pandas

Lecture 08

Dr. Colin Rundel

Sta 663 - Spring 2023

pandas

pandas is an implementation of data frames in Python - it takes much of its inspiration from R and NumPy.

> pandas aims to be the fundamental high-level building block for doing practical, real world data analysis in Python. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis / manipulation tool available in any language.

Key features:

- DataFrame and Series (column) object classes
- Reading and writing tabular data
- Data munging (filtering, grouping, summarizing, joining, etc.)
- Data reshaping

DataFrame

- Just like R a DataFrame is a collection of vectors with a common length
- Column types can be heterogeneous
- Both columns and rows can have names

```
iris = pd.read_csv("data/iris.csv")
type(iris)
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
iris
```

```
     Sepal.Length  Sepal.Width  Petal.Length  Petal.Width    Species
0             5.1          3.5           1.4          0.2     setosa
1             4.9          3.0           1.4          0.2     setosa
2             4.7          3.2           1.3          0.2     setosa
3             4.6          3.1           1.5          0.2     setosa
4             5.0          3.6           1.4          0.2     setosa
..            ...          ...           ...          ...        ...
145           6.7          3.0           5.2          2.3  virginica
146           6.3          2.5           5.0          1.9  virginica
147           6.5          3.0           5.2          2.0  virginica
148           6.2          3.4           5.4          2.3  virginica
149           5.9          3.0           5.1          1.8  virginica
```

Series

The columns of a DataFrame are constructed as Series - a 1d array like object containing values of the same type (similar to an ndarray).

```
pd.Series([1,2,3,4])
```

```
0    1
1    2
2    3
3    4
dtype: int64
```

```
pd.Series(["C","B","A"])
```

```
0    C
1    B
2    A
dtype: object
```

```
pd.Series([True])
```

```
0    True
dtype: bool
```

```
pd.Series(range(5))
```

```
0    0
1    1
2    2
3    3
4    4
dtype: int64
```

```
pd.Series([1,"A",True])
```

```
0       1
1       A
2    True
dtype: object
```

Series methods

Once constructed the components of a series can be accessed via `array` and `index` attributes.

```
s = pd.Series([4,2,1,3])
```

```
s
```

```
0    4
1    2
2    1
3    3
dtype: int64
```

```
s.array
```

```
<PandasArray>
[4, 2, 1, 3]
Length: 4, dtype: int64
```

```
s.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

An index can also be explicitly provided when constructing a Series,

```
t = pd.Series([4,2,1,3], index=["a","b","c","d"])
```

```
t
```

```
a    4
b    2
c    1
d    3
dtype: int64
```

```
t.array
```

```
<PandasArray>
[4, 2, 1, 3]
Length: 4, dtype: int64
```

```
t.index
```

```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

Series + NumPy

Series objects are compatible with NumPy like functions (vectorized)

```python
t = pd.Series([4,2,1,3], index=["a","b","c","d"])
```

```python
t + 1
```

```
a    5
b    3
c    2
d    4
dtype: int64
```

```python
t / 2 + 1
```

```
a    3.0
b    2.0
c    1.5
d    2.5
dtype: float64
```

```python
np.log(t)
```

```
a    1.386294
b    0.693147
c    0.000000
d    1.098612
dtype: float64
```

```python
np.exp(-t**2/2)
```

```
a    0.000335
b    0.135335
c    0.606531
d    0.011109
dtype: float64
```

Series indexing

Series can be indexed in the same was as NumPy arrays with the addition of being able to use index label(s) when selecting elements.

```
t = pd.Series([4,2,1,3], index=["a","b","c","d"])
```

```
t[1]
```

```
2
```

```
t[[1,2]]
```

```
b    2
c    1
dtype: int64
```

```
t["c"]
```

```
1
```

```
t[["a","d"]]
```

```
a    4
d    3
dtype: int64
```

```
t[t == 3]
```

```
d    3
dtype: int64
```

```
t[t % 2 == 0]
```

```
a    4
b    2
dtype: int64
```

```
t["d"] = 6
t
```

```
a    4
b    2
c    1
d    6
dtype: int64
```

Index alignment

When performing (arithmetic) operations on series, they will attempt to align the operation by the index values,

```
m = pd.Series([1,2,3,4], index = ["a","b","c","d"])
n = pd.Series([4,3,2,1], index = ["d","c","b","a"])
o = pd.Series([1,1,1,1,1], index = ["b","d","a","c","e"])
```

```
m + n
```

```
a    2
b    4
c    6
d    8
dtype: int64
```

```
n + m
```

```
a    2
b    4
c    6
d    8
dtype: int64
```

```
n + o
```

```
a    2.0
b    3.0
c    4.0
d    5.0
e    NaN
dtype: float64
```

Series and dicts

Series can also be constructed from dicts, in which case the keys are used to create the index,

```
d = {"anna": "A+", "bob": "B-", "carol": "C", "dave": "D+"}
pd.Series(d)
```

```
anna     A+
bob      B-
carol     C
dave     D+
dtype: object
```

Index order will follow key order, unless overriden by `index`,

```
pd.Series(d, index = ["dave","carol","bob","anna"])
```

```
dave     D+
carol     C
bob      B-
anna     A+
dtype: object
```

Missing values

Pandas encodes missing values using NaN (mostly),

```
s = pd.Series(
  {"anna": "A+", "bob": "B-",
   "carol": "C", "dave": "D+"},
  index = ["erin","dave","carol","bob","anna"]
)
```

```
s
```

```
erin     NaN
dave      D+
carol      C
bob       B-
anna      A+
dtype: object
```

```
pd.isna(s)
```

```
erin      True
dave     False
carol    False
bob      False
anna     False
dtype: bool
```

```
s = pd.Series(
  {"anna": 97, "bob": 82,
   "carol": 75, "dave": 68},
  index = ["erin","dave","carol","bob","anna"],
  dtype = 'int64'
)
```

```
s
```

```
erin      NaN
dave     68.0
carol    75.0
bob      82.0
anna     97.0
dtype: float64
```

```
pd.isna(s)
```

```
erin      True
dave     False
carol    False
bob      False
anna     False
dtype: bool
```

Aside - why `np.isna()`?

```python
s = pd.Series([1,2,3,None])
s
```

```
0    1.0
1    2.0
2    3.0
3    NaN
dtype: float64
```

```python
pd.isna(s)
```

```
0    False
1    False
2    False
3     True
dtype: bool
```

```python
s == np.nan
```

```
0    False
1    False
2    False
3    False
dtype: bool
```

```python
np.nan == np.nan
```

```
False
```

```python
np.nan != np.nan
```

```
True
```

```python
np.isnan(np.nan)
```

```
True
```

```python
np.isnan(0)
```

```
False
```

Native NAs

Recent versions of pandas have attempted to adopt a more native missing value, particularly for integer and boolean types,

```
pd.Series([1,2,3,None])
```

```
0     1.0
1     2.0
2     3.0
3     NaN
dtype: float64
```

```
pd.isna( pd.Series([1,2,3,None]) )
```

```
0    False
1    False
2    False
3     True
dtype: bool
```

```
pd.Series([True,False,None])
```

```
0     True
1    False
2     None
dtype: object
```

```
pd.isna( pd.Series([True,False,None
```

```
0    False
1    False
2     True
dtype: bool
```

Setting dtype

We can force things by setting the Series dtype,

```
pd.Series(
  [1,2,3,None],
  dtype = pd.Int64Dtype()
)
```

```
0       1
1       2
2       3
3    <NA>
dtype: Int64
```

```
pd.Series(
  [True, False,None],
  dtype = pd.BooleanDtype()
)
```

```
0     True
1    False
2     <NA>
dtype: boolean
```

String series

Series containing strings can be accessed via the `str` attribute,

```python
s = pd.Series(["the quick", "brown fox", "jumps over", "a lazy dog"])
```

```python
s
```

```
0     the quick
1     brown fox
2    jumps over
3    a lazy dog
dtype: object
```

```python
s.str.upper()
```

```
0     THE QUICK
1     BROWN FOX
2    JUMPS OVER
3    A LAZY DOG
dtype: object
```

```python
s.str.split(" ")
```

```
0      [the, quick]
1      [brown, fox]
2     [jumps, over]
3    [a, lazy, dog]
dtype: object
```

```python
s.str.split(" ").str[1]
```

```
0    quick
1      fox
2     over
3     lazy
dtype: object
```

```python
pd.Series([1,2,3]).str
```

```
Error: AttributeError: Can only use .str accessor
with string values!
```

Categorical Series

```python
pd.Series(
  ["Mon", "Tue", "Wed", "Thur", "Fri"]
)
```

```
0     Mon
1     Tue
2     Wed
3    Thur
4     Fri
dtype: object
```

```python
pd.Series(
  ["Mon", "Tue", "Wed", "Thur", "Fri"],
  dtype="category"
)
```

```
0     Mon
1     Tue
2     Wed
3    Thur
4     Fri
dtype: category
Categories (5, object): ['Fri', 'Mon', 'Thur',
'Tue', 'Wed']
```

```python
pd.Series(
  ["Mon", "Tue", "Wed", "Thur", "Fri"],
  dtype=pd.CategoricalDtype(ordered=True)
)
```

```
0     Mon
1     Tue
2     Wed
3    Thur
4     Fri
dtype: category
Categories (5, object): ['Fri' < 'Mon' < 'Thur' < 'Tue' < 'Wed']
```

Category orders

```python
pd.Series(
  ["Tue", "Thur", "Mon", "Sat"],
  dtype=pd.CategoricalDtype(categories=["Mon", "Tue", "Wed", "Thur", "Fri"],
)
```

```
0     Tue
1    Thur
2     Mon
3     NaN
dtype: category
Categories (5, object): ['Mon' < 'Tue' < 'Wed' < 'Thur' < 'Fri']
```

Constructing DataFrames

Earlier we saw reading a DataFrame in via `read_csv()`, but data frames can also be constructed via `DataFrame()`, in general this is done using a dictionary of columns:

```
n = 5
d = {
  "id":     np.random.randint(100, 999, n),
  "weight": np.random.normal(70, 20, n),
  "height": np.random.normal(170, 15, n),
  "date":   pd.date_range(start='2/1/2022',
                          periods=n, freq='D')
}
d
```

```
{'id': array([623, 142, 336, 846, 570]), 'weight':
array([54.73107509, 98.3655868 , 60.01839835,
91.87785014, 33.72013207]), 'height':
array([176.97949842, 175.06520576, 184.06318659,
171.27255432, 168.20730921]), 'date':
DatetimeIndex(['2022-02-01', '2022-02-02', '2022-
02-03', '2022-02-04',
               '2022-02-05'],
              dtype='datetime64[ns]', freq='D')}
```

```
df = pd.DataFrame(d)
df
```

```
    id     weight      height       date
0  623  54.731075  176.979498 2022-02-01
1  142  98.365587  175.065206 2022-02-02
2  336  60.018398  184.063187 2022-02-03
3  846  91.877850  171.272554 2022-02-04
4  570  33.720132  168.207309 2022-02-05
```

See more IO functions here

DataFrame from nparray

For 2d ndarrays it is also possible to construct a DataFrame - generally it is a good idea to provide column names and row names (indexes)

```
pd.DataFrame(
  np.diag([1,2,3]),
  columns = ["x","y","z"]
)
```

```
   x  y  z
0  1  0  0
1  0  2  0
2  0  0  3
```

```
pd.DataFrame(
  np.diag([1,2,3]),
  columns = ["x","y","z"]
)
```

```
   x  y  z
0  1  0  0
1  0  2  0
2  0  0  3
```

```
pd.DataFrame(
  np.tri(5,3,-1),
  columns = ["x","y","z"],
  index = ["a","b","c","d","e"]
)
```

```
     x    y    z
a  0.0  0.0  0.0
b  1.0  0.0  0.0
c  1.0  1.0  0.0
d  1.0  1.0  1.0
e  1.0  1.0  1.0
```

DataFrame indexing

```
df
```

```
    id     weight      height       date
0  623  54.731075  176.979498 2022-02-01
1  142  98.365587  175.065206 2022-02-02
2  336  60.018398  184.063187 2022-02-03
3  846  91.877850  171.272554 2022-02-04
4  570  33.720132  168.207309 2022-02-05
```

Selecting a column,

```
df[0]
```

```
Error: KeyError: 0
```

```
df["id"]
```

```
0    623
1    142
2    336
3    846
4    570
Name: id, dtype: int64
```

```
df.id
```

```
0    623
1    142
2    336
3    846
4    570
Name: id, dtype: int64
```

Selecting rows (a single slice is assumed to refer to the rows)

```
df[1:3]
```

```
    id     weight      height       date
1  142  98.365587  175.065206 2022-02-02
2  336  60.018398  184.063187 2022-02-03
```

```
df[0::2]
```

```
    id     weight      height       date
0  623  54.731075  176.979498 2022-02-01
2  336  60.018398  184.063187 2022-02-03
4  570  33.720132  168.207309 2022-02-05
```

Index by position

```python
df
```

```
    id     weight      height       date
0  623  54.731075  176.979498 2022-02-01
1  142  98.365587  175.065206 2022-02-02
2  336  60.018398  184.063187 2022-02-03
3  846  91.877850  171.272554 2022-02-04
4  570  33.720132  168.207309 2022-02-05
```

```python
df.iloc[1]
```

```
id                        142
weight              98.365587
height             175.065206
date      2022-02-02 00:00:00
Name: 1, dtype: object
```

```python
df.iloc[[1]]
```

```
    id     weight      height       date
1  142  98.365587  175.065206 2022-02-02
```

```python
df.iloc[0:2]
```

```
    id     weight      height       date
0  623  54.731075  176.979498 2022-02-01
1  142  98.365587  175.065206 2022-02-02
```

```python
df.iloc[lambda x: x.index % 2 != 0]
```

```
    id     weight      height       date
1  142  98.365587  175.065206 2022-02-02
3  846  91.877850  171.272554 2022-02-04
```

```python
df.iloc[1:3,1:3]
```

```
      weight      height
1  98.365587  175.065206
2  60.018398  184.063187
```

```python
df.iloc[0:3, [0,3]]
```

```
    id       date
0  623 2022-02-01
1  142 2022-02-02
2  336 2022-02-03
```

```python
df.iloc[0:3, [True, True, False, False]]
```

```
    id     weight
0  623  54.731075
1  142  98.365587
2  336  60.018398
```

Index by name

```
df.index = (["anna","bob","carol", "dave", "erin"])
df
```

```
        id     weight      height       date
anna   623  54.731075  176.979498 2022-02-01
bob    142  98.365587  175.065206 2022-02-02
carol  336  60.018398  184.063187 2022-02-03
dave   846  91.877850  171.272554 2022-02-04
erin   570  33.720132  168.207309 2022-02-05
```

```
df.loc["anna"]
```

```
id                        623
weight              54.731075
height             176.979498
date      2022-02-01 00:00:00
Name: anna, dtype: object
```

```
df.loc[["anna"]]
```

```
       id     weight      height       date
anna  623  54.731075  176.979498 2022-02-01
```

```
df.loc["bob":"dave"]
```

```
        id     weight      height       date
bob    142  98.365587  175.065206 2022-02-02
carol  336  60.018398  184.063187 2022-02-03
dave   846  91.877850  171.272554 2022-02-04
```

```
df.loc[df.id < 300]
```

```
      id     weight      height       date
bob  142  98.365587  175.065206 2022-02-02
```

```
df.loc[:, "date"]
```

```
anna    2022-02-01
bob     2022-02-02
carol   2022-02-03
dave    2022-02-04
erin    2022-02-05
Name: date, dtype: datetime64[ns]
```

```
df.loc[["bob","erin"], "weight":"height"]
```

```
         weight      height
bob   98.365587  175.065206
erin  33.720132  168.207309
```

```
df.loc[0:2, "weight":"height"]
```

```
Error: TypeError: cannot do slice indexing on Index with these indexers
[0] of type int
```

Views vs. Copies

In general most pandas operations will generate a new object but some will return views, mostly the later occurs with subsetting.

```python
d = pd.DataFrame(np.arange(6).reshape(3,2), colu
d
```

```
   x  y
0  0  1
1  2  3
2  4  5
```

```python
v = d.iloc[0:2,0:2]
v
```

```
   x  y
0  0  1
1  2  3
```

```python
d.iloc[0,1] = -1
v
```

```
   x  y
0  0 -1
1  2  3
```

```python
v.iloc[0,0] = np.pi
v
```

```
          x  y
0  3.141593 -1
1  2.000000  3
```

```python
d
```

```
   x  y
0  0 -1
1  2  3
2  4  5
```

See the documetation here for more details

Filtering rows

The `query()` method can be used for filtering rows, it evaluates a string expression in the context of the data frame.

```
df
```

```
        id     weight      height        date
anna   623  54.731075  176.979498 2022-02-01
bob    142  98.365587  175.065206 2022-02-02
carol  336  60.018398  184.063187 2022-02-03
dave   846  91.877850  171.272554 2022-02-04
erin   570  33.720132  168.207309 2022-02-05
```

```
df.query('date == "2022-02-01"')
```

```
       id     weight      height        date
anna  623  54.731075  176.979498 2022-02-01
```

```
df.query('weight > 50')
```

```
        id     weight      height        date
anna   623  54.731075  176.979498 2022-02-01
bob    142  98.365587  175.065206 2022-02-02
carol  336  60.018398  184.063187 2022-02-03
dave   846  91.877850  171.272554 2022-02-04
```

```
df.query('weight > 50 & height < 165')
```

```
Empty DataFrame
Columns: [id, weight, height, date]
Index: []
```

```
qid = 414
df.query('id == @qid')
```

```
Empty DataFrame
Columns: [id, weight, height, date]
Index: []
```

For more details on query syntax see here

Element access

```python
df
```

```
          id     weight      height       date
anna     623  54.731075  176.979498 2022-02-01
bob      142  98.365587  175.065206 2022-02-02
carol    336  60.018398  184.063187 2022-02-03
dave     846  91.877850  171.272554 2022-02-04
erin     570  33.720132  168.207309 2022-02-05
```

```python
df[0,0]
```

```
Error: KeyError: (0, 0)
```

```python
df.iat[0,0]
```

```
623
```

```python
df.id[0]
```

```
623
```

```python
df[0:1].id[0]
```

```
623
```

```python
df["anna", "id"]
```

```
Error: KeyError: ('anna', 'id')
```

```python
df.at["anna", "id"]
```

```
623
```

```python
df["id"]["anna"]
```

```
623
```

```python
df["id"][0]
```

```
623
```

DataFrame properties

```
df.size
```

```
20
```

```
df.shape
```

```
(5, 4)
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 5 entries, anna to erin
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   id      5 non-null      int64
 1   weight  5 non-null      float64
 2   height  5 non-null      float64
 3   date    5 non-null      datetime64[ns]
dtypes: datetime64[ns](1), float64(2), int64(1)
memory usage: 372.0+ bytes
```

```
df.dtypes
```

```
id                  int64
weight            float64
height            float64
date       datetime64[ns]
dtype: object
```

```
df.describe()
```

```
               id     weight      height
count    5.000000   5.000000    5.000000
mean   503.400000  67.742608  175.117551
std    271.453127  26.957230    6.042132
min    142.000000  33.720132  168.207309
25%    336.000000  54.731075  171.272554
50%    570.000000  60.018398  175.065206
75%    623.000000  91.877850  176.979498
max    846.000000  98.365587  184.063187
```

Selecting Columns

Beyond the use of `loc()` and `iloc()` there is also the `filter()` method which can be used to select columns (or indices) by name with pattern matching

```
df.filter(items=["id","weight"])
```

```
        id     weight
anna   623  54.731075
bob    142  98.365587
carol  336  60.018398
dave   846  91.877850
erin   570  33.720132
```

```
df.filter(regex="ght$")
```

```
          weight      height
anna   54.731075  176.979498
bob    98.365587  175.065206
carol  60.018398  184.063187
dave   91.877850  171.272554
erin   33.720132  168.207309
```

```
df.filter(like = "i")
```

```
        id     weight      height
anna   623  54.731075  176.979498
bob    142  98.365587  175.065206
carol  336  60.018398  184.063187
dave   846  91.877850  171.272554
erin   570  33.720132  168.207309
```

```
df.filter(like="o", axis=0)
```

```
        id     weight      height        date
bob    142  98.365587  175.065206  2022-02-02
carol  336  60.018398  184.063187  2022-02-03
```

Adding columns

Indexing with assignment allows for inplace modification of a DataFrame, while `assign()` creates a new object (but is chainable)

```
df['student'] = [True, True, True, False, None]
df['age'] = [19, 22, 25, None, None]
df
```

```
        id     weight      height       date student   age
anna   623  54.731075  176.979498 2022-02-01    True  19.0
bob    142  98.365587  175.065206 2022-02-02    True  22.0
carol  336  60.018398  184.063187 2022-02-03    True  25.0
dave   846  91.877850  171.272554 2022-02-04   False   NaN
erin   570  33.720132  168.207309 2022-02-05    None   NaN
```

```
df.assign(
  student = lambda x: np.where(x.student, "yes", "no"),
  rand = np.random.rand(5)
)
```

```
        id     weight      height       date student   age      rand
anna   623  54.731075  176.979498 2022-02-01     yes  19.0  0.752240
bob    142  98.365587  175.065206 2022-02-02     yes  22.0  0.876159
carol  336  60.018398  184.063187 2022-02-03     yes  25.0  0.797148
dave   846  91.877850  171.272554 2022-02-04      no   NaN  0.863391
erin   570  33.720132  168.207309 2022-02-05      no   NaN  0.527163
```

```
df
```

```
        id     weight      height       date student   age
anna   623  54.731075  176.979498 2022-02-01    True  19.0
bob    142  98.365587  175.065206 2022-02-02    True  22.0
carol  336  60.018398  184.063187 2022-02-03    True  25.0
dave   846  91.877850  171.272554 2022-02-04   False   NaN
erin   570  33.720132  168.207309 2022-02-05    None   NaN
```

Removing columns (and rows)

Columns can be dropped via the `drop()` method,

```python
df.drop(['student'])
```

```
Error: KeyError: "['student'] not found in axis"
```

```python
df.drop(['student'], axis=1)
```

```
        id     weight      height       date   age
anna   623  54.731075  176.979498 2022-02-01  19.0
bob    142  98.365587  175.065206 2022-02-02  22.0
carol  336  60.018398  184.063187 2022-02-03  25.0
dave   846  91.877850  171.272554 2022-02-04   NaN
erin   570  33.720132  168.207309 2022-02-05   NaN
```

```python
df.drop(['anna','dave'])
```

```
        id     weight      height       date student   age
bob    142  98.365587  175.065206 2022-02-02    True  22.0
carol  336  60.018398  184.063187 2022-02-03    True  25.0
erin   570  33.720132  168.207309 2022-02-05    None   NaN
```

```python
df.drop(columns = df.columns == "age")
```

```
Error: KeyError: '[False, False, False, False, False, True] not found in
axis'
```

```python
df.drop(columns = df.columns[df.columns == "age"])
```

```
        id     weight      height       date student
anna   623  54.731075  176.979498 2022-02-01    True
bob    142  98.365587  175.065206 2022-02-02    True
carol  336  60.018398  184.063187 2022-02-03    True
dave   846  91.877850  171.272554 2022-02-04   False
erin   570  33.720132  168.207309 2022-02-05    None
```

```python
df.drop(columns = df.columns[df.columns.str.contains("ght")])
```

```
        id       date student   age
anna   623 2022-02-01    True  19.0
bob    142 2022-02-02    True  22.0
carol  336 2022-02-03    True  25.0
dave   846 2022-02-04   False   NaN
erin   570 2022-02-05    None   NaN
```

Dropping missing values

Columns can be dropped via the `drop()` method,

```python
df
```

```
        id     weight      height       date student   age
anna   623  54.731075  176.979498 2022-02-01    True  19.0
bob    142  98.365587  175.065206 2022-02-02    True  22.0
carol  336  60.018398  184.063187 2022-02-03    True  25.0
dave   846  91.877850  171.272554 2022-02-04   False   NaN
erin   570  33.720132  168.207309 2022-02-05    None   NaN
```

```python
df.dropna()
```

```
        id     weight      height       date student   age
anna   623  54.731075  176.979498 2022-02-01    True  19.0
bob    142  98.365587  175.065206 2022-02-02    True  22.0
carol  336  60.018398  184.063187 2022-02-03    True  25.0
```

```python
df.dropna(how="all")
```

```
        id     weight      height       date student   age
anna   623  54.731075  176.979498 2022-02-01    True  19.0
bob    142  98.365587  175.065206 2022-02-02    True  22.0
carol  336  60.018398  184.063187 2022-02-03    True  25.0
dave   846  91.877850  171.272554 2022-02-04   False   NaN
erin   570  33.720132  168.207309 2022-02-05    None   NaN
```

```python
df.dropna(axis=1)
```

```
        id     weight      height       date
anna   623  54.731075  176.979498 2022-02-01
bob    142  98.365587  175.065206 2022-02-02
carol  336  60.018398  184.063187 2022-02-03
dave   846  91.877850  171.272554 2022-02-04
erin   570  33.720132  168.207309 2022-02-05
```

```python
df.dropna(axis=1, thresh=4)
```

```
        id     weight      height       date student
anna   623  54.731075  176.979498 2022-02-01    True
bob    142  98.365587  175.065206 2022-02-02    True
carol  336  60.018398  184.063187 2022-02-03    True
dave   846  91.877850  171.272554 2022-02-04   False
erin   570  33.720132  168.207309 2022-02-05    None
```

Sorting

DataFrames can be sorted on one or more columns via `sort_values()`,

```
df
```

```
        id     weight      height       date student   age
anna   623  54.731075  176.979498 2022-02-01    True  19.0
bob    142  98.365587  175.065206 2022-02-02    True  22.0
carol  336  60.018398  184.063187 2022-02-03    True  25.0
dave   846  91.877850  171.272554 2022-02-04   False   NaN
erin   570  33.720132  168.207309 2022-02-05    None   NaN
```

```
df.sort_values(by=["student","id"], ascending=[True,False])
```

```
        id     weight      height       date student   age
dave   846  91.877850  171.272554 2022-02-04   False   NaN
anna   623  54.731075  176.979498 2022-02-01    True  19.0
carol  336  60.018398  184.063187 2022-02-03    True  25.0
bob    142  98.365587  175.065206 2022-02-02    True  22.0
erin   570  33.720132  168.207309 2022-02-05    None   NaN
```

Row binds

DataFrames can have their rows joined via the the `concat()` function (`append()` is also available but deprecated),

```python
df1 = pd.DataFrame(
  np.arange(6).reshape(3,2),
  columns=list("xy")
)
df1
```

```
   x  y
0  0  1
1  2  3
2  4  5
```

```python
df2 = pd.DataFrame(
  np.arange(12,6,-1).reshape(3,2),
  columns=list("xy")
)
df2
```

```
    x   y
0  12  11
1  10   9
2   8   7
```

```python
pd.concat([df1,df2])
```

```
    x   y
0   0   1
1   2   3
2   4   5
0  12  11
1  10   9
2   8   7
```

```python
pd.concat([df1.loc[:,["y","x"]],df2])
```

```
    y   x
0   1   0
1   3   2
2   5   4
0  11  12
1   9  10
2   7   8
```

Imputing columns

When binding rows missing columns will be added with NaN or <NA> entries.

```
df3 = pd.DataFrame(np.ones((3,3)), columns=list("xbz"))
df3
```

```
     x    b    z
0  1.0  1.0  1.0
1  1.0  1.0  1.0
2  1.0  1.0  1.0
```

```
pd.concat([df1,df3,df2])
```

```
      x     y    b    z
0   0.0   1.0  NaN  NaN
1   2.0   3.0  NaN  NaN
2   4.0   5.0  NaN  NaN
0   1.0   NaN  1.0  1.0
1   1.0   NaN  1.0  1.0
2   1.0   NaN  1.0  1.0
0  12.0  11.0  NaN  NaN
1  10.0   9.0  NaN  NaN
2   8.0   7.0  NaN  NaN
```

Column binds

Similarly, columns can be joined with `concat()` where `axis=1`,

```
df1 = pd.DataFrame(
  np.arange(6).reshape(3,2),
  columns=list("xy"),
  index=list("abc")
)
df1
```

```
   x  y
a  0  1
b  2  3
c  4  5
```

```
df2 = pd.DataFrame(
  np.arange(10,6,-1).reshape(2,2),
  columns=list("mn"),
  index=list("ac")
)
df2
```

```
    m  n
a  10  9
c   8  7
```

```
pd.concat([df1,df2], axis=1)
```

```
   x  y     m    n
a  0  1  10.0  9.0
b  2  3   NaN  NaN
c  4  5   8.0  7.0
```

```
pd.concat([df1,df2], axis=1, join="inner")
```

```
   x  y   m  n
a  0  1  10  9
c  4  5   8  7
```

Joining DataFrames

Table joins are implemented via the `merge()` function or method,

```
df1 = pd.DataFrame(
  {'a': ['foo', 'bar'], 'b': [1, 2]}
)
df1
```

```
     a  b
0  foo  1
1  bar  2
```

```
pd.merge(df1,df2, how="inner")
```

```
     a  b  c
0  foo  1  3
```

```
pd.merge(df1,df2, how="outer", on="a")
```

```
     a    b    c
0  foo  1.0  3.0
1  bar  2.0  NaN
2  baz  NaN  4.0
```

```
df2 = pd.DataFrame(
  {'a': ['foo', 'baz'], 'c': [3, 4]}
)
df2
```

```
     a  c
0  foo  3
1  baz  4
```

```
df1.merge(df2, how="left")
```

```
     a  b    c
0  foo  1  3.0
1  bar  2  NaN
```

```
df1.merge(df2, how="right")
```

```
     a    b  c
0  foo  1.0  3
1  baz  NaN  4
```

join vs merge vs concat

All three can be used to accomplish the same thing, in terms of “column bind” type operations.

- `concat()` stacks DataFrames on either axis, with basic alignment based on (row) indexes. `join` argument only supports “inner” and “outer”.
- `merge()` aligns based on one or more shared columns. `how` supports “inner”, “outer”, “left”, “right”, and “cross”.
- `join()` uses `merge()` behind the scenes, but prefers to join based on (row) indexes. Also has different default `how` compared to `merge()`, “left” vs “inner”.

groupby and agg

Groups can be created within a DataFrame via `groupby()` - these groups are then used by the standard summary methods (e.g. `sum()`, `mean()`, `std()`, etc.).

```
df.groupby("student")
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x2921ee500>
```

```
df.groupby("student").groups
```

```
{False: ['dave'], True: ['anna', 'bob', 'carol']}
```

```
df.groupby("student").mean(numeric_only=True)
```

```
            id     weight      height   age
student
False    846.0  91.877850  171.272554   NaN
True     367.0  71.038353  178.702630  22.0
```

```
df.groupby("student", dropna=False).groups
```

```
Error: ValueError: Categorical categories cannot be
null
```

```
df.groupby("student", dropna=False).mean(numeric_onl
```

```
            id     weight      height   age
student
False    846.0  91.877850  171.272554   NaN
True     367.0  71.038353  178.702630  22.0
NaN      570.0  33.720132  168.207309   NaN
```

Selecting groups

```python
df
```

```
        id     weight      height        date student   age
anna   623  54.731075  176.979498 2022-02-01    True  19.0
bob    142  98.365587  175.065206 2022-02-02    True  22.0
carol  336  60.018398  184.063187 2022-02-03    True  25.0
dave   846  91.877850  171.272554 2022-02-04   False   NaN
erin   570  33.720132  168.207309 2022-02-05    None   NaN
```

```python
df.groupby("student").get_group(True)
```

```
        id     weight      height        date student   age
anna   623  54.731075  176.979498 2022-02-01    True  19.0
bob    142  98.365587  175.065206 2022-02-02    True  22.0
carol  336  60.018398  184.063187 2022-02-03    True  25.0
```

```python
df.groupby("student").get_group(False)
```

```
       id    weight      height        date student  age
dave  846  91.87785  171.272554 2022-02-04   False  NaN
```

```python
df.groupby("student", dropna=False).get_group(np.nan)
```

```
Error: KeyError: nan
```

Aggregation

```python
df = df.drop("date", axis=1)
```

```python
df.groupby("student").agg("mean")
```

```
            id     weight      height   age
student                                    
False    846.0  91.877850  171.272554   NaN
True     367.0  71.038353  178.702630  22.0
```

```python
df.groupby("student").agg([np.mean, np.std])
```

```
            id                weight  ...    height   age     
          mean         std      mean  ...       std  mean  std
student                               ...                     
False    846.0         NaN  91.877850  ...       NaN   NaN  NaN
True     367.0  241.993802  71.038353  ...  4.740021  22.0  3.0

[2 rows x 8 columns]
```

More on multindexes and other aggregation/summary methods next time.