

Refugees

July 14, 2026

1 Pandas

- <https://pandas.pydata.org>
- very high-level data containers with corresponding functionality
- many useful tools to work with time-series (look at `Series.rolling`)
- many SQL-like data operations (group, join, merge)
- Interface to a large variety of file formats (see `pd.read_...` functions)
- additional package with data-interface/API to many data repositories (https://pandas-datareader.readthedocs.io/en/latest/remote_data.html)

```
[1]: import pandas as pd
```

1.1 Basic Data Structures

1.1.1 Series

One-dimensional ndarray with axis labels (called index).

Series can be created like an array

```
[2]: pd.Series([11,13,17,19,23])
```

```
[2]: 0    11
     1    13
     2    17
     3    19
     4    23
     dtype: int64
```

or, if you want a special index

```
[3]: series = pd.Series([11,13,17,19,23], index=['a', 'b', 'c', 'd', 'e'])
     print(series)
```

```
a    11
b    13
c    17
d    19
e    23
dtype: int64
```

to get the content back you can use

```
[4]: series.index
```

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

```
[5]: series.values
```

```
[5]: array([11, 13, 17, 19, 23])
```

but the power of pandas lies in all the other attributes

```
[6]: #series. [TAB]
```

1.1.2 DataFrame

The primary pandas data structure.

Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes. (index: row labels, columns: column labels) Can be thought of as a dict-like container for Series objects.

The easiest way to create a DataFrame is to read it from an input file (see later)

In addition there are many ways to create DataFrames manually. Most straight forward probably is to use a dict of iterables. (Series, Lists, Arrays). Pandas tries to choose sensible indexes.

```
[7]: frame = pd.DataFrame({"primes": series, "fibonacci": [1,1,2,3,5], "0-4": range(5)})
```

```
[8]: print(frame)
```

	primes	fibonacci	0-4
a	11	1	0
b	13	1	1
c	17	2	2
d	19	3	3
e	23	5	4

2 Refugee Example

We now want to use pandas to work with data from the World Bank. My goal is to create a plot showing the burden refugees put on different countries. For this we will plot the fraction of refugee in a give countries population versus that countries GDP.

I downloaded and extracted the following data-sets from the Worldbank website manually:

- * Refugee population by country or territory of asylum: <https://data.worldbank.org/indicator/SM.POP.REFG> (discontinued since 2023)
- * Population, total: <https://data.worldbank.org/indicator/SP.POP.TOTL>
- * GDP per capita (current US\$): <https://data.worldbank.org/indicator/NY.GDP.PCAP.CD>

```
[9]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

2.1 Loading and Accessing Data

loading a data file with pandas is trivial

```
[10]: refugees = pd.read_csv("data/refugee-population.csv", skiprows=4)
```

```
[11]: refugees.head()
```

```
[11]:
```

	Country Name	Country Code	\
0	Aruba	ABW	
1	Africa Eastern and Southern	AFE	
2	Afghanistan	AFG	
3	Africa Western and Central	AFW	
4	Angola	AGO	

	Indicator Name	Indicator Code	1960	\
0	Refugee population by country or territory of ...	SM.POP.REFG	NaN	
1	Refugee population by country or territory of ...	SM.POP.REFG	NaN	
2	Refugee population by country or territory of ...	SM.POP.REFG	NaN	
3	Refugee population by country or territory of ...	SM.POP.REFG	NaN	
4	Refugee population by country or territory of ...	SM.POP.REFG	NaN	

	1961	1962	1963	1964	1965	...	2015	2016	2017	\
0	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	
1	NaN	NaN	NaN	NaN	NaN	...	3333273.0	3990478.0	5155400.0	
2	NaN	NaN	NaN	NaN	NaN	...	257553.0	59770.0	75927.0	
3	NaN	NaN	NaN	NaN	NaN	...	1138010.0	1200854.0	1172523.0	
4	NaN	NaN	NaN	NaN	NaN	...	15547.0	15547.0	41119.0	

	2018	2019	2020	2021	2022	2023	\
0	NaN	NaN	NaN	NaN	NaN	NaN	
1	5114399.0	5087755.0	5183533.0	5436720.0	5412266.0	5553759.0	
2	72228.0	72227.0	72278.0	66949.0	52159.0	34826.0	
3	1285773.0	1315229.0	1474135.0	1631057.0	1702392.0	2296159.0	
4	39856.0	25793.0	25791.0	26045.0	25514.0	25174.0	


```
Unnamed: 68
```

0	NaN
1	NaN
2	NaN
3	NaN
4	NaN


```
[5 rows x 69 columns]
```

As you can see pandas choose the right column labels and numbered the rows continuously.

We can easily change the row labels (the index) to one of the columns.

```
[12]: refugees.set_index(["Country Code"], inplace=True)
```

```
[13]: refugees.head()
```

```
[13]:
```

	Country Name \
Country Code	
ABW	Aruba
AFE	Africa Eastern and Southern
AFG	Afghanistan
AFW	Africa Western and Central
AGO	Angola

	Indicator Name \
Country Code	
ABW	Refugee population by country or territory of ...
AFE	Refugee population by country or territory of ...
AFG	Refugee population by country or territory of ...
AFW	Refugee population by country or territory of ...
AGO	Refugee population by country or territory of ...

	Indicator Code	1960	1961	1962	1963	1964	1965	1966	...	\
Country Code									...	
ABW	SM.POP.REFG	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	
AFE	SM.POP.REFG	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	
AFG	SM.POP.REFG	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	
AFW	SM.POP.REFG	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	
AGO	SM.POP.REFG	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	

	2015	2016	2017	2018	2019	\
Country Code						
ABW	NaN	NaN	NaN	NaN	NaN	
AFE	3333273.0	3990478.0	5155400.0	5114399.0	5087755.0	
AFG	257553.0	59770.0	75927.0	72228.0	72227.0	
AFW	1138010.0	1200854.0	1172523.0	1285773.0	1315229.0	
AGO	15547.0	15547.0	41119.0	39856.0	25793.0	

	2020	2021	2022	2023	Unnamed: 68
Country Code					
ABW	NaN	NaN	NaN	NaN	NaN
AFE	5183533.0	5436720.0	5412266.0	5553759.0	NaN
AFG	72278.0	66949.0	52159.0	34826.0	NaN
AFW	1474135.0	1631057.0	1702392.0	2296159.0	NaN
AGO	25791.0	26045.0	25514.0	25174.0	NaN

[5 rows x 68 columns]

Now it's easy to select rows or columns

```
[14]: refugees.loc[["CHE", "DEU"]]
```

```
[14]:
```

	Country Name	Indicator Name	\
	Country Code		
CHE	Switzerland	Refugee population by country or territory of ...	
DEU	Germany	Refugee population by country or territory of ...	

	Indicator Code	1960	1961	1962	1963	1964	\
	Country Code						
CHE	SM.POP.REFG	20000.0	20000.0	20000.0	20000.0	20000.0	
DEU	SM.POP.REFG	197000.0	190000.0	185000.0	182000.0	180000.0	

	1965	1966	...	2015	2016	2017	\
	Country Code		...				
CHE	20000.0	20500.0	...	73326.0	82668.0	93030.0	
DEU	180000.0	140000.0	...	316098.0	669468.0	970357.0	

	2018	2019	2020	2021	2022	\
	Country Code					
CHE	104011.0	110162.0	115798.0	118829.0	182474.0	
DEU	1063835.0	1146682.0	1210596.0	1255694.0	2075445.0	

	2023	Unnamed: 68
	Country Code	
CHE	192507.0	NaN
DEU	2593007.0	NaN

[2 rows x 68 columns]

```
[15]: refugees[["1990", "2000"]].head()
```

```
[15]:
```

	1990	2000
	Country Code	
ABW	NaN	NaN
AFE	4709569.0	2444941.0
AFG	50.0	NaN
AFW	932052.0	968325.0
AGO	11557.0	12086.0

```
[16]: refugees.get(["1990", "2000"]).head()
```

```
[16]:
```

	1990	2000
	Country Code	
ABW	NaN	NaN

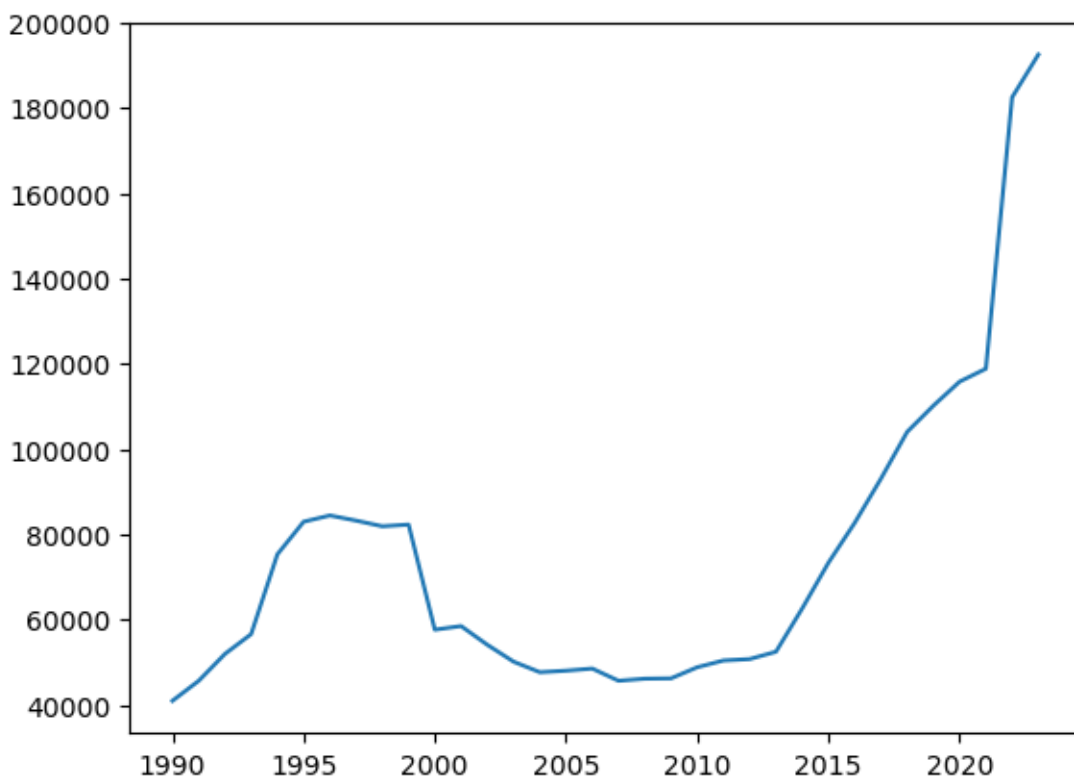
AFE	4709569.0	2444941.0
AFG	50.0	NaN
AFW	932052.0	968325.0
AGO	11557.0	12086.0

2.2 Working with a Single Country

With this we now choose the data for one country, remove all missing values and then create a plot:

```
[17]: che = refugees.loc["CHE"][[str(year) for year in range(1990,2024)]]
```

```
[18]: che.dropna().plot()
plt.show()
```



Usually it is easier to work with real datetime objects instead of strings. So we convert the index to datetime

```
[19]: che.index.values
```

```
[19]: array(['1990', '1991', '1992', '1993', '1994', '1995', '1996', '1997',
          '1998', '1999', '2000', '2001', '2002', '2003', '2004', '2005',
          '2006', '2007', '2008', '2009', '2010', '2011', '2012', '2013',
```

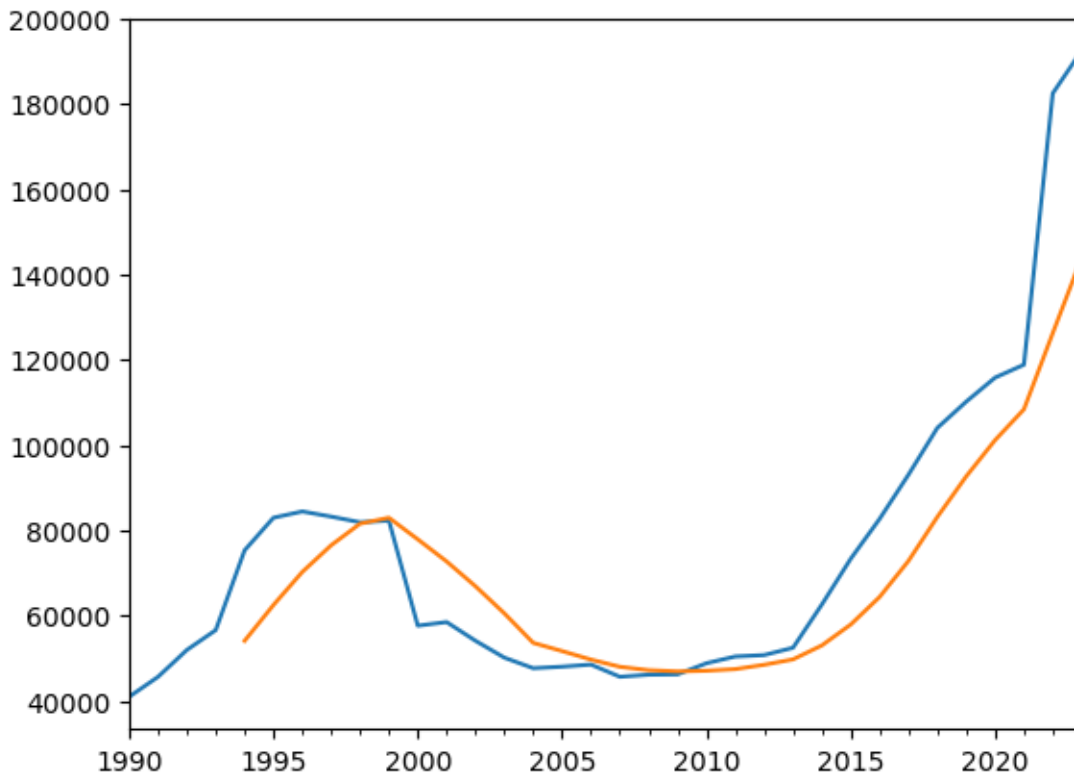
```
'2014', '2015', '2016', '2017', '2018', '2019', '2020', '2021',
'2022', '2023'], dtype=object)
```

```
[20]: che.index = pd.to_datetime(che.index, format="%Y")
print(che.index)
```

```
DatetimeIndex(['1990-01-01', '1991-01-01', '1992-01-01', '1993-01-01',
               '1994-01-01', '1995-01-01', '1996-01-01', '1997-01-01',
               '1998-01-01', '1999-01-01', '2000-01-01', '2001-01-01',
               '2002-01-01', '2003-01-01', '2004-01-01', '2005-01-01',
               '2006-01-01', '2007-01-01', '2008-01-01', '2009-01-01',
               '2010-01-01', '2011-01-01', '2012-01-01', '2013-01-01',
               '2014-01-01', '2015-01-01', '2016-01-01', '2017-01-01',
               '2018-01-01', '2019-01-01', '2020-01-01', '2021-01-01',
               '2022-01-01', '2023-01-01'],
              dtype='datetime64[ns]', freq=None)
```

As mentioned in the introduction, **pandas** offers a very usefull **rolling** method

```
[21]: che.plot()
che.rolling(center=False,window=5).mean().plot()
plt.show()
```



2.3 Removing Unwanted Data

We now want to create a scatter plot with refugees divided by population vs. gdp-per-captita. For each data set we will use the mean of the last 5 years.

Some of the rows and columns in the World-Bank Files are of no interest for this. We can remove these easily.

2.3.1 Excluding Non-Countries

The World-Bank provides meta-data for each country, where we can identify rows with non-countries (e.g. regional aggregates)

```
[22]: !head data/metadata-countries_population.csv
```

```
"Country Code","Region","IncomeGroup","SpecialNotes","TableName",  
"ABW","Latin America & Caribbean","High income","", "Aruba",  
"AFE","", "", "26 countries, stretching from the Red Sea in the North to the Cape  
of Good Hope in the South (https://www.worldbank.org/en/region/afr/eastern-and-southern-africa)", "Africa Eastern and Southern",  
"AFG","South Asia","Low income","The reporting period for national accounts data  
is designated as either calendar year basis (CY) or fiscal year basis (FY). For  
this country, it is fiscal year-based (fiscal year-end: March 20). Also, an  
estimate (PA.NUS.ATLS) of the exchange rate covers the same period and thus  
differs from the official exchange rate (CY).
```

In addition, the World Bank systematically assesses the appropriateness of official exchange rates as conversion factors. In this country, multiple or dual exchange rate activity exists and must be accounted for appropriately in underlying statistics. An alternative estimate ("alternative conversion factor" - PA.NUS.ATLS) is thus calculated as a weighted average of the different exchange rates in use in the country. Doing so better reflects economic reality and leads to more accurate cross-country comparisons and country classifications by income level. For this country, this applies to the period 1960-2006.

Alternative conversion factors are used in the Atlas methodology and elsewhere in World Development Indicators as single-year conversion factors.", "Afghanistan",

```
"AFW","", "", "22 countries, stretching from the westernmost point of Africa,  
across the equator, and partly along the Atlantic Ocean till the Republic of  
Congo in the South (https://www.worldbank.org/en/region/afr/western-and-central-africa)", "Africa Western and Central",
```

```
"AGO","Sub-Saharan Africa","Lower middle income","The World Bank systematically  
assesses the appropriateness of official exchange rates as conversion factors.  
In this country, multiple or dual exchange rate activity exists and must be  
accounted for appropriately in underlying statistics. An alternative estimate  
("alternative conversion factor" - PA.NUS.ATLS) is thus calculated as a weighted  
average of the different exchange rates in use in the country. Doing so better  
reflects economic reality and leads to more accurate cross-country comparisons  
and country classifications by income level. For this country, this applies to
```

the period 1994–2023. Alternative conversion factors are used in the Atlas methodology and elsewhere in World Development Indicators as single-year conversion factors.", "Angola",
 "ALB", "Europe & Central Asia", "Upper middle income", "", "Albania",
 "AND", "Europe & Central Asia", "High income", "", "Andorra",

We load this file and extract the two relevant columns

```
[23]: meta = pd.read_csv("data/metadata-countries_population.csv")
```

```
[24]: meta.columns
```

```
[24]: Index(['Country Code', 'Region', 'IncomeGroup', 'SpecialNotes', 'TableName',  
        'Unnamed: 5'],  
        dtype='object')
```

```
[25]: meta = meta[['Country Code', 'Region']]
```

```
[26]: meta.head()
```

```
[26]:   Country Code      Region
0          ABW  Latin America & Caribbean
1          AFE                      NaN
2          AFG          South Asia
3          AFW                      NaN
4          AGO      Sub-Saharan Africa
```

```
[27]: meta.set_index("Country Code", inplace=True)
```

From this we create a list of non-countries

```
[28]: non_countries = meta.loc[meta.Region.isnull()].index
      print(non_countries)
```

```
Index(['AFE', 'AFW', 'ARB', 'CEB', 'CSS', 'EAP', 'EAR', 'EAS', 'ECA', 'ECS',  
      'EMU', 'EUU', 'FCS', 'HIC', 'HPC', 'IBD', 'IBT', 'IDA', 'IDB', 'IDX',  
      'LAC', 'LCN', 'LDC', 'LIC', 'LMC', 'LMY', 'LTE', 'MEA', 'MIC', 'MNA',  
      'NAC', 'OED', 'OSS', 'PRE', 'PSS', 'PST', 'SAS', 'SSA', 'SSF', 'SST',  
      'TEA', 'TEC', 'TLA', 'TMN', 'TSA', 'TSS', 'UMC', 'WLD'],  
      dtype='object', name='Country Code')
```

and finally exclude the relevant rows

```
[29]: refugees = refugees.drop(non_countries)
```

2.3.2 Excluding Columns

The data contains a few rows with unneeded text

```
[30]: refugees.columns
```

```
[30]: Index(['Country Name', 'Indicator Name', 'Indicator Code', '1960', '1961',
          '1962', '1963', '1964', '1965', '1966', '1967', '1968', '1969', '1970',
          '1971', '1972', '1973', '1974', '1975', '1976', '1977', '1978', '1979',
          '1980', '1981', '1982', '1983', '1984', '1985', '1986', '1987', '1988',
          '1989', '1990', '1991', '1992', '1993', '1994', '1995', '1996', '1997',
          '1998', '1999', '2000', '2001', '2002', '2003', '2004', '2005', '2006',
          '2007', '2008', '2009', '2010', '2011', '2012', '2013', '2014', '2015',
          '2016', '2017', '2018', '2019', '2020', '2021', '2022', '2023',
          'Unnamed: 68'],
          dtype='object')
```

In addition, the last column might be missig a lot of data

```
[31]: np.sum(refugees["2023"].notnull())
```

```
[31]: np.int64(165)
```

so we can create a list of all interesting columns

```
[32]: useful_cols = []
      last_year = 2023 # depending on output above
      for year in range(last_year-5, last_year+1):
          useful_cols.append(str(year))
```

```
[33]: useful_cols
```

```
[33]: ['2018', '2019', '2020', '2021', '2022', '2023']
```

with this, we:

- select the reduced dataset
- switch the index to Country Code
- calculate the mean for each country

```
[34]: refugees = refugees[useful_cols]
```

```
[35]: refugee_means = refugees.mean(axis=1)
```

2.4 Loading Additional Files

Of course we could execute these commands again manually for the two remaining data-files. However, the proper way to solve this is to create a function for this. Especially since all files have the exact same structure.

```
[36]: def load_file(file):
      """Load and process a Worldbank File"""
      data = pd.read_csv(file, skiprows=4)
      data.set_index("Country Code", inplace=True)
      data.drop(non_countries, inplace=True)
      data = data[useful_cols]
```

```
return data.mean(axis=1), data
```

```
[37]: gdp_means, gdp = load_file("data/gdp-per-capita.csv")
```

```
[38]: gdp_means.head()
```

```
[38]: Country Code
ABW      29851.603885
AFG       442.053744
AGO      2225.540349
ALB      6263.667603
AND     42067.093210
dtype: float64
```

```
[39]: gdp.head()
```

```
[39]:
```

	2018	2019	2020	2021 \
Country Code				
ABW	30918.483584	31902.809818	24008.127822	29127.759384
AFG	492.090631	497.741431	512.055098	355.777826
AGO	2540.508879	2191.347764	1450.905111	1927.474078
ALB	5287.660801	5396.214243	5343.037704	6377.203096
AND	42904.828456	41328.600498	37207.221999	42066.490518

	2022	2023
Country Code		
ABW	33300.838819	NaN
AFG	352.603733	NaN
AGO	2933.484644	2309.521620
ALB	6810.114041	8367.775731
AND	42350.697069	46544.720720

```
[40]: population_means, population = load_file("data/population.csv")
```

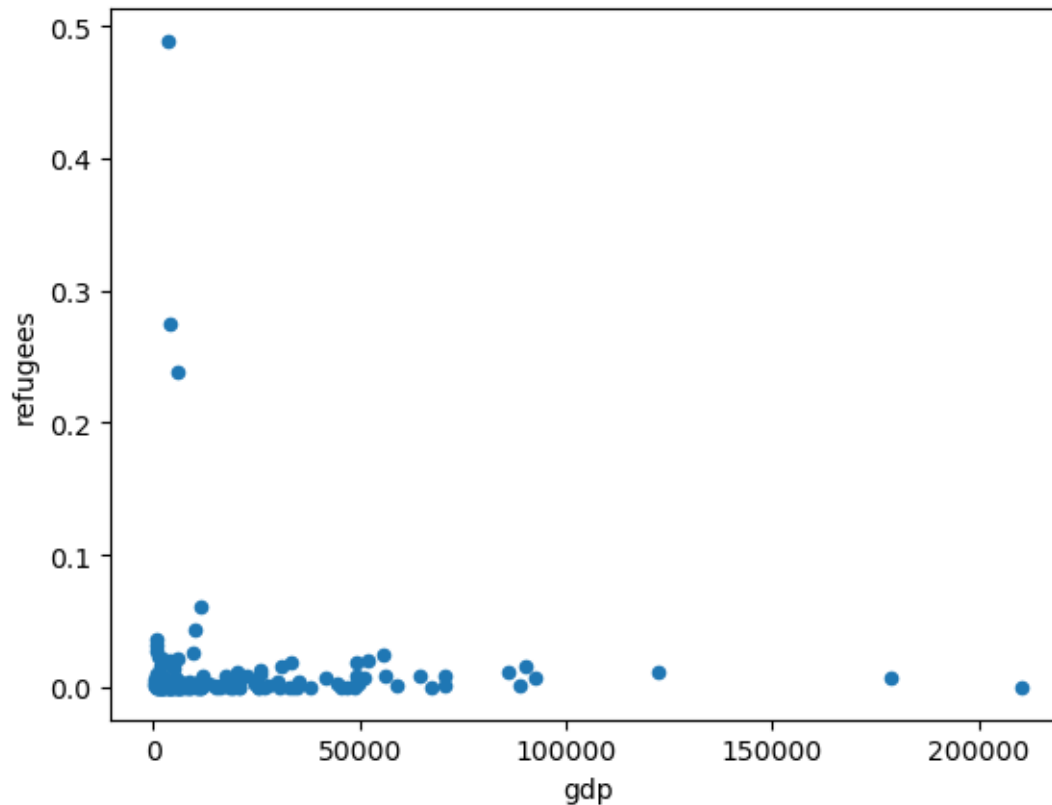
2.5 Creating the Plot

We now combine our three Series with means into one DataFrame and create our plot.

```
[41]: data = pd.DataFrame({"gdp": gdp_means, "refugees": refugee_means/  
↪ population_means}).dropna()
```

(Here we loose some countries with missing data.)

```
[42]: data.plot.scatter("gdp", "refugees")  
plt.show()
```



We can quickly find out who the three top countries are:

```
[43]: data.where(data["refugees"]>0.1).dropna()
```

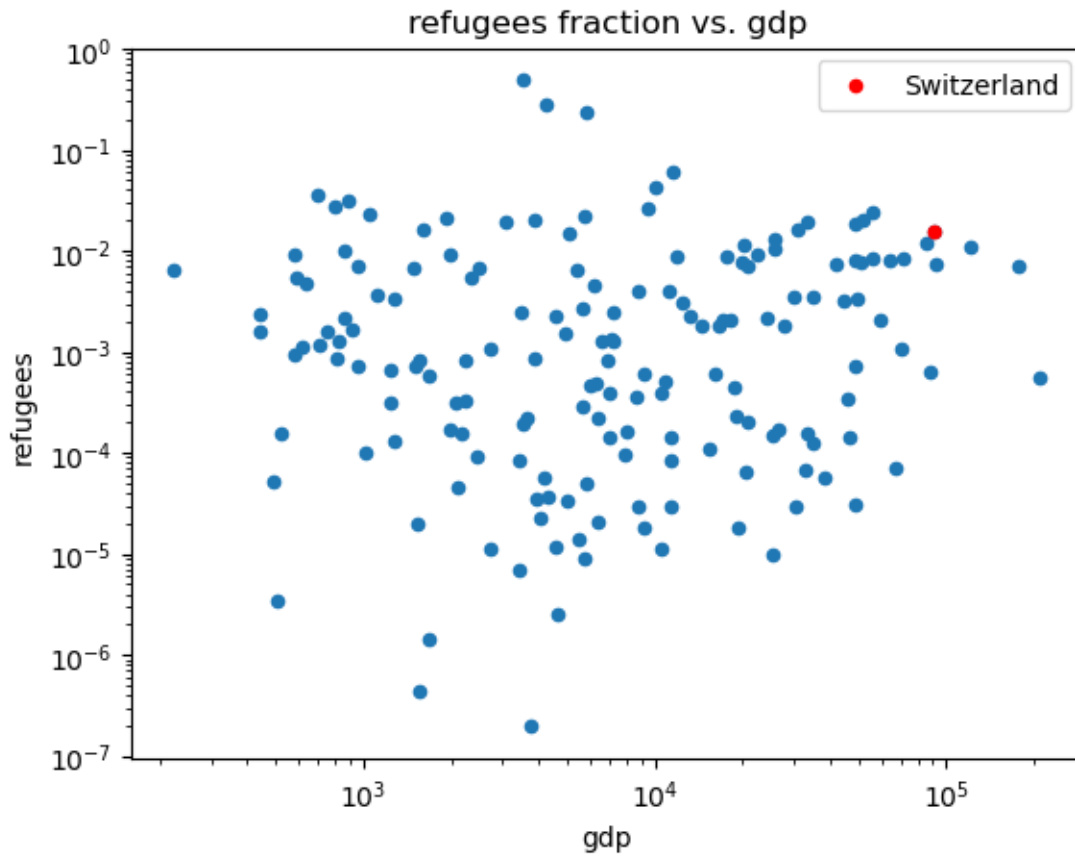
```
[43]:
```

	gdp	refugees
Country Code		
JOR	4208.429706	0.274945
LBN	5843.601048	0.238846
PSE	3549.825962	0.488927

To improve readability:

- we switch to a log-log axis (we need to exclude countries with too small refugee numbers)
- we highlight one selected country
- We add a title

```
[44]: ax = data[data["refugees"] > 1e-10].plot.scatter(y="refugees", x="gdp",
↳loglog=True)
ax = data.loc[["CHE"]].plot.scatter(y="refugees", x="gdp", ax=ax, color="r",
↳label="Switzerland")
plt.title("refugees fraction vs. gdp")
plt.show()
```



again we can print the info for one country

```
[45]: data.loc["CHE"]
```

```
[45]: gdp          90323.060451
      refugees      0.015824
      Name: CHE, dtype: float64
```

2.5.1 Highlighting a Full Region

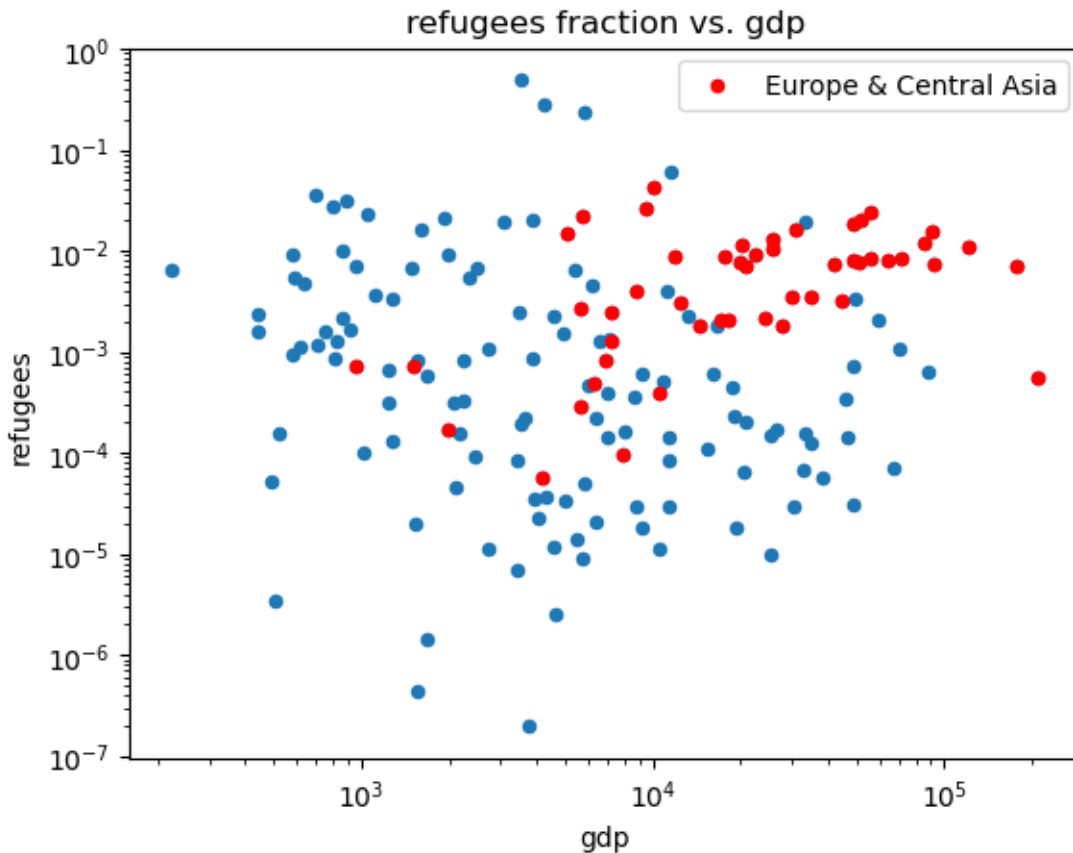
Based on the meta data provided by the World Bank, we can highlight a region

```
[46]: europe = meta.loc[meta.Region == "Europe & Central Asia"].index
```

```
[47]: europe[:10]
```

```
[47]: Index(['ALB', 'AND', 'ARM', 'AUT', 'AZE', 'BEL', 'BGR', 'BIH', 'BLR', 'CHE'],
      dtype='object', name='Country Code')
```

```
[48]: ax = data[data["refugees"] > 1e-10].plot.scatter(y="refugees", x="gdp",
    ↪loglog=True)
ax = data.loc[data.index.intersection(europe)].plot.scatter(y="refugees",
    ↪x="gdp", ax=ax, color="r", label="Europe & Central Asia")
plt.title("refugees fraction vs. gdp")
plt.show()
```



(As we lost some countries with missing data when we called `dropna` above, we need the `data.index.intersection`-call to select only country codes really contained in our data.)

2.6 Fitting

We now look at a tiny subset of this data and look at ways to fit a function to it.

Scipy prepare a huge number of options, we will look at three options of increasing complexity and flexibility.

2.6.1 Preparations

first we select our subset

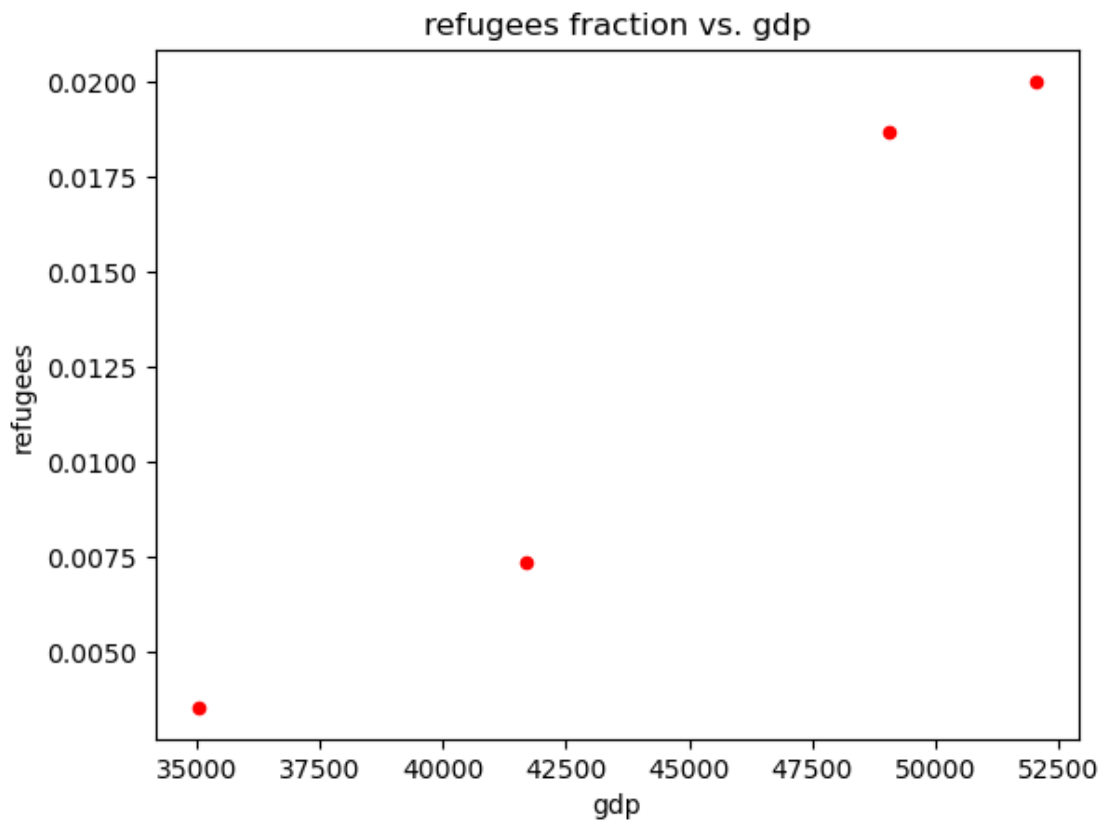
```
[49]: europe_small = ['AUT',  
    'DEU',  
    'FRA',  
    'ITA',  
    ]
```

```
[50]: data_eu = data.loc[europe_small].dropna()  
data_eu
```

```
[50]:
```

	gdp	refugees
Country Code		
AUT	52072.030011	0.020003
DEU	49064.065027	0.018667
FRA	41708.479489	0.007363
ITA	35016.983603	0.003547

```
[51]: ax = data_eu.plot.scatter(y="refugees", x="gdp", color="r")  
plt.title("refugees fraction vs. gdp")  
plt.show()
```



and we create a vector with all the x values we will need to plot our fit result

```
[52]: x = np.linspace(data_eu["gdp"].min(), data_eu["gdp"].max(), 100)
```

2.6.2 polyfit

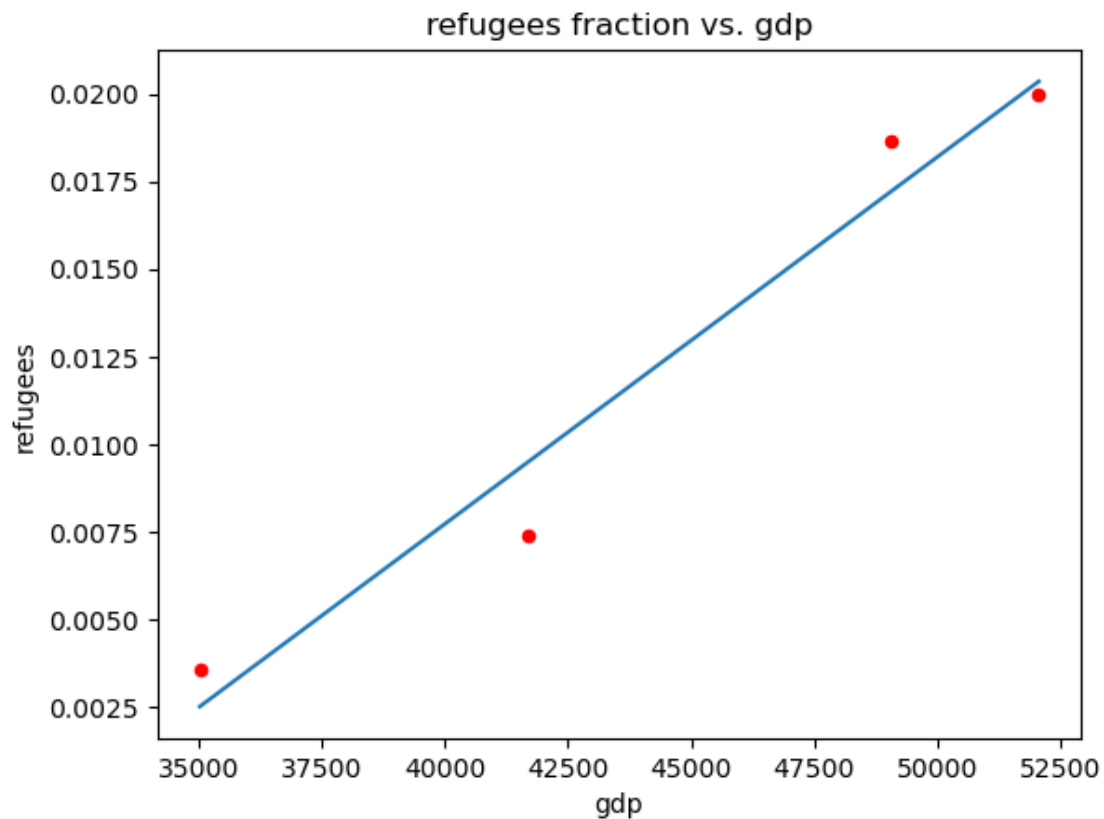
Polyfit is probably the easiest way to fit a polynome to given data.

```
[53]: from numpy import polyfit, polyval
```

```
[54]: res = polyfit(data_eu["gdp"], data_eu["refugees"],1)
print(res)
```

```
[ 1.04719896e-06 -3.41693157e-02]
```

```
[55]: ax = data_eu.plot.scatter(y="refugees", x="gdp", color="r")
ax.plot(x, polyval(res, x))
plt.title("refugees fraction vs. gdp")
plt.show()
```



2.6.3 curve_fit

With `curve_fit` you can define a complex fit function.

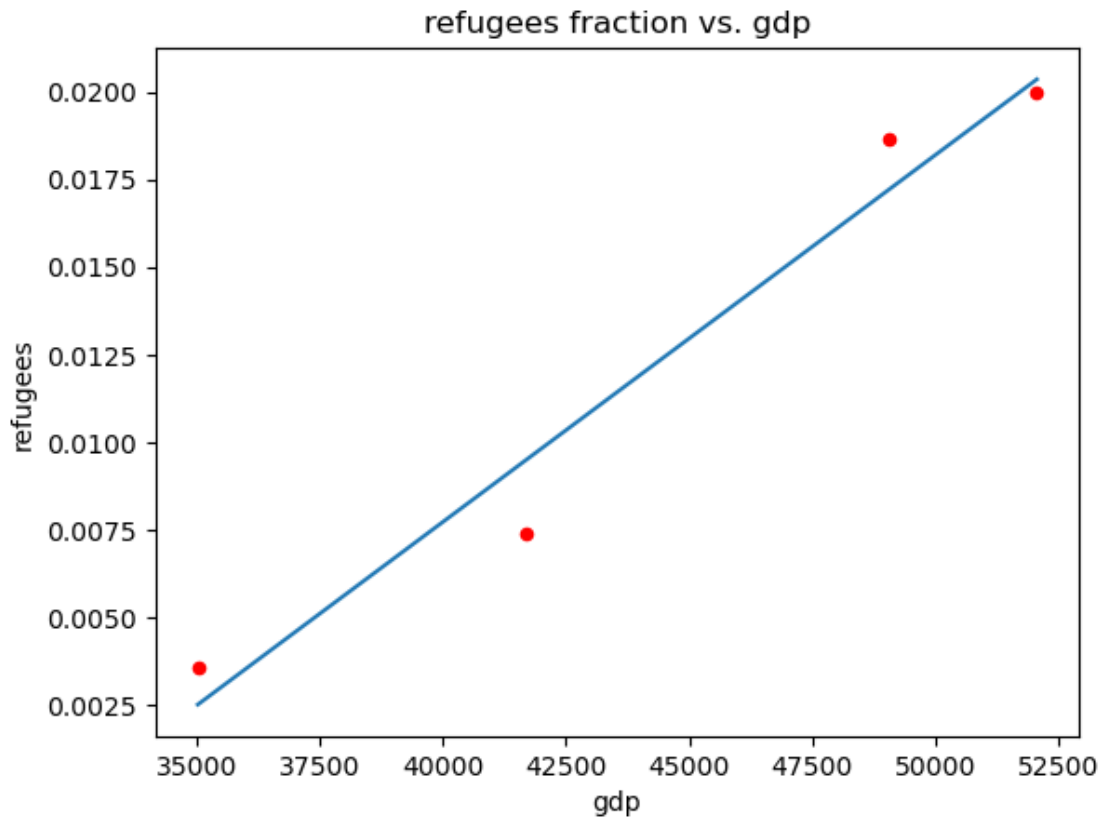
```
[56]: from scipy.optimize import curve_fit
```

```
[57]: def fit_function(x,b,c):  
      return b*x+c
```

```
[58]: res = curve_fit(fit_function, data_eu["gdp"], data_eu["refugees"])  
      print(res)
```

```
(array([ 1.04719896e-06, -3.41693157e-02]), array([[ 2.25835729e-14,  
-1.00418736e-09],  
          [-1.00418736e-09,  4.56445915e-05]]))
```

```
[59]: ax = data_eu.plot.scatter(y="refugees", x="gdp", color="r")  
      ax.plot(x, fit_function(x, *(res[0])))  
      plt.title("refugees fraction vs. gdp")  
      plt.show()
```



2.6.4 leastsq

Finally, least-squares allows you to even specify the cost function. With this you can factor in uncertainties or weights for your data points.

```
[60]: from scipy.optimize import leastsq
```

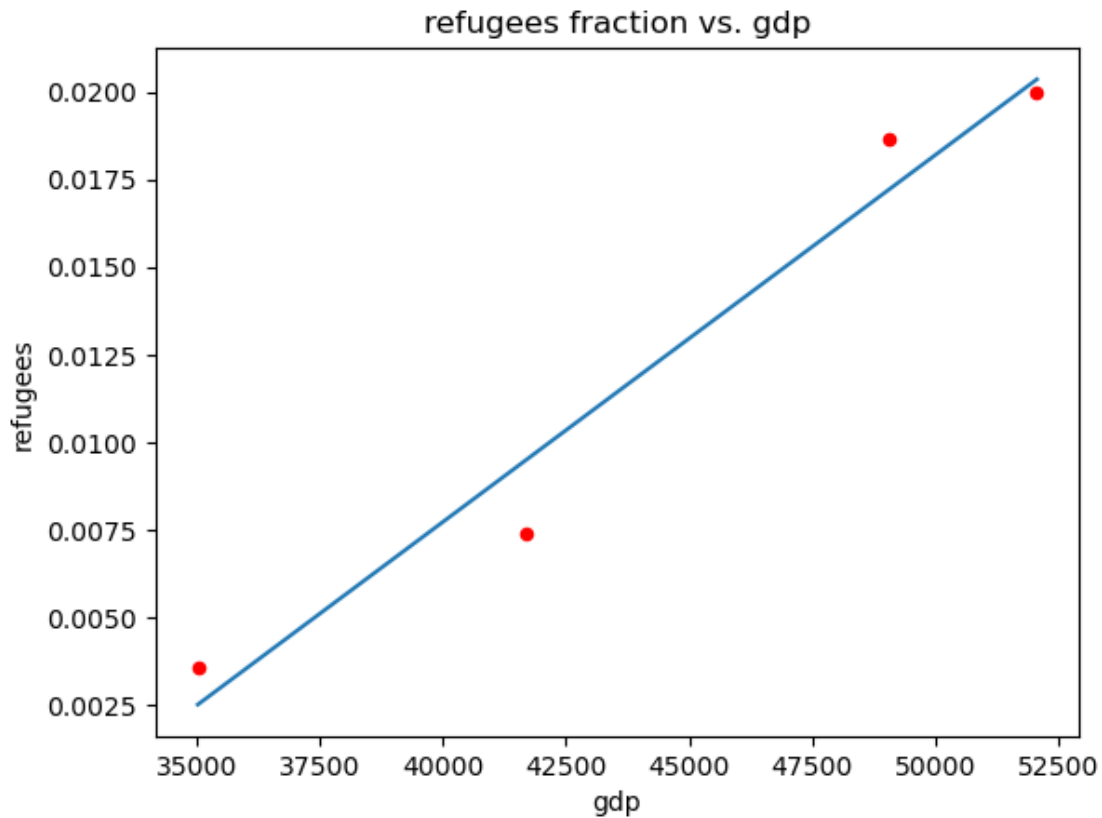
```
[61]: def fit_function(x, p):  
      return x*p[0]+p[1]
```

```
[62]: def error_function(params):  
      return data_eu["refugees"] - fit_function(data_eu["gdp"], params)
```

```
[63]: res = leastsq(error_function, [0,0])  
      print(res)
```

```
(array([ 1.04719896e-06, -3.41693157e-02]), 1)
```

```
[64]: ax = data_eu.plot.scatter(y="refugees", x="gdp", color="r")  
      ax.plot(x, fit_function(x, res[0]))  
      plt.title("refugees fraction vs. gdp")  
      plt.show()
```



2.6.5 statsmodels

```
[65]: import statsmodels.formula.api as smf
```

```
[66]: res = smf.ols("refugees ~ gdp", data=data_eu).fit()
```

```
[67]: print(res.summary())
```

```

                        OLS Regression Results
=====
Dep. Variable:          refugees      R-squared:                0.960
Model:                  OLS          Adj. R-squared:            0.941
Method:                 Least Squares   F-statistic:             48.56
Date:                  Tue, 14 Jul 2026   Prob (F-statistic):       0.0200
Time:                  07:41:10         Log-Likelihood:          20.583
No. Observations:      4              AIC:                   -37.17
Df Residuals:          2              BIC:                   -38.39
Df Model:              1
Covariance Type:       nonrobust
=====
                        coef      std err          t      P>|t|      [0.025      0.975]
-----
Intercept             -0.0342      0.007      -5.058      0.037      -0.063      -0.005
gdp                   1.047e-06    1.5e-07      6.968      0.020      4.01e-07    1.69e-06
=====
Omnibus:              nan      Durbin-Watson:           3.329
Prob(Omnibus):        nan      Jarque-Bera (JB):        0.452
Skew:                 -0.507     Prob(JB):               0.798
Kurtosis:             1.703     Cond. No.                3.05e+05
=====
```

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

[2] The condition number is large, 3.05e+05. This might indicate that there are strong multicollinearity or other numerical problems.

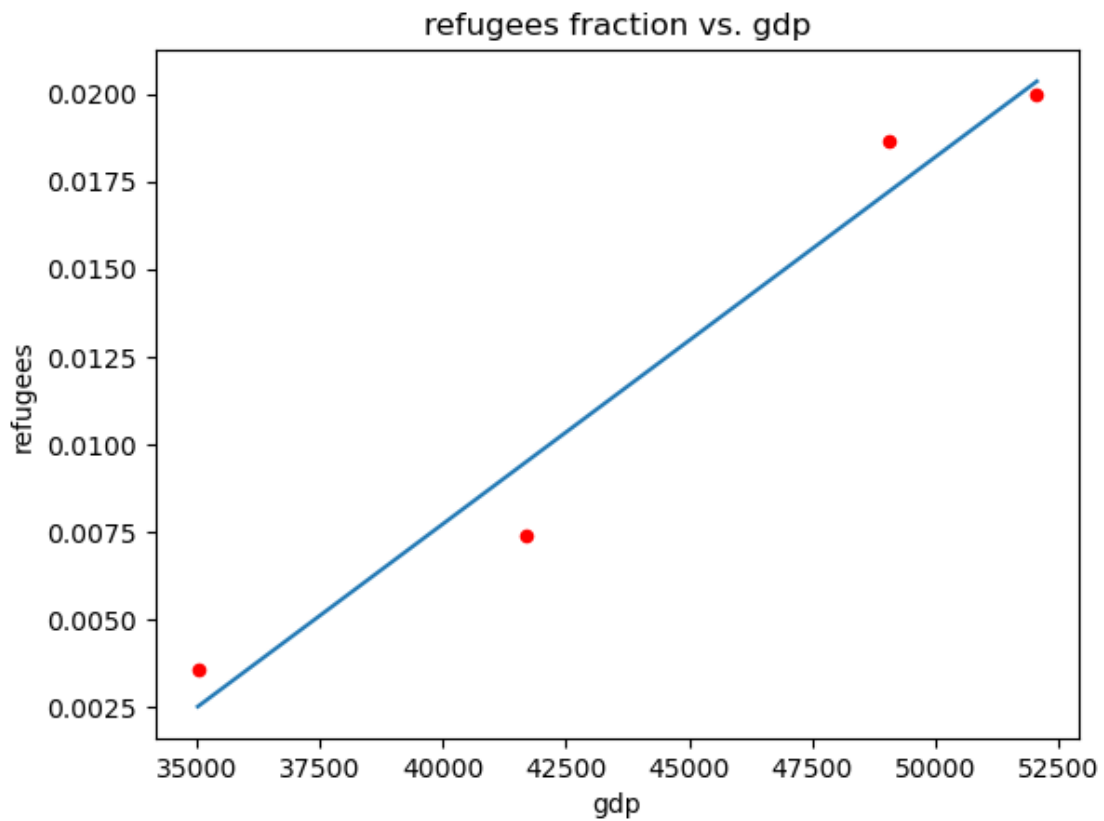
```
/usr/lib/python3/dist-packages/statsmodels/stats/stattools.py:74: ValueWarning:
omni_normtest is not valid with less than 8 observations; 4 samples were given.
  warn("omni_normtest is not valid with less than 8 observations; %i "
```

```
[68]: print(res.params)
```

```
Intercept    -0.034169
gdp           0.000001
dtype: float64
```

```
[69]: ax = data_eu.plot.scatter(y="refugees", x="gdp", color="r")
      ax.plot(x, res.params.gdp*x+res.params.Intercept)
```

```
plt.title("refugees fraction vs. gdp")
plt.show()
```



2.7 Appendix: Selecting from DataFrames

2.7.1 Accessing Rows

Passing a single value to `loc` returns a `Series`

```
[70]: frame.loc["a"]
```

```
[70]: primes    11
      fibo      1
      0-4      0
      Name: a, dtype: int64
```

Passing a list to `loc` returns a `DataFrame` (even if the list contains a single a single value)

```
[71]: frame.loc[["a"]]
```

```
[71]:   primes  fibo  0-4
      a      11     1     0
```

```
[72]: frame.loc[["a","c"]]
```

```
[72]:   primes  fibo  0-4
a      11     1    0
c      17     2    2
```

Also slicing works (but includes the upper boundary)

```
[73]: frame.loc["b":"d"]
```

```
[73]:   primes  fibo  0-4
b      13     1    1
c      17     2    2
d      19     3    3
```

A list of boolean values with n-Rows entries, is considered a mask to select rows

```
[74]: frame.loc[[True,False,True,False,True]]
```

```
[74]:   primes  fibo  0-4
a      11     1    0
c      17     2    2
e      23     5    4
```

Instead of a list, a boolean-series can be used. Rows are matched on the index. (frame[["primes"]] > 20 would not work as this returns a frame instead of a series.)

```
[75]: frame.loc[frame["primes"] > 20]
```

```
[75]:   primes  fibo  0-4
e      23     5    4
```

When using a mask, .loc is optional (but recommended to avoid confusion with columns).

```
[76]: frame[frame["primes"] > 20]
```

```
[76]:   primes  fibo  0-4
e      23     5    4
```

Using iloc it is possible to access rows by position as well. (without using the index)

```
[77]: frame.iloc[2:-1]
```

```
[77]:   primes  fibo  0-4
c      17     2    2
d      19     3    3
```

2.7.2 Accessing Columns

The frame is subscripted directly. Again, passing a singel value returns a series.

```
[78]: frame["primes"]
```

```
[78]: a    11  
      b    13  
      c    17  
      d    19  
      e    23  
      Name: primes, dtype: int64
```

While a list returns a DataFrame

```
[79]: frame[["primes"]]
```

```
[79]:   primes  
      a     11  
      b     13  
      c     17  
      d     19  
      e     23
```

```
[80]: frame[["primes", "0-4"]]
```

```
[80]:   primes  0-4  
      a     11   0  
      b     13   1  
      c     17   2  
      d     19   3  
      e     23   4
```

Instead of subscripting, the `get`-method can be used.

```
[81]: frame.get(["primes", "0-4"])
```

```
[81]:   primes  0-4  
      a     11   0  
      b     13   1  
      c     17   2  
      d     19   3  
      e     23   4
```

For single columns, an attribute with the same name exists

```
[82]: frame.primes
```

```
[82]: a    11  
      b    13  
      c    17  
      d    19  
      e    23
```

Name: primes, dtype: int64

But this fails, if the column-name is not a valid attribute-name

```
[83]: # Raises SyntaxError  
      #frame.0-4
```

For even more options have a look at the pandas-website: <https://pandas.pydata.org/pandas-docs/stable/indexing.html>