# pandas

## Lecture 09

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

```
1 import pandas as pd
2 pd.__version__
```

```
'3.0.0'
```

# 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 a 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: str
```

```
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 the `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>
[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
```

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

```
1 t.index
```

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

# 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 subset by their index values (not position) or by logical expressions (as with NumPy arrays or R vectors)

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

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

```
np.int64(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; t
```

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

# Position based indexing

Series cannot be directly indexed by position, but the `iloc` *attribute* can be used for this purpose.

```
1 t[1]
```

```
KeyError: 1
```

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

```
KeyError: "None of [Index([1, 2], dtype='int64')] are in the [index]"
```

```
1 t.iloc[1]
```

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

```
1 t.iloc[[1,2]]
```

```
b    2
c    1
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: str
```

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: str
```

# Missing values

Pandas encodes missing values using NaN (mostly),

```
s1 = pd.Series(
  {"anna": "A+", "bob": "B-",
   "carol": "C", "dave": "D+"},
  index = ["erin","dave","carol",
           "bob","anna"]
)
```

```
s2 = pd.Series(
  {"anna": 97, "bob": 82,
   "carol": 75, "dave": 68},
  index = ["erin","dave","carol",
           "bob","anna"],
  dtype = 'int64'
)
```

```
s1
```

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

```
s2
```

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

```
pd.isna(s1)
```

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

```
pd.isna(s2)
```

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

# Aside - why `pd.isna()`?

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

# Missing via None

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

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

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

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

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

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

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

```
1 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,

```
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 have 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: str
```

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

```
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: str
```

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

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

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

```
AttributeError: Can only use .str accessor with string values, not integer. Did you mean: 'std'?
```

# Categorical Series

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

```
0    Mon
1    Tue
2    Wed
3    Thu
4    Fri
dtype: str
```

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

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

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

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

# Category orders

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

```
0    Tue
1    Thu
2    Mon
3    NaN
dtype: category
Categories (5, str): ['Mon' < 'Tue' < 'Wed' < 'Thu' < '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.

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

```
<class 'pandas.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 × 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`.

```
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='2/1/2026', periods=n, freq='D')
}
```

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

| | id | weight | height | date |
|---|---|---|---|---|
| **0** | 200 | 57.715447 | 156.243857 | 2026-02-01 |
| **1** | 995 | 69.552249 | 196.166608 | 2026-02-02 |
| **2** | 238 | 48.327161 | 178.808975 | 2026-02-03 |
| **3** | 768 | 126.885781 | 168.055900 | 2026-02-04 |
| **4** | 342 | 93.758617 | 181.371930 | 2026-02-05 |

# 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 attributes & methods

```
1 df.size
```

```
20
```

```
1 df.shape
```

```
(5, 4)
```

```
1 df.info()
```

```
<class 'pandas.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[us]
dtypes: datetime64[us](1), float64(2), int64(1)
memory usage: 292.0 bytes
```

```
1 df.dtypes
```

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

```
1 df.columns
```

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

```
1 df.index
```

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

```
1 df.axes
```

```
[RangeIndex(start=0, stop=5, step=1), Index(['id
```

# DataFrame indexing

Selecting a column - use name or via the `.` accessor,

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

```
0    200
1    995
2    238
3    768
4    342
Name: id, dtype: int64
```

```
1 df.id
```

```
0    200
1    995
2    238
3    768
4    342
Name: id, dtype: int64
```

Selecting rows - use a single slice

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

| | id | weight | height | date |
|---|---|---|---|---|
| **1** | 995 | 69.552249 | 196.166608 | 2026-02-02 |
| **2** | 238 | 48.327161 | 178.808975 | 2026-02-03 |

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

| | id | weight | height | date |
|---|---|---|---|---|
| **0** | 200 | 57.715447 | 156.243857 | 2026-02-01 |
| **2** | 238 | 48.327161 | 178.808975 | 2026-02-03 |
| **4** | 342 | 93.758617 | 181.371930 | 2026-02-05 |

Integer based subsets are not allowed:

```
1 df[0]
```

```
KeyError: 0
```

# Indexing by position

As with Series, position based indexing is done via the `iloc` attribute

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

```
id                          995
weight                69.552249
height               196.166608
date        2026-02-02 00:00:00
Name: 1, dtype: object
```

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

| | id | weight | height | date |
|---|---|---|---|---|
| **1** | 995 | 69.552249 | 196.166608 | 2026-02-02 |

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

| | id | weight | height | date |
|---|---|---|---|---|
| **0** | 200 | 57.715447 | 156.243857 | 2026-02-01 |
| **1** | 995 | 69.552249 | 196.166608 | 2026-02-02 |

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

| | weight | height |
|---|---|---|
| **1** | 69.552249 | 196.166608 |
| **2** | 48.327161 | 178.808975 |

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

| | id | date |
|---|---|---|
| **0** | 200 | 2026-02-01 |
| **1** | 995 | 2026-02-02 |
| **2** | 238 | 2026-02-03 |

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

| | id | weight |
|---|---|---|
| **0** | 200 | 57.715447 |
| **1** | 995 | 69.552249 |
| **2** | 238 | 48.327161 |

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

| | id | weight | height | date |
|---|---|---|---|---|
| **1** | 995 | 69.552249 | 196.166608 | 2026-02-02 |
| **3** | 768 | 126.885781 | 168.055900 | 2026-02-04 |

# Index by name

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

| | id | weight | height | date |
|---|---|---|---|---|
| **anna** | 200 | 57.715447 | 156.243857 | 2026-02-01 |
| **bob** | 995 | 69.552249 | 196.166608 | 2026-02-02 |
| **carol** | 238 | 48.327161 | 178.808975 | 2026-02-03 |
| **dave** | 768 | 126.885781 | 168.055900 | 2026-02-04 |
| **erin** | 342 | 93.758617 | 181.371930 | 2026-02-05 |

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

```
id                          200
weight                57.715447
height               156.243857
date      2026-02-01 00:00:00
Name: anna, dtype: object
```

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

| | id | weight | height | date |
|---|---|---|---|---|
| **anna** | 200 | 57.715447 | 156.243857 | 2026-02-01 |

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

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

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

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

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

| | id | weight | height | date |
|---|---|---|---|---|
| **bob** | 995 | 69.552249 | 196.166608 | 2026-02-02 |
| **carol** | 238 | 48.327161 | 178.808975 | 2026-02-03 |
| **dave** | 768 | 126.885781 | 168.055900 | 2026-02-04 |

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

| | id | weight | height | date |
|---|---|---|---|---|
| **anna** | 200 | 57.715447 | 156.243857 | 2026-02-01 |
| **carol** | 238 | 48.327161 | 178.808975 | 2026-02-03 |

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

```
anna    2026-02-01
bob     2026-02-02
carol   2026-02-03
dave    2026-02-04
erin    2026-02-05
Name: date, dtype: datetime64[us]
```

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

| | weight | height |
|---|---|---|
| **bob** | 69.552249 | 196.166608 |
| **erin** | 93.758617 | 181.371930 |

```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. Previously some subset operations would return views, this has mostly been resolved with pandas 3.0.0.

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

| | 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; d
```

| | x | y |
|---|---|---|
| **0** | 0 | -1 |
| **1** | 2 | 3 |
| **2** | 4 | 5 |

```
1 v
```

| | x | y |
|---|---|---|
| **0** | 0 | 1 |
| **1** | 2 | 3 |

```
1 v.iloc[0,0] = 100; v
```

| | x | y |
|---|---|---|
| **0** | 100 | 1 |
| **1** | 2 | 3 |

```
1 d
```

| | x | y |
|---|---|---|
| **0** | 0 | -1 |
| **1** | 2 | 3 |
| **2** | 4 | 5 |

# Element access

```
df
```

| | id | weight | height | date |
|---|---|---|---|---|
| **anna** | 200 | 57.715447 | 156.243857 | 2026-02-01 |
| **bob** | 995 | 69.552249 | 196.166608 | 2026-02-02 |
| **carol** | 238 | 48.327161 | 178.808975 | 2026-02-03 |
| **dave** | 768 | 126.885781 | 168.055900 | 2026-02-04 |
| **erin** | 342 | 93.758617 | 181.371930 | 2026-02-05 |

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

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

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

```
np.int64(200)
```

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

```
KeyError: 0
```

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

```
KeyError: 0
```

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

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

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

```
np.int64(200)
```

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

```
np.int64(200)
```

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

```
KeyError: 0
```

# Index objects

# Columns and index

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

```
df1 = pd.DataFrame(
  np.random.randn(5, 3),
  columns=['A', 'B', 'C']
)
df1
```

| | A | B | C |
|---|---|---|---|
| 0 | 0.555298 | 1.237481 | -1.096399 |
| 1 | -0.457000 | -0.817442 | -1.658612 |
| 2 | -0.943632 | -0.190040 | -0.850275 |
| 3 | -0.325382 | -0.152456 | 0.426736 |
| 4 | 1.276448 | -1.010736 | 1.063402 |

```
df2 = pd.DataFrame(
  np.random.randn(3, 3),
  index=['x','y','z'],
  columns=['A', 'B', 'C']
)
df2
```

| | A | B | C |
|---|---|---|---|
| x | -1.080576 | -0.032645 | -0.379919 |
| y | -2.691178 | -0.247386 | 0.541314 |
| z | -0.217621 | 1.012104 | -0.367918 |

```
df1.columns
```

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

```
df2.columns
```

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

```
df1.index
```

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

```
df2.index
```

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

# 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).

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

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

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

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

```
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 are shown when printing 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 | -0.649065 | -0.881551 | 1.478708 |
| y | 1.534376 | 0.202521 | 0.195817 |
| z | 0.834751 | 1.269891 | -1.427342 |

```
df.columns
```

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

```
df.index
```

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

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='str')
```

# Changing a DataFrame's index

Existing columns can be made into 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('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 |

# drop argument

```
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('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(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.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.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 |

```
1 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 |

```
1 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

```
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","2nd"]
)
```

```
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** | 0.432674 | 0.954728 |
| | **y** | 0.519791 | 0.771165 |
| **B** | **x** | 0.949299 | 0.521805 |
| | **y** | 0.809213 | 0.208332 |
| **C** | **x** | 0.777001 | 0.002249 |
| | **y** | 0.762001 | 0.976363 |

# 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 | 0.660493 | 0.181417 | 0.784601 | 0.691375 |
| 1 | 0.137302 | 0.545351 | 0.640257 | 0.525258 |
| 2 | 0.009936 | 0.646707 | 0.128306 | 0.315168 |
| 3 | 0.320345 | 0.371164 | 0.558344 | 0.280129 |

```
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 | 0.524746 | 0.096756 | 0.664596 | 0.450048 |
| | p | 0.122310 | 0.842365 | 0.867014 | 0.858448 |
| n | l | 0.465433 | 0.998055 | 0.875562 | 0.567632 |
| | p | 0.911802 | 0.745366 | 0.449562 | 0.006176 |

# MultiIndex indexing

```
1 data
```

| | c1 | A | | B | |
|---|---|---|---|---|---|
| | c2 | x | y | x | y |
| r1 | r2 | | | | |
| m | l | 0.179763 | 0.787379 | 0.565183 | 0.665208 |
| | p | 0.220432 | 0.483390 | 0.292793 | 0.589280 |
| n | l | 0.823848 | 0.654454 | 0.494629 | 0.924918 |
| | p | 0.344536 | 0.541508 | 0.544303 | 0.591460 |

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

| | c2 | x | y |
|---|---|---|---|
| r1 | r2 | | |
| m | l | 0.179763 | 0.787379 |
| | p | 0.220432 | 0.483390 |
| n | l | 0.823848 | 0.654454 |
| | p | 0.344536 | 0.541508 |

```
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     0.179763
    p     0.220432
n   l     0.823848
    p     0.344536
Name: (A, x), dtype: float64
```

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

```
r1  r2
m   l     0.179763
    p     0.220432
n   l     0.823848
    p     0.344536
Name: x, dtype: float64
```

# MultiIndex indexing via `iloc`

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

```
c1  c2
A   x     0.179763
    y     0.787379
B   x     0.565183
    y     0.665208
Name: (m, l), dtype: float64
```

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

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

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

```
np.float64(0.7873788607957514)
```

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

| | c1 | A | | B | |
|---|---|---|---|---|---|
| | c2 | x | y | x | y |
| r1 | r2 | | | | |
| m | l | 0.179763 | 0.787379 | 0.565183 | 0.665208 |
| | p | 0.220432 | 0.483390 | 0.292793 | 0.589280 |

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

```
r1  r2
m   l     0.179763
    p     0.220432
n   l     0.823848
    p     0.344536
Name: (A, x), dtype: float64
```

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

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

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

```
np.float64(0.7873788607957514)
```

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

```
c1  c2
A   x     0.179763
    y     0.787379
Name: (m, l), dtype: float64
```

# MultiIndex indexing via `loc`

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

| c1 | A | | B | |
|---|---|---|---|---|
| c2 | x | y | x | y |
| r2 | | | | |
| l | 0.179763 | 0.787379 | 0.565183 | 0.665208 |
| p | 0.220432 | 0.483390 | 0.292793 | 0.589280 |

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

```
KeyError: 'l'
```

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

| | c2 | x | y |
|---|---|---|---|
| r1 | r2 | | |
| m | l | 0.179763 | 0.787379 |
| | p | 0.220432 | 0.483390 |
| n | l | 0.823848 | 0.654454 |
| | p | 0.344536 | 0.541508 |

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

```
c1  c2
A   x     0.179763
    y     0.787379
B   x     0.565183
    y     0.665208
Name: (m, l), dtype: float64
```

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

```
r1  r2
m   l     0.787379
    p     0.483390
n   l     0.654454
    p     0.541508
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 | 0.179763 | 0.787379 | 0.565183 | 0.665208 |
| | p | 0.220432 | 0.483390 | 0.292793 | 0.589280 |
| n | l | 0.823848 | 0.654454 | 0.494629 | 0.924918 |
| | p | 0.344536 | 0.541508 | 0.544303 | 0.591460 |

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

| | c1 | A | | B | |
|---|---|---|---|---|---|
| | c2 | x | y | x | y |
| r1 | r2 | | | | |
| m | p | 0.220432 | 0.483390 | 0.292793 | 0.589280 |
| n | l | 0.823848 | 0.654454 | 0.494629 | 0.924918 |
| | p | 0.344536 | 0.541508 | 0.544303 | 0.591460 |

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

| | c1 | A | | B | |
|---|---|---|---|---|---|
| | c2 | x | y | x | y |
| r1 | r2 | | | | |
| m | l | 0.179763 | 0.787379 | 0.565183 | 0.665208 |
| | p | 0.220432 | 0.483390 | 0.292793 | 0.589280 |
| n | l | 0.823848 | 0.654454 | 0.494629 | 0.924918 |

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

| | c1 | A | | B | |
|---|---|---|---|---|---|
| | c2 | x | y | x | y |
| r1 | r2 | | | | |
| m | p | 0.220432 | 0.483390 | 0.292793 | 0.589280 |
| n | l | 0.823848 | 0.654454 | 0.494629 | 0.924918 |

# 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 | 0.220432 | 0.483390 | 0.292793 | 0.58928 |
| n | 0.344536 | 0.541508 | 0.544303 | 0.59146 |

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

| | c1 | A | B |
|---|---|---|---|
| r1 | r2 | | |
| m | l | 0.787379 | 0.665208 |
| | p | 0.483390 | 0.589280 |
| n | l | 0.654454 | 0.924918 |
| | p | 0.541508 | 0.591460 |

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

| c1 | A | | B | |
|---|---|---|---|---|
| c2 | x | y | x | y |
| r2 | | | | |
| l | 0.179763 | 0.787379 | 0.565183 | 0.665208 |
| p | 0.220432 | 0.483390 | 0.292793 | 0.589280 |

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

| | c2 | x | y |
|---|---|---|---|
| r1 | r2 | | |
| m | l | 0.565183 | 0.665208 |
| | p | 0.292793 | 0.589280 |
| n | l | 0.494629 | 0.924918 |
| | p | 0.544303 | 0.591460 |

# 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(['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('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(level=1)
```

| | 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.

```
1 df.query('date == "2026-02-01"')
```

| | id | weight | height | date |
|---|---|---|---|---|

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

| | id | weight | height | date |
|---|---|---|---|---|
| **anna** | 202 | 82.953771 | 193.688192 | 2026-02-01 |
| **bob** | 535 | 100.460597 | 181.511521 | 2026-02-02 |
| **carol** | 960 | 65.316933 | 162.957884 | 2026-02-03 |
| **dave** | 370 | 65.317261 | 178.138401 | 2026-02-04 |

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

| | id | weight | height | date |
|---|---|---|---|---|
| **carol** | 960 | 65.316933 | 162.957884 | 2026-02-03 |

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

| | id | weight | height | date |
|---|---|---|---|---|
| **anna** | 202 | 82.953771 | 193.688192 | 2026-02-01 |

# 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

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

| | id | weight |
|---|---|---|
| **anna** | 202 | 82.953771 |
| **bob** | 535 | 100.460597 |
| **carol** | 960 | 65.316933 |
| **dave** | 370 | 65.317261 |

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

| | weight | height |
|---|---|---|
| **anna** | 82.953771 | 193.688192 |
| **bob** | 100.460597 | 181.511521 |
| **carol** | 65.316933 | 162.957884 |
| **dave** | 65.317261 | 178.138401 |

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

| | id | weight | height |
|---|---|---|---|
| **anna** | 202 | 82.953771 | 193.688192 |
| **bob** | 535 | 100.460597 | 181.511521 |
| **carol** | 960 | 65.316933 | 162.957884 |
| **dave** | 370 | 65.317261 | 178.138401 |

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

| | id | weight | height | date |
|---|---|---|---|---|
| **anna** | 202 | 82.953771 | 193.688192 | 2026-02-01 |
| **carol** | 960 | 65.316933 | 162.957884 | 2026-02-03 |
| **dave** | 370 | 65.317261 | 178.138401 | 2026-02-04 |

# 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, None]
df['age'] = [19, 22, None, None]
df
```

| | id | weight | height | date | student | age |
|---|---|---|---|---|---|---|
| **anna** | 202 | 82.953771 | 193.688192 | 2026-02-01 | True | 19.0 |
| **bob** | 535 | 100.460597 | 181.511521 | 2026-02-02 | True | 22.0 |
| **carol** | 960 | 65.316933 | 162.957884 | 2026-02-03 | True | NaN |
| **dave** | 370 | 65.317261 | 178.138401 | 2026-02-04 | None | NaN |

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

| | id | weight | height | date | student | age | rand |
|---|---|---|---|---|---|---|---|
| **anna** | 202 | 82.953771 | 193.688192 | 2026-02-01 | yes | 19.0 | 0.181825 |
| **bob** | 535 | 100.460597 | 181.511521 | 2026-02-02 | yes | 22.0 | 0.183405 |
| **carol** | 960 | 65.316933 | 162.957884 | 2026-02-03 | yes | NaN | 0.304242 |
| **dave** | 370 | 65.317261 | 178.138401 | 2026-02-04 | no | NaN | 0.524756 |

# pd.col()

As of pandas 3.0.0 there is also a `pd.col()` function which can be used within methods like `assign()` and `query()` to refer to columns by name,

```
df.assign(
  bmi = pd.col("weight") / np.power(pd.col("height")/100, 2)
)
```

| | id | weight | height | date | student | age | bmi |
|---|---|---|---|---|---|---|---|
| **anna** | 202 | 82.953771 | 193.688192 | 2026-02-01 | True | 19.0 | 22.112092 |
| **bob** | 535 | 100.460597 | 181.511521 | 2026-02-02 | True | 22.0 | 30.492102 |
| **carol** | 960 | 65.316933 | 162.957884 | 2026-02-03 | True | NaN | 24.596597 |
| **dave** | 370 | 65.317261 | 178.138401 | 2026-02-04 | None | NaN | 20.583199 |

It is still early days for `pd.col()` so expect some rough edges - e.g. `(pd.col("height")/100) ** 2` did not work as

# 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** | 202 | 82.953771 | 193.688192 | 2026-02-01 | 19.0 |
| **bob** | 535 | 100.460597 | 181.511521 | 2026-02-02 | 22.0 |
| **carol** | 960 | 65.316933 | 162.957884 | 2026-02-03 | NaN |
| **dave** | 370 | 65.317261 | 178.138401 | 2026-02-04 | NaN |

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

| | id | weight | height | date | student | age |
|---|---|---|---|---|---|---|
| **bob** | 535 | 100.460597 | 181.511521 | 2026-02-02 | True | 22.0 |
| **carol** | 960 | 65.316933 | 162.957884 | 2026-02-03 | True | 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** | 202 | 82.953771 | 193.688192 | 2026-02-01 | True |
| **bob** | 535 | 100.460597 | 181.511521 | 2026-02-02 | True |
| **carol** | 960 | 65.316933 | 162.957884 | 2026-02-03 | True |
| **dave** | 370 | 65.317261 | 178.138401 | 2026-02-04 | None |

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

| | id | date | student | age |
|---|---|---|---|---|
| **anna** | 202 | 2026-02-01 | True | 19.0 |
| **bob** | 535 | 2026-02-02 | True | 22.0 |
| **carol** | 960 | 2026-02-03 | True | NaN |
| **dave** | 370 | 2026-02-04 | None | NaN |

# Sorting

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

```
1 df
```

| | id | weight | height | date | student | age |
|---|---|---|---|---|---|---|
| **anna** | 202 | 82.953771 | 193.688192 | 2026-02-01 | True | 19.0 |
| **bob** | 535 | 100.460597 | 181.511521 | 2026-02-02 | True | 22.0 |
| **carol** | 960 | 65.316933 | 162.957884 | 2026-02-03 | True | NaN |
| **dave** | 370 | 65.317261 | 178.138401 | 2026-02-04 | None | NaN |

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

| | id | weight | height | date | student | age |
|---|---|---|---|---|---|---|
| **carol** | 960 | 65.316933 | 162.957884 | 2026-02-03 | True | NaN |
| **bob** | 535 | 100.460597 | 181.511521 | 2026-02-02 | True | 22.0 |
| **anna** | 202 | 82.953771 | 193.688192 | 2026-02-01 | True | 19.0 |
| **dave** | 370 | 65.317261 | 178.138401 | 2026-02-04 | 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”.

# Next week

- Pivoting
- Split-Apply-Combine (group by / summarize)
- Pandas & pyarrow
- Polars