OPERATING ON DATA

Pandas inherits much of this functionality from NumPy, and the ufuncs. So Pandas having the ability to perform quick element-wise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) and with more sophisticated operations (trigonometric functions, exponential and logarithmic functions, etc.).

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

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

Here we are going to see how the universal functions are working in series and DataFrames by

•Index preservation

•Index alignment

Index Preservation

Pandas is designed to work with NumPy, any NumPy ufunc will work on Pandas Series and DataFrame objects. We can use all arithmetic and special universal functions as in NumPy on pandas. In outputs the index will preserved (maintained) as shown below.

For series

```
x=pd.Series([1,2,3,4]) x

0 1

1 2

2 3

3 4

dtype: int64
```

For DataFrame

```
df=pd.DataFrame(np.random.randint(0,10,(3,4)),

columns=['a','b','c','d'])

 df
```

Index Alignment

Pandas will align indices in the process of performing the operation. This is very convenient when you are working with incomplete data, as we’ll.

Index alignment in Series

suppose we are combining two different data sources, then the index will aligned accordingly.

```
x=pd.Series([2,4,6],index=[1,3,5])

y=pd.Series([1,3,5,7],index=[1,2,3,4]) x+y

1 3.0

2 NaN 3 9.0

4        NaN

5        NaN

dtype: float64
```

The resulting array contains the union of indices of the two input arrays, which we could determine using standard Python set arithmetic on these indices.

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 as missing data.

Fill value in missing data (fill_value)

If using NaN values is not the desired behavior, we can modify the fill value using appropriate object methods in place of the operators.

```
x.add(y,fill_value=0)

1 3.0

2 3.0

3 9.0

4 7.0

5 6.0

dtype: float64
```

Index alignment in DataFrame

A similar type of alignment takes place for both columns and indices when you are performing operations on DataFrames.

```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),columns=list('AB')) A
```

```
A B 0 1 11

1 5 1

B = pd.DataFrame(rng.randint(0, 10, (3, 3)), columns=list('BAC'))

B

B A C 0 4 0 9

1 5 8 0

2 9 2 6

A + B

A         B         C

0 1.0 15.0 NaN

1 13.0 6.0 NaN 2 NaN NaN NaN
```

HANDLING MISSING DATA

A number of schemes have been developed to indicate the presence of missing data in a table or DataFrame. Generally, they revolve around one of two strategies: using a mask that globally indicates missing values, or choosing a sentinel value that indicates a missing entry.

In the masking approach, the mask might be an entirely separate Boolean array, or it may involve appropriation of one bit in the data representation to locally indicate the null status of a value.

In the sentinel approach, the sentinel value could be some data-specific convention, such as indicating a missing integer value with –9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating-point value with NaN (Not a Number), a special value which is part of the IEEE floating-point specification.

Missing Data in Pandas

The way in which Pandas handles missing values is constrained by its NumPy package, which does not have a built-in notion of NA values for non floating- point data types.

NumPy supports fourteen basic integer types once you account for available precisions, signedness, and endianness of the encoding. Reserving a specific bit pattern in all available NumPy types would lead to an unwieldy amount of overhead in special-casing various operations for various types, likely even requiring a new fork of the NumPy package.

Panas chose to use sentinels for missing data, and further chose to use two already-existing Python null values: the special floatingpoint NaN value, and the Python None object. This choice has some side effects, as we will see, but in practice ends up being a good compromise in most cases of interest.

Operating on Null Values

there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:

•isnull() - Generate a Boolean mask indicating missing values

•notnull() - Opposite of isnull()

•dropna() - Return a filtered version of the data

•fillna() - Return a copy of the data with missing values filled or imputed