- ▶ Use `dropna()` to get a sub-series without the missing values

```
>>> s2.dropna()  
Texas      71000.0  
Utah       5000.0  
dtype: float64
```

- ▶ Use `.isnull()` and `.notnull()` to get a boolean series indicating whether the value was null or not:

```
>>> s2.isnull()  
California    True  
Texas         False  
Utah          False  
dtype: bool
```

Working with missing values

- ▶ Use `dropna()` to get a sub-series without the missing values

```
>>> s2.dropna()
Texas      71000.0
Utah       5000.0
dtype: float64
```

- ▶ Use `.isnull()` and `.notnull()` to get a boolean series indicating whether the value was null or not:

```
>>> s2.isnull()
California    True
Texas         False
Utah          False
dtype: bool
```

- ▶ You can use this series for indexing or in its own right:

```
>>> s2.notnull().sum()
2
```

Series alignment

If you perform an operation between two Series with different indexes, the result will be indexed by their union:

```
>>> s + s2
Alice      NaN
Bob        NaN
California NaN
Chris      NaN
Dorothy    NaN
Texas      NaN
Utah       NaN
dtype: float64
```

Pandas DataFrame

Overview

- ▶ A DataFrame is like a table or a spreadsheet
- ▶ It has an ordered list of columns
- ▶ The columns can have mixed types (e.g. numeric, boolean, string, etc.)

DataFrame anatomy

- ▶ Each column is a pandas Series
- ▶ The columns all share the same index, called the row index of the data frame
- ▶ The names of the columns form a second index, called the column index

Constructing a DataFrame

The most common way to construct a DataFrame in python is using a dictionary where the keys are column names and the values are equal length lists of data:

```
>>> data = {  
    'state': ['Ohio', 'Ohio', 'Ohio',  
             'Nevada', 'Nevada'],  
    'year': [2000, 2001, 2002, 2001, 2002],  
    'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
```

```
>>> df = pd.DataFrame(data)  
>>> df
```

	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002

Row-index

You can also pass a row index to the DataFrame constructor:

```
>>> df = pd.DataFrame(data,  
                        index=['one', 'two', 'three',  
                              'four', 'five'])
```

```
>>> df
```

	pop	state	year
one	1.5	Ohio	2000
two	1.7	Ohio	2001
three	3.6	Ohio	2002
four	2.4	Nevada	2001
five	2.9	Nevada	2002

Retrieving a column

- ▶ A column can be accessed in two equivalent ways
- ▶ They return a series that is *named* and index is the same as the DataFrame.

```
>>> df.state
one      Ohio
two      Ohio
three    Ohio
four     Nevada
five     Nevada
Name: state, dtype: object
```

```
>>> df['pop']
one      2000
two      2001
three    2002
four     2001
five     2002
Name: year, dtype: int64
```

Retrieving multiple columns

Retrieve multiple columns using the dict notation and passing a list of column names:

```
>>> df[['year', 'state']]
```

	year	state
one	2000	Ohio
two	2001	Ohio
three	2002	Ohio
four	2001	Nevada
five	2002	Nevada

Retrieving rows

Retrieve rows using `.loc[]`:

```
>>> df.loc['three']  
pop      3.6  
state    Ohio  
year     2002  
Name: three, dtype: object
```

```
>>> df.loc[['two', 'four']]  
      pop  state  year  
two  1.7   Ohio  2001  
four  2.4  Nevada  2001
```

Boolean Series indexing

As with Series, we can subset rows using booleans:

```
>>> df[df['pop'] > 2.5]
```

	pop	state	year
three	3.6	Ohio	2000
five	2.9	Nevada	2000

Boolean Series logic

You can perform logic on boolean series to create more complicated queries. Instead of using the usual python and, or, and not operators, we must use special symbols &, | and ~:

```
>>> (df.state == 'Ohio') & (df.year > 2000)
one      False
two      True
three    True
four     False
five     False
dtype: bool
```

```
>>> ~(df.state == 'Nevada')
one      True
two      True
three    True
four     False
five     False
Name: state, dtype: bool
```

Boolean Series logic indexing

```
>>> df[(df.state == 'Ohio') & (df.year > 2000)]
```

	pop	state	year
two	1.7	Ohio	2001
three	3.6	Ohio	2002

Create and assigning columns

- ▶ You can initialize a new column with a constant value:

```
>>> df['debt'] = 1.5
>>> df
```

	pop	state	year	debt
one	1.5	Ohio	2000	1.5
two	1.7	Ohio	2001	1.5
three	3.6	Ohio	2002	1.5
four	2.4	Nevada	2001	1.5
five	2.9	Nevada	2002	1.5

Create and assigning columns

- ▶ You can initialize a new column with a constant value:

```
>>> df['debt'] = 1.5
>>> df
```

	pop	state	year	debt
one	1.5	Ohio	2000	1.5
two	1.7	Ohio	2001	1.5
three	3.6	Ohio	2002	1.5
four	2.4	Nevada	2001	1.5
five	2.9	Nevada	2002	1.5

- ▶ Or you can use a list. If the column exists its values are updated:

```
>>> df['debt'] = [1.0, 3.0, 2.0, 4.0, 2.5]
```

More column creation

You can also create a column using existing columns:

```
>>> df['debt_per_pop'] = df.debt / df['pop']
>>> df
```

	pop	state	year	debt	debt_per_pop
one	1.5	Ohio	2000	1.5	1.000000
two	1.7	Ohio	2001	1.5	0.882353
three	3.6	Ohio	2002	1.5	0.416667
four	2.4	Nevada	2001	1.5	0.625000
five	2.9	Nevada	2002	1.5	0.517241