STRUCTURED ARRAY

This section demonstrates the use of NumPy's structured arrays and record arrays, which provide efficient storage for compound, heterogeneous data.

NumPy data types

Character Description Example

'b' Byte np.dtype('b')

'i' Signed integer np.dtype('i4') == np.int32

'u' Unsigned integer np.dtype('u1') == np.uint8

'f' Floating point np.dtype('f8') == np.int64

'c' Complex floating point np.dtype('c16') == np.complex128

'S', 'a' string np.dtype('S5')

'U' Unicode string np.dtype('U') == np.str_ 'V' Raw data (void) np.dtype('V') == np.void

Consider if we have several categories of data on a number of people (say, name, age, and weight), and we'd like to store these values for use in a Python program. It would be possible to store these in three separate arrays.

name = ['Alice', 'Bob', 'Cathy', 'Doug'] age = [25, 45, 37, 19]

weight = [55.0, 85.5, 68.0, 61.5]

Creating structured array

NumPy can handle this through structured arrays, which are arrays with compound data types. create a structured array using a compound data type specification as follows.

data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),

'formats':('U10', 'i4', 'f8')})

print(data.dtype)

[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')] U10 - Unicode string of maximum length 10 i4 - 4-byte (i.e., 32 bit) integer

f8 - 8-byte (i.e., 64 bit) float

Now we can fill the array with our lists of values

data['name'] = name data['age'] = age data['weight'] = weight print(data)

[('Alice', 25, 55.0) ('Bob', 45, 85.5) ('Cathy', 37, 68.0)('Doug', 19, 61.5)]

Refer values through index or name

The handy thing with structured arrays is that you can now refer to values either by index or by name.

i. data['name']# by name

array(['Alice', 'Bob', 'Cathy', 'Doug'],dtype='<U10')

ii.data[0]# by index

('Alice', 25, 55.0)

Using Boolean masking

This allows to do some more sophisticated operations such as filtering on any fields.

data[data['age'] < 30]['name']

array(['Alice', 'Doug'],dtype='<U10')

Creating Structured Arrays

Dictionary method

np.dtype({'names':('name', 'age', 'weight'),

'formats':('U10', 'i4', 'f8')}) dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])

Numerical types can be specified with Python types

np.dtype({'names':('name', 'age', 'weight'),

'formats':((np.str_, 10), int, np.float32)}) dtype([('name', '<U10'), ('age', '<i8'), ('weight', '<f4')])

List of tuples

np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])

dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])

Specify the types alone

np.dtype('S10,i4,f8')

dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])

DATA MANIPULATION WITH PANDAS

Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a DataFrame. DataFrames are essentially multidimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data.

Pandas, and in particular its Series and DataFrame objects, builds on the NumPy array structure and provides efficient access to these sorts of ―data munging‖ tasks that occupy much of a data scientist's time.

Here we will focus on the mechanics of using Series, DataFrame, and related structures effectively.

Introducing Pandas Objects

Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices.

Pandas provide a host of useful tools, methods, and functionality on top of the basic data structures. Three fundamental Pandas data structures: the Series, DataFrame, and Index.

The Pandas Series Object

A Pandas Series is a one-dimensional array of indexed data. It can be created from a list or array as follows:

data = pd.Series([0.25, 0.5, 0.75, 1.0]) data

0 0.25

1 0.50

2 0.75

3 1.00

dtype: float64

•Finding values

The values are simply a familiar NumPy array

data.values

array([0.25, 0.5 , 0.75, 1.])

•Finding index

The index is an array-like object of type pd.Index

data.index

•Access by index

Like with a NumPy array, data can be accessed by the associated index via the familiar Python square-bracket notation

data[1] 0.5

data[1:3]

1 0.50

2 0.75

dtype: float64

The Pandas DataFrame Object

The fundamental structure in Pandas is the DataFrame. The DataFrame can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary.

DataFrame as a generalized NumPy array

A DataFrame is an analog of a two-dimensional array with both flexible row indices and flexible column names. Just as you might think of a two-dimensional array as an ordered sequence of aligned one-dimensional columns, you can think of a DataFrame as a sequence of aligned Series objects. Here, by ―aligned‖ we mean that they share the same index.

To demonstrate this, let's first construct a new Series listing the marks of subject2.

sub2={'sai':91,'ram':95,'kasim':89,'tamil':90}

Constructing DataFrame objects

A Pandas DataFrame can be constructed in a variety of ways. Here we'll give several examples.

•From a single Series object.

•From a list of dicts.

•From a dictionary of Series objects.

•From a two-dimensional NumPy array.

•From a NumPy structured array.

From a single Series object.

A DataFrame is a collection of Series objects, and a single column DataFrame can be constructed from a single Series.

sub1=pd.Series({'sai':90,'ram':85,'kasim':92,'tamil':89}) pd.DataFrame(sub1,columns=['DS'])

	DS
sai	90
ram	85
kasim	92
tamil	89

From a list of dicts.

Any list of dictionaries can be made into a DataFrame. We'll use a simple list comprehension to create some data

data = [{'a': i, 'b': 2 * i} for i in range(3)] pd.DataFrame(data)

a b 0 0 0

1 1 2

2 2 4

Even if some keys in the dictionary are missing, Pandas will fill them in with NaN (i.e.,“not a number”) values.

pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}]) a b c

0 1.0 2 NaN

1 NaN 3 4.0

From a dictionary of Series objects.

As we saw before, a DataFrame can be constructed from a dictionary of Series objects as well.

pd.DataFrame({'DS':sub1,'FDS':sub2})

	DS	FDS
sai	90	91
ram	85	95
kasim 92	89	
tamil 89	90	

From a two-dimensional NumPy array.

Given a two-dimensional array of data, we can create a DataFrame with any specified column and index names. If omitted, an integer index will be used for each.

pd.DataFrame(np.random.rand(3, 2), columns=['food', 'water'],

index=['a', 'b', 'c'])

food water

a 0.865257 0.213169

b 0.442759 0.108267

c 0.047110 0.905718

From a NumPy structured array.

A Pandas DataFrame operates much like a structured array, and can be created directly.

A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')]) A

array([(0, 0.0), (0, 0.0), (0, 0.0)],

dtype=[('A', '<i8'), ('B', '<f8')])

pd.DataFrame(A) A B

0 0 0.0

1 0 0.0

2 0 0.0