

# pandas

## Lecture 08

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 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 dtypes are be heterogeneous
- Both columns and rows can have names

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

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

```
1 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).

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

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

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

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

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

```
0    True
dtype: bool
```

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

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

```
1 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.

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

```
1 s
```

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

```
1 s.array
```

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

```
1 s.index
```

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

An index (row names) can also be explicitly provided when constructing a Series,

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

```
1 t
```

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

```
1 t.array
```

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

```
1 t.index
```

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

# Series + NumPy

Series objects are compatible with NumPy like functions (i.e. vectorized)

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

```
1 t + 1
```

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

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

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

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

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

```
1 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 way as NumPy arrays with the addition of being able to use index label(s) when selecting elements.

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

```
1 t[1]
```

```
2
```

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

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

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

```
1
```

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

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

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

```
d    3
dtype: int64
```

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

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

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

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

# Index alignment

When performing operations with multiple series, generally pandas will attempt to align the operation by the index values,

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

```
1 m + n
```

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

```
1 n + m
```

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

```
1 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 dictionaries, in which case the keys are used as the index,

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

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

Index order will follow key order, unless overridden by [index](#),

```
1 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),

```
1 s = pd.Series(  
2     {"anna": "A+", "bob": "B-",  
3      "carol": "C", "dave": "D+"},  
4     index = ["erin", "dave", "carol", "bob", "anna"]  
5 )
```

```
1 s
```

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

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

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

```
1 s = pd.Series(  
2     {"anna": 97, "bob": 82,  
3      "carol": 75, "dave": 68},  
4     index = ["erin", "dave", "carol", "bob", "anna"],  
5     dtype = 'int64'  
6 )
```

```
1 s
```

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

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

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

# Aside - why `np.isnan()`?

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

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

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

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

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

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

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

False

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

True

```
1 np.isnan(np.nan)
```

True

```
1 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,

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

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

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

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

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

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

```
1 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,

```
1 pd.Series(  
2     [1,2,3,None],  
3     dtype = pd.Int64Dtype()  
4 )
```

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

```
1 pd.Series(  
2     [True, False,None],  
3     dtype = pd.BooleanDtype()  
4 )
```

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

# String series

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

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

```
1 s
```

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

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

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

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

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

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

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

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

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

# Categorical Series

```
1 pd.Series(  
2     ["Mon", "Tue", "Wed", "Thur", "Fri"]  
3 )
```

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

```
1 pd.Series(  
2     ["Mon", "Tue", "Wed", "Thur", "Fri"],  
3     dtype="category"  
4 )
```

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

```
1 pd.Series(  
2     ["Mon", "Tue", "Wed", "Thur", "Fri"],  
3     dtype=pd.CategoricalDtype(ordered=True)  
4 )
```

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

# Category orders

```
1 pd.Series(  
2     ["Tue", "Thur", "Mon", "Sat"],  
3     dtype=pd.CategoricalDtype(  
4         categories=["Mon", "Tue", "Wed", "Thur", "Fri"],  
5         ordered=True  
6     )  
7 )
```

```
0     Tue  
1     Thur  
2     Mon  
3     NaN
```

```
dtype: category
```

```
Categories (5, object): ['Mon' < 'Tue' < 'Wed' < 'Thur' < 'Fri']
```



# Constructing DataFrames

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

```
1 n = 5
2 d = {
3     "id":      np.random.randint(100, 999, n),
4     "weight": np.random.normal(70, 20, n),
5     "height": np.random.normal(170, 15, n),
6     "date":   pd.date_range(start='2/1/2022',
7                             periods=n, freq='D')
8 }
```

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

	id	weight	height	date
0	780	58.244369	180.317131	2022-02-01
1	675	63.680724	185.107711	2022-02-02
2	164	48.476114	180.108224	2022-02-03
3	531	32.834607	162.617524	2022-02-04
4	463	67.902511	172.070975	2022-02-05

# DataFrame from ndarray

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)

```
1 pd.DataFrame(  
2     np.diag([1,2,3]),  
3     columns = ["x","y","z"]  
4 )
```

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

```
1 pd.DataFrame(  
2     np.diag([1,2,3]),  
3     columns = ["x","y","z"]  
4 )
```

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

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

	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

Selecting a column,

```
1 df[0]
```

Error: KeyError: 0

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

```
0    780
1    675
2    164
3    531
4    463
```

Name: id, dtype: int64

```
1 df.id
```

```
0    780
1    675
2    164
3    531
4    463
```

Name: id, dtype: int64

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

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

```
   id  weight  height  date
1  675  63.680724  185.107711  2022-02-02
2  164  48.476114  180.108224  2022-02-03
```

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

```
   id  weight  height  date
0  780  58.244369  180.317131  2022-02-01
2  164  48.476114  180.108224  2022-02-03
4  463  67.902511  172.070975  2022-02-05
```

# Index by position

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

```
id          675
weight     63.680724
height    185.107711
date       2022-02-02 00:00:00
Name: 1, dtype: object
```

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

```
   id  weight  height  date
1  675  63.680724  185.107711  2022-02-02
```

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

```
   id  weight  height  date
0  780  58.244369  180.317131  2022-02-01
1  675  63.680724  185.107711  2022-02-02
```

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

```
   id  weight  height  date
1  675  63.680724  185.107711  2022-02-02
3  531  32.834607  162.617524  2022-02-04
```

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

```
   weight  height
1  63.680724  185.107711
2  48.476114  180.108224
```

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

```
   id  date
0  780  2022-02-01
1  675  2022-02-02
2  164  2022-02-03
```

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

```
   id  weight
0  780  58.244369
1  675  63.680724
2  164  48.476114
```

# Index by name

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

	id	weight	height	date
anna	780	58.244369	180.317131	2022-02-01
bob	675	63.680724	185.107711	2022-02-02
carol	164	48.476114	180.108224	2022-02-03
dave	531	32.834607	162.617524	2022-02-04
erin	463	67.902511	172.070975	2022-02-05

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

id	weight	height	date
780	58.244369	180.317131	2022-02-01 00:00:00

Name: anna, dtype: object

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

	id	weight	height	date
anna	780	58.244369	180.317131	2022-02-01

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

	id	weight	height	date
bob	675	63.680724	185.107711	2022-02-02
carol	164	48.476114	180.108224	2022-02-03
dave	531	32.834607	162.617524	2022-02-04

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

	id	weight	height	date
carol	164	48.476114	180.108224	2022-02-03

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

	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]

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

	weight	height
bob	63.680724	185.107711
erin	67.902511	172.070975

```
1 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.

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

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

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

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

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

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

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

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

```
1 d
```

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

# Filtering rows

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

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

	id	weight	height	date
anna	780	58.244369	180.317131	2022-02-01

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

	id	weight	height	date
anna	780	58.244369	180.317131	2022-02-01
bob	675	63.680724	185.107711	2022-02-02
erin	463	67.902511	172.070975	2022-02-05

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

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

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

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

# Element access

```
1 df
```

	id	weight	height	date
anna	780	58.244369	180.317131	2022-02-01
bob	675	63.680724	185.107711	2022-02-02
carol	164	48.476114	180.108224	2022-02-03
dave	531	32.834607	162.617524	2022-02-04
erin	463	67.902511	172.070975	2022-02-05

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

Error: KeyError: (0, 0)

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

780

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

780

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

780

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

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

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

780

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

780

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

780



# DataFrame properties

```
1 df.size
```

```
20
```

```
1 df.shape
```

```
(5, 4)
```

```
1 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
```

```
1 df.dtypes
```

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

```
1 df.describe()
```

```
           id      weight      height
count  5.000000  5.000000  5.000000
mean   522.600000  54.227665  176.044313
std    235.423661  13.993203   8.845672
min    164.000000  32.834607  162.617524
25%    463.000000  48.476114  172.070975
50%    531.000000  58.244369  180.108224
75%    675.000000  63.680724  180.317131
max    780.000000  67.902511  185.107711
```

# 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

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

	id	weight
anna	780	58.244369
bob	675	63.680724
carol	164	48.476114
dave	531	32.834607
erin	463	67.902511

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

	weight	height
anna	58.244369	180.317131
bob	63.680724	185.107711
carol	48.476114	180.108224
dave	32.834607	162.617524
erin	67.902511	172.070975

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

	id	weight	height
anna	780	58.244369	180.317131
bob	675	63.680724	185.107711
carol	164	48.476114	180.108224
dave	531	32.834607	162.617524
erin	463	67.902511	172.070975

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

	id	weight	height	date
bob	675	63.680724	185.107711	2022-02-02
carol	164	48.476114	180.108224	2022-02-03

# Adding columns

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

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

	id	weight	height	date	student	age
anna	780	58.244369	180.317131	2022-02-01	True	19.0
bob	675	63.680724	185.107711	2022-02-02	True	22.0
carol	164	48.476114	180.108224	2022-02-03	True	25.0
dave	531	32.834607	162.617524	2022-02-04	False	NaN
erin	463	67.902511	172.070975	2022-02-05	None	NaN

```
1 df.assign(
2     student = lambda x: np.where(x.student, "yes", "no"),
3     rand = np.random.rand(5)
4 )
```

	id	weight	height	date	student	age	rand
anna	780	58.244369	180.317131	2022-02-01	yes	19.0	0.046582
bob	675	63.680724	185.107711	2022-02-02	yes	22.0	0.644103
carol	164	48.476114	180.108224	2022-02-03	yes	25.0	0.077900
dave	531	32.834607	162.617524	2022-02-04	no	NaN	0.309793
erin	463	67.902511	172.070975	2022-02-05	no	NaN	0.416640

```
1 df
```

	id	weight	height	date	student	age
anna	780	58.244369	180.317131	2022-02-01	True	19.0
bob	675	63.680724	185.107711	2022-02-02	True	22.0
carol	164	48.476114	180.108224	2022-02-03	True	25.0
dave	531	32.834607	162.617524	2022-02-04	False	NaN
erin	463	67.902511	172.070975	2022-02-05	None	NaN

# Removing columns (and rows)

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

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

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

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

	id	weight	height	date	age
anna	780	58.244369	180.317131	2022-02-01	19.0
bob	675	63.680724	185.107711	2022-02-02	22.0
carol	164	48.476114	180.108224	2022-02-03	25.0
dave	531	32.834607	162.617524	2022-02-04	NaN
erin	463	67.902511	172.070975	2022-02-05	NaN

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

	id	weight	height	date	student	age
bob	675	63.680724	185.107711	2022-02-02	True	22.0
carol	164	48.476114	180.108224	2022-02-03	True	25.0
erin	463	67.902511	172.070975	2022-02-05	None	NaN

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

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

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

	id	weight	height	date	student
anna	780	58.244369	180.317131	2022-02-01	True
bob	675	63.680724	185.107711	2022-02-02	True
carol	164	48.476114	180.108224	2022-02-03	True
dave	531	32.834607	162.617524	2022-02-04	False
erin	463	67.902511	172.070975	2022-02-05	None

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

	id	date	student	age
anna	780	2022-02-01	True	19.0
bob	675	2022-02-02	True	22.0
carol	164	2022-02-03	True	25.0
dave	531	2022-02-04	False	NaN
erin	463	2022-02-05	None	NaN

# Dropping missing values

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

```
1 df
```

	id	weight	height	date	student	age
anna	780	58.244369	180.317131	2022-02-01	True	19.0
bob	675	63.680724	185.107711	2022-02-02	True	22.0
carol	164	48.476114	180.108224	2022-02-03	True	25.0
dave	531	32.834607	162.617524	2022-02-04	False	NaN
erin	463	67.902511	172.070975	2022-02-05	None	NaN

```
1 df.dropna()
```

	id	weight	height	date	student	age
anna	780	58.244369	180.317131	2022-02-01	True	19.0
bob	675	63.680724	185.107711	2022-02-02	True	22.0
carol	164	48.476114	180.108224	2022-02-03	True	25.0

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

	id	weight	height	date	student	age
anna	780	58.244369	180.317131	2022-02-01	True	19.0
bob	675	63.680724	185.107711	2022-02-02	True	22.0
carol	164	48.476114	180.108224	2022-02-03	True	25.0
dave	531	32.834607	162.617524	2022-02-04	False	NaN
erin	463	67.902511	172.070975	2022-02-05	None	NaN

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

	id	weight	height	date
anna	780	58.244369	180.317131	2022-02-01
bob	675	63.680724	185.107711	2022-02-02
carol	164	48.476114	180.108224	2022-02-03
dave	531	32.834607	162.617524	2022-02-04
erin	463	67.902511	172.070975	2022-02-05

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

	id	weight	height	date	student
anna	780	58.244369	180.317131	2022-02-01	True
bob	675	63.680724	185.107711	2022-02-02	True
carol	164	48.476114	180.108224	2022-02-03	True
dave	531	32.834607	162.617524	2022-02-04	False
erin	463	67.902511	172.070975	2022-02-05	None

# Sorting

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

```
1 df
```

	id	weight	height	date	student	age
anna	780	58.244369	180.317131	2022-02-01	True	19.0
bob	675	63.680724	185.107711	2022-02-02	True	22.0
carol	164	48.476114	180.108224	2022-02-03	True	25.0
dave	531	32.834607	162.617524	2022-02-04	False	NaN
erin	463	67.902511	172.070975	2022-02-05	None	NaN

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

	id	weight	height	date	student	age
dave	531	32.834607	162.617524	2022-02-04	False	NaN
anna	780	58.244369	180.317131	2022-02-01	True	19.0
bob	675	63.680724	185.107711	2022-02-02	True	22.0
carol	164	48.476114	180.108224	2022-02-03	True	25.0
erin	463	67.902511	172.070975	2022-02-05	None	NaN

# Row binds

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

```
1 df1 = pd.DataFrame(  
2     np.arange(6).reshape(3,2),  
3     columns=list("xy")  
4 )  
5 df1
```

	x	y
0	0	1
1	2	3
2	4	5

```
1 df2 = pd.DataFrame(  
2     np.arange(12,6,-1).reshape(3,2),  
3     columns=list("xy")  
4 )  
5 df2
```

	x	y
0	12	11
1	10	9
2	8	7

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

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

```
1 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.

```
1 df3 = pd.DataFrame(np.ones((3,3)), columns=list("xbz"))
2 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

```
1 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`,

```
1 df1 = pd.DataFrame(  
2     np.arange(6).reshape(3,2),  
3     columns=list("xy"),  
4     index=list("abc")  
5 )  
6 df1
```

	x	y
a	0	1
b	2	3
c	4	5

```
1 df2 = pd.DataFrame(  
2     np.arange(10,6,-1).reshape(2,2),  
3     columns=list("mn"),  
4     index=list("ac")  
5 )  
6 df2
```

	m	n
a	10	9
c	8	7

```
1 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

```
1 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,

```
1 df1 = pd.DataFrame(  
2     {'a': ['foo', 'bar'], 'b': [1, 2]}  
3 )  
4 df1
```

	a	b
0	foo	1
1	bar	2

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

	a	b	c
0	foo	1	3

```
1 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

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

	a	c
0	foo	3
1	baz	4

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

	a	b	c
0	foo	1	3.0
1	bar	2	NaN

```
1 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.).

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

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x1701554b0>
```

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

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

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

	id	weight	height	age
student				
False	531.000000	32.834607	162.617524	NaN
True	539.666667	56.800402	181.844355	22.0

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

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

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

	id	weight	height	age
student				
False	531.000000	32.834607	162.617524	NaN
True	539.666667	56.800402	181.844355	22.0
NaN	463.000000	67.902511	172.070975	NaN

# Selecting groups

```
1 df
```

	id	weight	height	date	student	age
anna	780	58.244369	180.317131	2022-02-01	True	19.0
bob	675	63.680724	185.107711	2022-02-02	True	22.0
carol	164	48.476114	180.108224	2022-02-03	True	25.0
dave	531	32.834607	162.617524	2022-02-04	False	NaN
erin	463	67.902511	172.070975	2022-02-05	None	NaN

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

	id	weight	height	date	student	age
anna	780	58.244369	180.317131	2022-02-01	True	19.0
bob	675	63.680724	185.107711	2022-02-02	True	22.0
carol	164	48.476114	180.108224	2022-02-03	True	25.0

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

	id	weight	height	date	student	age
dave	531	32.834607	162.617524	2022-02-04	False	NaN

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

Error: KeyError: nan

# Aggregation

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

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

```
          id      weight      height  age
student
False  531.000000  32.834607  162.617524  NaN
True   539.666667  56.800402  181.844355  22.0
```

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

```
          id      weight  ...      height  age
          mean      std      mean  ...      std  mean  std
student
False  531.000000      NaN  32.834607  ...      NaN  NaN  NaN
True   539.666667  329.545647  56.800402  ...  2.828079  22.0  3.0
```

```
[2 rows x 8 columns]
```

