ITECH2302 Big Data Management

Foundation of Python programming
Pandas, Series and DataFrames

Big data applications often need more flexible collections that support mixed data types, custom indexing, missing data, and data that's not structured consistently, etc. For dealing with the applications, in this week, we'll learn pandas array-like one-dimensional Series and two-dimensional DataFrames.

Task 1: Reshaping and transposing

NumPy provides various ways to reshape arrays. In this Task, we learn how to change an array's dimension, transpose an array's rows and columns and add rows or columns.

1. reshape and resize

The array methods **reshape** and **resize** both enable us to change an array's dimensions. Method **reshape** returns a view(shallow copy) of the original array with the new dimensions. It does not modify the original array.

Type the following codes:

```
In [1]: import numpy as np

In [2]: grades = np.array([[87, 96, 70], [100, 87, 90]])

In [3]: grades
Out[3]:
array([[ 87,  96,  70],
       [100,  87,  90]])

In [4]: grades.reshape(1, 6)
Out[4]: array([[ 87,  96,  70, 100,  87,  90]])

In [5]: grades
Out[5]:
array([[ 87,  96,  70],
       [100,  87,  90]])
```

In the codes above, line 4 calls **reshape** to create a 1 row by 6 columns one-dimensional array, but the original array grades is still a 2 rows by 3 columns array.

However, method **resize** modifies the original array's shape.

```
In [6]: grades.resize(1, 6)

In [7]: grades
Out[7]: array([[ 87,  96,  70, 100,  87,  90]])
```

In the codes above, calling **resize**, the original array grades becomes a 1 row by 6 columns one-dimensional array.

2. flatten and ravel

We can take a multidimensional array and flatten it into a single dimension with the methods **flatten** and **ravel**. Method **flatten** deep copies the original array's data.

```
In [8]: grades = np.array([[87, 96, 70], [100, 87, 90]])

In [9]: grades
Out[9]:
array([[ 87,  96,  70],
       [100,  87,  90]])
In [13]: flattened[0] = 100

In [14]: flattened
Out[14]: array([100,  96,  70, 100,  87,  90])

In [15]: grades
Out[15]:
array([[ 87,  96,  70],
       [100,  87,  90]])
```

Please note that grades and flattened do not share the data. For example, let's modify an element of flattened, than display both array:

Method **ravel** produces a view of the original array, which shares the grades array's data.

```
In [16]: raveled = grades.ravel()

In [17]: raveled
Out[17]: array([ 87,  96,  70, 100,  87,  90])

In [18]: grades
Out[18]:
array([[ 87,  96,  70],
       [100,  87,  90]])
```

Please note that grades and reveled share the same data. For example, let's modify an element of raveled, then display both array.

```
In [19]: raveled[0] = 100

In [20]: raveled
Out[20]: array([100,  96,  70, 100,  87,  90])

In [21]: grades
Out[21]:
array([[100,  96,  70],
       [100,  87,  90]])
```

3. Transposing rows and columns

We can transpose an array's rows and columns, that is, rows become the columns and the columns become the rows. The **T** attribute returns a transposed view(shallow copy) of the array. For example, the original grades array represents two students' grades(rows) on three exams(columns).

Type the following codes:

```
In [22]: grades.T
Out[22]:
array([[100, 100],
       [ 96,  87],
       [ 70,  90]])
```

Transposing does not modify the original array.

4.

```
In [23]: grades
Out[23]:
array([[100,  96,  70],
       [100,  87,  90]])
```

Horizontal and vertical stacking

We can combine arrays by adding **more columns or more rows. This operation is called as horizontal stacking and vertical stacking**. For example, we create a new array grades2 that represents three additional exam grades for the two students in the grades array.

```
In [6]: grades2 =np.array([[94, 77, 90], [100, 81, 82]])
```

We can use NumPy's **hstack** (horizontal stack) function to combine grades and grades 2 by passing a tuple containing the array to combine. The extra parentheses are required because **hstack** expects one argument.

Type the following codes:

```
In [6]: grades2 =np.array([[94, 77, 90], [100, 81, 82]])

In [7]: np.hstack((grades, grades2))
Out[7]:
array([[100,  96,  70,  94,  77,  90],
       [100,  87,  90, 100,  81,  82]])
```

If we assume that grades2 represents two more students' grades on three exams, we can combine grades and grades2 with NumPy's **vstack** (vertical stack) function.

Type the following codes:

```
In [8]: np.vstack((grades, grades2))
Out[8]:
array([[100,  96,  70],
       [100,  87,  90],
       [ 94,  77,  90],
       [100,  81,  82]])
```

Task 2 Pandas series

Pandas is the most popular library for supporting mixed data types, missing data and data that's not structured consistently, etc, for big data applications. It provides two key collections, **Series** for one-dimensional collections and **DataFrames** for two-dimensional collection.

A Series is an enhanced one-dimensional array. Whereas arrays use only zero-based integer indices, Series support custom indexing, including even non-integer indices like strings. Series may have missing data, and many series operations ignore missing data by default.

1. Creating a Series with default indices

A Series has integer indices numbered sequentially from 0 by default.

Type the following codes:

```
In [9]: import pandas as pd

In [10]: grades = pd.Series([87, 100, 94])
```

Pandas displays a Series in two-column format with the indices left aligned in the left column and the values right aligned in the right column, and shows data type(**dtype**) of the underlying array's elements.

Type the following codes:

```
In [11]: grades
Out[11]:
0     87
1    100
2     94
dtype: int64
```

2. Creating a Series with all elements having the same value

We can create a series of elements that all have the same value.

Type the following codes:

```
In [12]: pd.Series(98.6, range(3))
Out[12]:
0    98.6
1    98.6
2    98.6
dtype: float64
```

In the codes above, the second argument is a one-dimensional iterable object(such as a list, an array or a range) containing the Series' indices. The number of indices determines the number of elements.

We can access a Series's elements by via **square brackets** containing an index.

```
In [13]: grades[0]
Out[13]: 87
```

3. Producing descriptive statistics for a series

Series provides many methods for common tasks including producing various descriptive statistics, such as **count, mean, min, max** and **std**(standard deviation).

Type the following codes:

```
In [14]: grades.count()
Out[14]: 3

In [15]: grades.mean()
Out[15]: 93.66666666666667

In [16]: grades.min()
Out[16]: 87

In [17]: grades.max()
Out[17]: 100

In [18]: grades.std()
Out[18]: 6.506407098647712
```

We can call Series method **describe** to produce all these stats and more.

Type the following codes:

```
In [19]: grades.describe()
Out[19]:
count      3.000000
mean      93.666667
std        6.506407
min       87.000000
25%       90.500000
50%       94.000000
75%       97.000000
max      100.000000
dtype: float64
```

In the codes above, 25%, 50% and 75% are quartiles. For example, 50% represents the median of the sorted values; 25% for the median of the first half of the sorted values and 75% for median of the second half of the sorted values. In the example above, we have three values in the Series, so 25% quartile is the average of 87 and 94 and 75% quartile is the average of 94 and 100.

4. Creating a Series with custom indices

We can specify custom indices with the **index** keyword argument.

Type the following codes:

```
In [20]: grades = pd.Series([87, 100, 94], index = ['Wally', 'Eva', 'Sam'])

In [21]: grades
Out[21]:
Wally     87
Eva      100
Sam       94
dtype: int64
```

In the codes above, we use string indices, but we can also use other immutable types such as integers not beginning at 0 and non-consecutive integers, etc.

5. Dictionary initializers

If we initialize a series with a dictionary , its keys become the series' indices, and its values become the series' element values.

Type the following codes:

```
In [22]: grades = pd.Series({'Wally': 87, 'Eva':  100, 'Sam': 94})

In [23]: grades
Out[23]:
Wally     87
Eva      100
Sam       94
dtype: int64
```

6. Accessing elements of a series via custom indices

We can access individual elements via **square brackets** containing a custom index value in a series with custom indices.

Type the following codes:

```
In [24]: grades['Eva']
Out[24]: 100
```

If custom indices are strings that could represent valid Python identifiers, pandas automatically adds them to the Series as attributes so that we can access via a dot(**.**) operation.

```
In [25]: grades.Wally
Out[25]: 87
```

If a Series contains strings, we can use its **str** attribute to call string methods on the elements. We first create a Series of hardware-related strings.

Type the following codes:

```
In [26]: hardware = pd.Series(['Hammer', 'Saw', 'Wrench'])

In [27]: hardware
Out[27]:
0    Hammer
1       Saw
2    Wren
dtype: ob
```

```
In [28]: hardware.str.contains('a')
Out[28]:
0     True
1     True
2    False
dtype: bool
```

We then call string method **contains** on each element to determine whether the value of each element contains a lowercase 'a'.

In the codes above, pandas returns a Series containing **bool** values indicating the **contains** method's result for each element. For example, element at index 2('Wrench') does not contain an 'a', so its element in the resulting Series is False.

Task 3. DataFrames

A DataFrame is an enhanced two-dimensional array. Like Series, DataFrames can have custom row and column indices, and offer additional operations that are suitable for many data-science tasks. DataFrames also support missing data. Each column in a DataFrame is a Series.

1. Creating a DataFrame from a dictionary

We now create a DataFrame from a dictionary that represents student grades on three exams.

Type the following codes:

```
In [32]: import pandas as pd

In [33]: grades_dict = {'Wally' :[87, 96, 70], 'Eva' :[100, 87, 90], 'Sam' :[94
    ...: , 77, 90], 'Katie' : [100, 81, 82], 'Bob' : [83, 65, 85]}

In [34]: grades = pd.DataFrame(grades_dict)

In [35]: grades
Out[35]:
   Wally  Eva  Sam  Katie  Bob
0     87  100   94    100   83
1     96   87   77     81   65
2     70   90   90     82   85
```

In the codes above, the dictionary's keys become the column names and values associated with each key become the element values in the corresponding column.

2. Customizing a DataFrame's indices with the index attribute

We could have specified custom indices with the **index** keyword argument when we created the DataFrame.

Type the following codes:

```
In [36]: pd.DataFrame(grades_dict, index=['Test1', 'Test2', 'Test3'])
Out[36]:
       Wally  Eva  Sam  Katie  Bob
Test1     87  100   94    100   83
Test2     96   87   77     81   65
Test3     70   90   90     82   85
```

We can use the index attribute to change the DataFrame's indices from sequential integers to labels.

Type the following codes:

```
In [38]: grades.index = ['Test1', 'Test2', 'Test3']

In [39]: grades
Out[39]:
       Wally  Eva  Sam  Katie  Bob
Test1     87  100   94    100   83
Test2     96   87   77     81   65
Test3     70   90   90     82   85
```

In the codes above, when specifying the indices, we must provide a one-dimensional collection that has the same number of elements as there are rows in the DataFrame. Otherwise, a ValueEffor occurs.

3. Accessing a DataFrame's columns

We can quickly and conveniently look at our data in many different ways, including selecting portions of the data.

Type the following codes:

```
In [40]: grades['Eva']
Out[40]:
Test1    100
Test2     87
Test3     90
Name: Eva, dtype: int64
```

In the above, Eva's codes we get grades by name, which displays her column as a Series.

If a DataFrame's column-name strings are valid Python identifiers, we can use them as attributes. In the following codes, we get Sam's grades with the Sam attribute.

Type the following codes:

```
In [42]: grades.Sam
Out[42]:
Test1    94
Test2    77
Test3    90
Name: Sam, dtype: int64
```

4. Selecting rows via the loc **and** iloc **attribute**

The pandas recommends using the attributes **loc, iloc, at** and **iat** to access DataFrames although DataFrames support indexing capabilities with [ ]. We can access a row by its label via the DataFrame's loc attribute.

Type the following codes:

```
In [12]: grades.loc['Test1']
Out[12]:
Wally     87
Eva      100
Sam       94
Katie    100
Bob       83
Name: Test1, dtype: int64
```

We also access integer based can rows by zero- indices using the **iloc** attribute(the **i** in **iloc** means that it's used with integer indices).

Type the following codes:

```
In [13]: grades.iloc[1]
Out[13]:
Wally    96
Eva      87
Sam      77
Katie    81
Bob      65
Name: Test2, dtype: int64
```

In the codes above, the line 13 lists all the grades in the second row.

5. selecting rows via slices and lists with the loc and iloc attributes

The index can be a slice. When using slices containing labels with **loc**, the range specified includes the high index('Test3').

Type the following codes:

```
In [14]: grades.loc['Test1' : 'Test3']
Out[14]:
       Wally  Eva  Sam  Katie  Bob
Test1     87  100   94    100   83
Test2     96   87   77     81   65
Test3     70   90   90     82   85
```

When using slices containing integer indices with **iloc**, the range we specify excludes the high index (2).

Type the following codes:

```
In [15]: grades.iloc[0:2]
Out[15]:
       Wally  Eva  Sam  Katie  Bob
Test1     87  100   94    100   83
Test2     96   87   77     81   65
```

To select specific rows, we can use a list rather than slice notation with **loc** or **iloc**.

Type the following codes:

```
In [16]: grades.loc[['Test1', 'Test3']]
Out[16]:
       Wally  Eva  Sam  Katie  Bob
Test1     87  100   94    100   83
Test3     70   90   90     82   85

In [17]: grades.iloc[[0, 2]]
Out[17]:
       Wally  Eva  Sam  Katie  Bob
Test1     87  100   94    100   83
Test3     70   90   90     82   85
```

6. Selecting subsets of the rows and columns

We can use two slices, two lists or a combination of slices and lists to select rows and columns in which we focus on small subsets of a DataFrame.

Type the following codes:

```
In [18]: grades.loc['Test1':'Test2' , ['Eva', 'Katie']]
Out[18]:
       Eva  Katie
Test1  100    100
Test2   87     81
```

In the codes above, we want to view only Eva's and Katie's grades on Test1 and Test2. The slice 'Test1' : 'Test2' selects the rows for Test1 and Test2. The list ['Eva', 'Katie'] selects only the corresponding grades from those two columns.

We can use **iloc** with a list and a slice to select the first and third tests and the first three columns for those tests by using the following codes.

Type the following codes:

```
In [19]: grades.iloc[[0, 2], 0:3]
Out[19]:
       Wally  Eva  Sam
Test1     87  100   94
Test3     70   90   90
```

7. Boolean Index

Boolean indexing provides powerful selection capabilities.

Type the following codes:

```
In [20]: grades[grades >= 90]
Out[20]:
       Wally    Eva   Sam  Katie  Bob
Test1    NaN  100.0  94.0  100.0  NaN
Test2   96.0    NaN   NaN    NaN  NaN
Test3    NaN   90.0  90.0    NaN  NaN
```

In the codes above, we select all grades that are greater than or equal 90. Pandas checks every grade to determine whether its value is greater than or equal to 90 and, if so, includes it in the new DataFrame. Grades for which the condition is False are represented as NaN(not a number) in the new DataFrame. NaN is notation for missing values.

Now we try to type the following codes.

```
In [21]: grades[(grades >= 80) & (grades < 90)]
Out[21]:
       Wally   Eva  Sam  Katie   Bob
Test1   87.0   NaN  NaN    NaN  83.0
Test2    NaN  87.0  NaN   81.0   NaN
Test3    NaN   NaN  NaN   82.0  85.0
```

In the codes above, we select all grades in the range 80-89.

8. Accessing a specific DataFrame cell by row and column

We can use a DataFrame's **at** and **iat** attributes to get a single value from a DataFrame. Like loc and iloc, **at** uses labels and **iat** uses integer indices. For example, we can select Eva's Test2 grade(87) and Wally's Test3 grade (70).

Type the following codes:

```
In [22]: grades.at['Test2', 'Eva']
Out[22]: 87

In [23]: grades.iat[2, 0]
Out[23]: 70
```

We also can assign new values to specific elements.

```
In [24]: grades.at['Test2', 'Eva'] = 100

In [25]: grades.at['Test2', 'Eva']
Out[25]: 100

In [26]: grades.iat[1, 2] = 87

In [27]: grades.iat[1, 2]
Out[27]: 87
```

In the codes above, we change Eva's Test2 grade to 100 using at, then change it back to 87 using **iat**.

Task 4. Answering questions (Please do this at your home by using your own computer)

1. What are the critical success factors for a kNN algorithm?

2. What is the major difference between cluster analysis and classification?

3. Describe K-means algorithm.

4. Given the following dictionary:

temps = {'Mon' : [68, 89], 'Tue' : [71, 93], 'Wed' : [66, 82], 'Thu' : [75, 97], 'Fri' : [62, 79] }

perform the following tasks:

a) Convert the dictionary into the DataFrame named temperatures with 'Low' and 'High' as the indices, then display the DataFrame.

b) Use the column names to select only the columns for 'Mon' through 'Wed'.

c) Use the row index 'Low' to select only the low temperatures for each day.