Intro. Comp. for Data Science (FMI08)

Dr. Nono Saha

May 26, 2023

Max Planck Institute for Mathematics in the Sciences
University of Leipzig/ScaDS.AI

Spring 2023

Course plan

1. Index objects
2. MultiIndexes
3. Reshaping data
4. Split-Apply-Combine

More pandas - Index objects

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 -0.091 -0.37 -2.39
## 1  0.54  1.17  1.23
## 2 -0.6  1.8 0.67

df.columns
## Index(['A', 'B', 'C'], dtype='object')

df.index
## RangeIndex(start=0, stop=5, step=1)

```

```
df = pd.DataFrame(np.random.randn(3, 3),
index=['x','y','z'], columns=['A', 'B', 'C'])
##   A  B  C
## x  0.61  1.12 -0.8
## y  0.90 -1.46  0.54
## z -1.95  0.75  0.65

df.columns
## Index(['A', 'B', 'C'], dtype='object')

df.index
## Index(['x', 'y', 'z'], dtype='object')
```

Index objects: creating an index object

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')

```

Index objects: 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], dtype='float64')

```

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

Index objects: 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')

```

pandas - changing a DataFrame's index

Existing columns can 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')

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

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

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

```

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

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

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

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

More pandas - multiIndexes

MultiIndex: 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"])

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

idx = pd.MultiIndex.from_tuples(tuples, names=["1st","2nd"])
pd.DataFrame(np.random.rand(6,2), index = idx, columns=["m","n"])

#Column MultiIndex
cidx = pd.MultiIndex.from_product([["A","B"],["x","y"]], names=["c1","c2"])
pd.DataFrame(np.random.rand(4,4), columns = cidx)

```

MultiIndex: indexing multiIndex

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

```

Examples

```
data["A"]

data["x"]

data["m"]

```

```
data["m","A"]
## KeyError: ('m', 'A')

data["A","x"]

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

MultiIndex: 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]
#Try and see the output

data.iloc[(0,1)]
## 0.519055710819791

data.iloc[[0,1]]

```

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

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

```

MultiIndex: 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"]
#Try and see the output

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

```

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

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

```

MultiIndex: 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")
#Try and see the output

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

```

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

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

```

MultiIndex: 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'])
#Try and see the output

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

```

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

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

```

pandas - Reshaping data

Reshaping data: long to wide (pivot) and wide to long (melt)

```
df = pd.read_csv("../data/reshaping.csv", index_col=0)

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

```

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

df_wide.columns
## Index(['cases', 'pop'], dtype='object', name='type')

```

```
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

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

Reshaping data: separate example - splits and explosions

```
df = pd.read_csv("../data/rate.csv", index_col=0)
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
```

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

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

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()
```

Reshaping data: separate example - a better way

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

If you dont 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)

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

Exercice

Create a DataFrame from the data available at `../data/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.