

Lec 09 - more pandas

Statistical Computing and Computation

Sta 663 | Spring 2022

Dr. Colin Rundel

Index objects

Columns and Indexes

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

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

```
##           A         B         C  
## 0 -0.099711 -0.381847 -2.392402  
## 1  0.544186  1.175953  1.237503  
## 2 -0.605081  1.869954  0.618847  
## 3 -0.988612  0.656876 -0.179668  
## 4 -0.400453  0.555089  1.098572
```

```
df.columns
```

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

```
df.index
```

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

```
##           A         B         C  
## x  0.612553  1.120081 -0.891120  
## y  0.990790 -1.405966  0.544711  
## z -1.958477  0.750223  0.655311
```

```
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).

```
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 Indexs 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.intersection(b)
```

```
## Index(['c'], 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], dt
```

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

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

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

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

```
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      0.300764  0.557300  0.067900  
## y     -0.445863 -2.123754 -0.430039  
## z     -0.963107  0.451464 -1.386723
```

```
df.columns
```

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

```
df.index
```

```
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      0.300764  0.557300  0.067900  
## y     -0.445863 -2.123754 -0.430039  
## z     -0.963107  0.451464 -1.386723
```

```
df.columns.name = "o"  
df.index.rename("p", inplace=True)  
df
```

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"])  
  
## Index(['A', 'B', nan, 'D'], 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"]).fillna("Z")  
  
## Index(['A', 'B', 'Z', 'D'], dtype='object')
```

Changing a DataFrame's index

Existing columns can be used as an index via `set_index()` and removed via `reset_index()`,

```
data
```

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

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

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

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

```
##      a   b   c   d
## c
```

```
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
```

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

```
##      a   b   d
## 0  bar  one  1
## 1  bar  two  2
```

Creating a new index

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

```
data
```

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

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

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

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

```
##      a    b    c    d
## 5  NaN  NaN  NaN  NaN
## 4  NaN  NaN  NaN  NaN
```

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

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

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

```
##      a    b    c    d
## w  bar  one   z   1
```

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
```

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

```
##      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
```

MultiIndexes

MultIndex 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"]])
```

```
## MultiIndex([('A', 'x'),
##               ('A', 'y'),
##               ('B', 'x'),
##               ('B', 'y'),
##               ('C', 'x'),
##               ('C', 'y')])
```

```
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	0.756919	0.890441
##	y	0.337214	0.719306
## B	x	0.422858	0.619788
##	y	0.299592	0.991226
## C	x	0.810381	0.157312
##	y	0.098790	0.996070

Column MultiIndex

```
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    0.929432  0.824582  0.621965  0.072779
## 1    0.136231  0.818250  0.100441  0.326754
## 2    0.120870  0.580064  0.506232  0.764804
## 3    0.639309  0.478697  0.318579  0.790524
```

```
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    0.182841  0.847452  0.770413  0.103425
##     p    0.494563  0.004254  0.297815  0.282648
## n l    0.821943  0.999260  0.413331  0.046545
##     p    0.337158  0.984532  0.418133  0.358786
```

MultIndex indexing

```
data
```

```
## c1          A          B
## c2          x          y      x      y
## r1 r2
## m  l    0.019149  0.519056  0.924092  0.996320
##   p    0.219535  0.537471  0.962619  0.968074
## n  l    0.020447  0.817611  0.493241  0.632190
##   p    0.432398  0.854118  0.774252  0.838321
```

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

```
## c2          x          y
## r1 r2
## m  l    0.019149  0.519056
##   p    0.219535  0.537471
## n  l    0.020447  0.817611
##   p    0.432398  0.854118
```

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

```
## KeyError: 'x'
```

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

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

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

```
## r1  r2
## m  l    0.019149
##   p    0.219535
## n  l    0.020447
##   p    0.432398
## Name: (A, x), dtype: float64
```

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

```
## r1  r2
## m  l    0.019149
##   p    0.219535
## n  l    0.020447
##   p    0.432398
## Name: x, dtype: float64
```

MultIndex indexing via iloc

data

```
## c1          A          B
## c2          x          y      x      y
## r1 r2
## m  l  0.019149  0.519056  0.924092  0.996320
##   p  0.219535  0.537471  0.962619  0.968074
## n  l  0.020447  0.817611  0.493241  0.632190
##   p  0.432398  0.854118  0.774252  0.838321
```

data.iloc[0]

```
## c1  c2
## A    x    0.019149
##       y    0.519056
## B    x    0.924092
##       y    0.996320
## Name: (m, l), dtype: float64
```

data.iloc[(0,1)]

```
## 0.519055710819791
```

data.iloc[[0,1]]

```
## c1          A          B
## c2          x          y      x      y
## r1 r2
## m  l  0.019149  0.519056  0.924092  0.996320
##   p  0.219535  0.537471  0.962619  0.968074
```

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

data.iloc[:,0]

```
## r1  r2
## m  l  0.019149
##   p  0.219535
## n  l  0.020447
##   p  0.432398
## Name: (A, x), dtype: float64
```

data.iloc[0,1]

```
## 0.519055710819791
```

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

```
## c1  c2
## A    x    0.019149
##       y    0.519056
## Name: (m, l), dtype: float64
```

MultIndex indexing via loc

```
data
```

```
## c1          A          B
## c2          x          y      x      y
## r1 r2
## m  l  0.019149  0.519056  0.924092  0.996320
##   p  0.219535  0.537471  0.962619  0.968074
## n  l  0.020447  0.817611  0.493241  0.632190
##   p  0.432398  0.854118  0.774252  0.838321
```

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

```
## c1          A          B
## c2          x          y      x      y
## r2
## l  0.019149  0.519056  0.924092  0.996320
## p  0.219535  0.537471  0.962619  0.968074
```

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

```
## KeyError: 'l'
```

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

```
## c2          x          y
## r1 r2
## m  l  0.019149  0.519056
##   p  0.219535  0.537471
## n  l  0.020447  0.817611
##   p  0.432398  0.854118
```

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

```
## c1  c2
## A    x  0.019149
##       y  0.519056
## B    x  0.924092
##       y  0.996320
## Name: (m, l), dtype: float64
```

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

```
## r1  r2
## m   l  0.519056
##   p  0.537471
## n   l  0.817611
##   p  0.854118
## Name: (A, y), dtype: float64
```

Fancier indexing with loc

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

```
data
```

```
## c1          A          B
## c2          x          y      x      y
## r1 r2
## m l  0.019149  0.519056  0.924092  0.996320
## p   0.219535  0.537471  0.962619  0.968074
## n l  0.020447  0.817611  0.493241  0.632190
## p   0.432398  0.854118  0.774252  0.838321
```

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

```
## c1          A          B
## c2          x          y      x      y
## r1 r2
## m l  0.019149  0.519056  0.924092  0.996320
## p   0.219535  0.537471  0.962619  0.968074
## n l  0.020447  0.817611  0.493241  0.632190
## p   0.432398  0.854118  0.774252  0.838321
```

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

```
## c1          A          B
## c2          x          y      x      y
## r1 r2
## m l  0.019149  0.519056  0.924092  0.996320
## p   0.219535  0.537471  0.962619  0.968074
```

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

```
## c1          A          B
## c2          x          y      x      y
## r1 r2
## m p  0.219535  0.537471  0.962619  0.968074
## n l  0.020447  0.817611  0.493241  0.632190
## p   0.432398  0.854118  0.774252  0.838321
```

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

```
## c1          A          B
## c2          x          y      x      y
## r1 r2
## m p  0.219535  0.537471  0.962619  0.968074
## n l  0.020447  0.817611  0.493241  0.632190
```

Selecting nested levels

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

```
data
```

```
## c1          A          B
## c2          x          y      x      y
## r1 r2
## m  l  0.019149  0.519056  0.924092  0.996320
##   p  0.219535  0.537471  0.962619  0.968074
## n  l  0.020447  0.817611  0.493241  0.632190
##   p  0.432398  0.854118  0.774252  0.838321
```

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

```
## c1          A          B
## c2          x          y      x      y
## r1
## m  l  0.219535  0.537471  0.962619  0.968074
## n  l  0.432398  0.854118  0.774252  0.838321
```

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

```
## c1          A          B
## c2          x          y      x      y
## r2
## l  l  0.019149  0.519056  0.924092  0.996320
##   p  0.219535  0.537471  0.962619  0.968074
```

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

```
## c1          A          B
## r1 r2
## m  l  0.519056  0.996320
##   p  0.537471  0.968074
## n  l  0.817611  0.632190
##   p  0.854118  0.838321
```

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

```
## c2          x          y
## r1 r2
## m  l  0.924092  0.996320
##   p  0.962619  0.968074
```

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  
## 3  foo  two  w   4
```

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

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

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

```
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  
## 3  foo  two  w   4
```

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

```
##          b   c   d
```

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.index

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

```
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
```

```
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
```

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("/")
  )
  .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
```

```
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.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
```

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 don't want to repeat the split,

```
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
```

Exercise 1

Create a DataFrame from the data available at https://sta663-sp22.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.).

```
cereal = pd.read_csv("https://sta663-sp22.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        ... ... ... ...
## 73        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]
```

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

```
## <pandas.core.groupby.generic.DataFrameGroupBy object at 0x143e2a460>
```

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

```
## {'Cold': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 1
```

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

```
## {'General Mills': [5, 7, 11, 12, 13, 14, 18, 22, 31, 36, 40, 42, 47,
```

Selecting and iterating groups

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

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

```
##          name    mfr type calories sugar
## 20  Cream of Wheat (Quick) Nabisco Hot     100
## 43      Maypo   Maltex Hot     100
## 57 Quaker Oatmeal Quaker Oats Hot     100
```

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

```
##          name    mfr ... sugars
## 9      Bran Flakes Post ...     5
## 27 Fruit & Fibre Dates; Walnuts; and Oats Post ...    10
## 29      Fruity Pebbles Post ...    12
## 30      Golden Crisp Post ...    15
## 32      Grape Nuts Flakes Post ...     5
## 33      Grape-Nuts Post ...     3
## 34      Great Grains Pecan Post ...     4
## 37      Honey-comb Post ...    11
## 52      Post Nat. Raisin Bran Post ...    14
##
## [9 rows x 6 columns]
```

```
for name, group in cereal.groupby("type"):
    print(name)
    print(group)
    print("")
```

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 59.425050
## 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          Wheatus General Mills ...     3 51.592193
## 76      Wheaties Honey Gold General Mills ...     8 36.187559
##
## [74 rows x 6 columns]
```

##

Hot

```
##          name    mfr type calories sugars r
## 20  Cream of Wheat (Quick) Nabisco Hot     100     0 64.5
## 43      Maypo   Maltex Hot     100     3 54.8
## 57 Quaker Oatmeal Quaker Oats Hot     100    100 50.8
```

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
```

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

```
##             calories          sugars
##                   mean            std
##                   mean            std
## mfr
## General Mills 111.363636 10.371873 7.954545 3.872704 34
## Kellogg's      108.695652 22.218818 7.565217 4.500768 44
## Maltex         100.000000       NaN 3.000000       NaN 54
## Nabisco        86.666667 10.327956 1.833333 2.857738 67
## Post           108.888889 10.540926 8.777778 4.576510 41
## Quaker Oats    95.000000 29.277002 5.500000 4.780914 42
## Ralston Purina 115.000000 21.77868 6.125000 3.563205 41
##
```

Think summarized from dplyr

```
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: Dropping invalid col
```

Note that we have lost the non-numeric columns

```
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: Dropping invalid 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
##
```

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,

```
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
```

```
cereal.groupby("mfr").filter(lambda x: len(x) > 10)
```

```
##                               name      mfr ... sugars
## 2                      All-Bran  Kellogg's ...     5  5
## 3  All-Bran with Extra Fiber  Kellogg's ...     0  9
## 5      Apple Cinnamon Cheerios General Mills ...  10  2
## 6          Apple Jacks       Kellogg's ...  14  3
## 7             Basic 4       General Mills ...     8  3
## 11            Cheerios       General Mills ...     1  5
## 12  Cinnamon Toast Crunch  General Mills ...     9  1
## 13            Clusters       General Mills ...     7  4
## 14            Cocoa Puffs  General Mills ...  13  2
## 16            Corn Flakes    Kellogg's ...     2  4
```

```
( cereal
  .groupby("mfr")
  .filter(lambda x: len(x) > 10)
  .groupby("mfr")
  .size()
)
```

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