Panda Tutorial

**Series**

A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its index.

The simplest Series is formed from only an array of data:

In [11]: obj = pd.Series([4, 7, -5, 3])

obj.index # gives RangeIndex(start, stop, step)

obj.values # returns array

We can create a series with each data point signified by a index:

obj2 = pd.Series([1,2,3], index=[‘q’,’w’,’e’])

It’s kind of dictionary and same as ‘dict’ data structure of python. That’s why we can pass a dictionary to it.

sdata = { ‘ohio’:780, ‘colarado’:1200,’tennessee’:900 }

obj3 = pd.Series(sdata)

The resultant object is in sorting order. To disable sorting you can pass explicit keys list as in

states = [ ‘tenessee’, ‘ohio’, ‘colarado’ ]

pd.isnull()/notnull() functions can be used to check if argument passed is NULL or not. pd.Series also has these instances.

You can associate name with Series object and index as in:

obj3.name =’Population’

obj3.index.name = ‘State’

Series.value\_counts() returns the number of repetitions of all values.

Series.isin(<list>) returns list of bool with True for the element of Series which is contained in the list.

To replace an element with the another in Series, use:

Series.replace(old\_val, new\_val) # We can have list in place of single old\_val, then new\_val will also be a list

Index.get\_indexer method, which gives you an index array

from an array of possibly non-distinct values into another array of distinct values

**DataFrame**

A DataFrame is a rectangular table of data and contains an ordered collections of columns each of which can have a different value(i.e. int, str or Boolean)

We can pass a dict with an index and a series associated with a key to DataFrame constructor. All series should be of equal length. Each index refers to a column

frame = pd.DataFrame(data, columns=[<sequence\_of\_columns>)

To retrieve the header – frame.columns

Rows can be retrieved either by name or by ‘loc’ attribute. e.g.

frame.loc[‘three’]

We can access each element using row and column position by frame.iat[I, j]

You can retrieved aggregated data for both rows and columns by passing rows and columns to loc in that order. e.g frame.loc[‘three’, [ ‘colorado’, ‘nevada’][<Boolean\_condition>]

You can deploy ‘loc’ functionality using ‘iloc’. It’s just that you have to pass index numbers instead of whole strings.

A single column can be assigned a common value as:

frame[‘<col>’] = ‘<value>’

Obj.index() returns the indices or keys used to refer to records.

The del method can then be used to remove this column as in: del frame2['eastern']

*Essentials of Panda:*

> Reindexing: When you want to rearrange data according to new index.

obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c']

In [93]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [94]: obj2

Out[94]:

a -5.3, b 7.2, c 3.6, d 4.5, e NaN

You might want to fill certain values while reindexing. This can be done by passing method as key parameter to reindex function as in:

obj3.reindex(range(6), method='ffill')

You can pass either row, column or both to DataFrame constructor

frame = pd.DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'], columns=['Ohio', 'Texas', 'California'])

frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [101]: frame2

Ohio Texas California

a 0.0 1.0 2.0

b NaN NaN NaN

c 3.0 4.0 5.0

d 6.0 7.0 8.0

> **Dropping a row/column:**

You can drop a row or column from a DataFrame using:

obj.drop(<label\_either\_col\_or\_row>, axis=1 or 0) # Here 0 refers to indices and 1 refers to columns. It returns the copy of frame with requisite changes. To make changes in the current frame itself pass ->

inplace=True to drop function.

You can filter by column value as:

data[data['three'] > 5] # Here, ‘three’ is column name

> **Addition**

Addition of data frames will be like addition of elements present in 2-D array with non-common element tagged as NaN which can be filled to ‘0’ if required

df1.add(df2, fill\_value=0)

Following arithmetic operators ara available: add()/mul()/sub()/div()/pow() with r substitute of each

When we subtract list from an array. The subtraction operator is applied to all rows in array. This phenomenon is called broadcasting.

If you want to subtract a series from 2-D array column-wise pass axis=’column’

Some NumPy methods can work on DataFrame too:

np.abs(frame) # Change sign of all –ve numbers in DataFrame

func = lambda x: x.max()-x.min()

frame.apply(func)

This will work per column. If you rather want to apply this function row-wise, pass axis=’column’

> Sorting and Ranking:

series.sort\_index() sorts the series by indices. To sort by columns and by descending order:

series.sort\_index(axis=1, ascending=False)

Simply sort a series by .sort\_values() function.

To sort a frame by a given column, use frame.sort\_values(by=’<col\_name>’) function

DataFrame.idxmax/idmin returns the index with upper/lower values

**Files - Data loading and storing**

Parsing functions available in panda:

read\_csv()/table()/excel()/html()/json()/clipboard()/sql()

A .csv can be read in two ways:

pd.read\_table('examples/ex1.csv', sep=',')

df = pd.read\_csv('examples/ex1.csv')

nFollowing are the options we can pass to read\_csv() function of panda.

header=None # To make the panda not consider 1st row as header

names=[‘a’, ‘b’, … ] etc. # To designate a column by name rather than by indices 0,1,2… etc.

skiprows=[] # Feed in the row numbers to be skipped. The equivalent function on reverse side is skip\_footer()

nrows=int # Specifies the number of rows to be picked from file

na\_values=[] # The input in list would be dict that mentions sentinels which should be marked NaN for each column. E.g.:

sentinels = {'message': ['foo', 'NA'], 'something': ['two']} # ‘foo’ and ‘NA’ are modified to NaN in column ‘message’

na\_values=sentinels

coverters # Can be used to specify a function to be implemented on a given column

**Writing to .csv files**

To do this, we’ll use DataFrame.to\_csv(“<csv\_file>”) function.

Following arguments can be passed to to\_csv() function:

sep=’<char>’ # To set a delimiter

na\_rep = ‘NULL’ # Missing values appear as empty strings in output.

index=False # To suppress writing of row index and

header= False # To suppress writing of column names

columns=[] # To mention only subsets of columns we want to write

Dialect in csv refers to custom conventions used in user’s specialized .csv files such as: line terminator, delimiter etc

Example:

class my\_dialect(csv.Dialect):

lineterminator = '\n'

delimiter = ';'

skipinitialspace=True

quotechar = '"'

quoting = csv.QUOTE\_MINIMAL

reader = csv.reader(f, dialect=my\_dialect)

read\_table() # Used to read data from a table

We can have variable amount of white space in between strings. To escape that, you can pass regular expressions as:

result = pd.read\_table('examples/ex3.txt', sep='\s+')

**JSON:**

import json

result = json.loads(obj) # convert string ‘obj’ ( in form of json ) to a JSON object

asjson = json.dumps(result) # convert result to DataFrame – ‘asjson’

Panda supports reading below file formats:

|  |  |  |
| --- | --- | --- |
| Format | Library | Common function |
| SQLite | in-built – sqlite3 | con = sqlite3.connect('mydata.sqlite')  con.execute(query)  con.commit() |
| XML | lxml, beautifulsoup4 and html5lib | read\_html()  objectify.parse(open(path)) |
| Binary Data | pandas | pd.read\_pickle()  frame.to\_pickle() |
| Microsoft Excel | xlrd and openpyxl | pd.read\_excel()  writer = pd.ExcelWriter('examples/ex2.xlsx')  frame.to\_excel(writer, 'Sheet1')  writer.save() |

Data Filtering:

Filtering out missing (Not Available) data

data = pd.Series([1,2,None,3,None,4,5])

data.dropna() # This will remove all entries which are either NA or None. It will drop the complete row even if that row contains one ‘NA’ entry. To prevent this, pass how=’all’ to function:

data.dropna(how=’all’)

Pass axis=1 to drop by column.

Also, rather than dropping all rows, you can fill those NA/None with certain predetermined values such as 0.0 using

data.fillna(0)

you can also specify values column-specific

data.fillna({1:0.5, 2:0})

Pass inPlace=True to change the DataFrame in-place.

Finding duplicate rows can be done using DataFrame.duplicated() function. To render the result, use drop\_duplicates() function.

data.drop\_duplicates([]) # We can pass column names in the list. The later repetitive entries are deleted. To keep the last ones, pass keep=’last’ to the function:

data.drop\_duplicates(['k1', 'k2'], keep='last')

Data Transformation

Suppose you want to add another column based on values of other column. In that case, you can just do

data['new\_col'] = data[‘reference’].map(<dict\_of\_mapping>)

Like a Series, the axis indexes have a map method:

In [67]: transform = lambda x: x[:4].upper()  
In [68]: data.index.map(transform)  
Out[68]: Index(['OHIO', 'COLO', 'NEW '], dtype='object')

Another method called ‘rename’ can be used by passing index=[], columns=[]

Discretization and Binning

AnoSometimes continuous data needs to be discretized or divided into chunks. e.g A series of ages needs to be furcated into pieces of a given age bracket.

ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]

Let’s divide these into bins of 18 to 25, 26 to 35 etc.

do so, you have to use cut, a function in pandas:

bins = [18, 25, 35, 60, 100]

cats = pd.cut(ages, bins)

cats:

[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35,

60], (35, 60], (25, 35]]

Length: 12

Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]