# Synopsis

This notebook will explain the following topics and concepts:

**Missing Data**

- detecting
- removing
- filling in

**Data Transformation**

- counting values
- Imputing
- Removing Duplicates
- Replacing Values
- Common String Methods

**Importing formatted numerics**

**Pandas Options and Customisation**

- String Formatting
- Display Options
- Style

# Import packages

In [2]:

```
# import pandas and numpy
import pandas as pd
import numpy as np
```

# Missing Data

three main problems that missing data causes: >

> introduction of a substantial amount of bias
> make the handling and analysis of the data more arduous
> and create reductions in efficiency

## Filtering out missing data

- **dropna()** - Will detect and remove rows or columns (it's usually used for rows) where data is missing.

- Returns a copy, not the original.
- Catch result in a new variable OR set **inplace=True** to alter the original DataFrame.

In [3]:

```
# Simple Series for demonstration
arr = ['AAA', 'BBB', np.nan, 'DDD']
demo_series = pd.Series(data = arr)
demo_series
```

Out[3]:

```
0    AAA
1    BBB
2    NaN
3    DDD
dtype: object
```

In [4]:

```
# drop all invalid values - what happens?

demo_series.dropna()
```

Out[4]:

```
0    AAA
1    BBB
3    DDD
dtype: object
```

## Import Test Data

In [5]:

```
# read in some data from an excel file, sheet is called 'MissingData'
df_missing = pd.read_excel(io='https://s3.eu-west-1.amazonaws.com/neueda.conygre.com/
                           sheet_name='MissingData')

df_missing.head
```

Out[5]:

```
<bound method NDFrame.head of          A    B    C
0   38.0    I  1.0
1   40.0   II  2.0
2   35.0    I  NaN
3    NaN   II  4.0
4   38.0    I  1.0
5   40.0   II  2.0
6   35.0    I  3.0
7    NaN  NaN  4.0
8   38.0  III  3.0
9   38.0    I  1.0
10  40.0  NaN  NaN
11  35.0    I  3.0
12   NaN   II  4.0
13  38.0    I  1.0
14  40.0   II  2.0
15  35.0    I  3.0
16   NaN   II  4.0
17  38.0  III  3.0
18  38.0    I  NaN
19  40.0   II  2.0
20  35.0    I  3.0
21   NaN   II  4.0
22  38.0    I  1.0
23  40.0  NaN  2.0
24  35.0    I  3.0
25   NaN   II  NaN
26  38.0  III  3.0>
```

## Detecting Missing Data

Pandas includes a number of functions to detect missing or invalid data.

- isnull - Returns a Series containing True/False indicating if each value is missing.
- notnull - Opposite (negation) of isnull: True if value is not null, False otherwise.
- sum - how many null or not nulls exist

In [6]:

```
# try isnull() and notnull()
df_missing.isnull()

df_missing.notnull()

# How many in each column
df_missing.isnull().sum()

# How amny are empty in the entire dataset
df_missing.isnull().sum().sum()
```

Out[6]:

```
13
```

## Filling in missing values

- **fillna()** - Will detect and empty values and fill them in.

- You can give it a value to fill with

- Alternatively, it can fill with values from cells before or after the missing value (backfill or forwardfill).
- Again, catch result in a new variable OR set **inplace=True** to alter the original DataFrame.

In [7]:

```
# use fillna - returns a new object, can use inplace=True if desired
df_missing.fillna(0.42, inplace=True)
df_missing
```

Out[7]:

| | A | B | C |
|---|---|---|---|
| **0** | 38.00 | I | 1.00 |
| **1** | 40.00 | II | 2.00 |
| **2** | 35.00 | I | 0.42 |
| **3** | 0.42 | II | 4.00 |
| **4** | 38.00 | I | 1.00 |
| **5** | 40.00 | II | 2.00 |
| **6** | 35.00 | I | 3.00 |
| **7** | 0.42 | 0.42 | 4.00 |
| **8** | 38.00 | III | 3.00 |
| **9** | 38.00 | I | 1.00 |
| **10** | 40.00 | 0.42 | 0.42 |
| **11** | 35.00 | I | 3.00 |
| **12** | 0.42 | II | 4.00 |
| **13** | 38.00 | I | 1.00 |
| **14** | 40.00 | II | 2.00 |
| **15** | 35.00 | I | 3.00 |
| **16** | 0.42 | II | 4.00 |
| **17** | 38.00 | III | 3.00 |
| **18** | 38.00 | I | 0.42 |
| **19** | 40.00 | II | 2.00 |
| **20** | 35.00 | I | 3.00 |
| **21** | 0.42 | II | 4.00 |
| **22** | 38.00 | I | 1.00 |
| **23** | 40.00 | 0.42 | 2.00 |
| **24** | 35.00 | I | 3.00 |
| **25** | 0.42 | II | 0.42 |
| **26** | 38.00 | III | 3.00 |

# Data Transformation

## Removing duplicates

- **duplicated()** : indicates whether each row is a duplicate.
- **drop_duplicates()** : returns a copy of the DataFrame with the duplicates removed (or inplace=True).

In [ ]:

```
# view all duplicates in df_missing?
display(df_missing)
df_missing.duplicated()
```

In [ ]:

```
# drop all duplicates  - what parameters are there?
df_missing.drop_duplicates()
```

## Replacing Values

- **df.replace(to_replace, value)** : find and replace specific values.
- The parameters **to_replace** and **value** can both be either single values or lists of values.
- Returns a copy so again either use **inplace=True** OR catch the returned DataFrame in a new variable.

In [ ]:

```
# replace all 2 with 22
df_missing.replace(2,22, inplace=True)
df_missing
```

In [ ]:

```
# replace all 'I' with 11 AND 'III' with 33
df_missing = df_missing.replace(['I', 'III'],[11, 33])
df_missing
```

In [ ]:

```
# or use variables for the originals and replacements
orig_vals = ['I', 'III']
new_vals = [11, 33]

df_missing.replace(orig_vals,new_vals, inplace=True)
df_missing
```

# Importing Formatted Numerics

Some files may have had their numeric data formatted.
Pandas will interpret such values as string.

for example

- 23.45% ( as a string)
- 12,342 ( also a string)

Use the string **replace()** function in conjunction with **pandas.to_numeric()** to correctly import formatted numeric values.

In [10]:

```
# Read data into a DataFrame
df_SPX = pd.read_csv('https://s3.eu-west-1.amazonaws.com/neueda.conygre.com/pydata/SP
                     index_col='Date', parse_dates=True)

# Use the dtypes attribute to check what types are in each column
# the word 'object' is used to denote a string
print(df_SPX.dtypes)

df_SPX.head()
```

```
Price        object
Open         object
High         object
Low          object
Change %     object
dtype: object
```

Out[10]:

| Date | Price | Open | High | Low | Change % |
|---|---|---|---|---|---|
| **2017-12-29** | 2,673.61 | 2,689.15 | 2,692.12 | 2,673.61 | -0.52% |
| **2017-12-28** | 2,687.54 | 2,686.10 | 2,687.66 | 2,682.69 | 0.18% |
| **2017-12-27** | 2,682.62 | 2,682.10 | 2,685.64 | 2,678.91 | 0.08% |
| **2017-12-26** | 2,680.50 | 2,679.09 | 2,682.74 | 2,677.96 | -0.11% |
| **2017-12-22** | 2,683.34 | 2,684.22 | 2,685.35 | 2,678.13 | -0.05% |

In [ ]:

```
# Convert the value in the 'Price' column from a String to a numeric (notice we also
df_SPX['Price'] = pd.to_numeric(df_SPX['Price'].str.replace(',', ''))

# Now check the dtypes and compare to the previous cell - price is now a "float64" i.
print(df_SPX.dtypes)


df_SPX.head()
```

# Exercise

- Update the "Change %" column
- Remove the '%' character and convert to numeric values
- Print the dtypes for the updated DataFrame to verify your change
- Display the first 5 rows of the updated DataFrame

In [ ]:

```
# Do the exercise here
```

# Pandas Options, Customisation

This sections shows for reference some ways to format strings and use pandas options

4 ways to format strings

- C Style formatting
- “New Style” String Formatting
- Formatted String Literals
- Template Strings

## C-Style String Formatting

Based on C language `printf` function - the %-operator

- Single Substitution

- Multiple Substitution: wrap the right-hand side in a tuple,

In [ ]:

```
# Single Substitution
fav_song = "Hey Jude"
s = 'Favourite song is %s' % fav_song
print(s)

# Multiple Substitution:
```

```
fname = "Bob"
lname = "Dylan"
s = 'Favourite singer is %s %s' % (fname, lname)
print(s)
```

## “New Style” String Formatting

Introduced in Python 3, back ported to python 2.7
Replaces `%operator` with a `.format()` function and variable substitution

In [ ]:

```
fav_song = "Hey Jude"
s = 'Favourite song is {}'.format(fav_song)
print(s)

fname = "Bob"
lname = "Dylan"
s = 'Favourite singer is {} {}'.format(fname, lname)
print(s)

# Same as previous but using named parameters
s = 'Favourite singer is {s1} {s2}'.format(s1=fname, s2=lname)
print(s)
```

## Formatted String Literals

Added in python 3.6

In [ ]:

```
# Use embedded Python expressions inside string constants
fav_song = "Hey Jude"
s = f'Favourite song is, {fav_song}!'
print(s)

# embed arbitrary Python expressions
a = 5
b = 10
s = f'Five plus ten is {a + b} and not {2 * (a + b)}.'
print(s)
```

## Template Strings

Simpler and less powerful mechanism

In [ ]:

```
from string import Template

t = Template('Favourite singer is $s1 $s2')

s = t.substitute(s1=fname, s2=lname)
print(s)
```

# Display Options

Pandas have some default factors which restrict the analysis of data.

Therefore to have a stronghold over the library and to make the most out of its uses, it is important to know the various methods to change the default pandas values.

Common default values-

- `display.max_rows` and `display.max_columns` which shows the default number of rows and columns.
- `display.max_colwidth` which gives us the maximum width of the column
- `display.expand_frame_repr` which gives us DataFrames that is spread across numerous pages.
- `display.precision` gives us the precision of the decimal numbers

Full list of options https://pandas.pydata.org/pandas-docs/stable/user_guide/options.html#available-options

## Pandas.get_option()

- return particular detail about the default values in pandas.

Using `display.max_rows` and `display.max_columns”` as parameters we get a maximum number of rows and columns that can display by default.

In [ ]:

```
opts = pd.get_option("display.max_rows")
print(opts)

opts = pd.get_option("display.max_columns")
print(opts)
```

## Pandas.set_option()

- change a default value to something of our choice.

e.g. change the “display.max_rows” from 60 to 90.

In [ ]:

```
pd.set_option("display.max_rows",90)
opts = pd.get_option("display.max_rows")
print(opts)

pd.set_option("display.max_columns",10)
opts= pd.get_option("display.max_columns")
print(opts)
```

## Pandas.reset_option

- get back the default values which may change previously.

In [ ]:

```
pd.reset_option("display.max_rows")
opts = pd.get_option("display.max_rows")
print(opts)

pd.reset_option("display.max_columns")
opts = pd.get_option("display.max_columns")
print(opts)
```

## Pandas.describe_option

- describes the parameter.

In [ ]:

```
pd.describe_option("display.max_rows")
```

## Pandas.option_context

invoke a pandas option function which will be only active within the scope of the function.

In the below example, display.max_rows is set to 30 only inside the .option-context scope. Outside the function scope, it returns back to being 60.

In [ ]:

```
with pd.option_context("display.max_rows",30):
      print(pd.get_option("display.max_rows"))

print(pd.get_option("display.max_rows"))
```