Human Development Index

The Human Development Index (HDI) is an index used by the UN to measure the progression of human development around the world. Two of the key aspects they look at are:

1. Life expectancy
2. Gross National Income per capita (adjusted for the price level of the country).

We're going to take a look at some of the data for this and perform some analyses using some more advanced techniques from the `pandas` and `numpy` libraries.

We'll begin by loading in the libraries, and taking a look at the life expectancy data.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

life_expectancy  = pd.read_csv('data/life-expectancy.csv')

life_expectancy.head(5)
```

1. Create a pivot table so that there is one column per year, and only one row per country, with the values in each cell being the life expectancy. The rows should be ordered alphabetically.

Store the answer in a variable called `pivot`.

In [2]:
```
# Add your code below
pivot_table = life_expectancy.pivot(index='Entity', columns='Year', values='Life expe
pivot = pivot_table.sort_index()
pivot
```

2. How many years contain NaNs and how many do not?

On the pivot table, check which entries contain NaNs using `.isnull()`. You can then use `.any()` to get a single boolean value for a whole column (indicating whether *any* value in that column is False). Use `.value_counts()` to get counts for True and False. Assign the output of `.value_counts()` to a variable called `num_nans`.

In [3]:
```
# Add your code below
num_nans = pivot.isnull().any().value_counts()
num_nans
```

3. Using similar logic, return a boolean which indicates whether any row in 2019 contains a NaN.

Store the result in a variable named `nan_in_2019`.

In [4]:
```
# Add your code below
nan_in_2019 = pivot[2019].isnull().any()
nan_in_2019
```

4. Using `.groupby`, show the `mean` life expectancy throughout the world for each year.

Store the resulting dataframe in a variable named `year_vs_life_exp`.

In [5]:
```
# Add your code below
year_vs_life_exp = life_expectancy.groupby(by='Year').mean()
year_vs_life_exp
```

Once you have implemented `year_vs_life_exp`, we can uncomment the cell below and visualise the trend for this data:

In [6]:
```
year_vs_life_exp.reset_index().plot('Year','Life expectancy');
```

5. `melt` the dataframe `pivot` into a table with an index and three columns: Entity, Year, Life expectancy. Assign this to a variable called `melted_pivot`

Hint: you may need to use `.reset_index()` to make 'Entity' indexable again.

In [7]:
```
# Add your code below
melted_pivot = pivot.reset_index().melt(id_vars='Entity', value_name='Life expectancy
melted_pivot
```

Gross National Income per capita

Let's take a look at the GNI data:

In [8]:
```
gni_per_capita = pd.read_csv('data/gross-national-income-per-capita.csv')
gni_per_capita.head(5)
```

The problem is that it doesn't actually tell us population size. There's another dataframe with this data in it:

In [9]:
```
hihd = pd.read_csv('data/hihd-without-gdp-vs-gdp-per-capita.csv')
hihd.head(5)
```

6. Create a new column called 'Population' in `gni_per_capita` which is filled with `np.nan`s.

In [10]:
```
# Add your code below
gni_per_capita['Population'] = np.nan
gni_per_capita
```

Make a list called `popns` which has the correct population value for each row of the dataframe.

To achieve this, you will need to use information from both the `hihd` and the `gni_per_capita` dataframes:

1. First, get a unique list of countries from `gni_per_capita` (specifically, the `Entity` column).
2. Iterate through this list and, on each iteration:
   - Retrieve the `Year` column from `gni_per_capita` for that country. Assign it to a variable called `years`
   - Get the rows containing the total population from the `hihd` dataframe for that country and those `years` (the `Total population (Gapminder, HYDE & UN)` column)
   - If the number of rows for the two above dataframe slices are equal, add the population values to the `popns` list
   - Otherwise, add as many `np.nan`s to the `popns` list as there are items in `years`

The idea here is to add population data where it matches the country and year. However, if we find a mismatch in the number of years of data for any given country, we discard all population data from `hihd` and just add null values (`np.nan`) for all the years of data we have for that country in `gni_per_capita`.

In [11]:
```
# Add your code here
popns = []

countries = gni_per_capita['Entity'].unique()
#countries

for country in countries:
    # extract the years for each country from gni_per_capita
    years = gni_per_capita[gni_per_capita.Entity == country].Year

    # make a filter for hihd so that I can access Entity, Year corresponding to gni_p
    _filter = (hihd.Entity == country) & hihd.Year.isin(years)
    population_by_year = hihd[_filter]['Total population (Gapminder, HYDE & UN)']

    if len(population_by_year) == len(years):
        popns += population_by_year.tolist()
    else:
        popns += [np.nan] * len(years)
```

In [12]:
```
#len(popns), gni_per_capita.shape
```

Now we should be able to fill this `Population` column in properly and inspect the dataframe:

In [13]:
```
gni_per_capita['Population'] = popns
```

In [14]:
```
gni_per_capita
```

7. Let's take a closer look at the year 2011 (since that's where our GNI per capita data comes from).

- Use `gni_per_capita.loc[...]` to sample only the rows from `gni_per_capita` where the `Year` equals 2011. It may be useful to create a mask. Assign this output to a variable called `df`
- Since we actually have only a few countries here, visualising all of them at the same time might look a bit cluttered. To filter down the contents of `df`, use the Python slice operator (`::`) with a step-size of 10 to reduce the number of rows in `df`.
- Next, let's do some cleaning up of our dataframe: use the `.dropna()` and `.reset_index(drop=True)` functions to remove unwanted data and wipe the `df` index (remember to reassign your `df` variable to the results of these functions)
- Note that you should make use of the `.copy()` method when defining `df`. The sample code below provides some guidance as to how this should look.

In [15]:
```
# Add your code here
df = gni_per_capita.loc[gni_per_capita.Year == 2011].copy()

#df[start:stop:step]

df = df[::10]


df = df.dropna().reset_index(drop=True)

df
```

8. Create a new DataFrame as a `.copy()` of `df` and call it `df_life_expectancy`. Add a new column 'Life expectancy' to `df_life_expectancy`, taken from the `life_expectancy` dataframe (from the year 2011).

In [16]:
```
#df.head()
```

In [17]:
```
#life_expectancy.head()
```

In [18]:
```
# Add your code here...
df_life_expectancy = df.copy()

df_life_expectancy = df_life_expectancy.merge(life_expectancy)

df_life_expectancy
```

Now we're going to use the function `plot_blobs` below to visualise the data:

In [19]:
```
# df: the DataFrame to plot
# s: a list of sizes for each blob
def plot_blobs(df, s):
    fig, ax = plt.subplots(figsize=(10,5))

    gni = 'GNI per capita, PPP (constant 2011 international $)'
    df.plot('Life expectancy', gni, kind='scatter',ax=ax,alpha=0.5,s=s) # s = <array>

    for i, (k, v) in enumerate(df.sort_values('Life expectancy').iterrows()):
        if i % 2 == 0: # annotate every other country
            ax.annotate(v['Entity'],(v['Life expectancy'], v[gni]))
```

9. Scale each blob by their population.

Create a list `s` which contains the sizes. Assume `100` is the size for the smallest population, and then scale the others by their ratio to the smallest. For example, if country A is the smallest then their size should be 100. If country B has double the population, their size should be 200.

Don't worry about overlapping text. If you're curious, you can check out the adjustText library for matplotlib - but since this is unrelated to pandas and numpy we'll omit it for now.

In [20]:
```
# Add your code below
s = list(100 * df_life_expectancy['Population'] / df_life_expectancy['Population'].mi
```

The following call should now produced a nicely scaled plot:

In [21]:
```
plot_blobs(df_life_expectancy, s);
```

10. Use `qcut` to create three population size groups ["Small","Medium","Large"].

First create a new Dataframe called `df_category` as a `.copy()` of `df_life_expectancy`. Then assign the output of `qcut` to a new column in `df_category` called `Popn. category`.

In [22]:
```
#df_life_expectancy
```

In [23]:
```
# Add your code below
df_category = df_life_expectancy.copy()
df_category['Popn. category'] = pd.qcut(df_category['Population'], q=3, labels = ["Sma
df_category
```

11. Create a new DataFrame from `df_category` called `df_multi_index` so it has a row MultiIndex over each category.

You should start by creating `df_multi_index` as a `.copy()` of `df_category`

Hint: `pivot` creates a column MultiIndex. `stack` converts a column MultiIndex into a row MultiIndex.

In [24]:
```
# Add your code below
df_multi_index = df_category.copy()
df_multi_index = df_multi_index.pivot(index='Popn. category', columns='Entity').stack
df_multi_index
```

12. Use `.loc` to get a sub-dataframe of countries in the Medium category.

Assign it to a variable called `df_medium`

In [25]:
```
# Add your code here
df_medium = df_multi_index.loc['Medium']
df_medium
```