Pandas-01

Outline

- **Installation of pandas**
  - Importing pandas
  - Importing the dataset
  - Dataframe/Series
- **Basic ops on a DataFrame**
  - df.info()
  - df.head()
  - df.tail()
  - df.shape()
  - df.describe()
- **Basic ops on columns**
  - Different ways of accessing cols
  - Check for Unique values
  - Rename column
  - Deleting col
  - Creating new cols
  - Quiz1 added
- **Basic ops on rows**
  - Implicit/explicit index
  - df.index[]
  - Indexing in series
  - Slicing in series
  - loc/iloc
  - Indexing/Slicing in dataframe
  - Adding a row
  - Check for duplicates
  - Deleting a row
- **Working with both rows and cols**
  - Quiz2 added
- **More in-built ops in pandas**
  - sum()
  - count()
  - mean()
- **Sorting**
  - Quiz3 added
- **Creating series and Dataframes from scratch**

Today's Agenda

- Today's lecture is about **Pandas** library
- We'll see **what** is Pandas
- **Why** we use this library
- We'll also look at some **interesting tasks** we can do **using Pandas**

Installing Pandas

In [ ]:

```
# import sys
# !{sys.executable} -m pip install pandas
```

In [8]:

```
# !pip install pandas
```

Importing Pandas

- You should be able to import Pandas after installing it
- We'll import `pandas` as its **alias name** **`pd`**

Loading [MathJax]/extensions/Safe.js

In [2]:

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

Question: How many of you have set-up pandas now?

Introduction: Why to use Pandas?

How is it different from numpy ?

- The major **limitation of numpy** is that it can only work with 1 datatype at a time
- Most real-world datasets contain a mixture of number (int, float etc) and non-number (string) datatypes.
  - Like **names of places would be string** but their **population would be int**
- So, it is difficult to work with data having heterogeneous values using Numpy

Pandas can work with numbers and strings together

- If our **data has only numbers**, we are better off using **Numpy**
  - It's **lighter** and **easier**
- But if our data has both **number and non-number vals**, it makes sense to use **Pandas**

So lets see how we can use pandas in our applications

Imagine that you are a Data Scientist with McKinsey

- MyKinsey wants to understand the relation between GDP per capita and life expectancy and various trends for their clients.
- The company has acquired data from multiple surveys in different countries in the past
- The survey contains info of several years about:
  - country
  - population size
  - life expectancy
  - GDP per Capita
- Now you have to analyse the data and draw inferences from it that is meaningful to the company

Reading dataset

- Lets first download the dataset
- Link:https://drive.google.com/file/d/1E3bwvYGf1ig32RmcYiWc0IXPN-mD_bI_/view?usp=sharing (https://drive.google.com/file/d/1E3bwvYGf1ig32RmcYiWc0IXPN-mD_bI_/view?usp=sharing)

In [3]:

```
!wget "https://drive.google.com/uc?export=download&id=1E3bwvYGf1ig32RmcYiWc0IXPN-mD_bI_" -O gapminder.csv




```

```
'wget' is not recognized as an internal or external command,
operable program or batch file.
```

Now how should we read this dataset?

- We can do so using pandas
- Pandas makes it very easy to work with these kinds of files
- Pass the file path and name in `pd.read_csv()` function

Loading [MathJax]/extensions/Safe.js

In [4]:

```
df = pd.read_csv('gapminder.csv')
df
```

Out[4]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106
...	...	...	...	...	...	...
1699	Zimbabwe	1987	9216418	Africa	62.351	706.157306
1700	Zimbabwe	1992	10704340	Africa	60.377	693.420786
1701	Zimbabwe	1997	11404948	Africa	46.809	792.449960
1702	Zimbabwe	2002	11926563	Africa	39.989	672.038623
1703	Zimbabwe	2007	12311143	Africa	43.487	469.709298

1704 rows × 6 columns

Question: How many of you were able to import the dataset? Or were there any errors?

- Resolve the issues (if any)

What can we observe from the above dataset ?

- We can see that it has:
  - 6 columns
  - 1704 rows

We have stored the data in `df`

Lets analyse `df` a bit more

What do you think is the datatype of `df` ?

- Lets find it out

In [19]:

```
type(df)
```

Out[19]:

```
pandas.core.frame.DataFrame
```

Its a **pandas DataFrame**

What is a pandas DataFrame ?

- It is a table-like representation of data in Pandas - Structured Data
- Considered as **counterpart of Matrix** in Numpy

Now lets check the data type of df's columns

First we will see how we can access the column 'country' of df

Loading [MathJax]/extensions/Safe.js

In [20]:

```
df["country"]
```

Out[20]:

```
0       Afghanistan
1       Afghanistan
2       Afghanistan
3       Afghanistan
4       Afghanistan
           ...
1699       Zimbabwe
1700       Zimbabwe
1701       Zimbabwe
1702       Zimbabwe
1703       Zimbabwe
Name: country, Length: 1704, dtype: object
```

As you can see we get all the values in the column **country**

Now check its type

In [21]:

```
type(df["country"])
```

Out[21]:

```
pandas.core.series.Series
```

Its a **pandas Series**

Pandas Series

What is a pandas Series ?

- **Series** in Pandas is what a **Vector** is in Numpy

What exactly does that mean?

- It means a Series is or a **single column** of **data**
- **Multiple Series stack together to form a DataFrame**

Now we have understood what Series and DataFrames are

But the dataset is difficult to analyse in this form

What if a dataset has 100 rows ... Or 1000 rows ?

How can we find the datatype, name, total entries in each column ?

- This is where df.info() comes into picture
- It gives a **list of columns** with:
  - **Name/Title** of Columns
  - **How many non-null values (blank cells)** each column has
  - **Type of values** in each column - int, float, string
- **By default**, it shows **data-type as `object` for anything other than int or float**

In [22]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   country     1704 non-null   object
 1   year        1704 non-null   int64
 2   population  1704 non-null   int64
 3   continent   1704 non-null   object
 4   life_exp    1704 non-null   float64
 5   gdp_cap     1704 non-null   float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
```

Now what if we want to see the first 20 rows in the dataset ? How can we do that ?

- Using df.head()
- It gives **specified number** top rows

Loading [MathJax]/extensions/Safe.js

- **Prints top 5 rows by default**

In [150]:

```
df.head()

```

Out[150]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106

We can also **pass in number of rows we want to see** in `head()`

In [23]:

```
df.head(20)
```

Out[23]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106
5	Afghanistan	1977	14880372	Asia	38.438	786.113360
6	Afghanistan	1982	12881816	Asia	39.854	978.011439
7	Afghanistan	1987	13867957	Asia	40.822	852.395945
8	Afghanistan	1992	16317921	Asia	41.674	649.341395
9	Afghanistan	1997	22227415	Asia	41.763	635.341351
10	Afghanistan	2002	25268405	Asia	42.129	726.734055

Similarly what if we want to see the last 20 rows ?

- We can use df.tail() for this purpose
- Its used to see specific number of last rows
- Shows last 5 rows by default

In [152]:

```
df.tail(20)
```

Out[152]:

	country	year	population	continent	life_exp	gdp_cap
1684	Zambia	1972	4506497	Africa	50.107	1773.498265
1685	Zambia	1977	5216550	Africa	51.386	1588.688299
1686	Zambia	1982	6100407	Africa	51.821	1408.678565
1687	Zambia	1987	7272406	Africa	50.821	1213.315116
1688	Zambia	1992	8381163	Africa	46.100	1210.884633
1689	Zambia	1997	9417789	Africa	40.238	1071.353818
1690	Zambia	2002	10595811	Africa	39.193	1071.613938
1691	Zambia	2007	11746035	Africa	42.384	1271.211593
1692	Zimbabwe	1952	3080907	Africa	48.451	406.884115
1693	Zimbabwe	1957	3646340	Africa	50.469	518.764268
1694	Zimbabwe	1962	4277736	Africa	52.358	527.272182
1695	Zimbabwe	1967	4995432	Africa	53.995	569.795071
1696	Zimbabwe	1972	5861135	Africa	55.635	799.362176
1697	Zimbabwe	1977	6642107	Africa	57.674	685.587682
1698	Zimbabwe	1982	7636524	Africa	60.363	788.855041
1699	Zimbabwe	1987	9216418	Africa	62.351	706.157306
1700	Zimbabwe	1992	10704340	Africa	60.377	693.420786
1701	Zimbabwe	1997	11404948	Africa	46.809	792.449960
1702	Zimbabwe	2002	11926563	Africa	39.989	672.038623
1703	Zimbabwe	2007	12311143	Africa	43.487	469.709298

We can also find the shape of dataframe using df.shape()

- Similar to Numpy
- Gives **No. of Rows and Columns -- Dimensions**

In [153]:

```
df.shape
```

Out[153]:

```
(1704, 6)
```

Now we have seen our data

Lets look at some statistics of the data

These stats will help us:

- To understand the patterns in data
- To analyse it in a better way

How can we achieve that ?

- Using df.describe()

What will df.describe() do ?

- Show **statistical summary** of **only columns having numerical values**
  - **count** - How many values does each column has
  - **mean** - average of values in each column
  - **std** - **standard deviation** - measure of **how spread** the data is
  - **min** - **smallest value** in the entire column
  - **max** - **largest value** in the entire column
- It also gives **25th, 50th and 75th percentile** of values in each column
  - If we **sort** the values in a column **in ascending order**
  - **50% gives median** of the values
  - Similarly **25% and 75% give 1/4th and 3/4th percentile**

Loading [MathJax]/extensions/Safe.js

In [5]:

```
df.describe()
```

Out[5]:

	year	population	life_exp	gdp_cap
count	1704.00000	1.704000e+03	1704.000000	1704.000000
mean	1979.50000	2.960121e+07	59.474439	7215.327081
std	17.26533	1.061579e+08	12.917107	9857.454543
min	1952.00000	6.001100e+04	23.599000	241.165876
25%	1965.75000	2.793664e+06	48.198000	1202.060309
50%	1979.50000	7.023596e+06	60.712500	3531.846988
75%	1993.25000	1.958522e+07	70.845500	9325.462346
max	2007.00000	1.318683e+09	82.603000	113523.132900

What can we infer from this info ?

- Avg life expectancy of countries being surveyed is approx 59 yrs
- But its std is approx 13
- This means it varies a lot across diff countries
- A similar inference can also be drawn for GDP per Capita

But this does not give any info about cols with `object` datatype

How can we get info about object datatype columns ? ?

- To print the info of such cols we will have to use the `include` parameter of the function
- It takes list of dtypes as the input
- Lets see how it works

In [8]:

```
df.describe(include = ["object", "int64", "float64"])
```

Out[8]:

	country	year	population	continent	life_exp	gdp_cap
count	1704	1704.00000	1.704000e+03	1704	1704.000000	1704.000000
unique	142	NaN	NaN	5	NaN	NaN
top	Afghanistan	NaN	NaN	Africa	NaN	NaN
freq	12	NaN	NaN	624	NaN	NaN
mean	NaN	1979.50000	2.960121e+07	NaN	59.474439	7215.327081
std	NaN	17.26533	1.061579e+08	NaN	12.917107	9857.454543
min	NaN	1952.00000	6.001100e+04	NaN	23.599000	241.165876
25%	NaN	1965.75000	2.793664e+06	NaN	48.198000	1202.060309
50%	NaN	1979.50000	7.023596e+06	NaN	60.712500	3531.846988
75%	NaN	1993.25000	1.958522e+07	NaN	70.845500	9325.462346
max	NaN	2007.00000	1.318683e+09	NaN	82.603000	113523.132900

Now what can you observe from this ?

- All the column's info has been displayed
- For `object` cols, the information printed is of:
  - count: Total non-null vals in the col
  - unique: Tells no. of unique vals in the col
  - top: Most common val
  - freq: No. of occurences of the most common val

Lets move on now

We now have a basic idea about the dataset

But there is still more to do

To perform a more in-depth analysis we need to process the columns of the dataset

Basic operations on columns [00:34 - 00:44]

We can see that our dataset has 6 cols

Loading [MathJax]/extensions/Safe.js

But what if our dataset has 20 cols ? ... or 100 cols ?

How can we get the names of all these cols ?

- We can use:
  - df.columns
  - df.keys

In [9]:

```
df.columns  # using attribute `columns` of dataframe
```

Out[9]:

```
Index(['country', 'year', 'population', 'continent', 'life_exp', 'gdp_cap'], dtype='object')
```

In [156]:

```
df.keys()  # using method keys() of dataframe
```

Out[156]:

```
Index(['country', 'year', 'population', 'continent', 'life_exp', 'gdp_cap'], dtype='object')
```

This tells that **pandas dataframe treat column names as keys**

Question: In which built-in data-type have we seen keys before?

- **Dictionary**
- Remember in dictionary, **we pass in the key as index** and it **gives the value**
- Same thing happens with pandas dataframe

Pandas DataFrame and Series are **specialised dictionary**

In [157]:

```
df['country'].head()  # Gives values in Top 5 rows pertaining to the key
```

Out[157]:

```
0    Afghanistan
1    Afghanistan
2    Afghanistan
3    Afghanistan
4    Afghanistan
Name: country, dtype: object
```

In [10]:

```
type(df["country"])
```

Out[10]:

```
pandas.core.series.Series
```

But what is so "special" about this dictionary?

- It can take multiple keys
- **Pack the column names (Keys) into a list** and **pass it in as a single index**

In [13]:

```
df[['country', 'life_exp']].head()
```

Out[13]:

	country	life_exp
0	Afghanistan	28.801
1	Afghanistan	30.332
2	Afghanistan	31.997
3	Afghanistan	34.020
4	Afghanistan	36.088

Loading [MathJax]/extensions/Safe.js

Now we want to find the countries that have been surveyed

How can we do that ?

- For this, we need to find the unique vals in the `country` col
- Lets see how we can do that using pandas

In [159]:

```
df['country'].unique()
```

Out[159]:

```
array(['Afghanistan', 'Albania', 'Algeria', 'Angola', 'Argentina',
       'Australia', 'Austria', 'Bahrain', 'Bangladesh', 'Belgium',
       'Benin', 'Bolivia', 'Bosnia and Herzegovina', 'Botswana', 'Brazil',
       'Bulgaria', 'Burkina Faso', 'Burundi', 'Cambodia', 'Cameroon',
       'Canada', 'Central African Republic', 'Chad', 'Chile', 'China',
       'Colombia', 'Comoros', 'Congo, Dem. Rep.', 'Congo, Rep.',
       'Costa Rica', "Cote d'Ivoire", 'Croatia', 'Cuba', 'Czech Republic',
       'Denmark', 'Djibouti', 'Dominican Republic', 'Ecuador', 'Egypt',
       'El Salvador', 'Equatorial Guinea', 'Eritrea', 'Ethiopia',
       'Finland', 'France', 'Gabon', 'Gambia', 'Germany', 'Ghana',
       'Greece', 'Guatemala', 'Guinea', 'Guinea-Bissau', 'Haiti',
       'Honduras', 'Hong Kong, China', 'Hungary', 'Iceland', 'India',
       'Indonesia', 'Iran', 'Iraq', 'Ireland', 'Israel', 'Italy',
       'Jamaica', 'Japan', 'Jordan', 'Kenya', 'Korea, Dem. Rep.',
       'Korea, Rep.', 'Kuwait', 'Lebanon', 'Lesotho', 'Liberia', 'Libya',
       'Madagascar', 'Malawi', 'Malaysia', 'Mali', 'Mauritania',
       'Mauritius', 'Mexico', 'Mongolia', 'Montenegro', 'Morocco',
       'Mozambique', 'Myanmar', 'Namibia', 'Nepal', 'Netherlands',
       'New Zealand', 'Nicaragua', 'Niger', 'Nigeria', 'Norway', 'Oman',
       'Pakistan', 'Panama', 'Paraguay', 'Peru', 'Philippines', 'Poland',
       'Portugal', 'Puerto Rico', 'Reunion', 'Romania', 'Rwanda',
       'Sao Tome and Principe', 'Saudi Arabia', 'Senegal', 'Serbia',
       'Sierra Leone', 'Singapore', 'Slovak Republic', 'Slovenia',
       'Somalia', 'South Africa', 'Spain', 'Sri Lanka', 'Sudan',
       'Swaziland', 'Sweden', 'Switzerland', 'Syria', 'Taiwan',
       'Tanzania', 'Thailand', 'Togo', 'Trinidad and Tobago', 'Tunisia',
       'Turkey', 'Uganda', 'United Kingdom', 'United States', 'Uruguay',
       'Venezuela', 'Vietnam', 'West Bank and Gaza', 'Yemen, Rep.',
       'Zambia', 'Zimbabwe'], dtype=object)
```

In [19]:

```
df['country'].nunique()
```

Out[19]:

```
142
```

In [20]:

```
len(df['country'].unique())
```

Out[20]:

```
142
```

Now if you also want to check for count for each country name appears in df['column'] ?

- That also we can do using value_counts()

In [21]:

```
df['country'].value_counts()
```

Out[21]:

```
Afghanistan          12
Pakistan             12
New Zealand          12
Nicaragua            12
Niger                12
                     ..
Eritrea              12
Equatorial Guinea    12
El Salvador          12
Egypt                12
Zimbabwe             12
Name: country, Length: 142, dtype: int64
```

Loading [MathJax]/extensions/Safe.js

In [23]:

```
df['continent'].value_counts()
```

Out[23]:

```
Africa      624
Asia        396
Europe      360
Americas    300
Oceania      24
Name: continent, dtype: int64
```

So you can see here that we have total 142 unique country names and also the number of times they appeared in the data

And what if we want to change the name of a column ?

- We can do so using df.rename()

In [24]:

```
df.rename({"country": "Country"})
```

Out[24]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106
...	...	...	...	...	...	...
1699	Zimbabwe	1987	9216418	Africa	62.351	706.157306
1700	Zimbabwe	1992	10704340	Africa	60.377	693.420786
1701	Zimbabwe	1997	11404948	Africa	46.809	792.449960
1702	Zimbabwe	2002	11926563	Africa	39.989	672.038623
1703	Zimbabwe	2007	12311143	Africa	43.487	469.709298

1704 rows × 6 columns

In [25]:

```
df.rename({"country": "Country"}, axis = 1)
```

Out[25]:

	Country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106
...	...	...	...	...	...	...
1699	Zimbabwe	1987	9216418	Africa	62.351	706.157306
1700	Zimbabwe	1992	10704340	Africa	60.377	693.420786
1701	Zimbabwe	1997	11404948	Africa	46.809	792.449960
1702	Zimbabwe	2002	11926563	Africa	39.989	672.038623
1703	Zimbabwe	2007	12311143	Africa	43.487	469.709298

1704 rows × 6 columns

Loading [MathJax]/extensions/Safe.js

In [26]:

```
df.rename({'Country':'country'},axis = 1)
```

Out[26]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106
...	...	...	...	...	...	...
1699	Zimbabwe	1987	9216418	Africa	62.351	706.157306
1700	Zimbabwe	1992	10704340	Africa	60.377	693.420786
1701	Zimbabwe	1997	11404948	Africa	46.809	792.449960
1702	Zimbabwe	2002	11926563	Africa	39.989	672.038623
1703	Zimbabwe	2007	12311143	Africa	43.487	469.709298

1704 rows × 6 columns

In [163]:

```
df.head()
```

Out[163]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106

To make it inplace set the `inplace` argument = True

In [27]:

```
df.rename({"country": "Country"}, axis = 1, inplace = True)
df
```

Out[27]:

	Country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106
...	...	...	...	...	...	...
1699	Zimbabwe	1987	9216418	Africa	62.351	706.157306
1700	Zimbabwe	1992	10704340	Africa	60.377	693.420786
1701	Zimbabwe	1997	11404948	Africa	46.809	792.449960
1702	Zimbabwe	2002	11926563	Africa	39.989	672.038623
1703	Zimbabwe	2007	12311143	Africa	43.487	469.709298

1704 rows × 6 columns

Now lets try another way of accessing column vals which is through attribute-style access

Loading [MathJax]/extensions/Safe.js

In [28]:

```
df.Country
```

Out[28]:

```
0       Afghanistan
1       Afghanistan
2       Afghanistan
3       Afghanistan
4       Afghanistan
           ...
1699       Zimbabwe
1700       Zimbabwe
1701       Zimbabwe
1702       Zimbabwe
1703       Zimbabwe
Name: Country, Length: 1704, dtype: object
```

In [166]:

```
df.Country is df["Country"]
```

Out[166]:

```
True
```

This however doesn't work everytime

For example,

- if the column names are not strings
- or if the column names conflict with methods of the DataFrame

It is generally better to avoid this type of accessing columns

Lets change back our column name from `Country` to `country` now

In [29]:

```
df.rename({"Country": "country"}, axis = 1, inplace = True)
df
```

Out[29]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106
...	...	...	...	...	...	...
1699	Zimbabwe	1987	9216418	Africa	62.351	706.157306
1700	Zimbabwe	1992	10704340	Africa	60.377	693.420786
1701	Zimbabwe	1997	11404948	Africa	46.809	792.449960
1702	Zimbabwe	2002	11926563	Africa	39.989	672.038623
1703	Zimbabwe	2007	12311143	Africa	43.487	469.709298

1704 rows × 6 columns

Now that we know which cols exist in our data, are all of them necessary ?

- We already know the continents in which each country lies
- So we don't need it for now
- Lets delete that col

How can we delete cols in pandas dataframe ?

- Remember we loaded our dataset from .csv file in memory and stored it in variable `df` ?
- `df = pd.read_csv('data/gapminder.csv')`
- So, whatever **changes** we make to **`df`** will **NOT affect original data** in the .csv file

Let's see how we can drop or delete entire column from our dataframe

Loading [MathJax]/extensions/Safe.js

In [30]:

```
df.drop('continent')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
C:\Users\SHELEN~1\AppData\Local\Temp/ipykernel_26976/869055503.py in <module>
----> 1 df.drop('continent')

~\anaconda3\lib\site-packages\pandas\util\_decorators.py in wrapper(*args, **kwargs)
    309                     stacklevel=stacklevel,
    310                 )
--> 311             return func(*args, **kwargs)
    312 
    313         return wrapper

~\anaconda3\lib\site-packages\pandas\core\frame.py in drop(self, labels, axis, index, columns, level, inplace, errors)
   4904                 weight  1.0     0.8
   4905         """
-> 4906         return super().drop(
   4907             labels=labels,
   4908             axis=axis,

~\anaconda3\lib\site-packages\pandas\core\generic.py in drop(self, labels, axis, index, columns, level, inplace, error
s)
   4148         for axis, labels in axes.items():
   4149             if labels is not None:
-> 4150                 obj = obj._drop_axis(labels, axis, level=level, errors=errors)
   4151 
   4152         if inplace:

~\anaconda3\lib\site-packages\pandas\core\generic.py in _drop_axis(self, labels, axis, level, errors)
   4183                 new_axis = axis.drop(labels, level=level, errors=errors)
   4184             else:
-> 4185                 new_axis = axis.drop(labels, errors=errors)
   4186             result = self.reindex(**{axis_name: new_axis})
   4187 

~\anaconda3\lib\site-packages\pandas\core\indexes\base.py in drop(self, labels, errors)
   6015         if mask.any():
   6016             if errors != "ignore":
-> 6017                 raise KeyError(f"{labels[mask]} not found in axis")
   6018             indexer = indexer[~mask]
   6019         return self.delete(indexer)

KeyError: "['continent'] not found in axis"
```

Now why did this error happen?

- We did not specify the `axis` along which it should look for

Remember the concept of axis from previous class?

- `axis=0` ---> Rows collapse
- `axis=1` ---> Columns collapse
- By **default**, it takes **`axis=0`**
- Since, we want to **delete a column**, we'll pass **`axis=1`**

Loading [MathJax]/extensions/Safe.js

In [31]:

```
df.drop('continent', axis=1)
```

Out[31]:

	country	year	population	life_exp	gdp_cap
0	Afghanistan	1952	8425333	28.801	779.445314
1	Afghanistan	1957	9240934	30.332	820.853030
2	Afghanistan	1962	10267083	31.997	853.100710
3	Afghanistan	1967	11537966	34.020	836.197138
4	Afghanistan	1972	13079460	36.088	739.981106
...	...	...	...	...	...
1699	Zimbabwe	1987	9216418	62.351	706.157306
1700	Zimbabwe	1992	10704340	60.377	693.420786
1701	Zimbabwe	1997	11404948	46.809	792.449960
1702	Zimbabwe	2002	11926563	39.989	672.038623
1703	Zimbabwe	2007	12311143	43.487	469.709298

1704 rows × 5 columns

- As you can see, **column `contintent` is dropped**

Has the column permanently been deleted from `df` ?

- Let's check

In [32]:

```
df.head()
```

Out[32]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
2	Afghanistan	1962	10267083	Asia	31.997	853.100710
3	Afghanistan	1967	11537966	Asia	34.020	836.197138
4	Afghanistan	1972	13079460	Asia	36.088	739.981106

- NO, the **column `continent` is still there**

Do you see what's happening here?

- We only got a **view of dataframe with column `continent` dropped**
- If we want to **permanently drop the column** from `df`
- We can either **re-assign** it

```
df = df.drop('continent', axis=1 )

    OR
```

- We can **set parameter `inplace=True`**
- (By **default, `inplace=False`**)

In [34]:

```
df.drop('continent', axis=1, inplace=True)
```

Loading [MathJax]/extensions/Safe.js

In [35]:

```
df.head()
```

Out[35]:

	country	year	population	life_exp	gdp_cap
0	Afghanistan	1952	8425333	28.801	779.445314
1	Afghanistan	1957	9240934	30.332	820.853030
2	Afghanistan	1962	10267083	31.997	853.100710
3	Afghanistan	1967	11537966	34.020	836.197138
4	Afghanistan	1972	13079460	36.088	739.981106

- Now, we can see the column `continent` is permanently dropped

Adding new columns in DataFrame

And what if we want to create a new column in the dataframe ?

- Lets see how we can do that
- We can either **use values from existing columns** OR **create our own values**

Using values from existing columns

In [36]:

```
df["New"] = df["life_exp"] + df["year"]
df
```

Out[36]:

	country	year	population	life_exp	gdp_cap	New
0	Afghanistan	1952	8425333	28.801	779.445314	1980.801
1	Afghanistan	1957	9240934	30.332	820.853030	1987.332
2	Afghanistan	1962	10267083	31.997	853.100710	1993.997
3	Afghanistan	1967	11537966	34.020	836.197138	2001.020
4	Afghanistan	1972	13079460	36.088	739.981106	2008.088
...	...	...	...	...	...	...
1699	Zimbabwe	1987	9216418	62.351	706.157306	2049.351
1700	Zimbabwe	1992	10704340	60.377	693.420786	2052.377
1701	Zimbabwe	1997	11404948	46.809	792.449960	2043.809
1702	Zimbabwe	2002	11926563	39.989	672.038623	2041.989
1703	Zimbabwe	2007	12311143	43.487	469.709298	2050.487

1704 rows × 6 columns

As you can see

- An **additional column** has been **created**
- **Values** in this column are **sum of respective values in Column `year` and `population`**

We can use any other operation as well b/w values of existing columns

- Like, Subtraction, Multiplication, etc.

Loading [MathJax]/extensions/Safe.js

In [42]:

```
df["Sub"] = df["life_exp"] - df["year"]
df
```

Out[42]:

	country	year	population	life_exp	gdp_cap	New	Own	Sub
0	Afghanistan	1952	8425333	28.801	779.445314	1980.801	0	-1923.199
1	Afghanistan	1957	9240934	30.332	820.853030	1987.332	1	-1926.668
2	Afghanistan	1962	10267083	31.997	853.100710	1993.997	2	-1930.003
3	Afghanistan	1967	11537966	34.020	836.197138	2001.020	3	-1932.980
4	Afghanistan	1972	13079460	36.088	739.981106	2008.088	4	-1935.912
...	...	...	...	...	...	...	...	...
1699	Zimbabwe	1987	9216418	62.351	706.157306	2049.351	1699	-1924.649
1700	Zimbabwe	1992	10704340	60.377	693.420786	2052.377	1700	-1931.623
1701	Zimbabwe	1997	11404948	46.809	792.449960	2043.809	1701	-1950.191
1702	Zimbabwe	2002	11926563	39.989	672.038623	2041.989	1702	-1962.011
1703	Zimbabwe	2007	12311143	43.487	469.709298	2050.487	1703	-1963.513

1704 rows × 8 columns

Creating own values for new column

- We can **create a list**

OR

- We can **create a Pandas Series** for our new column

OR

- We can **create a Numpy Array and convert it into Pandas Series**

Let's look at that

In [39]:

```
df["Own"] = [i for i in range(1704)]  # count of these values should be correct
df
```

Out[39]:

	country	year	population	life_exp	gdp_cap	New	Own
0	Afghanistan	1952	8425333	28.801	779.445314	1980.801	0
1	Afghanistan	1957	9240934	30.332	820.853030	1987.332	1
2	Afghanistan	1962	10267083	31.997	853.100710	1993.997	2
3	Afghanistan	1967	11537966	34.020	836.197138	2001.020	3
4	Afghanistan	1972	13079460	36.088	739.981106	2008.088	4
...	...	...	...	...	...	...	...
1699	Zimbabwe	1987	9216418	62.351	706.157306	2049.351	1699
1700	Zimbabwe	1992	10704340	60.377	693.420786	2052.377	1700
1701	Zimbabwe	1997	11404948	46.809	792.449960	2043.809	1701
1702	Zimbabwe	2002	11926563	39.989	672.038623	2041.989	1702
1703	Zimbabwe	2007	12311143	43.487	469.709298	2050.487	1703

1704 rows × 7 columns

Now that we know how to create new cols lets see some basic ops on rows

Before that lets drop the newly created cols from df

Loading [MathJax]/extensions/Safe.js

In [43]:

```
df.drop(columns=["New", "Own", "Sub"], axis = 1, inplace = True)
df
```

Out[43]:

	country	year	population	life_exp	gdp_cap
0	Afghanistan	1952	8425333	28.801	779.445314
1	Afghanistan	1957	9240934	30.332	820.853030
2	Afghanistan	1962	10267083	31.997	853.100710
3	Afghanistan	1967	11537966	34.020	836.197138
4	Afghanistan	1972	13079460	36.088	739.981106
...	...	...	...	...	...
1699	Zimbabwe	1987	9216418	62.351	706.157306
1700	Zimbabwe	1992	10704340	60.377	693.420786
1701	Zimbabwe	1997	11404948	46.809	792.449960
1702	Zimbabwe	2002	11926563	39.989	672.038623
1703	Zimbabwe	2007	12311143	43.487	469.709298

1704 rows × 5 columns

Quiz1 :

1. To delete a column, the parameter axis of function drop( ) is assigned the value ___ ?

a. 0

b. 1

c. 2

Answer: 1

2.For a given dataframe

[(https://imgur.com/dM1sGNr)](https://imgur.com/dM1sGNr)

Python will store data for ‘x’ column as

a. float b. int c. string d. object

Answer: object

Working with Rows

Now what if we want to access the 6th row ? Or what if we want to access 6th:15th row ?

How can we do that ?

We will first check these ops for series and generalise to a dataframe

In [46]:

```
ser = df["country"]
ser
```

Out[46]:

```
0       Afghanistan
1       Afghanistan
2       Afghanistan
3       Afghanistan
4       Afghanistan
           ...
1699       Zimbabwe
1700       Zimbabwe
1701       Zimbabwe
1702       Zimbabwe
1703       Zimbabwe
Name: country, Length: 1704, dtype: object
```

How to access a row ?

To access a row in a Series we can use its indices much like we do in a np array

Loading [MathJax]/extensions/Safe.js For e.g. If we want to access the second row (with index 6) the code will be:

In [47]:

```
ser[6]
```

Out[47]:

```
'Afghanistan'
```

And what about accessing the 6th:15th row ?

In [50]:

```
ser[7:20]
```

Out[50]:

```
7     Afghanistan
8     Afghanistan
9     Afghanistan
10    Afghanistan
11    Afghanistan
12        Albania
13        Albania
14        Albania
15        Albania
16        Albania
17        Albania
18        Albania
19        Albania
Name: country, dtype: object
```

This is known as slicing Looks pretty easy

Notice the numbers of row printed alongwith each row

How start indexing with 1 instead of 0 ?

- This is where df.index() method comes into picture
- Takes a series/list/vector of vals having same no. of vals as rows in the df/series
- Lets code this

In [52]:

```
ser.shape

```

Out[52]:

```
(1704,)
```

In [53]:

```
import numpy as np

ser.index = np.arange(1, ser.shape[0]+1, dtype=np.int32, step = 1)
ser
```

Out[53]:

```
1       Afghanistan
2       Afghanistan
3       Afghanistan
4       Afghanistan
5       Afghanistan
           ...
1700       Zimbabwe
1701       Zimbabwe
1702       Zimbabwe
1703       Zimbabwe
1704       Zimbabwe
Name: country, Length: 1704, dtype: object
```

As you can see the indexing is now starting from 1 instead of 0.

Explicit and Implicit Indices

What are these ?

- Index of the row
- These indices are known as **explicit indices**
- Additionally series/dataframes can also use python style indexing
- These are known as **implicit indices**

How can we access explicit index of row though ?

- Using df.index[]
- Takes **impicit index** of row to give its explicit index
- Lets see how it works

In [182]:

```
# ser.index[0]
```

Out[182]:

```
1
```

But why not use just implicit indexing ?

- Explicit indices can be changed to any value of any datatype
  - Eg: Explicit Index of 1st row can be changes to "First"

Now lets go back to indexing and slicing

There is a slight problem in it

Lets look at another dummy series to understand this

In [183]:

```
import pandas as pd
data = pd.Series(['a', 'b', 'c'], index=[1, 5, 3])
data
```

Out[183]:

```
1    a
5    b
3    c
dtype: object
```

In [186]:

```
data[1] # Uses explicit index
```

Out[186]:

```
'a'
```

In [187]:

```
data[1:3] # Uses implicit index
```

Out[187]:

```
5    b
3    c
dtype: object
```

You can also provide index as str

In [188]:

```
data = pd.Series(['a', 'b', 'c'], index=['x', 'y', 'z'])
data
```

Out[188]:

```
x    a
y    b
z    c
dtype: object
```

Pandas supports non-unique index values as well

In [56]:

```
data = pd.Series(['a', 'b', 'c','d'], index=[1, 2, 2,3])
data
```

Out[56]:

```
1    a
2    b
2    c
3    d
dtype: object
```

Loading [MathJax]/extensions/Safe.js

What can we infer from this ?

- **Indexing in Series** used **explicit index**
- **Slicing** however used **implicit index**

This can be a cause for confusion

To avoid this pandas provides special indexers

Lets look at them one by one

1. loc

Allows indexing and slicing that always references the explicit index

In [57]:

```
data.loc[1]
```

Out[57]:

```
'a'
```

In [58]:

```
data.loc[2]
```

Out[58]:

```
2    b
2    c
dtype: object
```

In [61]:

```
data.loc[2:3]
```

Out[61]:

```
2    b
2    c
3    d
dtype: object
```

Did you notice something strange here?

- The **range is inclusive of end point for `loc`**
- **Row with Label 3 is included in the result**

2. iloc

Allows indexing and slicing that always references the implicit Python-style index

In [63]:

```
data.iloc[1]
```

Out[63]:

```
'b'
```

Now will `iloc` also consider the range inclusive?

In [66]:

```
df.year.iloc[0:6]
```

Out[66]:

```
0    1952
1    1957
2    1962
3    1967
4    1972
5    1977
Name: year, dtype: int64
```

- **NO**
- Because **`iloc` works with implicit Python-style indices**

It is important to know about these conceptual differences

- Not just b/w `loc` and `iloc`
- But in general while working in DS and ML

Which one should we use ?

- Generally explicit indexing is considered to be better than implicit one
- But it is recommended to always use both loc and iloc to avoid any confusions

Lets look at Data Selection in DataFrames now

How to access the ith row ?

- Lets say we want to access the 2nd row

We can also use iloc and loc here to access the rows

In [203]:

```
df.head()
```

Out[203]:

	country	year	population	life_exp	gdp_cap
0	Afghanistan	1952	8425333	28.801	779.445314
1	Afghanistan	1957	9240934	30.332	820.853030
2	Afghanistan	1962	10267083	31.997	853.100710
3	Afghanistan	1967	11537966	34.020	836.197138
4	Afghanistan	1972	13079460	36.088	739.981106

In [67]:

```
df.loc[3]  # Row with label 3
```

Out[67]:

```
country       Afghanistan
year                 1967
population       11537966
life_exp            34.02
gdp_cap        836.197138
Name: 3, dtype: object
```

In [205]:

```
df.iloc[3] # Row at position 3
```

Out[205]:

```
country       Afghanistan
year                 1967
population       11537966
life_exp            34.02
gdp_cap        836.197138
Name: 3, dtype: object
```

What if we want to access multiple non-consecutive rows at same time ?

- For eg: rows 1, 10, 100
- We can just **pack the indices in `[]`** and pass it in `loc` or `iloc`

In [69]:

```
df.iloc[[1, 10, 100]]
```

Out[69]:

	country	year	population	life_exp	gdp_cap
1	Afghanistan	1957	9240934	30.332	820.853030
10	Afghanistan	2002	25268405	42.129	726.734055
100	Bangladesh	1972	70759295	45.252	630.233627

Loading [MathJax]/extensions/Safe.js

In [70]:

```
df.loc[[1, 10, 100]]
```

Out[70]:

	country	year	population	life_exp	gdp_cap
1	Afghanistan	1957	9240934	30.332	820.853030
10	Afghanistan	2002	25268405	42.129	726.734055
100	Bangladesh	1972	70759295	45.252	630.233627

What if we pass negative index in `iloc` and `loc` ?

Which one will work?

In [71]:

```
df.iloc[-1]

# Works and gives last row in dataframe
```

Out[71]:

```
country          Zimbabwe
year                 2007
population       12311143
life_exp           43.487
gdp_cap        469.709298
Name: 1703, dtype: object
```

In [72]:

```
df.loc[-1]

# Does NOT work
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\anaconda3\lib\site-packages\pandas\core\indexes\range.py in get_loc(self, key, method, tolerance)
    384                 try:
--> 385                     return self._range.index(new_key)
    386                 except ValueError as err:

ValueError: -1 is not in range

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
C:\Users\SHELEN~1\AppData\Local\Temp/ipykernel_26976/4126833739.py in <module>
----> 1 df.loc[-1]
      2 
      3 # Does NOT work

~\anaconda3\lib\site-packages\pandas\core\indexing.py in __getitem__(self, key)
    929
```

So, why did `iloc[-1]` worked, but `loc[-1]` didn't?

- Because **`iloc` works with positional indices**
- [-1] is the **row at last position**

- **`loc` works with assigned labels**
- There is **no such row with a label of -1**

But What if I want to use one of the columns as row index?

- Using the set_index method
- Here we can make a column an index whose values are not unique

In [75]:

```
temp = df.set_index("country")
temp
```

Out[75]:

	year	population	life_exp	gdp_cap
country				
Afghanistan	1952	8425333	28.801	779.445314
Afghanistan	1957	9240934	30.332	820.853030
Afghanistan	1962	10267083	31.997	853.100710
Afghanistan	1967	11537966	34.020	836.197138
Afghanistan	1972	13079460	36.088	739.981106
...	...	...	...	...
Zimbabwe	1987	9216418	62.351	706.157306
Zimbabwe	1992	10704340	60.377	693.420786
Zimbabwe	1997	11404948	46.809	792.449960
Zimbabwe	2002	11926563	39.989	672.038623
Zimbabwe	2007	12311143	43.487	469.709298

1704 rows × 4 columns

In [77]:

```
#temp["life_exp"]["Afghanistan"]
```

Out[77]:

```
country
Afghanistan    28.801
Afghanistan    30.332
Afghanistan    31.997
Afghanistan    34.020
Afghanistan    36.088
Afghanistan    38.438
Afghanistan    39.854
Afghanistan    40.822
Afghanistan    41.674
Afghanistan    41.763
Afghanistan    42.129
Afghanistan    43.828
Name: life_exp, dtype: float64
```

It is generally a good idea to keep the index val for each row unique

Why is this ?

- Lets see in `temp` what the row corresponding to index `Asia`

In [212]:

```
temp.loc['Afghanistan']
```

Out[212]:

	year	population	life_exp	gdp_cap
country				
Afghanistan	1952	8425333	28.801	779.445314
Afghanistan	1957	9240934	30.332	820.853030
Afghanistan	1962	10267083	31.997	853.100710
Afghanistan	1967	11537966	34.020	836.197138
Afghanistan	1972	13079460	36.088	739.981106
Afghanistan	1977	14880372	38.438	786.113360
Afghanistan	1982	12881816	39.854	978.011439
Afghanistan	1987	13867957	40.822	852.395945
Afghanistan	1992	16317921	41.674	649.341395
Afghanistan	1997	22227415	41.763	635.341351
Afghanistan	2002	25268405	42.129	726.734055
Afghanistan	2007	31889923	43.828	974.580338

Loading [MathJax]/extensions/Safe.js

In [79]:

```
1 temp.loc['Zimbabwe']
```

Out[79]:

	year	population	life_exp	gdp_cap
country				
Zimbabwe	1952	3080907	48.451	406.884115
Zimbabwe	1957	3646340	50.469	518.764268
Zimbabwe	1962	4277736	52.358	527.272182
Zimbabwe	1967	4995432	53.995	569.795071
Zimbabwe	1972	5861135	55.635	799.362176
Zimbabwe	1977	6642107	57.674	685.587682
Zimbabwe	1982	7636524	60.363	788.855041
Zimbabwe	1987	9216418	62.351	706.157306
Zimbabwe	1992	10704340	60.377	693.420786
Zimbabwe	1997	11404948	46.809	792.449960
Zimbabwe	2002	11926563	39.989	672.038623
Zimbabwe	2007	12311143	43.487	469.709298

As you can see we got the rows all having index `Afghanistan`

In [213]:

```
1 df.head()
```

Out[213]:

	country	year	population	life_exp	gdp_cap
0	Afghanistan	1952	8425333	28.801	779.445314
1	Afghanistan	1957	9240934	30.332	820.853030
2	Afghanistan	1962	10267083	31.997	853.100710
3	Afghanistan	1967	11537966	34.020	836.197138
4	Afghanistan	1972	13079460	36.088	739.981106

Adding a row

If you want to add a new row of values to an existing dataframe.

This can be used when we want to insert a new entry in our data that we might have missed adding earlier.

There are different methods to achieve this-

- **Using Append**

  We can use the append() method to append a row to an existing dataframe.

In a dictonary variable named Dict , we can add the values to the columns to be added in the dataframe in key-values pair

Loading [MathJax]/extensions/Safe.js

In [82]:

```
Dict = {'country': 'India', 'year': 2000,'life_exp':37.08,'population':13500000,'gdp_cap':900.23}

df = df.append(Dict, ignore_index = True)

df
```

Out[82]:

	country	year	population	life_exp	gdp_cap
0	Afghanistan	1952	8425333	28.801	779.445314
1	Afghanistan	1957	9240934	30.332	820.853030
2	Afghanistan	1962	10267083	31.997	853.100710
3	Afghanistan	1967	11537966	34.020	836.197138
4	Afghanistan	1972	13079460	36.088	739.981106
...	...	...	...	...	...
1701	Zimbabwe	1997	11404948	46.809	792.449960
1702	Zimbabwe	2002	11926563	39.989	672.038623
1703	Zimbabwe	2007	12311143	43.487	469.709298
1704	India	2000	13500000	37.080	900.230000
1705	India	2000	13500000	37.080	900.230000

1706 rows × 5 columns

- As you can see new row is added to the data i.e now we have 1705 rows
- ignore_index = True Means the index from the series or the source dataframe will be ignored.
- The index available in the target dataframe will be used i.e at 1704 index in the target dataframe.

But Please Note that:

- append() doesn't mutate the the dataframe.

  It does not change the DataFrame, but returns a new DataFrame with the row appended.

- **Using loc:**

We can also add a single row using df.loc

We can add the row at the last in our dataframe.

We can get the number of rows using len(df.index) for determining the position at which we need to add the new row.

In [215]:

```
df.loc[len(df.index)] = ['India',2000 ,13500000,37.08,900.23]
```

In [216]:

```
df
```

Out[216]:

	country	year	population	life_exp	gdp_cap
0	Afghanistan	1952	8425333	28.801	779.445314
1	Afghanistan	1957	9240934	30.332	820.853030
2	Afghanistan	1962	10267083	31.997	853.100710
3	Afghanistan	1967	11537966	34.020	836.197138
4	Afghanistan	1972	13079460	36.088	739.981106
...	...	...	...	...	...
1701	Zimbabwe	1997	11404948	46.809	792.449960
1702	Zimbabwe	2002	11926563	39.989	672.038623
1703	Zimbabwe	2007	12311143	43.487	469.709298
1704	India	2000	13500000	37.080	900.230000
1705	India	2000	13500000	37.080	900.230000

1706 rows × 5 columns

Again the new row but with duplicate data added !

####What you can infer from this ?

Dataframe allow us to feed duplicate rows in the data

- **Using iloc:**

You can use the iLoc[] attribute to add a row at a specific position in the dataframe.

As we know iloc is an integer-based indexing for selecting rows from the dataframe.

You can also use it to assign new rows at that position.

Adding a row at a specific index position will replace the existing row at that position.

In [217]:

```python
df.iloc[len(df.index)] = ['India', 'Asia',2000 ,13500000,37.08,900.23]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-217-b9da360aa3ef> in <module>
----> 1 df.iloc[len(df.index)] = ['India', 'Asia',2000 ,13500000,37.08,900.23]

~\anaconda3\lib\site-packages\pandas\core\indexing.py in __setitem__(self, key, value)
    687             key = com.apply_if_callable(key, self.obj)
    688         indexer = self._get_setitem_indexer(key)
--> 689         self._has_valid_setitem_indexer(key)
    690 
    691         iloc = self if self.name == "iloc" else self.obj.iloc

~\anaconda3\lib\site-packages\pandas\core\indexing.py in _has_valid_setitem_indexer(self, indexer)
   1399             elif is_integer(i):
   1400                 if i >= len(ax):
-> 1401                     raise IndexError("iloc cannot enlarge its target object")
   1402             elif isinstance(i, dict):
   1403                 raise IndexError("iloc cannot enlarge its target object")

IndexError: iloc cannot enlarge its target object
```

####Why we are getting error ?

- When you're using iLoc to add a row, the dataframe must already have a row in the position.
- If a row is not available, you'll see an error IndexError: iloc cannot enlarge its target object.
- iLoc will not expand the size of the dataframe automatically.

Please Note:

- When using the loc[] attribute, it's not mandatory that a row already exists with a specific label.
- It'll automatically extend the dataframe and add a row with that label, unlike the iloc[] method.

Drop Duplicates:

Lets first check for duplicate row:

To take a look at the duplication in the DataFrame , just call the duplicated() method on the DataFrame.

It outputs True if an entire row is identical to a previous row.

In [83]:

```python
df.duplicated()
```

Out[83]:

```
0       False
1       False
2       False
3       False
4       False
        ...  
1701    False
1702    False
1703    False
1704    False
1705     True
Length: 1706, dtype: bool
```

However, it is not practical to see a list of True and False when we need to perform some data analysis.

We can use Pandas loc data selector to extract those duplicate rows:

In [92]:

```
# Extract duplicate rows
df.loc[df.duplicated(),:]
```

Out[92]:

	country	year	population	life_exp	gdp_cap
1705	India	2000	13500000	37.08	900.23

The first argument **df.duplicated()** will find the rows that were identified by **duplicated()**.

The second argument : will display all columns.

Now if you want to remove all **duplicate rows** ?

for that we can use **drop_duplicates()** method that helps in removing duplicates from the data frame.

But the another question is among all duplicate rows which one you want to keep ?

In [93]:

```
df = df.drop_duplicates(keep='first')
df
```

Out[93]:

	country	year	population	life_exp	gdp_cap
0	Afghanistan	1952	8425333	28.801	779.445314
1	Afghanistan	1957	9240934	30.332	820.853030
2	Afghanistan	1962	10267083	31.997	853.100710
3	Afghanistan	1967	11537966	34.020	836.197138
4	Afghanistan	1972	13079460	36.088	739.981106
...	...	...	...	...	...
1700	Zimbabwe	1992	10704340	60.377	693.420786
1701	Zimbabwe	1997	11404948	46.809	792.449960
1702	Zimbabwe	2002	11926563	39.989	672.038623
1703	Zimbabwe	2007	12311143	43.487	469.709298
1704	India	2000	13500000	37.080	900.230000

1705 rows × 5 columns

Here we have argument as **keep**:

This Controls how to consider duplicate value.

It has only three distinct value and default is ‘first’.

- If `first` , This considers first value as unique and rest of the same values as duplicate.
- If `last` , This considers last value as unique and rest of the same values as duplicate.
- If `False` , This considers all of the same values as duplicates.

What if you want to look for duplicacy only for a few columns?

- We can use the argument subset to mention the list of columns which we want to use.
- It's default value is none.
- After passing column name, it will consider that column only for duplicates.

In [221]:

```
print(df.drop_duplicates(subset=['country'],keep='first'))
```

```
                 country  year  population  life_exp      gdp_cap
0            Afghanistan  1952     8425333    28.801   779.445314
12               Albania  1952     1282697    55.230  1601.056136
24               Algeria  1952     9279525    43.077  2449.008185
36                Angola  1952     4232095    30.015  3520.610273
48             Argentina  1952    17876956    62.485  5911.315053
...                  ...   ...         ...       ...          ...
1644             Vietnam  1952    26246839    40.412   605.066492
1656  West Bank and Gaza  1952     1030585    43.160  1515.592329
1668         Yemen, Rep.  1952     4963829    32.548   781.717576
1680              Zambia  1952     2672000    42.038  1147.388831
1692            Zimbabwe  1952     3080907    48.451   406.884115

[142 rows x 5 columns]
```

Deleting a row

Now we know how to access a single/multiple rows in a dataframe

What if we want to delete a row ?

- We can simply use the df.drop() command that we used earlier

But how can we change df.drop() method to drop a row ?

- **Hint**: Take **analogy from deleting a column** we saw earlier
- To drop column, we did:

```
df.drop('continent', axis=1, inplace=True)
```

What will be value of `axis` parameter for deleting a row?

- **`axis=0`**
- OR we can just leave it, because default value of `axis` is `0`

Does `drop()` method uses positional indices or labels?

What do you think by looking at code for deleting column?

- We had to specify column title

- So **`drop()` uses labels**, NOT positional indices

Let's implement this for a row now

In [98]:

```
df.head()
```

Out[98]:

	country	year	population	life_exp	gdp_cap
0	Afghanistan	1952	8425333	28.801	779.445314
1	Afghanistan	1957	9240934	30.332	820.853030
2	Afghanistan	1962	10267083	31.997	853.100710
4	Afghanistan	1972	13079460	36.088	739.981106
5	Afghanistan	1977	14880372	38.438	786.113360

In [99]:

```
# Let's drop row with label
df.drop(4, axis=0, inplace=True)
```

```
C:\Users\Shelendra\anaconda3\lib\site-packages\pandas\core\frame.py:4906: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a
-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-co
py)
  return super().drop(
```

In [224]:

```
df.head()
```

Out[224]:

	country	year	population	life_exp	gdp_cap
0	Afghanistan	1952	8425333	28.801	779.445314
1	Afghanistan	1957	9240934	30.332	820.853030
2	Afghanistan	1962	10267083	31.997	853.100710
4	Afghanistan	1972	13079460	36.088	739.981106
5	Afghanistan	1977	14880372	38.438	786.113360

- Now we see that **row with label 3 is deleted**

- Now we have **rows with labels 0, 1, 2, 4, 5, ...** 3 is not there

- **Labels do NOT change on their own**

Now `df.loc[4]` and `df.iloc[4]` will give different rows

In [225]:

```
df.loc[5]
```

Out[225]:

```
country       Afghanistan
year                 1977
population       14880372
life_exp           38.438
gdp_cap         786.11336
Name: 5, dtype: object
```

In [226]:

```
df.iloc[5]
```

Out[226]:

```
country       Afghanistan
year                 1982
population       12881816
life_exp           39.854
gdp_cap        978.011439
Name: 6, dtype: object
```

Working with Rows and Columns together [00:55 - 01:14]

- We'll use same `loc` and `iloc`
- Pass in **2 different ranges for slicing** - **one for row** and **one for column**

In [101]:

```
df.iloc[1:5, 1:4]

# Gives rows from index 1 to 4 (5 NOT included)

# Gives columns from index 1 to 3 (4 NOT included)
```

Out[101]:

	year	population	life_exp
1	1957	9240934	30.332
2	1962	10267083	31.997
5	1977	14880372	38.438
6	1982	12881816	39.854

Can we do the same thing with `loc` ?

In [102]:

```
df.loc[1:5, 1:4]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
C:\Users\SHELEN~1\AppData\Local\Temp/ipykernel_26976/3462270486.py in <module>
----> 1 df.loc[1:5, 1:4]

~\anaconda3\lib\site-packages\pandas\core\indexing.py in __getitem__(self, key)
    923                 with suppress(KeyError, IndexError):
    924                     return self.obj._get_value(*key, takeable=self._takeable)
--> 925             return self._getitem_tuple(key)
    926         else:
    927             # we by definition only have the 0th axis

~\anaconda3\lib\site-packages\pandas\core\indexing.py in _getitem_tuple(self, tup)
   1107             return self._multi_take(tup)
   1108 
-> 1109         return self._getitem_tuple_same_dim(tup)
   1110 
   1111     def _get_label(self, label, axis: int):

~\anaconda3\lib\site-packages\pandas\core\indexing.py in _getitem_tuple_same_dim(self, tup)
```

Slicing using indices doesn't work with `loc`

- **Column labels are NOT correct**

Loading [MathJax]/extensions/Safe.js

Why?

- Because `loc` **works with labels**
- **Labels for rows are 0, 1, 3, ...**
- **Labels for columns are** `country` **,** `continent` **,** `year` **, ...**
  - NOT 0, 1, 2, 3, ...

In [103]:

```
df.loc[1:5, ['country','life_exp']]

# Row with label 5 will be included

# Columns labels are packed in []
```

Out[103]:

	country	life_exp
1	Afghanistan	30.332
2	Afghanistan	31.997
5	Afghanistan	38.438

We can mention ranges using column labels as well in `loc`

- Column range `'continent':'lifeExp'` works !!

In [230]:

```
df.loc[1:5, 'year':'population']

# Row range 1 to 5 (inclusive)

# Column range 'continent' to 'lifeExp' (inclusive)
```

Out[230]:

	year	population
1	1957	9240934
2	1962	10267083
4	1972	13079460
5	1977	14880372

How can we get specific rows and columns?

- **Pass in those specific indices packed in []**, instead of giving slice ranges

In [231]:

```
df.iloc[[0,10,100], [0,2,3]]
```

Out[231]:

	country	population	life_exp
0	Afghanistan	8425333	28.801
11	Afghanistan	31889923	43.828
101	Bangladesh	80428306	46.923

We can do Step Slicing as well, just like we did in Numpy

In [ ]:

```
#df.iloc[1:10:2]
```

Quiz2:

1. What happens if you pass argument value keep=False in drop_duplicates() as:

```
df= df.drop_duplicates(keep=False)
```

a. it will not remove any duplicate rows

b. it will delete all duplicate rows

c. it will keep only the first row and delete others

Answer: b

2. To display 4 rows from bottom of a dataframe which of the following can be written?

> (i) df.tail(4)

> (ii) df.iloc[-4:]

a. (i) is incorrect, (ii) is correct

b. (i) is correct, (ii) is incorrect

c. both are correct

d. both are incorrect

Ans: both are correct

3. What will be output for the following code?

   import pandas as pd
   s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
   print(s['a'])

   A. 1
   B. 2
   C. 3
   D. 4

   Ans: 1

4. How to select records from 30th to 40th row for the last 3 columns using iloc?

a. df.iloc[29:40,-3:]
b. df.iloc[30:39,-3:]
c. df.iloc[31:41,-3:]
d. df.iloc[29:39,-3:]

Answer: a

In [232]:

```
df.head()
```

Out[232]:

	country	year	population	life_exp	gdp_cap
0	Afghanistan	1952	8425333	28.801	779.445314
1	Afghanistan	1957	9240934	30.332	820.853030
2	Afghanistan	1962	10267083	31.997	853.100710
4	Afghanistan	1972	13079460	36.088	739.981106
5	Afghanistan	1977	14880372	38.438	786.113360

In [233]:

```
df.iloc[29:40,-3:]
```

Out[233]:

	population	life_exp	gdp_cap
30	20033753	61.368	5745.160213
31	23254956	65.799	5681.358539
32	26298373	67.744	5023.216647
33	29072015	69.152	4797.295051
34	31287142	70.994	5288.040382
35	33333216	72.301	6223.367465
36	4232095	30.015	3520.610273
37	4561361	31.999	3827.940465
38	4826015	34.000	4269.276742
39	5247469	35.985	5522.776375
40	5894858	37.928	5473.288005

Let's look at more in-built operations in Pandas [01:15 - 01:17]

Let's store the `'life_exp'` column in a separate variable `le`

Loading [MathJax]/extensions/Safe.js

- `mean()` gives us the mean of values in entire column

In [234]:

```
le = df['life_exp']
le.mean()
```

Out[234]:

```
59.47623514084503
```

We can see more methods by pressing "tab" after `le.`

- `sum()`
- `count()`
- `min()`
- `max()`
- ... and so on

In [235]:

```
# Gives us the sum of values in a column

le.sum()
```

Out[235]:

```
101347.50468000001
```

In [236]:

```
# Gives us the number of values in a column

le.count()
```

Out[236]:

```
1704
```

What will happen we get if we divide `sum()` by `count()` ?

In [237]:

```
le.sum() / le.count()
```

Out[237]:

```
59.47623514084508
```

Sorting

Moving on, if you look at the dataset you would find that `year` col is not sorted

How can we perform sorting in pandas ?

- It's easy!!
- Just **use `df.sort_values()`**

Loading [MathJax]/extensions/Safe.js

In [238]:

```
1 df.sort_values(['year'])
```

Out[238]:

	country	year	population	life_exp	gdp_cap
0	Afghanistan	1952	8425333	28.801	779.445314
552	Gambia	1952	284320	30.000	485.230659
564	Germany	1952	69145952	67.500	7144.114393
576	Ghana	1952	5581001	43.149	911.298937
588	Greece	1952	7733250	65.860	3530.690067
...	...	...	...	...	...
1187	Panama	2007	3242173	75.537	9809.185636
659	Honduras	2007	7483763	70.198	3548.330846
1175	Pakistan	2007	169270617	65.483	2605.947580
1211	Peru	2007	28674757	71.421	7408.905561
275	Chad	2007	10238807	50.651	1704.063724

1704 rows × 5 columns

- Rows get sorted **based on values in `lifeExp` column**
- By **default**, values are sorted in **ascending order**
- If we **set the parameter `ascending=False`** , rows will be sorted in **descending order** of values

Let's try sorting based on `'life_exp'`

In [239]:

```
1 df.sort_values(['life_exp'])
```

Out[239]:

	country	year	population	life_exp	gdp_cap
1292	Rwanda	1992	7290203	23.599	737.068595
0	Afghanistan	1952	8425333	28.801	779.445314
552	Gambia	1952	284320	30.000	485.230659
36	Angola	1952	4232095	30.015	3520.610273
1344	Sierra Leone	1952	2143249	30.331	879.787736
...	...	...	...	...	...
1487	Switzerland	2007	7554661	81.701	37506.419070
695	Iceland	2007	301931	81.757	36180.789190
802	Japan	2002	127065841	82.000	28604.591900
671	Hong Kong, China	2007	6980412	82.208	39724.978670
803	Japan	2007	127467972	82.603	31656.068060

1704 rows × 5 columns

- Now the rows are sorted in **ascending order of year**

Can we do sorting on multiple columns?

- YES, it's possible

Now what will Sorting based on `'year'` and `'lifeExp'` mean?

- It means **rows will first be sorted based on ascending order of `'year'`**
- Then, **rows with same values of `'year'`** will be sorted based on **ascending order of `'lifeExp'`**
- **`'year'` is 1st level** of sorting
- **`'lifeExp'` is 2nd level** of sorting

Loading [MathJax]/extensions/Safe.js

In [240]:

```
1 df.sort_values(['life_exp', 'year'])
```

Out[240]:

	country	year	population	life_exp	gdp_cap
1292	Rwanda	1992	7290203	23.599	737.068595
0	Afghanistan	1952	8425333	28.801	779.445314
552	Gambia	1952	284320	30.000	485.230659
36	Angola	1952	4232095	30.015	3520.610273
1344	Sierra Leone	1952	2143249	30.331	879.787736
...	...	...	...	...	...
1487	Switzerland	2007	7554661	81.701	37506.419070
695	Iceland	2007	301931	81.757	36180.789190
802	Japan	2002	127065841	82.000	28604.591900
671	Hong Kong, China	2007	6980412	82.208	39724.978670
803	Japan	2007	127467972	82.603	31656.068060

1704 rows × 5 columns

Now you can see:

- First rows are sorted in increasing order of `'year'`
- Rows having same `'year'` are sorted in increasing order of `'lifeExp'`

This way, we can do multi-level sorting of our data

We can also have different orders for different columns in multi-level sorting

- **`'year'` in descending** order
- Then **within same values of `'year'`**, we can do **`'lifeExp'` in ascending** order
- Just **pack `True` and `False` for respective columns in a list `[]`**

In [241]:

```
1 df.sort_values(['year', 'life_exp'], ascending=[False, True])
```

Out[241]:

	country	year	population	life_exp	gdp_cap
1463	Swaziland	2007	1133066	39.613	4513.480643
1043	Mozambique	2007	19951656	42.082	823.685621
1691	Zambia	2007	11746035	42.384	1271.211593
1355	Sierra Leone	2007	6144562	42.568	862.540756
887	Lesotho	2007	2012649	42.592	1569.331442
...	...	...	...	...	...
408	Denmark	1952	4334000	70.780	9692.385245
1464	Sweden	1952	7124673	71.860	8527.844662
1080	Netherlands	1952	10381988	72.130	8941.571858
684	Iceland	1952	147962	72.490	7267.688428
1140	Norway	1952	3327728	72.670	10095.421720

1704 rows × 5 columns

Quiz3:

1. How you can calculate mean of two columns population and gdp_cap and store it in another column as "average":

a. df["average"] = df[["population", "gdp_cap"]].mean(axis=1)

b. df["average"] = df[["population", "gdp_cap"]].mean(axis=0)

c. df["average"] = df[["population", "gdp_cap"]].mean()

Ans: a

2. How to sort a pandas data frame in place based on the values of Columns `country` and `population` in desc order?

a. df.sort_values(['country','population'])

b. df.sort_values(['country','population'],inplace=True)

c. df.sort_values(['country','population'],inplace=True, ascending=False)

d. df.sort_values(['country','population'],inplace=True, ascending=True)

Ans: c

Creating dataframes from scratch

Lets now see how to creating a Series and DataFrame from scratch [01:50 - 02:00]

- So far we used an existing dataset

Remember we loaded `gapminder.csv` in the beginning and have been working with it since then?

Remember what was a series in our dataset?

- A **single row** or a **single column**

In [242]:

```
df.loc[0]
```

Out[242]:

```
country       Afghanistan
year                 1952
population        8425333
life_exp           28.801
gdp_cap        779.445314
Name: 0, dtype: object
```

In [243]:

```
df.life_exp # df['lifeExp']
```

Out[243]:

```
0       28.801
1       30.332
2       31.997
4       36.088
5       38.438
         ...
1700    60.377
1701    46.809
1702    39.989
1703    43.487
1704    37.080
Name: life_exp, Length: 1704, dtype: float64
```

- Both these were series

Now we'll see how to create a Series from scratch

- We'll use a **class constructor `Series()`**

In [244]:

```
pd.Series([10, 20, 30]) # We'll pass in a list of values in the constructor
```

Out[244]:

```
0    10
1    20
2    30
dtype: int64
```

How can we create a DataFrame?

- Using **class constructor `DataFrame()`**

Approach 1: Row-oriented

- It takes **2 arguments** - Because DataFrame is **2-dimensional**
  - A **list of rows**
  - A **list of column names/labels**

Loading [MathJax]/extensions/Safe.js

- **Values in each row are packed in a list `[]`**
- **Then all rows are packed in an outside list `[]`** - To **pass a list of rows**

- And a **list of names/labels of columns**

In [245]:

```
pd.DataFrame([[10,20],[30,40]], columns=['A','B'])
```

Out[245]:

	A	B
0	10	20
1	30	40

Let's just add 1 row to see the difference for a better understanding

In [246]:

```
pd.DataFrame([10,20], columns=['A','B'])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\anaconda3\lib\site-packages\pandas\core\internals\managers.py in create_block_manager_from_blocks(blocks, axes)
   1674                 blocks = [
-> 1675                     make_block(
   1676                         values=blocks[0], placement=slice(0, len(axes[0])), ndim=2

~\anaconda3\lib\site-packages\pandas\core\internals\blocks.py in make_block(values, placement, klass, ndim, dtype)
   2741
-> 2742     return klass(values, ndim=ndim, placement=placement)
   2743

~\anaconda3\lib\site-packages\pandas\core\internals\blocks.py in __init__(self, values, placement, ndim)
    141         if self._validate_ndim and self.ndim and len(self.mgr_locs) != len(self.values):
--> 142             raise ValueError(
    143                 f"Wrong number of items passed {len(self.values)}, "

ValueError: Wrong number of items passed 1, placement implies 2

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
<ipython-input-246-0201905ec1c1> in <module>
----> 1 pd.DataFrame([10,20], columns=['A','B'])

~\anaconda3\lib\site-packages\pandas\core\frame.py in __init__(self, data, index, columns, dtype, copy)
    582                     mgr = arrays_to_mgr(arrays, columns, index, columns, dtype=dtype)
    583                 else:
--> 584                     mgr = init_ndarray(data, index, columns, dtype=dtype, copy=copy)
    585             else:
    586                 mgr = init_dict({}, index, columns, dtype=dtype)

~\anaconda3\lib\site-packages\pandas\core\internals\construction.py in init_ndarray(values, index, columns, dtype, cop
y)
    236         block_values = [values]
    237
--> 238     return create_block_manager_from_blocks(block_values, [columns, index])
    239
    240

~\anaconda3\lib\site-packages\pandas\core\internals\managers.py in create_block_manager_from_blocks(blocks, axes)
   1685         blocks = [getattr(b, "values", b) for b in blocks]
   1686         tot_items = sum(b.shape[0] for b in blocks)
-> 1687         raise construction_error(tot_items, blocks[0].shape[1:], axes, e)
   1688
   1689

ValueError: Shape of passed values is (2, 1), indices imply (2, 2)
```

Now Why did this give an error?

- Because we passed in a **list of values**

- `DataFrame()` expects a **list of rows**

- So, we **need to pass `[10,20]` as `[[10,20]]`**

Loading [MathJax]/extensions/Safe.js

In [247]:

```
pd.DataFrame([[10,20]], columns=['A','B'])
```

Out[247]:

	A	B
0	10	20

There's another approach to create a DataFrame

Approach 2: Column-oriented

- We **pass in a dictionary** in `DataFrame()` constructor
- **Key** is the **Column Name/Label**
- **Value** is the **list of values column-wise**

In [248]:

```
pd.DataFrame({'A':[10,30], 'B':[20,40]})
```

Out[248]:

	A	B
0	10	20
1	30	40

Concatenating DataFrames

- We can **join 2 or more DataFrames to form a single DataFrame**
- Let's start by creating 2 DataFrames

In [249]:

```
import pandas as pd
```

In [250]:

```
a = pd.DataFrame({'A':[10,30], 'B':[20,40]})
b = pd.DataFrame({'A':[10,30], 'C':[20,40]})
a
```

Out[250]:

	A	B
0	10	20
1	30	40

In [251]:

```
b
```

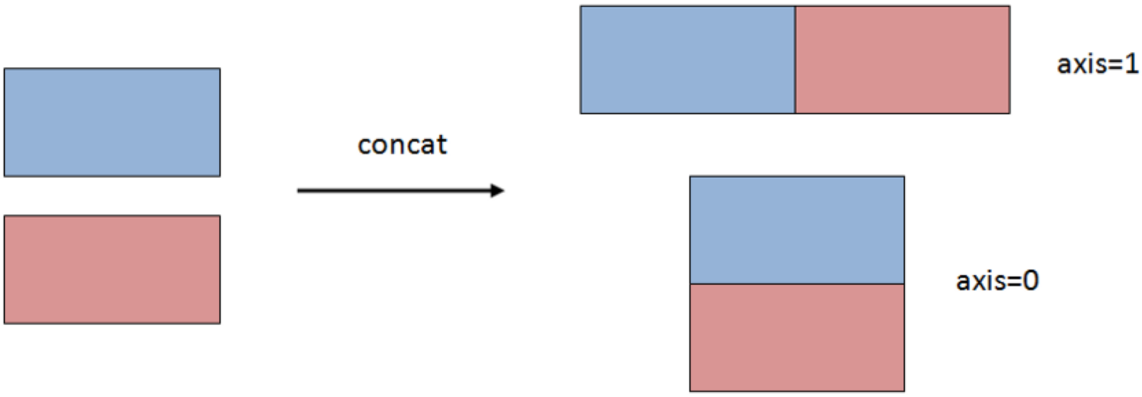
Out[251]:

	A	C
0	10	20
1	30	40

We just use `pd.concat()`

- Pass in a list of DataFrames that we want to combine

Loading [MathJax]/extensions/Safe.js



In [252]:

```
pd.concat([a, b])
```

Out[252]:

	A	B	C
0	10	20.0	NaN
1	30	40.0	NaN
0	10	NaN	20.0
1	30	NaN	40.0

Notice a few things here:

- By **default,** `axis=0` **for concatenation**
- These **means concatenation is done row-wise**
- **Column** `A` in both DataFrames is **combined into a single column**
  - Column **name matching**
- It concatenated in such a way as if
  - **DataFrame** `a` did **NOT have any values in Column** `C`
  - **DataFrame** `b` did **NOT have any values in Column** `B`
- Also the indices of the rows are preserved

In [253]:

```
pd.concat([a, b]).loc[0]
```

Out[253]:

	A	B	C
0	10	20.0	NaN
0	10	NaN	20.0

We obviously want the indices to be unique for each row

How can we do this ?

- By setting `ignore_index = True`

In [254]:

```
pd.concat([a, b], ignore_index = True)
```

Out[254]:

	A	B	C
0	10	20.0	NaN
1	30	40.0	NaN
2	10	NaN	20.0
3	30	NaN	40.0

We can concatenate column-wise as well

What do we need to change to concatenate them column-wise?

- `axis=1`

In [255]:

```
pd.concat([a, b], axis=1)
```

Out[255]:

	A	B	A	C
0	10	20	10	20
1	30	40	30	40

As you can see here:

- **Column `A` is NOT combined as one**
- It gives 2 columns with **different positional index**, but **same label**

We can also create a multi-indexed dataframe by mentioning the keys for each dataframe being concatenateed

In [256]:

```
pd.concat([a, b], keys=["x", "y"])
```

Out[256]:

		A	B	C
x	**0**	10	20.0	NaN
	1	30	40.0	NaN
y	**0**	10	NaN	20.0
	1	30	NaN	40.0

Also By default, the entries for which no data is available are filled with NA values

We can change this behaviour by specifying the type of `join` that should be used to combine data

Which join can we use if we want a union of cols ?

- Outer join
- Set as default by pd.concat

In [257]:

```
pd.concat([a, b], join="outer")
```

Out[257]:

	A	B	C
0	10	20.0	NaN
1	30	40.0	NaN
0	10	NaN	20.0
1	30	NaN	40.0

And what if we want an intersection of cols ?

- We need to use the inner join for that
- There will be no null values in any cell

In [258]:

```
pd.concat([a, b], join="inner")
```

Out[258]:

	A
0	10
1	30
0	10
1	30

There also exists a shorter method of appending 1 dataframe to the other

This is through the `append()` method

Loading [MathJax]/extensions/Safe.js

Concatentaion takes place only through axis = 0

In [259]:

```
a.append(b, ignore_index = False)
```

Out[259]:

	A	B	C
0	10	20.0	NaN
1	30	40.0	NaN
0	10	NaN	20.0
1	30	NaN	40.0

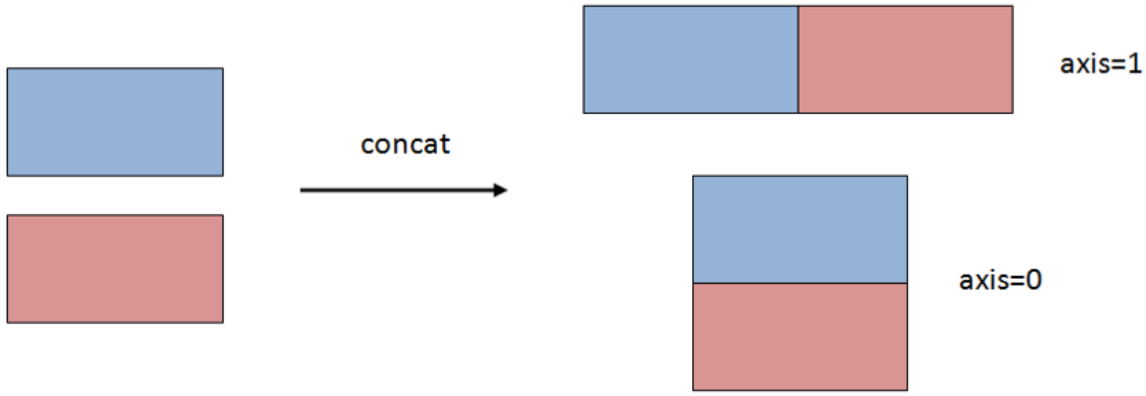
Note:

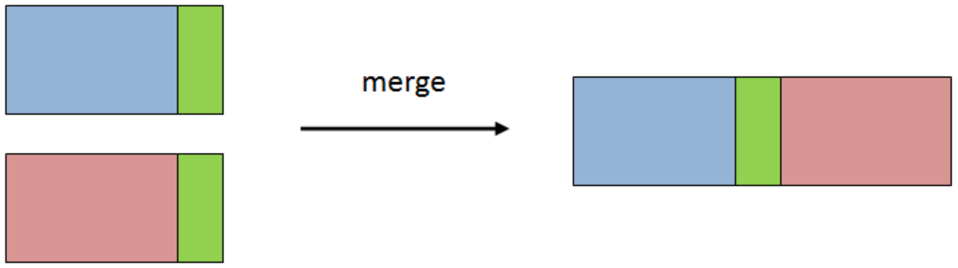
- The append() method does not modify the orginial object
- It creates a new one with combined data
- Hence, it is not a very efficient method

So far we have only concatenated and not merged data

Bur whats the difference between concat and merge ?

- `concat`
  - simply stacks multiple DataFrame together along an axis
- `merge`
  - combines dataframes side-by-side based on values in shared columns





Lets explore merging in more detail

- This **works like join in SQL**
- Lets see what this means

Let's create 2 DataFrames

1. `users` --> **Stores the user details** - **IDs** and **Names of users**

Loading [MathJax]/extensions/Safe.js

In [3]:

```
users = pd.DataFrame({'userid':[1, 2, 3], 'name':['A', 'B', 'C']})
users
```

Out[3]:

	userid	name
0	1	A
1	2	B
2	3	C

2. `msgs` --> **Stores the messages** users have sent - **User IDs** and **messages**

In [4]:

```
msgs = pd.DataFrame({'userid':[1, 1, 2], 'msg':['hello', 'bye', 'hi']})
msgs
```

Out[4]:

	userid	msg
0	1	hello
1	1	bye
2	2	hi

Now suppose you want to know the name of the person who sent a message

How can we do that ?

- We need to create a new dataframe
- It will take data from both `msgs` and `users`

So should can we use pd.concat() for this ?

- No
- pd.concat() does not work according to the values in the columns

How can we do this then ?

- Using pd.merge()

How does it work ?

- Uses cols with same name as keys
- Merges dataframes using these keys
- We can specify the cols to use as keys
- This is done through `on` parameter

In [ ]:

```
users.merge(msgs, on="userid")
```

We can see user ids, user names and the messages they sent together

But sometimes the column names might be different even if they contain the same data

For eg:

- Dataframe 1: col for employees name might be `name`
- Dataframe 2: col for employees name might be `employee`

How can we merge the 2 dataframes in this situation ?

- Using the `left_on` and `right_on` keywords
- `left_on` : Specifies the key of the 1st dataframe
- `right_on` : Specifies the key of the 2nd dataframe

Lets see how it works

Loading [MathJax]/extensions/Safe.js

In [ ]:

```
users.rename(columns = {"userid": "id"}, inplace = True)
users.merge(msgs, left_on="id", right_on="userid") # this is inner join

# Notice that left_on is column from users
# right_on is column from msgs
```

In above codes we have skipped one 1 important part

Specifying type of joins to merge the dataframes

Where does it become relevant ?

- Notice that `users` has a userid = 3 but `msgs` does not
- When we merge these dataframes the userid = 3 is not included
- Only the userid common in both dataframes is shown

- **What if we want to change this behaviour ?**
  - This is where joins can be used

There are different types of joins

Lets say we want to find msg text of people only in the `users` table. Which join can we use for that ?

- Inner join
- It takes intersection of values in key cols
- Set by default in pd.merge()
- Lets code it now

In [ ]:

```
users.merge(msgs, how = "inner", left_on = "id", right_on = "userid")
```

Now lets say we want 1 dataframe having all info of all the users. How can we do that ?

- Using `outer` join
- It returns a join over the union of the input columns
- Replaces all missing values with `Na`

In [ ]:

```
users.merge(msgs, how = "outer", left_on = "id", right_on = "userid")
```

And what if we want vals in key col of left dataframe ?

- We can use `left` join for that

In [ ]:

```
users.merge(msgs, how = "left", left_on = "id", right_on = "userid")
```

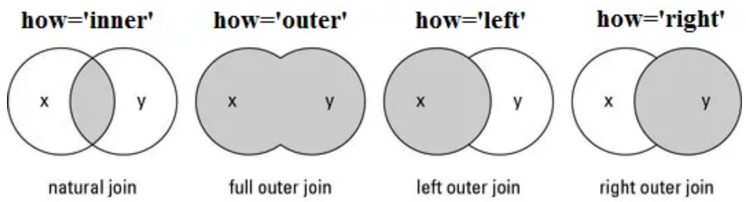
Similarly, what if we want vals in key cols of only right dataframe ?

- Returns join over cols of right input

In [ ]:

```
users.merge(msgs, how = "right", left_on = "id", right_on = "userid")
```

Lets visualise these joins using a venn diagram



Question:

In given code dataframe df has **_ rows and _** columns.

```
import pandas as pd
S1 = pd.Series([1, 2, 3, 4], index = ['a', 'b','c','d'])
S2 = pd.Series([10, 20, 30, 40], index = ['a', 'bb','c','dd'])
df = pd.DataFrame([S1,S2])
```

In [ ]:

```
import pandas as pd
S1 = pd.Series([1, 2, 3, 4], index = ['a', 'b','c','d'])
S2 = pd.Series([10, 20, 30, 40], index = ['a', 'bb','c','dd'])
df = pd.DataFrame([S1,S2])
df.shape
```

Quiz4:

1. What we are doing in the following statement?

```
dF1=dF1.append(dF2) #dF1 and dF2 are DataFrame object
```

a. We are appending dF1 in dF2

b. We are appending dF2 in dF1

c. We are creating Series from DataFrame

Answer: We are appending dF2 in dF1

2. For the concat(), if the axis=1, it will join the dataframes

a. vertically

b. horizontally

Ans: horizontally

This was all about Pandas for today

Pandas is also a very vast library

- You can explore other methods for performing different tasks on your own
- We'll cover a few more important concepts in the next lecture
- We'll also do some practice questions using Pandas

Final Q&A for Today's Lecture

Extra Material : Shivank

Point 1

1M rows and ~300 features, taking up a whopping 2.2GB of disk space.

%time

tps_october = pd.read_csv("data/train.csv") Wall time: 21.8 s

better handling with datatable or dask

import datatable as dt # pip install datatble

%%time

tps_dt_october = dt.fread("data/train.csv").to_pandas()

Wall time: 2 s

Point 2

if col_type in numerics: c_min = df[col].min() c_max = df[col].max() if str(col_type)[:3] == "int": if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8).max: df[col] = df[col].astype(np.int8)

saving from 3 GB to 510 MB

Change the dtypes (int64 -> int32)

df[['col_1','col_2', 'col_3', 'col_4', 'col_5']] = df[['col_1','col_2', 'col_3', 'col_4', 'col_5']].astype('int32')

Change the dtypes (float64 -> float32)

df[['col_6', 'col_7', 'col_8', 'col_9', 'col_10']] = df[['col_6', 'col_7', 'col_8', 'col_9', 'col_10']].astype('float32')

checking memory usage

df[col1].memory_usage()

categorical type

it returns objects but if output only have Y or N using value_counts() astype('category')

In [ ]:

```
# https://pandas.pydata.org/docs/user_guide/style.html
df = pd.read_csv('gapminder.csv')
df.describe().T.style.bar(subset=["mean"], color="#205ff2").background_gradient(subset=["std"], cmap="Reds").background_gradient(
    subset=["50%"], cmap="coolwarm"
)
```

In [ ]:

```
import pandas as pd
%timeit a = pd.read_csv("test.csv")
```

In [ ]:

```
a.describe().T.style.bar(subset=["mean"], color="#205ff2").background_gradient(subset=["std"], cmap="Reds").background_gradient(
    subset=["50%"], cmap="coolwarm"
)
```

In [ ]:

```
!pip install dask
```

In [ ]:

```
import dask.dataframe as dd
%timeit ddf = dd.read_csv("test.csv").compute()
```

In [ ]:

```
type(ddf)
```

In [ ]:

```

```

In [ ]:

```
ddf.head()
```

In [ ]:

```

```

Pont 4 : Saving to csv

%time

tps_october.to_csv("data/copy.csv")

Wall time: 2min 43s

better to save to parquet

%time

tps_october.to_parquet("data/copy.parquet")

Wall time: 7.84 s

reading chunkwise

incremental_dataframe = pd.read_csv("train.csv", chunksize=100000) # Number of lines to rea

In [262]:

```
import pandas as pd
df = pd.read_csv('trip.csv')
```

In [263]:

```
chunk_size=50000
batch_no=1
for chunk in pd.read_csv('trip.csv',chunksize=chunk_size):
    chunk.to_csv('chunk'+str(batch_no)+'.csv',index=False)
    batch_no+=1
```

In [264]:

```
df1 = pd.read_csv('chunk1.csv')
df1.head()
```

Out[264]:

	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	pickup_longitude	pickup_latitude	RateCodeID	store_and_f
0	2	2015-01-15 19:05:39	2015-01-15 19:23:42	1	1.59	-73.993896	40.750111	1	
1	1	2015-01-10 20:33:38	2015-01-10 20:53:28	1	3.30	-74.001648	40.724243	1	
2	1	2015-01-10 20:33:38	2015-01-10 20:43:41	1	1.80	-73.963341	40.802788	1	
3	1	2015-01-10 20:33:39	2015-01-10 20:35:31	1	0.50	-74.009087	40.713818	1	
4	1	2015-01-10 20:33:39	2015-01-10 20:52:58	1	3.00	-73.971176	40.762428	1	

In [5]:

```
import pandas as pd
import numpy as np
df = pd.DataFrame({
  "Accessories": ["Laptop", "Laptop", "Ipad", "Ipad", "Tablet", "Laptop"],
  "customer": ["Andrew", "Andrew", "Tom", "Andrew", "Tobey", "Peter"],
  "quantity": [1, 2, 2, 3, 1, 2],
})
df
```

Out[5]:

	Accessories	customer	quantity
0	Laptop	Andrew	1
1	Laptop	Andrew	2
2	Ipad	Tom	2
3	Ipad	Andrew	3
4	Tablet	Tobey	1
5	Laptop	Peter	2

In [6]:

```
df.pivot_table(index="Accessories", columns="customer", values="quantity", aggfunc=np.sum)
```

Out[6]:

customer	Andrew	Peter	Tobey	Tom
Accessories				
Ipad	3.0	NaN	NaN	2.0
Laptop	3.0	2.0	NaN	NaN
Tablet	NaN	NaN	1.0	NaN

Loading [MathJax]/extensions/Safe.js

In [7]:

```
df.groupby(['Accessories', 'customer']).quantity.sum().unstack()
```

Out[7]:

customer	Andrew	Peter	Tobey	Tom
Accessories				
Ipad	3.0	NaN	NaN	2.0
Laptop	3.0	2.0	NaN	NaN
Tablet	NaN	NaN	1.0	NaN

In [9]:

```
df.pivot_table(index="Accessories", columns="customer", values="quantity", aggfunc=np.sum, fill_value=0)
```

Out[9]:

customer	Andrew	Peter	Tobey	Tom
Accessories				
Ipad	3	0	0	2
Laptop	3	2	0	0
Tablet	0	0	1	0

Loading [MathJax]/extensions/Safe.js