pandas

Lecture 07

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

Series

Series

The columns of a DataFrame are constructed using the `Series` class - these are a 1d array like object containing values of the same type (similar to an numpy array).

```
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.

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

```python
s
```

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

```python
s.array
```

```
<NumpyExtensionArray>
[np.int64(4), np.int64(2),
np.int64(1), np.int64(3)]
Length: 4, dtype: int64
```

```python
s.index
```

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

An index (row names) 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
```

```
<NumpyExtensionArray>
[np.int64(4), np.int64(2),
np.int64(1), np.int64(3)]
Length: 4, dtype: int64
```

```
t.index
```

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

Series + NumPy

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

```
t / 2 + 1
```

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

```
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.

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

```python
t[1]
```

```
np.int64(2)
```

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

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

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

```
np.int64(1)
```

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

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

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

```
d    3
dtype: int64
```

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

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

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

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

```python
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`,

```python
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()`?

```
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
dtype: bool
```

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

```
False
```

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

```
True
```

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

```
np.True_
```

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

```
np.False_
```

Missing via none

In some cases none can also be used as a missing value, for example:

```
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
```

This can have a side effect of changing the dtype of the series.

Native NAs

If instead of using base dtypes we use Pandas' built-in dtypes we get "native" support for missing values,

```
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 their strings accessed via the `str` attribute,

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

```
s
```

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

```
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 values!. Did you mean: 'std'?
```

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"],
    ordered=True
  )
)
```

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

DataFrames

DataFrame

- Just like R a DataFrame is a collection of vectors (Series) with a common length (and a common index)
- Column dtypes can be heterogeneous
- Columns have names stored in the `columns` index.
- It can be useful to think of a dictionary of Series objects where the keys are the column names.

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

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

```python
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

[150 rows x 5 columns]
```

Constructing DataFrames

We just saw how to read a DataFrame via `read_csv()`, `DataFrames` can also be constructed via `DataFrame()`, in general this is done using a dictionary of columns / `Series`:

```python
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')
}
```

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

```
    id     weight      height       date
0  482  64.162174  169.468134 2022-02-01
1  541  33.469345  195.730662 2022-02-02
2  213  93.782322  147.946539 2022-02-03
3  523  48.479028  164.486509 2022-02-04
4  505  70.096410  144.124685 2022-02-05
```

DataFrame from nparray

2d ndarrays can also be used to construct a `DataFrame` - generally it is a good idea to provide column 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]),
  index = ["x","y","z"]
)
```

```
   0  1  2
x  1  0  0
y  0  2  0
z  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 properties

```
df.size
```

```
20
```

```
df.shape
```

```
(5, 4)
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
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: 292.0 bytes
```

```
df.dtypes
```

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

```
df.columns
```

```
Index(['id', 'weight', 'height', 'date'],
dtype='object')
```

```
df.index
```

```
RangeIndex(start=0, stop=5, step=1)
```

```
df.axes
```

```
[RangeIndex(start=0, stop=5, step=1),
Index(['id', 'weight', 'height', 'date'],
dtype='object')]
```

DataFrame indexing

Selecting a column:

Columns can be selected by name or via . accessor,

```
df[0]
```

```
KeyError: 0
```

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

```
0    482
1    541
2    213
3    523
4    505
Name: id, dtype: int64
```

```
df.id
```

```
0    482
1    541
2    213
3    523
4    505
Name: id, dtype: int64
```

Selecting rows:

a single slice is assumed to refer to the rows

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

```
    id     weight      height       date
1  541  33.469345  195.730662 2022-02-02
2  213  93.782322  147.946539 2022-02-03
```

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

```
    id     weight      height       date
0  482  64.162174  169.468134 2022-02-01
2  213  93.782322  147.946539 2022-02-03
4  505  70.096410  144.124685 2022-02-05
```

Indexing by position

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

```
id                          541
weight                33.469345
height               195.730662
date        2022-02-02 00:00:00
Name: 1, dtype: object
```

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

```
    id     weight      height       date
1  541  33.469345  195.730662 2022-02-02
```

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

```
    id     weight      height       date
0  482  64.162174  169.468134 2022-02-01
1  541  33.469345  195.730662 2022-02-02
```

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

```
      weight      height
1  33.469345  195.730662
2  93.782322  147.946539
```

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

```
    id       date
0  482 2022-02-01
1  541 2022-02-02
2  213 2022-02-03
```

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

```
    id     weight
0  482  64.162174
1  541  33.469345
2  213  93.782322
```

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

```
    id     weight      height       date
1  541  33.469345  195.730662 2022-02-02
3  523  48.479028  164.486509 2022-02-04
```

Index by name

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

```
        id     weight      height       date
anna   482  64.162174  169.468134 2022-02-01
bob    541  33.469345  195.730662 2022-02-02
carol  213  93.782322  147.946539 2022-02-03
dave   523  48.479028  164.486509 2022-02-04
erin   505  70.096410  144.124685 2022-02-05
```

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

```
id                        482
weight              64.162174
height             169.468134
date      2022-02-01 00:00:00
Name: anna, dtype: object
```

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

```
       id     weight      height       date
anna  482  64.162174  169.468134 2022-02-01
```

```python
type(df.loc["anna"])
```

```
<class 'pandas.core.series.Series'>
```

```python
type(df.loc[["anna"]])
```

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

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

```
          id     weight      height       date
bob      541  33.469345  195.730662 2022-02-02
carol    213  93.782322  147.946539 2022-02-03
dave     523  48.479028  164.486509 2022-02-04
```

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

```
          id     weight      height       date
carol    213  93.782322  147.946539 2022-02-03
```

```python
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]
```

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

```
          weight      height
bob    33.469345  195.730662
erin   70.096410  144.124685
```

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

```
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.

```
d = pd.DataFrame(np.arange(6).reshape(
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
```

Element access

```python
df
```

```
        id     weight      height       date
anna   482  64.162174  169.468134 2022-02-01
bob    541  33.469345  195.730662 2022-02-02
carol  213  93.782322  147.946539 2022-02-03
dave   523  48.479028  164.486509 2022-02-04
erin   505  70.096410  144.124685 2022-02-05
```

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

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

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

```
np.int64(482)
```

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

```
np.int64(482)
```

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

```
np.int64(482)
```

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

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

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

```
np.int64(482)
```

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

```
np.int64(482)
```

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

```
np.int64(482)
```

Index objects

Columns and index

When constructing a DataFrame we can specify the indexes for both the rows (`index`) and columns (`columns`),

```python
df = pd.DataFrame(
  np.random.randn(5, 3),
  columns=['A', 'B', 'C']
)
df
```

```
          A         B         C
0 -0.875270  1.313213 -0.528093
1  1.136586 -0.645874 -0.945650
2  0.616353  0.541941 -0.273260
3  0.322153 -0.424912 -0.195107
4  1.491297 -0.304639  1.245868
```

```python
df.columns
```

```
Index(['A', 'B', 'C'], dtype='object')
```

```python
df.index
```

```
RangeIndex(start=0, stop=5, step=1)
```

```python
df = pd.DataFrame(
  np.random.randn(3, 3),
  index=['x','y','z'],
  columns=['A', 'B', 'C']
)
df
```

```
          A         B         C
x  1.026820  0.968886 -0.394275
y -0.394368  1.120510  0.482164
z -0.442381  1.016932  0.270455
```

```python
df.columns
```

```
Index(['A', 'B', 'C'], dtype='object')
```

```python
df.index
```

```
Index(['x', 'y', 'z'], dtype='object')
```

Index objects

pandas’ `Index` class and its subclasses provide the infrastructure necessary for lookups, data alignment, and other related tasks. You can think of them as being an immutable *multiset* (i.e. duplicate values are allowed).

```python
pd.Index(['A','B','C'])
```

```
Index(['A', 'B', 'C'], dtype='object')
```

```python
pd.Index(['A','B','C','A'])
```

```
Index(['A', 'B', 'C', 'A'], dtype='object')
```

```python
pd.Index(range(5))
```

```
RangeIndex(start=0, stop=5, step=1)
```

```python
pd.Index(list(range(5)))
```

```
Index([0, 1, 2, 3, 4], dtype='int64')
```

Index names

Index objects can have names which are shown when printing the DataFrame or Index,

```python
df = pd.DataFrame(
  np.random.randn(3, 3),
  index=pd.Index(['x','y','z'], name="rows"),
  columns=pd.Index(['A', 'B', 'C'], name="cols")
)
df
```

```
cols         A         B         C
rows                              
x     2.627667 -1.008846 -2.032781
y    -1.360111  0.195396 -0.230822
z     0.154858 -0.921222 -0.204913
```

```python
df.columns
```

```
Index(['A', 'B', 'C'], dtype='object', name='cols')
```

```python
df.index
```

```
Index(['x', 'y', 'z'], dtype='object', name='rows')
```

Indexes and missing values

It is possible for an index to contain missing values (e.g. `np.nan`) but this is generally a bad idea and should be avoided.

```python
pd.Index([1,2,3,np.nan,5])
```

```
Index([1.0, 2.0, 3.0, nan, 5.0], dtype='float64')
```

```python
pd.Index(["A","B",np.nan,"D", None])
```

```
Index(['A', 'B', nan, 'D', None], dtype='object')
```

Missing values can be replaced via the `fillna()` method,

```python
pd.Index([1,2,3,np.nan,5]).fillna(0)
```

```
Index([1.0, 2.0, 3.0, 0.0, 5.0], dtype='float64')
```

```python
pd.Index(["A","B",np.nan,"D", None]).fillna("Z")
```

```
Index(['A', 'B', 'Z', 'D', 'Z'], dtype='object')
```

Changing a DataFrame’s index

Existing columns can be made into an index via `set_index()` and removed via `reset_index()`,

```
data
```

```
     a    b  c  d
0  bar  one  z  1
1  bar  two  y  2
2  foo  one  x  3
3  foo  two  w  4
```

Creating a new index

New index values can be attached to a DataFrame via `reindex()`,

```
data
```

```
     a    b  c  d
0  bar  one  z  1
1  bar  two  y  2
2  foo  one  x  3
3  foo  two  w  4
```

```
data.reindex(columns = ["a","b","c","d","e"
```

```
     a    b  c  d   e
0  bar  one  z  1 NaN
1  bar  two  y  2 NaN
2  foo  one  x  3 NaN
3  foo  two  w  4 NaN
```

```
data.reindex(["w","x","y","z"])
```

```
     a    b    c   d
w  NaN  NaN  NaN NaN
x  NaN  NaN  NaN NaN
y  NaN  NaN  NaN NaN
z  NaN  NaN  NaN NaN
```

```
data.index = ["w","x","y","z"]; data
```

```
     a    b  c  d
w  bar  one  z  1
x  bar  two  y  2
y  foo  one  x  3
z  foo  two  w  4
```

```
data.reindex(range(4,0,-1))
```

```
     a    b    c   d
4  NaN  NaN  NaN NaN
3  NaN  NaN  NaN NaN
2  NaN  NaN  NaN NaN
1  NaN  NaN  NaN NaN
```

```
data.index = range(4,0,-1); data
```

```
     a    b  c  d
4  bar  one  z  1
3  bar  two  y  2
2  foo  one  x  3
1  foo  two  w  4
```

MultiIndexes

MultiIndex objects

These are a hierarchical analog of standard Index objects and are used to represent nested indexes. There are a number of methods for constructing them based on the initial object

```python
tuples = [('A','x'), ('A','y'),
          ('B','x'), ('B','y'),
          ('C','x'), ('C','y')]
pd.MultiIndex.from_tuples(
  tuples, names=["1st","2nd"]
)
```

```
MultiIndex([('A', 'x'),
            ('A', 'y'),
            ('B', 'x'),
            ('B', 'y'),
            ('C', 'x'),
            ('C', 'y')],
           names=['1st', '2nd'])
```

```python
pd.MultiIndex.from_product(
  [["A","B","C"],
   ["x","y"]],
  names=["1st","2nd"]
)
```

```
MultiIndex([('A', 'x'),
            ('A', 'y'),
            ('B', 'x'),
            ('B', 'y'),
            ('C', 'x'),
            ('C', 'y')],
           names=['1st', '2nd'])
```

DataFrame with MultiIndex

```python
idx = pd.MultiIndex.from_tuples(
  tuples, names=["1st","2nd"]
)

pd.DataFrame(
  np.random.rand(6,2),
  index = idx,
  columns=["m","n"]
)
```

```
                m         n
1st 2nd                    
A   x    0.790082  0.925531
    y    0.746117  0.588927
B   x    0.551212  0.967833
    y    0.834056  0.315126
C   x    0.541365  0.929809
    y    0.645596  0.860370
```

Column MultiIndex

MultiIndexes can also be used for columns as well,

```
cidx = pd.MultiIndex.from_product(
  [["A","B"],["x","y"]], names=["c1","c2"]
)

pd.DataFrame(
  np.random.rand(4,4), columns = cidx
)
```

```
c1         A                   B          
c2         x         y         x         y
0   0.046749  0.956836  0.586292  0.163044
1   0.674666  0.209365  0.535706  0.576642
2   0.940093  0.112004  0.075553  0.331692
3   0.666209  0.393802  0.217746  0.933467
```

```
ridx = pd.MultiIndex.from_product(
  [["m","n"],["l","p"]], names=["r1","r2"]
)

pd.DataFrame(
  np.random.rand(4,4),
  index= ridx, columns = cidx
)
```

```
c1            A                   B          
c2            x         y         x         y
r1 r2                                        
m  l   0.569111  0.639990  0.693537  0.170564
   p   0.367974  0.961939  0.573365  0.527121
n  l   0.637089  0.860972  0.008284  0.141591
   p   0.665466  0.060594  0.121356  0.941145
```

MultiIndex indexing

```
data
```

```
c1            A                   B          
c2            x         y         x         y
r1 r2                                        
m  l   0.361458  0.506350  0.429574  0.342180
   p   0.406104  0.755411  0.416626  0.938283
n  l   0.384019  0.734839  0.455678  0.423700
   p   0.224225  0.684089  0.813723  0.471471
```

```
data["A"]
```

```
c2            x         y
r1 r2                    
m  l   0.361458  0.506350
   p   0.406104  0.755411
n  l   0.384019  0.734839
   p   0.224225  0.684089
```

```
data["x"]
```

```
KeyError: 'x'
```

```
data["m"]
```

```
KeyError: 'm'
```

```
data["m","A"]
```

```
KeyError: ('m', 'A')
```

```
data["A","x"]
```

```
r1  r2
m   l     0.361458
    p     0.406104
n   l     0.384019
    p     0.224225
Name: (A, x), dtype: float64
```

```
data["A"]["x"]
```

```
r1  r2
m   l     0.361458
    p     0.406104
n   l     0.384019
    p     0.224225
Name: x, dtype: float64
```

MultiIndex indexing via `iloc`

```
data.iloc[0]
```

```
c1  c2
A   x     0.361458
    y     0.506350
B   x     0.429574
    y     0.342180
Name: (m, l), dtype: float64
```

```
type(data.iloc[0])
```

```
<class 'pandas.core.series.Series'>
```

```
data.iloc[(0,1)]
```

```
np.float64(0.5063499771744547)
```

```
data.iloc[[0,1]]
```

```
c1            A                   B          
c2            x         y         x         y
r1 r2                                        
m  l   0.361458  0.506350  0.429574  0.342180
   p   0.406104  0.755411  0.416626  0.938283
```

```
data.iloc[:,0]
```

```
r1  r2
m   l     0.361458
    p     0.406104
n   l     0.384019
    p     0.224225
Name: (A, x), dtype: float64
```

```
type(data.iloc[:,0])
```

```
<class 'pandas.core.series.Series'>
```

```
data.iloc[0,1]
```

```
np.float64(0.5063499771744547)
```

```
data.iloc[0,[0,1]]
```

```
c1  c2
A   x     0.361458
    y     0.506350
Name: (m, l), dtype: float64
```

MultiIndex indexing via `loc`

```python
data.loc["m"]
```

```
c1          A                   B          
c2          x         y         x         y
r2                                         
l    0.361458  0.506350  0.429574  0.342180
p    0.406104  0.755411  0.416626  0.938283
```

```python
data.loc["l"]
```

```
KeyError: 'l'
```

```python
data.loc[:,"A"]
```

```
c2             x         y
r1 r2                     
m  l    0.361458  0.506350
   p    0.406104  0.755411
n  l    0.384019  0.734839
   p    0.224225  0.684089
```

```python
data.loc[("m","l")]
```

```
c1  c2
A   x     0.361458
    y     0.506350
B   x     0.429574
    y     0.342180
Name: (m, l), dtype: float64
```

```python
data.loc[:,("A","y")]
```

```
r1  r2
m   l     0.506350
    p     0.755411
n   l     0.734839
    p     0.684089
Name: (A, y), dtype: float64
```

Fancier indexing with `loc`

Index slices can also be used with combinations of indexes and index tuples,

```
data.loc["m":"n"]
```

```
c1             A                   B          
c2             x         y         x         y
r1 r2                                          
m  l    0.361458  0.506350  0.429574  0.342180
   p    0.406104  0.755411  0.416626  0.938283
n  l    0.384019  0.734839  0.455678  0.423700
   p    0.224225  0.684089  0.813723  0.471471
```

```
data.loc[("m","l"):("n","l")]
```

```
c1             A                   B          
c2             x         y         x         y
r1 r2                                          
m  l    0.361458  0.506350  0.429574  0.342180
   p    0.406104  0.755411  0.416626  0.938283
n  l    0.384019  0.734839  0.455678  0.423700
```

```
data.loc[("m","p"):"n"]
```

```
c1             A                   B          
c2             x         y         x         y
r1 r2                                          
m  p    0.406104  0.755411  0.416626  0.938283
n  l    0.384019  0.734839  0.455678  0.423700
   p    0.224225  0.684089  0.813723  0.471471
```

```
data.loc[[("m","p"),("n","l")]]
```

```
c1             A                   B          
c2             x         y         x         y
r1 r2                                          
m  p    0.406104  0.755411  0.416626  0.938283
n  l    0.384019  0.734839  0.455678  0.423700
```

Selecting nested levels

The previous methods don’t give easy access to indexing on nested index levels, this is possible via the cross-section method `xs()`,

```
data.xs("p", level="r2")
```

```
c1          A                     B
c2          x          y          x          y
r1
m    0.406104   0.755411   0.416626   0.938283
n    0.224225   0.684089   0.813723   0.471471
```

```
data.xs("m", level="r1")
```

```
c1          A                     B
c2          x          y          x          y
r2
l    0.361458   0.506350   0.429574   0.342180
p    0.406104   0.755411   0.416626   0.938283
```

```
data.xs("y", level="c2", axis=1)
```

```
c1              A          B
r1 r2
m  l     0.506350   0.342180
   p     0.755411   0.938283
n  l     0.734839   0.423700
   p     0.684089   0.471471
```

```
data.xs("B", level="c1", axis=1)
```

```
c2              x          y
r1 r2
m  l     0.429574   0.342180
   p     0.416626   0.938283
n  l     0.455678   0.423700
   p     0.813723   0.471471
```

Setting MultiIndexes

It is also possible to construct a MultiIndex or modify an existing one using `set_index()` and `reset_index()`,

```
data
```

```
     a    b  c  d
0  bar  one  z  1
1  bar  two  y  2
2  foo  one  x  3
```

```
data.set_index(['a','b'])
```

```
         c  d
a   b
bar one  z  1
    two  y  2
foo one  x  3
```

```
data.set_index('c', append=True)
```

```
       a    b  d
  c
0 z  bar  one  1
1 y  bar  two  2
2 x  foo  one  3
```

```
data.set_index(['a','b']).reset_index()
```

```
     a    b  c  d
0  bar  one  z  1
1  bar  two  y  2
2  foo  one  x  3
```

```
data.set_index(['a','b']).reset_index(level=
```

```
       b  c  d
a
bar  one  z  1
bar  two  y  2
foo  one  x  3
```

Working with DataFrames

Filtering rows

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

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

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

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

```
        id     weight      height        date
anna   202  79.477217  162.607949  2025-02-01
bob    535  97.369002  175.888696  2025-02-02
carol  960  51.663463  156.062230  2025-02-03
dave   370  67.517056  171.197477  2025-02-04
```

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

```
        id     weight      height        date
anna   202  79.477217  162.607949  2025-02-01
carol  960  51.663463  156.062230  2025-02-03
```

```
qid = 202
df.query('id == @qid')
```

```
       id     weight      height        date
anna  202  79.477217  162.607949  2025-02-01
```

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   202  79.477217
bob    535  97.369002
carol  960  51.663463
dave   370  67.517056
erin   206  29.780742
```

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

```
          weight      height
anna   79.477217  162.607949
bob    97.369002  175.888696
carol  51.663463  156.062230
dave   67.517056  171.197477
erin   29.780742  167.607252
```

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

```
        id     weight      height
anna   202  79.477217  162.607949
bob    535  97.369002  175.888696
carol  960  51.663463  156.062230
dave   370  67.517056  171.197477
erin   206  29.780742  167.607252
```

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

```
        id     weight      height       date
anna   202  79.477217  162.607949 2025-02-01
carol  960  51.663463  156.062230 2025-02-03
dave   370  67.517056  171.197477 2025-02-04
```

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   202  79.477217  162.607949 2025-02-01    True  19.0
bob    535  97.369002  175.888696 2025-02-02    True  22.0
carol  960  51.663463  156.062230 2025-02-03    True  25.0
dave   370  67.517056  171.197477 2025-02-04   False   NaN
erin   206  29.780742  167.607252 2025-02-05    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   202  79.477217  162.607949 2025-02-01     yes  19.0  0.938553
bob    535  97.369002  175.888696 2025-02-02     yes  22.0  0.000779
carol  960  51.663463  156.062230 2025-02-03     yes  25.0  0.992212
dave   370  67.517056  171.197477 2025-02-04      no   NaN  0.617482
erin   206  29.780742  167.607252 2025-02-05      no   NaN  0.611653
```

Removing columns (and rows)

Columns or rows can be removed via the `drop()` method,

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

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

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

```
        id     weight      height       date   age
anna   202  79.477217  162.607949 2025-02-01  19.0
bob    535  97.369002  175.888696 2025-02-02  22.0
carol  960  51.663463  156.062230 2025-02-03  25.0
dave   370  67.517056  171.197477 2025-02-04   NaN
erin   206  29.780742  167.607252 2025-02-05   NaN
```

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

```
        id     weight      height       date student   age
bob    535  97.369002  175.888696 2025-02-02    True  22.0
carol  960  51.663463  156.062230 2025-02-03    True  25.0
erin   206  29.780742  167.607252 2025-02-05    None   NaN
```

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

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

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

```
        id     weight      height       date student
anna   202  79.477217  162.607949 2025-02-01    True
bob    535  97.369002  175.888696 2025-02-02    True
carol  960  51.663463  156.062230 2025-02-03    True
dave   370  67.517056  171.197477 2025-02-04   False
erin   206  29.780742  167.607252 2025-02-05    None
```

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

```
        id       date student   age
anna   202 2025-02-01    True  19.0
bob    535 2025-02-02    True  22.0
carol  960 2025-02-03    True  25.0
dave   370 2025-02-04   False   NaN
erin   206 2025-02-05    None   NaN
```

Sorting

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

```
df
```

```
        id     weight      height       date student   age
anna   202  79.477217  162.607949 2025-02-01    True  19.0
bob    535  97.369002  175.888696 2025-02-02    True  22.0
carol  960  51.663463  156.062230 2025-02-03    True  25.0
dave   370  67.517056  171.197477 2025-02-04   False   NaN
erin   206  29.780742  167.607252 2025-02-05    None   NaN
```

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

```
        id     weight      height       date student   age
dave   370  67.517056  171.197477 2025-02-04   False   NaN
carol  960  51.663463  156.062230 2025-02-03    True  25.0
bob    535  97.369002  175.888696 2025-02-02    True  22.0
anna   202  79.477217  162.607949 2025-02-01    True  19.0
erin   206  29.780742  167.607252 2025-02-05    None   NaN
```

join vs merge vs concat

All three can be used to combine data frames,

- `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”.