4. Wrangling DataFrames

Questions

- How can you select individual columns or rows from a `DataFrame`?
- How can you subset a `DataFrame`?
- How can you sort a `DataFrame`?

Objectives

- Learn how to select specific columns or rows from a `DataFrame`.
- Learn how to select rows based on conditions.
- Learn how to sort a `DataFrame`'s rows or columns.

What is Data Wrangling?

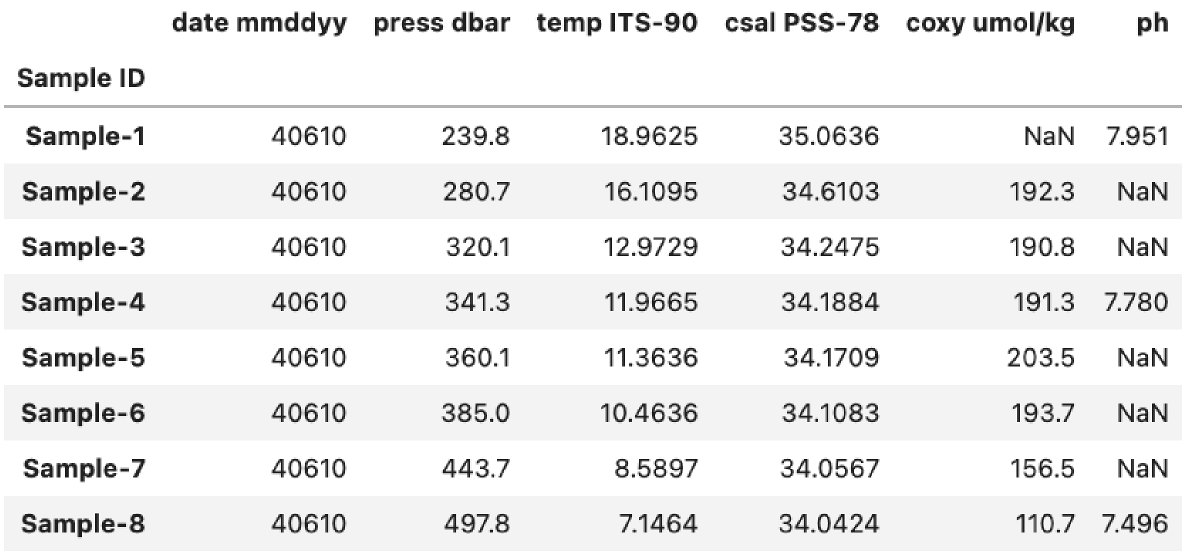
Data wrangling is the process of cleaning, transforming, and mapping data from various sources into a format that can be used for analysis or modeling. It involves various tasks such as handling missing or duplicate data, handling outliers, and combining data from different sources.

Subsetting, Selection and Sorting a DataFrame

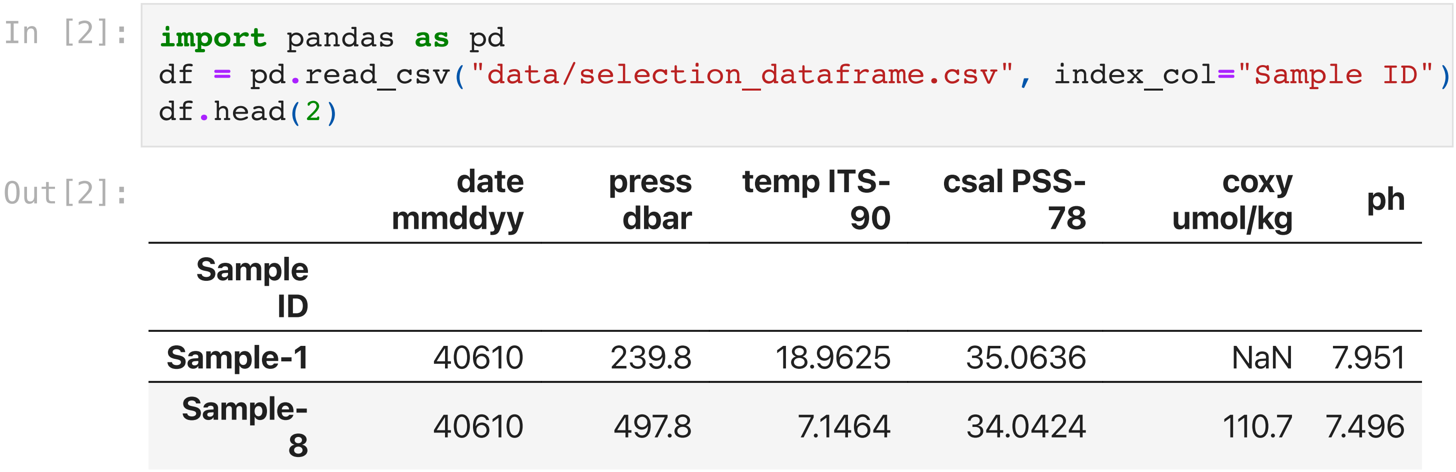
- Selecting, subsetting, and sorting are essential aspects of data wrangling.
- Subsetting refers to extracting a portion of the data. For example,
    - Extract only columns 1 and 2 of the `DataFrame`
    - Give me only the rows with an even index (i.e., 2, 4, 6, ...)
- Selecting consists of extracting rows based on some condition. For example,
    - Give me all the value for which the temperature is larger than 20
    - Give me all samples that start with `ABC`, etc.
- Sorting consists of reordering data using either the index or the values of one or more columns.

Subsetting

To illustrate subsetting, selection and sorting, we will be using the the following `DataFrame` stored in a value called `df`,



	date mmddyy	press dbar	temp ITS-90	csal PSS-78	coxy umol/kg	ph
Sample ID						
Sample-1	40610	239.8	18.9625	35.0636	NaN	7.951
Sample-2	40610	280.7	16.1095	34.6103	192.3	NaN
Sample-3	40610	320.1	12.9729	34.2475	190.8	NaN
Sample-4	40610	341.3	11.9665	34.1884	191.3	7.780
Sample-5	40610	360.1	11.3636	34.1709	203.5	NaN
Sample-6	40610	385.0	10.4636	34.1083	193.7	NaN
Sample-7	40610	443.7	8.5897	34.0567	156.5	NaN
Sample-8	40610	497.8	7.1464	34.0424	110.7	7.496



```
In [2]: import pandas as pd
        df = pd.read_csv("data/selection_dataframe.csv", index_col="Sample ID")
        df.head(2)
```

```
Out[2]:
                 date    press  temp ITS-  csal PSS-       coxy     ph
               mmddyy     dbar         90         78  umol/kg
   Sample
       ID
 Sample-1       40610    239.8    18.9625    35.0636        NaN  7.951
  Sample-
        8       40610    497.8     7.1464    34.0424      110.7  7.496
```

Subsetting Columns

- Selecting colums is trivial and follows the same notation as that used to with Python lists
- Selecting the `‘ph’` can be written as:

```
In [3]: df['ph']
```

```
Out[3]: Sample ID
        Sample-1    7.951
        Sample-8    7.496
        Sample-2      NaN
        Sample-4    7.780
        Sample-6      NaN
        Sample-5      NaN
        Sample-7      NaN
        Sample-3      NaN
        Name: ph, dtype: float64
```

Selecting Rows

- Subsetting on rows uses the `.loc` or `.iloc` operators
- Both `.loc` or `.iloc` require information abour the rows and column.
- Example, to extract subset consisting of the row 0 and column 2, we would write:

```
df.iloc[0, 2]
```

```
In [4]: df.iloc[0, 2]
```

```
Out[4]: 18.9625
```

Subsetting format using `iloc`

- In general, subsetting rows using `.loc` or `.iloc` can take of the following forms
- `df.iloc[row_index(es), col_index(es)]`
- As the notation indicates, we can provide a single or list of index.

Example:

```
df.iloc[1, 2]
```

```
In [5]: df.iloc[1, 2]
```

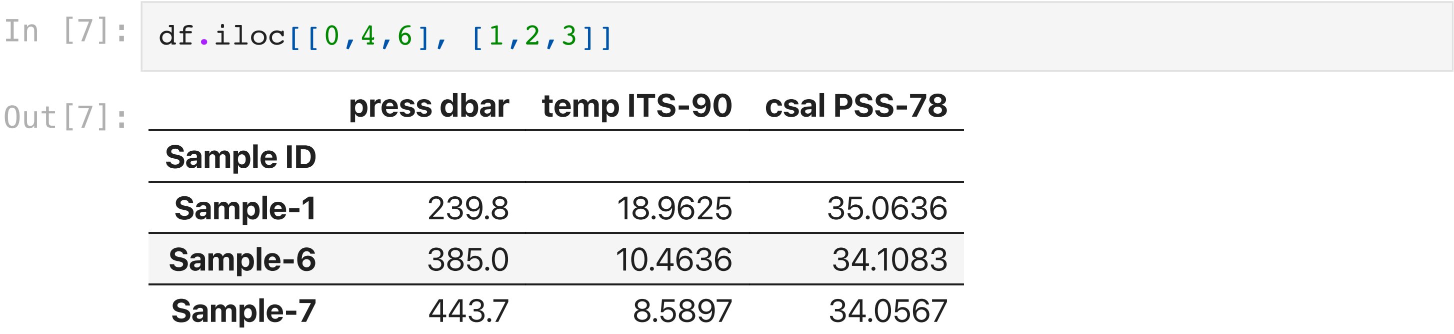
```
Out[5]: 7.1464
```

In [6]:

```
df.iloc[1, [1,2,3]]
```

Out[6]:

```
press dbar      497.8000
temp ITS-90       7.1464
csal PSS-78      34.0424
Name: Sample-8, dtype: float64
```



```
In [7]: df.iloc[[0,4,6], [1,2,3]]
```

```
Out[7]:
            press dbar  temp ITS-90  csal PSS-78
 Sample ID
  Sample-1       239.8      18.9625      35.0636
  Sample-6       385.0      10.4636      34.1083
  Sample-7       443.7       8.5897      34.0567
```

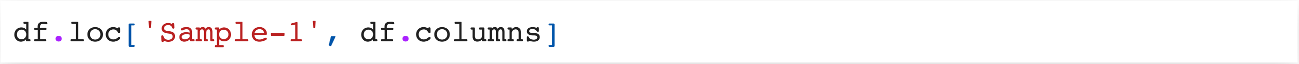
```
In [8]:  df.loc['Sample-1', df.columns]
```

```
Out[8]:  date mmddyy        40610.0000
         press dbar          239.8000
         temp ITS-90          18.9625
         csal PSS-78          35.0636
         coxy umol/kg             NaN
         ph                    7.9510
         Name: Sample-1, dtype: float64
```

Subsetting format using `loc`

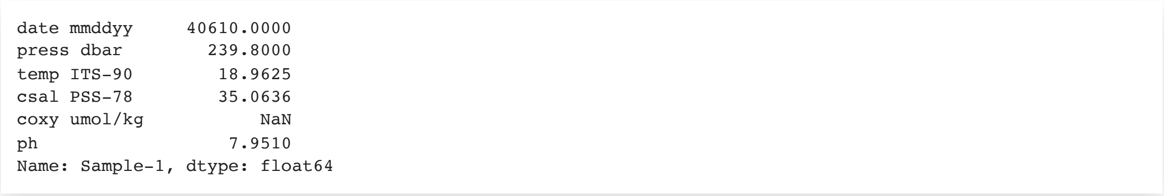
- in general subsessing rows using `.loc` is written as
- `df.loc[row_label(s), column_label(es)]`
- As the notation indicates, we can provide a single or list of labels

INPUT



```
df.loc['Sample-1', df.columns]
```

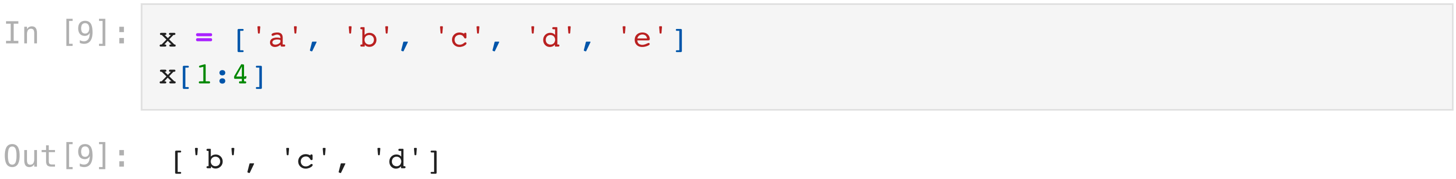
Output



```
date mmddyy     40610.0000
press dbar        239.8000
temp ITS-90        18.9625
csal PSS-78        35.0636
coxy umol/kg           NaN
ph                  7.9510
Name: Sample-1, dtype: float64
```

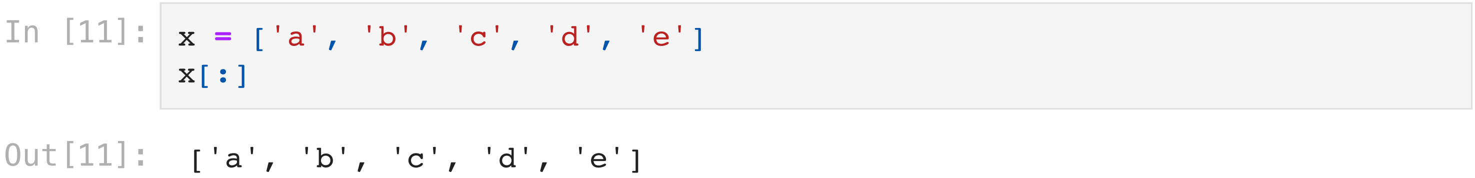
The `:` Operator In Python (Refresher)

- In python the `:` operator is used for slicing ordered collections (e.g. lists).
- "`:`" can optionally take left and right operators:
    - `X:Y` means from X to and not including Y
    - `X:` means from X to the end of the list
    - `:Y` means form the beginning all the way to and not including Y
    - `:` the entire collection



```
In [9]: x = ['a', 'b', 'c', 'd', 'e']
        x[1:4]

Out[9]: ['b', 'c', 'd']
```

```
In [10]: x = ['a', 'b', 'c', 'd', 'e']
         x[1:]

Out[10]: ['b', 'c', 'd', 'e']
```



```
In [11]: x = ['a', 'b', 'c', 'd', 'e']
         x[:]

Out[11]: ['a', 'b', 'c', 'd', 'e']
```

In [12]:

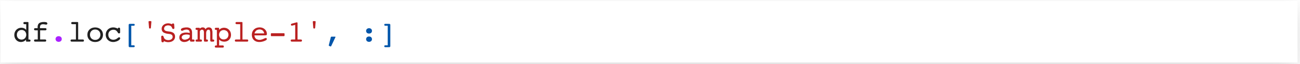
```
df.columns
```

Out[12]:

```
Index(['date mmddyy', 'press dbar', 'temp ITS-90', 'csal PSS-7
8',
       'coxy umol/kg', 'ph'],
      dtype='object')
```

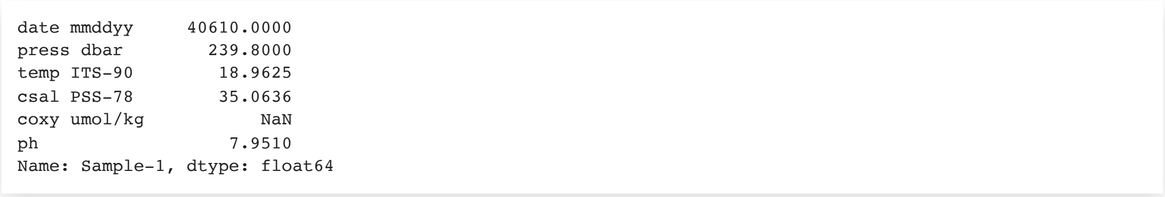
Indexing Using Slicing operator

INPUT

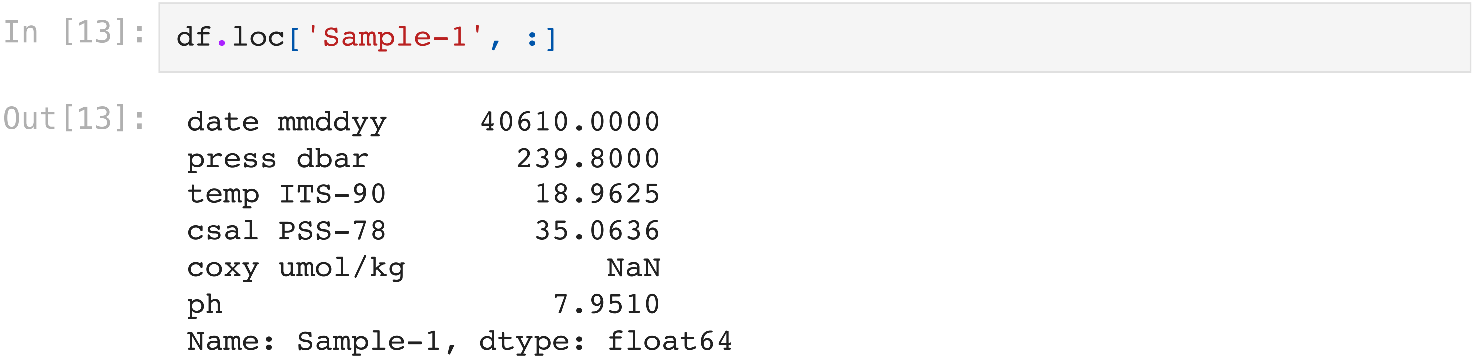


```
df.loc['Sample-1', :]
```

Output



```
date mmddyy     40610.0000
press dbar        239.8000
temp ITS-90        18.9625
csal PSS-78        35.0636
coxy umol/kg           NaN
ph                  7.9510
Name: Sample-1, dtype: float64
```



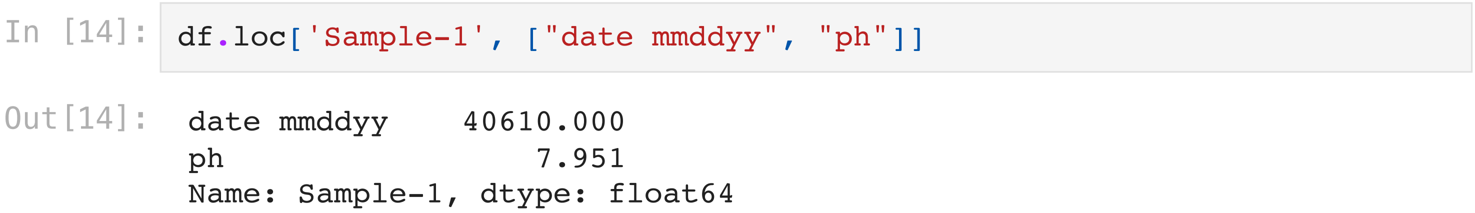
```
In [13]: df.loc['Sample-1', :]
```

```
Out[13]: date mmddyy     40610.0000
         press dbar        239.8000
         temp ITS-90        18.9625
         csal PSS-78        35.0636
         coxy umol/kg           NaN
         ph                  7.9510
         Name: Sample-1, dtype: float64
```

Difference Between `.loc` and `.iloc`

Both used to subset of a `DataFrame`.

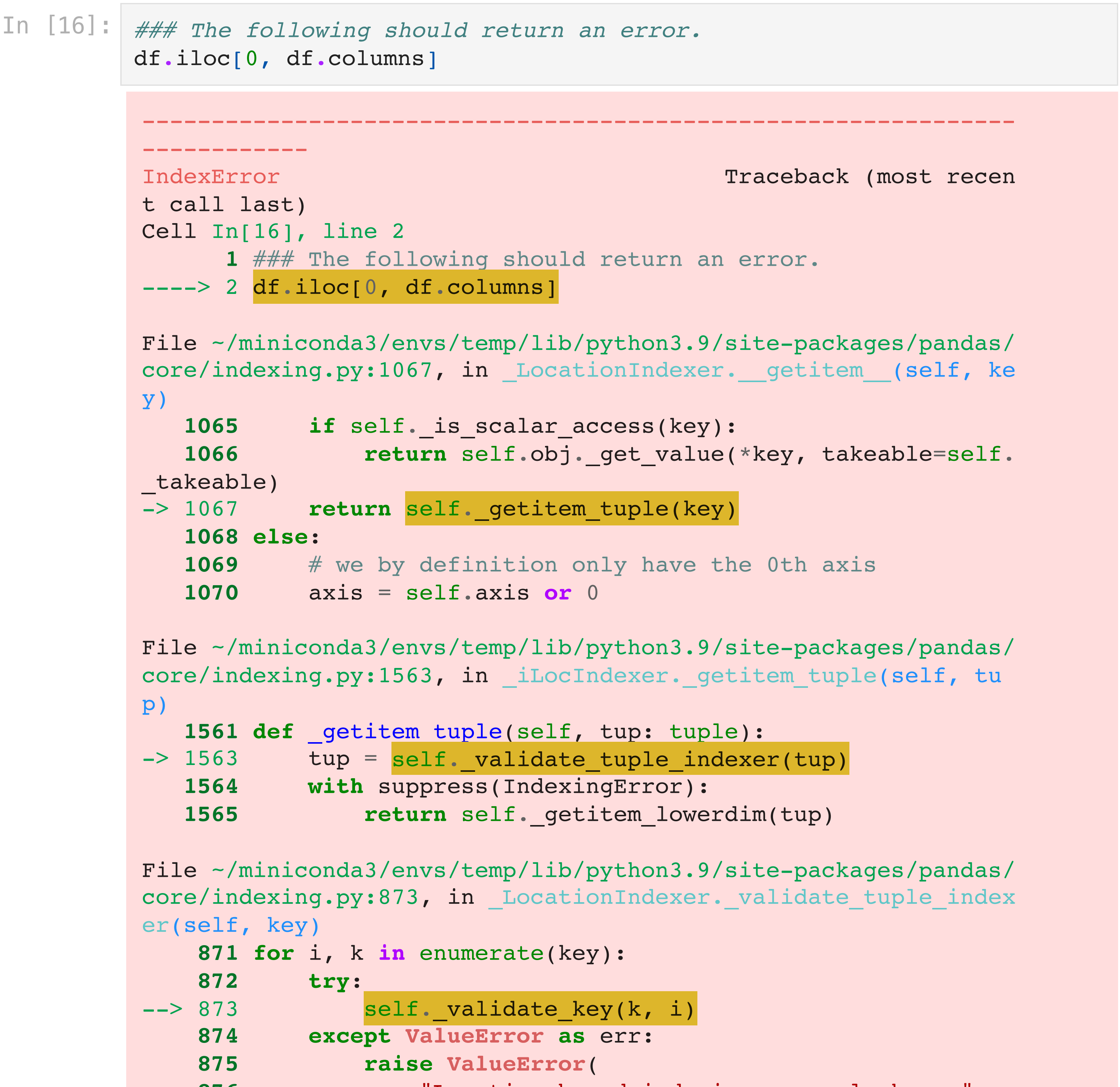
- `.loc` relies on the names of the indexes and headers
- `.iloc` relies instead on the index and header number
- `df.loc['Sample-1', ["date mmddyy", "ph"]]` means row index by the label 'Sample-1' and columns "date mmddyy", "ph"
- `df.iloc[0, [0,1,2]]` means row 0 and columns 0, 1 and 2
- You cannot mix indexes and labels and the following is incorrect
    - `df.iloc[0, ["date mmddyy", "ph"]]` returns error
    - `df.iloc[0, df.columns]` returns error



```
In [14]: df.loc['Sample-1', ["date mmddyy", "ph"]]

Out[14]: date mmddyy    40610.000
         ph                 7.951
         Name: Sample-1, dtype: float64
```

```
In [15]: df.iloc[0, [0,1,2]]
```

```
Out[15]: date mmddyy    40610.0000
         press dbar       239.8000
         temp ITS-90       18.9625
         Name: Sample-1, dtype: float64
```



In [16]:

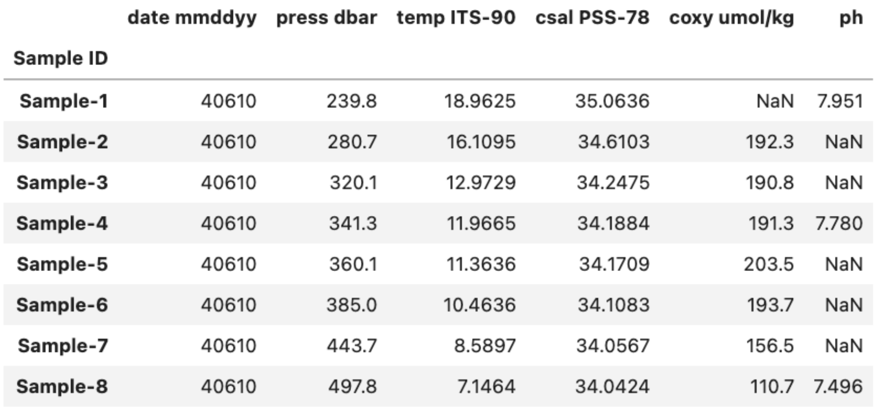
```
### The following should return an error.
df.iloc[0, df.columns]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[16], line 2
      1 ### The following should return an error.
----> 2 df.iloc[0, df.columns]

File ~/miniconda3/envs/temp/lib/python3.9/site-packages/pandas/core/indexing.py:1067, in _LocationIndexer.__getitem__(self, key)
   1065     if self._is_scalar_access(key):
   1066         return self.obj._get_value(*key, takeable=self._takeable)
-> 1067     return self._getitem_tuple(key)
   1068 else:
   1069     # we by definition only have the 0th axis
   1070     axis = self.axis or 0

File ~/miniconda3/envs/temp/lib/python3.9/site-packages/pandas/core/indexing.py:1563, in _iLocIndexer._getitem_tuple(self, tup)
   1561 def _getitem_tuple(self, tup: tuple):
-> 1563     tup = self._validate_tuple_indexer(tup)
   1564     with suppress(IndexingError):
   1565         return self._getitem_lowerdim(tup)

File ~/miniconda3/envs/temp/lib/python3.9/site-packages/pandas/core/indexing.py:873, in _LocationIndexer._validate_tuple_indexer(self, key)
    871 for i, k in enumerate(key):
    872     try:
--> 873         self._validate_key(k, i)
    874     except ValueError as err:
    875         raise ValueError(
    877                 f"[{self._valid_types}] types"
    878             ) from err

File ~/miniconda3/envs/temp/lib/python3.9/site-packages/pandas/
core/indexing.py:1477, in _iLocIndexer._validate_key(self, key,
axis)
   1475 # check that the key has a numeric dtype
   1476 if not is_numeric_dtype(arr.dtype):
-> 1477     raise IndexError(f".iloc requires numeric indexers,
got {arr}")
   1479 # check that the key does not exceed the maximum size o
f the index
   1480 if len(arr) and (arr.max() >= len_axis or arr.min() < -
len_axis):

IndexError: .iloc requires numeric indexers, got ['date mmddyy'
'press dbar' 'temp ITS-90' 'csal PSS-78' 'coxy umol/kg'
 'ph']
```

Selecting using Conditional Expressions

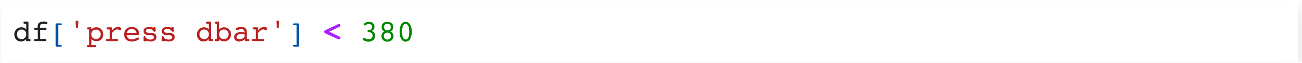
In Pandas, comparison operations (“<” , “>” , “==” , “>=” , “<=” , “!=”) works in a similar as in Python

- An applied series, then a series of booleans (True or False) are returned



	date mmddyy	press dbar	temp ITS-90	csal PSS-78	coxy umol/kg	ph
Sample ID						
Sample-1	40610	239.8	18.9625	35.0636	NaN	7.951
Sample-2	40610	280.7	16.1095	34.6103	192.3	NaN
Sample-3	40610	320.1	12.9729	34.2475	190.8	NaN
Sample-4	40610	341.3	11.9665	34.1884	191.3	7.780
Sample-5	40610	360.1	11.3636	34.1709	203.5	NaN
Sample-6	40610	385.0	10.4636	34.1083	193.7	NaN
Sample-7	40610	443.7	8.5897	34.0567	156.5	NaN
Sample-8	40610	497.8	7.1464	34.0424	110.7	7.496

- What does the following return?



```
df['press dbar'] < 380
```

In [17]:

```
df['press dbar'] < 380
```

Out[17]:

```
Sample ID
Sample-1     True
Sample-8    False
Sample-2     True
Sample-4     True
Sample-6    False
Sample-5     True
Sample-7    False
Sample-3     True
Name: press dbar, dtype: bool
```

Subsetting Based on Conditional Expressions

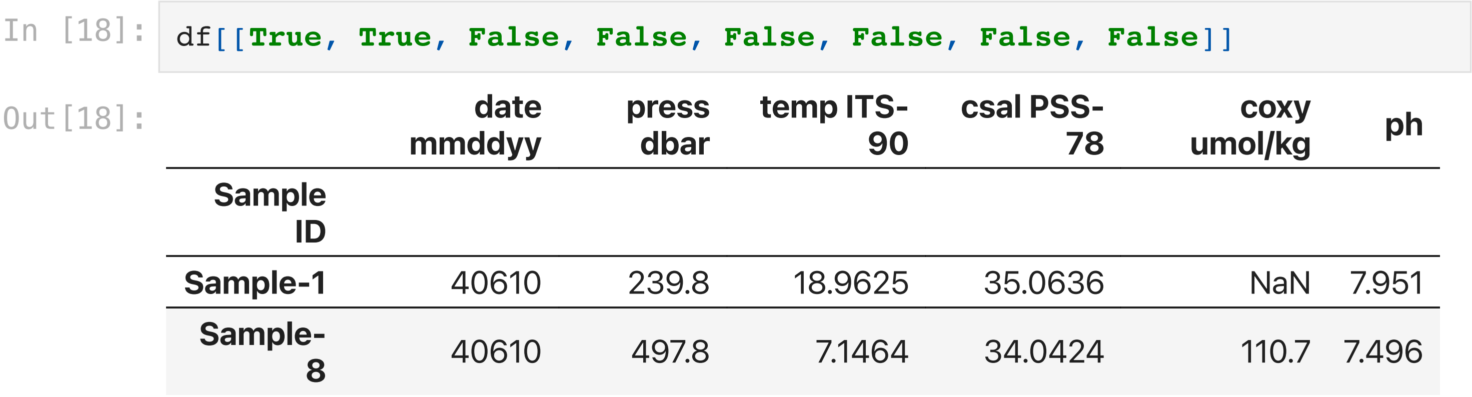
- It turns out that we can also subset a `DataFrame` using a list of booleans. For example:

INPUT

```
df[[True, True, False, False, False, False, False, False]]
```

OUTPUT

```
date mmddyy     press dbar      temp ITS-90     csal PSS-78     coxy
umol/kg ph
Sample ID
Sample-1        40610   239.8   18.9625 35.0636 NaN     7.951
Sample-2        40610   280.7   16.1095 34.6103 192.3   NaN
```



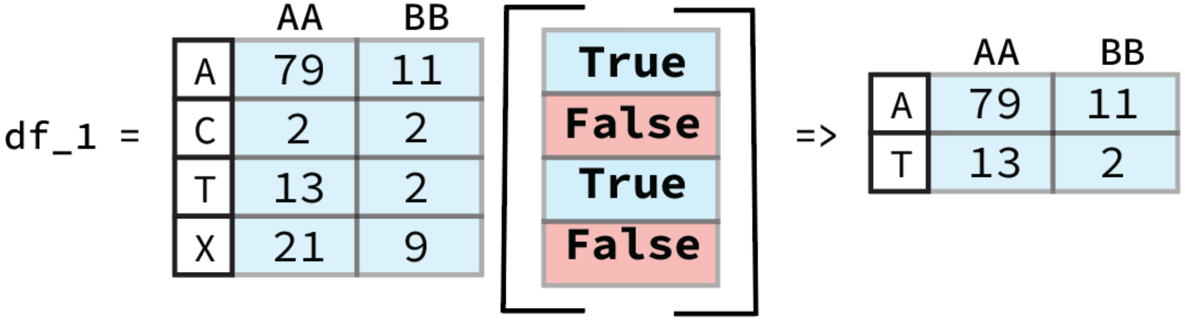
In [18]:

```
df[[True, True, False, False, False, False, False, False]]
```

Out[18]:

	date mmddyy	press dbar	temp ITS-90	csal PSS-78	coxy umol/kg	ph
Sample ID						
Sample-1	40610	239.8	18.9625	35.0636	NaN	7.951
Sample-8	40610	497.8	7.1464	34.0424	110.7	7.496

- The python input means return the first two row (associted with True) and ingnore the lines associated wiht False



	AA	BB
A	79	11
C	2	2
T	13	2
X	21	9

True
False
True
False

	AA	BB
A	79	11
T	13	2

- Note that in when subsetting using Booleans, we don't need to use either `.loc` or `.loc`

Subsetting Based on Conditional Expressions - Cont’d

- Conditional expression can be used directly in a `DataFrame`

INPUT

```
df[ df['press dbar'] < 380 ]
```

OUTPUT

```
date mmddyy     press dbar     temp ITS-90     csal PSS-78     coxy
umol/kg ph
Sample ID
Sample-1        40610    239.8    18.9625 35.0636 NaN      7.951
Sample-2        40610    280.7    16.1095 34.6103 192.3    NaN
Sample-3        40610    320.1    12.9729 34.2475 190.8    NaN
Sample-4        40610    341.3    11.9665 34.1884 191.3    7.780
Sample-5        40610    360.1    11.3636 34.1709 203.5    NaN
```

In [19]:

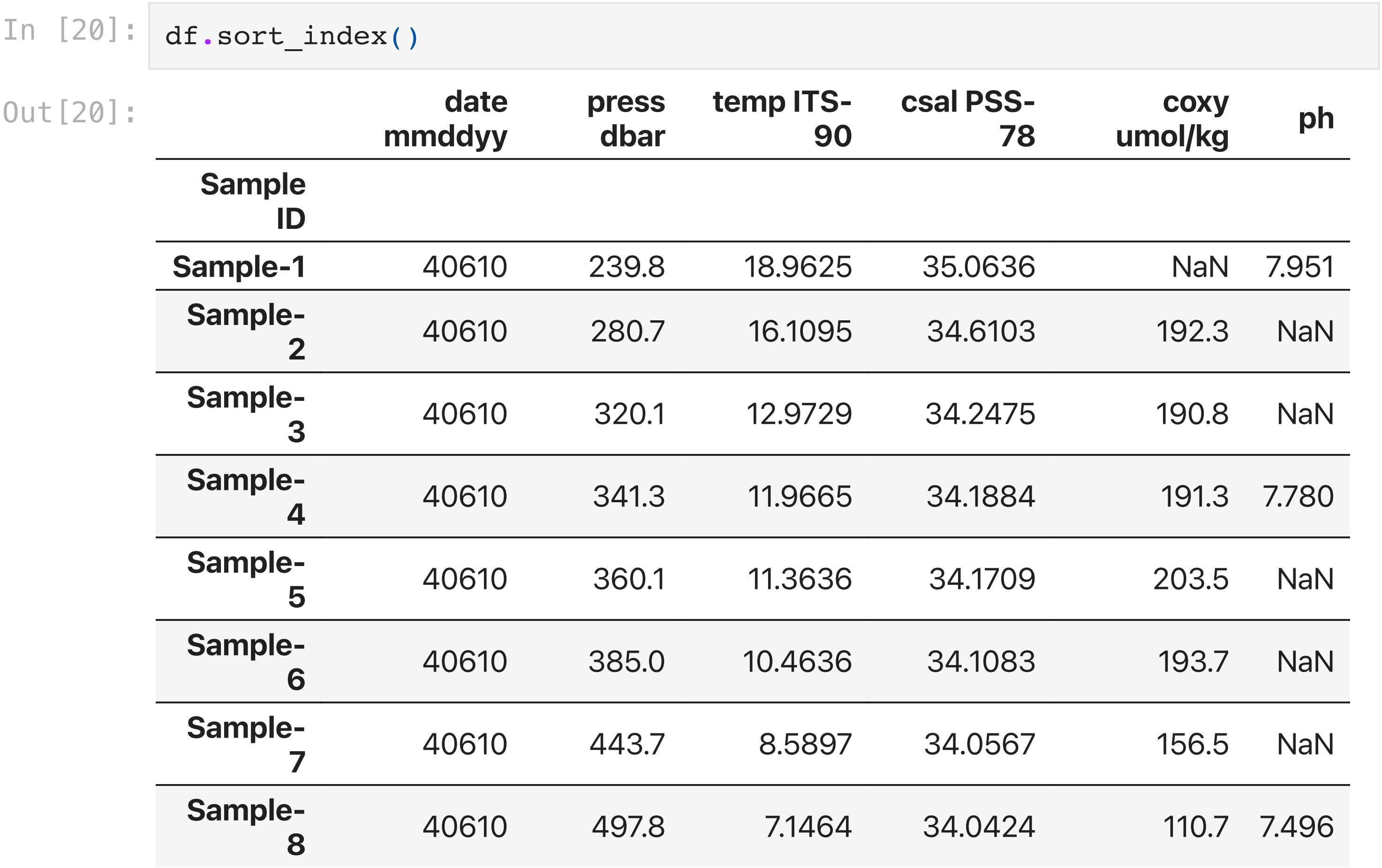
```
df[ df['press dbar'] < 380 ]
```

Out[19]:

	date mmddyy	press dbar	temp ITS-90	csal PSS-78	coxy umol/kg	ph
Sample ID						
Sample-1	40610	239.8	18.9625	35.0636	NaN	7.951
Sample-2	40610	280.7	16.1095	34.6103	192.3	NaN
Sample-4	40610	341.3	11.9665	34.1884	191.3	7.780
Sample-5	40610	360.1	11.3636	34.1709	203.5	NaN
Sample-3	40610	320.1	12.9729	34.2475	190.8	NaN

Sorting

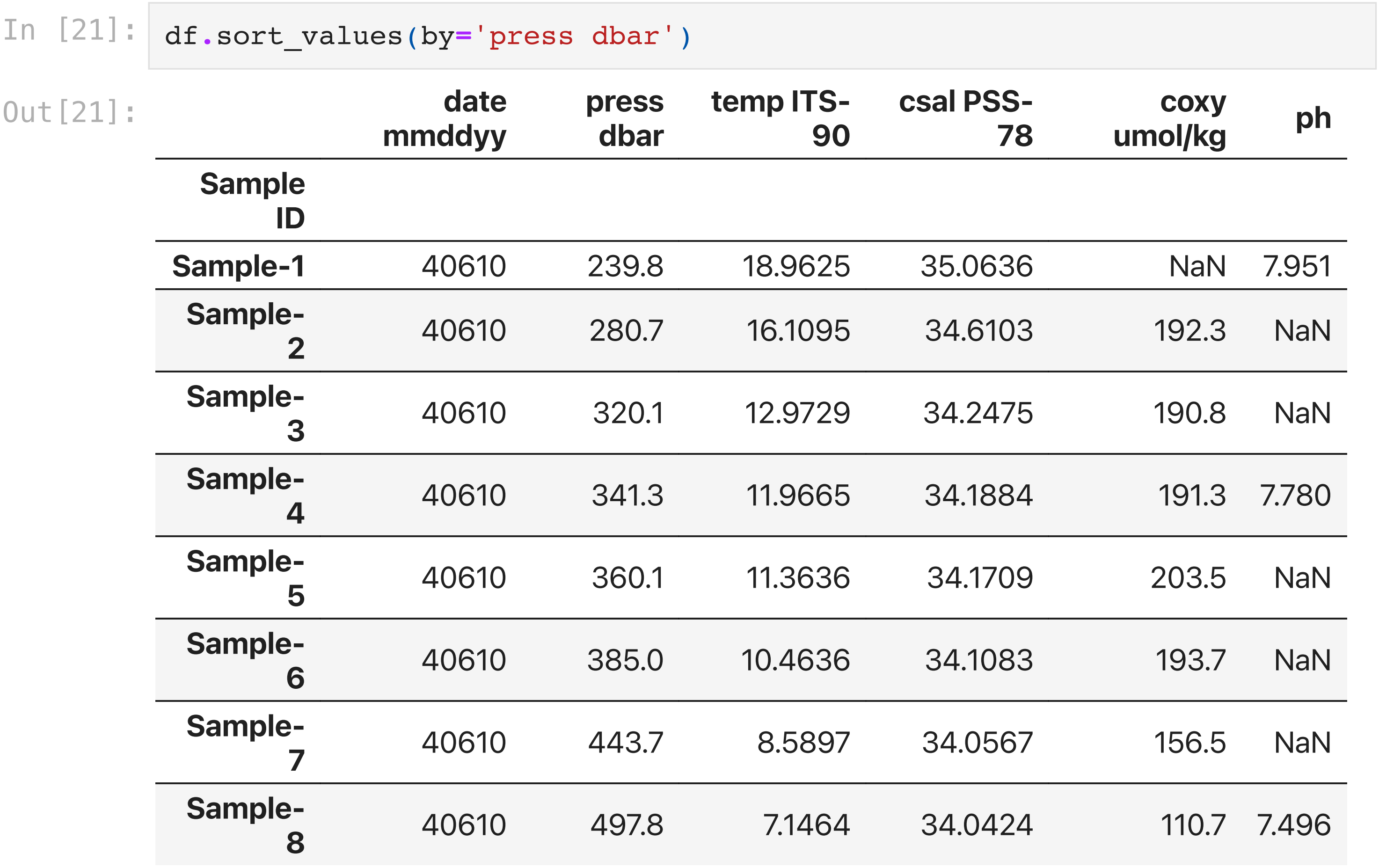
- Sorting is important aspect of data wrangling. For example:
    - In data analysis: sorting is used to identify patterns, trends, and outliers in the data.
    - in data cleaning, sorting can help to identify and remove duplicate entries, missing values, and other errors in the data.
    - etc.
- Data can sorted on the index using `sort_index()`
- Data can sorted on of the columns using `sort_values()`
    - Can take a single column or a list of columns to sort
- By default index and vlaues are sorted in ascending order but the behavior can be changed by setting `ascending=True` as parameter to the function.



```
In [20]: df.sort_index()
```

Out[20]:

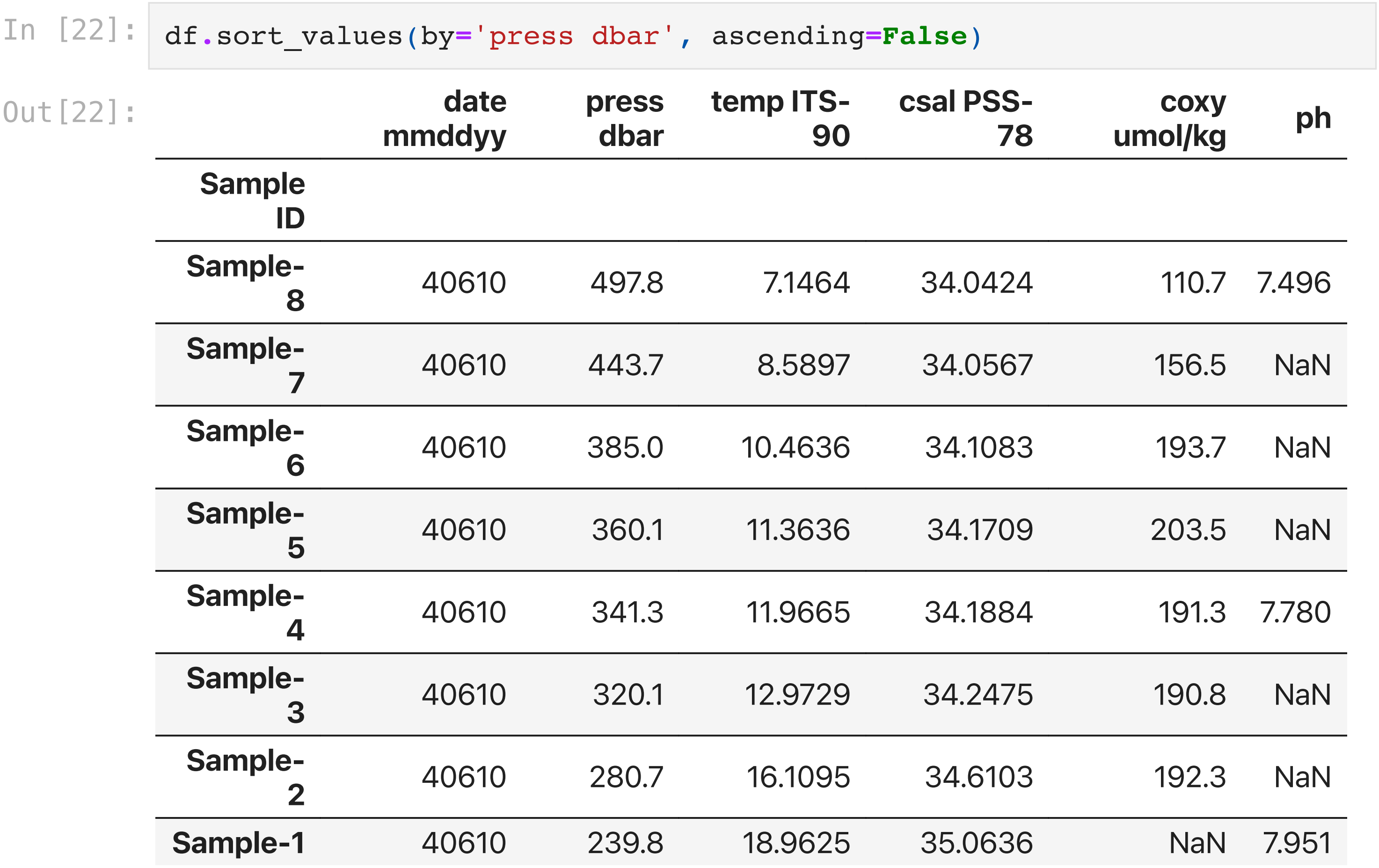
	date mmddyy	press dbar	temp ITS-90	csal PSS-78	coxy umol/kg	ph
Sample ID						
Sample-1	40610	239.8	18.9625	35.0636	NaN	7.951
Sample-2	40610	280.7	16.1095	34.6103	192.3	NaN
Sample-3	40610	320.1	12.9729	34.2475	190.8	NaN
Sample-4	40610	341.3	11.9665	34.1884	191.3	7.780
Sample-5	40610	360.1	11.3636	34.1709	203.5	NaN
Sample-6	40610	385.0	10.4636	34.1083	193.7	NaN
Sample-7	40610	443.7	8.5897	34.0567	156.5	NaN
Sample-8	40610	497.8	7.1464	34.0424	110.7	7.496



```
In [21]: df.sort_values(by='press dbar')
```

Out[21]:

	date mmddyy	press dbar	temp ITS-90	csal PSS-78	coxy umol/kg	ph
Sample ID						
Sample-1	40610	239.8	18.9625	35.0636	NaN	7.951
Sample-2	40610	280.7	16.1095	34.6103	192.3	NaN
Sample-3	40610	320.1	12.9729	34.2475	190.8	NaN
Sample-4	40610	341.3	11.9665	34.1884	191.3	7.780
Sample-5	40610	360.1	11.3636	34.1709	203.5	NaN
Sample-6	40610	385.0	10.4636	34.1083	193.7	NaN
Sample-7	40610	443.7	8.5897	34.0567	156.5	NaN
Sample-8	40610	497.8	7.1464	34.0424	110.7	7.496

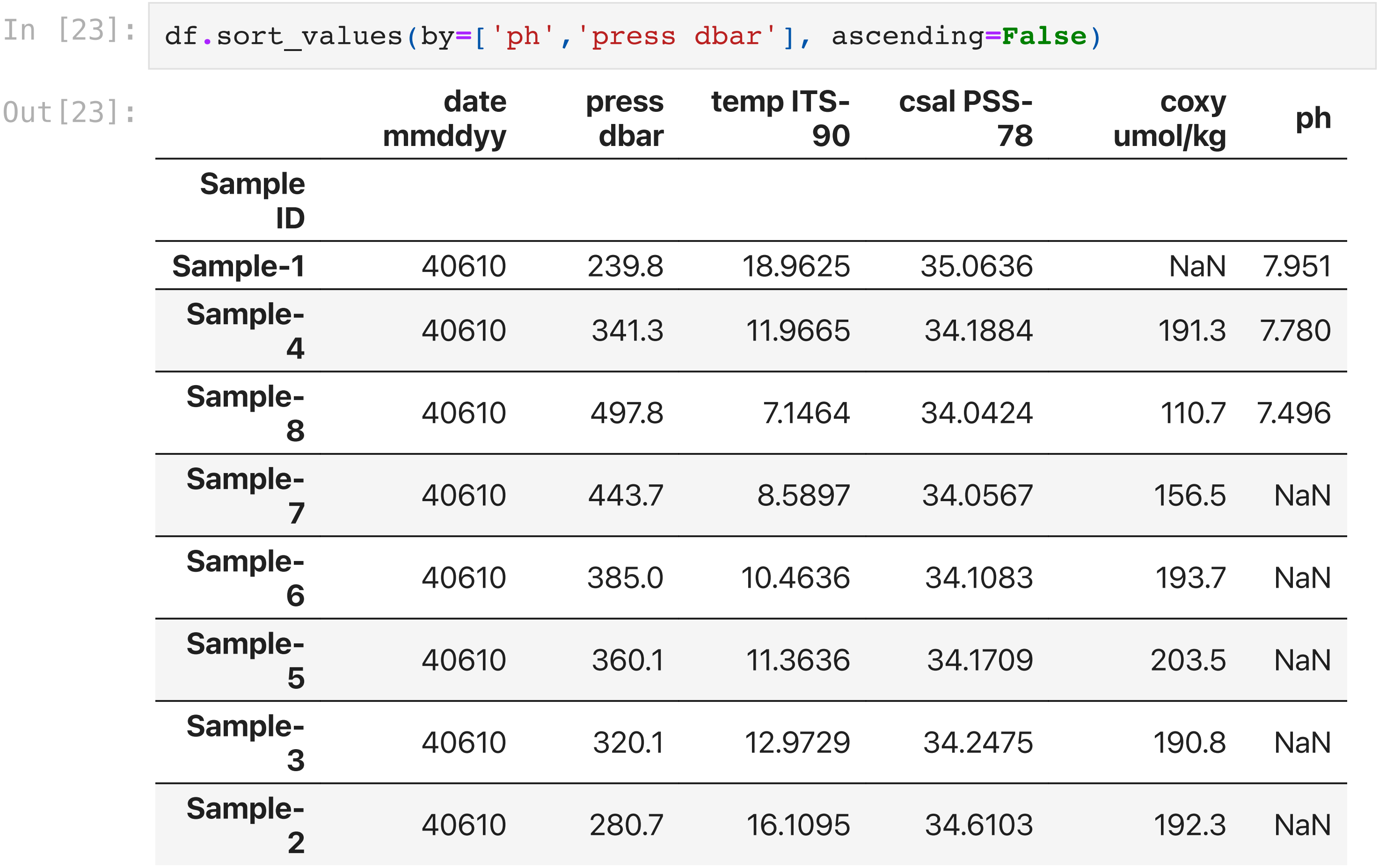


In [22]:

```
df.sort_values(by='press dbar', ascending=False)
```

Out[22]:

	date mmddyy	press dbar	temp ITS-90	csal PSS-78	coxy umol/kg	ph
Sample ID						
Sample-8	40610	497.8	7.1464	34.0424	110.7	7.496
Sample-7	40610	443.7	8.5897	34.0567	156.5	NaN
Sample-6	40610	385.0	10.4636	34.1083	193.7	NaN
Sample-5	40610	360.1	11.3636	34.1709	203.5	NaN
Sample-4	40610	341.3	11.9665	34.1884	191.3	7.780
Sample-3	40610	320.1	12.9729	34.2475	190.8	NaN
Sample-2	40610	280.7	16.1095	34.6103	192.3	NaN
Sample-1	40610	239.8	18.9625	35.0636	NaN	7.951



```
In [23]: df.sort_values(by=['ph','press dbar'], ascending=False)
```

Out[23]:

	date mmddyy	press dbar	temp ITS-90	csal PSS-78	coxy umol/kg	ph
Sample ID						
Sample-1	40610	239.8	18.9625	35.0636	NaN	7.951
Sample-4	40610	341.3	11.9665	34.1884	191.3	7.780
Sample-8	40610	497.8	7.1464	34.0424	110.7	7.496
Sample-7	40610	443.7	8.5897	34.0567	156.5	NaN
Sample-6	40610	385.0	10.4636	34.1083	193.7	NaN
Sample-5	40610	360.1	11.3636	34.1709	203.5	NaN
Sample-3	40610	320.1	12.9729	34.2475	190.8	NaN
Sample-2	40610	280.7	16.1095	34.6103	192.3	NaN

Key Points

- Select columns by using `["column name"]` or rows by using the `loc` attribute.
- Sort based on values in a column by using the `sort_values` method.

Exercise

Try it yourself! Going back to our `20_sales_records.xlsx` file, idenitfy which orders are `Online` and `High Priority`

- Read the first couple rows to get a sense of the data. Which column reflects `Online` or `Offline` Status.
- A `High Priority` order is denoted by `H` in one of the columns. Identify which column.
- HINT: Use the loc method