Pandas 2

Content

- Working with both rows & columns
- Handling duplicate records
- Pandas built-in operations
  - Aggregate functions
  - Sorting values
- Concatenating DataFrames
- Merging DataFrames

Working with rows & columns together

In [1]:
```
import pandas as pd
import numpy as np
```

In [2]:
```
df = pd.read_csv('mckinsey.csv')
df
```

Out[2]:

	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

How can we add a row to our dataframe?

There are multiple ways to do this.

- `concat()`
- `loc/iloc`

How can we do add a row using the `concat()` method?

In [3]:
```
new_row = {'country': 'India', 'year': 2000,'population':13500000, 'continer

df = pd.concat([df, pd.DataFrame([new_row])], ignore_index=True)
df
```

Out[3]:

	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
...	...	...	...	...	...	...
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	India	2000	13500000	Asia	37.080	900.230000

1705 rows × 6 columns

Why are we using `ignore_index=True`?

- This parameter tells Pandas to ignore the existing index and create a new one based on the length of the resulting DataFrame.

Perfect! Our row is now added at the bottom of the dataframe.

Note:

- `concat()` doesn't mutate the the dataframe.
- It does not change the DataFrame, but returns a new DataFrame with the appended row.

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

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

What do you think this positional value would be?

- `len(df.index)` since we will add the new row at the end.

For this method we only need to insert the values of columns in the respective manner.

In [4]:
```
new_row = {'country': 'India', 'year': 2000,'population':13500000, 'continer
new_row_val = list(new_row.values())
new_row_val
```

Out[4]:
```
['India', 2000, 13500000, 'Asia', 37.08, 900.23]
```

In [5]:
```
df.loc[len(df.index)] = new_row_val
df
```

Out[5]:

	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
...	...	...	...	...	...	...
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	India	2000	13500000	Asia	37.080	900.230000
1705	India	2000	13500000	Asia	37.080	900.230000

1706 rows × 6 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.

In [6]:
```
df.iloc[len(df.index)-1] = ['Japan', 1000, 1350000, 'Asia', 37.08, 100.23]
df
```

Out[6]:

	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
...	...	...	...	...	...	...
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	India	2000	13500000	Asia	37.080	900.230000
1705	Japan	1000	1350000	Asia	37.080	100.230000

1706 rows × 6 columns

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

In [7]:
```
df.iloc[len(df.index)] = ['India', 2000, 13500000, 'Asia', 37.08, 900.23]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[7], line 1
----> 1 df.iloc[len(df.index)] = ['India', 2000, 13500000, 'Asia', 37.08, 9
00.23]

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:815,
in _LocationIndexer.__setitem__(self, key, value)
    813     key = com.apply_if_callable(key, self.obj)
    814 indexer = self._get_setitem_indexer(key)
--> 815 self._has_valid_setitem_indexer(key)
    817 iloc = self if self.name == "iloc" else self.obj.iloc
    818 iloc._setitem_with_indexer(indexer, value, self.name)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1518,
in _iLocIndexer._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
```

Why are we getting an 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`.

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

What if we want to delete a row?

- use `df.drop()`

If you remember we specified axis=1 for columns.

We can modify this - `axis=0` for rows.

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

- We had to specify column title.
- So **`drop()` uses labels**, NOT positional indices.

\ Let's drop the row with label 3.

In [8]:
```
df
```

Out[8]:

	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
...	...	...	...	...	...	...
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	India	2000	13500000	Asia	37.080	900.230000
1705	Japan	1000	1350000	Asia	37.080	100.230000

1706 rows × 6 columns

In [9]:
```
df = df.drop(3, axis=0)
df
```

Out[9]:

	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
4	Afghanistan	1972	13079460	Asia	36.088	739.981106
5	Afghanistan	1977	14880372	Asia	38.438	786.113360
...	...	...	...	...	...	...
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	India	2000	13500000	Asia	37.080	900.230000
1705	Japan	1000	1350000	Asia	37.080	100.230000

1705 rows × 6 columns

We can see that the **row with label 3 is deleted**.

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

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

In [10]:
```
df.loc[4] # The 4th row is printed
```

Out[10]:
```
country       Afghanistan
year                 1972
population       13079460
continent            Asia
life_exp           36.088
gdp_cap        739.981106
Name: 4, dtype: object
```

In [11]:
```
df.iloc[4] # The 5th row is printed
```

Out[11]:
```
country       Afghanistan
year                 1977
population       14880372
continent            Asia
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,..) whereas the `iloc` function selects rows using their integer positions (staring from 0 and +1 for each row).

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

How can we drop multiple rows?

In [12]:
```
df.drop([1, 2, 4], axis=0) # drops rows with labels 1, 2, 4
```

Out[12]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
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
...	...	...	...	...	...	...
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	India	2000	13500000	Asia	37.080	900.230000
1705	Japan	1000	1350000	Asia	37.080	100.230000

1702 rows × 6 columns

Let's reset our indices now.

In [13]:
```
df.reset_index(drop=True,inplace=True) # since we removed a row earlier, we
df
```

Out[13]:

	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	1972	13079460	Asia	36.088	739.981106
4	Afghanistan	1977	14880372	Asia	38.438	786.113360
...	...	...	...	...	...	...
1700	Zimbabwe	1997	11404948	Africa	46.809	792.449960
1701	Zimbabwe	2002	11926563	Africa	39.989	672.038623
1702	Zimbabwe	2007	12311143	Africa	43.487	469.709298
1703	India	2000	13500000	Asia	37.080	900.230000
1704	Japan	1000	1350000	Asia	37.080	100.230000

1705 rows × 6 columns

Handling duplicate records

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.

In [14]:
```
df.loc[len(df.index)] = ['India', 2000, 13500000, 'Asia', 37.08, 900.23]
df.loc[len(df.index)] = ['Sri Lanka',2022 ,130000000, 'Asia', 80.00,500.00]
df.loc[len(df.index)] = ['Sri Lanka',2022 ,130000000, 'Asia', 80.00,500.00]
df.loc[len(df.index)] = ['India',2000 ,13500000, 'Asia', 80.00,900.23]
df
```

Out[14]:

	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	1972	13079460	Asia	36.088	739.981106
4	Afghanistan	1977	14880372	Asia	38.438	786.113360
...	...	...	...	...	...	...
1704	Japan	1000	1350000	Asia	37.080	100.230000
1705	India	2000	13500000	Asia	37.080	900.230000
1706	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1707	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1708	India	2000	13500000	Asia	80.000	900.230000

1709 rows × 6 columns

How to check for duplicate rows?

- We use `duplicated()` method on the DataFrame.

In [15]:
```
df.duplicated()
```

Out[15]:
```
0       False
1       False
2       False
3       False
4       False
        ...
1704    False
1705     True
1706    False
1707     True
1708    False
Length: 1709, dtype: bool
```

It gives True if an entire row is identical to the previous row.

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

We can the `loc` data selector to extract those duplicate rows.

In [16]:
```
df.loc[df.duplicated()]
```

Out[16]:

	country	year	population	continent	life_exp	gdp_cap
1705	India	2000	13500000	Asia	37.08	900.23
1707	Sri Lanka	2022	130000000	Asia	80.00	500.00

How do we get rid of these duplicate rows?

- We can use the `drop_duplicates()` function.

In [17]:
```
df.drop_duplicates()
```

Out[17]:

	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	1972	13079460	Asia	36.088	739.981106
4	Afghanistan	1977	14880372	Asia	38.438	786.113360
...	...	...	...	...	...	...
1702	Zimbabwe	2007	12311143	Africa	43.487	469.709298
1703	India	2000	13500000	Asia	37.080	900.230000
1704	Japan	1000	1350000	Asia	37.080	100.230000
1706	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1708	India	2000	13500000	Asia	80.000	900.230000

1707 rows × 6 columns

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

Here we can use the `keep` argument.

It has only three distinct values -

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

The default is 'first'.

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

In [18]:
```
df.drop_duplicates(keep='first')
```

Out[18]:

	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	1972	13079460	Asia	36.088	739.981106
4	Afghanistan	1977	14880372	Asia	38.438	786.113360
...	...	...	...	...	...	...
1702	Zimbabwe	2007	12311143	Africa	43.487	469.709298
1703	India	2000	13500000	Asia	37.080	900.230000
1704	Japan	1000	1350000	Asia	37.080	100.230000
1706	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1708	India	2000	13500000	Asia	80.000	900.230000

1707 rows × 6 columns

If `last`, this considers last value as unique and rest of the identical values as duplicate.

In [19]:
```
df.drop_duplicates(keep='last')
```

Out[19]:

	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	1972	13079460	Asia	36.088	739.981106
4	Afghanistan	1977	14880372	Asia	38.438	786.113360
...	...	...	...	...	...	...
1702	Zimbabwe	2007	12311143	Africa	43.487	469.709298
1704	Japan	1000	1350000	Asia	37.080	100.230000
1705	India	2000	13500000	Asia	37.080	900.230000
1707	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1708	India	2000	13500000	Asia	80.000	900.230000

1707 rows × 6 columns

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

In [20]:
```
df.drop_duplicates(keep=False)
```

Out[20]:

	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	1972	13079460	Asia	36.088	739.981106
4	Afghanistan	1977	14880372	Asia	38.438	786.113360
...	...	...	...	...	...	...
1700	Zimbabwe	1997	11404948	Africa	46.809	792.449960
1701	Zimbabwe	2002	11926563	Africa	39.989	672.038623
1702	Zimbabwe	2007	12311143	Africa	43.487	469.709298
1704	Japan	1000	1350000	Asia	37.080	100.230000
1708	India	2000	13500000	Asia	80.000	900.230000

1705 rows × 6 columns

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

We can use the `subset` argument to mention the list of columns which we want to use.

In [21]:
```
df.drop_duplicates(subset=['country'],keep='first')
```

Out[21]:

	country	year	population	continent	life_exp	gdp_cap
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
11	Albania	1952	1282697	Europe	55.230	1601.056136
23	Algeria	1952	9279525	Africa	43.077	2449.008185
35	Angola	1952	4232095	Africa	30.015	3520.610273
47	Argentina	1952	17876956	Americas	62.485	5911.315053
...	...	...	...	...	...	...
1643	Vietnam	1952	26246839	Asia	40.412	605.066492
1655	West Bank and Gaza	1952	1030585	Asia	43.160	1515.592329
1667	Yemen, Rep.	1952	4963829	Asia	32.548	781.717576
1679	Zambia	1952	2672000	Africa	42.038	1147.388831
1691	Zimbabwe	1952	3080907	Africa	48.451	406.884115

142 rows × 6 columns

Slicing the DataFrame

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

- We can use `iloc`

In [22]:
```
df.iloc[0:4, 0:3]
```

Out[22]:

	country	year	population
0	Afghanistan	1952	8425333
1	Afghanistan	1957	9240934
2	Afghanistan	1962	10267083
3	Afghanistan	1972	13079460

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`?

In [23]:
```
df.loc[1:5, 1:4]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[23], line 1
----> 1 df.loc[1:5, 1:4]

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1067,
in _LocationIndexer.__getitem__(self, key)
   1065     if self._is_scalar_access(key):
   1066         return self.obj._get_value(*key, takeable=self._takeable)
-> 1067     return self._getitem_tuple(key)
   1068 else:
   1069     # we by definition only have the 0th axis
   1070     axis = self.axis or 0

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1256,
in _LocIndexer._getitem_tuple(self, tup)
   1253 if self._multi_take_opportunity(tup):
   1254     return self._multi_take(tup)
-> 1256 return self._getitem_tuple_same_dim(tup)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:924,
in _LocationIndexer._getitem_tuple_same_dim(self, tup)
    921 if com.is_null_slice(key):
    922     continue
--> 924 retval = getattr(retval, self.name)._getitem_axis(key, axis=i)
    925 # We should never have retval.ndim < self.ndim, as that should
    926 #  be handled by the _getitem_lowerdim call above.
    927 assert retval.ndim == self.ndim

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1290,
in _LocIndexer._getitem_axis(self, key, axis)
   1288 if isinstance(key, slice):
   1289     self._validate_key(key, axis)
-> 1290     return self._get_slice_axis(key, axis=axis)
   1291 elif com.is_bool_indexer(key):
   1292     return self._getbool_axis(key, axis=axis)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexing.py:1324,
in _LocIndexer._get_slice_axis(self, slice_obj, axis)
   1321     return obj.copy(deep=False)
   1323 labels = obj._get_axis(axis)
-> 1324 indexer = labels.slice_indexer(slice_obj.start, slice_obj.stop, sli
ce_obj.step)
   1326 if isinstance(indexer, slice):
   1327     return self.obj._slice(indexer, axis=axis)

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexes/base.py:6
559, in Index.slice_indexer(self, start, end, step, kind)
   6516 """
   6517 Compute the slice indexer for input labels and step.
   6518
   (...)
   6555 slice(1, 3, None)
   6556 """
   6557 self._deprecated_arg(kind, "kind", "slice_indexer")
-> 6559 start_slice, end_slice = self.slice_locs(start, end, step=step)
   6561 # return a slice
   6562 if not is_scalar(start_slice):

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexes/base.py:6
767, in Index.slice_locs(self, start, end, step, kind)
   6765 start_slice = None
   6766 if start is not None:
-> 6767     start_slice = self.get_slice_bound(start, "left")
```

```
   6768 if start_slice is None:
   6769     start_slice = 0

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexes/base.py:6
676, in Index.get_slice_bound(self, label, side, kind)
   6672 original_label = label
   6674 # For datetime indices label may be a string that has to be convert
ed
   6675 # to datetime boundary according to its resolution.
-> 6676 label = self._maybe_cast_slice_bound(label, side)
   6678 # we need to look up the label
   6679 try:

File ~/anaconda3/lib/python3.11/site-packages/pandas/core/indexes/base.py:6
623, in Index._maybe_cast_slice_bound(self, label, side, kind)
   6618 # We are a plain index here (sub-class override this method if they
   6619 # wish to have special treatment for floats/ints, e.g. Float64Index
and
   6620 # datetimelike Indexes
   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)
   6625 return label

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

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

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

In [24]:
```
df.loc[1:5, ['country','life_exp']]
```

Out[24]:

	country	life_exp
1	Afghanistan	30.332
2	Afghanistan	31.997
3	Afghanistan	36.088
4	Afghanistan	38.438
5	Afghanistan	39.854

In `loc`, we can mention ranges using column labels as well.

In [25]:
```
df.loc[1:5, 'year':'life_exp']
```

Out[25]:

	year	population	continent	life_exp
1	1957	9240934	Asia	30.332
2	1962	10267083	Asia	31.997
3	1972	13079460	Asia	36.088
4	1977	14880372	Asia	38.438
5	1982	12881816	Asia	39.854

How can we get specific rows and columns?

In [26]:
```
df.iloc[[0,10,100], [0,2,3]]
```

Out[26]:

	country	population	continent
0	Afghanistan	8425333	Asia
10	Afghanistan	31889923	Asia
100	Bangladesh	80428306	Asia

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

Can we do step slicing? Yes!

In [27]:
```
df.iloc[1:10:2]
```

Out[27]:

	country	year	population	continent	life_exp	gdp_cap
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
3	Afghanistan	1972	13079460	Asia	36.088	739.981106
5	Afghanistan	1982	12881816	Asia	39.854	978.011439
7	Afghanistan	1992	16317921	Asia	41.674	649.341395
9	Afghanistan	2002	25268405	Asia	42.129	726.734055

Does step slicing work for `loc` too? Yes!

In [28]:
```
df.loc[1:10:2]
```

Out[28]:

	country	year	population	continent	life_exp	gdp_cap
1	Afghanistan	1957	9240934	Asia	30.332	820.853030
3	Afghanistan	1972	13079460	Asia	36.088	739.981106
5	Afghanistan	1982	12881816	Asia	39.854	978.011439
7	Afghanistan	1992	16317921	Asia	41.674	649.341395
9	Afghanistan	2002	25268405	Asia	42.129	726.734055

Pandas built-in operations

Aggregate functions

Let's select the feature `'life_exp'` -

In [29]:
```
le = df['life_exp']
le
```

Out[29]:
```
0       28.801
1       30.332
2       31.997
3       36.088
4       38.438
         ...
1704    37.080
1705    37.080
1706    80.000
1707    80.000
1708    80.000
Name: life_exp, Length: 1709, dtype: float64
```

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

In [30]:
```
le.mean()
```

Out[30]:
```
59.486053060269164
```

What other operations can we do?

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

... and so on

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

In [31]:
```
le.sum()
```

Out[31]:
```
101661.66468
```

In [32]:
```
le.count()
```

Out[32]:
```
1709
```

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

In [33]:
```
le.sum() / le.count()
```

Out[33]:
```
59.486053060269164
```

It gives us the **mean/average** of life expectancy.

Sorting Values

If you notice, the `life_exp` column is not sorted.

How can we perform sorting in Pandas?

In [34]:
```
df.sort_values(['life_exp'])
```

Out[34]:

	country	year	population	continent	life_exp	gdp_cap
1291	Rwanda	1992	7290203	Africa	23.599	737.068595
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
551	Gambia	1952	284320	Africa	30.000	485.230659
35	Angola	1952	4232095	Africa	30.015	3520.610273
1343	Sierra Leone	1952	2143249	Africa	30.331	879.787736
...	...	...	...	...	...	...
1486	Switzerland	2007	7554661	Europe	81.701	37506.419070
694	Iceland	2007	301931	Europe	81.757	36180.789190
801	Japan	2002	127065841	Asia	82.000	28604.591900
670	Hong Kong, China	2007	6980412	Asia	82.208	39724.978670
802	Japan	2007	127467972	Asia	82.603	31656.068060

1709 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?

In [35]:
```
df.sort_values(['life_exp'], ascending=False)
```

Out[35]:

	country	year	population	continent	life_exp	gdp_cap
802	Japan	2007	127467972	Asia	82.603	31656.068060
670	Hong Kong, China	2007	6980412	Asia	82.208	39724.978670
801	Japan	2002	127065841	Asia	82.000	28604.591900
694	Iceland	2007	301931	Europe	81.757	36180.789190
1486	Switzerland	2007	7554661	Europe	81.701	37506.419070
...	...	...	...	...	...	...
1343	Sierra Leone	1952	2143249	Africa	30.331	879.787736
35	Angola	1952	4232095	Africa	30.015	3520.610273
551	Gambia	1952	284320	Africa	30.000	485.230659
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
1291	Rwanda	1992	7290203	Africa	23.599	737.068595

1709 rows × 6 columns

Can we perform sorting on multiple columns? Yes!

In [36]:
```
df.sort_values(['year', 'life_exp'])
```

Out[36]:

	country	year	population	continent	life_exp	gdp_cap
1704	Japan	1000	1350000	Asia	37.080	100.230000
0	Afghanistan	1952	8425333	Asia	28.801	779.445314
551	Gambia	1952	284320	Africa	30.000	485.230659
35	Angola	1952	4232095	Africa	30.015	3520.610273
1343	Sierra Leone	1952	2143249	Africa	30.331	879.787736
...	...	...	...	...	...	...
694	Iceland	2007	301931	Europe	81.757	36180.789190
670	Hong Kong, China	2007	6980412	Asia	82.208	39724.978670
802	Japan	2007	127467972	Asia	82.603	31656.068060
1706	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1707	Sri Lanka	2022	130000000	Asia	80.000	500.000000

1709 rows × 6 columns

What exactly happened here?

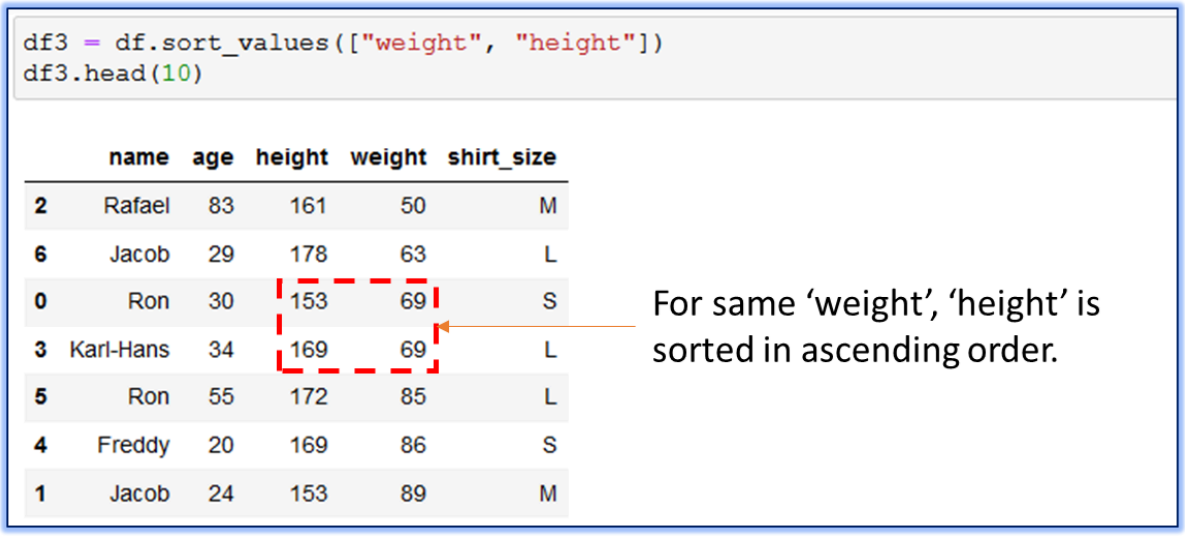
- Rows were **first sorted** based on `'year'`
- Then, **rows with same values of** `'year'` were sorted based on `'lifeExp'`

In [37]:
```
from IPython.display import Image
Image(filename='sort.png')
```

Out[37]:

```
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?

In [38]:
```
df.sort_values(['year', 'life_exp'], ascending=[False, True])
```

Out[38]:

	country	year	population	continent	life_exp	gdp_cap
1706	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1707	Sri Lanka	2022	130000000	Asia	80.000	500.000000
1462	Swaziland	2007	1133066	Africa	39.613	4513.480643
1042	Mozambique	2007	19951656	Africa	42.082	823.685621
1690	Zambia	2007	11746035	Africa	42.384	1271.211593
...	...	...	...	...	...	...
1463	Sweden	1952	7124673	Europe	71.860	8527.844662
1079	Netherlands	1952	10381988	Europe	72.130	8941.571858
683	Iceland	1952	147962	Europe	72.490	7267.688428
1139	Norway	1952	3327728	Europe	72.670	10095.421720
1704	Japan	1000	1350000	Asia	37.080	100.230000

1709 rows × 6 columns

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

Often times our data is separated into multiple tables, and we would require to work with them.

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

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

In [42]:
```
users = pd.DataFrame({"userid":[1, 2, 3], "name":["sharadh", "shahid", "khus
users
```

Out[42]:

	userid	name
0	1	sharadh
1	2	shahid
2	3	khusalli

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

In [43]:
```
msgs = pd.DataFrame({"userid":[1, 1, 2, 4], "msg":['hmm', "acha", "theek ha
msgs
```

Out[43]:

	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?

In [44]:
```
pd.concat([users, msgs])
```

Out[44]:

	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.

How can we make the indices unique for each row?

In [45]:
```
pd.concat([users, msgs], ignore_index = True)
```

Out[45]:

	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?

In [46]:
```
pd.concat([users, msgs], axis=1)
```

Out[46]:

	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 but not merged data.

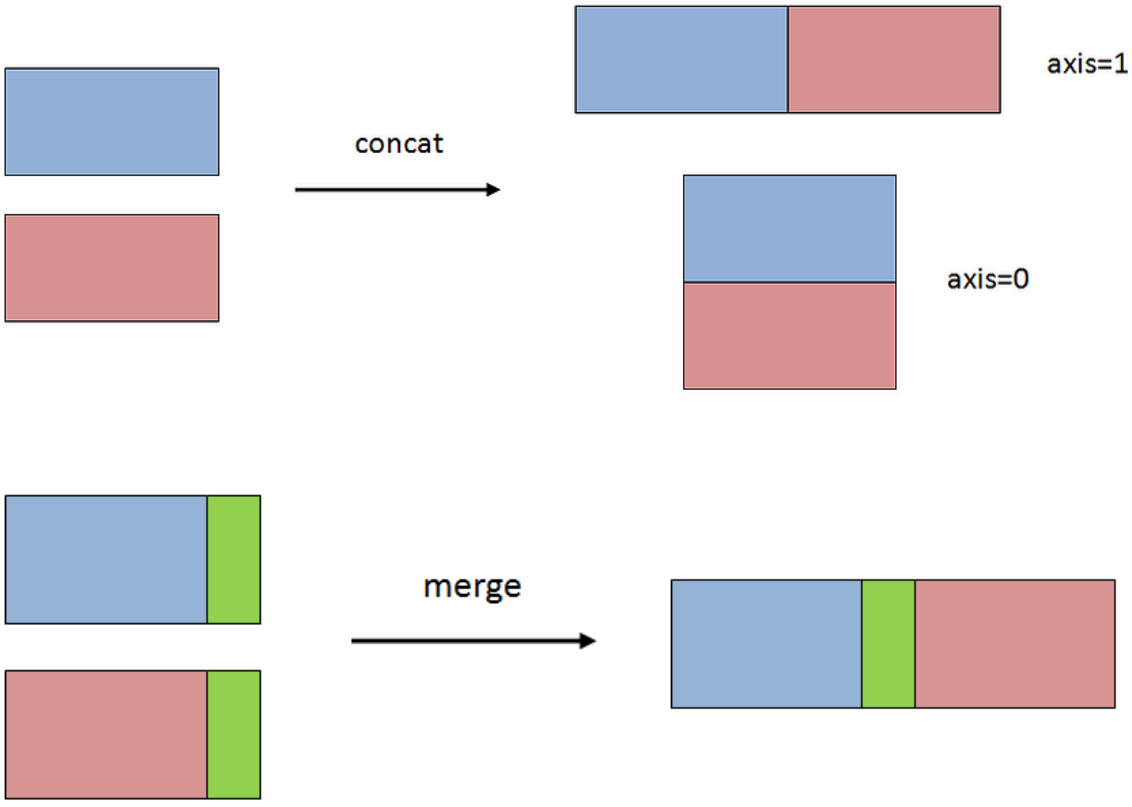
But what is the difference between `concat` and `merge`?

`concat`

- simply stacks multiple dataframes together along an axis.

`merge`

- combines dataframes in a **smart** way based on values in shared column(s).



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.

In [47]:
```
pd.concat([users, msgs], axis=1)
```

Out[47]:

	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 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.

We need to **merge** the data.

How can we join the dataframes?

In [48]:
```
users.merge(msgs, on="userid")
```

Out[48]:

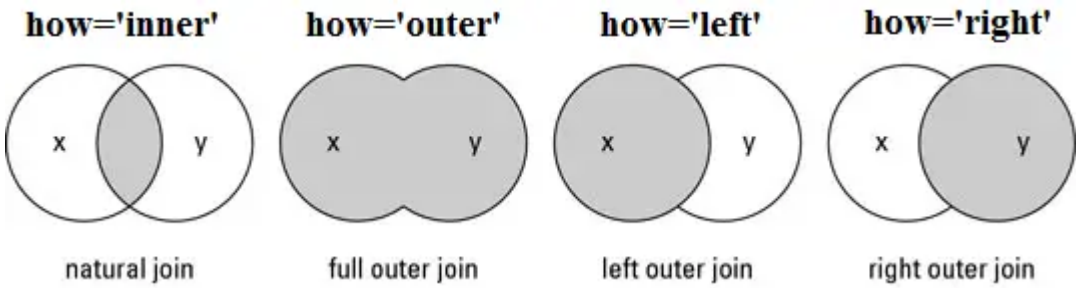
	userid	name	msg
0	1	sharadh	hmm
1	1	sharadh	acha
2	2	shahid	theek hai

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.

\ **What join we want to use to get info of all the users and all the messages?**

In [49]:
```
users.merge(msgs, on="userid", how="outer")
```

Out[49]:

	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`.

What if we want the info of all the users in the dataframe?

In [50]:
```
users.merge(msgs, on="userid", how="left")
```

Out[50]:

	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?

In [51]:
```
users.merge(msgs, on="userid", how="right")
```

Out[51]:

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

`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`.

In [52]:
```
users.rename(columns = {"userid": "id"}, inplace=True)
users
```

Out[52]:

	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 value?

In [53]:
```
users.merge(msgs, left_on="id", right_on="userid")
```

Out[53]:

	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).
- `right_on` : Specifies the **key of the 2nd dataframe** (msgs).

In [ ]:
```

```