* Numpy ufunc :

Most ufuncs work with NA and generally return NA.

```
np.log(Pd.NA)
```

o/p : <NA>

```
np.add(Pd.NA, 1)
```

o/p : <NA>

```
a = np.array([1,2,3])
np.greater(a, Pd.NA)
```

element wise comparsion

o/p : array([<NA>, <NA>, <NA>], dtype = object)

* Conversion : If you have a "Dataframe / Series" using np.nan, Series.convert_dtypes() and "DataFrame.convert_dtypes()" in "DataFrame" that can convert data to use the data types that use "NA" such as "Int64Dtype" (or) "ArrowDtype".

```
Import io
Import Pandas as Pd
data = io.StringIO("a,b\n,True\n2")
df = Pd.read_csv(data)
df.dtypes
```

o/p :

a	float64
b	object
dtype:	object

```
df_conv = df.convert_dtypes()

df_conv          a       b
```

o/p : 0 <NA> True

1 2 <NA>

```
df_conv.dtypes
```

o/p : a Int64

b boolean

dtype : object

* Inserting missing data : we can insert missing values by simply assigning to a series / Dataframe.

```
Ser = Pd.Series([1., 2., 3.])

ser.loc[0] = None

Ser
```

o/p : 0 NaN → why NaN? because the input dtype is float 64

1 2.0 " if the input is Timestamp" we get "NaT"

2 3.0 as o/p

dtype = float64

For object types, Pandas will use the given :

```
S = Pd.Series(["a", "b", "c"], dtype = object)

S.loc[0] = None

S.loc[1] = np.Nan

S
```

o/p : 0 None

1 NaN

2 c

* Calculations with missing data :

```
Ser1 = Pd.Series([np.nan, 1, 2, np.nan])
Ser2 = Pd.Series([ 2  , 3, np.nan, 4])
a= Ser1 + Ser2
Print(a)
```

o/p :

0	NaN
1	4
2	NaN
3	NaN

dtype = float64

→ When summing data, NA values or empty data will be treated as zero.

```
Pd.Series([np.nan]).sum()
```

o/p : 0.0

→ When taking the Product NA values will be treated as 1.

```
Pd.Series([np.nan]).Prod()
```

o/p : 1.0

→ Cumulative methods like "cumsum()" and "cumProd()" ignore NA values by default, but Preserve them in the resulting arrays.

```
Ser = Pd.Series([1, np.nan, 3, np.nan])
Ser
```

o/p :

0	1.0
1	NaN
2	3.0
3	NaN

→ To override this behaviour and include NA Values, use skipna = false.

*Dropping missing data :

→ dropna() drops rows/columns with missing data.

```
pd.DataFrame([[np.nan, 1, 2], [1, 2, np.nan], [1, 2, 3]]) = df
```

df

o/p :

	0	1	2
0	NaN	1	2.0
1	1.0	2	NaN
2	1.0	2	3.0

df.dropna() ⟶ in default the axis=0 (row)

	0	1	2
2	1.0	2	3.0

df.dropna(axis=1) ⟶ for changing the order of dropna (column)

	1
0	1
1	2
2	2

*Filling missing data :

→ By Value :

```
data = {"np": [1.0, np.nan, np.nan, 2], "arrow": pd.array([1.0, pd.NA, pd.NA, 2], dtype = "float64[pyarrow]")}
df = pd.DataFrame(data)
df
```

Output on Next Page ⟶

	np	arrow
0	1.0	1.0
1	NaN	<NA>
2	NaN	<NA>
3	2.0	2.0

df.fillna(1)

	np	arrow
0	1.0	1.0
1	1.0	1.0
2	1.0	1.0
3	2.0	2.0

* we can fill either from forward/backward.

df.ffill() → forward

df.bfill() → backward

* we can also add limit to fill NA values

df.ffill(limit = 1)

↳ only 1st row values will be filled

* Interpolation :

df = pd.DataFrame({"A": [1, 2.1, np.nan, 4], "B": [0.25, np.nan, np.nan, 4]}

	A	B
0	1	0.25
1	2.1	NaN
2	NaN	NaN
3	4	4

df.interpolate()

→ This fills the NaN values with various interpolation methods.

→ By using scipy, we can pass a 1-d interpolation routine to method

```
df.interpolate(method = "barycentric")
df.interpolate(method = "pchip")
```

* Regular expression replacement :

```
d = {"a" : list(range(4)), "b" : list("ab.."), "c" :
["a", "b", np.nan, "d"]
df = pd.DataFrame(d)
df.replace(".", np.nan)
```

	a	b	c
0	0	a	a
1	1	b	b
2	2	NaN	NaN
3	3	NaN	d