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

DataFrame

- Just like R a DataFrame is a collection of vectors with a common length
- Column dtypes can 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

[150 rows x 5 columns]
```

Series

Series

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

```
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
```

```
<NumpyExtensionArray>
[np.int64(4), np.int64(2),
np.int64(1), np.int64(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
```

```
<NumpyExtensionArray>
[np.int64(4), np.int64(2),
np.int64(1), np.int64(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 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]
```

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

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

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

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

```
np.int64(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 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 overriden 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),

Aside - why `np.isna()`?

```
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)
```

```
np.True_
```

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

```
np.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,

```
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,

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

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']
```

Category orders

```
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

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:

```
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')
}
```

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

	id	weight	height	date
0	227	82.008243	183.307840	2022-02-01
1	980	98.990870	156.002651	2022-02-02
2	937	72.771810	138.812473	2022-02-03
3	589	58.040194	155.248613	2022-02-04
4	669	45.101217	169.530693	2022-02-05

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]),
  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

```
1 df.size
```

```
20
```

```
1 df.shape
```

```
(5, 4)
```

```
1 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
```

```
1 df.dtypes
```

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

```
1 df.axes
```

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

```
1 df.columns
```

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

```
1 df.index
```

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

DataFrame indexing

Selecting a column:

```
1 df[0]
```

```
KeyError: 0
```

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

```
0    227
1    980
2    937
3    589
4    669
Name: id, dtype: int64
```

```
1 df.id
```

```
0    227
1    980
2    937
3    589
4    669
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	980	98.99087	156.002651	2022-02-02
2	937	72.77181	138.812473	2022-02-03

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

	id	weight	height	date
0	227	82.008243	183.307840	2022-02-01
2	937	72.771810	138.812473	2022-02-03
4	669	45.101217	169.530693	2022-02-05

Indexing by position

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

```
id                          980
weight                98.99087
height               156.002651
date      2022-02-02 00:00:00
Name: 1, dtype: object
```

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

```
    id    weight      height       date
1  980  98.99087  156.002651 2022-02-02
```

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

```
    id     weight      height       date
0  227  82.008243  183.307840 2022-02-01
1  980  98.990870  156.002651 2022-02-02
```

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

```
    id     weight      height       date
1  980  98.990870  156.002651 2022-02-02
3  589  58.040194  155.248613 2022-02-04
```

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

```
     weight      height
1  98.99087  156.002651
2  72.77181  138.812473
```

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

```
    id       date
0  227 2022-02-01
1  980 2022-02-02
2  937 2022-02-03
```

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

```
    id     weight
0  227  82.008243
1  980  98.990870
2  937  72.771810
```

Index by name

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

```
        id     weight      height       date
anna   227  82.008243  183.307840 2022-02-01
bob    980  98.990870  156.002651 2022-02-02
carol  937  72.771810  138.812473 2022-02-03
dave   589  58.040194  155.248613 2022-02-04
erin   669  45.101217  169.530693 2022-02-05
```

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

```
id                         227
weight                82.008243
height                183.30784
date      2022-02-01 00:00:00
Name: anna, dtype: object
```

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

```
        id     weight     height       date
anna   227  82.008243  183.30784 2022-02-01
```

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

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

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

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

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

	id	weight	height	date
bob	980	98.990870	156.002651	2022-02-02
carol	937	72.771810	138.812473	2022-02-03
dave	589	58.040194	155.248613	2022-02-04

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

	id	weight	height	date
anna	227	82.008243	183.30784	2022-02-01

```
1 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]
```

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

	weight	height
bob	98.990870	156.002651
erin	45.101217	169.530693

```
1 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.

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

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

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

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

```
1 d.iloc[0,1] = -1; 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  227  82.008243  183.30784 2022-02-01
```

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

```
        id     weight      height       date
anna   227  82.008243  183.307840 2022-02-01
bob    980  98.990870  156.002651 2022-02-02
carol  937  72.771810  138.812473 2022-02-03
dave   589  58.040194  155.248613 2022-02-04
```

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

```
        id     weight      height       date
bob    980  98.990870  156.002651 2022-02-02
carol  937  72.771810  138.812473 2022-02-03
dave   589  58.040194  155.248613 2022-02-04
```

```
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   227  82.008243  183.307840 2022-02-01
bob    980  98.990870  156.002651 2022-02-02
carol  937  72.771810  138.812473 2022-02-03
dave   589  58.040194  155.248613 2022-02-04
erin   669  45.101217  169.530693 2022-02-05
```

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

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

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

```
np.int64(227)
```

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

```
np.int64(227)
```

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

```
np.int64(227)
```

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

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

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

```
np.int64(227)
```

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

```
np.int64(227)
```

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

```
np.int64(227)
```

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	227	82.008243
bob	980	98.990870
carol	937	72.771810
dave	589	58.040194
erin	669	45.101217

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

	weight	height
anna	82.008243	183.307840
bob	98.990870	156.002651
carol	72.771810	138.812473
dave	58.040194	155.248613
erin	45.101217	169.530693

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

	id	weight	height
anna	227	82.008243	183.307840
bob	980	98.990870	156.002651
carol	937	72.771810	138.812473
dave	589	58.040194	155.248613
erin	669	45.101217	169.530693

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

	id	weight	height	date
bob	980	98.99087	156.002651	2022-02-02
carol	937	72.77181	138.812473	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	227	82.008243	183.307840	2022-02-01	True	19.0
bob	980	98.990870	156.002651	2022-02-02	True	22.0
carol	937	72.771810	138.812473	2022-02-03	True	25.0
dave	589	58.040194	155.248613	2022-02-04	False	NaN
erin	669	45.101217	169.530693	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	227	82.008243	183.307840	2022-02-01	yes	19.0	0.020081
bob	980	98.990870	156.002651	2022-02-02	yes	22.0	0.606127
carol	937	72.771810	138.812473	2022-02-03	yes	25.0	0.309883
dave	589	58.040194	155.248613	2022-02-04	no	NaN	0.349361
erin	669	45.101217	169.530693	2022-02-05	no	NaN	0.181331

Removing columns (and rows)

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

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

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

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

	id	weight	height	date	age
anna	227	82.008243	183.307840	2022-02-01	19.0
bob	980	98.990870	156.002651	2022-02-02	22.0
carol	937	72.771810	138.812473	2022-02-03	25.0
dave	589	58.040194	155.248613	2022-02-04	NaN
erin	669	45.101217	169.530693	2022-02-05	NaN

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

	id	weight	height	date	student	age
bob	980	98.990870	156.002651	2022-02-02	True	22.0
carol	937	72.771810	138.812473	2022-02-03	True	25.0
erin	669	45.101217	169.530693	2022-02-05	None	NaN

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

```
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   227  82.008243  183.307840 2022-02-01    True
bob    980  98.990870  156.002651 2022-02-02    True
carol  937  72.771810  138.812473 2022-02-03    True
dave   589  58.040194  155.248613 2022-02-04   False
erin   669  45.101217  169.530693 2022-02-05    None
```

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

```
        id       date student   age
anna   227 2022-02-01    True  19.0
bob    980 2022-02-02    True  22.0
carol  937 2022-02-03    True  25.0
dave   589 2022-02-04   False   NaN
erin   669 2022-02-05    None   NaN
```

Sorting

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

```
1 df
```

	id	weight	height	date	student	age
anna	227	82.008243	183.307840	2022-02-01	True	19.0
bob	980	98.990870	156.002651	2022-02-02	True	22.0
carol	937	72.771810	138.812473	2022-02-03	True	25.0
dave	589	58.040194	155.248613	2022-02-04	False	NaN
erin	669	45.101217	169.530693	2022-02-05	None	NaN

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

	id	weight	height	date	student	age
dave	589	58.040194	155.248613	2022-02-04	False	NaN
bob	980	98.990870	156.002651	2022-02-02	True	22.0
carol	937	72.771810	138.812473	2022-02-03	True	25.0
anna	227	82.008243	183.307840	2022-02-01	True	19.0
erin	669	45.101217	169.530693	2022-02-05	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”.

Index objects

Columns and index

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

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

```
          A         B         C
0  0.002727 -1.369377  0.688220
1 -0.550959  0.664234 -0.541148
2 -0.280580  1.274335  0.622401
3 -1.066624  0.335441  0.455829
4  0.371230  0.973063 -0.455515
```

```
df.columns
```

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

```
df.index
```

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

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

```
          A         B         C
x -0.514988  0.190823 -0.075768
y -0.317333 -0.463206  0.585036
z  0.301767 -0.144967  1.845279
```

```
df.columns
```

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

```
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* (duplicate values are allowed).

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

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

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

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

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

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

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

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

Index names

Index objects can have names which show when displaying the DataFrame or Index,

```
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    -0.983048  0.597640 -0.292334
y    -0.454094 -0.080177  0.551344
z    -0.991798 -0.396921  0.098984
```

```
df.columns
```

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

```
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.

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

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

```
1 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,

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

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

```
1 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 an index via `set_index()` and removed via `reset_index()`,

```
1 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
```

```
1 data.set_index('a')
```

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

```
1 data.set_index('c', drop=False)
```

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

```
1 data.set_index('a').reset_index()
```

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

```
1 data.set_index('c').reset_index(drop=True)
```

```
     a    b  d
0  bar  one  1
1  bar  two  2
2  foo  one  3
3  foo  two  4
```

Creating a new index

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

```
1 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
```

```
1 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
```

```
1 data.reindex(range(5,-1,-1))
```

```
     a    b    c    d
5  NaN  NaN  NaN  NaN
4  NaN  NaN  NaN  NaN
3  foo  two    w  4.0
2  foo  one    x  3.0
1  bar  two    y  2.0
0  bar  one    z  1.0
```

```
1 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
```

```
1 data.index = ["w","x","y","z"]
2 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
```

MultiIndexes

MultiIndex objects

These are a hierarchical analog of standard Index objects (nested indexes). There are a number of methods for constructing them based on the initial object

```
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'])
```

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

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

DataFrame with MultiIndex

```
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    0.621798  0.876034
    y    0.317657  0.608538
B   x    0.470411  0.170864
    y    0.681680  0.605478
C   x    0.857115  0.915991
    y    0.991827  0.552677
```

Column MultiIndex

MultiIndexes can also be used for columns (or both rows and columns),

```
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	0.718345	0.223572	0.394455	0.632339
1	0.165013	0.918670	0.076620	0.741082
2	0.930161	0.568155	0.292802	0.737838
3	0.364802	0.781620	0.995068	0.996934

```
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	0.010682	0.328527	0.854047	0.262931
	p	0.121074	0.722093	0.770239	0.381377
n	l	0.052525	0.444460	0.634272	0.424026
	p	0.004650	0.395490	0.285159	0.519998

MultiIndex indexing

```
1 data
```

```
c1            A                   B
c2            x         y         x         y
r1 r2
m  l   0.868151  0.347342  0.335235  0.073226
   p   0.570382  0.468908  0.936485  0.320975
n  l   0.712225  0.341401  0.543012  0.749072
   p   0.469549  0.901303  0.903090  0.597538
```

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

```
c2            x         y
r1 r2
m  l   0.868151  0.347342
   p   0.570382  0.468908
n  l   0.712225  0.341401
   p   0.469549  0.901303
```

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

```
KeyError: 'x'
```

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

```
KeyError: 'm'
```

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

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

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

```
r1  r2
m   l     0.868151
    p     0.570382
n   l     0.712225
    p     0.469549
Name: (A, x), dtype: float64
```

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

```
r1  r2
m   l     0.868151
    p     0.570382
n   l     0.712225
    p     0.469549
Name: x, dtype: float64
```

MultiIndex indexing via `iloc`

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

```
c1  c2
A   x     0.868151
    y     0.347342
B   x     0.335235
    y     0.073226
Name: (m, l), dtype: float64
```

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

```
np.float64(0.3473421714313134)
```

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

```
c1            A                   B
c2            x         y         x         y
r1 r2
m  l   0.868151  0.347342  0.335235  0.073226
   p   0.570382  0.468908  0.936485  0.320975
```

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

```
r1  r2
m   l     0.868151
    p     0.570382
n   l     0.712225
    p     0.469549
Name: (A, x), dtype: float64
```

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

```
np.float64(0.3473421714313134)
```

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

```
c1  c2
A   x     0.868151
    y     0.347342
Name: (m, l), dtype: float64
```

MultiIndex indexing via `loc`

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

```
c1         A                   B
c2         x         y         x         y
r2
l   0.868151  0.347342  0.335235  0.073226
p   0.570382  0.468908  0.936485  0.320975
```

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

```
KeyError: 'l'
```

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

```
c2            x         y
r1 r2
m  l   0.868151  0.347342
   p   0.570382  0.468908
n  l   0.712225  0.341401
   p   0.469549  0.901303
```

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

```
c1  c2
A   x     0.868151
    y     0.347342
B   x     0.335235
    y     0.073226
Name: (m, l), dtype: float64
```

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

```
r1  r2
m   l     0.347342
    p     0.468908
n   l     0.341401
    p     0.901303
Name: (A, y), dtype: float64
```

Fancier indexing with `loc`

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

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

c1		A		B	
c2		x	y	x	y
r1	r2				
m	l	0.868151	0.347342	0.335235	0.073226
	p	0.570382	0.468908	0.936485	0.320975
n	l	0.712225	0.341401	0.543012	0.749072
	p	0.469549	0.901303	0.903090	0.597538

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

c1		A		B	
c2		x	y	x	y
r1	r2				
m	l	0.868151	0.347342	0.335235	0.073226
	p	0.570382	0.468908	0.936485	0.320975
n	l	0.712225	0.341401	0.543012	0.749072

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

c1		A		B	
c2		x	y	x	y
r1	r2				
m	p	0.570382	0.468908	0.936485	0.320975
n	l	0.712225	0.341401	0.543012	0.749072
	p	0.469549	0.901303	0.903090	0.597538

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

c1		A		B	
c2		x	y	x	y
r1	r2				
m	p	0.570382	0.468908	0.936485	0.320975
n	l	0.712225	0.341401	0.543012	0.749072

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()`,

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

c1	A		B	
c2	x	y	x	y
r1				
m	0.570382	0.468908	0.936485	0.320975
n	0.469549	0.901303	0.903090	0.597538

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

c1	A		B	
c2	x	y	x	y
r2				
l	0.868151	0.347342	0.335235	0.073226
p	0.570382	0.468908	0.936485	0.320975

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

c1		A	B
r1	r2		
m	l	0.347342	0.073226
	p	0.468908	0.320975
n	l	0.341401	0.749072
	p	0.901303	0.597538

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

c2		x	y
r1	r2		
m	l	0.335235	0.073226
	p	0.936485	0.320975
n	l	0.543012	0.749072
	p	0.903090	0.597538

Setting MultiIndexes

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

```
1 data
```

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

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

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

```
1 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
```

```
1 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
```

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

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