

MR2020: Coding for METOC

Module 12: Pandas

What is Pandas?

Pandas is a Python library for data manipulation and analysis. It is best-used for tabular data and provides advanced functionality for data manipulation. Below is an example of what a key Pandas object, the DataFrame, looks like in Interactive Python.

```
df
✓ 0.0s
```

b	vpg	pt1	parcelID	dwdt_hadv	dwdt_vadv	th0conv	qv0conv	pt1conv	cloud_edge_distance
0.065629	-0.021256	0.643482	24899.0	-0.095424	NaN	314.488336	0.011690	0.405142	NaN
0.059478	-0.036742	0.452575	24899.0	-0.075201	NaN	314.451074	0.011665	0.365690	NaN
0.040453	-0.039690	0.382675	24899.0	-0.055048	NaN	314.418553	0.011671	0.475028	NaN
0.025119	-0.032636	0.390716	24899.0	-0.027863	NaN	314.409586	0.011673	0.520689	NaN
0.011536	-0.017074	0.470834	24899.0	-0.030021	NaN	314.415951	0.011655	0.562367	NaN
...
0.043917	-0.175364	0.160459	318836.0	-0.609163	NaN	323.269089	0.001276	0.025568	282.842712
-0.009558	0.163141	0.118840	318836.0	-1.199441	NaN	324.797950	0.001079	0.018192	200.000000
-0.018186	-0.142444	0.123282	318836.0	-1.064771	NaN	328.830050	0.000329	0.005392	200.000000
-0.043341	-0.164271	0.074638	318836.0	0.121079	NaN	329.552825	0.000357	0.006398	NaN
-0.051496	-0.029866	0.069244	318836.0	-0.079110	NaN	329.164487	0.000563	0.011170	100.000000

Data Input

Pandas is capable of reading numerous types of tabular data files. Perhaps the most commonly opened are **CSV** or delimited text (**TXT**) files or Excel (**XLS** or **XLSX**) files.

```
# Read a CSV file  
df = pd.read_csv(filename)
```

```
# Read a text file  
df = pd.read_csv(filename, sep='\t') →
```

```
# Read Excel file  
df = pd.read_excel(filename)
```

When reading a text file, the character(s) separating columns could be anything. If no `sep` is specified, Pandas will default to `sep = ','` (comma).

Other common options:

```
sep = '\t' (tab)  
sep = ' ' (space)  
sep = ';' (semicolon)  
sep = '|' (pipe)  
sep = ':' (colon)
```

Otherwise, any custom text can also be used.

Pandas Classes

DataFrames are two-dimensional labeled data structure. You can almost think of them as a Python version of table or a spreadsheet.

`df` Variable name containing DataFrame

✓ 0.0s

b	vpg	pt1	parcelID	dwdt_hadv	dwdt_vadv	th0conv	qv0conv	pt1conv	cloud_edge_distance
0.065629	-0.021256	0.643482	24899.0	-0.095424	NaN	314.488336	0.011690	0.405142	NaN
0.059478	-0.036742	0.452575	24899.0	-0.075201	NaN	314.451074	0.011665	0.365690	NaN
0.040453	-0.039690	0.382675	24899.0	-0.055048	NaN	314.418553	0.011671	0.475028	NaN
0.025119	-0.032636	0.390716	24899.0	-0.027863	NaN	314.409586	0.011673	0.520689	NaN
0.011536	-0.017074	0.470834	24899.0	-0.030021	NaN	314.415951	0.011655	0.562367	NaN
...
0.043917	-0.175364	0.160459	318836.0	-0.609163	NaN	323.269089	0.001276	0.025568	282.842712
-0.009558	0.163141	0.118840	318836.0	-1.199441	NaN	324.797950	0.001079	0.018192	200.000000
-0.018186	-0.142444	0.123282	318836.0	-1.064771	NaN	328.830050	0.000329	0.005392	200.000000
-0.043341	-0.164271	0.074638	318836.0	0.121079	NaN	329.552825	0.000357	0.006398	NaN
-0.051496	-0.029866	0.069244	318836.0	-0.079110	NaN	329.164487	0.000563	0.011170	100.000000

NOTE: In this DataFrame, only a few random columns are displayed. The **index** is not displayed. We will look at Pandas indices later.

DataFrames are two-dimensional labeled data structure. You can almost think of them as a Python version of table or a spreadsheet.

```
df
✓ 0.0s
```

Column headers

b	vpg	pt1	parcelID	dwdt_hadv	dwdt_vadv	th0conv	qv0conv	pt1conv	cloud_edge_distance
0.065629	-0.021256	0.643482	24899.0	-0.095424	NaN	314.488336	0.011690	0.405142	NaN
0.059478	-0.036742	0.452575	24899.0	-0.075201	NaN	314.451074	0.011665	0.365690	NaN
0.040453	-0.039690	0.382675	24899.0	-0.055048	NaN	314.418553	0.011671	0.475028	NaN
0.025119	-0.032636	0.390716	24899.0	-0.027863	NaN	314.409586	0.011673	0.520689	NaN
0.011536	-0.017074	0.470834	24899.0	-0.030021	NaN	314.415951	0.011655	0.562367	NaN
...
0.043917	-0.175364	0.160459	318836.0	-0.609163	NaN	323.269089	0.001276	0.025568	282.842712
-0.009558	0.163141	0.118840	318836.0	-1.199441	NaN	324.797950	0.001079	0.018192	200.000000
-0.018186	-0.142444	0.123282	318836.0	-1.064771	NaN	328.830050	0.000329	0.005392	200.000000
-0.043341	-0.164271	0.074638	318836.0	0.121079	NaN	329.552825	0.000357	0.006398	NaN
-0.051496	-0.029866	0.069244	318836.0	-0.079110	NaN	329.164487	0.000563	0.011170	100.000000

DataFrames are two-dimensional labeled data structure. You can almost think of them as a Python version of table or a spreadsheet.

```
df
✓ 0.0s
```

Columns

b	vpg	pt1	parcelID	dwdt_hadv	dwdt_vadv	th0conv	qv0conv	pt1conv	cloud_edge_distance
0.065629	-0.021256	0.643482	24899.0	-0.095424	NaN	314.488336	0.011690	0.405142	NaN
0.059478	-0.036742	0.452575	24899.0	-0.075201	NaN	314.451074	0.011665	0.365690	NaN
0.040453	-0.039690	0.382675	24899.0	-0.055048	NaN	314.418553	0.011671	0.475028	NaN
0.025119	-0.032636	0.390716	24899.0	-0.027863	NaN	314.409586	0.011673	0.520689	NaN
0.011536	-0.017074	0.470834	24899.0	-0.030021	NaN	314.415951	0.011655	0.562367	NaN
...
0.043917	-0.175364	0.160459	318836.0	-0.609163	NaN	323.269089	0.001276	0.025568	282.842712
-0.009558	0.163141	0.118840	318836.0	-1.199441	NaN	324.797950	0.001079	0.018192	200.000000
-0.018186	-0.142444	0.123282	318836.0	-1.064771	NaN	328.830050	0.000329	0.005392	200.000000
-0.043341	-0.164271	0.074638	318836.0	0.121079	NaN	329.552825	0.000357	0.006398	NaN
-0.051496	-0.029866	0.069244	318836.0	-0.079110	NaN	329.164487	0.000563	0.011170	100.000000

DataFrames are two-dimensional labeled data structure. You can almost think of them as a Python version of table or a spreadsheet.

```
df
✓ 0.0s
```

b	vpg	pt1	parcelID	dwdt_hadv	dwdt_vadv	th0conv	qv0conv	pt1conv	cloud_edge_distance
0.065629	-0.021256	0.643482	24899.0	-0.095424	NaN	314.488336	0.011690	0.405142	NaN
0.059478	-0.036742	0.452575	24899.0	-0.075201	NaN	314.451074	0.011665	0.365690	NaN
0.040453	-0.039690	0.382675	24899.0	-0.055048	NaN	314.418553	0.011671	0.475028	NaN
0.025119	-0.032636	0.390716	24899.0	-0.027863	NaN	314.409586	0.011673	0.520689	NaN
0.011536	-0.017074	0.470834	24899.0	-0.030021	NaN	314.415951	0.011655	0.562367	NaN
...
0.043917	-0.175364	0.160459	318836.0	-0.609163	NaN	323.269089	0.001276	0.025568	282.842712
-0.009558	0.163141	0.118840	318836.0	-1.199441	NaN	324.797950	0.001079	0.018192	200.000000
-0.018186	-0.142444	0.123282	318836.0	-1.064771	NaN	328.830050	0.000329	0.005392	200.000000
-0.043341	-0.164271	0.074638	318836.0	0.121079	NaN	329.552825	0.000357	0.006398	NaN
-0.051496	-0.029866	0.069244	318836.0	-0.079110	NaN	329.164487	0.000563	0.011170	100.000000

For large DataFrames, a row with “...” means that there are additional rows not displayed. You can change this to display all rows by running `pd.set_option('display.max_rows', None)` but be careful if you have a very large dataset!

Pandas Classes

Series are one-dimensional data arrays. They are like columns in a spreadsheet. A column in a Pandas DataFrame is equivalent to a Series.

```
df.parcelID
✓ 0.0s
0      24899.0
1      24899.0
2      24899.0
3      24899.0
4      24899.0
...
202493  318836.0
202494  318836.0
202495  318836.0
202496  318836.0
202497  318836.0
Name: parcelID, Length: 202498, dtype: float64
```

A Series can be manipulated much like a NumPy array. For example, I can do

```
# Assign series to variable.
values = df.pt1

# Get the mean of the
series.
values.mean()

# Index or slice the series.
values[:4]
values[8:12]
```


Pandas Classes

The Pandas **Index** is a label that uniquely identifies data in the rows or columns of Series or DataFrames. Usually, the index is the left-most column in a DataFrame or Series.

```
df.parcelID
✓ 0.0s
```

0	24899.0
1	24899.0
2	24899.0
3	24899.0
4	24899.0
...	
202493	318836.0
202494	318836.0
202495	318836.0
202496	318836.0
202497	318836.0

```
Name: parcelID, Length: 202498, dtype: float64
```

The index does not have to be numbers. It could be, for example, a sequence of strings. But each index must uniquely identify a single row (i.e., no repeating indices).

Accessing Data

We've already seen how to access a single column or to use indexing and slicing to access certain rows in a DataFrame or Series. Two-dimensional indexing can be done using the **iloc** or **loc** methods.

The labeled columns also have integer indices.

	0	1	2	3	4	5
	x	y	z	zs	fileNum	qv
0	-40358.720	-41812.582	706.51715	231.75581	66	0.011765
1	-40204.650	-42306.574	909.54504	172.21255	67	0.011707
2	-39986.992	-42702.477	1178.01750	155.48470	68	0.011677
3	-39831.490	-43050.450	1453.22780	154.48726	69	0.011670
4	-39768.758	-43340.113	1733.62890	163.59282	70	0.011667
5	-39789.812	-43608.310	2021.89560	167.77339	71	0.011594
6	-39865.330	-43887.074	2327.00510	163.28467	72	0.010951
7	-39936.500	-44174.742	2687.64000	151.76657	73	0.010199
8	-39893.254	-44431.754	3150.64090	159.40360	74	0.009151
9	-39571.367	-44578.652	3742.26640	225.73923	75	0.007823



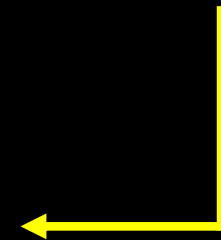
The index could also be a bunch of strings, like "a", "b", "c", etc., instead of 0, 1, 2, 3.

Accessing Data

The `iloc` method uses the integer indices to identify 2D blocks of data.

	0	1	2	3	4	5
	x	y	z	zs	fileNum	qv
0	-40358.720	-41812.582	706.51715	231.75581	66	0.011765
1	-40204.650	-42306.574	909.54504	172.21255	67	0.011707
2	-39986.992	-42702.477	1178.01750	155.48470	68	0.011677
3	-39831.490	-43050.450	1453.22780	154.48726	69	0.011670
4	-39768.758	-43340.113	1733.62890	163.59282	70	0.011667
5	-39789.812	-43608.310	2021.89560	167.77339	71	0.011594
6	-39865.330	-43887.074	2327.00510	163.28467	72	0.010951
7	-39936.500	-44174.742	2687.64000	151.76657	73	0.010199
8	-39893.254	-44431.754	3150.64090	159.40360	74	0.009151
9	-39571.367	-44578.652	3742.26640	225.73923	75	0.007823

```
# Using iloc  
df.iloc[4:8,1:3]
```

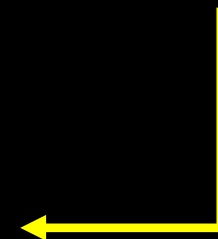


Accessing Data

The loc method uses Index and Column labels to return data. In this case, the index (row labels) happen to also be integers.

	0	1	2	3	4	5
	x	y	z	zs	fileNum	qv
0	-40358.720	-41812.582	706.51715	231.75581	66	0.011765
1	-40204.650	-42306.574	909.54504	172.21255	67	0.011707
2	-39986.992	-42702.477	1178.01750	155.48470	68	0.011677
3	-39831.490	-43050.450	1453.22780	154.48726	69	0.011670
4	-39768.758	-43340.113	1733.62890	163.59282	70	0.011667
5	-39789.812	-43608.310	2021.89560	167.77339	71	0.011594
6	-39865.330	-43887.074	2327.00510	163.28467	72	0.010951
7	-39936.500	-44174.742	2687.64000	151.76657	73	0.010199
8	-39893.254	-44431.754	3150.64090	159.40360	74	0.009151
9	-39571.367	-44578.652	3742.26640	225.73923	75	0.007823

```
# loc usage  
df.loc[4:8, 'y': 'z']
```




NOTE: The loc method does not use normal Python indexing, so the last element in a slice (8 and 'z') in the example above is included.

Accessing Data

The rows and columns need not be consecutive.

```
df.loc[4:8, ['y', 'fileNum']]
```

	0	1	2	3	4	5
	x	y	z	zs	fileNum	qv
0	-40358.720	-41812.582	706.51715	231.75581	66	0.011765
1	-40204.650	-42306.574	909.54504	172.21255	67	0.011707
2	-39986.992	-42702.477	1178.01750	155.48470	68	0.011677
3	-39831.490	-43050.450	1453.22780	154.48726	69	0.011670
4	-39768.758	-43340.113	1733.62890	163.59282	70	0.011667
5	-39789.812	-43608.310	2021.89560	167.77339	71	0.011594
6	-39865.330	-43887.074	2327.00510	163.28467	72	0.010951
7	-39936.500	-44174.742	2687.64000	151.76657	73	0.010199
8	-39893.254	-44431.754	3150.64090	159.40360	74	0.009151
9	-39571.367	-44578.652	3742.26640	225.73923	75	0.007823



	y	fileNum
4	-43340.113	70
5	-43608.310	71
6	-43887.074	72
7	-44174.742	73
8	-44431.754	74

NOTE: The loc method does not use normal Python indexing, so the last element in a slice (8 and 'z') in the example above is included.

Accessing Data

We can also use Boolean indexing to access certain rows.

```
df[df.x>0] Enter the condition as the index.  
✓ 0.0s
```

	x	y	z	zs	fileNum	qv
11139	358.4733	-70707.480	11088.736	0.0	282	0.000065
13301	928.6042	-78101.164	11452.072	0.0	287	0.000057
13302	2427.4830	-77950.375	11630.534	0.0	288	0.000057
13303	3930.6870	-77768.420	11762.759	0.0	289	0.000055
13304	5409.0850	-77576.630	11713.694	0.0	290	0.000055
...
201530	9688.8730	-24898.210	11675.580	0.0	255	0.000069
201531	11026.6110	-24695.980	11772.450	0.0	256	0.000068
201532	12368.4040	-24451.377	11802.651	0.0	257	0.000065
201533	13678.0940	-24232.494	11790.684	0.0	258	0.000064
201534	14962.0650	-24043.910	11812.292	0.0	259	0.000063

199 rows x 24 columns

Notice how all the rows contain values of x that are greater than 0. The indices are also no longer consecutive and correspond to the rows where the condition was met.

Grouping

We can also group data to isolate rows that have some common value for in a given column. This is similar to grouping with Xarray since its grouping functionality is built on Pandas.

For example, earlier we saw part of a DataFrame like the one below. A lot of the numbers in the parcelID column were the same.

```
df
✓ 0.0s
```

b	vpg	pt1	parcelID	dwdt_hadv	dwdt_vadv	th0conv	qv0conv	pt1conv	cloud_edge_distance
0.065629	-0.021256	0.643482	24899.0	-0.095424	NaN	314.488336	0.011690	0.405142	NaN
0.059478	-0.036742	0.452575	24899.0	-0.075201	NaN	314.451074	0.011665	0.365690	NaN
0.040453	-0.039690	0.382675	24899.0	-0.055048	NaN	314.418553	0.011671	0.475028	NaN
0.025119	-0.032636	0.390716	24899.0	-0.027863	NaN	314.409586	0.011673	0.520689	NaN
0.011536	-0.017074	0.470834	24899.0	-0.030021	NaN	314.415951	0.011655	0.562367	NaN
...
0.043917	-0.175364	0.160459	318836.0	-0.609163	NaN	323.269089	0.001276	0.025568	282.842712
-0.009558	0.163141	0.118840	318836.0	-1.199441	NaN	324.797950	0.001079	0.018192	200.000000
-0.018186	-0.142444	0.123282	318836.0	-1.064771	NaN	328.830050	0.000329	0.005392	200.000000
-0.043341	-0.164271	0.074638	318836.0	0.121079	NaN	329.552825	0.000357	0.006398	NaN
-0.051496	-0.029866	0.069244	318836.0	-0.079110	NaN	329.164487	0.000563	0.011170	100.000000

Grouping

We can also group data to isolate rows that have some common value for in a given column. This is similar to grouping with Xarray since its grouping functionality is built on Pandas.

For example, earlier we saw part of a DataFrame like the one below. A lot of the numbers in the parcelID column were the same.

```
df
✓ 0.0s
```

b	vpg	pt1	parcelID	dwdt_hadv	dwdt_vadv	th0conv	qv0conv	pt1conv	cloud_edge_distance
0.065629	-0.021256	0.643482	24899.0	-0.095424	NaN	314.451074	0.011665	0.365690	NaN
0.059478	-0.036742	0.452575	24899.0	-0.075201	NaN	314.418553	0.011671	0.475023	NaN
0.040453	-0.039690	0.382675	24899.0	-0.055048	NaN	314.455336	0.011671	0.320339	NaN
0.025119	-0.032636	0.390716	24899.0	-0.027863	NaN	314.455336	0.011671	0.320339	NaN
0.011536	-0.017074	0.470834	24899.0	-0.030021	NaN	314.455336	0.011671	0.320339	NaN
...
0.043917	-0.175364	0.160459	318836.0	-0.609163	NaN	323.269089	0.001276	0.025568	282.842712
-0.009558	0.163141	0.118840	318836.0	-1.199441	NaN	324.797950	0.001079	0.018192	200.000000
-0.018186	-0.142444	0.123282	318836.0	-1.064771	NaN	328.830050	0.000329	0.005392	200.000000
-0.043341	-0.164271	0.074638	318836.0	0.121079	NaN	329.552825	0.000357	0.006398	NaN
-0.051496	-0.029866	0.069244	318836.0	-0.079110	NaN	329.164487	0.000563	0.011170	100.000000

What if we want to do some sort of analysis individually on all rows with the same parcelID? Suppose, for example, we want to find the mean 'z' for each parcelID.

Grouping

We can also group data to isolate rows that have some common value for in a given column. This is similar to grouping with Xarray since its grouping functionality is built on Pandas.

For example, earlier we saw part of a DataFrame like the one below. A lot of the numbers in the parcelID column were the same.

```
df
✓ 0.0s
```

b	vpg	pt1	parcelID	dwdt_hadv	dwdt_vadv	th0conv	qv0conv	pt1conv	cloud_edge_distance
0.065629	-0.021256	0.643482	24899.0	-0.095424	NaN	314.418553	0.011671	0.477355	NaN
0.059478	-0.036742	0.452575	24899.0	-0.075201	NaN	314.418553	0.011671	0.477355	NaN
0.040453	-0.039690	0.382675	24899.0	-0.055048	NaN	314.418553	0.011671	0.477355	NaN
0.025119	-0.032636	0.390716	24899.0	-0.027863	NaN	314.409586	0.011673	0.520689	NaN
0.011536	-0.017074	0.470834	24899.0	-0.030021	NaN	314.418551	0.011655	0.662367	NaN
...
0.043917	-0.175364	0.160459	318836.0	-0.609163	NaN	323.269089	0.001276	0.025568	282.842712
-0.009558	0.163141	0.118840	318836.0	-1.199441	NaN	324.784450	0.001170	0.018182	285.000000
-0.018186	-0.142444	0.123282	318836.0	-1.064771	NaN	328.830050	0.000329	0.005392	200.000000
-0.043341	-0.164271	0.074638	318836.0	0.121079	NaN	329.552825	0.000357	0.006398	NaN
-0.051496	-0.029866	0.069244	318836.0	-0.079110	NaN	329.164487	0.000563	0.011170	100.000000

What if we want to do some sort of analysis individually on all rows with the same parcelID? Suppose, for example, we want to find the mean 'z' for each parcelID.

```
df.groupby('parcelID').mean()
```

```
df.groupby('parcelID').mean()
```

✓ 0.0s

parcelID	x	y	z	zs	fileNum	qv
16765.0	-37160.249815	-104023.907370	2304.284861	0.000000	177.0	0.009123
17033.0	-38731.965222	-70083.748778	3035.993350	182.163472	55.0	0.008092
20231.0	-32704.738042	-78866.216083	2693.078910	294.934056	122.5	0.009104
20257.0	-34553.879727	-97910.690409	3220.694971	22.766547	139.5	0.008157
20352.0	-34097.087154	-98608.689769	3255.139588	8.480120	186.5	0.007743
...
319235.0	-22501.192545	13509.911394	3053.545361	686.153194	195.0	0.008114
319498.0	-25487.516296	18070.956185	4886.403961	730.888861	163.5	0.005576
319528.0	-24231.647467	17097.080733	3608.515080	879.184701	150.0	0.007587
319644.0	-26823.209846	13413.663846	3425.787808	941.946672	163.0	0.007704
319774.0	-23387.236345	13343.110707	3372.333743	833.373343	172.0	0.008435

Note how parcelID is the index of the resulting DataFrame.

Each value is the mean for each column over all rows in the original DataFrame (`df`) with the parcelID listed on the left column.

6552 rows x 23 columns

Pivoting

In Pandas, pivoting is a technique that can be used to better visualize certain data. Consider the following example (generated by ChatGPT):

```
import pandas as pd

# Sample data
data = {
    'Date': ['2023-01-01', '2023-01-01', '2023-01-02', '2023-01-02', '2023-01-03', '2023-01-03'],
    'City': ['New York', 'Los Angeles', 'New York', 'Los Angeles', 'New York', 'Los Angeles'],
    'Temperature': [32, 75, 30, 78, 28, 80]
}

df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)
```

Original DataFrame:

	Date	City	Temperature
0	2023-01-01	New York	32
1	2023-01-01	Los Angeles	75
2	2023-01-02	New York	30
3	2023-01-02	Los Angeles	78
4	2023-01-03	New York	28
5	2023-01-03	Los Angeles	80

Pivoting

Notice how we have repeat dates and cities. Perhaps we want to give each city its own column and we want to use the date as the index.

```
pivot_df = df.pivot(index='Date', columns='City', values='Temperature')
print("Pivoted DataFrame:")
print(pivot_df)
```

```
Pivoted DataFrame:
City          Los Angeles  New York
Date
2023-01-01             75         32
2023-01-02             78         30
2023-01-03             80         28
```

In this new DataFrame object, how would you access the temperature on Jan. 1, 2023 in New York using loc or iloc?