Operating on Data in Pandas

One of the **strengths of NumPy** is that it allows us to perform **quick element-wise** operations, **both** with

basic arithmetic (addition, subtraction, multiplication, etc.)

and with more **complicated operations** (trigonometric functions, exponential and logarithmic functions, etc.).

Pandas inherits much of this functionality from NumPy, and the **ufuncs** introduced in **Computation on NumPy Arrays: Universal Functions** are **key** to this.

Pandas includes a **couple of useful twists,** however:

for **unary operations** like negation and trigonometric functions, these ufuncs will **preserve index and column labels** in the output,

and for **binary operations** such as addition and multiplication, Pandas will automatically **align indices** when passing the objects to the ufunc.

This means that **keeping the context of data** and **combining data from different sources** — both potentially error-prone tasks with raw NumPy arrays — become essentially **foolproof with Pandas.**

We will **additionally** see that there are **well-defined operations** between one-dimensional `Series` structures and two-dimensional `DataFrame` structures.

Ufuncs: Index Preservation

Because **Pandas is designed to work with NumPy,** any **NumPy ufunc will work on Pandas** `Series` and `DataFrame` objects.

Let's start by defining a simple `Series` and `DataFrame` on which to **demonstrate** this:

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

In [2]:
```
rng = np.random.default_rng(42)
ser = pd.Series(rng.integers(0, 10, 4))
ser
```

Out[2]:

	0
0	0
1	7
2	6
3	4

dtype: int64

In [3]:
```
df = pd.DataFrame(rng.integers(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
df
```

Out[3]:

	A	B	C	D
0	4	8	0	6
1	2	0	5	9
2	7	7	7	7

If we **apply a NumPy ufunc on either of these objects,** the result will be another **Pandas object with the indices preserved:**

In [4]:
```
np.exp(ser)
```

Out[4]:

	0
0	1.000000
1	1096.633158
2	403.428793
3	54.598150

dtype: float64

This is **true** also for **more involved sequences of operations:**

In [ ]:
```
np.sin(df * np.pi / 4)
```

Out[ ]:

	A	B	C	D
0	1.224647e-16	-2.449294e-16	0.000000	-1.000000
1	1.000000e+00	0.000000e+00	-0.707107	0.707107
2	-7.071068e-01	-7.071068e-01	-0.707107	-0.707107

Any of the ufuncs discussed in **Computation on NumPy Arrays: Universal Functions** can be used in a **similar manner.**

Ufuncs: Index Alignment

For **binary operations** on two `Series` or `DataFrame` objects, **Pandas will align indices** in the process of performing the operation.

This is very **convenient** when working with **incomplete data,** as we'll see in some of the examples that follow.

Index Alignment in Series

As an **example,** suppose we are **combining two different data sources** and wish to **find only the top three US states** by **area** and the top three US states by **population**:

In [5]:
```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                  'California': 423967}, name='area')
population = pd.Series({'California': 39538223, 'Texas': 2914
                        'Florida': 21538187}, name='populatio
```

Let's see **what happens when we divide** these to compute the population **density:**

In [6]:
```
population / area
```

Out[6]:

	0
Alaska	NaN
California	93.257784
Florida	NaN
Texas	41.896072

dtype: float64

The **resulting array** contains the **union of indices** of the two input arrays, which could be determined directly from these indices:

In [7]:
```
area.index.union(population.index)
```

Out[7]:
```
Index(['Alaska', 'California', 'Florida', 'Texas'], dtype='o
bject')
```

Any item for which one or the other **does not have an entry** is marked with `NaN`, or **"Not a Number,"**

which is **how Pandas marks missing data** (see further discussion of missing data in Handling Missing Data).

This index matching is implemented this way for **any of Python's built-in arithmetic expressions;** any **missing values are marked by `NaN`:**

In [10]:
```
A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])

print(A)
print(B)
```

```
0    2
1    4
2    6
dtype: int64
1    1
2    3
3    5
dtype: int64
```

In [9]:
```
A + B
```

Out[9]:

	0
0	NaN
1	5.0
2	9.0
3	NaN

dtype: float64

If using `NaN` values is **not the desired behavior,** the **fill value can be modified** using appropriate object methods in place of the operators.

For example, calling `A.add(B)` is equivalent to calling `A + B`, but allows optional **explicit specification of the fill value** for any elements in `A` or `B` that might be missing:

In [11]:
```
A.add(B, fill_value=0)
```

Out[11]:

	0
0	2.0
1	5.0
2	9.0
3	5.0

dtype: float64

Index Alignment in DataFrames

A **similar type of alignment** takes place for **both** columns and indices when performing operations on `DataFrame` **objects:**

In [12]:
```
A = pd.DataFrame(rng.integers(0, 20, (2, 2)),
                 columns=['a', 'b'])
A
```

Out[12]:

	a	b
0	10	2
1	16	9

In [13]:
```
B = pd.DataFrame(rng.integers(0, 10, (3, 3)),
                 columns=['b', 'a', 'c'])
B
```

Out[13]:

	b	a	c
0	5	3	1
1	9	7	6
2	4	8	5

In [14]:
```
A + B
```

Out[14]:

	a	b	c
0	13.0	7.0	NaN
1	23.0	18.0	NaN
2	NaN	NaN	NaN

Notice that indices are **aligned** correctly **irrespective of their order** in the two objects, and indices in the result are sorted.

As was the case with `Series`, we can use the associated object's arithmetic methods and pass any **desired** `fill_value` to be used in place of **missing entries.**

Here we'll **fill** with the **mean of all values in** `A`**:**

In [16]:
```
A.values.mean()
```

Out[16]:
```
9.25
```

In [15]:
```
A.add(B, fill_value=A.values.mean())
```

Out[15]:

	a	b	c
0	13.00	7.00	10.25
1	23.00	18.00	15.25
2	17.25	13.25	14.25

The following table lists **Python operators** and their **equivalent Pandas object methods:**

Python operator	Pandas method(s)
`+`	`add`
`-`	`sub`, `subtract`
`*`	`mul`, `multiply`
`/`	`truediv`, `div`, `divide`

Python operator	Pandas method(s)
`//`	`floordiv`
`%`	`mod`
`**`	`pow`

Ufuncs: Operations Between DataFrames and Series

When performing **operations between a `DataFrame` and a `Series`**, the **index and column alignment** is similarly maintained,

and the **result is similar** to operations **between a two-dimensional and one-dimensional NumPy array.**

Consider one **common operation,** where we find the **difference of a two-dimensional array and one of its rows:**

In [17]:
```
A = rng.integers(10, size=(3, 4))
A
```

Out[17]:
```
array([[4, 4, 2, 0],
       [5, 8, 0, 8],
       [8, 2, 6, 1]])
```

In [18]:
```
A - A[0]
```

Out[18]:
```
array([[ 0,  0,  0,  0],
       [ 1,  4, -2,  8],
       [ 4, -2,  4,  1]])
```

According to NumPy's broadcasting rules (see **Computation on Arrays: Broadcasting**),

subtraction between a two-dimensional array and one of its rows is applied row-wise.

In **Pandas,** the convention **similarly operates row-wise by default:**

In [20]:
```
df = pd.DataFrame(A, columns=['Q', 'R', 'S', 'T'])
df
```

Out[20]:

	Q	R	S	T
0	4	4	2	0
1	5	8	0	8
2	8	2	6	1

In [21]:
```
df - df.iloc[0]
```

Out[21]:

	Q	R	S	T
0	0	0	0	0
1	1	4	-2	8
2	4	-2	4	1

If you would **instead** like to operate **column-wise,** you can use the object methods mentioned earlier, while specifying the `axis` **keyword:**

In [22]:
```
df.subtract(df['R'], axis=0)
```

Out[22]:

	Q	R	S	T
0	0	0	-2	-4
1	-3	0	-8	0
2	6	0	4	-1

Note that these `DataFrame` / `Series` **operations,** like the operations discussed previously, will **automatically align indices between the two elements:**

In [25]:
```
df
```

Out[25]:

	Q	R	S	T
0	4	4	2	0
1	5	8	0	8
2	8	2	6	1

In [30]:
```
halfrow = df.iloc[0, ::2]
halfrow
```

Out[30]:

	0
Q	4
S	2

dtype: int64

In [34]:
```
df.iloc[0, ::2]
```

Out[34]:

	0
Q	4
S	2

dtype: int64

In [35]:
```
df - halfrow
```

Out[35]:

	Q	R	S	T
0	0.0	NaN	0.0	NaN
1	1.0	NaN	-2.0	NaN
2	4.0	NaN	4.0	NaN

This **preservation** and **alignment of indices and columns** means that **operations on data in Pandas will always maintain the data context,**

which **prevents the common errors** that might arise when **working with heterogeneous and/or misaligned data in raw NumPy arrays.**