

Lec 08 - pandas

Statistical Computing and Computation

Sta 663 | Spring 2022

Dr. Colin Rundel

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 object class
- Reading and writing tabular data
- Data munging (filtering, grouping, summarizing, joining, etc.)
- Data reshaping

DataFrame

```
import pandas as pd
```

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

##      Sepal.Length  Sepal.Width   Petal.Length   Petal.Width
## 0           5.1         3.5          1.4          0.2
## 1           4.9         3.0          1.4          0.2
## 2           4.7         3.2          1.3          0.2
## 3           4.6         3.1          1.5          0.2
## 4           5.0         3.6          1.4          0.2
## ..
## 145          6.7         3.0          5.2          2.3  vi
## 146          6.3         2.5          5.0          1.9  vi
## 147          6.5         3.0          5.2          2.0  vi
## 148          6.2         3.4          5.4          2.3  vi
## 149          5.9         3.0          5.1          1.8  vi
##
## [150 rows x 5 columns]
```

```
print( type(iris) )
## <class 'pandas.core.frame.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

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()` methods.

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

```
t.array
```

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

Series + NumPy

Series objects are compatible with NumPy like functions (vectorized)

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

```
t + 1
```

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

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

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

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

```
## a    0.000335
```

Series indexing

Series can be indexed in the same was as NumPy arrays with the addition of being able to use 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
```

```
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
```

Index alignment

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

```
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 + o

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

n + m

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

Series and dicts

Series can also be constructed from dicts, in which case the keys are used to form 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 overridden 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":  
     index = ["erin", "dave", "carol", "bob", "anna"]  
)
```

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

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

```
## erin      True  
## dave     False  
## carol    False  
## bob      False
```

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

```
## 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
```

Aside - why np.isna()?

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

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

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

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

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

```
## 0    False  
## 1    False  
## 2    False  
## 3    False
```

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

```
## False
```

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

```
## True
```

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

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

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

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

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

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

We can force things by setting the Series dtype,

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

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

String series

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

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

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

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

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

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

```
## 0      [the, quick]
```

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

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

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

```
## AttributeError: Can only use .str accessor with string-like data
```

Categorical Series

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

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

```
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']
```

```
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']
```

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

```
## 0      Tue
## 1      Thur
## 2      Mon
```

Constructing DataFrames

We just saw reading a DataFrame in via `read_csv()`, but data frames can also be constructed via `DataFrame()`, general this is done via dict 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([168, 615, 346, 390, 556]), 'weight': array([102.91553493, 71.7673641 , 76.66605866, 74.735491, 66.776415, 71.7673641 , 155.907801, 2022-02-05]),
##          'height': array([188.677769, 171.386839, 171.386839, 171.386839, 171.386839], dtype='datetime64[ns]', freq='D')}

df = pd.DataFrame(d)
df

##      id    weight    height        date
## 0   168  102.915535  188.677769 2022-02-01
## 1   615   71.767364  171.386839 2022-02-02
## 2   346   76.666059  171.386839 2022-02-03
```

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  168 102.915535 188.677769 2022-02-01
## 1  615   71.767364 155.907801 2022-02-02
## 2  346   76.666059 171.386839 2022-02-03
## 3  390   74.735465 173.151008 2022-02-04
## 4  556   50.538488 183.083407 2022-02-05
```

Selecting a column,

```
df[0]
```

```
## KeyError: 0
```

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

```
## 0    168
## 1    615
## 2    346
## 3    390
## 4    556
## Name: id, dtype: int64
```

```
df.id
```

```
## 0    168
## 1    615
## 2    346
```

Selecting rows, a single slice is assumed to refer to the rows

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

```
##      id    weight    height     date
## 1  615   71.767364 155.907801 2022-02-02
## 2  346   76.666059 171.386839 2022-02-03
```

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

```
##      id    weight    height     date
## 0  168 102.915535 188.677769 2022-02-01
## 2  346   76.666059 171.386839 2022-02-03
## 4  556   50.538488 183.083407 2022-02-05
```

Index by position

```
df
```

```
##      id    weight    height     date
## 0 168 102.915535 188.677769 2022-02-01
## 1 615  71.767364 155.907801 2022-02-02
## 2 346  76.666059 171.386839 2022-02-03
## 3 390  74.735465 173.151008 2022-02-04
## 4 556  50.538488 183.083407 2022-02-05
```

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

```
## id              615
## weight          71.767364
## height          155.907801
## date 2022-02-02 00:00:00
## Name: 1, dtype: object
```

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

```
##      id    weight    height     date
## 1 615  71.767364 155.907801 2022-02-02
```

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

```
##      id    weight    height     date
## 0 168 102.915535 188.677769 2022-02-01
## 1 615  71.767364 155.907801 2022-02-02
```

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

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

```
##      weight    height
## 1 71.767364 155.907801
## 2 76.666059 171.386839
```

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

```
##      id     date
## 0 168 2022-02-01
## 1 615 2022-02-02
## 2 346 2022-02-03
```

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

```
##      id    weight
## 0 168 102.915535
## 1 615  71.767364
## 2 346  76.666059
```

Index by name

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

```
##      id    weight    height      date  
## anna 168 102.915535 188.677769 2022-02-01  
## bob  615  71.767364 155.907801 2022-02-02  
## carol 346  76.666059 171.386839 2022-02-03  
## dave 390  74.735465 173.151008 2022-02-04  
## erin 556  50.538488 183.083407 2022-02-05
```

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

```
## id           168  
## weight      102.915535  
## height     188.677769  
## date 2022-02-01 00:00:00  
## Name: anna, dtype: object
```

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

```
##      id    weight    height      date  
## anna 168 102.915535 188.677769 2022-02-01
```

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

```
##      id    weight    height      date  
## bob  615  71.767364 155.907801 2022-02-02  
## carol 346  76.666059 171.386839 2022-02-03  
## dave 390  74.735465 173.151008 2022-02-04
```

```
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  71.767364 155.907801  
## erin 50.538488 183.083407
```

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

```
## TypeError: cannot do slice indexing on Index with these indexers [0]
```

Views vs. Copies

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

```
d = pd.DataFrame(np.arange(6).reshape(3,2), columns = ["x", "y"])
d
```

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

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

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

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

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

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

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

```
d
```

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

See the documentation 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 168 102.915535 188.677769 2022-02-01
## bob  615  71.767364 155.907801 2022-02-02
## carol 346  76.666059 171.386839 2022-02-03
## dave 390  74.735465 173.151008 2022-02-04
## erin 556  50.538488 183.083407 2022-02-05
```

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

```
##      id    weight    height      date
## anna 168 102.915535 188.677769 2022-02-01
```

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

```
##      id    weight    height      date
## anna 168 102.915535 188.677769 2022-02-01
## bob  615  71.767364 155.907801 2022-02-02
## carol 346  76.666059 171.386839 2022-02-03
## dave 390  74.735465 173.151008 2022-02-04
## erin 556  50.538488 183.083407 2022-02-05
```

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

```
##      id    weight    height      date
## bob  615  71.767364 155.907801 2022-02-02
```

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

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

For more details on query syntax see [here](#)

Element access

```
df
```

```
##      id    weight    height     date
## anna  168 102.915535 188.677769 2022-02-01
## bob   615  71.767364 155.907801 2022-02-02
## carol 346  76.666059 171.386839 2022-02-03
## dave  390  74.735465 173.151008 2022-02-04
## erin  556  50.538488 183.083407 2022-02-05
```

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

```
## KeyError: (0, 0)
```

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

```
## 168
```

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

```
## 168
```

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

```
## 168
```

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

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

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

```
## 168
```

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

```
## 168
```

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

```
## 168
```

DataFrame properties

```
df
```

```
##      id    weight    height     date
## anna 168 102.915535 188.677769 2022-02-01
## bob  615  71.767364 155.907801 2022-02-02
## carol 346  76.666059 171.386839 2022-02-03
## dave 390  74.735465 173.151008 2022-02-04
## erin 556  50.538488 183.083407 2022-02-05
```

```
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
```

```
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    415.000000 75.324582 174.441365
## std     177.676673 18.644111 12.568296
```

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
```

```
##      id    weight    height     date
## anna  168 102.915535 188.677769 2022-02-01
## bob   615  71.767364 155.907801 2022-02-02
## carol 346  76.666059 171.386839 2022-02-03
## dave  390  74.735465 173.151008 2022-02-04
## erin  556  50.538488 183.083407 2022-02-05
```

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

```
##      id    weight
## anna  168 102.915535
## bob   615  71.767364
## carol 346  76.666059
## dave  390  74.735465
## erin  556  50.538488
```

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

```
##      id    weight    height
## anna  168 102.915535 188.677769
## bob   615  71.767364 155.907801
## carol 346  76.666059 171.386839
## dave  390  74.735465 173.151008
```

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

```
##      weight    height
## anna 102.915535 188.677769
## bob  71.767364 155.907801
## carol 76.666059 171.386839
## dave 74.735465 173.151008
## erin 50.538488 183.083407
```

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

```
##      id    weight    height     date
## bob   615  71.767364 155.907801 2022-02-02
## carol 346  76.666059 171.386839 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
```

```
##      id    weight    height      date
## anna 168 102.915535 188.677769 2022-02-01
## bob  615  71.767364 155.907801 2022-02-02
## carol 346  76.666059 171.386839 2022-02-03
## dave 390  74.735465 173.151008 2022-02-04
## erin 556  50.538488 183.083407 2022-02-05
```

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

```
##      id    weight    height      date student   age
## anna 168 102.915535 188.677769 2022-02-01    True 19.0
## bob  615  71.767364 155.907801 2022-02-02    True 22.0
## carol 346  76.666059 171.386839 2022-02-03    True 25.0
## dave 390  74.735465 173.151008 2022-02-04   False NaN
## erin 556  50.538488 183.083407 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 168 102.915535 188.677769 2022-02-01    yes 19.0 0.60772
## bob  615  71.767364 155.907801 2022-02-02    yes 22.0 0.54165
## carol 346  76.666059 171.386839 2022-02-03    yes 25.0 0.56239
## dave 390  74.735465 173.151008 2022-02-04     no NaN 0.84917
## erin 556  50.538488 183.083407 2022-02-05     no NaN 0.44029
```

```
df
```

```
##      id    weight    height      date student   age
## anna 168 102.915535 188.677769 2022-02-01    True 19.0
```

Removing columns (and rows)

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

```
df
```

```
##      id    weight    height      date student   age
## anna  168 102.915535 188.677769 2022-02-01  True  19.0
## bob   615  71.767364 155.907801 2022-02-02  True  22.0
## carol 346  76.666059 171.386839 2022-02-03  True  25.0
## dave  390  74.735465 173.151008 2022-02-04 False   NaN
## erin  556  50.538488 183.083407 2022-02-05  None   NaN
```

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

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

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

```
##      id    weight    height      date   age
## anna  168 102.915535 188.677769 2022-02-01 19.0
## bob   615  71.767364 155.907801 2022-02-02 22.0
## carol 346  76.666059 171.386839 2022-02-03 25.0
## dave  390  74.735465 173.151008 2022-02-04  NaN
## erin  556  50.538488 183.083407 2022-02-05  NaN
```

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

```
##      id    weight    height      date student   age
## bob   615  71.767364 155.907801 2022-02-02  True  22.0
## carol 346  76.666059 171.386839 2022-02-03  True  25.0
```

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

```
## KeyError: '[False, False, False, False, True] not found in ax
```

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

```
##      id    weight    height      date student
## anna  168 102.915535 188.677769 2022-02-01  True
## bob   615  71.767364 155.907801 2022-02-02  True
## carol 346  76.666059 171.386839 2022-02-03  True
## dave  390  74.735465 173.151008 2022-02-04 False
## erin  556  50.538488 183.083407 2022-02-05  None
```

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

```
##      id      date student   age
## anna  168 2022-02-01  True  19.0
## bob   615 2022-02-02  True  22.0
```

Dropping missing values

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

```
df
```

```
##      id    weight    height      date student   age
## anna 168 102.915535 188.677769 2022-02-01  True 19.0
## bob  615  71.767364 155.907801 2022-02-02  True 22.0
## carol 346  76.666059 171.386839 2022-02-03  True 25.0
## dave 390  74.735465 173.151008 2022-02-04 False NaN
## erin 556  50.538488 183.083407 2022-02-05 None NaN
```

```
df.dropna()
```

```
##      id    weight    height      date student   age
## anna 168 102.915535 188.677769 2022-02-01  True 19.0
## bob  615  71.767364 155.907801 2022-02-02  True 22.0
## carol 346  76.666059 171.386839 2022-02-03  True 25.0
```

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

```
##      id    weight    height      date student   age
## anna 168 102.915535 188.677769 2022-02-01  True 19.0
## bob  615  71.767364 155.907801 2022-02-02  True 22.0
## carol 346  76.666059 171.386839 2022-02-03  True 25.0
## dave 390  74.735465 173.151008 2022-02-04 False NaN
## erin 556  50.538488 183.083407 2022-02-05 None NaN
```

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

```
##      id    weight    height      date
## anna 168 102.915535 188.677769 2022-02-01
## bob  615  71.767364 155.907801 2022-02-02
## carol 346  76.666059 171.386839 2022-02-03
## dave 390  74.735465 173.151008 2022-02-04
## erin 556  50.538488 183.083407 2022-02-05
```

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

```
##      id    weight    height      date student
## anna 168 102.915535 188.677769 2022-02-01  True
## bob  615  71.767364 155.907801 2022-02-02  True
## carol 346  76.666059 171.386839 2022-02-03  True
## dave 390  74.735465 173.151008 2022-02-04 False
## erin 556  50.538488 183.083407 2022-02-05 None
```

Sorting

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

```
df
```

```
##      id    weight     height       date student   age
## anna  168 102.915535 188.677769 2022-02-01   True 19.0
## bob   615  71.767364 155.907801 2022-02-02   True 22.0
## carol 346  76.666059 171.386839 2022-02-03   True 25.0
## dave  390  74.735465 173.151008 2022-02-04  False  NaN
## erin  556  50.538488 183.083407 2022-02-05    None  NaN
```

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

```
##      id    weight     height       date student   age
## dave  390  74.735465 173.151008 2022-02-04  False  NaN
## bob   615  71.767364 155.907801 2022-02-02   True 22.0
## carol 346  76.666059 171.386839 2022-02-03   True 25.0
## anna  168 102.915535 188.677769 2022-02-01   True 19.0
## erin  556  50.538488 183.083407 2022-02-05    None  NaN
```

Row binds

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

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

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

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

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

```
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
```

```
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

```
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")
df1
```

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

```
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
```

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

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

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

```
##      x  y      m      n
## a    0  1  10.0   9.0
## c    4  5   8.0   7.0
```

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
```

```
##      id    weight     height       date student   age
## anna  168 102.915535 188.677769 2022-02-01  True 19.0
## bob   615  71.767364 155.907801 2022-02-02  True 22.0
## carol 346  76.666059 171.386839 2022-02-03  True 25.0
## dave  390  74.735465 173.151008 2022-02-04 False NaN
## erin  556  50.538488 183.083407 2022-02-05  None NaN
```

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

```
## <pandas.core.groupby.generic.DataFrameGroupBy object at 0x14c653e20>
```

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

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

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

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

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

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

Selecting groups

```
df
```

```
##      id    weight    height      date student   age
## anna  168 102.915535 188.677769 2022-02-01   True 19.0
## bob   615  71.767364 155.907801 2022-02-02   True 22.0
## carol 346  76.666059 171.386839 2022-02-03   True 25.0
## dave  390  74.735465 173.151008 2022-02-04  False  NaN
## erin  556  50.538488 183.083407 2022-02-05   None  NaN
```

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

```
##      id    weight    height      date student   age
## anna  168 102.915535 188.677769 2022-02-01   True 19.0
## bob   615  71.767364 155.907801 2022-02-02   True 22.0
## carol 346  76.666059 171.386839 2022-02-03   True 25.0
```

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

```
##      id    weight    height      date student   age
## dave  390  74.735465 173.151008 2022-02-04  False  NaN
```

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

Aggregation

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

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

```
##           id      weight     height    age
## student
## False    390.000000  74.735465 173.151008   NaN
## True     376.333333  83.782986 171.990803 22.0
```

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

```
##           id      weight     ...     height    age
##          mean       std     mean     ...       std  mean   std
## student
## ... 
## False    390.000000      NaN  74.735465 ...      NaN  NaN  NaN
## True     376.333333  225.038515  83.782986 ...  16.39333 22.0  3.0
##
## [2 rows x 8 columns]
```

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

```
## MultiIndex([(  'id', 'mean'),
##              (  'id', 'std'),
##              ('weight', 'mean'),
##              ('weight', 'std'),
##              ('height', 'mean'),
##              ('height', 'std'),
##              ('age', 'mean'),
##              ('age', 'std')])
```

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