NumPy and Pandas I

2023-10-04

Agenda

- Introduction to NumPy fundamentals
- Getting started with Pandas

Why is NumPy good?

- Short for **Num**erical **Py**thon
- A foundational package for numerical computing in Python
- Many computational packages use NumPy's **array** as one of the standard interfaces for data exchange
- `ndarray` (n-dimensional array) from NumPy is faster and more efficient than Python's native lists for a variety of reasons
  - Homogeneity, vectorization, broadcasting, etc

NumPy and Pandas

- NumPy provides support for large multidimensional data (`ndarray`), and Pandas' key data structures (`Series` and `DataFrame`) are built on it

```
import pandas as pd
import numpy as np

s = pd.Series([1, 2, 3, 4])
print(type(s))
print(type(s.values))
```

```
<class 'pandas.core.series.Series'>
<class 'numpy.ndarray'>
```

NumPy and Pandas

- Pandas' key data structures (`Series` and `DataFrame`) are built on NumPy's `ndarray`

```
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
print(type(df))
print(type(df['A']))
print(type(df['A'].values))
```

```
<class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.series.Series'>
<class 'numpy.ndarray'>
```

NumPy and Pandas

- NumPy is mainly used or lower-level mathematical and numerical operation
- Pandas provides a comprehensive toolkit for structured data manipulation
  - Reading/writing data, handling missing data, merging/joining data sets, reshaping data, grouping and aggregating data, etc.

Generating `ndarray`

- Use `np.array()` function
- Converts input data to an `ndarray`
- Pass lists, tuples, or other sequence type data objects

Generating ndarray

- Data should be homogeneous

```
arr1 = np.array([1, 2, 4])
print(arr1)
arr2 = np.array([1, '2', 4])
print(arr2)
arr3 = np.array([1, False, 4])
print(arr3)
arr4 = np.array([1, None, 4])
print(arr4)
```

```
[1 2 4]
['1' '2' '4']
[1 0 4]
[1 None 4]
```

Don't get confused with lists

- Note the commas

```
arr1 = np.array([1, 2, 4])
print(arr1)
list1 = [1, 2, 4]
print(list1)
```

```
[1 2 4]
[1, 2, 4]
```

Other ways

- `np.zeros()`, `np.ones()`
- For multidimensional arrays, pass a tuple

```
print(np.zeros(10))
print(np.ones((2, 5)))
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

Other ways

- `np.arange()` (similar to `range()`)

```
print(np.arange(5))
```

```
[0 1 2 3 4]
```

Shape, dimension, and dtype

```
arr5 = np.array([[1, 2, 3], [3, 4, 5]])
print(arr5.ndim)
print(arr5.shape)
print(arr5.dtype) # different than type()
```

```
2
(2, 3)
int64
```

Casting with `astype()` method

- E.g., from string to float

```
print(arr1)
arr1.astype(np.float64)
```

```
[1 2 4]
```

```
array([1., 2., 4.])
```

Casting with astype() method

- Be careful though

```
arr6 = np.array([1.2, 38.3])
print(arr6)
arr6 = arr6.astype(np.int32)
print(arr6)
```

```
[ 1.2 38.3]
[ 1 38]
```

Precision

```
l = 1.0
s = 0.000000000001

sum_32 = np.float32(l) + np.float32(s)
sum_64 = np.float64(l) + np.float64(s)

print("Sum in float32:", sum_32)
print("Sum in float64:", sum_64)
```

```
Sum in float32: 1.0
Sum in float64: 1.000000000001
```

Precision

```
arr_32 = np.array([1.5] * 1000000, dtype = np.float32)
arr_64 = np.array([1.5] * 1000000, dtype = np.float64)

print(arr_32.nbytes, "bytes")
print(arr_64.nbytes, "bytes")
```

```
4000000 bytes
8000000 bytes
```

Vectorization

- Can express batch operations without writing for loops
- This is called **vectorization**
  - Converts an operation into a form that works on multiple data elements simultaneously
- Arrays are treated like scalars

```
arr1 = np.array([[1., 2., 3.], [4., 1., 6.]])
arr2 = np.array([[0.5, 1., 1.], [2., 2., -1.]])
```

Vectorization

- Can express batch operations without writing for loops
- This is called **vectorization**
  - Converts an operation into a form that works on multiple data elements simultaneously
- Arrays are treated like scalars

Vectorization

```
print(arr1 * arr2)
print(1 / arr1)
print(arr1 / arr2)
print(arr1 >= arr2)
print(np.exp(arr2))
```

```
[[ 0.5  2.   3. ]
 [ 8.   2.  -6. ]]
[[1.         0.5        0.33333333]
 [0.25       1.         0.16666667]]
[[ 2.   2.   3. ]
 [ 2.   0.5 -6. ]]
[[ True  True  True]
 [ True False  True]]
[[1.64872127 2.71828183 2.71828183]
 [7.3890561  7.3890561  0.36787944]]
```

Loop, comprehension, and vectorized array arithmetic

- Setup

```
from timeit import default_timer as timer

size = 1000000  # one million elements

a_list = [i for i in range(size)]
b_list = [i for i in range(size, 0, -1)]

a_array = np.array(a_list)
b_array = np.array(b_list)
```

Loop, comprehension, and vectorized array arithmetic

- For loop

```
start = timer()
result_loop = []
for i in range(len(a_list)):
    result_loop.append(a_list[i] + b_list[i])
end = timer()
loop_time = end - start
print(f"Traditional for loop: {loop_time:.3f} seconds")
```

```
Traditional for loop: 0.162 seconds
```

Loop, comprehension, and vectorized array arithmetic

- List comprehension

```
start = timer()
result_list_comp = [a_list[i] + b_list[i] for i in range(len(a_list))]
end = timer()
list_comp_time = end - start
print(f"List comprehension: {list_comp_time:.3f} seconds")
```

```
List comprehension: 0.089 seconds
```

Loop, comprehension, and vectorized array arithmetic

- Vectorized array arithmetic

```
start = timer()
result_numpy = a_array + b_array
end = timer()
numpy_time = end - start
print(f"NumPy vectorized addition: {numpy_time:.3f} seconds")
```

```
NumPy vectorized addition: 0.002 seconds
```

Indexing and slicing

- One-dimensional arrays behave similarly to lists

```
print(arr4)
arr4[0]
arr4[-2]
arr4[1:3]
```

```
[1 None 4]
```

```
array([None, 4], dtype=object)
```

Indexing and slicing

- N-dimensional arrays

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d)
```

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

Indexing and slicing

```
print(arr2d)
print(arr2d[2][-1])
print(arr2d[2, -1])
```

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

Indexing and slicing

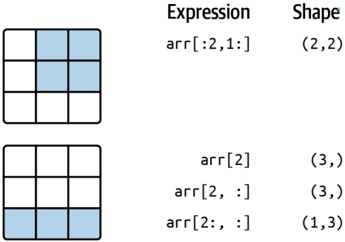


Figure 1: Python Tiobe

- There are many more approaches to indexing/slicing for n-dimensional arrays
- Recommend to check here

Pandas Series

- One-dimensional array-like object
- Contains a sequence of values of the same type
- Also contains a sequence of data labels, called *index*

```
height = pd.Series([163, 156, 177])
print(height)
```

```
0    163
1    156
2    177
dtype: int64
```

index

- Create a Series with an index identifying each data point with a string

```
height2 = pd.Series([163, 156, 177],
                    index = ['Lee', 'Kim', 'Jung'])
print(height2)
```

```
Lee     163
Kim     156
Jung    177
dtype: int64
```

- See the index

```
height.index
height2.index
height2.values
```

```
array([163, 156, 177])
```

index

- Change the index

```
height2.index = ['Choi', 'Park', 'Han']
print(height2)
```

```
Choi    163
Park    156
Han     177
dtype: int64
```

name, values, etc.

```
height2.name = 'name_height'
print(height2)
```

```
Choi    163
Park    156
Han     177
Name: name_height, dtype: int64
```

```
height2.index.name = 'name'
print(height2)
```

```
name
Choi    163
Park    156
Han     177
Name: name_height, dtype: int64
```

Indexing and slicing

```
print(height[0:2])
print(height[[0, 2]])
```

```
0    163
1    156
dtype: int64
0    163
2    177
dtype: int64
```

Indexing and slicing

```
print(height2)
height2['Choi':'Park'] # inclusive with string indices
height2[:2] # it also works with positions
```

```
name
Choi    163
Park    156
Han     177
Name: name_height, dtype: int64
```

```
name
Choi    163
Park    156
Name: name_height, dtype: int64
```

Indexing and slicing

- []-based indexing treat integers as labels if the index contains integers

```
obj = pd.Series([1, 2, 3], index = [2, 0, 1])
print(obj)
print(obj[[0, 1, 2]])
```

```
2    1
0    2
1    3
dtype: int64
0    2
1    3
2    1
dtype: int64
```

Indexing and slicing

- []-based indexing treat integers as labels if the index contains integers

```
obj = pd.Series([1, 2, 3], index = ['c', 'a', 'b'])
print(obj)
print(obj[[0, 1, 2]])
```

```
c    1
a    2
b    3
dtype: int64
c    1
a    2
b    3
dtype: int64
```

Using NumPy functions or NumPy-like operations

```
height[height > 160]
feet = height * 0.0328084
np.exp(pd.Series([1, 2, 3]))
```

```
0     2.718282
1     7.389056
2    20.085537
dtype: float64
```

- Note that index is preserved

Commonly used methods

```
grades = pd.Series([np.nan, 2, 'b', 'a+',  'a', 'f'])
grades.head(3) # similarly tail()
grades.value_counts(dropna = False)
```

```
NaN    1
2      1
b      1
a+     1
a      1
f      1
Name: count, dtype: int64
```

Commonly used methods

```
grades.isin(['a+', 'a', 'b+', 'b'])
grades.isnull() # the opposite is notnull()
```

```
0     True
1    False
2    False
3    False
4    False
5    False
dtype: bool
```

Commonly used methods

```
grades.dropna()
grades.fillna('f')
```

```
0     f
1     2
2     b
3    a+
4     a
5     f
dtype: object
```

Commonly used methods

```
grades.str.contains('a|a+')
grades.str.replace('a+', 'a')
```

```
0    NaN
1    NaN
2      b
3      a
4      a
5      f
dtype: object
```

Commonly used methods

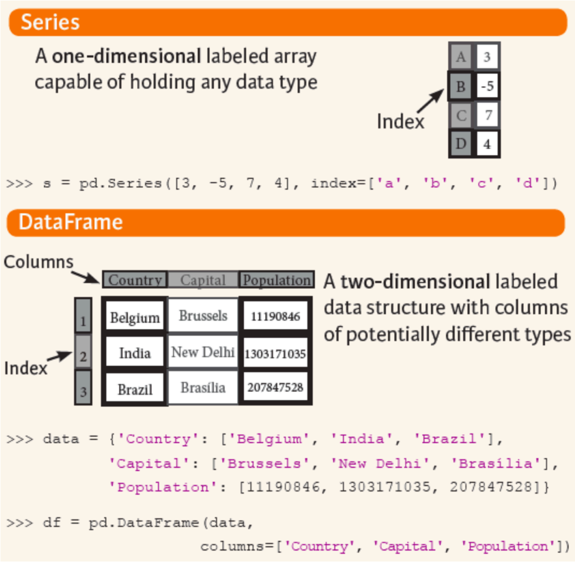
```
grades.str.upper()
grades.str.lower()
```

```
0    NaN
1    NaN
2      b
3     a+
4      a
5      f
dtype: object
```

DataFrame

- Represents a rectangular table of data
- Contains an ordered, named collection of columns
- Can be thought of as a dictionary of Series all sharing the same index

DataFrame


Series
A one-dimensional labeled array capable of holding any data type
Index
A 3
B -5
C 7
D 4
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
DataFrame
Columns
Country
Capital
Population
A two-dimensional labeled data structure with columns of potentially different types
Index
1
2
3
Belgium
Brussels
11190846
India
New Delhi
1303171035
Brazil
Brasília
207847528
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
'Capital': ['Brussels', 'New Delhi', 'Brasília'],
'Population': [11190846, 1303171035, 207847528]}
>>> df = pd.DataFrame(data,
columns=['Country', 'Capital', 'Population'])

DataFrame

- Many ways to construct a DataFrame
- One of the most common is from a dictionary of equal-length lists or NumPy arrays

```
df = pd.DataFrame(
  {'age': [23, 34, 23, 45, 67, 26],
  'income': [30000, 56000, None, 112000, 179000, 78000],
  'gender': ['F', None, 'M', 'F', 'F', 'M'],
  'married': [True, False, False, False, True, True]},
  index = ['Jessica', 'Jisoo', 'Peter', 'Susan', 'Rui', 'Alex'])
print(df)
```

```
         age    income gender  married
Jessica   23   30000.0      F     True
Jisoo     34   56000.0   None    False
Peter     23       NaN      M    False
Susan     45  112000.0      F    False
Rui       67  179000.0      F     True
Alex      26   78000.0      M     True
```

Indexing and slicing

- `loc` indexer works with labels (integer or string)
- Get rows with `loc`

```
print(df.loc['Alex'])
print(df.loc[['Jisoo', 'Alex']])
```

```
age              26
income      78000.0
gender            M
married        True
Name: Alex, dtype: object
       age   income gender  married
Jisoo   34  56000.0   None    False
Alex    26  78000.0      M     True
```

Indexing and slicing

- `loc` indexer works with labels (integer or string)
- Get rows with `loc`

```
print(df.loc['Jisoo':'Alex']) # inclusive
```

```
       age    income gender  married
Jisoo   34   56000.0   None    False
Peter   23       NaN      M    False
Susan   45  112000.0      F    False
Rui     67  179000.0      F     True
Alex    26   78000.0      M     True
```

Indexing and slicing

- What will this return?

```
df.loc[0]
```

Indexing and slicing

- What will this return?

```
df.loc[0] # won't work (with string labels)
```

Indexing and slicing

- Get rows and columns with `loc`

```
print(df.loc[['Alex', 'Jisoo'], ['gender', 'married']])
print(df.loc['Jisoo':'Rui', ['gender', 'married']]) # inclusive
```

```
      gender  married
Alex       M     True
Jisoo   None    False
      gender  married
Jisoo   None    False
Peter      M    False
Susan      F    False
Rui        F     True
```

Indexing and slicing

- `iloc` indexer works with positions (0, 1, 2, etc.)
- This is the case *regardless of labels*

```
print(df)
```

```
         age    income gender  married
Jessica   23   30000.0      F     True
Jisoo     34   56000.0   None    False
Peter     23       NaN      M    False
Susan     45  112000.0      F    False
Rui       67  179000.0      F     True
Alex      26   78000.0      M     True
```

```
print(df.iloc[1:3, 1:]) # non-inclusive
```

```
        income gender  married
Jisoo  56000.0   None    False
Peter      NaN      M    False
```

Indexing and slicing

- Let's assign numeric labels

```
df.index = range(1001, 1007) # let see this as survey id
df = df.reindex(range(1001, 1007))
print(df)
```

```
      age    income gender  married
1001   23   30000.0      F     True
1002   34   56000.0   None    False
1003   23       NaN      M    False
1004   45  112000.0      F    False
1005   67  179000.0      F     True
1006   26   78000.0      M     True
```

Indexing and slicing

- Get rows and columns with `iloc`, like this

```
print(df.iloc[0:3, :]) # non-inclusive
```

```
      age   income gender  married
1001   23  30000.0      F     True
1002   34  56000.0   None    False
1003   23      NaN      M    False
```

Indexing and slicing

- Subset columns and then rows

```
print(df[['age', 'income']].iloc[:3])
```

```
      age   income
1001   23  30000.0
1002   34  56000.0
1003   23      NaN
```

Indexing and slicing

- What will this return?

```
print(df.loc[1001:1002])
```

- What will this return?

```
print(df.loc[1001:1002])
```

```
      age   income gender  married
1001   23  30000.0      F     True
1002   34  56000.0   None    False
```

Indexing and slicing

- What will this return?

```
print(df.iloc[1001])
```

Indexing and slicing

- What will this return?

```
print(df.iloc[1001]) # won't work
```

Sorting

- By index

```
obj = pd.Series(np.arange(4),
                index=['d', 'a', 'b', 'c'])
frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
                     index=['y', 'x'],
                     columns=['d', 'a', 'b', 'c'])
print(obj)
print(frame)
```

```
d    0
a    1
b    2
c    3
dtype: int64
   d  a  b  c
y  0  1  2  3
x  4  5  6  7
```

Sorting

- By index

```
print(obj.sort_index())
```

```
a    1
b    2
c    3
d    0
dtype: int64
```

```
print(frame.sort_index())
```

```
   d  a  b  c
x  4  5  6  7
y  0  1  2  3
```

Sorting

- By index

```
print(frame.sort_index(axis = 'columns',
                       ascending = False))
```

```
   d  c  b  a
y  0  3  2  1
x  4  7  6  5
```

- By value

```
frame = pd.DataFrame({'b': [4, 7, -3, 2],
                      'a': [0, 1, 0, 1]})
print(frame)
```

```
   b  a
0  4  0
1  7  1
2 -3  0
3  2  1
```

Sorting

- By value

```
print(frame.sort_values(['a'], ascending = False))
```

```
   b  a
1  7  1
3  2  1
0  4  0
2 -3  0
```

Sorting

- By value

```
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 2]
print(frame)
```

```
   b  a
0  4  0
1  7  1
2 -3  0
3  2  2
```

```
print(frame.sort_values(['a']))
```

```
   b  a
0  4  0
2 -3  0
1  7  1
3  2  2
```

Sorting

- By value

```
print(frame)
```

```
   b  a
0  4  0
1  7  1
2 -3  0
3  2  2
```

```
print(frame.sort_values(['a', 'b'])) # note how values are
```

```
   b  a
2 -3  0
0  4  0
1  7  1
3  2  2
```

Handling missing data

- `isna()` or `isnull()`

```
ser = pd.Series([4.3, 3.3, np.nan, None, 0])
print(ser.isna())
```

```
0    False
1    False
2     True
3     True
4    False
dtype: bool
```

```
print(ser.isnull())
```

```
0    False
1    False
2     True
3     True
4    False
```

Handling missing data

- `dropna()`

```
ser = pd.Series([4.3, 3.3, np.nan, None, 0])
print(ser.dropna()) # re-assignment necessary
```

```
0    4.3
1    3.3
4    0.0
dtype: float64
```

Handling missing data

- `dropna()`

```
data = pd.DataFrame([[1., 6.5, 3.],
                     [1., np.nan, np.nan],
                     [np.nan, np.nan, np.nan],
                     [np.nan, 6.5, 3.]])
print(data)
```

```
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

```
print(data.dropna()) # any row with missingness
```

```
     0    1    2
0  1.0  6.5  3.0
```

Handling missing data

- dropna()

```
# rows where all values are missing
print(data.dropna(how = 'all'))
```

```
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0
```

Removing duplicates

- `duplicated()` return Boolean values

```
data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'] + ['one'],
                     'k2': [1, 1, 2, 3, 3, 4, 4, 3]})
print(data)
```

```
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
6  two   4
7  one   3
```

Removing duplicates

- `duplicated()` return Boolean values

```
print(data.duplicated()) # note rows 4 and 5
```

```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
7     True
dtype: bool
```

Removing duplicates

- `drop_duplicates()` remove duplicates

```
print(data.drop_duplicates())
```

```
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
```

Removing duplicates

- `drop_duplicates()` remove duplicates

```
data['k3'] = range(8)
print(data)
```

```
    k1  k2  k3
0  one   1   0
1  two   1   1
2  one   2   2
3  two   3   3
4  one   3   4
5  two   4   5
6  two   4   6
7  one   3   7
```

Removing duplicates

- `drop_duplicates()` remove duplicates

```
print(data.drop_duplicates(subset = 'k1', keep = 'first'))
```

```
    k1  k2  k3
0  one   1   0
1  two   1   1
```