By Shreeyansh Das, Source: gfg, Pandas Documentation

In [1]:

```
import pandas as pd
import numpy as np
import seaborn as sns
```

---

1. Grouping Data

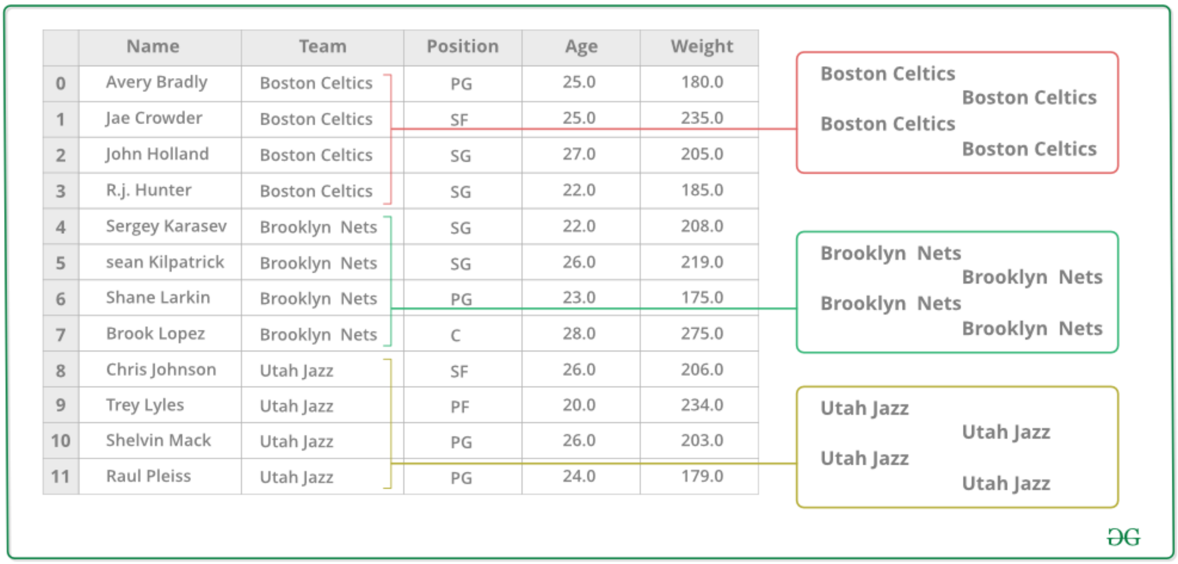
1.1 Pandas GroupBy

We can create a grouping of categories and apply a function to the categories. It's a simple concept but it's an extremely valuable technique that's widely used in data science. In real data science projects, you'll be dealing with large amounts of data and trying things over and over, so for efficiency, we use GroupBy concept. GroupBy concept is really important because it's ability to aggregate data efficiently, both in performance and the amount code is magnificent. GroupBy mainly refers to a process involving one or more of the following steps which are:

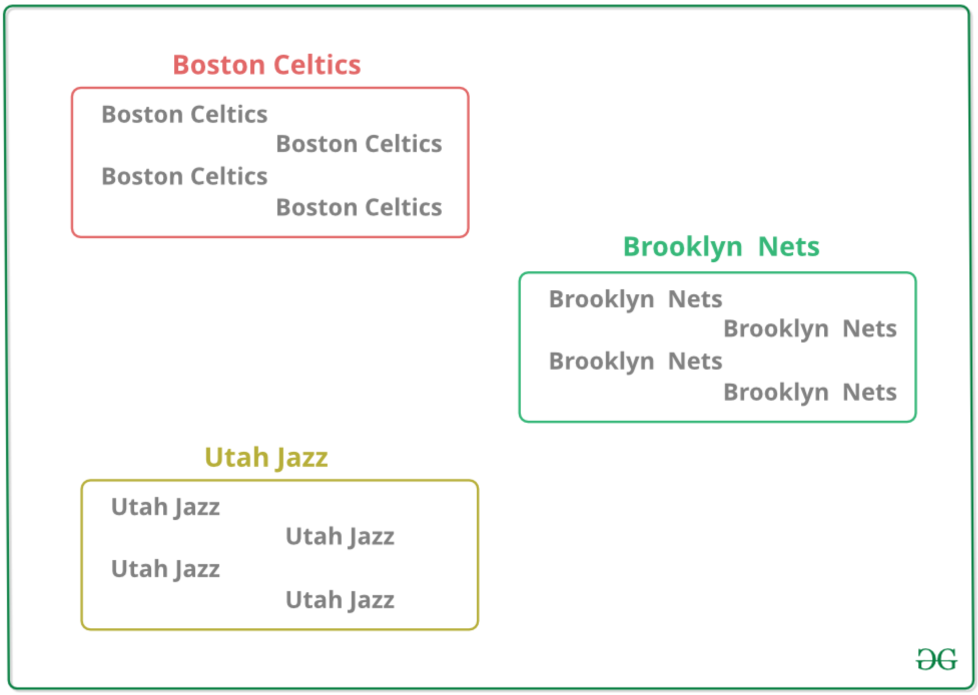
- **Splitting** : It is a process in which we split data into group by applying some conditions on datasets.
- **Applying** : It is a process in which we apply a function to each group independently
- **Combining** : It is a process in which we combine different datasets after applying GroupBy and results into a data structure

GroupBy Process

Step 1 : Group the unique values from a column.

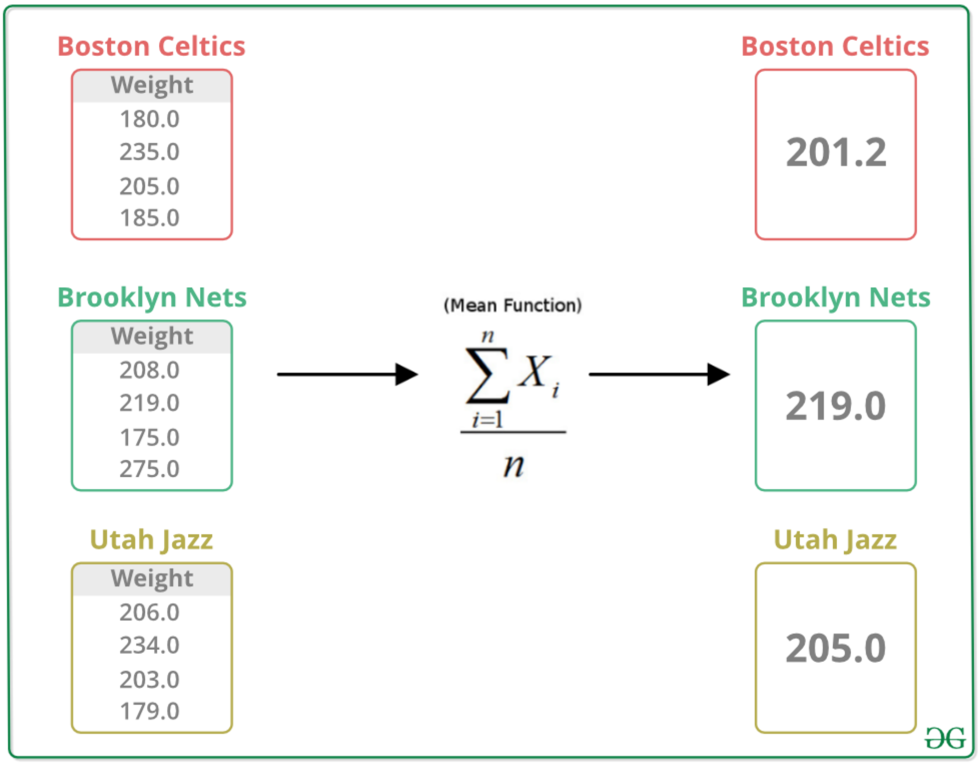


A group for each value created



Step 2: Toss other data into the groups

Step 3: Apply a function in desired column



1.1.1 Splitting Data into Groups

Splitting is a process in which we split data into a group by applying some conditions on datasets. We use `groupby()` function which is used to split the data into groups based on some criteria. Pandas objects can be split on any of their axes. The abstract definition of grouping is to provide a mapping of labels to group names. Pandas datasets can be split into any of their objects. There are multiple ways to split data like:

- `obj.groupby(key)`
- `obj.groupby(key, axis=1)`
- `obj.groupby([key1, key2])`

Note :In this we refer to the grouping objects as the keys.

Preparing DataSet for Operation

In [2]:
```
iris = sns.load_dataset('iris')
```

In [3]:
```
iris = iris.sample(30)
```

In [4]:
```
iris.reset_index(inplace = True, drop = True)
```

In [5]:
```
iris['category'] = np.random.choice(a = ['A','B','C'], size = 30, p = [0.34,0.33,0.3
```

In [6]:
```
iris.head()
```

Out[6]:

	sepal_length	sepal_width	petal_length	petal_width	species	category
0	5.1	3.8	1.6	0.2	setosa	A

	sepal_length	sepal_width	petal_length	petal_width	species	category
1	6.3	2.3	4.4	1.3	versicolor	B
2	5.7	2.8	4.1	1.3	versicolor	C
3	5.5	4.2	1.4	0.2	setosa	B
4	5.8	2.7	4.1	1.0	versicolor	B

- **Grouping Data with one key**

In order to group data with one key, we pass only one key as an argument in groupby function. Group keys are sorted by default during the groupby operation. *We can pass* `sort=False` *for potential speedups.*

In [7]:
```
gp = iris.groupby('species')
```

Groups with respective indices

In [8]:
```
gp.groups
```

Out[8]:
```
{'setosa': [0, 3, 5, 6, 8, 12, 14, 15, 21, 27, 28], 'versicolor': [1, 2, 4, 7, 9, 1
1, 13, 16, 18, 23, 24, 25, 26, 29], 'virginica': [10, 17, 19, 20, 22]}
```

First entry in all the groups formed

In [9]:
```
gp.first()
```

Out[9]:

	sepal_length	sepal_width	petal_length	petal_width	category
species					
setosa	5.1	3.8	1.6	0.2	A
versicolor	6.3	2.3	4.4	1.3	B
virginica	7.7	3.8	6.7	2.2	A

Some Basic Methods

In [10]:
```
gp.mean()
```

Out[10]:

	sepal_length	sepal_width	petal_length	petal_width
species				
setosa	5.045455	3.436364	1.509091	0.254545
versicolor	6.114286	2.878571	4.450000	1.385714
virginica	6.420000	2.980000	5.440000	2.000000

In [11]:
```
gp.sum()
```

Out[11]:

	sepal_length	sepal_width	petal_length	petal_width

species	sepal_length	sepal_width	petal_length	petal_width
species				
setosa	55.5	37.8	16.6	2.8
versicolor	85.6	40.3	62.3	19.4
virginica	32.1	14.9	27.2	10.0

In [12]:
```
gp.std()
```

Out[12]:

	sepal_length	sepal_width	petal_length	petal_width
species				
setosa	0.367052	0.480151	0.181409	0.082020
versicolor	0.489674	0.309288	0.354640	0.214322
virginica	0.798123	0.526308	0.712741	0.122474

- **Grouping Data with Multiple Keys**

In order to group data with multiple keys, we pass multiple keys in groupby function

In [13]:
```
gps = iris.groupby(['species','category'])
```

In [14]:
```
gps.groups
```

Out[14]:
```
{('setosa', 'A'): [0, 5, 6, 12, 14, 21], ('setosa', 'B'): [3, 15, 27, 28], ('setos
a', 'C'): [8], ('versicolor', 'A'): [13, 23, 24, 29], ('versicolor', 'B'): [1, 4, 7,
9, 11, 18, 25, 26], ('versicolor', 'C'): [2, 16], ('virginica', 'A'): [10, 17, 20],
('virginica', 'B'): [22], ('virginica', 'C'): [19]}
```

In [15]:
```
gps.first()
```

Out[15]:

		sepal_length	sepal_width	petal_length	petal_width
species	**category**				
setosa	**A**	5.1	3.8	1.6	0.2
	B	5.5	4.2	1.4	0.2
	C	4.6	3.4	1.4	0.3
versicolor	**A**	7.0	3.2	4.7	1.4
	B	6.3	2.3	4.4	1.3
	C	5.7	2.8	4.1	1.3
virginica	**A**	7.7	3.8	6.7	2.2
	B	6.5	3.2	5.1	2.0
	C	6.4	2.7	5.3	1.9

Understand that in dataset `gp` groups are made on the basis of species only while in `gps` groups are made on the basis of species and category.

1.1.2 Iterating over Groups

In order to iterate an element of groups, we can iterate through the object similar to itertools.obj.

In [16]:
```
for species,group in gp:
    print(species)
    print(group)
    print()
```

```
setosa
    sepal_length  sepal_width  petal_length  petal_width species category
0            5.1          3.8           1.6          0.2  setosa        A
3            5.5          4.2           1.4          0.2  setosa        B
5            5.0          3.3           1.4          0.2  setosa        A
6            5.5          3.5           1.3          0.2  setosa        A
8            4.6          3.4           1.4          0.3  setosa        C
12           5.4          3.4           1.5          0.4  setosa        A
14           4.5          2.3           1.3          0.3  setosa        A
15           5.4          3.9           1.7          0.4  setosa        B
21           4.7          3.2           1.6          0.2  setosa        A
27           4.8          3.4           1.9          0.2  setosa        B
28           5.0          3.4           1.5          0.2  setosa        B

versicolor
    sepal_length  sepal_width  petal_length  petal_width     species category
1            6.3          2.3           4.4          1.3  versicolor        B
2            5.7          2.8           4.1          1.3  versicolor        C
4            5.8          2.7           4.1          1.0  versicolor        B
7            5.6          3.0           4.5          1.5  versicolor        B
9            5.5          2.4           3.7          1.0  versicolor        B
11           6.8          2.8           4.8          1.4  versicolor        B
13           7.0          3.2           4.7          1.4  versicolor        A
16           5.9          3.2           4.8          1.8  versicolor        C
18           6.0          3.4           4.5          1.6  versicolor        B
23           6.0          2.7           5.1          1.6  versicolor        A
24           6.1          3.0           4.6          1.4  versicolor        A
25           6.7          3.1           4.4          1.4  versicolor        B
26           5.6          2.7           4.2          1.3  versicolor        B
29           6.6          3.0           4.4          1.4  versicolor        A

virginica
    sepal_length  sepal_width  petal_length  petal_width    species category
10           7.7          3.8           6.7          2.2  virginica        A
17           5.7          2.5           5.0          2.0  virginica        A
19           6.4          2.7           5.3          1.9  virginica        C
20           5.8          2.7           5.1          1.9  virginica        A
22           6.5          3.2           5.1          2.0  virginica        B
```

In [17]:
```
for species,group in gps:
    print(species)
    print(group)
    print()
```

```
('setosa', 'A')
    sepal_length  sepal_width  petal_length  petal_width species category
0            5.1          3.8           1.6          0.2  setosa        A
5            5.0          3.3           1.4          0.2  setosa        A
6            5.5          3.5           1.3          0.2  setosa        A
12           5.4          3.4           1.5          0.4  setosa        A
```

```
14            4.5          2.3           1.3          0.3  setosa        A
21            4.7          3.2           1.6          0.2  setosa        A

('setosa', 'B')
    sepal_length  sepal_width  petal_length  petal_width species category
3            5.5          4.2           1.4          0.2  setosa        B
15           5.4          3.9           1.7          0.4  setosa        B
27           4.8          3.4           1.9          0.2  setosa        B
28           5.0          3.4           1.5          0.2  setosa        B

('setosa', 'C')
   sepal_length  sepal_width  petal_length  petal_width species category
8           4.6          3.4           1.4          0.3  setosa        C

('versicolor', 'A')
    sepal_length  sepal_width  petal_length  petal_width     species category
13           7.0          3.2           4.7          1.4  versicolor        A
23           6.0          2.7           5.1          1.6  versicolor        A
24           6.1          3.0           4.6          1.4  versicolor        A
29           6.6          3.0           4.4          1.4  versicolor        A

('versicolor', 'B')
    sepal_length  sepal_width  petal_length  petal_width     species category
1            6.3          2.3           4.4          1.3  versicolor        B
4            5.8          2.7           4.1          1.0  versicolor        B
7            5.6          3.0           4.5          1.5  versicolor        B
9            5.5          2.4           3.7          1.0  versicolor        B
11           6.8          2.8           4.8          1.4  versicolor        B
18           6.0          3.4           4.5          1.6  versicolor        B
25           6.7          3.1           4.4          1.4  versicolor        B
26           5.6          2.7           4.2          1.3  versicolor        B

('versicolor', 'C')
    sepal_length  sepal_width  petal_length  petal_width     species category
2            5.7          2.8           4.1          1.3  versicolor        C
16           5.9          3.2           4.8          1.8  versicolor        C

('virginica', 'A')
    sepal_length  sepal_width  petal_length  petal_width    species category
10           7.7          3.8           6.7          2.2  virginica        A
17           5.7          2.5           5.0          2.0  virginica        A
20           5.8          2.7           5.1          1.9  virginica        A

('virginica', 'B')
    sepal_length  sepal_width  petal_length  petal_width    species category
22           6.5          3.2           5.1          2.0  virginica        B

('virginica', 'C')
    sepal_length  sepal_width  petal_length  petal_width    species category
19           6.4          2.7           5.3          1.9  virginica        C
```

1.1.3 Selecting Groups

We can select a group by applying a function `GroupBy.get_group()`

In [18]:

```
gp.get_group('setosa')
```

Out[18]:

	sepal_length	sepal_width	petal_length	petal_width	species	category
0	5.1	3.8	1.6	0.2	setosa	A
3	5.5	4.2	1.4	0.2	setosa	B
5	5.0	3.3	1.4	0.2	setosa	A
6	5.5	3.5	1.3	0.2	setosa	A

	sepal_length	sepal_width	petal_length	petal_width	species	category
8	4.6	3.4	1.4	0.3	setosa	C
12	5.4	3.4	1.5	0.4	setosa	A
14	4.5	2.3	1.3	0.3	setosa	A
15	5.4	3.9	1.7	0.4	setosa	B
21	4.7	3.2	1.6	0.2	setosa	A
27	4.8	3.4	1.9	0.2	setosa	B
28	5.0	3.4	1.5	0.2	setosa	B

In [19]:

```
gps.get_group(('versicolor','C'))
```

Out[19]:

	sepal_length	sepal_width	petal_length	petal_width	species	category
2	5.7	2.8	4.1	1.3	versicolor	C
16	5.9	3.2	4.8	1.8	versicolor	C

1.1.4 Applying a Function to a Group

After splitting a data into a group, we apply a function to each group in order to do that we perform some operation they are:

- **Aggregation** : It is a process in which we compute a summary statistic (or statistics) about each group. For Example, Compute group sums or means.
- **Transformation** : It is a process in which we perform some group-specific computations and return a like-indexed. For Example, Filling NAs within groups with a value derived from each group
- **Filtration** : It is a process in which we discard some groups, according to a group-wise computation that evaluates True or False. For Example, Filtering out data based on the group sum or mean

Aggregation

Aggregation is a process in which we compute a summary statistic about each group. Aggregated function returns a single aggregated value for each group. After splitting a data into groups using groupby function, several aggregation operations can be performed on the grouped data.

Applying Single Function

In [20]:

```
gp.aggregate(np.sum)
```

Out[20]:

species	sepal_length	sepal_width	petal_length	petal_width
setosa	55.5	37.8	16.6	2.8
versicolor	85.6	40.3	62.3	19.4

	sepal_length	sepal_width	petal_length	petal_width
species				
virginica	32.1	14.9	27.2	10.0

Applying Multiple Functions

In [21]:

```
gp.aggregate([np.sum,np.mean,np.std])
```

Out[21]:

	sepal_length			sepal_width			petal_length				
	sum	mean	std	sum	mean	std	sum	mean	std	sum	me
species											
setosa	55.5	5.045455	0.367052	37.8	3.436364	0.480151	16.6	1.509091	0.181409	2.8	0.2545
versicolor	85.6	6.114286	0.489674	40.3	2.878571	0.309288	62.3	4.450000	0.354640	19.4	1.3857
virginica	32.1	6.420000	0.798123	14.9	2.980000	0.526308	27.2	5.440000	0.712741	10.0	2.000

Applying Function for each column

In [22]:

```
gp.aggregate({'sepal_length':'sum', 'sepal_width':'mean'})
```

Out[22]:

	sepal_length	sepal_width
species		
setosa	55.5	3.436364
versicolor	85.6	2.878571
virginica	32.1	2.980000

Transformation

Transformation is a process in which we perform some group-specific computations and return a like-indexed. Transform method returns an object that is indexed the same (same size) as the one being grouped. The transform function must:

- Return a result that is either the same size as the group chunk
- Operate column-by-column on the group chunk
- Not perform in-place operations on the group chunk.

Basically, it means creating a function specifically meant for a group and the applying hat function on it.

In [23]:

```
fn = lambda x: (x - x.mean())/(np.max(x)-np.min(x))
```

In [24]:

```
gp.transform(fn).loc[0:5]
```

Out[24]:

sepal_length	sepal_width	petal_length	petal_width

	sepal_length	sepal_width	petal_length	petal_width
0	0.054545	0.191388	0.151515	-0.272727
1	0.123810	-0.525974	-0.035714	-0.107143
2	-0.276190	-0.071429	-0.250000	-0.107143
3	0.454545	0.401914	-0.181818	-0.272727
4	-0.209524	-0.162338	-0.250000	-0.482143
5	-0.045455	-0.071770	-0.181818	-0.272727

Filtration

Filtration is a process in which we discard some groups, according to a group-wise computation that evaluates True or False. Elements from groups are filtered if they do not satisfy the boolean criterion specified by func. In order to `filter` a group, we use filter method and apply some condition by which we filter group.

In [25]:

```
gp.filter(lambda x: x['sepal_width'].mean() < 3.0)
```

Out[25]:

	sepal_length	sepal_width	petal_length	petal_width	species	category
1	6.3	2.3	4.4	1.3	versicolor	B
2	5.7	2.8	4.1	1.3	versicolor	C
4	5.8	2.7	4.1	1.0	versicolor	B
7	5.6	3.0	4.5	1.5	versicolor	B
9	5.5	2.4	3.7	1.0	versicolor	B
10	7.7	3.8	6.7	2.2	virginica	A
11	6.8	2.8	4.8	1.4	versicolor	B
13	7.0	3.2	4.7	1.4	versicolor	A
16	5.9	3.2	4.8	1.8	versicolor	C
17	5.7	2.5	5.0	2.0	virginica	A
18	6.0	3.4	4.5	1.6	versicolor	B
19	6.4	2.7	5.3	1.9	virginica	C
20	5.8	2.7	5.1	1.9	virginica	A
22	6.5	3.2	5.1	2.0	virginica	B
23	6.0	2.7	5.1	1.6	versicolor	A
24	6.1	3.0	4.6	1.4	versicolor	A
25	6.7	3.1	4.4	1.4	versicolor	B
26	5.6	2.7	4.2	1.3	versicolor	B
29	6.6	3.0	4.4	1.4	versicolor	A

1.2 Combining Multiple Columns via GroupBy

In [26]:
```
gp_dict = {'sepal_length':'sepal','sepal_width':'sepal','petal_length':'petal','peta
```

In [27]:
```
gpt = iris.groupby(gp_dict, axis = 1).sum()
```

In [28]:
```
#To calculate the total sum we use .sum() which sums up all the values of the respec
gpt.loc[0:5]
```

Out[28]:

	petal	sepal
0	1.8	8.9
1	5.7	8.6
2	5.4	8.5
3	1.6	9.7
4	5.1	8.5
5	1.6	8.3

---

2. Merging, Joining, Concatenating

We can join, merge, and concat dataframe using different methods. In Dataframe `df.merge()`, `df.join()`, and `df.concat()` methods help in joining, merging and concatenating different dataframes.

2.1 Concatenating

In order to concat dataframe, we use concat() function which helps in concatenating a dataframe. We can concat a dataframe in many different ways, they are:

- Concatenating DataFrame using `.concat()`
- Concatenating DataFrame by setting logic on axes
- Concatenating DataFrame using `.append()`
- Concatenating DataFrame by ignoring indexes
- Concatenating DataFrame with group keys
- Concatenating with mixed `ndims`

Preparing DataSet for Operations

In [29]:
```
df1 = iris.sample(10)
```

In [30]:
```
df1.sort_index(inplace = True)
```

In [31]:
```
df1
```

Out[31]:

	sepal_length	sepal_width	petal_length	petal_width	species	category
0	5.1	3.8	1.6	0.2	setosa	A
2	5.7	2.8	4.1	1.3	versicolor	C
3	5.5	4.2	1.4	0.2	setosa	B
4	5.8	2.7	4.1	1.0	versicolor	B
6	5.5	3.5	1.3	0.2	setosa	A
9	5.5	2.4	3.7	1.0	versicolor	B
16	5.9	3.2	4.8	1.8	versicolor	C
19	6.4	2.7	5.3	1.9	virginica	C
20	5.8	2.7	5.1	1.9	virginica	A
21	4.7	3.2	1.6	0.2	setosa	A

In [32]:
```
df2 = iris.loc[24:30]
```

In [33]:
```
df2
```

Out[33]:

	sepal_length	sepal_width	petal_length	petal_width	species	category
24	6.1	3.0	4.6	1.4	versicolor	A
25	6.7	3.1	4.4	1.4	versicolor	B
26	5.6	2.7	4.2	1.3	versicolor	B
27	4.8	3.4	1.9	0.2	setosa	B
28	5.0	3.4	1.5	0.2	setosa	B
29	6.6	3.0	4.4	1.4	versicolor	A

In [34]:
```
df1.reset_index(inplace = True, drop = True)
```

In [35]:
```
df2.reset_index(inplace = True, drop = True)
```

In [36]:
```
print(df1.loc[:5])
print()
print(df2.loc[:5])
```

```
   sepal_length  sepal_width  petal_length  petal_width     species category
0           5.1          3.8           1.6          0.2      setosa        A
1           5.7          2.8           4.1          1.3  versicolor        C
2           5.5          4.2           1.4          0.2      setosa        B
3           5.8          2.7           4.1          1.0  versicolor        B
4           5.5          3.5           1.3          0.2      setosa        A
5           5.5          2.4           3.7          1.0  versicolor        B
```

```
   sepal_length  sepal_width  petal_length  petal_width     species category
0           6.1          3.0           4.6          1.4  versicolor        A
1           6.7          3.1           4.4          1.4  versicolor        B
2           5.6          2.7           4.2          1.3  versicolor        B
3           4.8          3.4           1.9          0.2      setosa        B
4           5.0          3.4           1.5          0.2      setosa        B
5           6.6          3.0           4.4          1.4  versicolor        A
```

2.1.1 Concatenating DataFrame using $.concat()$

In [37]:

```
pd.concat([df1,df2])
```

Out[37]:

	sepal_length	sepal_width	petal_length	petal_width	species	category
0	5.1	3.8	1.6	0.2	setosa	A
1	5.7	2.8	4.1	1.3	versicolor	C
2	5.5	4.2	1.4	0.2	setosa	B
3	5.8	2.7	4.1	1.0	versicolor	B
4	5.5	3.5	1.3	0.2	setosa	A
5	5.5	2.4	3.7	1.0	versicolor	B
6	5.9	3.2	4.8	1.8	versicolor	C
7	6.4	2.7	5.3	1.9	virginica	C
8	5.8	2.7	5.1	1.9	virginica	A
9	4.7	3.2	1.6	0.2	setosa	A
0	6.1	3.0	4.6	1.4	versicolor	A
1	6.7	3.1	4.4	1.4	versicolor	B
2	5.6	2.7	4.2	1.3	versicolor	B
3	4.8	3.4	1.9	0.2	setosa	B
4	5.0	3.4	1.5	0.2	setosa	B
5	6.6	3.0	4.4	1.4	versicolor	A

2.1.2 Concatenating DataFrame by setting logic on axes

In order to concat dataframe, we have to set different logic on axes. We can set axes in the following two ways:

- Taking the *union* of them all, `join = 'outer'`. This is the default option as it results in zero information loss.
- Taking the *intersection*, `join = 'inner'`.

In [38]:

```
df3 = df1.loc[0:4]
df4 = iris.loc[3:7]
print(df3)
print()
print(df4)
```

```
   sepal_length  sepal_width  petal_length  petal_width     species category
```

```
0           5.1          3.8           1.6          0.2      setosa        A
1           5.7          2.8           4.1          1.3  versicolor        C
2           5.5          4.2           1.4          0.2      setosa        B
3           5.8          2.7           4.1          1.0  versicolor        B
4           5.5          3.5           1.3          0.2      setosa        A

   sepal_length  sepal_width  petal_length  petal_width     species  category
3           5.5          4.2           1.4          0.2      setosa        B
4           5.8          2.7           4.1          1.0  versicolor        B
5           5.0          3.3           1.4          0.2      setosa        A
6           5.5          3.5           1.3          0.2      setosa        A
7           5.6          3.0           4.5          1.5  versicolor        B
```

In [39]:

```python
pd.concat([df3, df4], join = 'inner', axis = 1)
```

Out[39]:

	sepal_length	sepal_width	petal_length	petal_width	species	category	sepal_length	sepal_wid
3	5.8	2.7	4.1	1.0	versicolor	B	5.5	4
4	5.5	3.5	1.3	0.2	setosa	A	5.8	2

In [40]:

```python
pd.concat([df3, df4], join = 'outer', axis = 1)
```

Out[40]:

	sepal_length	sepal_width	petal_length	petal_width	species	category	sepal_length	sepal_wid
0	5.1	3.8	1.6	0.2	setosa	A	NaN	Na
1	5.7	2.8	4.1	1.3	versicolor	C	NaN	Na
2	5.5	4.2	1.4	0.2	setosa	B	NaN	Na
3	5.8	2.7	4.1	1.0	versicolor	B	5.5	4
4	5.5	3.5	1.3	0.2	setosa	A	5.8	2
5	NaN	NaN	NaN	NaN	NaN	NaN	5.0	3
6	NaN	NaN	NaN	NaN	NaN	NaN	5.5	3
7	NaN	NaN	NaN	NaN	NaN	NaN	5.6	3

2.1.3 Concatenating DataFrame using $.append()$

In order to concat a dataframe, we also use `.append()` function along axis=0, namely the index. This function existed before `.concat`. In case of collection of indices, all of them gets appended to the original index in the same order as they are passed to the `idx.append()` function. The function returns an appended index.

In [41]:

```python
df3.append(df4)
```

Out[41]:

	sepal_length	sepal_width	petal_length	petal_width	species	category
0	5.1	3.8	1.6	0.2	setosa	A
1	5.7	2.8	4.1	1.3	versicolor	C
2	5.5	4.2	1.4	0.2	setosa	B
3	5.8	2.7	4.1	1.0	versicolor	B

	sepal_length	sepal_width	petal_length	petal_width	species	category
4	5.5	3.5	1.3	0.2	setosa	A
3	5.5	4.2	1.4	0.2	setosa	B
4	5.8	2.7	4.1	1.0	versicolor	B
5	5.0	3.3	1.4	0.2	setosa	A
6	5.5	3.5	1.3	0.2	setosa	A
7	5.6	3.0	4.5	1.5	versicolor	B

Ignore the indices by passing the `ignore_index` parameter as True.

In [42]:
```
df3.append(df4, ignore_index = True)
```

Out[42]:

	sepal_length	sepal_width	petal_length	petal_width	species	category
0	5.1	3.8	1.6	0.2	setosa	A
1	5.7	2.8	4.1	1.3	versicolor	C
2	5.5	4.2	1.4	0.2	setosa	B
3	5.8	2.7	4.1	1.0	versicolor	B
4	5.5	3.5	1.3	0.2	setosa	A
5	5.5	4.2	1.4	0.2	setosa	B
6	5.8	2.7	4.1	1.0	versicolor	B
7	5.0	3.3	1.4	0.2	setosa	A
8	5.5	3.5	1.3	0.2	setosa	A
9	5.6	3.0	4.5	1.5	versicolor	B

2.1.4 Concatenating DataFrame by ignoring indexes

In order to concat a dataframe by ignoring indexes, we ignore index which don't have a meaning, i.e., you may wish to append them and ignore the fact that they may have overlapping indexes. In order to do that we use `ignore_index` as an argument.

In [43]:
```
pd.concat([df3,df4], ignore_index = True)
```

Out[43]:

	sepal_length	sepal_width	petal_length	petal_width	species	category
0	5.1	3.8	1.6	0.2	setosa	A
1	5.7	2.8	4.1	1.3	versicolor	C
2	5.5	4.2	1.4	0.2	setosa	B
3	5.8	2.7	4.1	1.0	versicolor	B
4	5.5	3.5	1.3	0.2	setosa	A
5	5.5	4.2	1.4	0.2	setosa	B
6	5.8	2.7	4.1	1.0	versicolor	B

	sepal_length	sepal_width	petal_length	petal_width	species	category
7	5.0	3.3	1.4	0.2	setosa	A
8	5.5	3.5	1.3	0.2	setosa	A
9	5.6	3.0	4.5	1.5	versicolor	B

2.1.5 Concatenating DataFrame with group keys

In order to concat dataframe with group keys, we override the column names with the use of the keys argument. Keys argument is used to override the column names when creating a new DataFrame based on existing Series.

In [44]:

```
pd.concat([df3,df4], keys = ['X','Y'])
```

Out[44]:

		sepal_length	sepal_width	petal_length	petal_width	species	category
X	**0**	5.1	3.8	1.6	0.2	setosa	A
	1	5.7	2.8	4.1	1.3	versicolor	C
	2	5.5	4.2	1.4	0.2	setosa	B
	3	5.8	2.7	4.1	1.0	versicolor	B
	4	5.5	3.5	1.3	0.2	setosa	A
Y	**3**	5.5	4.2	1.4	0.2	setosa	B
	4	5.8	2.7	4.1	1.0	versicolor	B
	5	5.0	3.3	1.4	0.2	setosa	A
	6	5.5	3.5	1.3	0.2	setosa	A
	7	5.6	3.0	4.5	1.5	versicolor	B

2.1.6 Concatenating with mixed ndims

User can concatenate a mix of Series and DataFrame. The Series will be transformed to DataFrame with the column name as the name of the Series.

In [45]:

```
sr = pd.Series(np.random.randint(100, size = 5), name = 'rad_level')
```

In [46]:

```
pd.concat([df3, sr], axis = 1)
```

Out[46]:

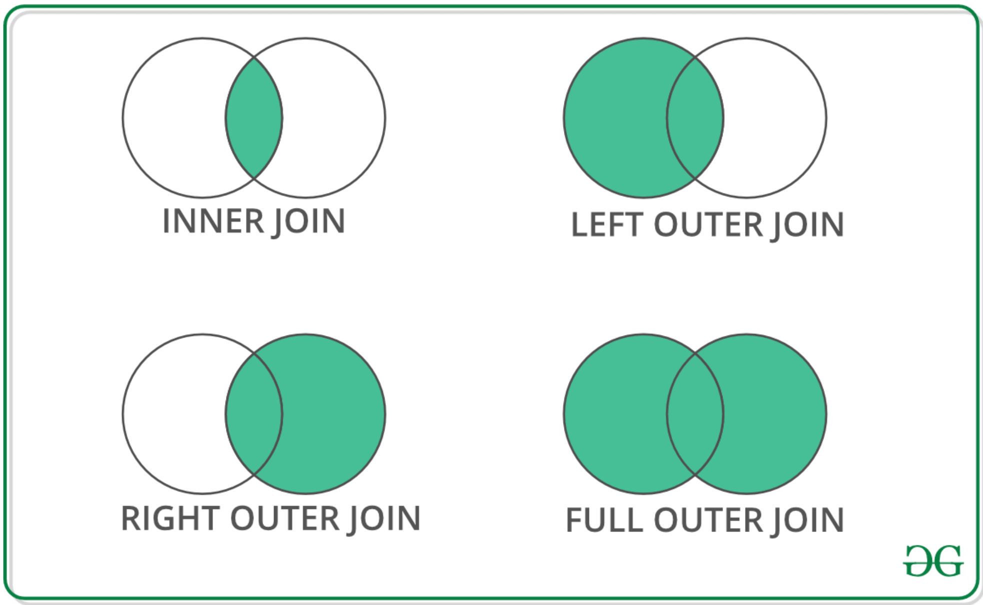
	sepal_length	sepal_width	petal_length	petal_width	species	category	rad_level
0	5.1	3.8	1.6	0.2	setosa	A	19
1	5.7	2.8	4.1	1.3	versicolor	C	23
2	5.5	4.2	1.4	0.2	setosa	B	61
3	5.8	2.7	4.1	1.0	versicolor	B	52
4	5.5	3.5	1.3	0.2	setosa	A	95

which is essentially same as `df3['rad_level'] = np.random.randint(100, size = 5)`

2.2 Merging

Pandas have options for high-performance in-memory merging and joining. When we need to combine very large DataFrames, joins serve as a powerful way to perform these operations swiftly. Joins can only be done on two DataFrames at a time, **denoted as *left* and *right* tables**. The *key* is the common column that the two DataFrames will be joined on. *It's a good practice to use keys which have unique values throughout the column to avoid unintended duplication of row values*. Pandas provide a single function, `merge()`, as the entry point for all standard database join operations between DataFrame objects. There are four basic ways to handle the join (inner, left, right, and outer), depending on which rows must retain their data.

There are four basic ways to handle the join (inner, left, right, and outer), depending on which rows must retain their data.



Preparing DataSet for Operations

In [47]:
```
df5 = pd.DataFrame({"id":['M','N','O','P','Q'], "sn":['A1','A1','A2','A3','A1'], 'co
                    'pattern':np.random.choice(a = ['st','r','sp'], size = 5, p = [0
```

In [48]:
```
df3.insert(0, "id", ['M','N','O','P','Q'])
```

In [49]:
```
df3.insert(1, "sn", ['A1','A2','A3','A4','A5'])
```

In [50]:
```
print(df3)
print()
print(df5)
```

```
  id  sn  sepal_length  sepal_width  petal_length  petal_width     species  \
0  M  A1           5.1          3.8           1.6          0.2      setosa
1  N  A2           5.7          2.8           4.1          1.3  versicolor
2  O  A3           5.5          4.2           1.4          0.2      setosa
```

```
3  P  A4          5.8         2.7         4.1         1.0  versicolor
4  Q  A5          5.5         3.5         1.3         0.2      setosa

  category
0        A
1        C
2        B
3        B
4        A

  id  sn color pattern
0  M  A1     R      st
1  N  A1     Y      st
2  O  A2     G      sp
3  P  A3     B       r
4  Q  A1     P      st
```

Using Single Key

In [51]:

```
pd.merge(df3, df5, on = 'id')
```

Out[51]:

	id	sn_x	sepal_length	sepal_width	petal_length	petal_width	species	category	sn_y	color
0	M	A1	5.1	3.8	1.6	0.2	setosa	A	A1	R
1	N	A2	5.7	2.8	4.1	1.3	versicolor	C	A1	Y
2	O	A3	5.5	4.2	1.4	0.2	setosa	B	A2	G
3	P	A4	5.8	2.7	4.1	1.0	versicolor	B	A3	B
4	Q	A5	5.5	3.5	1.3	0.2	setosa	A	A1	P

Using Multiple Keys

In [52]:

```
pd.merge(df3, df5, on = ['id','sn'])
```

Out[52]:

	id	sn	sepal_length	sepal_width	petal_length	petal_width	species	category	color	pattern
0	M	A1	5.1	3.8	1.6	0.2	setosa	A	R	st

Using `how` argument

We use how argument to merge specifies how to determine which keys are to be included in the resulting table. If a key combination does not appear in either the left or right tables, the values in the joined table will be NA. Here is a summary of the how options and their SQL equivalent names:

Merge Method	Join Name	Description
left	LEFT OUTER JOIN	Use keys from left frame only
right	RIGHT OUTER JOIN	Use keys from right frame only
outer	FULL OUTER JOIN	Use union of keys from both frames
inner	INNNER JOIN	Use intersection of keys from both frames

In [53]:

```
pd.merge(df3, df5, how = 'left', on = ['id','sn'])
```

Out[53]:

	id	sn	sepal_length	sepal_width	petal_length	petal_width	species	category	color	pattern
0	M	A1	5.1	3.8	1.6	0.2	setosa	A	R	st
1	N	A2	5.7	2.8	4.1	1.3	versicolor	C	NaN	NaN
2	O	A3	5.5	4.2	1.4	0.2	setosa	B	NaN	NaN
3	P	A4	5.8	2.7	4.1	1.0	versicolor	B	NaN	NaN
4	Q	A5	5.5	3.5	1.3	0.2	setosa	A	NaN	NaN

In [54]:

```
pd.merge(df3, df5, how = 'right', on = ['id','sn'])
```

Out[54]:

	id	sn	sepal_length	sepal_width	petal_length	petal_width	species	category	color	pattern
0	M	A1	5.1	3.8	1.6	0.2	setosa	A	R	st
1	N	A1	NaN	NaN	NaN	NaN	NaN	NaN	Y	st
2	O	A2	NaN	NaN	NaN	NaN	NaN	NaN	G	sp
3	P	A3	NaN	NaN	NaN	NaN	NaN	NaN	B	r
4	Q	A1	NaN	NaN	NaN	NaN	NaN	NaN	P	st

In [55]:

```
pd.merge(df3, df5, how = 'outer', on = ['id','sn'])
```

Out[55]:

	id	sn	sepal_length	sepal_width	petal_length	petal_width	species	category	color	pattern
0	M	A1	5.1	3.8	1.6	0.2	setosa	A	R	st
1	N	A2	5.7	2.8	4.1	1.3	versicolor	C	NaN	NaN
2	O	A3	5.5	4.2	1.4	0.2	setosa	B	NaN	NaN
3	P	A4	5.8	2.7	4.1	1.0	versicolor	B	NaN	NaN
4	Q	A5	5.5	3.5	1.3	0.2	setosa	A	NaN	NaN
5	N	A1	NaN	NaN	NaN	NaN	NaN	NaN	Y	st
6	O	A2	NaN	NaN	NaN	NaN	NaN	NaN	G	sp
7	P	A3	NaN	NaN	NaN	NaN	NaN	NaN	B	r
8	Q	A1	NaN	NaN	NaN	NaN	NaN	NaN	P	st

In [56]:

```
pd.merge(df3, df5, how = 'inner', on = ['id','sn'])
```

Out[56]:

	id	sn	sepal_length	sepal_width	petal_length	petal_width	species	category	color	pattern
0	M	A1	5.1	3.8	1.6	0.2	setosa	A	R	st

2.3 Joining

In order to join dataframe, we use `.join()` function. This function is used for combining the *columns of two potentially differently-indexed DataFrames* into a single result DataFrame. It results in a ValueError if both the keys are complete in both the dataframes.

Preapring Dataset for Operations

In [57]:
```
data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'], 'Age':[27, 24, 22, 32]}
data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kannuaj'],
         'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}
df8 = pd.DataFrame(data1,index=['K0', 'K1', 'K2', 'K3'])
df9 = df1 = pd.DataFrame(data2, index=['K0', 'K2', 'K3', 'K4'])
```

In [58]:
```
print(df8)
print()
print(df9)
```

```
      Name  Age
K0     Jai   27
K1  Princi   24
K2  Gaurav   22
K3    Anuj   32

      Address Qualification
K0  Allahabad           MCA
K2    Kannuaj           Phd
K3  Allahabad          Bcom
K4    Kannuaj        B.hons
```

In [59]:
```
df8.join(df9)
```

Out[59]:

	Name	Age	Address	Qualification
K0	Jai	27	Allahabad	MCA
K1	Princi	24	NaN	NaN
K2	Gaurav	22	Kannuaj	Phd
K3	Anuj	32	Allahabad	Bcom

Using `on` argument

In order to join dataframes we can also use `on` in an argument. `join()` takes this optional `on` argument which may be a column or multiple column names, which specifies that the passed DataFrame is to be aligned on that column in the DataFrame.

In [60]:
```
data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'], 'Age':[27, 24, 22, 32],
         'Key':['K0', 'K1', 'K2', 'K3']}
data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kannuaj'],
        'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}
df6 = pd.DataFrame(data1)
df7 = pd.DataFrame(data2, index=['K0', 'K2', 'K3', 'K4'])
print(df6)
print()
print(df7)
```

```
     Name  Age Key
0     Jai   27  K0
1  Princi   24  K1
2  Gaurav   22  K2
```

```
3     Anuj   32  K3

        Address Qualification
K0  Allahabad           MCA
K2    Kannuaj           Phd
K3  Allahabad          Bcom
K4    Kannuaj        B.hons
```

In [61]:

```
df6.join(df7, on='Key')
```

Out[61]:

	Name	Age	Key	Address	Qualification
0	Jai	27	K0	Allahabad	MCA
1	Princi	24	K1	NaN	NaN
2	Gaurav	22	K2	Kannuaj	Phd
3	Anuj	32	K3	Allahabad	Bcom

Joining singly-indexed DataFrame with multi-indexed DataFrame

In order to join singly indexed dataframe with multi-indexed dataframe, the level will match on the name of the index of the singly-indexed frame against a level name of the multi-indexed frame.

In [62]:

```
data1 = {'Name':['Jai', 'Princi', 'Gaurav'], 'Age':[27, 24, 22]}
data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kanpur'],
        'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}
df10 = pd.DataFrame(data1, index=pd.Index(['K0', 'K1', 'K2'], name='key'))

index = pd.MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'),
                                   ('K2', 'Y2'), ('K2', 'Y3')],
                                   names=['key', 'Y'])

df11 = pd.DataFrame(data2, index= index)
print(df10)
print()
print(df11)
```

```
       Name  Age
key
K0      Jai   27
K1   Princi   24
K2   Gaurav   22

           Address Qualification
key Y
K0  Y0  Allahabad           MCA
K1  Y1    Kannuaj           Phd
K2  Y2  Allahabad          Bcom
    Y3     Kanpur        B.hons
```

In [63]:

```
df10.join(df11, how = 'inner')
```

Out[63]:

		Name	Age	Address	Qualification
key	**Y**				
K0	**Y0**	Jai	27	Allahabad	MCA
K1	**Y1**	Princi	24	Kannuaj	Phd

		Name	Age	Address	Qualification
key	Y				
K2	Y2	Gaurav	22	Allahabad	Bcom
	Y3	Gaurav	22	Kanpur	B.hons

---

3. Concatenating Series Strings

Pandas `str.cat()` is used to concatenate strings to the passed caller series of string. Distinct values from a different series can be passed but the length of both the series has to be same. `str` has to be prefixed to differentiate it from the Python's default method.

Method 1

In [64]:
```
kl = pd.DataFrame(iris['species'].str.cat(iris['category'], sep = " | "))
```

In [65]:
```
kl.columns = ['species | category']
```

In [66]:
```
kl.head()
```

Out[66]:

	species | category
0	setosa | A
1	versicolor | B
2	versicolor | C
3	setosa | B
4	versicolor | B

`str.cat()` provides a way to handle null values through `na_rep` parameter. Whatever is passed to this parameter will be replaced at every occurrence of null value.

Method 2

Using + operator : We need to ensure data frame elements into string before join. We can also use different separators during join, e.g. -, _, ' ' etc.

In [67]:
```
(iris['species'] + " " + iris['category']).loc[:5]
```

Out[67]:
```
0        setosa A
1    versicolor B
2    versicolor C
3        setosa B
4    versicolor B
5        setosa A
dtype: object
```

3.1 Combining Series

Pandas `Series.combine()` is a series mathematical operation method. This is used to combine two series into one. The shape of output series is same as the caller series. The elements are decided by a function passed as parameter to `combine()` method. The shape of both series has to be same otherwise it will throw an error.

Syntax: `Series.combine(other, func, fill_value=nan)`

In [68]:

```
k1 = pd.Series(np.random.choice(a = ['A','K','C','M','E'], size = 20, p = [0.2,0.2,0
k2 = pd.Series(np.random.choice(a = ['F','D','H','I','J'], size = 20, p = [0.2,0.2,0
```

In [69]:

```
print(k1.loc[:5])
print()
print(k2.loc[:5])
```

```
0    M
1    C
2    M
3    M
4    E
5    A
dtype: object

0    F
1    H
2    H
3    I
4    J
5    H
dtype: object
```

In [70]:

```
k1.combine(k2, lambda x1,x2: x1 if x1 > x2 else x2)
```

Out[70]:

```
0     M
1     H
2     M
3     M
4     J
5     H
6     I
7     I
8     I
9     I
10    K
11    K
12    K
13    D
14    F
15    J
16    M
17    H
18    D
19    M
dtype: object
```

3.2 Join All Elements in List Present in Series

Pandas `str.join()` method is used to join all elements in list present in a series with passed delimiter. Since strings are also array of character (or List of characters), hence when this method is applied on a series of strings, the string is joined at every character with the passed delimiter.

`.str` has to be prefixed every time before calling this method to differentiate it from the Python's default string method.

In [71]:
```
iris['species'].str.join("-").loc[:5]
```

Out[71]:
```
0          s-e-t-o-s-a
1    v-e-r-s-i-c-o-l-o-r
2    v-e-r-s-i-c-o-l-o-r
3          s-e-t-o-s-a
4    v-e-r-s-i-c-o-l-o-r
5          s-e-t-o-s-a
Name: species, dtype: object
```

Split after a specific character

In [72]:
```
iris['species'].str.split("t").loc[:5]
```

Out[72]:
```
0       [se, osa]
1    [versicolor]
2    [versicolor]
3       [se, osa]
4    [versicolor]
5       [se, osa]
Name: species, dtype: object
```

In [73]:
```
iris['species'].str.join("_").loc[:5]
```

Out[73]:
```
0          s_e_t_o_s_a
1    v_e_r_s_i_c_o_l_o_r
2    v_e_r_s_i_c_o_l_o_r
3          s_e_t_o_s_a
4    v_e_r_s_i_c_o_l_o_r
5          s_e_t_o_s_a
Name: species, dtype: object
```

CONTD....