How to Deal with Missing Data in Python

Using Pandas and NumPy for handling missing values in your dataset

A common occurrence in a data-set is **missing values**. This can happen due to multiple reasons like unrecorded observations or data corruption.

In this tutorial, we will walk through many different ways of **handling missing values in Python using the Pandas library**.

Pandas library provides a variety of functions for **marking these corrupt values**. We will study how we can **remove or impute these values**.

The Employee Dataset

We will be working with a **small Employee Dataset** for this tutorial.

Download the dataset in CSV format from Moodle (employees.csv) and store it in your current working directory:

Let us import this dataset into Python and take a look at it.

Output:

	First Name	Gender	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	97308	6.945	TRUE	Marketing
1	Thomas	Male	61933	NaN	TRUE	NaN
2	Maria	Female	130590	11.858	FALSE	Finance
3	Jerry	Male	NaN	9.34	TRUE	Finance
4	Larry	Male	101004	1.389	TRUE	Client Services

Source: Author

We read the CSV file into a **Pandas DataFrame**. The `.head()` method returns the first five rows of the DataFrame.

The dataset has 1000 observations with six variables as given below:

Variable Name	Data Type	Acceptable Values
First Name	String	only alphabets
Gender	String	(Male or Female)
Salary	Integer	Greater than 0
Bonus %	Float	Greater than 0
Senior Management	Boolean	True or False
Team	String	one of the Teams in the company

How to mark invalid/ corrupt values as missing

There are two types of missing values in every dataset:

1. **Visible errors**: blank cells, special symbols like **NA** (Not Available), **NaN** (Not a Number), etc.
2. **Obscure errors**: non-corrupt but **invalid values**. For example, a negative salary or a number for a name.

The employee dataset has multiple missing values. Let us take a closer look:

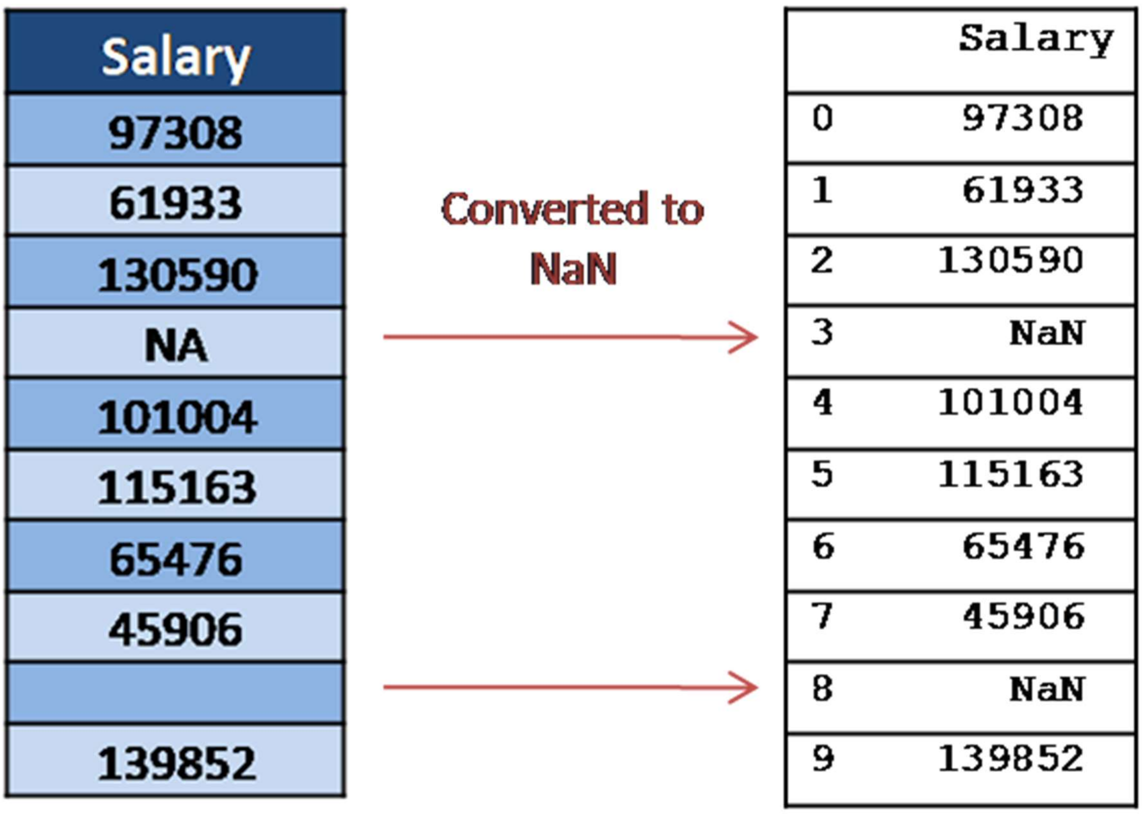
First Name	Gender	Salary	Bonus %	Senior Management	Team
Douglas	Male	97308	6.945	TRUE	Marketing
Thomas	Male	61933	NaN	TRUE	
Maria	Female	130590	11.858	FALSE	Finance
Jerry	Male	NA	9.34	TRUE	Finance
Larry	Male	101004	1.389	TRUE	Client Services
Dennis	n.a.	115163	10.125	FALSE	Legal
Ruby	Female	65476	10.012	TRUE	Product
	Female	45906	11.598		Finance
Angela			18.523	TRUE	Engineering

You would notice values like **NA**, **NaN**, **?** and also **blank cells**. These represent missing values in our dataset.

Let us print the first 10 rows of the '**Salary**' column.

```
print(df['Salary'].head(10))
```

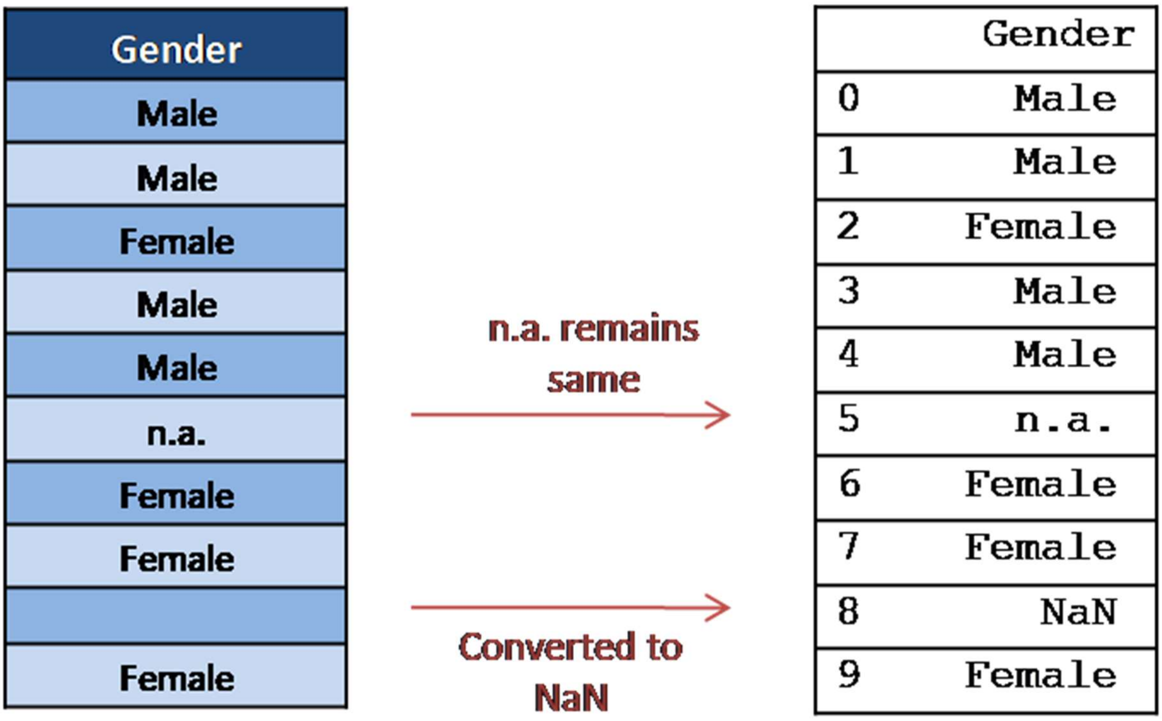
We depict both a snapshot from our dataset and the output of the above statement in the images below.



Pandas automatically marks blank values or values with NA as NaN (missing values).

Pandas also assigns an index value to each row. Values containing NaN are ignored from operations like mean, sum, etc.

While this works for NA and blank lines, **Pandas fails to identify other symbols like na, ?, n.a., n/a**. This can be seen in the '**Gender**' column:



As earlier, Pandas takes care of the blank value and converts it to NaN. **But it is unable to do so for the n.a. in the 6th row.**

With multiple users manually feeding data into a database, this is a common issue. **We can pass all of these symbols in the .read_csv() method as a list** to allow Pandas to recognize them as corrupt values. Take a look:

	Gender
0	Male
1	Male
2	Female
3	Male
4	Male
5	NaN
6	Female
7	Female
8	NaN
9	Female

Now, n.a. is also converted to NaN.

Handling Invalid Data Types

Till now, our missing values had unique identifiers which, made them pretty easy to catch. But what happens when we get an invalid data type.

For example, if we are expecting a numeric value but, the user inputs a string like '**No**' for salary. Technically, this is also a missing value.

I have designed a function that allows me to check for invalid data types in a column. Suppose I wish to ensure that all values in the '**Salary**' column are of type `int`. I will use the following:

After this, values in the salary column will be of type `int` with NaN wherever invalid entries occurred.

Marking missing values using isnull and notnull

In Pandas, we have two functions for marking missing values:

- `isnull()`: mark all NaN values in the dataset as True
- `notnull()`: mark all NaN values in the dataset as False.

Look at the code below:

```
# NaN values are marked True
print(df['Gender'].isnull().head(10))# NaN values are marked False
print(df['Gender'].notnull().head(10))
```

	Gender
0	Male
1	Male
2	Female
3	Male
4	Male
5	NaN
6	Female
7	Female
8	NaN
9	Female

Gender Column

	Gender
0	False
1	False
2	False
3	False
4	False
5	True
6	False
7	False
8	True
9	False

isnull Function

	Gender
0	True
1	True
2	True
3	True
4	True
5	False
6	True
7	True
8	False
9	True

notnull Function

Left: original Gender column, Middle: output of the `isnull()` function, Right: output of the `notnull()` function

We can use the outputs of the `isnull` and `notnull` functions for filtering.

Let us print all rows for which Gender is not missing.

```
# notnull will return False for all NaN values
null_filter = df['Gender'].notnull()# prints only those rows where
null_filter is True
print(df[null_filter])
```

Missing Value Statistics

`isnull` and `notnull` can also be used to summarize missing values.

To check if there are any missing values in our data frame:

```
print(df.isnull().values.any())# Output
True
```

Total number of missing values per column:

```
print(df.isnull().sum())# Output
First Name            70
Gender               149
Salary                 5
Bonus %                4
Senior Management     71
Team                  48
```

How to remove rows with missing values

Now that we have marked all missing values in our dataset as NaN, we need to decide how do we wish to handle them.

The most elementary strategy is to **remove all rows that contain missing values** or, in extreme cases, entire columns that contain missing values.

Pandas library provides the `dropna()` function that can be used to drop either columns or rows with missing data.

In the example below, we use `dropna()` to remove all rows with missing data:

```
# drop all rows with NaN values
df.dropna(axis=0,inplace=True)
```

`inplace=True` causes all changes to happen in the same data frame rather than returning a new one.

To drop columns, we need to set axis = 1.

We can also use the `how` parameter.

- **how = 'any'**: at least one value must be null.
- **how = 'all'**: all values must be null.

Some code examples using the `how` parameter:

Imputing Missing Values in our Dataset

Removing rows is a good option when missing values are rare. But this is not always practical. We need to replace these NaNs with intelligent guesses.

There are many options to pick from when replacing a missing value:

- A single pre-decided constant value, such as 0.
- Taking value from another randomly selected sample.
- Mean, median, or mode for the column.
- Interpolate value using a predictive model.

In order to fill missing values in a datasets, Pandas library provides us with `fillna()`, `replace()` and `interpolate()` functions.

Let us look at these functions one by one using examples.

Replacing NaNs with a single constant value

We will use `fillna()` to replace missing values in the **'Salary'** column with 0.

```
df['Salary'].fillna(0, inplace=True)# To check changes call
# print(df['Salary'].head(10))
```

We can also do the same for categorical variables like '**Gender'**.

```
df['Gender'].fillna('No Gender', inplace=True)
```

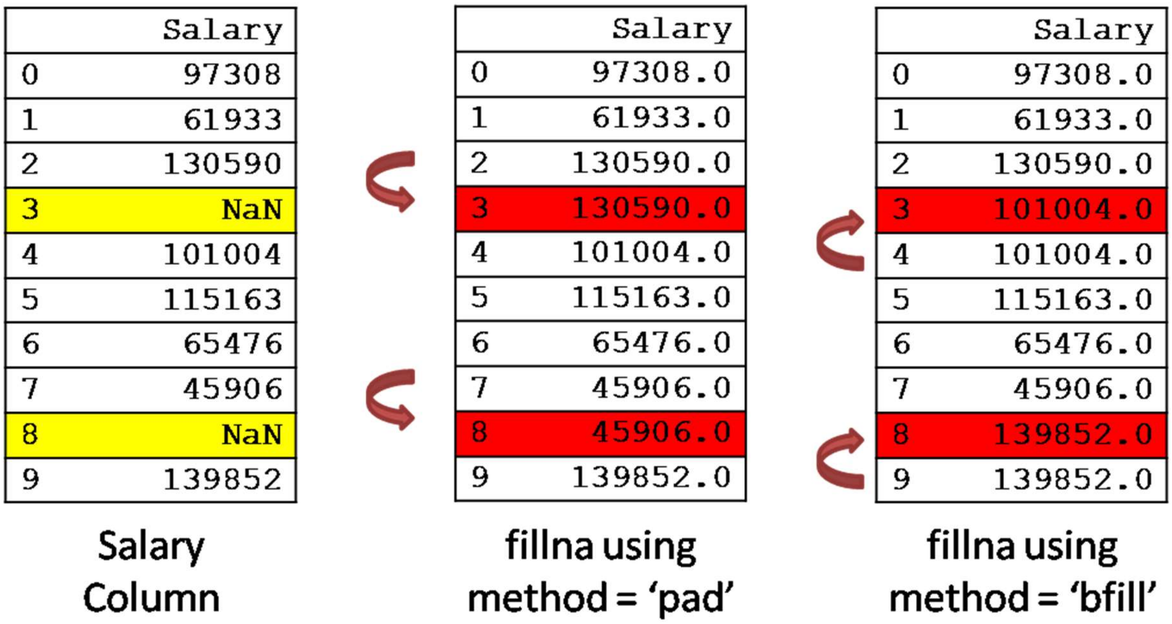
Replacing NaNs with the value from the previous row or the next row

This is a common approach when filling missing values in image data. We use `method = 'pad'` for taking values from **the previous row**.

```
df['Salary'].fillna(method='pad', inplace=True)
```

We use `method = 'bfill'` for taking values from **the next row**.

```
df['Salary'].fillna(method='bfill', inplace=True)
```



	Salary		Salary		Salary
0	97308	0	97308.0	0	97308.0
1	61933	1	61933.0	1	61933.0
2	130590	2	130590.0	2	130590.0
3	NaN	3	130590.0	3	101004.0
4	101004	4	101004.0	4	101004.0
5	115163	5	115163.0	5	115163.0
6	65476	6	65476.0	6	65476.0
7	45906	7	45906.0	7	45906.0
8	NaN	8	45906.0	8	139852.0
9	139852	9	139852.0	9	139852.0

Replacing NaNs using Median/Mean of the column

A common sensible approach.

```
# using median
df['Salary'].fillna(df['Salary'].median(), inplace=True)#using mean
df['Salary'].fillna(int(df['Salary'].mean()), inplace=True)
```

Note: Information about other options for `fillna` is available here.

Using the replace method

The `replace` method is a more generic form of the `fillna` method. Here, we specify both the value to be replaced and the replacement value.

```
# will replace NaN value in Salary with value 0
df['Salary'].replace(to_replace = np.nan, value = 0,inplace=True)
```

Using the interpolate method

`interpolate()` function is used to fill NaN values using various interpolation techniques. Read more about the interpolation methods here.

Let us interpolate the missing values using the **Linear Interpolation method**.

```
df['Salary'].interpolate(method='linear', direction = 'forward',
inplace=True)
print(df['Salary'].head(10))
```

	Salary
0	97308
1	61933
2	130590
3	NaN
4	101004
5	115163
6	65476
7	45906
8	NaN
9	139852

Salary Column

	Salary
0	97308.0
1	61933.0
2	130590.0
3	115797.0
4	101004.0
5	115163.0
6	65476.0
7	45906.0
8	92879.0
9	139852.0

Using Linear Interpolation

Whether we like it or not, real world data is messy. **Data cleaning** is a major part of every data science project.

Often data is missing due to random reasons like **data corruption, signal errors, etc**. But some times, there is a deeper reason for this missing data.

While we discussed replacement using mean, median, interpolation, etc, missing values usually have a **more complex statistical relationship** with our data.

.