# 02-Pandas-Lecture-IMDB

## Content

- **Working with both rows and columns**
- **Pandas built-in operations**
  - **Sorting**
  - **Concatenation**
  - **Merge**
- **Intoduction to IMDB dataset**
  - **Merging the dataframes**
  - **Feature Exploration**
  - **Fetching data using pandas**

## Working with Rows and Columns together

### Reading dataset

We will be using our earlier McKinsey dataset for now

Link:https://drive.google.com/file/d/1E3bwvYGf1ig32RmcYiWc0IXPN-mD_bI_/view?usp=sharing

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

```
--2023-09-13 15:17:05--  https://drive.google.com/uc?export=download&id=1E3bwvYGf1ig32RmcYiWc0IXPN-mD_bI_
Resolving drive.google.com (drive.google.com)... 142.251.16.100, 142.251.16.102, 142.251.16.113, ...
Connecting to drive.google.com (drive.google.com)|142.251.16.100|:443... connected.
HTTP request sent, awaiting response... 303 See Other
Location: https://doc-0s-68-docs.googleusercontent.com/docs/securesc/ha0ro937gcuc7l7deffksulhg5h7mbp1/1faj8dbj13vdr93ggo
Warning: wildcards not supported in HTTP.
--2023-09-13 15:17:05--  https://doc-0s-68-docs.googleusercontent.com/docs/securesc/ha0ro937gcuc7l7deffksulhg5h7mbp1/1fa
Resolving doc-0s-68-docs.googleusercontent.com (doc-0s-68-docs.googleusercontent.com)... 172.253.122.132, 2607:f8b0:4004
Connecting to doc-0s-68-docs.googleusercontent.com (doc-0s-68-docs.googleusercontent.com)|172.253.122.132|:443... connec
HTTP request sent, awaiting response... 200 OK
Length: 83785 (82K) [text/csv]
Saving to: ‘mckinsey.csv’

mckinsey.csv        100%[===================>]  81.82K  --.-KB/s    in 0.009s

2023-09-13 15:17:05 (8.77 MB/s) - ‘mckinsey.csv’ saved [83785/83785]
```

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

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

### Now how can we add a row to our dataframe?

There are multiple ways to do this:

- `append()`
- `loc/iloc`

### How can we do add a row using the **append()** method?

```
new_row = {'Country': 'India', 'year': 2000,'life_exp':37.08,'population':13500000,'gdp_cap':900.23}
df.append(new_row)
```

```
<ipython-input-4-714c78525e27>:2: FutureWarning: The frame.append method is deprecated and will be removed from pandas ir
  df.append(new_row)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-714c78525e27> in <cell line: 2>()
      1 new_row = {'Country': 'India', 'year': 2000,'life_exp':37.08,'population':13500000,'gdp_cap':900.23}
----> 2 df.append(new_row)

                                  1 frames
/usr/local/lib/python3.10/dist-packages/pandas/core/frame.py in _append(self, other, ignore_index, verify_integrity, sort
   9778             if isinstance(other, dict):
   9779                 if not ignore_index:
-> 9780                     raise TypeError("Can only append a dict if ignore_index=True")
   9781                 other = Series(other)
   9782             if other.name is None and not ignore_index:

TypeError: Can only append a dict if ignore_index=True
```

SEARCH STACK OVERFLOW

Why are we getting an error here?

Its' saying the `ignore_index()` parameter needs to be set to True. This parameter tells **Pandas** to ignore the existing index and create a new one based on the length of the resulting DataFrame.

```
new_row = {'Country': 'India', 'year': 2000,'life_exp':37.08,'population':13500000,'gdp_cap':900.23}
df = df.append(new_row, ignore_index=True)
df
```

```
<ipython-input-5-39ca58b35231>:2: FutureWarning: The frame.append method is deprecated and will be removed from pandas ir
  df = df.append(new_row, ignore_index=True)
```

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

1705 rows × 7 columns

Perfect! So now our row is added at the bottom of the 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.

Another method would be by **using loc:**

We will need to provide the position at which we will add the new row

What do you think this positional value would be?

```
df.loc[len(df.index)] = ['India',2000 ,13500000,37.08,900.23]  # len(df.index) since we will add at the last row
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-29b4966da254> in <cell line: 1>()
----> 1 df.loc[len(df.index)] = ['India',2000 ,13500000,37.08,900.23]  # len(df.index) since we will add at the last row

 2 frames
/usr/local/lib/python3.10/dist-packages/pandas/core/indexing.py in _setitem_with_indexer_missing(self, indexer, value)
   2158                     # must have conforming columns
   2159                     if len(value) != len(self.obj.columns):
-> 2160                         raise ValueError("cannot set a row with mismatched columns")
   2161
   2162                 value = Series(value, index=self.obj.columns, name=indexer)

ValueError: cannot set a row with mismatched columns
```

SEARCH STACK OVERFLOW

```
df
```

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

1705 rows × 7 columns

The new row was added but the data has been duplicated

What you can infer from last two duplicate rows ?

Dataframe allow us to feed duplicate rows in the data

## Now, can we also **use iloc**?

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

```
df.iloc[len(df.index)-1] = ['India', 2000,13500000,37.08,900.23]
df
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-8-c6ff7fd1a207> in <cell line: 1>()
----> 1 df.iloc[len(df.index)-1] = ['India', 2000,13500000,37.08,900.23]
      2 df

 2 frames
/usr/local/lib/python3.10/dist-packages/pandas/core/indexing.py in _setitem_with_indexer_split_path(self, indexer, value,
   1877
   1878             else:
-> 1879                 raise ValueError(
   1880                     "Must have equal len keys and value "
   1881                     "when setting with an iterable"

ValueError: Must have equal len keys and value when setting with an iterable
```

SEARCH STACK OVERFLOW

## What if we try to add the row with a new index?

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

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-9-1ad11b3daf34> in <cell line: 1>()
----> 1 df.iloc[len(df.index)] = ['India', 2000,13500000,37.08,900.23]

 1 frames
/usr/local/lib/python3.10/dist-packages/pandas/core/indexing.py in _has_valid_setitem_indexer(self, indexer)
   1516             elif is_integer(i):
   1517                 if i >= len(ax):
-> 1518                     raise IndexError("iloc cannot enlarge its target object")
   1519             elif isinstance(i, dict):
   1520                 raise IndexError("iloc cannot enlarge its target object")

IndexError: iloc cannot enlarge its target object
```

SEARCH STACK OVERFLOW

Why we are getting error ?

For using iloc to add a row, the dataframe must already have a row in that position.

If a row is not available, you'll see this IndexError

**Please Note:**

- When using the `loc[]` attribute, it's not mandatory that a row already exists with a specific label.

## Now what if we want to delete a row ?

Use `df.drop()`

If you remember we specified axis=1 for columns

We can modify this for rows

- We can use `axis=0` for rows

### 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 drop the row with label 3

```
df
```

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

1705 rows × 7 columns

```
# After dropping the row
df = df.drop(3, axis=0)
df
```

| | country | year | population | continent | life_exp | gdp_cap | Country |
|---|---|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | Asia | 28.801 | 779.445314 | NaN |
| **1** | Afghanistan | 1957 | 9240934 | Asia | 30.332 | 820.853030 | NaN |
| **2** | Afghanistan | 1962 | 10267083 | Asia | 31.997 | 853.100710 | NaN |
| **4** | Afghanistan | 1972 | 13079460 | Asia | 36.088 | 739.981106 | NaN |
| **5** | Afghanistan | 1977 | 14880372 | Asia | 38.438 | 786.113360 | NaN |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **1700** | Zimbabwe | 1992 | 10704340 | Africa | 60.377 | 693.420786 | NaN |
| **1701** | Zimbabwe | 1997 | 11404948 | Africa | 46.809 | 792.449960 | NaN |
| **1702** | Zimbabwe | 2002 | 11926563 | Africa | 39.989 | 672.038623 | NaN |
| **1703** | Zimbabwe | 2007 | 12311143 | Africa | 43.487 | 469.709298 | NaN |
| **1704** | NaN | 2000 | 13500000 | NaN | 37.080 | 900.230000 | India |

1704 rows × 7 columns

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

We now have **rows with labels 0, 1, 2, 4, 5, ...**

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

```
df.loc[4] # The 4th row is printed
```

```
Country       Afghanistan
year                 1972
population       13079460
life_exp           36.088
gdp_cap        739.981106
Name: 4, dtype: object
```

```
df.iloc[4] # The 5th row is printed
```

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

**Why did this happen?**

It is because the `loc` function selects rows using row labels (0,1,2,4 etc.) whereas the `iloc` function selects rows using their integer positions (staring from 0 and going up by one for each row).

So for `iloc` the 5th row starting from 0 index was printed

## And how can we drop multiple rows?

```
df.drop([1, 2, 4], axis=0) # drops rows with labels 1, 2, 4
```

Let's reset our indices now

```
df.reset_index(drop=True,inplace=True) # Since we removed a row earlier, we reset our indices
```

Now if you remember, the last two rows were duplicates.

## How can we deal with these duplicate rows?

Let's create some more duplicate rows to understand this

```
df.loc[len(df.index)] = ['India',2000,13500000,37.08,900.23]
df.loc[len(df.index)] = ['Sri Lanka',2022 ,130000000,80.00,500.00]
df.loc[len(df.index)] = ['Sri Lanka',2022 ,130000000,80.00,500.00]
df.loc[len(df.index)] = ['India',2000 ,13500000,80.00,900.23]
df
```

Now how can we check for duplicate rows?

Use `duplicated()` method on the DataFrame

```
df.duplicated()
```

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

However, it is not practical to see a list of True and False

We can Pandas `loc` data selector to extract those duplicate rows

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

The first argument **df.duplicated()** will find the duplicate rows

The second argument `:` will display all columns

Now how can we remove these **duplicate rows** ?

We can use `drop_duplicates()` of Pandas for this

```
df.drop_duplicates()
```

But how can we decide among all duplicate rows which ones we want to keep ?

Here we can use argument **keep**:

This Controls how to consider duplicate value.

It has only three distinct value

- `first`
- `last`
- `False`

The default is 'first'.

If `first`, this considers first value as unique and rest of the same values as duplicate.

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

If `last`, This considers last value as unique and rest of the same values as duplicate.

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

If `False`, this considers all of the same values as duplicates.

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

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.

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

How can we slice the dataframe into, say, first 4 rows and first 3 columns?

We can use iloc

```
gdf.iloc[0:4, 0:3]
```

| | country | year | population |
|---|---|---|---|
| 0 | Afghanistan | 1952 | 8425333 |
| 1 | Afghanistan | 1957 | 9240934 |
| 2 | Afghanistan | 1962 | 10267083 |
| 3 | Afghanistan | 1967 | 11537966 |

Pass in **2 different ranges for slicing** - **one for row** and **one for column** just like Numpy

Recall, `iloc` doesn't include the end index while slicing

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

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-494208dc7680> in <cell line: 1>()
----> 1 df.loc[1:5, 1:4]

                                 8 frames
/usr/local/lib/python3.10/dist-packages/pandas/core/indexes/base.py in _maybe_cast_slice_bound(self, label, side, kind)
   6621         # reject them, if index does not contain label
   6622         if (is_float(label) or is_integer(label)) and label not in self:
-> 6623             raise self._invalid_indexer("slice", label)
   6624
   6625         return label

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

SEARCH STACK OVERFLOW

## Why does slicing using indices doesn't work with `loc`?

Recall, we need to work with explicit labels while using loc

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

| | country | life_exp |
|---|---|---|
| 1 | Afghanistan | 30.332 |
| 2 | Afghanistan | 31.997 |
| 3 | Afghanistan | 34.020 |
| 4 | Afghanistan | 36.088 |
| 5 | Afghanistan | 38.438 |

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

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

| | year | population |
|---|---|---|
| 1 | 1957 | 9240934 |
| 2 | 1962 | 10267083 |
| 3 | 1967 | 11537966 |
| 4 | 1972 | 13079460 |
| 5 | 1977 | 14880372 |

## How can we get specific rows and columns?

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

| | country | population | continent |
|---|---|---|---|
| **0** | Afghanistan | 8425333 | Asia |
| **10** | Afghanistan | 25268405 | Asia |
| **100** | Bangladesh | 70759295 | Asia |

We pass in those **specific indices packed in `[]`**

Can we do step slicing?

**Yes**, just like we did in Numpy

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

| | country | year | population | continent | life_exp | gdp_cap |
|---|---|---|---|---|---|---|
| **1** | Afghanistan | 1957 | 9240934 | Asia | 30.332 | 820.853030 |
| **3** | Afghanistan | 1967 | 11537966 | Asia | 34.020 | 836.197138 |
| **5** | Afghanistan | 1977 | 14880372 | Asia | 38.438 | 786.113360 |
| **7** | Afghanistan | 1987 | 13867957 | Asia | 40.822 | 852.395945 |
| **9** | Afghanistan | 1997 | 22227415 | Asia | 41.763 | 635.341351 |

Does step slicing work for loc too?

**YES**

```
df.loc[1:10:2]
```

| | country | year | population | continent | life_exp | gdp_cap |
|---|---|---|---|---|---|---|
| **1** | Afghanistan | 1957 | 9240934 | Asia | 30.332 | 820.853030 |
| **3** | Afghanistan | 1967 | 11537966 | Asia | 34.020 | 836.197138 |
| **5** | Afghanistan | 1977 | 14880372 | Asia | 38.438 | 786.113360 |
| **7** | Afghanistan | 1987 | 13867957 | Asia | 40.822 | 852.395945 |
| **9** | Afghanistan | 1997 | 22227415 | Asia | 41.763 | 635.341351 |

## Pandas built-in operation

Let's select the feature `'life_exp'`

```
le = df['life_exp']
le
```

```
0       28.801
1       30.332
2       31.997
3       34.020
4       36.088
         ...
1699    62.351
1700    60.377
1701    46.809
1702    39.989
1703    43.487
Name: life_exp, Length: 1704, dtype: float64
```

How can we find the mean of the col `life_exp`?

```
[ ]
```
*1 cell hidden*

What other operations can we do?

- `sum()`

- `count()`
- `min()`
- `max()`

... and so on

Note:

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

```
le.sum()
```

```
101344.44467999999
```

```
le.count()
```

```
1704
```

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

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

```
59.474439366197174
```

It gives the **mean** of life expectancy

## Sorting

If you notice, `life_exp` col is not sorted

### How can we perform sorting in pandas ?

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

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

1704 rows × 6 columns

Rows get sorted **based on values in `life_exp` column**

By **default**, values are sorted in **ascending order**

### How can we sort the rows in descending order?

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

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

1704 rows × 6 columns

Now the rows are sorted in **descending**

Can we do sorting on multiple columns?

**YES**

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

| | country | year | population | continent | life_exp | gdp_cap |
|---|---|---|---|---|---|---|
| **0** | Afghanistan | 1952 | 8425333 | Asia | 28.801 | 779.445314 |
| **552** | Gambia | 1952 | 284320 | Africa | 30.000 | 485.230659 |
| **36** | Angola | 1952 | 4232095 | Africa | 30.015 | 3520.610273 |
| **1344** | Sierra Leone | 1952 | 2143249 | Africa | 30.331 | 879.787736 |
| **1032** | Mozambique | 1952 | 6446316 | Africa | 31.286 | 468.526038 |
| **...** | ... | ... | ... | ... | ... | ... |
| **71** | Australia | 2007 | 20434176 | Oceania | 81.235 | 34435.367440 |
| **1487** | Switzerland | 2007 | 7554661 | Europe | 81.701 | 37506.419070 |
| **695** | Iceland | 2007 | 301931 | Europe | 81.757 | 36180.789190 |
| **671** | Hong Kong, China | 2007 | 6980412 | Asia | 82.208 | 39724.978670 |
| **803** | Japan | 2007 | 127467972 | Asia | 82.603 | 31656.068060 |

1704 rows × 6 columns

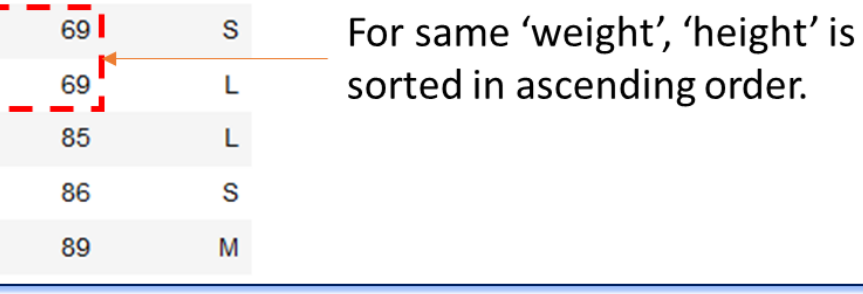
What exactly happened here?

- Rows were **first sorted** based on `'year'`
- Then, **rows with same values of** `'year'` were sorted based on `'lifeExp'`

**For Example**

```
df3 = df.sort_values(["weight", "height"])
df3.head(10)
```

| | name | age | height | weight | shirt_size |
|---|---|---|---|---|---|
| **2** | Rafael | 83 | 161 | 50 | M |
| **6** | Jacob | 29 | 178 | 63 | L |
| **0** | Ron | 30 | 153 | 69 | S |
| **3** | Karl-Hans | 34 | 169 | 69 | L |
| **5** | Ron | 55 | 172 | 85 | L |
| **4** | Freddy | 20 | 169 | 86 | S |
| **1** | Jacob | 24 | 153 | 89 | M |




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

- How can we have different sorting orders for different columns in multi-level sorting?

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

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

1704 rows × 6 columns

Just **pack `True` and `False` for respective columns in a list `[]`**

## Concatenating DataFrames

- Let's use a mini use-case of `users` and `messages`

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

```
users = pd.DataFrame({"userid":[1, 2, 3], "name":["sharadh", "shahid", "khusalli"]})
users
```

| | userid | name |
|---|---|---|
| **0** | 1 | sharadh |
| **1** | 2 | shahid |
| **2** | 3 | khusalli |

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

```
msgs = pd.DataFrame({"userid":[1, 1, 2, 4], "msg":['hmm', "acha", "theek hai", "nice"]})
msgs
```

| | userid | msg |
|---|---|---|
| **0** | 1 | hmm |
| **1** | 1 | acha |
| **2** | 2 | theek hai |
| **3** | 4 | nice |

Can we combine these 2 DataFrames to form a single DataFrame?

```
pd.concat([users, msgs])
```

| | userid | name | msg |
|---|---|---|---|
| **0** | 1 | sharadh | NaN |
| **1** | 2 | shahid | NaN |
| **2** | 3 | khusalli | NaN |
| **0** | 1 | NaN | hmm |
| **1** | 1 | NaN | acha |
| **2** | 2 | NaN | theek hai |
| **3** | 4 | NaN | nice |

How exactly did concat work?

- By **default, `axis=0` (row-wise) for concatenation**
- **`userid`**, being same in both DataFrames, was **combined into a single column**
  - First values of `users` dataframe were placed, with values of column `msg` as NaN
  - Then values of `msgs` dataframe were placed, with values of column `msg` as NaN
- The original indices of the rows were preserved

Now how can we make the indices unique for each row?

```
pd.concat([users, msgs], ignore_index = True)
```

| | userid | name | msg |
|---|---|---|---|
| **0** | 1 | sharadh | NaN |
| **1** | 2 | shahid | NaN |
| **2** | 3 | khusalli | NaN |
| **3** | 1 | NaN | hmm |
| **4** | 1 | NaN | acha |
| **5** | 2 | NaN | theek hai |
| **6** | 4 | NaN | nice |

How can we concatenate them horizontally?

```
pd.concat([users, msgs], axis=1)
```

| | userid | name | userid | msg |
|---|---|---|---|---|
| **0** | 1.0 | sharadh | 1 | hmm |
| **1** | 2.0 | shahid | 1 | acha |
| **2** | 3.0 | khusalli | 2 | theek hai |
| **3** | NaN | NaN | 4 | nice |

As you can see here:

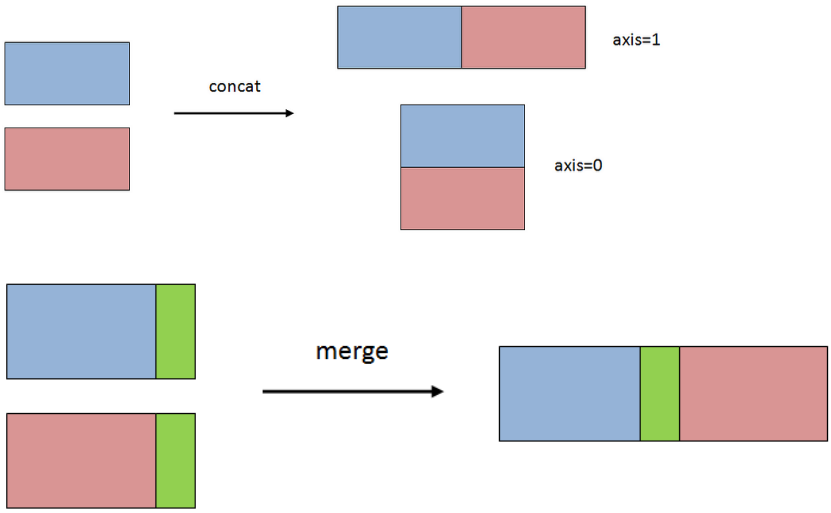
- Both the dataframes are combined horizontally (column-wise)
- It gives 2 columns with **different positional (implicit) index**, but **same label**

## Merging Dataframes

So far we have only concatenated and not merged data

But what is the difference between concat and merge ?

- `concat`
  - simply stacks multiple DataFrame together along an axis
- `merge`
  - combines dataframes in a **smart** way based on values in shared columns




### How can we know the **name of the person who sent a particular message?**

We need information from **both the dataframes**

So can we use pd.concat() for combining the dataframes ?

**No**

```
pd.concat([users, msgs], axis=1)
```

| | userid | name | userid | msg |
|---|---|---|---|---|
| **0** | 1.0 | sharadh | 1 | hmm |
| **1** | 2.0 | shahid | 1 | acha |
| **2** | 3.0 | khusalli | 2 | theek hai |
| **3** | NaN | NaN | 4 | nice |

What are the problems with concat here?

- `concat` simply **combined/stacked the dataframe horizontally**
- If you notice, `userid 3` for **user** dataframe is stacked against `userid 2` for msg dataframe
- This way of stacking **doesn't help us gain any insights**

=> pd.concat() does not work according to the values in the columns

We need to **merge** the data

▾ How can we join the dataframes ?

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

| | userid | name | msg |
|---|---|---|---|
| **0** | 1 | sharadh | hmm |
| **1** | 1 | sharadh | acha |
| **2** | 2 | shahid | theek hai |

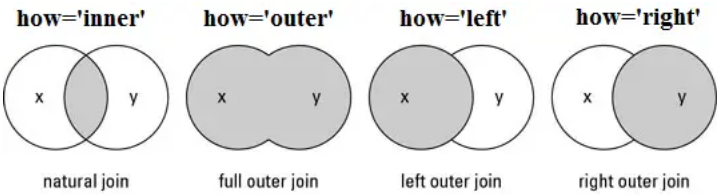
Notice that `users` has a userid = 3 but `msgs` does not

- When we **merge** these dataframes the **userid = 3 is not included**
- Similarly, **userid = 4 is not present** in `users`, and thus **not included**
- Only the userid **common in both dataframes** is shown

What type of join is this?

**Inner Join**

Remember joins from SQL?




The `on` parameter specifies the `key`, similar to `primary key` in SQL

▾ Now what join we want to use to get info of all the users and all the messages?

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

| | userid | name | msg |
|---|---|---|---|
| **0** | 1 | sharadh | hmm |
| **1** | 1 | sharadh | acha |
| **2** | 2 | shahid | theek hai |
| **3** | 3 | khusalli | NaN |
| **4** | 4 | NaN | nice |

Note:

All missing values are replaced with `NaN`

▾ And what if we want the info of all the users in the dataframe?

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

| | userid | name | msg |
|---|---|---|---|
| **0** | 1 | sharadh | hmm |
| **1** | 1 | sharadh | acha |
| **2** | 2 | shahid | theek hai |
| **3** | 3 | khusalli | NaN |

▾ Similarly, what if we want all the messages and info only for the users who sent a message?

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

| | userid | name | msg |
|---|---|---|---|
| **0** | 1 | sharadh | hmm |
| **1** | 1 | sharadh | acha |
| **2** | 2 | shahid | theek hai |
| **3** | 4 | NaN | nice |

Note,

**NaN** in **name** can be thought of as an anonymous message

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

Let's rename our users column `userid` to `id`

```
users.rename(columns = {"userid": "id"}, inplace = True)
users
```

| | id | name |
|---|---|---|
| **0** | 1 | sharadh |
| **1** | 2 | shahid |
| **2** | 3 | khusalli |

Now, how can we merge the 2 dataframes when the `key` has a different name ?

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

| | id | name | userid | msg |
|---|---|---|---|---|
| **0** | 1 | sharadh | 1 | hmm |
| **1** | 1 | sharadh | 1 | acha |
| **2** | 2 | shahid | 2 | theek hai |

Here,

- `left_on` : Specifies the **key of the 1st dataframe** (users here)
- `right_on` : Specifies the **key of the 2nd dataframe** (msgs here)

# IMDB Movie Business Use-case (Introduction)

Imagine you are working as a Data Scientist for an Analytics firm

Your task is to analyse some **movie trends** for a client

**IMDB** has online database of information related to movies

The database contains info of several years about:

- Movies
- Rating
- Director
- Popularity
- Revenue & Budget

Lets download and read the IMDB dataset

- File1: https://drive.google.com/file/d/1s2TkjSpzNc4SyxqRrQleZyDIHlc7bxnd/view?usp=sharing
- File2: https://drive.google.com/file/d/1Ws-_s1fHZ9nHfGLVUQurbHDvStePlEJm/view?usp=sharing

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

```
!gdown 1s2TkjSpzNc4SyxqRrQleZyDIHlc7bxnd
```

```
Downloading...
From: https://drive.google.com/uc?id=1s2TkjSpzNc4SyxqRrQleZyDIHlc7bxnd
To: /content/movies.csv
100% 112k/112k [00:00<00:00, 57.4MB/s]
```

```
!gdown 1Ws-_s1fHZ9nHfGLVUQurbHDvStePlEJm
```

```
Downloading...
From: https://drive.google.com/uc?id=1Ws-_s1fHZ9nHfGLVUQurbHDvStePlEJm
To: /content/directors.csv
100% 65.4k/65.4k [00:00<00:00, 62.8MB/s]
```

Here we have two csv files

- `movies.csv`
- `directors.csv`

```
movies = pd.read_csv('movies.csv')
movies.head()
```

| | Unnamed: 0 | id | budget | popularity | revenue | title | vote_average | vote_count | director_id | year | month | day |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 43597 | 237000000 | 150 | 2787965087 | Avatar | 7.2 | 11800 | 4762 | 2009 | Dec | Thursday |
| **1** | 1 | 43598 | 300000000 | 139 | 961000000 | Pirates of the Caribbean: At World's End | 6.9 | 4500 | 4763 | 2007 | May | Saturday |
| **2** | 2 | 43599 | 245000000 | 107 | 880674609 | Spectre | 6.3 | 4466 | 4764 | 2015 | Oct | Monday |

So what kind of questions can we ask from this dataset?

- **Top 10 most popular movies**, using `popularity`
- Or find some **highest rated movies**, using `vote_average`
- We can find number of **movies released per year** too
- Or maybe we can find **highest budget movies in a year** using both `budget` and `year`

But can we ask more interesting/deeper questions?

- Do you think we can find the **most productive directors**?
- Which **directors produce high budget films**?
- **Highest and lowest rated movies for every month** in a particular year?

Notice, there's a column **Unnamed: 0** which represents nothing but the index of a row.

How to get rid of this `Unnamed: 0` col?

```
movies = pd.read_csv('movies.csv', index_col=0)
movies.head()
```

| | id | budget | popularity | revenue | title | vote_average | vote_count | director_id | year | month | day |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 43597 | 237000000 | 150 | 2787965087 | Avatar | 7.2 | 11800 | 4762 | 2009 | Dec | Thursday |
| **1** | 43598 | 300000000 | 139 | 961000000 | Pirates of the Caribbean: At World's End | 6.9 | 4500 | 4763 | 2007 | May | Saturday |
| **2** | 43599 | 245000000 | 107 | 880674609 | Spectre | 6.3 | 4466 | 4764 | 2015 | Oct | Monday |
| **3** | 43600 | 250000000 | 112 | 1084939099 | The Dark Knight Rises | 7.6 | 9106 | 4765 | 2012 | Jul | Monday |

`index_col=0` explicitly states to treat the first column as the index

The default value is `index_col=None`

```
movies.shape
```