Lecture 3: Data Manipulation and Data Analysis Programming and Numerical Methods for Economics—ECNM10115

Albert Rodriguez-Sala & Jacob Adenbaum
School of Economics, The University of Edinburgh

Winter 2024

This class

Pandas and Pandas Data Frames

Data Manipulation

- Create Data Frame
- Selecting Data
- Some Data Tricks: Renaming, Adding, Modifying and Dropping Data
- Data Tricks: Missing Values
- Combining Data Sets

Data Analysis

Case Study: Macro Evidence Using the Uganda ISA-LSMS

Pandas and Pandas Data Frames

Pandas

To work with data in Python we work with **pandas**, the fundamental package for data manipulation and data analysis. Some useful links:

- pandas Documentation.
- Official tutorials.
- pandas cheat-sheet posted in Learn.
- Stata to Python equivalences by Daniel Sullivan.

Pandas data frame

In pandas, the fundamental object is a *pandas data frame*. A pandas data frame it is a 2-d object (like a matrix) where each column has a name. The key parts of a data frame are

- rows = observations
- columns = variables
- the index = keeps track of the rows' locations

Thus, a data frame is similar and it looks pretty much the same as the data sets in Stata. The data type of a column in a pandas data frame is a *pandas series*

Data Manipulation

Create a data frame

- From a dictionary.
- From Series. *Series* is a pandas structure that is basically a column or row in a data frame.
- From a Numpy array or from lists.
- Importing from a file: .CSV, .Dta, .xlsx, etc.
- importing data directly from the internet, geographical data in .shp files, etc.

Example 1

Create a data frame from a set of series using *pd.DataFrame()*.

```
# Example1: Create a dataframe from series
ids = pd.Series([1, 2, 3, 4, 5])
incs = pd.Series([3000,1000,1500,4500,2000])
names = pd.Series(['Jef','Mark', 'Claire', 'Laura','Amy'])
educ = pd.Series(['primary','secondary','tertiary','secondary','secondary'])
year = pd.Series([2021,2022,2022,2022,2021])
gen = pd.Series(['male','male','female','female','female'])

df1 = pd.DataFrame({'id': ids, 'income':incs, 'name': names, 'education':educ, 'ye
print(df1)
```

Example 2

Create a data frame from a set of lists using *pd.DataFrame()*.

```
#Example2: Create dataframe with lists
# list of lists. each inner list is a row observation.
list_data = [[1,    3000,     'Jef',    'primary',  2021,    'male'],
             [2,    1000,    'Mark',  'secondary',  2022,    'male'],
             [3,    1500,  'Claire',   'tertiary',  2022,  'female'],
             [4,    4500,   'Laura',  'secondary',  2022,  'female'],
             [5,    2000,     'Amy',  'secondary',  2021,  'female']]

var_names = ['id','income','name','education','year','gender']
df2 = pd.DataFrame(data = list_data, columns=var_names)
print(df2)
```

creating data frames by importing data

In many cases we will work with existing data frames that we need to import.

- CSV: pd.read_csv()
- Excel: pd.read_excel()
- Stata: pd.read_stata()
- And other data files.

Note that to import the file, we need the file to be in the current working directory or to write down the whole path file in the command.

Import functions have many useful arguments: Google the functions to figure out how to import data with Nans, with the first row as the variables, converting or not categorical variables, and many other import options.

Selecting data: indexing—subsetting a data frame

Indexing is to choose a subset of the rows and/or columns of a data frame. The main methods to index or "locate" subsets of the data are the following

- **.iloc[] : for numeric indexing.**

```
df2.iloc[0,0]  #first row, first column
df2.iloc[0:2,1:6] # first 2 rows. 2nd to 6th column
```

- **.loc[]: for name-based and logical indexing.**

```
          df2.loc[:,['name']]  # name column
df2.loc[0,['name','education']] # name and education first observation
```

Conditional sub-setting—slicing

In many occasions we want to subset data that accomplish a specific condition. Ex: get the observations with **missing values** or **outliers**. Select observations based on a **threshold of a continuous variable** (ex: top income) or a **categorical characteristic** (being married), etc.

For that, we use **conditional sub-setting**:

- **Synatax:** df.loc[CONDITION, [VARS]].
  CONDITION is a vector of logical statements with the same length as the number of rows in the dataframe. VARS is the list of variable that we wan to take.
  Examples:
- Select data for the year 2022.
  ```
  df2.loc[df2['year']==2022]
  ```
- Select data for the year 2022 on name and income.
  ```
  df2.loc[df2['year']==2022,['name','income']]
  ```

Conditional sub-setting examples

- select data of individuals above the median income.

```
rich = df2.loc[df2['income']>df2['income'].median()]
```

- Select data set only female.

```
df_fem = df2.loc[df2['gender']=='female']
# alternatively,
df_fem  = df2.loc[df2['gender']!='male'] #not equal
# with the invert operator ~: all data that is not..
df_fem  = df2.loc[~(df2['gender']=='male')]
```

Multiple conditions sub-setting

We can use more than one condition with the logical arguments *and*/*or* in bitwise format &/ |

- **and– & .** Gender female AND on top 50% of income

```
df_femrich = df2.loc[(df2['gender']=='female') & (df2['income']>med_inc)]
```

- **or– | .** Individuals with primary education OR below an income of 2000.

```
df_educ12 = df2.loc[(df2['education']=='primary') | (df2['income']<2000)]
```

Series

A column of data is a *series* object.

- Syntax: this is a series, *df2['income']*, this is also a series *df2.income*. This is a dataframe *df2[['income']]*. check: *type(df2['income'])*.
- The content of the series can be numeric (int/floats) or other types. If the series is numeric we can apply functions, operations and methods of numeric objects, If the content of the series is strings, we can apply string methods.

```
df2['income'].mean()
df2['education'].mean()  # can't compute
```

- There are some useful functions and methods that only work with series

```
pd.value_counts(df2['education'])  #the outcome is a series
>>secondary    3
  primary      1
  tertiary     1
```

Some data tricks: renaming, adding, and modifying variables

- Renaming variables:

```
df2.rename(columns={'education':'educ'}, inplace=True)
```

- Adding variables:

```
df2['country'] = 'UK'
df2['age'] = [27,40,53,29,34]
```

- Modifying variables

```
pound_euro = 1.14
df2['income_eur']  = df2['income']*pound_euro
```

Some data tricks: modifying and dropping data

- Modifying data—or modifying variables on conditionals.

```
df2.loc[df2['name']=='Amy','income'] = 1800  # for all observations with name
df2['below_30'] = 0  # generate dummy variable with zeros everywhere
df2.loc[df2['age']<30,'below_30'] = 1  #modifiy the dummy so that individuals
```

- Dropping data: *df.drop()* function/method.

```
df2.drop(index=0)  # drop first row/observation
df2.drop(columns='below_30')  # drop column below_30
# to replace the previous data set with the new data set with the dropped data
df2.drop(columns='below_30', inplace=True) # use inplace
df2 = df2.drop(index=0)  # asing the new dropped data set with the name of the
```

Some data tricks: dummy variables

- We can create dummy variables using conditional expressions

```
df2['female'] = 1*(df2['gender']=='female')
#1* converts the true/false in 1/0.
#This expression only works correctly if there are not Nans.
```

- We can also create dummy variables with the function *pd.get_dummies()*. Especially useful when variable has multiple categories.

```
dummies_ed =pd.get_dummies(df2['education'])
```

- Note that *pd.get_dummies()* creates a new data set. Then we need to concatenate or merge this data set with out previous data.

Data tricks: working with missing values

In almost all data sets we are going to have **missing values**. Moreover, after inspecting our data set we might detect some wrong or extreme values that we want to remove from our data without losing the entire observation—assign them as missing values.
In pandas, missing values are denoted by **NaN**.

- To detect NaNs: df.isnull(), df.isna(), df.notna() and other methods

```
df2_nans.isna()  # delivers a True(if nan)/False data set
```

- We can also do it by columns

```
df2_nans['id'].isna()
```

- To replace/fill NaNs: df.fillna()

```
df2_nans['education'].fillna('missing educ')
```

- To drop rows/columns with NaNs: df.dropna()

```
df2_nans.dropna(axis=0)  # drops all rows with nans
```

Data tricks: working with missing values

CAUTION: One needs to be careful at replacing/dropping nans. Especially from the original/main data set.

- For certain computations we might want to remove the nans (if we have a deep understanding of the data).

```
df2_nans['income'].fillna(np.median(df2_nans['income']))
df2_nans['income'].fillna(0)
```

- We might want to drop observations if it has NaNs in our key variables. Use argument subset.

```
df2_nans.dropna(axis=0, subset=['id'])
```

Most of the **descriptive statistics and computational methods** in python are all written to **account for missing data**.

- When summing data, NaN values will be treated as zero.
- If the data are all NaN, the result will be 0.

Combining data sets

We might want to combine 2 or more data sets.

Concatenate: stack rows or columns to the data set.

- To **add rows**: .append() or pd.concat().
- To **add columns**: pd.join(), joins columns with other data frame either on index or on a key column. Also pd.concat().

```
df2 = df2.join(dummies_ed) #adding the education dummy vars.
```

Merge: Combine two data sets that have different variables, but there are some common observations identified by the key variables.

- **syntax**: pd.merge(data set1, data set2, on=[keys], how=method).
- keys: uniquely identify observations in both data sets (or at least in 1).
- method: describes what type of merge must be done: inner, outer, left, right.

Merge types

- **Inner merge:** only keeps observations that are present in both data frames.

```
df2.merge(df3,on='id',how='inner')
```

- **Left merge:** keeps all of the observations of the original (or main data frame) and adds the new information on the left if available.

```
df2.merge(df3,on='id',how='left')
```

- **Right merge:** only keeps info from the new data frame and adds the info of the original main data frame.

```
df2.merge(df3,on='id',how='right')
```

- **Outer merge:** method merges whenever possible but keeps the info from both data frames, even for observations where no merge/overlap occurred.

```
df2.merge(df3,on='id',how='outer')
```

Writing/saving data

To save or export the data set

- To CSV: pd.to_csv()
- To Excel: pd.to_excel()
- To Stata: pd.to_stata()
- To SAS: pd.to_sas()
- Other formats available as well.

If we do not define a path, the file will be saved/exported in the current working directory.

Data Analysis

Data Analysis

We use data analysis tools to

- **Data exploration**.
  - Initial descriptive statistics to understand the data and
  - Clean the data.
- **Empirical evidence**:
  - Find stylized facts or trends in the data. Describe the "world".
  - Facts to motivate economic theories.
- **Inference from the data**.
  - Test economic theories or impact of policies.
  - To calibrate models. Combine data+theory to create counterfactual scenarios.
- **Prediction analysis**.
  - Forecasting models. VARs and other econometric models.
  - Part of Machine Learning.

-

Basic data analysis in Python: summary statistics

- **Df.describe()**: It delivers a summary of the distribution of the variable(s). Very useful for: get a first glance of the data, detect potential outliers, and as descriptive results.

```
df2[['income','age', 'female', 'secondary','tertiary']].describe(percentiles=[0.5])
>>
```

	income	age	female	secondary	tertiary
count	5.00	5.00	5.00	5.00	5.00
mean	2,360.00	36.60	0.60	0.60	0.20
std	1,404.64	10.45	0.55	0.55	0.45
min	1,000.00	27.00	0.00	0.00	0.00
50%	1,800.00	34.00	1.00	1.00	0.00
max	4,500.00	53.00	1.00	1.00	1.00

- **Df.count()**: counts the number of non-null observations.
- **other distributional methods**: .sum(), .min(), .max(), .mode(), .abs(), .std(), .var(), .quantile(), etc.

Groupby: data aggregation or summary statistics by subgroup

Groupby is a very useful method to analyse/organize the data (the same groupby as in stata).

- Use groupby to **calculate within means**, within sum, within median, within number of observations, and other functions.

```
df2[['income','consumption','age','gender']].groupby(by='gender').mean()
>>         income  consumption   age
gender
female 2,600.00     2,000.00 38.67
male   2,000.00     1,500.00 33.50
```

- Use groupby to sum data and create a new **aggregated data sets**.

```
year_data = df2[['income','consumption','year']].groupby(by='year').sum()
```

- We can also groupby in 2 or more categories.

```
df2[['income','consumption','year','gender']].groupby(by=['year','gender']).mean()
```

apply function

For both data manipulation and data analysis, we can also apply functions from NumPy, other libraries or our own created functions to the data using **.apply()**.

Examples:

- Apply series function (*pd.value_counts*) to a data frame:

```
 # Count the unique values per each column (i.e. discrete distribution if /n)
df2[['income','consumption','year','gender']].groupby(by=['year','gender']).mean()
```

- Apply user-created functions:

```
# Compute the Gini of income separating by gender
df2[['income','gender']].groupby(by='gender').apply(gini)
```

- Apply user-created lambda funtions:

```
# compute age squared
df2['age_sq'] = df2['age'].apply(lambda x: x**2)
```

- For element-wise operations that don't work with apply, use *applymap*.

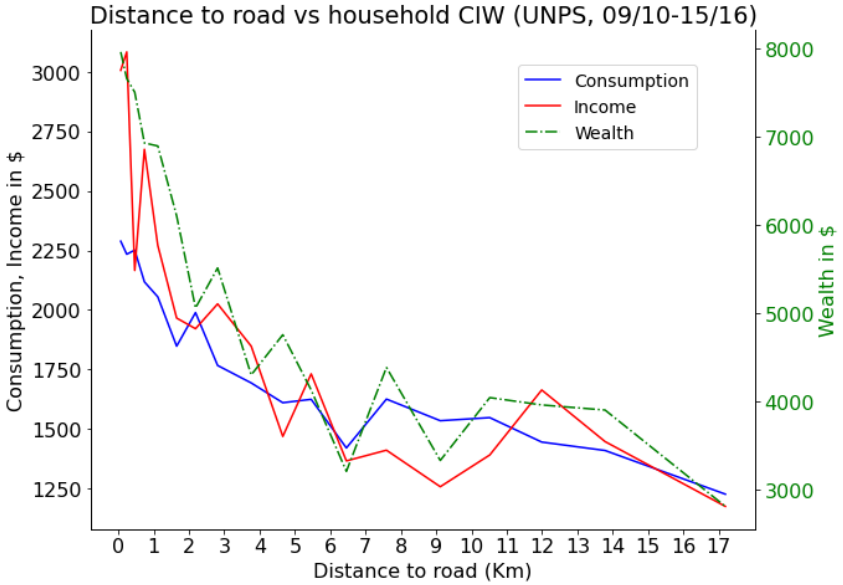
```
# Set the names in uppercase.
df2[['name']] = df2[['name']].applymap(str.upper)
```

Plotting the data

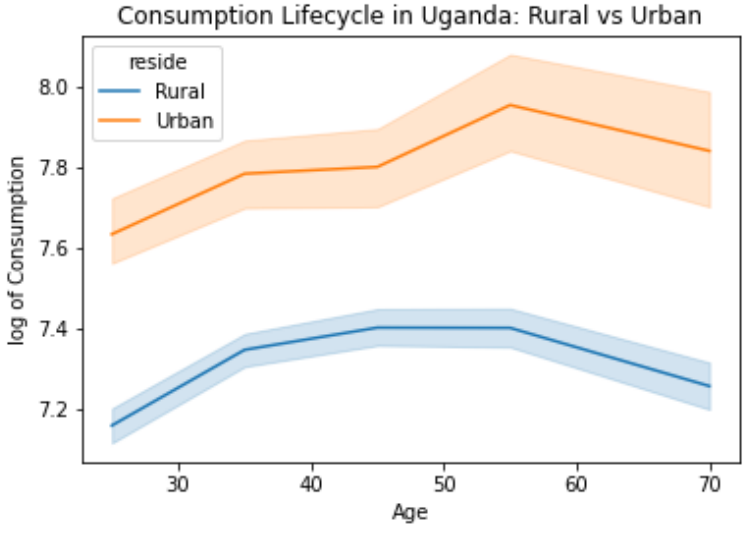
Visualizing the data through plots is a very useful method to understand and present the data. With Python we can create many different and beautiful plots using libraries as matplotlib and seaborn.

- Matplotlib. Standard plot library.
- Seaborn: Library for statistical data visualization.
- Note that we can do the plot directly, or create a new object (with figure,axis) and apply methods to that object (as in examples).

Plot example: roads and development (Uganda)



Plot example from case study: consumption lifecycle



Case Study: Macro Evidence Using the Uganda ISA-LSMS

Macro evidence in poor countries using micro data: ISA-LSMS

Nationally representative household data is scarce in poor countries. A special difficulty is to have good measures of household income. Yet this type of data is key to understanding progress and welfare of a country.

The World Bank together with national agencies implement the ISA-LSMS surveys across some Sub-Saharan countries. ISA-LSMS: Integrated Surveys in Agriculture and Living Strandards Measurement Surveys. Open data.

- Nationally representative panel household data. Open (free) data.
- We can get the consumption, income and wealth distributions and dynammics from a single data set.
- Rich also in labor, education, health, other social-demographic characteristics.
- Altogether... with a single data set we can recover the ENTIRE BUDGET CONSTRAINT of the HOUSEHOLDS.

ISA-LSMS Data: the Macroeconomics of Sub-Saharan Africa

De Magalhães, L., Santaeulàlia-Llopis, R. (2018). The consumption, income, and wealth of the poorest: An empirical analysis of economic inequality in rural and urban Sub-Saharan Africa for macroeconomists. Journal of Development Economics.

The paper show us how much we can learn in economics from mostly descriptive statistics of nationally representative panel household data. Some key insights:

- Income inequality is relative close to the US, yet consumption and wealth inequality are lower, specially in rural areas.
  1. Low transmission from income to wealth.
  2. Low transmission from income to consumption.
- high level of consumption insurance.
  1. Cannot reject full-insurance in rural areas.
  2. Trade-off insurance vs growth: urban areas richer but lower insurance.

Case study: Uganda ISA-LSMS

- Let's use the 2013-2014 Uganda ISA-LSMS (UNPS) as a case study to apply some of the tools learn in class and results in **De Magalhães, L., Santaeulàlia-Llopis, R. (2018)**.
- Download *data13.csv* in learn. This is already a clean data set.
- Open *lecture3_casestudy.py*. Change the directory file to your path of data13.csv. Let's run the code!