more pandas

Lecture 09

Dr. Colin Rundel

Index objects

Columns and Indexes

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  1.214010  1.293817 -2.529983
1 -0.979902 -0.985834 -1.479055
2 -0.458005  0.527957 -0.402826
3  1.488340  0.380165  0.020759
4 -1.092834 -0.125232  0.625418
```

```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  0.777821  2.477700  0.814513
y  1.737038 -1.112535 -0.723814
z  0.041552 -1.455307  0.417292
```

```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* (duplicate values are allowed).

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

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

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

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

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

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

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

```
Int64Index([0, 1, 2, 3, 4], dtype='int64')
```

Indexes as sets

While it is not something you will need to do very often, since Indexes are "sets" the various set operations and methods are available.

```
a = pd.Index(['c', 'b', 'a'])
b = pd.Index(['c', 'e', 'd'])
```

```
a.union(b)
```

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

```
a.difference(b)
```

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

```
a.intersection(b)
```

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

```
a.symmetric_difference(b)
```

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

```
c = pd.Index([1.0, 1.5, 2.0])
d = pd.Index(range(5))

c.union(d)
```

```
Float64Index([0.0, 1.0, 1.5, 2.0, 3.0, 4.0],
dtype='float64')
```

```
e = pd.Index(["A","B","C"])
f = pd.Index(range(5))

e.union(f)
```

```
Index(['A', 'B', 'C', 0, 1, 2, 3, 4],
dtype='object')
```

Index metadata

You can attach names to an index, which will then 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     -0.779489  1.053635  1.439538
y      1.732166 -0.471819  0.437045
z     -1.624165 -0.074081  1.694972
```

```
df.columns
```

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

```
df.index
```

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

```
df.columns.rename("m")
```

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

```
df.index.set_names("n")
```

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

```
df
```

```
cols          A         B         C
rows
x     -0.779489  1.053635  1.439538
y      1.732166 -0.471819  0.437045
z     -1.624165 -0.074081  1.694972
```

Renaming indexes inplace

If you want to change the index names inplace either assign directly to the `name` attribute or use the `inplace=TRUE` argument with `rename()`.

```
df
```

```
cols         A         B         C
rows
x    -0.779489  1.053635  1.439538
y     1.732166 -0.471819  0.437045
z    -1.624165 -0.074081  1.694972
```

```
df.columns.name = "o"
df.index.name = "p"
df
```

```
o         A         B         C
p
x -0.779489  1.053635  1.439538
y  1.732166 -0.471819  0.437045
z -1.624165 -0.074081  1.694972
```

```
df.columns.rename("q", inplace=True)
df.index.rename("r", inplace=True)
df
```

```
q         A         B         C
r
x -0.779489  1.053635  1.439538
y  1.732166 -0.471819  0.437045
z -1.624165 -0.074081  1.694972
```

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.

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

```
Float64Index([1.0, 2.0, 3.0, nan, 5.0], dtype='float64')
```

```
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,

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

```
Float64Index([1.0, 2.0, 3.0, 0.0, 5.0], dtype='float64')
```

```
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()`,

```python
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
```

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

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

```python
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
```

```python
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
```

```python
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()`,

```python
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

```python
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

```python
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

```python
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

```python
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

Renaming levels

Alternatively, row or column index levels can be renamed via `rename()`,

```
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.rename(index = pd.Series(["m","n","o","p"]))
```

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

```
data.rename_axis(index="rows")
```

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

```
data.rename(columns = {"a":"w", "b":"x",
                       "c":"y", "d":"z"})
```

```
     w    x  y  z
0  bar  one  z  1
1  bar  two  y  2
2  foo  one  x  3
3  foo  two  w  4
```

```
data.rename_axis(columns="cols")
```

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

MultiIndexes

MultiIndex objects

These are a hierarchical analog of standard Index objects, 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    0.477958  0.110635
    y    0.176785  0.524042
B   x    0.686342  0.254666
    y    0.366431  0.335209
C   x    0.526298  0.868879
    y    0.281066  0.523480
```

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   0.468450  0.956413  0.703818  0.262485
1   0.628330  0.063963  0.837249  0.867690
2   0.280684  0.258858  0.810500  0.730225
3   0.590576  0.607856  0.573620  0.788613
```

```
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   0.503954  0.687235  0.571731  0.062033
   p   0.890484  0.855219  0.250289  0.254360
n  l   0.131349  0.445940  0.408909  0.366915
   p   0.589651  0.551470  0.995567  0.952875
```

MultiIndex indexing

```
data
```

```
c1              A                   B
c2              x          y        x          y
r1 r2
m  l     0.558355  0.802381  0.501756  0.213913
   p     0.783600  0.142549  0.129497  0.972312
n  l     0.976518  0.737849  0.054503  0.756322
   p     0.429930  0.924574  0.291695  0.494732
```

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

```
c2              x          y
r1 r2
m  l     0.558355  0.802381
   p     0.783600  0.142549
n  l     0.976518  0.737849
   p     0.429930  0.924574
```

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

```
Error: KeyError: 'x'
```

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

```
Error: KeyError: 'm'
```

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

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

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

```
r1  r2
m   l     0.558355
    p     0.783600
n   l     0.976518
    p     0.429930
Name: (A, x), dtype: float64
```

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

```
r1  r2
m   l     0.558355
    p     0.783600
n   l     0.976518
    p     0.429930
Name: x, dtype: float64
```

MultiIndex indexing via `iloc`

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

```
c1  c2
A   x     0.558355
    y     0.802381
B   x     0.501756
    y     0.213913
Name: (m, l), dtype: float64
```

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

```
r1  r2
m   l     0.558355
    p     0.783600
n   l     0.976518
    p     0.429930
Name: (A, x), dtype: float64
```

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

```
0.8023809565660679
```

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

```
0.8023809565660679
```

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

```
c1            A                   B
c2            x         y         x         y
r1 r2
m  l   0.558355  0.802381  0.501756  0.213913
   p   0.783600  0.142549  0.129497  0.972312
```

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

```
c1  c2
A   x     0.558355
    y     0.802381
Name: (m, l), dtype: float64
```

Note that tuples and lists are not treated the same by pandas when it comes to indexing

MultiIndex indexing via `loc`

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

```
c1          A                   B
c2          x         y         x         y
r2
l    0.558355  0.802381  0.501756  0.213913
p    0.783600  0.142549  0.129497  0.972312
```

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

```
Error: KeyError: 'l'
```

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

```
c2             x         y
r1 r2
m  l    0.558355  0.802381
   p    0.783600  0.142549
n  l    0.976518  0.737849
   p    0.429930  0.924574
```

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

```
c1  c2
A   x      0.558355
    y      0.802381
B   x      0.501756
    y      0.213913
Name: (m, l), dtype: float64
```

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

```
r1  r2
m   l      0.802381
    p      0.142549
n   l      0.737849
    p      0.924574
Name: (A, y), dtype: float64
```

Fancier indexing with `loc`

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

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

```
c1            A                   B
c2            x         y         x         y
r1 r2
m  l   0.558355  0.802381  0.501756  0.213913
   p   0.783600  0.142549  0.129497  0.972312
n  l   0.976518  0.737849  0.054503  0.756322
   p   0.429930  0.924574  0.291695  0.494732
```

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

```
c1            A                   B
c2            x         y         x         y
r1 r2
m  l   0.558355  0.802381  0.501756  0.213913
   p   0.783600  0.142549  0.129497  0.972312
n  l   0.976518  0.737849  0.054503  0.756322
```

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

```
c1            A                   B
c2            x         y         x         y
r1 r2
m  p   0.783600  0.142549  0.129497  0.972312
n  l   0.976518  0.737849  0.054503  0.756322
   p   0.429930  0.924574  0.291695  0.494732
```

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

```
c1            A                   B
c2            x         y         x         y
r1 r2
m  p   0.783600  0.142549  0.129497  0.972312
n  l   0.976518  0.737849  0.054503  0.756322
```

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()`,

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

```
c1         A                   B
c2         x         y         x         y
r1
m    0.78360  0.142549  0.129497  0.972312
n    0.42993  0.924574  0.291695  0.494732
```

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

```
c1          A                   B
c2          x         y         x         y
r2
l    0.558355  0.802381  0.501756  0.213913
p    0.783600  0.142549  0.129497  0.972312
```

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

```
c1              A         B
r1 r2
m  l     0.802381  0.213913
   p     0.142549  0.972312
n  l     0.737849  0.756322
   p     0.924574  0.494732
```

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

```
c2              x         y
r1 r2
m  l     0.501756  0.213913
   p     0.129497  0.972312
n  l     0.054503  0.756322
   p     0.291695  0.494732
```

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=1)
```

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

Reshaping data

Long to wide (pivot)

```
df
```

```
   country  year   type count
0        A  1999  cases  0.7K
1        A  1999    pop   19M
2        A  2000  cases    2K
3        A  2000    pop   20M
4        B  1999  cases   37K
5        B  1999    pop  172M
6        B  2000  cases   80K
7        B  2000    pop  174M
8        C  1999  cases  212K
9        C  1999    pop    1T
10       C  2000  cases  213K
11       C  2000    pop    1T
```

```
df_wide = df.pivot(
  index=["country","year"],
  columns="type",
  values="count"
)
df_wide
```

```
type          cases   pop
country year
A       1999   0.7K   19M
        2000     2K   20M
B       1999    37K  172M
        2000    80K  174M
C       1999   212K    1T
        2000   213K    1T
```

pivot indexes

```python
df_wide.index
```

```
MultiIndex([('A', 1999),
            ('A', 2000),
            ('B', 1999),
            ('B', 2000),
            ('C', 1999),
            ('C', 2000)],
           names=['country', 'year'])
```

```python
df_wide.columns
```

```
Index(['cases', 'pop'], dtype='object',
name='type')
```

```python
df_wide.reset_index(
).rename_axis(
  columns=None
)
```

```
  country  year cases   pop
0       A  1999  0.7K   19M
1       A  2000    2K   20M
2       B  1999   37K  172M
3       B  2000   80K  174M
4       C  1999  212K    1T
5       C  2000  213K    1T
```

Wide to long (melt)

```
df
```

```
  country  1999  2000
0       A  0.7K    2K
1       B   37K   80K
2       C  212K  213K
```

```
df_long = df.melt(
  id_vars="country",
  var_name="year"
)
df_long
```

```
  country  year value
0       A  1999  0.7K
1       B  1999   37K
2       C  1999  212K
3       A  2000    2K
4       B  2000   80K
5       C  2000  213K
```

Separate Example - splits and explosions

```
df
```

```
  country  year       rate
0       A  1999  0.7K/19M
1       A  2000    2K/20M
2       B  1999  37K/172M
3       B  2000  80K/174M
4       C  1999   212K/1T
5       C  2000   213K/1T
```

```
df.assign(
  rate = lambda d: d.rate.str.split("/")
)
```

```
  country  year          rate
0       A  1999  [0.7K, 19M]
1       A  2000    [2K, 20M]
2       B  1999  [37K, 172M]
3       B  2000  [80K, 174M]
4       C  1999   [212K, 1T]
5       C  2000   [213K, 1T]
```

```
( df.assign(
    rate = lambda d: d.rate.str.split("/")
  )
  .explode("rate")
  .assign(
    type = lambda d: ["cases", "pop"] * int(d.sh
  )
)
```

```
  country  year  rate   type
0       A  1999  0.7K  cases
0       A  1999   19M    pop
1       A  2000    2K  cases
1       A  2000   20M    pop
2       B  1999   37K  cases
2       B  1999  172M    pop
3       B  2000   80K  cases
3       B  2000  174M    pop
4       C  1999  212K  cases
4       C  1999    1T    pop
5       C  2000  213K  cases
5       C  2000    1T    pop
```

Putting it together

```
( df
  .assign(
    rate = lambda d: d.rate.str.split("/")
  )
  .explode("rate")
  .assign(
    type = lambda d: ["cases", "pop"] *
                     int(d.shape[0]/2)
  )
)
```

```
  country  year  rate   type
0       A  1999  0.7K  cases
0       A  1999   19M    pop
1       A  2000    2K  cases
1       A  2000   20M    pop
2       B  1999   37K  cases
2       B  1999  172M    pop
3       B  2000   80K  cases
3       B  2000  174M    pop
4       C  1999  212K  cases
4       C  1999    1T    pop
5       C  2000  213K  cases
5       C  2000    1T    pop
```

```
( df.assign(
    rate = lambda d: d.rate.str.split("/")
  )
  .explode("rate")
  .assign(
    type = lambda d: ["cases", "pop"] *
                     int(d.shape[0]/2)
  )
  .pivot(
    index=["country","year"],
    columns="type",
    values="rate"
  )
  .reset_index()
)
```

```
type country  year cases   pop
0          A  1999  0.7K   19M
1          A  2000    2K   20M
2          B  1999   37K  172M
3          B  2000   80K  174M
4          C  1999  212K    1T
5          C  2000  213K    1T
```

Separate Example - A better way

```
df
```

```
  country  year       rate
0       A  1999   0.7K/19M
1       A  2000     2K/20M
2       B  1999   37K/172M
3       B  2000   80K/174M
4       C  1999    212K/1T
5       C  2000    213K/1T
```

```
df.assign(
  counts = lambda d: d.rate.str.split("/").str[0
  pop    = lambda d: d.rate.str.split("/").str[1
)
```

```
  country  year       rate counts   pop
0       A  1999   0.7K/19M   0.7K   19M
1       A  2000     2K/20M     2K   20M
2       B  1999   37K/172M    37K  172M
3       B  2000   80K/174M    80K  174M
4       C  1999    212K/1T   212K    1T
5       C  2000    213K/1T   213K    1T
```

If you dont want to repeat the split,

```python
df.assign(
  rate = lambda d: d.rate.str.split("/"),
  counts = lambda d: d.rate.str[0],
  pop    = lambda d: d.rate.str[1]
).drop("rate", axis=1)
```

```
  country  year counts   pop
0       A  1999   0.7K   19M
1       A  2000     2K   20M
2       B  1999    37K  172M
3       B  2000    80K  174M
4       C  1999   212K    1T
5       C  2000   213K    1T
```

Exercise 1

Create a DataFrame from the data available at https://sta663-sp23.github.io/slides/data/us_rent.csv using `pd.read_csv()`.

These data come from the 2017 American Community Survey and reflect the following values: * `name` - name of state * `variable` - Variable name: income = median yearly income, rent = median monthly rent * `estimate` - Estimated value * `moe` - 90% margin of error

Using these data find the state(s) with the lowest income to rent ratio.

Split-Apply-Combine

groupby

Groups can be created within a DataFrame via `groupby()` - these groups are then used by the standard summary methods (e.g. `sum()`, `mean()`, `std()`, etc.).

```python
cereal = pd.read_csv("https://sta663-sp23.github.io/slides/data/cereal.csv")
cereal
```

```
                         name             mfr  ... sugars     rating
0                   100% Bran         Nabisco  ...      6  68.402973
1           100% Natural Bran     Quaker Oats  ...      8  33.983679
2                    All-Bran       Kellogg's  ...      5  59.425505
3   All-Bran with Extra Fiber       Kellogg's  ...      0  93.704912
4              Almond Delight  Ralston Purina  ...      8  34.384843
..                        ...             ...  ...    ...        ...
72                    Triples   General Mills  ...      3  39.106174
73                       Trix   General Mills  ...     12  27.753301
74                 Wheat Chex  Ralston Purina  ...      3  49.787445
75                   Wheaties   General Mills  ...      3  51.592193
76        Wheaties Honey Gold   General Mills  ...      8  36.187559

[77 rows x 6 columns]
```

```python
cereal.groupby("type")
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x290553850>
```

GroupBy attributes and methods

```python
cereal.groupby("type").groups
```

```
{'Cold': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 21, 22, 23, 24, 25, 26,
27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53,
54, 55, 56, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76], 'Hot': [20, 43,
57]}
```

```python
cereal.groupby("type").mean(numeric_only=True)
```

```
        calories    sugars     rating
type                                 
Cold  107.162162  7.175676  42.095218
Hot   100.000000  1.333333  56.737708
```

```python
cereal.groupby("mfr").groups
```

```
{'General Mills': [5, 7, 11, 12, 13,
14, 18, 22, 31, 36, 40, 42, 47, 51, 59,
69, 70, 71, 72, 73, 75, 76],
'Kellogg's': [2, 3, 6, 16, 17, 19, 21,
24, 25, 26, 28, 38, 39, 46, 48, 49, 50,
53, 58, 60, 62, 66, 67], 'Maltex':
[43], 'Nabisco': [0, 20, 63, 64, 65,
68], 'Post': [9, 27, 29, 30, 32, 33,
34, 37, 52], 'Quaker Oats': [1, 10, 35,
41, 54, 55, 56, 57], 'Ralston Purina':
[4, 8, 15, 23, 44, 45, 61, 74]}
```

```python
cereal.groupby("mfr").size()
```

```
mfr
General Mills     22
Kellogg's         23
Maltex             1
Nabisco            6
Post               9
Quaker Oats        8
Ralston Purina     8
dtype: int64
```

Selecting groups

Groups can be accessed via `get_group()` or the DataFrameGroupBy can be iterated over,

```python
cereal.groupby("type").get_group("Hot")
```

```
                      name           mfr type  calories  sugars     rating
20  Cream of Wheat (Quick)       Nabisco  Hot       100       0  64.533816
43                   Maypo        Maltex  Hot       100       3  54.850917
57          Quaker Oatmeal  Quaker Oats  Hot       100       1  50.828392
```

```python
cereal.groupby("mfr").get_group("Post")
```

```
                                      name   mfr  ... sugars     rating
9                              Bran Flakes  Post  ...      5  53.313813
27  Fruit & Fibre Dates; Walnuts; and Oats  Post  ...     10  40.917047
29                          Fruity Pebbles  Post  ...     12  28.025765
30                            Golden Crisp  Post  ...     15  35.252444
32                       Grape Nuts Flakes  Post  ...      5  52.076897
33                              Grape-Nuts  Post  ...      3  53.371007
34                      Great Grains Pecan  Post  ...      4  45.811716
37                              Honey-comb  Post  ...     11  28.742414
52                   Post Nat. Raisin Bran  Post  ...     14  37.840594

[9 rows x 6 columns]
```

Iterating groups

```
for name, group in cereal.groupby("type"):
  print(f"# {name}\n{group}\n\n")
```

```
# Cold
                         name             mfr  ... sugars     rating
0                   100% Bran         Nabisco  ...      6  68.402973
1           100% Natural Bran     Quaker Oats  ...      8  33.983679
2                    All-Bran       Kellogg's  ...      5  59.425505
3   All-Bran with Extra Fiber       Kellogg's  ...      0  93.704912
4              Almond Delight  Ralston Purina  ...      8  34.384843
..                        ...             ...  ...    ...        ...
72                    Triples   General Mills  ...      3  39.106174
73                       Trix   General Mills  ...     12  27.753301
74                Wheat Chex  Ralston Purina  ...      3  49.787445
75                   Wheaties   General Mills  ...      3  51.592193
76         Wheaties Honey Gold   General Mills  ...      8  36.187559

[74 rows x 6 columns]
```

Aggregation

The `aggregate()` function or `agg()` method can be used to compute summary statistics for each group,

```
cereal.groupby("mfr").agg("mean")
```

```
                 calories    sugars     rating
mfr
General Mills  111.363636  7.954545  34.485852
Kellogg's      108.695652  7.565217  44.038462
Maltex         100.000000  3.000000  54.850917
Nabisco         86.666667  1.833333  67.968567
Post           108.888889  8.777778  41.705744
Quaker Oats     95.000000  5.500000  42.915990
Ralston Purina 115.000000  6.125000  41.542997
```

```
<string>:1: FutureWarning: The default value of numeric_only in DataFrameGroupBy.mean is deprecated. In a future version, numeric_only will default to False. Either specify numeric_only or select only columns which should be valid for the function.
```

Think `summarize()` from dplyr.

Aggregation with multiple functions

```
cereal.groupby("mfr").agg([np.mean, np.std])
```

```
                 calories             sugars              rating           
                     mean        std      mean       std       mean       std
mfr                                                                          
General Mills  111.363636  10.371873  7.954545  3.872704  34.485852  8.946704
Kellogg's      108.695652  22.218818  7.565217  4.500768  44.038462 14.457434
Maltex         100.000000        NaN  3.000000       NaN  54.850917       NaN
Nabisco         86.666667  10.327956  1.833333  2.857738  67.968567  5.509326
Post           108.888889  10.540926  8.777778  4.576510  41.705744 10.047647
Quaker Oats     95.000000  29.277002  5.500000  4.780914  42.915990 16.797673
Ralston Purina 115.000000  22.677868  6.125000  3.563205  41.542997  6.080841

<string>:1: FutureWarning: ['name', 'type'] did not aggregate successfully. If any error is raised this
will raise in a future version of pandas. Drop these columns/ops to avoid this warning.
```

Aggregation by column

```
cereal.groupby("mfr").agg({
  "calories": ['min', 'max'],
  "sugars":   ['mean', 'median'],
  "rating":   ['sum', 'count']
})
```

```
               calories          sugars               rating      
                    min  max       mean median          sum count
mfr                                                              
General Mills       100  140  7.954545    8.5   758.688737    22
Kellogg's            50  160  7.565217    7.0  1012.884634    23
Maltex              100  100  3.000000    3.0    54.850917     1
Nabisco              70  100  1.833333    0.0   407.811403     6
Post                 90  120  8.777778   10.0   375.351697     9
Quaker Oats          50  120  5.500000    6.0   343.327919     8
Ralston Purina       90  150  6.125000    5.5   332.343977     8
```

Named aggregation

It is also possible to use special syntax to aggregate specific columns into a named output column,

```
cereal.groupby("mfr", as_index=False).agg(
  min_cal = ("calories", "min"),
  max_cal = ("calories", "max"),
  med_sugar = ("sugars", "median"),
  avg_rating = ("rating", "mean")
)
```

```
               mfr  min_cal  max_cal  med_sugar  avg_rating
0    General Mills      100      140        8.5   34.485852
1        Kellogg's       50      160        7.0   44.038462
2           Maltex      100      100        3.0   54.850917
3          Nabisco       70      100        0.0   67.968567
4             Post       90      120       10.0   41.705744
5      Quaker Oats       50      120        6.0   42.915990
6   Ralston Purina       90      150        5.5   41.542997
```

Tuples can also be passed using `pd.NamedAgg()` but this offers no additional functionality.

Transformation

The `transform()` method returns a DataFrame with the aggregated result matching the size (or length 1) of the input group(s),

```
cereal.groupby("mfr").transform(np.mean)
```

```
      calories    sugars     rating
0    86.666667  1.833333  67.968567
1    95.000000  5.500000  42.915990
2   108.695652  7.565217  44.038462
3   108.695652  7.565217  44.038462
4   115.000000  6.125000  41.542997
..         ...       ...        ...
72  111.363636  7.954545  34.485852
73  111.363636  7.954545  34.485852
74  115.000000  6.125000  41.542997
75  111.363636  7.954545  34.485852
76  111.363636  7.954545  34.485852

[77 rows x 3 columns]

<string>:1: FutureWarning: The default value of
numeric only in DataFrameGroupBy.mean is
```

```
cereal.groupby("type").transform("mean")
```

```
      calories    sugars     rating
0   107.162162  7.175676  42.095218
1   107.162162  7.175676  42.095218
2   107.162162  7.175676  42.095218
3   107.162162  7.175676  42.095218
4   107.162162  7.175676  42.095218
..         ...       ...        ...
72  107.162162  7.175676  42.095218
73  107.162162  7.175676  42.095218
74  107.162162  7.175676  42.095218
75  107.162162  7.175676  42.095218
76  107.162162  7.175676  42.095218

[77 rows x 3 columns]

<string>:1: FutureWarning: The default value of
numeric only in DataFrameGroupBy.mean is
```


Note that we have lost the non-numeric columns

Practical transformation

`transform()` will generally be most useful via a user defined function, the lambda argument is each column of each group.

```
( cereal
  .groupby("mfr")
  .transform(
    lambda x: (x - np.mean(x))/np.std(x)
  )
)
```

```
    calories    sugars    rating
0  -1.767767  1.597191  0.086375
1   0.912871  0.559017 -0.568474
2  -1.780712 -0.582760  1.088220
3  -2.701081 -1.718649  3.512566
4  -0.235702  0.562544 -1.258442
..       ...       ...       ...
72 -0.134568 -1.309457  0.528580
73 -0.134568  1.069190 -0.770226
74 -0.707107 -0.937573  1.449419
75 -1.121403 -1.309457  1.957022
76 -0.134568  0.012013  0.194681

[77 rows x 3 columns]

<string>:3: FutureWarning: Dropping invalid columns in DataFrameGroupBy.transform is deprecated. In a
```


Above we are standardizing each numerical column of each manufacturer

Filtering groups

`filter()` also respects groups and allows for the inclusion / exclusion of groups based on user specified criteria,

```python
( cereal
  .groupby("mfr")
  .filter(lambda x: len(x) > 8)
)
```

```
                           name            mfr  ...  sugars     rating
2                      All-Bran      Kellogg's  ...       5  59.425505
3     All-Bran with Extra Fiber      Kellogg's  ...       0  93.704912
5       Apple Cinnamon Cheerios  General Mills  ...      10  29.509541
6                   Apple Jacks      Kellogg's  ...      14  33.174094
7                       Basic 4  General Mills  ...       8  37.038562
9                   Bran Flakes           Post  ...       5  53.313813
11                     Cheerios  General Mills  ...       1  50.764999
12        Cinnamon Toast Crunch  General Mills  ...       9  19.823573
13                     Clusters  General Mills  ...       7  40.400208
14                  Cocoa Puffs  General Mills  ...      13  22.736446
16                  Corn Flakes      Kellogg's  ...       2  45.863324
17                    Corn Pops      Kellogg's  ...      12  35.782791
18               Count Chocula  General Mills  ...      13  22.396513
19           Cracklin' Oat Bran      Kellogg's  ...       7  40.448772
21                      Crispix      Kellogg's  ...       3  46.895644
22       Crispy Wheat & Raisins  General Mills  ...      10  36.176196
```

```python
( cereal
  .groupby("mfr")
  .size()
)
```

```
mfr
General Mills     22
Kellogg's         23
Maltex             1
Nabisco            6
Post               9
Quaker Oats        8
Ralston Purina     8
dtype: int64
```

```python
( cereal
  .groupby("mfr")
  .filter(lambda x: len(x) > 8)
  .groupby("mfr")
  .size()
)
```

```
mfr
General Mills    22
Kellogg's        23
Post              9
dtype: int64
```