Pandas

29-2-2024

By Anam Suri, NET-JRF

- Pandas is most popular Python library for tabular data structures. You can think of Pandas as an extremely powerful version of Excel (but free and with a lot more features!).
- We usually import pandas with the alias pd.

```
import pandas as pd
```

Pandas Series

- A Series is like a NumPy array but with labels.
- They are strictly 1-dimensional and can contain any data type (integers, strings, floats, objects, etc), including a mix of them.
- Series can be created from a scalar, a list, ndarray or dictionary using pd.Series()

Series Example

Series 1

INDEX	DATA
0	A
1	B
2	C
3	D
4	E
5	F

Series 2

INDEX	DATA
A	1
B	2
C	3
D	4
E	5
F	6

Series 3

INDEX	DATA
0	[1, 2]
1	A
2	1
3	(4, 5)
4	{"a": 1}
5	6

Series 4

INDEX	DATA
Jan-18	11
Feb-18	23
Mar-18	43
Apr-18	21
May-18	17
Jun-18	6

Creating Series

By default, series are labelled with indices starting from 0. For example:

```
pd.Series(data = [-5, 1.3, 21, 6, 3])
```

Output:

```
0    -5.0
1     1.3
2    21.0
3     6.0
4     3.0
dtype: float64
```

But you can add a custom index:

```
pd.Series(data = [-5, 1.3, 21, 6, 3],
          index = ['a', 'b', 'c', 'd', 'e'])
```

Output

```
a    -5.0
b     1.3
c    21.0
d     6.0
e     3.0
dtype: float64
```

You can create a Series from a dictionary:

```
pd.Series(data = {'a': 10, 'b': 20, 'c': 30})
```

Output

```
a    10
b    20
c    30
dtype: int64
```

- Or from an ndarray:

```
pd.Series(data = np.random.randn(3))
```

- Or even a scalar:

```
pd.Series(3.141)
pd.Series(data=3.141, index=['a', 'b', 'c'])
```

Series Characteristics

- Series can be given a name attribute

```
s = pd.Series(data = np.random.randn(5), name='random_series’)
s

s.name #RUN

s.rename("another_name")
s.name #RUN
```

We can access the index labels of your series using the .index attributes.

```
s.index
```

o/p:RangeIndex(start=0, stop=5, step=1)

access the underlying data array using .to_numpy()

```
s.to_numpy()

pd.Series([[1, 2, 3], "b", 1]).to_numpy()
```

Indexing and Slicing Series

Series are very much like ndarrays (in fact, series can be passed to most NumPy functions!). They can be indexed using square brackets [] and sliced using colon : notation.

```
s = pd.Series(data = range(5),
          index = ['A', 'B', 'C', 'D', 'E'])
S
S[0]
S[[0]]
S[[1,2,3]]
S[0:3]
```

- Note above how array-based indexing and slicing also returns the series index.
- Series are also like dictionaries in that we can access values using index labels:

```
S["A"]
s[["B", "D", "C"]]
s["A":"C"]
"A" in s
"Z" in s
```

- Series do allow for non-unique indexing, but **be careful** because indexing operations won't return unique values:

```
x = pd.Series(data = range(5),
          index = ["A", "A", "A", "B", "C"])
x
x["A"]
```

- Finally, we can also do boolean indexing with series:

```
s[s >= 1]
s[s > s.mean()]
(s != 1)
```

Series operations

Unlike ndarrays, operations between Series (+, -, /, *) align values based on their LABELS (not their position in the structure).

The resulting index will be the ***sorted union*** of the two indexes. This gives you the flexibility to run operations on series regardless of their labels.

```
s1 = pd.Series(data = range(4),
          index = ["A", "B", "C", "D"])
s1
s2 = pd.Series(data = range(10, 14),
          index = ["B", "C", "D", "E"])
S2

s1+s2???
```

s1+s2

Series 1

INDEX	DATA
A	0
B	1
C	2
D	3
-	-

Series 2

INDEX	DATA
-	-
B	10
C	11
D	12
E	13

Series 1 + Series 2

INDEX	DATA
A	NaN
B	11
C	13
D	15
E	NaN

- **"Chaining"** operations together is also common with pandas:

s1.add(3.141).pow(2).mean().astype(int)

Data Types

Series can hold all the data types (dtypes) we're used to int, float, bool, etc. Series can hold all the data types (object, DateTime, and Categorical)

x = pd.Series(range(5))

x.Dtype #int64

- The dtype "obejct" is used for series of strings or mixed data.

```
x = pd.Series(['A', 'B'])
X
x = pd.Series(['A', 1, ["I", "AM", "A", "LIST"]])
X
```

While flexible, it is recommended to avoid the use of object dtypes because of higher memory requirements. Essentially, in an object dtype series, every single element stores information about its individual dtype. We can inspect the dtypes of all the elements in a mixed series in several ways, below I'll use the map function:

```
x.map(type)
O/P:
0    <class 'str'>
1    <class 'int'>
2   <class 'list'>
dtype: object
```

```
x1 = pd.Series([1, 2, 3])
print(f"x1 dtype: {x1.dtype}")
print(f"x1 memory usage: {x1.memory_usage(deep=True)} bytes")
print("")
x2 = pd.Series([1, 2, "3"])
print(f"x2 dtype: {x2.dtype}")
print(f"x2 memory usage: {x2.memory_usage(deep=True)} bytes")
print("")
x3 = pd.Series([1, 2, "3"]).astype('int8')  # coerce the object series to int8
print(f"x3 dtype: {x3.dtype}")
print(f"x3 memory usage: {x3.memory_usage(deep=True)} bytes")
```

One more gotcha .NaN (frequently used to represent missing values in data) is a float:

```
type(np.NaN)
```

This can be problematic if you have a series of integers and one missing value, because Pandas will cast the whole series to a float:

```
pd.Series([1, 2, 3, np.NaN])
```

Only recently, Pandas has implemented a "nullable integer dtype", which can handle NaN in an integer series without affecting the dtype. Note the captial "I" in the type below, differentiating it from numpy's int64 dtype

```
pd.Series([1, 2, 3, np.NaN]).astypeInt64')
```

- This is not the default in Pandas yet and functionality of this new feature is still subject to change.

Pandas Dataframe

What is a DataFrame?

- DataFrames are really just Series stuck together!
- Think of a DataFrame as a dictionary of series, with the “keys” being the column labels and the “values” being the series data.
- A two-dimensional data structure, like a 2 dimensional array, or a table with rows and columns.

pip install pandas

import pandas as pd

Creating DataFrames

- Dataframes can be created using pd.DataFrame() (note the capital “D” and “F”).
- Like series, index and column labels of dataframes are labelled starting from 0 by default:

```
pd.DataFrame([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
```

- ```
  import pandas as pd

  data = {
    "calories": [420, 380, 390],
    "duration": [50, 40, 45]
  }

  #load data into a DataFrame object:
  df = pd.DataFrame(data)

  print(df)
  ```

- We can use the index and columns arguments to give them labels:
  - ```
    pd.DataFrame([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]],
                  index = ["R1", "R2", "R3"],
                  columns = ["C1", "C2", "C3"])
    ```
- There are so many ways to create dataframes.
- We can create it by using dictionaries or ndarrays:

```
pd.DataFrame({"C1": [1, 2, 3],
              "C2": ['A', 'B', 'C']},
               index=["R1", "R2", "R3"])

pd.DataFrame(np.random.randn(5, 5),
    index=[f"row_{_}" for _ in range(1, 6)],
    columns=[f"col_{_}" for _ in range(1, 6)])

pd.DataFrame(np.array([['Tom', 7], ['Mike', 15], ['Tiffany', 3]]))
```

Create DataFrame from	Code
Lists of lists	pd.DataFrame([['Tom', 7], ['Mike', 15], ['Tiffany', 3]])
ndarray	pd.DataFrame(np.array([['Tom', 7], ['Mike', 15], ['Tiffany', 3]]))
Dictionary	pd.DataFrame({"Name": ['Tom', 'Mike', 'Tiffany'], "Number": [7, 15 , 3]})
List of tuples	pd.DataFrame(zip(['Tom', 'Mike', 'Tiffany'], [7, 15, 3]))
Series	pd.DataFrame({"Name": pd.Series(['Tom', 'Mike', 'Tiffany']), "Num ber": pd.Series([7,

Indexing and Slicing DataFrames

- There are several ways to select data from a DataFrame:
  - []
  - .loc[]
  - .iloc[]
  - Boolean indexing
  - .query()

```
df = pd.DataFrame({"Name": ["Tom", "Mike", "Tiffany"],
             "Language": ["Python", "Python", "R"],
              "Courses": [5, 4, 7]})
```

Indexing with [ ]

- Select columns by single labels, lists of labels, or slices:
  - df['Name'] *# returns a series*
  - df[['Name']] *# returns a dataframe!*
- df[['Name', 'Language’]]
- Note: You can only index rows by using slices, not single values. For example: df[0] # doesn't work
- df[0:1] # does work
- df[1:] # does work

Indexing with .loc and .iloc

- Pandas created the methods .loc[] and .iloc[] as more flexible alternatives for accessing data from a dataframe.
- Use df.iloc[] for indexing with integers and df.loc[] for indexing with labels.
- These are typically the recommended methods of indexing in Pandas.
- **iloc**
  - .iloc is integer-location-based indexing, where you use the integer positions to make selections, similar to array indexing in Python.
  - You can use integers, slices, or boolean arrays to make selections.
  - When using .iloc, the stop of the slice is excluded, following the standard Python behavior.
  - The syntax for .iloc is df.iloc[row_integer, column_integer] or df.iloc[row_integer_slice, column_integer_slice].
  - Example:
    - df.iloc[0] # returns a series
    - df.iloc[0:2] # slicing returns a dataframe
    - df.iloc[2, 1] # returns the indexed object
    - df.iloc[[0, 1], [1, 2]] # returns a dataframe

- **.loc:**
  - .loc is primarily label-based indexing, which means you use the labels of rows and columns to make selections.
  - You can use actual labels (like strings or integers), slices, or boolean arrays to make selections.
  - When using .loc, both the start and stop of the slice are included.
  - The syntax for .loc is df.loc[row_label, column_label] or df.loc[row_label_slice, column_label_slice].
  - Now let's look at .loc which accepts labels as references to rows/columns:
    - df.loc[:, 'Name']
    - df.loc[:, 'Name':'Language']
    - df.loc[[0, 2], ['Language']]

- Sometimes we want to use a mix of integers and labels to reference data in a dataframe.
- The easiest way to do this is to use .loc[] with a label then use an integer in combinations with .index or .columns:
  - df.index
  - df.columns
  - df.loc[df.index[0], 'Courses'] # referencing the first row and the column named "Courses"
  - df.loc[2, df.columns[1]] # referencing row "2" and the second column

Boolean Indexing

- Just like with series, we can select data based on boolean masks.
- For example:
  - df[df['Courses'] > 5]
  - df[df['Name'] == "Tom"]

Indexing with .query()

- df.query() is a powerful tool for filtering data.
- df.query() accepts a string expression to evaluate and it "knows" the names of the columns in your dataframe.
  - df.query("Courses > 4 & Language == 'Python'")
  - df[(df['Courses'] > 4) & (df['Language'] == 'Python')]
  - Query also allows you to reference variable in the current workspace using the @ symbol:
    - course_threshold = 4
      df.query("Courses > @course_threshold")

Method	Syntax	Output
Select column	df[col_label]	Series
Select row slice	df[row_1_int:row_2_int]	DataFrame
Select row/column by label	df.loc[row_label(s), col_label(s)]	Object for single selection, Series for one row/column, otherwise DataFrame
Select row/column by integer	df.iloc[row_int(s), col_int(s)]	Object for single selection, Series for one row/column, otherwise DataFrame
Select by row integer & column label	df.loc[df.index[row_int], col_label]	Object for single selection, Series for one row/column, otherwise DataFrame
Select by row label & column integer	df.loc[row_label, df.columns[col_int]]	Object for single selection, Series for one row/column, otherwise DataFrame
Select by boolean	df[bool_vec]	Object for single selection, Series for one row/column, otherwise DataFrame
Select by boolean expression	df.query("expression")	Object for single selection, Series for one row/column, otherwise DataFrame

Reading/Writing Data From External Sources

- A lot of the time you will be loading .csv files for use in pandas.
- You can use the pd.read_csv() function for this.
- Read_csv accepts the following common arguments:

pandas.read_csv(*filepath_or_buffer*, *, *sep*=***_NoDefault.no_default*, *delimiter*=*None*, *header*=*'infer'*, *names*=*_NoDefault.no_default*, *index_col*=*None*, *usecols*=*None*, *dtype*=*None*, etc……)**

- Pandas also facilitates reading directly from a url - pd.read_csv() accepts urls as input:
  - url = 'https://raw.githubusercontent.com/TomasBeuzen/toy-datasets/master/wine_1.csv'
  - pd.read_csv(url)

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	Fixed-Width Text File	read_fwf	
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	LaTeX		Styler.to_latex
text	XML	read_xml	to_xml
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	OpenDocument	read_excel	
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	ORC Format	read_orc	to_orc
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	SPSS	read_spss	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google BigQuery	read_gbq	to_gbq

Common DataFrame Operations

- DataFrames have built-in functions for performing most common operations, e.g., .min(), idxmin(), sort_values(), etc.
- df.min()
- df['Time'].min()
- df.iloc[20]

- we’ve seen how to go from: ndarray (np.array()) -> series (pd.series()) -> dataframe (pd.DataFrame()). Remember that we can also go the other way: dataframe/series -> ndarray using df.to_numpy().

Data Wrangling with Pandas

- Inspect a dataframe with df.head(), df.tail(), df.info(), df.describe().
- Obtain dataframe summaries with df.info() and df.describe().
- Rename columns of a dataframe using the df.rename() function or by accessing the df.columns attribute.
- Modify the index name and index values of a dataframe using .set_index(), .reset_index() , df.index.name, .index.
- Use df.melt() and df.pivot() to reshape dataframes, specifically to make tidy dataframes.
- Combine dataframes using df.merge() and pd.concat() and know when to use these different methods.
- Apply functions to a dataframe df.apply() and df.applymap()
- Perform grouping and aggregating operations using df.groupby() and df.agg().
- Perform aggregating methods on grouped or ungrouped objects such as finding the minimum, maximum and sum of values in a dataframe using df.agg().
- Remove or fill missing values in a dataframe with df.dropna() and df.fillna().

DataFrame Characteristics

Head/Tail

The .head() and .tail() methods allow you to view the top/bottom n (default 5) rows of a dataframe.

```
import numpy as np
import pandas as pd
df = pd.read_csv('data/cycling_data.csv')
df.head()
df.head(10)
df.tail()
```

DataFrame Summaries

Three very helpful attributes/functions for getting high-level summaries of your dataframe are:

- .shape
- .info()
- .describe()

❑.shape is just like the ndarray attribute we've seen previously. It gives the shape (rows, cols) of your dataframe:

```
df.shape
```

❑.info() prints information about the dataframe itself, such as dtypes, memory usages, non-null values, etc:

```
df.info()
```

❑.describe() provides summary statistics of the values within a dataframe:

```
df.describe()
```

Note: By default, .describe() only print summaries of numeric features. We can force it to give summaries on all features using the argument include='all' (although they may not make sense!):

```
df.describe(include='all')
```

Displaying Dataframes

- pd.DataFrame(np.random.rand(100))
- df or print(df)
- You can change the setting using pd.set_option("display.max_rows", 20) so that anything with more than 20 rows will be summarised by the first and last 5 rows as before: