Exploring & Modifying Data

1405 Instructor: Ruiqing (Sam) Cao

Required Python Libraries for Today

Core

- NumPy, Pandas

```
import numpy as np
import pandas as pd
```

Visualization

- Matplotlib, Seaborn

```
from matplotlib import pyplot as plt
import seaborn as sns
```

Statistical learning

- scikit-learn, SciPy, statsmodels

```
import sklearn
import scipy
import statsmodel as sm
```

NumPy & Pandas Printing Format

It's better to print only *the first few decimal digits* of large real numbers. Set the print options to keep 3 decimal digits and supress scientific notation (for NumPy arrays and Pandas DataFrames):

- **NumPy**

```
np.set_printoptions(precision=3,suppress=True)
```

- **Pandas**

```
pd.options.display.float_format = '{:.3f}'.format
```

Pandas DataFrame: Review

Pandas DataFrame: a 2D labeled array storing tabular data, where rows represent observations and columns represent variables

Column names *(`df.columns`)*

df=

	name	age	isFemale
0	Char	10	False
1	Lin	-1	True

Row index *(`df.index`)*

Q: Must column names and row indexes be unique?

Q: Is each column or row a Pandas Series?

Explore a DataFrame

Examine a Pandas DataFrame

Read the CSV file **census_data.csv** into a DataFrame **df**:

	state_name	state_fips	year	population	median_age	median_income	below_poverty_line
0	Alabama	01	2010	4712651	37.5	22141	0.171106
1	Alaska	02	2010	691189	33.8	31238	0.095206
2	Arizona	04	2010	6246816	35.5	26913	0.152711
3	Arkansas	05	2010	2872684	37.2	21286	0.180122
4	California	06	2010	36637290	34.9	27733	0.137134
...	...	...	...	...	...	...	...
47	Delaware	10	2020	967679	41.0	35089	0.114358
48	Puerto Rico	72	2020	3255642	42.4	13898	0.434075
49	Kentucky	21	2020	4461952	39.0	28270	0.166069
50	South Dakota	46	2020	879336	37.2	32789	0.128088
51	Tennessee	47	2020	6772268	38.8	29605	0.146168

104 rows × 7 columns

View the first few rows of a DataFrame

Display the first **n** rows of the DataFrame **df**

```
df.head(n)
```

	state_name	state_fips	year	population	median_age	median_income	below_poverty_line
0	Alabama	01	2010	4712651	37.5	22141	0.171106
1	Alaska	02	2010	691189	33.8	31238	0.095206
2	Arizona	04	2010	6246816	35.5	26913	0.152711
3	Arkansas	05	2010	2872684	37.2	21286	0.180122
4	California	06	2010	36637290	34.9	27733	0.137134
5	Colorado	08	2010	4887061	35.8	30261	0.122386
6	Connecticut	09	2010	3545837	39.5	33293	0.091504

Examine a Pandas DataFrame

When exploring a new dataset, ask yourself:

- **What are the key quantitative variables** measured?
- What is the **data type** of each variable?
- **How many observations** are there?
- Which variables **uniquely identify** each observation?

Examine a Pandas DataFrame

When exploring a new dataset, ask yourself:

- **What are the key quantitative variables** measured?
  - population, median_age, median_income, below_poverty_line
- What is the **data type** of each variable?
  - int, float, int, float
- **How many observations** are there?

- Which variables **uniquely identify** each observation?

Shape of a DataFrame

Shape of the DataFrame `df` is a tuple `(columns, rows)`

```
df.shape
```
→ `(104, 7)`

Number of rows (observations) in `df`

```
df.shape[0]
```
→ `104`

Number of columns (variables) in `df`

```
df.shape[1]
```
→ `7`

Examine a Pandas DataFrame

When exploring a new dataset, ask yourself:

- **What are the key quantitative variables** measured?
  - population, median_age, median_income, below_poverty_line
- What is the **data type** of each variable?
  - int, float, int, float
- **How many observations** are there?
  - 104
- Which variables **uniquely identify** each observation?

Column Names of a DataFrame

Return all column names (variables) of the DataFrame **df**

```
df.columns
```

output

```
Index(['state_name', 'state_fips', 'year', 'population', 'median_age',
       'median_income', 'below_poverty_line'],
      dtype='object')
```

Convert the index array into a list of column names

```
df.columns.tolist()
```

output

```
['state_name', 'state_fips', 'year', 'population', 'median_age', 'median_income', 'below_poverty_line']
```

Row Indexes of a DataFrame

Return the entire index of the DataFrame

```
df.index
```

output

```
RangeIndex(start=0, stop=104, step=1)
```

Very often simplify the result by converting it to a list

```
df.index.tolist()
```

output

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88,
89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103]
```

Frequency Counts of Unique Values

- Get the frequency counts of *one variable* (Series) or *a group of variables* (DataFrame)

```
DataFrame.value_counts()
```

...Returns a Pandas Series *indexed by the unique values* of the variables in `DataFrame` and *containing their frequency counts*

Example:

```
df[['state_name','year']].value_counts()
```

output

The counts are sorted in descending order by default. Top value of 1 means (state_name, year) is a candidate key

```
state_name    year
Alabama       2010    1
              2020    1
Pennsylvania  2010    1
Oregon        2020    1
              2010    1
                      ..
Iowa          2020    1
              2010    1
Indiana       2020    1
              2010    1
Wyoming       2020    1
Name: count, Length: 104, dtype: int64
```

Examine a Pandas DataFrame

When exploring a new dataset, ask yourself:

- **What are the key quantitative variables** measured?
  - population, median_age, median_income, below_poverty_line
- What is the **data type** of each variable?
  - int, float, int, float
- **How many observations** are there?
  - 104
- Which variables **uniquely identify** each observation?
  - (state_name, year) or (state_fips, year)

Row Indexes can be *other than 0 to N-1*

- Create some data and convert it into a Pandas DataFrame

```
dict_cols = {'order':[1,2,3,4,5],'isFemale':[False,True,True,True,False]}
df = pd.DataFrame(dict_cols,index=['Tom','Char','Karen','Lin','Jess’])
display(df)
```

output

	order	isFemale
Tom	1	False
Char	2	True
Karen	3	True
Lin	4	True
Jess	5	False

```
>>> display(df.index)
```

output

```
Index(['Tom', 'Char', 'Karen', 'Lin', 'Jess'], dtype='object')
```

Row Indexes can be *non-unique*

- Create some data and convert it into a Pandas DataFrame

```
dict_cols = {'month':[2,3,3,2,3],'name':['Tom','Char','Karen','Lin','Jess']}
df = pd.DataFrame(dict_cols,index=[0,0,1,1,2])
display(df)
```

output

	month	name
0	2	Tom
0	3	Char
1	3	Karen
1	2	Lin
2	3	Jess

```
>>> display(df.index)
```

output

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

Select a Row by Row Index

- Get the row Series corresponding to the index “Char” (note the row number is 1)

df=

	order	isFemale
Tom	1	False
Char	2	True
Karen	3	True
Lin	4	True
Jess	5	False

Equivalently:

```
df.loc['Char']
```

Or

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

output

```
order           2
isFemale     True
Name: Char, dtype: object
```

Select Several Rows by Row Indexes

- Get the data corresponding to the index “Tom” and “Lin” (note the row numbers are 0 and 3)

df=

	order	isFemale
Tom	1	False
Char	2	True
Karen	3	True
Lin	4	True
Jess	5	False

Equivalently:

```
df.loc[['Tom','Lin']]
```

Or

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

output

	order	isFemale
Tom	1	False
Lin	4	True

Filter a DataFrame Conditionally

- Similar to Boolean indexing: Select observations (i.e., rows) in the DataFrame **df** that satisfies conditions

```
b = conditions(df.x1,…,df.xk)
```

- Filter the original DataFrame

```
df_filtered = df[b]
```

➢In one step (do not separately store the Boolean array **b**):

```
df_filtered =df[conditions(df.x1,…,df.xk)]
```

Filter a DataFrame Conditionally

Example: Read the CSV file **census_data.csv** into a DataFrame, and select observations where the state is California

```
df[df.state_name=='California']
```

output

	state_name	state_fips	year	population	median_age	median_income	below_poverty_line
4	California	6	2010	36637290	34.9	27733	0.137134
53	California	6	2020	39346023	36.7	34196	0.125770

Select a Column by Column Name

- Get the column Series corresponding to the variable (or column called) “isFemale”

df=

	name	order	isFemale
0	Tom	1	False
1	Char	2	True
2	Karen	3	True
3	Lin	4	True
4	Jess	5	False

Equivalently:

```
df['isFemale']
```

Or

```
df.isFemale
```

output

```
0     False
1      True
2      True
3      True
4     False
Name: isFemale, dtype: bool
```

Select Several Columns by Column Names

- Get the column Series corresponding to the variables (or columns called) “name” and “isFemale”

df=

	name	order	isFemale
0	Tom	1	False
1	Char	2	True
2	Karen	3	True
3	Lin	4	True
4	Jess	5	False

```
df[['name','isFemale']]
```

output

	name	isFemale
0	Tom	False
1	Char	True
2	Karen	True
3	Lin	True
4	Jess	False

Summary Statistics of a DataFrame

- Basic summary statistics: mean, median, standard deviation, variance, weighted average (requires sum), and quantiles (any number q in the interval [0,1])
- Operates on all columns of `df` at once (when `axis=0`)

Mean	Median	StDev	Variance	Quantiles	Sum
`df.mean()`	`df.median()`	`df.std()`	`df.var()`	`df.quantile(q)`	`df.sum()`

Notes on optional arguments:

- `skipna=True` by default (skip missing values)
- `axis='index' (0)` by default (operate across rows)

Frequency Histogram of a DataFrame

- Plot the frequency distribution of all numerical variables in DataFrame `df`

```
df.hist()
```

Notes on optional arguments:

- `density=False` by default (plots frequency histogram)
- `column` specifies one or a list of variable(s) to plot
- `bins` specifies how values are aggregated into bins for plotting

Exercise: Filter & Summarize Data

1. Read the file **census_data.csv** into a Pandas DataFrame named **df**, and print the last 10 rows of the DataFrame using `df.tail()`
2. Create a new DataFrame that includes only the rows where the **year** variable is equal to 2020, and print the shape of the new DataFrame
3. In the new DataFrame, keep only the following columns: **state_name**, **population**, **median_age**, **median_income**, and **below_poverty_line**
4. Display the rows where **state_name** includes **Texas**, **Florida**, and **Ohio**.
5. Find the number of observations where **median_income** is higher than the value for **Ohio**, and list the state names for these observations
6. Find the number of observations where **median_age** is lower than the value for **Florida**. and list the state names for these observations
7. Create a subset of the data where **population** is smaller than the population of **Texas**. For this subset: Generate summary statistics (mean, StDev, minimum, maximum, P25, and P75), and plot frequency distributions for all numeric variables (hint: use `df.select_dtypes(include="number")`)

Modify a DataFrame

Row Indexes: Default & Primary Key

- The `index` is sometimes left alone as the default index (sequence of integers from 0 to N)
- The `index` is sometimes set to the primary key, which locates each observation uniquely in the data

	ord	name	isFemale
0	1	Tom	False
1	2	Char	True
2	3	Karen	True
3	4	Lin	True
4	5	Jess	False

```
df.set_index('ord')
```

```
df.reset_index()
```

ord	name	isFemale
1	Tom	False
2	Char	True
3	Karen	True
4	Lin	True
5	Jess	False

Set Row Indexes to Some Column(s)

	`DataFrame.set_index(columns)`
Argument	The column or list of columns to be set as the new indexes
Returns	a Pandas DataFrame indexed by `columns`

	`DataFrame.set_index(columns,inplace=True)`
Argument	The column or list of columns to be set as the new indexes
Returns	None [DataFrame is directly modified, with `columns` as the indexes without returning a new object]

Reset Row Indexes to Default Integers

	`DataFrame.reset_index()`
Argument	N/A
Returns	a Pandas DataFrame with the default indexes, and the original indexes recovered in a new column

	`DataFrame.reset_index(inplace=True)`
Argument	N/A
Returns	None [DataFrame is directly modified to have the default indexes, and the original indexes recovered in a new column, without returning a new object]

Drop One or Multiple Column(s)

	`DataFrame.drop(columns)`
Argument	`columns`: column or list of columns to be dropped
Returns	a new Pandas DataFrame without `columns`

	`DataFrame.drop(columns,inplace=True)`
Argument	`columns`: column or list of columns to be dropped
Returns	None [`columns` are dropped from DataFrame, without returning a new object]

Rename One or More Column(s)

	`DataFrame.rename(columns=dict({old:new})`
Argument	`columns`: a dictionary mapping the old column names into new names (must specify `columns=` explicitly)
Returns	a Pandas DataFrame with the new column names

- `DataFrame.rename()` becomes an *inplace method* when `inplace=True` is passed as an argument

Create & Modify an Entire Column

Create (if does not exist) or modify (if exists) a column named **`col`**

```
df[col] = values
```

Examples:

➢Create a new column with numeric missing values

```
df['placeholder'] = np.nan
```

- Replace a column by casting it to string type

```
df['median_income'] = df['median_income'].astype(str)
```

- Create a new column from arithmetic operations on existing columns

```
df['pop1'] = df['population']*df['below_poverty_line']
```

Modify a Column Conditionally

→ Similar to Boolean indexing for np.array and pd.Series

Create a 1D Boolean array (of same length as number of rows in **`df`**)

```
b = (conditional expression)
```

Modify **`df`**'s **`column`** into **`values`** wherever the condition **`b`** is `True`

```
df.loc[b,column] = values
```

All in one step (no need to store the Boolean array separately):

```
df.loc[conditional statement,column] = values
```

Drop Rows Containing Missing Values

	`DataFrame.dropna(subset)`
Argument	`subset`: column or list of columns to check for missing values (default to all columns if not provided)
Returns	a Pandas DataFrame after removing any rows with NaN or None values in one of the `subset` columns

- `DataFrame.dropna()` becomes an *inplace method* when `inplace=True` is passed as an argument

Fill Missing Values in One Column

	`Series.fillna(value)`
Argument	`value`: a single scalar value to fill missing values with
Returns	a Pandas Series with missing values filled as `value`

- Avoid using the *inplace* method when you need to **modify a column Series in a larger DataFrame**. Return the new values, and use the assignment statement to modify the column instead.
- Starting in Pandas 3.0, intermediate objects (such as a Series from a larger DataFrame) will behave as a copy (hence the inplace method on the intermediate object should not change the entire DataFrame)

Fill Missing Values in Multiple Columns

	`DataFrame.fillna(value)`
Argument	`value`: a dictionary mapping each column name to a single value to fill missing data in that column with
Returns	a Pandas DataFrame with missing data filled according to the rule specified in `value`

- `DataFrame.fillna()` becomes an *inplace method* when `inplace=True` is passed as an argument

Exercise: Data Cleaning (Basic)

1. Read the file **census_data.csv** into a Pandas DataFrame named **df**. Randomly replace some values in the columns **year**, **state_fips**, or **median_income** with missing values
2. Rename **below_poverty_line** to **povline**, and rename **population** to **pop**
3. Remove the column **state_name** from the DataFrame
4. Add a new column **post**: Set it to 1 if **year** is 2020; Set it to 0 if **year** is 2010
5. Remove rows with missing values in either **year** or **state_fips**, and make sure they are both integer types
6. Fill missing values in **median_income** with its sample average
7. For rows where **year** is 2010, set **median_income** above 32000 to missing
8. Set the DataFrame's index to a combination of **state_fips** and **year**

Add a New Row

`df.loc[df.shape[0]]= newrow` where `newrow` must have matched columns (same size and type as each row Series in `df`)

`data=`

	state_name	year	median_income	population	below_poverty_line
4	California	2010	27733	36637290	0.137
53	California	2020	34196	39346023	0.126

```
data.loc[data.shape[0]] = ['Texas',2010,25385,24311891,0.168]
```

	state_name	year	median_income	population	below_poverty_line
4	California	2010	27733	36637290	0.137
53	California	2020	34196	39346023	0.126
2	Texas	2010	25385	24311891	0.168

Concatenate Rows of Two DataFrames

`pd.concat([df1,df2])` where `df1` and `df2` are two DataFrames that *may or may not have overlapping column names*

data_CA=

	state_name	year	median_income
4	California	2010	27733
53	California	2020	34196

data_TX=

	state_name	year	population
43	Texas	2010	24311891
76	Texas	2020	28635442

```
data_stacked = pd.concat([data_CA,data_TX])
```

	state_name	year	median_income	population
4	California	2010	27733.000	NaN
53	California	2020	34196.000	NaN
43	Texas	2010	NaN	24311891.000
76	Texas	2020	NaN	28635442.000

Sort Rows (Observations) by Index

	`DataFrame.sort_index(ascending,inplace)`
Argument	`ascending`: Optional and defaults to `True` `inplace`: optional and defaults to `False`
Returns	DataFrame sorted on the row indexes, or None (*inplace*)

- **Arguments**: If no argument is passed, the method returns the sorted DataFrame in ascending order of its row indexes; If **`ascending=False`**, the sort is reversed to descending order; if composite indexes, **`ascending`** can be a list of `True`/`False` values; If **`inplace=True`**, DataFrame is directly modified, and the method returns None
- **Composite indexes** consisting of multiple variables (`key0, key1,...`): DataFrame is sorted lexicographically on (`key0, key1, ...`), with earlier keys taking precedence over later keys

Sort Rows by One or More Columns

	`DataFrame.sort_values(by,ascending,inplace)`

Argument	`by`: name(s) of a column(s) to sort on
	`ascending`: Optional and defaults to `True`
	`inplace`: Optional and defaults to `False`
	[`axis`: Optional and defaults to `0` (`'index'`)]

Returns	a DataFrame with rows sorted based on by

- The **`axis`** argument: since we are primarily concerned with sorting rows (not columns), `axis` should always be 0 (default), we can simply ignore it for now
- `ascending` & `inplace` work similarly to those in `DataFrame.sort_index()`

Unique Identifier in a DataFrame

Primary Key: one or several columns that uniquely identify each row (indexes should be reset to default, as they should not contain useful data)

Two related methods to clean the data and ensure no two rows in the DataFrame should have the same primary key

- **Drop duplicates**: Keep only one (and any arbitrary one) among all the observations with the same primary key
- **Aggregate the data** *(next time)*: Group by the primary key, and aggregate the values of remaining columns into (1) the average (2) the first/last observation after sorting (3) some other value

Drop Duplicate Rows

	`DataFrame.drop_duplicates(subset,inplace)`
Argument	`subset`: Optional and defaults to all variables `inplace`: Optional and defaults to `False`
Returns	a DataFrame where no two rows have the same value across all variables in `subset`

- The **`subset`** argument: one column or a list of columns (as primary key)
- `DataFrame.drop_duplicates()` becomes an *inplace* method when `inplace=True` is passed as an argument

Exercise: Data Cleaning (Medium)

1. Read the file **census_data.csv** into a Pandas DataFrame named **df**. Create two DataFrames: **df2010** containing rows where year is 2010; **df2020** containing rows where year is 2020
2. Sort **df2010** by median_age in descending order and population in ascending order (within ties of median_age). Delete the first 10 rows of the sorted **df2010**
3. Sort **df2020** by median_income in ascending order. Delete the last 10 rows of the sorted **df2020**
4. Concatenate **df2010** and **df2020** along the rows to create a combined DataFrame
5. Sort the combined DataFrame by population in descending order
6. Drop the column year from the combined DataFrame, and then drop duplicate rows based on state_fips and state_name