Introduction to Pandas & Data Structures

August 28, 2022

1 Introduction to Pandas

Pandas is an open source library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Today, pandas is actively supported by a community of like-minded individuals around the world who contribute their valuable time and energy to help make open source pandas possible. We will learn to use pandas for data analysis. If you have never used this library, you can think about pandas as an extremely powerful version of Excel and with lot more features

2 pandas Data Structures

Series and **DataFrame** are two workhorse data structures in pandas.
Lets talk about series first:

2.1 Series

Series is a one-dimensional array-like object, which contains values and an array of labels, associated with the values. Series can be indexed using labels. (Series is similar to NumPy array – actually, it is built on top of the NumPy array object) Series can hold any arbitrary Python object. Let's get hands-on and learn the concepts of Series with examples:

[1]:
```python
# first thing first, we need to import NumPy and pandas
# np and pd are alias for NumPy and pandas

import numpy as np
import pandas as pd

# just to check ther versions we are using
print('numpy version:', np.__version__)
print('pandas version:', pd.__version__)
```

```
numpy version: 1.21.5
pandas version: 1.4.2
```

We can create a Series using list, numpy array, or dictionary Let's create these objects and convert them into panda's Series! Series using lists Lets create a Python lists, one containing labels and another with data

[2]:
```python
my_labels = ['x', 'y', 'z']
my_data = [100, 200, 300]

print(my_labels)
print(my_data)
```

```
['x', 'y', 'z']
[100, 200, 300]
```

So, we have two Python's list objects,

- `my_labels - a list of strings,`
- `my_data - a list of numbers`

We can use pd.Series (with capital S) to convert the Python's list object to pandas Series.

[3]:
```python
# Converting my_data (Python list) to Series (pandas series)
result = pd.Series(data=my_data)
print(result)
```

```
0    100
1    200
2    300
dtype: int64
```

Column "0 1 2" is automatically generated index for the elements in series with data "100 200 300". We can specify index values and grab the respective data/values using these indexes. Let's pass my_labels to the Series as index.

[4]:
```python
result = pd.Series(data=my_data, index=my_labels)
print(result)
```

```
x    100
y    200
z    300
dtype: int64
```

2.2 Series using NumPy arrays

[5]:
```python
# Let's create NumPy array from my_data and then Series from that array
my_array = np.array(my_data)  # creating numpy's array from list
result = pd.Series(data=my_array)  # creating series from numpy's array
print(result)
```

```
0    100
1    200
2    300
dtype: int32
```

Notice, we got the index column "012" again, let's pass our own index values!

[6]:
```
result = pd.Series(data=my_data, index=my_labels)
print(result)
# pd.Series(my_array, my_labels) # data and index are in order
```

```
x    100
y    200
z    300
dtype: int64
```

2.3 Series using dictionary

[7]:
```
# Let's create a dictionary my_dict
my_dict = {'x': 100, 'y': 200, 'z': 300}  # creating a dictionary my_dict
result = pd.Series(data=my_dict)  # creating series from dictionary
print(result)
```

```
x    100
y    200
z    300
dtype: int64
```

Notice the difference here, if we pass a dictionary to Series, pandas will take the keys as index/labels and values as data.

2.4 Grabbing data from Series

Indexes are the key thing to understand in Series. Pandas use these indexes (numbers or names) for fast information retrieval. (Index works just like a hash table or a dictionary). To understand the concepts, Let's create three Series, ser1, ser2, ser3 from dictionaries with some random data

[8]:
```
# Creating three dictionaries dict_1, dict_2, dict_3
dict_1 = {'Toronto': 500, 'Calgary': 200, 'Vancouver': 300, 'Montreal': 700}
dict_2 = {'Calgary': 200, 'Vancouver': 300, 'Montreal': 700}
dict_3 = {'Calgary': 200, 'Vancouver': 300, 'Montreal': 700, 'Jasper': 1000}
```

[9]:
```
# Creating pandas series from the dictionaries
ser1 = pd.Series(dict_1)
ser2 = pd.Series(dict_2)
ser3 = pd.Series(dict_3)
```

[10]:
```
print(ser1)
```

```
Toronto      500
Calgary      200
Vancouver    300
Montreal     700
dtype: int64
```

[11]:
```
# Grabbing information for series is very much similar to dictionary. Simply␣
↪pass,!the index and it will return the value!
print(ser1['Calgary'])  # its case sensitive "calgary" is not the same as␣
↪"Calgary"
```

```
200
```

[12]:
```
ser4 = ser1 + ser2  # adding series and assigning/passing results to a new␣
↪variable,!ser4
print(ser4)
```

```
Calgary       400.0
Montreal     1400.0
Toronto         NaN
Vancouver     600.0
dtype: float64
```

2.5 Builtin Function

Below are some commonly used built-in functions and attributes for series during the data processing. **isnull()** detect missing data

[13]:
```
# pd.isnull(ser4) is same as ser4.isnull()
print(ser4.isnull())
# shift+tab, its Type is method
```

```
Calgary      False
Montreal     False
Toronto       True
Vancouver    False
dtype: bool
```

[14]:
```
# notnull() * Detect existing (non-missing) values.
#pd.notnull(ser5) is same as ser5.notnull()
print(ser4.notnull())
```

```
Calgary       True
Montreal      True
Toronto      False
Vancouver     True
dtype: bool
```

head(), tail()
To view a small sample of a Series or DataFrame (we will learn DataFrame in the next lecture) object, use the **head()** and **tail()** methods. The default number of elements to display is five, but you may pass a custom number.

[15]:
```
print(ser1.head(1))  # head(1) will return the first row only
```

```
Toronto    500
dtype: int64
```

[16]:
```python
print(ser1.tail(1))  # tail(1) will return the last row only
```

```
Montreal    700
dtype: int64
```

[17]:
```python
# axes Returns list of the row axis labels
# row axis labels (index) list can be obtained
print(ser1.axes)
```

```
[Index(['Toronto', 'Calgary', 'Vancouver', 'Montreal'], dtype='object')]
```

values returns list of values/data

[18]:
```python
# returns the values/data
print(ser1.values)
```

```
[500 200 300 700]
```

size Returns the number of elements in the series
empty True if the series in empty

[19]:
```python
# True for empty series
print(ser1.empty)
```

```
False
```

[20]:
```python
print(ser1.size)
```

```
4
```

2.6 DataFrame

A very simple way to think about the DataFrame is, "bunch of Series together such as they share the same index". * A DataFrams is a rectangular table of data that contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc). DataFrame has both row & column index; it can be thought of as a dictionary of Series all sharing the same index (any row or column). Let's learn DataFrame with examples:

[21]:
```python
# Let's create two labels or indexes: * index: for rows 'r1 to r10' * columns:␣
↪for columns 'c1 to c10'
# Using split() for revision!

import pandas as pd
import numpy as np

index = 'r1 r2 r3 r4 r5 r6 r7 r8 r9 r10'.split()
columns = 'c1 c2 c3 c4 c5 c6 c7 c8 c9 c10'.split()

print(index)
print(columns)
```

```
['r1', 'r2', 'r3', 'r4', 'r5', 'r6', 'r7', 'r8', 'r9', 'r10']
['c1', 'c2', 'c3', 'c4', 'c5', 'c6', 'c7', 'c8', 'c9', 'c10']
```

[22]:
```
# Let's start with a simple example, using arange() and reshape() together to
↪create a 2D array (matrix).

array_2d = np.arange(0, 100).reshape(10, 10)  # creating a 2D array "array_2d"

print(array_2d)
```

```
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47 48 49]
 [50 51 52 53 54 55 56 57 58 59]
 [60 61 62 63 64 65 66 67 68 69]
 [70 71 72 73 74 75 76 77 78 79]
 [80 81 82 83 84 85 86 87 88 89]
 [90 91 92 93 94 95 96 97 98 99]]
```

[23]:
```
# Now, let's create our first DataFrame using index, columns and array_2d!
df = pd.DataFrame(data=array_2d, index=index, columns=columns)
print(df)
```

```
     c1  c2  c3  c4  c5  c6  c7  c8  c9  c10
r1    0   1   2   3   4   5   6   7   8    9
r2   10  11  12  13  14  15  16  17  18   19
r3   20  21  22  23  24  25  26  27  28   29
r4   30  31  32  33  34  35  36  37  38   39
r5   40  41  42  43  44  45  46  47  48   49
r6   50  51  52  53  54  55  56  57  58   59
r7   60  61  62  63  64  65  66  67  68   69
r8   70  71  72  73  74  75  76  77  78   79
r9   80  81  82  83  84  85  86  87  88   89
r10  90  91  92  93  94  95  96  97  98   99
```

[24]:
```
df
```

[24]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10
r1	0	1	2	3	4	5	6	7	8	9
r2	10	11	12	13	14	15	16	17	18	19
r3	20	21	22	23	24	25	26	27	28	29
r4	30	31	32	33	34	35	36	37	38	39
r5	40	41	42	43	44	45	46	47	48	49
r6	50	51	52	53	54	55	56	57	58	59
r7	60	61	62	63	64	65	66	67	68	69
r8	70	71	72	73	74	75	76	77	78	79

```
r9    80  81  82  83  84  85  86  87  88   89
r10   90  91  92  93  94  95  96  97  98   99
```

df is our first dataframe. We have columns, c1 to c10, and their corresponding rows, r1 to r10. Each column is actually a pandas series, sharing a common index, which is the row labels. Now, we can play with this dataframe df to learn how to Grab data that we need, which is the most important concept we want to learn to move one in this course!

Grabbing Columns from dataframe Just pass the name of the required column in square brackets!

[25]:
```python
# Grabbing a single column
print(df['c1'])
```

```
r1      0
r2     10
r3     20
r4     30
r5     40
r6     50
r7     60
r8     70
r9     80
r10    90
Name: c1, dtype: int32
```

[26]:
```python
# We can grab more than one column, simply pass the list of columns you need!
df[['c1', 'c10']]
```

[26]:
```
     c1  c10
r1    0    9
r2   10   19
r3   20   29
r4   30   39
r5   40   49
r6   50   59
r7   60   69
r8   70   79
r9   80   89
r10  90   99
```

2.7 Adding new column to dataframe

pandas dataframes are very handy, Let's add a column 'new into our dataframe df by adding any two existing columns using simple "+" operator!

[27]:
```python
df['cnew'] = df['c1'] + df['c2']  # adding a column "new" which is sum of "c1"␣
 ↪and "c2"
```

[28]:
```
df
```

[28]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	cnew
r1	0	1	2	3	4	5	6	7	8	9	1
r2	10	11	12	13	14	15	16	17	18	19	21
r3	20	21	22	23	24	25	26	27	28	29	41
r4	30	31	32	33	34	35	36	37	38	39	61
r5	40	41	42	43	44	45	46	47	48	49	81
r6	50	51	52	53	54	55	56	57	58	59	101
r7	60	61	62	63	64	65	66	67	68	69	121
r8	70	71	72	73	74	75	76	77	78	79	141
r9	80	81	82	83	84	85	86	87	88	89	161
r10	90	91	92	93	94	95	96	97	98	99	181

2.8 Adding new row to dataframe

[29]:
```
row = np.random.randint(1,100, 11)

df.loc[len(df.index)] = row
```

[30]:
```
df
```

[30]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	cnew
r1	0	1	2	3	4	5	6	7	8	9	1
r2	10	11	12	13	14	15	16	17	18	19	21
r3	20	21	22	23	24	25	26	27	28	29	41
r4	30	31	32	33	34	35	36	37	38	39	61
r5	40	41	42	43	44	45	46	47	48	49	81
r6	50	51	52	53	54	55	56	57	58	59	101
r7	60	61	62	63	64	65	66	67	68	69	121
r8	70	71	72	73	74	75	76	77	78	79	141
r9	80	81	82	83	84	85	86	87	88	89	161
r10	90	91	92	93	94	95	96	97	98	99	181
10	26	76	92	81	2	93	51	35	44	9	23

2.9 Deleting column from dataframe

drop() We can delete any column form a dataframe using drop() method. Few important parameters that we need to consider: * label: column name that we need to pass, if we need to drop more than one columns, it must be a list of column names. * axis: default value is 0 which refers to row, to drop a column, we need to pass axis = 1 * inplace: default is False, we need to pass True for permanent delete. Inplace make sure that we don't delete column by mistake. If we don't pass this parameter, the column will not be dropped from the dataframe.

[31]:
```
# So, we have 10 rows and 11 columns in our dataframe df, "new" is the 11th one␣
↪that we have added.
# Let's delete this column.
```

```
df.drop(['cnew'], axis=1, inplace=True)  # If we don't pass inplce =
 ↪True,the,change will not be permanent

print(df)
```

```
     c1  c2  c3  c4  c5  c6  c7  c8  c9  c10
r1    0   1   2   3   4   5   6   7   8    9
r2   10  11  12  13  14  15  16  17  18   19
r3   20  21  22  23  24  25  26  27  28   29
r4   30  31  32  33  34  35  36  37  38   39
r5   40  41  42  43  44  45  46  47  48   49
r6   50  51  52  53  54  55  56  57  58   59
r7   60  61  62  63  64  65  66  67  68   69
r8   70  71  72  73  74  75  76  77  78   79
r9   80  81  82  83  84  85  86  87  88   89
r10  90  91  92  93  94  95  96  97  98   99
10   26  76  92  81   2  93  51  35  44    9
```

2.10 Grabbing Rows from dataframe

We can retrieve a row by its name or position with loc and iloc. * loc: Access a rows by label(s). * iloc: Using row's index location.

[32]:
```
# using loc, this will return rows r2 and r3, notice the list [r2, r3] in,
 ↪square brackets
df.loc[['r2', 'r3']]
```

[32]:
```
    c1  c2  c3  c4  c5  c6  c7  c8  c9  c10
r2  10  11  12  13  14  15  16  17  18   19
r3  20  21  22  23  24  25  26  27  28   29
```

[33]:
```
# Uisng iloc, this will again return rows r2 and r3, but here our selection in,
 ↪index based!
df.iloc[[1, 2]]  # remember, index starts with 0
```

[33]:
```
    c1  c2  c3  c4  c5  c6  c7  c8  c9  c10
r2  10  11  12  13  14  15  16  17  18   19
r3  20  21  22  23  24  25  26  27  28   29
```

2.11 Grabbing a single element form a dataframe

[34]:
```
# We need to tell the location of the element, [row, col]
# df.loc(req_row, req_col) -- pass row, col for the element!
print(df.loc['r2', 'c1'])
```

```
10
```

```
[35]: # another element, say 10 which is at [r2,c10]
print(df.loc['r2', 'c10'])
```

```
19
```

Grabbing sub-set of a dataframe We can grab a sub-set by passing list of required rows and list of required columns

```
[36]: # for a sub-set, pass the list
df.loc[['r1', 'r2'], ['c1', 'c2']]
```

```
[36]:     c1  c2
r1   0   1
r2  10  11
```

```
[37]: # another example - random columns and rows in the list
df.loc[['r2', 'r5'], ['c3', 'c4']]
```

```
[37]:     c3  c4
r2  12  13
r5  42  43
```

2.12 Conditional Selection or masking

pandas got excellent features, we can do a conditional selection. For example, all the values that are greater than some value, e.g. greater that 5 in the case below!

```
[38]: # We can do a conditional selection as well
df > 5
# df!=0 # try this yourself
# df=0 # try this yourself
```

```
[38]:
        c1     c2     c3     c4     c5     c6    c7    c8    c9   c10
r1   False  False  False  False  False  False  True  True  True  True
r2    True   True   True   True   True   True  True  True  True  True
r3    True   True   True   True   True   True  True  True  True  True
r4    True   True   True   True   True   True  True  True  True  True
r5    True   True   True   True   True   True  True  True  True  True
r6    True   True   True   True   True   True  True  True  True  True
r7    True   True   True   True   True   True  True  True  True  True
r8    True   True   True   True   True   True  True  True  True  True
r9    True   True   True   True   True   True  True  True  True  True
r10   True   True   True   True   True   True  True  True  True  True
10    True   True   True   True  False   True  True  True  True  True
```

```
[39]: # Return Divisible by 2 or even
bool_mask = df % 2 == 0  # creating mask for the required condition
df[bool_mask]  # passing mask to get the required results
```

```
# df[df % 2 == 0] # Similar to the above 2 lines of code
```

[39]:

```
      c1    c2  c3  c4  c5  c6    c7  c8  c9  c10
r1     0   NaN   2 NaN   4 NaN   6.0 NaN   8  NaN
r2    10   NaN  12 NaN  14 NaN  16.0 NaN  18  NaN
r3    20   NaN  22 NaN  24 NaN  26.0 NaN  28  NaN
r4    30   NaN  32 NaN  34 NaN  36.0 NaN  38  NaN
r5    40   NaN  42 NaN  44 NaN  46.0 NaN  48  NaN
r6    50   NaN  52 NaN  54 NaN  56.0 NaN  58  NaN
r7    60   NaN  62 NaN  64 NaN  66.0 NaN  68  NaN
r8    70   NaN  72 NaN  74 NaN  76.0 NaN  78  NaN
r9    80   NaN  82 NaN  84 NaN  86.0 NaN  88  NaN
r10   90   NaN  92 NaN  94 NaN  96.0 NaN  98  NaN
10    26  76.0  92 NaN   2 NaN   NaN NaN  44  NaN
```

2.12.1 info()

Provides a concise summary of the DataFrame. This is a very useful method.

[40]:

```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 11 entries, r1 to 10
Data columns (total 10 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   c1      11 non-null     int32
 1   c2      11 non-null     int32
 2   c3      11 non-null     int32
 3   c4      11 non-null     int32
 4   c5      11 non-null     int32
 5   c6      11 non-null     int32
 6   c7      11 non-null     int32
 7   c8      11 non-null     int32
 8   c9      11 non-null     int32
 9   c10     11 non-null     int32
dtypes: int32(10)
memory usage: 828.0+ bytes
None
```

2.12.2 describe()

Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

[41]:

```
df.describe()
```

[41]:

	c1	c2	c3	c4	c5	c6	\
count	11.000000	11.000000	11.000000	11.000000	11.000000	11.000000	
mean	43.272727	48.727273	51.090909	51.000000	44.727273	53.909091	
std	29.288533	30.113422	31.766191	30.397368	32.028396	31.513345	
min	0.000000	1.000000	2.000000	3.000000	2.000000	5.000000	
25%	23.000000	26.000000	27.000000	28.000000	19.000000	30.000000	
50%	40.000000	51.000000	52.000000	53.000000	44.000000	55.000000	
75%	65.000000	73.500000	77.000000	77.000000	69.000000	80.000000	
max	90.000000	91.000000	92.000000	93.000000	94.000000	95.000000	

	c7	c8	c9	c10
count	11.000000	11.000000	11.000000	11.000000
mean	51.000000	50.454545	52.181818	49.909091
std	28.722813	29.176578	28.850713	31.766191
min	6.000000	7.000000	8.000000	9.000000
25%	31.000000	31.000000	33.000000	24.000000
50%	51.000000	47.000000	48.000000	49.000000
75%	71.000000	72.000000	73.000000	74.000000
max	96.000000	97.000000	98.000000	99.000000

2.12.3 Assignment is Coming!

[ ]:

```
```