STUDY OF PYTHON PROGRAMMING LEARNING ASSISTANT SYSTEM (PYPLAS)

Pandas Library Usage

Funabiki Laboratory

Department of Electrical and Communication Engineering

Okayama University

Python Programming Learning Assistant System (PyPLAS)

Correspondence Between Each Topic and Related VTPs

Correspondence Between Each Topic and Related VTPs

※ Click the link to go reference pages directly.

Pandas Concepts	Related VTPs Version	Related Problem Numbers
What is a Pandas dataframe and dataframe creation?	VTP_P5	1
Indexing and Selecting Data	VTP_P5	6
Excel File Reading	VTP_P5	2
Excel File Writing	VTP_P5	3
What is CSV file?	VTP_P5	4, 5
Iterate Pandas Dataframe	VTP_P5	7
Aggregration in Pandas Dataframe	VTP_P5	8
Pandas Datetime Function	VTP_P5	9
Options and Customizations	VTP_P5	10

- What is Pandas?

Pandas is an open source Python package that is most widely used for data science/data analysis and machine learning tasks. It is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language. It presents a diverse range of utilities, ranging from parsing multiple file formats to converting an entire data table into a NumPy matrix array. It is built on top of another package named Numpy, which provides support for multi-dimensional arrays. As one of the most popular data wrangling packages, Pandas works well with many other data science modules inside the Python ecosystem, and is typically included in every Python distribution, from those that come with the operating system to commercial vendor distributions like ActiveState's ActivePython. Pandas makes it simple to do many of the time consuming, repetitive tasks associated with working with data, including:

1. Data cleansing
2. Data fill
3. Data normalization
4. Merges and joins
5. Data visualization
6. Statistical analysis
7. Data inspection
8. Loading and saving data

Correspondence Between Each Topic and Related VTPs

9. And much more

- What is a Pandas DataFrame and DataFrame Creation?

To use pandas library, firstly we need to change the dataframe format. Pandas dataFrame is the data Structure, which is a 2 dimensional Array. DataFrames are visually represented in the form of a table. DataFrame stores tabular data to easily manipulate it like rows and columns. A dataframe can be created from a list, or a dictionary or numpy array.

1. Create DataFrame from list
   A single list can be turned into a pandas dataframe:

```
import pandas as pd
data = [9,4,3]
df = pd.DataFrame(data)
print(df)
Output
      0
0     9
1     4
2     3
```

Every element has an index (0,1,2).

2. Create DataFrame from n-demitional arrays

   This works for tables (n-dimensional arrays) too:

```
import pandas as pd
data = [['Brown',22], ['John', 20], ['Smitt', 45]]
df = pd.DataFrame(data,columns=['Name','Age']))
print(df)
Output
      Name    Age
0     Brown   22
```

Correspondence Between Each Topic and Related VTPs

```
1       John      20
2       Smitt     45
```

3. Create DataFrame from a dictionary

   From a dictionary, a dataframe can be created.

```
import pandas as pd
d = {'one':[7,2,9], 'two':[2,0,4], 'three':[3,2,1] }
df = pd.DataFrame(d)
print(df)
```

Output:

```
      one   two   three
0     7     2       9
1     2     0       4
2     3     2       1
```

The default index value begins form zero. However, the index value can be changed as follows.

```
df = pd.DataFrame(d, index=['first','second','third'])
print(df)
```

Output:

```
             one   two   three
first        7     2       9
second       2     0       4
third        3     2       1
```

4. Create DataFrame from array

   An array (numpy array) can be converted into an dataframe too.

```
import numpy as np
ar = np.array([[1,2,3],[4,5,6],[6,7,8]])
df = pd.DataFrame(ar)
print(df)
```

Output:

	0	1	2
0	1	2	3
1	4	5	6
2	6	7	8

5. Create DataFrame from DataFrame
   The parts of a dataframe can be copied into a new dataframe.

```
df2 = df[[‘One’,’Two’]].copy();          #using the dataframe above
print( df2)
```

Output:

	One	Two
A	1	2
B	4	5
C	6	7

- Indexing and Selecting Data in Pandas Frame

Indexing in pandas is a very crucial function. It lets to select and observe data according to requirements and thus allows getting of one step closer to improve data analysis. Without indexing and selection of data in Pandas, analyzing data would be extremely difficult. With the help of custom indices, we can access our data properly and also manage it efficiently. Pandas indexing and selecting help to efficiently customize the data.

1. Select column
   To select a column, the column name can be used.

```
data = [['Brown',22], ['John', 20], ['Smitt', 45]]
df = pd.DataFrame(data,columns=['Name','Age'])
print(df['Name'])
```

Output:

0 Brown
1 John

```
2 smitt
```

2. Column Addition

```
data = [['Brown',22], ['John', 20], ['Smitt', 45]]
df = pd.DataFrame(data,columns=['Name','Age'])
c = pd.DataFrame([$10,$11,$40], columns=['Salary'])
df[‘Salary’] = c[‘Salary’]
print(df)
Output:
    Name   Age   Salary
0   Brown   22     $10
1   John    20     $11
2   Smitt    45     $40
```

3. Column Deletion

   To delete a column, the keywork **del** can be used.

```
data = [['Brown',22], ['John', 20], ['Smitt', 45]]
df = pd.DataFrame(data,columns=['Name','Age'])
c = pd.DataFrame([$10,$11,$40], columns=['Salary'])
df[‘Salary’] = c[‘Salary’]
>>> del df[‘Salary’]
Output:
    Name   Age
0   Brown   22
1   John    20
2   Smitt    45
```

4. Select row

   To select a row, the keywork **.loc[index]** or **.iloc[index]** can be used.

```
data = [['Brown',22], ['John', 20], ['Smitt', 45]]
df = pd.DataFrame(data,columns=['Name','Age'])
>>> df.loc[0]
Output:
Name   Brown
```

```
Age       22
Name: 0. dtype: object
```

5. Append row

   To append a row, the method **.append()** can be used.

```
data = [['Brown',22], ['John', 20], ['Smitt', 45]]
df = pd.DataFrame(data,columns=['Name','Age'])
user = pd.DataFrame([['Tony',53]], columns= ['Name','Age'])
df = df.append(user)
print(df)
```

Output:

```
    Name    Age
0   Brown    22
1   John     20
2   Smitt     45
3   Tony     53
```

6. Row Deletion

   To delete a column, the method .**drop(index)** can be used.

```
data = [['Brown',22], ['John', 20], ['Smitt', 45]]
df = pd.DataFrame(data,columns=['Name','Age'])
print(df.drop(0))
```

Output:

```
    Name    Age
1   John     20
2   Smitt     45
```

- Excel file Reading

Excel files (the first sheet, specific sheets, multiple sheets or all sheets) can be read by using **read_excel**(‘sample.xlsx’) method. If there are multiple sheets, only the first sheet is used by pandas. It reads as DataFrame.

1. Read Excel file

   To read an excel file, the method **.read_excel()** can be used.

```
import pandas as pd
df = pd.read_excel('sample.xlsx')
```

2. Get sheet from Excel file

   To specify the sheet, the argument **sheet_name** can be used. It specifies by number (starting at 0).

```
import pandas as pd
df = pd.read_excel('sample.xlsx', sheet_name=1)
```

3. Load multiple sheets from Excel file

   It is also possible to specify a list in the argument sheet_name. It is OK even if it is a number of 0 starting or the sheet name.

```
import pandas as pd
df = pd.read_excel('sample.xlsx', sheet_name=[0, 'sheet2'])
```

4. Load all sheets from Excel file

   If sheet_name argument is none, all sheets are read.

```
import pandas as pd
df = pd.read_excel('sample.xlsx', sheet_name= None)
```

- Excel file Writing

Any data (lists, strings, numbers etc) can be written to Excel, by first converting it into a Pandas DataFrame and then writing the DataFrame to Excel. Importing **openpyxl** is required to append it to an existing Excel file described at the end.

1. Write Excel file

   To write an excel file, the method **.to_excel()** can be used.

```
import pandas as pd
import openpyxl
df = pd.read_excel('sample.xlsx')
df = pd.DataFrame([[11, 21, 31], [12, 22, 32], [31, 32, 33]],
                  index=['one', 'two', 'three'], columns=['a', 'b', 'c'])
```

```
df.to_excel('pandas_to_excel.xlsx', sheet_name='new_sheet_name')
```

Note: that the data in the original file is deleted when overwriting. The argument new_sheet_name is the name of the sheet. If omitted, it will be named Sheet1.

2. Write multiple DataFrames toExcel file

   To write multiple data frames to an excel file as separate sheets, the method **ExcelWriter()**can be used.

```
import pandas as pd
import openpyxl
df = pd.read_excel('sample.xlsx')
df = pd.DataFrame([[11, 21, 31], [12, 22, 32], [31, 32, 33]],
                  index=['one', 'two', 'three'], columns=['a', 'b', 'c'])
df2 = df[['a', 'c']] // prepare another data frame
with pd.ExcelWriter('pandas_to_excel.xlsx') as writer:
  df.to_excel(writer, sheet_name='sheet1')
  df2.to_excel(writer, sheet_name='sheet2')
```

3. Append to an existing Excel file

   To append a DataFrame to an existing excel file, the method **.**The code below opens an existing file, then adds two sheets with the data of the dataframes.

```
import pandas as pd
import openpyxl
path = 'pandas_to_excel.xlsx'
df = pd.read_excel('sample.xlsx')
df = pd.DataFrame([[11, 21, 31], [12, 22, 32], [31, 32, 33]],
                  index=['one', 'two', 'three'], columns=['a', 'b', 'c'])
df2 = df[['a', 'c']] // prepare another data frame
with pd.ExcelWriter(path) as writer:
    writer.book = openpyxl.load_workbook(path)
    df.to_excel(writer, sheet_name='new_sheet1')
    df2.to_excel(writer, sheet_name='new_sheet2')
```

- What is a CSV File (CSV file Reading and Writing)?

A CSV file is nothing more than a simple text file. However, it is the most common, simple, and easiest method to store tabular data. This particular format arranges tables by following a specific structure divided into rows and columns. It is these rows and columns that contain the data. A new line terminates each row to start the next row. Similarly, a comma, also known as the delimiter, separates columns within each row.

1. Read CSV file

To read a CSV file, the method **read_csv()** can be used.

```
import pandas as pd
df = pd.read_csv('sample.csv')
df.data_head() # display the first five rows of the CSV file
```

The read_csv() method then returns a Pandas DataFrame that contains the data of the CSV file. By default, the read_csv() method treats the values in the first row of a CSV file as column headers. However, custom header names can be passed while reading a file via the read_csv() method as bellow code.

```
import pandas as pd
col_names = [‘Name’, ‘Age’, ‘Salary’]
df = pd.read_csv(r'sample.csv', names=col_names, header=None)
```

2. Write CSV file

The process of creating or writing a CSV file through Pandas can be a little more complicated than reading CSV, but it's still relatively simple. **to_csv()** function can be used to perform this task. However, a Pandas DataFrame must be first created, followed by writing that DataFrame to the CSV file.

```
import pandas as pd
city = pd.DataFrame([['Sacramento', 'California'], ['Miami', 'Florida']], columns=['City', 'State’]
city_to_csv(‘city.csv’)
```

Output (city.csv)
City,State
Sacramento, California
Miami, Florida

3. Writing to CSV files using CSV module

Just like reading CSVs, the csv module appropriately provides plenty of functionality to write data to a CSV file as well. The writer object presents two functions, namely writerow() and writerows(). The difference between them, is that the first function will only write one row, and the function writerows() writes several rows at on csv file.

```
import csv
myData = [[1,2,3], ['Good Morning', 'Good Evening', 'Good Afternoon']]
myFile = open('csvexample3.csv', 'w')
with myFile:
    writer = csv.writer(myFile)
    writer.writerows(myData)
```

Output (csvexample3.csv)
1,2,3
Good Morning,Good Evening,Good Afternoon

- Iterate Pandas Dataframe

DataFrame looping (iteration) with a for statement. Over a pandas dataframe can be looped, for each column row by row. There are 3 ways to loop over data frame.

1. **iteritems()** : Helps to iterate over each element of the set, column-wise.
2. **iterrows():** Each element of the set, row-wise.
3. **itertuple():** Each row and form a tuple out of them.

```
import pandas as pd
df = pd.DataFrame({'age': [20, 32], 'state': ['NY', 'CA'], 'point': [64, 92]},
                  index=['Alice', 'Bob'])
for key,values in df.iteritems():
```

```
        print(key, values)
for row_index,row in df.iterrows():
        print(row_index, row)
for row in df.itertuples():
    print(row)
```

- Aggregation in Pandas DataFrame

Pandas provide with a variety of aggregate functions. These functions help to perform various activities on the datasets. The functions are:

1. **count():** This gives a count of the data in a column.
2. **.sum():** This gives the sum of data in a column.
3. **.min() and .max():** This helps to find the minimum value and maximum value, ina function, respectively.
4. **.mean() and .median():** Helps to find the mean and median, of the values in a column, respectively.

```
import pandas as pd
df = pd.read_csv(r'sample.csv')
 print(df.count())
print(df.sum())
print(df.min())
print(df.max())
print(df.mean())
print(df.median())
```

- Pandas Datetime Functionality

To create pandas DateTime object, **.date_range function** is used. It has the following parameter:

1. **First parameter**: start= ‘dd/mm/yyyy’.
2. **Second parameter:** periods= n, where n is no of periods or date time elements you need.
3. **Third parameter**: freq= ‘x’, where ‘x’ can be ‘H’(hour), ‘D’(days), ‘W’(weeks), ‘M’(month), ‘Y’(years), etc.

```
Import pandas as pd
dataflair = pd.date_range(start='1/1/2011', periods = 10, freq =’H’)
print(dataflair)
```

Output

```
DatetimeIndex([‘2011-01-01 00:00:00’, ‘2011-01-01 01:00:00’,
‘2011-01-01 02:00:00’, ‘2011-01-01 03:00:00’,
‘2011-01-01 04:00:00’, ‘2011-01-01 05:00:00’,
‘2011-01-01 06:00:00’, ‘2011-01-01 07:00:00’,
‘2011-01-01 08:00:00’, ‘2011-01-01 09:00:00′],
dtype=’datetime64[ns]’, freq=’H’)
```

Breaking the dime and date into separate features

```
dataflair_rng['year'] = dataflair_rng['date'].dt.year # creates ‘year’ column and extracts year
dataflair_rng['month'] = dataflair_rng['date'].dt.month # creates ‘month’ column and extracts month
dataflair_rng['day'] = dataflair_rng['date'].dt.day # creates ‘day’ column and extracts the day
dataflair_rng['hour'] = dataflair_rng['date'].dt.hour # creates ‘hour’ column and extracts the hour
dataflair_rng['minute'] = dataflair_rng['date'].dt.minute # creates ‘minute’ column and extracts minute
```

Get the present time as a timestamp

.Timestamp.now() function is used to get the current time and date details.

```
dataflair_time = pd.Timestamp.now()
```

- Pandas Options and Customizations

Pandas options allow a user to customize the data according. Pandas have some default factors which restrict the analysis of data. Therefore to have a stronghold over the library and to make the most out of its uses, it is important to know the various methods to change the default pandas values.

Correspondence Between Each Topic and Related VTPs

Common default values-

1. **display.max_rows** and **display.max_columns** which shows the default number of rows and columns.
2. **display.max_colwidth** which gives us the maximum width of the column
3. **display.expand_frame_repr** which gives us DataFrames that is spread across numerous pages.
4. **display.precision** gives us the precision of the decimal numbers

Types of Pandas Options and Customization

There are five types of pandas options. They are

1. **.get_option():** can define parameter which will give a particular detail about the default values in pandas.
2. **.set_option()**: allows to change a default value to something of choice.
3. **.reset_option():** can get back the default values which may change previously.
4. **.describe_option():** describes the details about each parameter.
5. **.opton_context():** can invoke a pandas option function which will be only active within the scope of the function.

```
import pandas as pd
dataflair= pd.get_option("display.max_rows")
pd.set_option("display.max_rows",90)
pd.reset_option("display.max_rows")
pd.describe_option("display.max_rows")
with pd.option_context("display.max_rows",30):
    print(pd.get_option("display.max_rows"))
```