**Pre-lecture Content**

Download datasets:   
<https://drive.google.com/drive/folders/1mS7IM7kwohmGHuFMol3VL-oqF16b_Tv3?usp=sharing>

Or

<https://drive.google.com/drive/folders/1p0mhctIjBiDKT-UGw9EatscSJdNVihfi?usp=share_link>

# Pandas-1

## Outline

* **Installation of pandas**
  + Importing pandas
  + Importing the dataset
  + Dataframe/Series
* **Basic ops on a DataFrame**
  + df.tail()
  + df.shape()
  + [df.info](http://df.info/)()
  + df.describe()
* **Basic ops on columns**
  + Different ways of accessing cols
  + Deleting col
  + Creating new cols
* **Basic ops on rows**
  + Implicit/explicit index
  + df.index[]
  + Indexing in series
  + Slicing in series
  + loc/iloc
  + Indexing/Slicing in dataframe
  + Deleting a row
* **Working with both rows and cols**
* **More in-built ops in pandas**
  + sum()
  + count()
  + mean()
  + Numpy Ufuncs on pandas
* **Sorting**
* **Creating series and Dataframes from scratch**
* **Concatenation**
  + pd.concat()
    - axis for concat
    - Inner join
    - Outer join
* **Merging dataframes**
  + Concat v/s Merge
  + Outer join
  + Left join
  + Inner join

## Installing Pandas

import sys

!{sys.executable} -m pip install pandas

!pip install pandas

## Importing Pandas

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

import pandas as pd

import numpy as np

## 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**

## Reading dataset

* Lets first download the dataset
* Link: <https://drive.google.com/file/d/1fOeDQuMaV0Mp5eFHPxlrRX6lgq-23z35/view?usp=sharing>

!wget "https://drive.google.com/uc?export=download&id=1fOeDQuMaV0Mp5eFHPxlrRX6lgq-23z35" -O gapminder.csv

gapminder.csv 100%[===================>] 81.82K --.-KB/s in 0.001s

#### Now how should we read this dataset?

* Pass the file path and name in pd.read\_csv() function

df = pd.read\_csv('gapminder.csv')

df

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

1704 rows × 6 columns

#### 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

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

* Lets find it out

type(df)

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

df["country"]

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 lets check the data type of df’s columns

type(df["country"])

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**

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

* This is where [df.info](http://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**

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 continent 1704 non-null object

2 year 1704 non-null int64

3 lifeExp 1704 non-null float64

4 population 1704 non-null int64

5 gdpPerCap 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**
* **Prints top 5 rows by default**

df.head()

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

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

df.head(20)

|  | **country** | **continent** | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | Afghanistan | Asia | 1952 | 28.801 | 8425333 | 779.445314 |
| **1** | Afghanistan | Asia | 1957 | 30.332 | 9240934 | 820.853030 |
| **2** | Afghanistan | Asia | 1962 | 31.997 | 10267083 | 853.100710 |
| **3** | Afghanistan | Asia | 1967 | 34.020 | 11537966 | 836.197138 |
| **4** | Afghanistan | Asia | 1972 | 36.088 | 13079460 | 739.981106 |
| **5** | Afghanistan | Asia | 1977 | 38.438 | 14880372 | 786.113360 |
| **6** | Afghanistan | Asia | 1982 | 39.854 | 12881816 | 978.011439 |
| **7** | Afghanistan | Asia | 1987 | 40.822 | 13867957 | 852.395945 |
| **8** | Afghanistan | Asia | 1992 | 41.674 | 16317921 | 649.341395 |
| **9** | Afghanistan | Asia | 1997 | 41.763 | 22227415 | 635.341351 |
| **10** | Afghanistan | Asia | 2002 | 42.129 | 25268405 | 726.734055 |
| **11** | Afghanistan | Asia | 2007 | 43.828 | 31889923 | 974.580338 |
| **12** | Albania | Europe | 1952 | 55.230 | 1282697 | 1601.056136 |
| **13** | Albania | Europe | 1957 | 59.280 | 1476505 | 1942.284244 |
| **14** | Albania | Europe | 1962 | 64.820 | 1728137 | 2312.888958 |
| **15** | Albania | Europe | 1967 | 66.220 | 1984060 | 2760.196931 |
| **16** | Albania | Europe | 1972 | 67.690 | 2263554 | 3313.422188 |
| **17** | Albania | Europe | 1977 | 68.930 | 2509048 | 3533.003910 |
| **18** | Albania | Europe | 1982 | 70.420 | 2780097 | 3630.880722 |
| **19** | Albania | Europe | 1987 | 72.000 | 3075321 | 3738.932735 |

#### 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

df.tail(20)

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

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

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

df.shape

(1704, 6)

#### Lets look at some statistics of the data. How can we do 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**

df.describe()

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

#### What can we do for cols with object datatype?

* 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

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

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

## Basic operations on columns

#### How can we get the names of all these cols ?

* We can use:
  + df.columns
  + df.keys

df.columns # using attribute `columns` of dataframe

Index(['country', 'continent', 'year', 'lifeExp', 'population', 'gdpPerCap'], dtype='object')

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

Index(['country', 'continent', 'year', 'lifeExp', 'population', 'gdpPerCap'], dtype='object')

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

#### 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**

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

0 Afghanistan

1 Afghanistan

2 Afghanistan

3 Afghanistan

4 Afghanistan

Name: country, dtype: object

#### 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**

df[['country', 'lifeExp']].head()

|  | **country** | **lifeExp** |
| --- | --- | --- |
| **0** | Afghanistan | 28.801 |
| **1** | Afghanistan | 30.332 |
| **2** | Afghanistan | 31.997 |
| **3** | Afghanistan | 34.020 |
| **4** | Afghanistan | 36.088 |

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

* We can do so using df.rename()

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

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

1704 rows × 6 columns

To make it inplace set the inplace argument = True

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

df

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

1704 rows × 6 columns

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

df.Country

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

df.Country is df["Country"]

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

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

df

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

1704 rows × 6 columns

#### How can we delete cols in pandas dataframe ?

* Remember we loaded our dataset from .csv file in memory and stored it in variable df?
* 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

df.drop('continent')

---------------------------------------------------------------------------

KeyError Traceback (most recent call last)

<ipython-input-25-ebe0cc5cf629> in <module>()

----> 1 df.drop('continent')

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**

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

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

df.head()

|  | **country** | **continent** | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | Afghanistan | Asia | 1952 | 28.801 | 8425333 | 779.445314 |
| **1** | Afghanistan | Asia | 1957 | 30.332 | 9240934 | 820.853030 |
| **2** | Afghanistan | Asia | 1962 | 31.997 | 10267083 | 853.100710 |
| **3** | Afghanistan | Asia | 1967 | 34.020 | 11537966 | 836.197138 |
| **4** | Afghanistan | Asia | 1972 | 36.088 | 13079460 | 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**)

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

df.head()

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

### Adding new columns in DataFrame

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

* We can either **use values from existing columns** OR **create our own values**

### Using values from existing columns

df["New"] = df["lifeExp"] + df["year"]

df

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

* Like, Subtraction, Multiplication, etc.

df["Sub"] = df["lifeExp"] - df["year"]

df

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

1704 rows × 7 columns

### Creating 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**

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

df

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

1704 rows × 8 columns

Lets drop the newly created cols from df

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

df

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

1704 rows × 5 columns

## Basic ops on Rows

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

ser = df["country"]

ser

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

For eg, if we want to access the seventh row (with index 6) the code will be:

ser[6]

'Afghanistan'

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

ser[6:15]

6 Afghanistan

7 Afghanistan

8 Afghanistan

9 Afghanistan

10 Afghanistan

11 Afghanistan

12 Albania

13 Albania

14 Albania

Name: country, dtype: object

This is known as slicing

### Explicit and Implicit Indices

Notice the numbers of row printed alongwith each row

#### 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**

#### 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”

#### How can we access explicit index of row though ?

* Using df.index[]
* Takes **impicit index** of row to give its explicit index

ser.index[1]

1

We can change the explicit indices to any other value of any other datatype

#### For eg: How start explicit indexing with 1 instead of 0 ?

* This is where set\_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

import numpy as np

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

ser

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 explicit indexing is now starting from 1 instead of 0

Now lets go back to indexing and slicing

#### There is a slight problem in it

Lets look at another dummy series to understand this

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

data

1 a

5 b

3 c

dtype: object

data[1] # Uses explicit index

'a'

data[1:3] # Uses implicit index

5 b

3 c

dtype: object

#### 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

data.loc[1]

'a'

data.loc[1:3]

1 a

5 b

3 c

dtype: object

* 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

data.iloc[1]

'b'

#### Now will iloc also consider the range inclusive?

data.iloc[0:2]

1 a

5 b

dtype: object

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

#### 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 ?

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

df.loc[3] # Row with label 3

country Afghanistan

year 1967

lifeExp 34.02

population 11537966

gdpPerCap 836.197138

Name: 3, dtype: object

df.iloc[3] # Row at position 3

country Afghanistan

year 1967

lifeExp 34.02

population 11537966

gdpPerCap 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

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

|  | **country** | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- | --- |
| **1** | Afghanistan | 1957 | 30.332 | 9240934 | 820.853030 |
| **10** | Afghanistan | 2002 | 42.129 | 25268405 | 726.734055 |
| **100** | Bangladesh | 1972 | 45.252 | 70759295 | 630.233627 |

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

|  | **country** | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- | --- |
| **1** | Afghanistan | 1957 | 30.332 | 9240934 | 820.853030 |
| **10** | Afghanistan | 2002 | 42.129 | 25268405 | 726.734055 |
| **100** | Bangladesh | 1972 | 45.252 | 70759295 | 630.233627 |

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

#### Which one will work?

df.iloc[-1]

# Works and gives last row in dataframe

country Zimbabwe

year 2007

lifeExp 43.487

population 12311143

gdpPerCap 469.709298

Name: 1703, dtype: object

df.loc[-1]

# Does NOT work

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

/usr/local/lib/python3.7/dist-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)

<ipython-input-52-aa5e0c2173ab> in <module>()

----> 1 df.loc[-1]

2

3 # Does NOT work

KeyError: -1

#### 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 we want to use one of the columns as row index?

* Using the set\_index method

temp = df.set\_index("country")

temp

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

1704 rows × 4 columns

temp["lifeExp"]["Afghanistan"]

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: lifeExp, 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 Afghanistan is

temp.loc['Afghanistan']

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

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

### Deleting a row

#### What if we want to delete a row ?

* Using the df.drop() command `

#### 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?

* **drop() uses labels**, NOT positional indices

df.head()

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

# Let's drop row with label 3

df.drop(3, axis=0, inplace=True)

df.head()

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

* Now we see that **Row with label 3 is deleted**
* Row with label 3 is not there
* **Labels do NOT change on their own**

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

df.loc[4]

country Afghanistan

year 1972

lifeExp 36.088

population 13079460

gdpPerCap 739.981106

Name: 4, dtype: object

df.iloc[4]

country Afghanistan

year 1977

lifeExp 38.438

population 14880372

gdpPerCap 786.11336

Name: 5, dtype: object

## Working with Rows and Columns together

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

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)

|  | **year** | **lifeExp** | **population** |
| --- | --- | --- | --- |
| **1** | 1957 | 30.332 | 9240934 |
| **2** | 1962 | 31.997 | 10267083 |
| **4** | 1972 | 36.088 | 13079460 |
| **5** | 1977 | 38.438 | 14880372 |

#### Can we do the same thing with loc?

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

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-62-494208dc7680> in <module>()

----> 1 df.loc[1:5, 1:4]

TypeError: cannot do slice indexing on Index with these indexers [1] of type int

#### Slicing using indices doesn’t work with loc

* **Column labels are NOT correct**
* Because **loc works with labels**
* **Labels for rows are 0, 1, 3, …**
* **Labels for columns are country, continent, year, …**
  + NOT 0, 1, 2, 3, …

df.loc[1:5, ['country','lifeExp']]

# Row with label 5 will be included

# Columns labels are packed in []

|  | **country** | **lifeExp** |
| --- | --- | --- |
| **1** | Afghanistan | 30.332 |
| **2** | Afghanistan | 31.997 |
| **4** | Afghanistan | 36.088 |
| **5** | Afghanistan | 38.438 |

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

* Column range 'continent':'lifeExp' works !!

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

# Row range 1 to 5 (inclusive)

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

|  | **year** | **lifeExp** | **population** |
| --- | --- | --- | --- |
| **1** | 1957 | 30.332 | 9240934 |
| **2** | 1962 | 31.997 | 10267083 |
| **4** | 1972 | 36.088 | 13079460 |
| **5** | 1977 | 38.438 | 14880372 |

#### How can we get specific rows and columns?

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

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

|  | **country** | **lifeExp** | **population** |
| --- | --- | --- | --- |
| **0** | Afghanistan | 28.801 | 8425333 |
| **11** | Afghanistan | 43.828 | 31889923 |
| **101** | Bangladesh | 46.923 | 80428306 |

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

df.iloc[1:10:2]

|  | **country** | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- | --- |
| **1** | Afghanistan | 1957 | 30.332 | 9240934 | 820.853030 |
| **4** | Afghanistan | 1972 | 36.088 | 13079460 | 739.981106 |
| **6** | Afghanistan | 1982 | 39.854 | 12881816 | 978.011439 |
| **8** | Afghanistan | 1992 | 41.674 | 16317921 | 649.341395 |
| **10** | Afghanistan | 2002 | 42.129 | 25268405 | 726.734055 |

## Let’s look at more in-built operations in Pandas

#### Let’s store the 'lifeExp' column in a separate variable le

* mean() gives us the mean of values in entire column

le = df['lifeExp']

le.mean()

59.489386189078054

#### We can see more methods by pressing “tab” after le.

* sum()
* count()
* min()
* max()
* … and so on

# Gives us the sum of values in a column

le.sum()

101310.42468000001

# Gives us the number of values in a column

le.count()

1703

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

le.sum() / le.count()

59.489386189078104

#### Now what if we want sine of a column ? Or if we want to multiply it with pi ?

* To do so we can simply use numpy universal functions

import numpy as np

np.sin(le \* np.pi / 4)

0 -0.588420

1 -0.966196

2 -0.002356

4 -0.069060

5 -0.941412

...

1699 -0.962242

1700 -0.291787

1701 -0.804842

1702 -0.008639

1703 0.392096

Name: lifeExp, Length: 1703, dtype: float64

## Sorting

#### How can we perform sorting in pandas ?

* Just **use df.sort\_values()**

df.sort\_values(['year'])

|  | **country** | **year** | **lifeExp** | **population** | **gdpPerCap** |
| --- | --- | --- | --- | --- | --- |
| **0** | Afghanistan | 1952 | 28.801 | 8425333 | 779.445314 |
| **552** | Gambia | 1952 | 30.000 | 284320 | 485.230659 |
| **564** | Germany | 1952 | 67.500 | 69145952 | 7144.114393 |
| **576** | Ghana | 1952 | 43.149 | 5581001 | 911.298937 |
| **588** | Greece | 1952 | 65.860 | 7733250 | 3530.690067 |
| **...** | ... | ... | ... | ... | ... |
| **611** | Guatemala | 2007 | 70.259 | 12572928 | 5186.050003 |
| **1223** | Philippines | 2007 | 71.688 | 91077287 | 3190.481016 |
| **1571** | Tunisia | 2007 | 73.923 | 10276158 | 7092.923025 |
| **623** | Guinea | 2007 | 56.007 | 9947814 | 942.654211 |
| **1703** | Zimbabwe | 2007 | 43.487 | 12311143 | 469.709298 |

1703 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 'lifeExp'

df.sort\_values(['lifeExp'])

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

1703 rows × 5 columns

* Now the rows are sorted in **ascending order of lifeExp**

#### 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

df.sort\_values(['lifeExp', 'year'])

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

1703 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 []**

df.sort\_values(['year', 'lifeExp'], ascending=[False, True])

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

1703 rows × 5 columns

## Creating dataframes from scratch

#### Now we’ll see how to create a Series from scratch

* We’ll use a **class constructor Series()**

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

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**
* **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**

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

|  | **A** | **B** |
| --- | --- | --- |
| **0** | 10 | 20 |
| **1** | 30 | 40 |

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

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

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-78-0201905ec1c1> in <module>()

----> 1 pd.DataFrame([10,20], columns=['A','B'])

/usr/local/lib/python3.7/dist-packages/pandas/core/frame.py in \_\_init\_\_(self, data, index, columns, dtype, copy)

715 dtype=dtype,

716 copy=copy,

--> 717 typ=manager,

718 )

719 else:

/usr/local/lib/python3.7/dist-packages/pandas/core/internals/construction.py in ndarray\_to\_mgr(values, index, columns, dtype, copy, typ)

322 )

323

--> 324 \_check\_values\_indices\_shape\_match(values, index, columns)

325

326 if typ == "array":

/usr/local/lib/python3.7/dist-packages/pandas/core/internals/construction.py in \_check\_values\_indices\_shape\_match(values, index, columns)

391 passed = values.shape

392 implied = (len(index), len(columns))

--> 393 raise ValueError(f"Shape of passed values is {passed}, indices imply {implied}")

394

395

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]]**

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

|  | **A** | **B** |
| --- | --- | --- |
| **0** | 10 | 20 |

#### 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**

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

|  | **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

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

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

a

|  | **A** | **B** |
| --- | --- | --- |
| **0** | 10 | 20 |
| **1** | 30 | 40 |

b

|  | **A** | **C** |
| --- | --- | --- |
| **0** | 10 | 20 |
| **1** | 30 | 40 |

#### We just use pd.concat()

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

pd.concat([a, b])

|  | **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

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

0 10

0 10

Name: A, dtype: int64

We obviously want the indices to be unique for each row

#### How can we do this ?

* By setting ignore\_index = True

pd.concat([a, b], ignore\_index = True)

|  | **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

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

|  | **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**

#### 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

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

|  | **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

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

|  | **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

Concatentaion takes place only through axis = 0

a.append(b, ignore\_index = False)

|  | **A** | **B** | **C** |
| --- | --- | --- | --- |
| **0** | 10 | 20.0 | NaN |
| **1** | 30 | 40.0 | NaN |
| **0** | 10 | NaN | 20.0 |
| **1** | 30 | NaN | 40.0 |

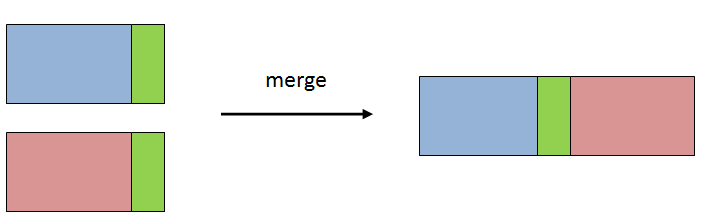
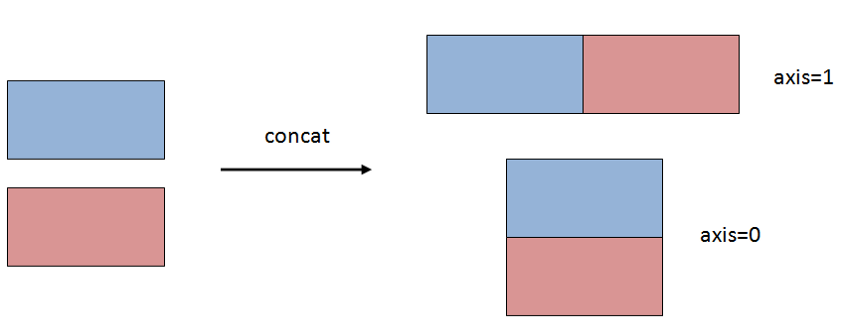
Note:

* 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**

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

users

|  | **userid** | **name** |
| --- | --- | --- |
| **0** | 1 | A |
| **1** | 2 | B |
| **2** | 3 | C |

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

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

msgs

|  | **userid** | **name** |
| --- | --- | --- |
| **0** | 1 | A |
| **1** | 2 | B |
| **2** | 3 | C |

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

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

msgs

|  | **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

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

|  | **userid** | **name** | **msg** |
| --- | --- | --- | --- |
| **0** | 1 | A | hello |
| **1** | 1 | A | bye |
| **2** | 2 | B | hi |

#### 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

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

|  | **id** | **name** | **userid** | **msg** |
| --- | --- | --- | --- | --- |
| **0** | 1 | A | 1 | hello |
| **1** | 1 | A | 1 | bye |
| **2** | 2 | B | 2 | hi |

### 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()

users.merge(msgs, how = "inner", left\_on = "id", right\_on = "userid")

|  | **id** | **name** | **userid** | **msg** |
| --- | --- | --- | --- | --- |
| **0** | 1 | A | 1 | hello |
| **1** | 1 | A | 1 | bye |
| **2** | 2 | B | 2 | hi |

#### 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

users.merge(msgs, how = "outer", left\_on = "id", right\_on = "userid")

|  | **id** | **name** | **userid** | **msg** |
| --- | --- | --- | --- | --- |
| **0** | 1 | A | 1.0 | hello |
| **1** | 1 | A | 1.0 | bye |
| **2** | 2 | B | 2.0 | hi |
| **3** | 3 | C | NaN | NaN |

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

* We can use left join for that

users.merge(msgs, how = "left", left\_on = "id", right\_on = "userid")

|  | **id** | **name** | **userid** | **msg** |
| --- | --- | --- | --- | --- |
| **0** | 1 | A | 1.0 | hello |
| **1** | 1 | A | 1.0 | bye |
| **2** | 2 | B | 2.0 | hi |
| **3** | 3 | C | NaN | NaN |

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

* Returns join over cols of right input

users.merge(msgs, how = "right", left\_on = "id", right\_on = "userid")

|  | **id** | **name** | **userid** | **msg** |
| --- | --- | --- | --- | --- |
| **0** | 1 | A | 1 | hello |
| **1** | 1 | A | 1 | bye |
| **2** | 2 | B | 2 | hi |

Lets visualise these joins using a venn diagram

### This is all about Pandas for now

* You can explore other methods for performing different tasks on your own
* We’ll cover a few more important concepts in the next lecture

Last changed by

**Scaler Topics**Scaler Topics

 369 [1](https://hackmd.io/@scaler-topics-main/SJr6tqubq)

Published on [**HackMD**](https://hackmd.io/)