Pandas 5

Content

- Null/Missing values
  - `None` vs `NaN` values
  - `isna()` & `isnull()`
- Removing null values
  - `dropna()`
- Data Imputation
  - `fillna()`
- String methods
- Datetime values
- Writing to a file

In [2]:

```
import pandas as pd
import numpy as np

data = pd.read_csv('Pfizer_1.csv')

data_melt = pd.melt(data,id_vars = ['Date', 'Drug_Name', 'Parameter'],
            var_name = "time",
            value_name = 'reading')

data_tidy = data_melt.pivot(index=['Date','time', 'Drug_Name'],
                            columns = 'Parameter',
                            values='reading')

data_tidy = data_tidy.reset_index()
data_tidy.columns.name = None
```

In [3]:

```
data.head()
```

Out[3]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00
0	15-10-2020	diltiazem hydrochloride	Temperature	23.0	22.0	NaN	21.0	21.0	22
1	15-10-2020	diltiazem hydrochloride	Pressure	12.0	13.0	NaN	11.0	13.0	14
2	15-10-2020	docetaxel injection	Temperature	NaN	17.0	18.0	NaN	17.0	18
3	15-10-2020	docetaxel injection	Pressure	NaN	22.0	22.0	NaN	22.0	23
4	15-10-2020	ketamine hydrochloride	Temperature	24.0	NaN	NaN	27.0	NaN	26

In [4]:

```
data_melt.head()
```

Out[4]:

	Date	Drug_Name	Parameter	time	reading
0	15-10-2020	diltiazem hydrochloride	Temperature	1:30:00	23.0
1	15-10-2020	diltiazem hydrochloride	Pressure	1:30:00	12.0
2	15-10-2020	docetaxel injection	Temperature	1:30:00	NaN
3	15-10-2020	docetaxel injection	Pressure	1:30:00	NaN
4	15-10-2020	ketamine hydrochloride	Temperature	1:30:00	24.0

In [5]:

```
data_tidy.head()
```

Out[5]:

	Date	time	Drug_Name	Pressure	Temperature
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0

`None` vs `NaN`

If you notice, there are many `NaN` values in our data.

What are these `NaN` values?

- They are basically **missing/null values**.
- A null value signifies an **empty cell/no data**.

There can be 2 kinds of missing values:

1. `None`
2. `NaN` (Not a Number)

Whats the difference between the `None` and `NaN`?

Both `None` and `NaN` can be used for missing values, but their representation and behaviour may differ based on the **column's data type**.

In [6]:

```
type(None)
```

Out[6]:

```
NoneType
```

In [7]:

```
type(np.nan)
```

Out[7]:

```
float
```

1. **None in Non-numeric** columns: None can be used directly, and it will appear as None.

2. **None in Numeric** columns: Pandas automatically converts None to NaN.
3. **NaN in Numeric** columns: NaN is used to represent missing values and appears as NaN.
4. **NaN in Non-numeric** Columns: NaN can be used, and it appears as NaN.

```
In [8]: pd.Series([1, np.nan, 2, None])
```

```
Out[8]: 0    1.0
        1    NaN
        2    2.0
        3    NaN
        dtype: float64
```

For **numerical** type, Pandas changes `None` to `NaN`.

```
In [9]: pd.Series(["1", "np.nan", "2", None])
```

```
Out[9]: 0         1
        1    np.nan
        2         2
        3      None
        dtype: object
```

For **object** type, the `None` is preserved and not changed to `NaN`.

`isna()` & `isnull()`

How to get the count of missing values for each row/column?

- `df.isna()`
- `df.isnull()`

```
In [10]: data.isna().head()
```

Out[10]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00	7
0	False	False	False	False	False	True	False	False	False	
1	False	False	False	False	False	True	False	False	False	
2	False	False	False	True	False	False	True	False	False	
3	False	False	False	True	False	False	True	False	False	
4	False	False	False	False	True	True	False	True	False	

```
In [11]: data.isnull().head()
```

Out[11]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00	7
0	False	False	False	False	False	True	False	False	False	
1	False	False	False	False	False	True	False	False	False	
2	False	False	False	True	False	False	True	False	False	
3	False	False	False	True	False	False	True	False	False	
4	False	False	False	False	True	True	False	True	False	

Notice that both `isna()` and `isnull()` give the same results.

But why do we have two methods, `isna()` and `isnull()` for the same operation?

- `isnull()` is just an alias for `isna()`

In [12]:
```
pd.isnull
```

Out[12]:
```
<function pandas.core.dtypes.missing.isna(obj: 'object') -> 'bool | npt.NDA
rray[np.bool_] | NDFrame'>
```

In [13]:
```
pd.isna
```

Out[13]:
```
<function pandas.core.dtypes.missing.isna(obj: 'object') -> 'bool | npt.NDA
rray[np.bool_] | NDFrame'>
```

As we can see, the function signature is same for both.

- `isna()` returns a **boolean dataframe**, with each cell as a boolean value.
- This value corresponds to **whether the cell has a missing value**.
- On top of this, we can use `.sum()` to find the count of the missing values.

In [14]:
```
data.isna().sum()
```

Out[14]:
```
Date          0
Drug_Name     0
Parameter     0
1:30:00       2
2:30:00       2
3:30:00       6
4:30:00       4
5:30:00       2
6:30:00       0
7:30:00       2
8:30:00       4
9:30:00       2
10:30:00      0
11:30:00      2
12:30:00      0
dtype: int64
```

This gives us the total number of missing values in each column.

How can we get the number of missing values in each row?

In [15]:
```
data.isna().sum(axis=1)
```

Out[15]:
```
0     1
1     1
2     4
3     4
4     3
5     3
6     1
7     1
8     1
9     1
10    2
11    2
12    1
13    1
14    0
15    0
16    0
17    0
dtype: int64
```

Note: By default, the value is `axis=0` for `sum()`.

We now have identified the null count, but how do we deal with them?

We have two options:

- Delete the rows/columns containing the null values.
- Fill the missing values with some data/estimate.

Let's first look at deleting the rows.

Removing null values

How can we drop rows containing null values?

In [16]:
```
data.dropna()
```

Out[16]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00
14	17-10-2020	docetaxel injection	Temperature	12.0	13.0	14.0	15.0	16.0	17
15	17-10-2020	docetaxel injection	Pressure	20.0	22.0	22.0	22.0	22.0	23
16	17-10-2020	ketamine hydrochloride	Temperature	13.0	14.0	15.0	16.0	17.0	18
17	17-10-2020	ketamine hydrochloride	Pressure	8.0	9.0	10.0	11.0	11.0	12

Notice that rows with even a single missing value have been deleted.

What if we want to delete the columns having missing value?

In [18]:

```
data.dropna(axis=1)
```

Out[18]:

	Date	Drug_Name	Parameter	6:30:00	10:30:00	12:30:00
0	15-10-2020	diltiazem hydrochloride	Temperature	22	20	21
1	15-10-2020	diltiazem hydrochloride	Pressure	14	18	20
2	15-10-2020	docetaxel injection	Temperature	18	23	25
3	15-10-2020	docetaxel injection	Pressure	23	26	28
4	15-10-2020	ketamine hydrochloride	Temperature	26	22	20
5	15-10-2020	ketamine hydrochloride	Pressure	9	9	11
6	16-10-2020	diltiazem hydrochloride	Temperature	38	40	42
7	16-10-2020	diltiazem hydrochloride	Pressure	23	24	27
8	16-10-2020	docetaxel injection	Temperature	49	56	58
9	16-10-2020	docetaxel injection	Pressure	27	28	30
10	16-10-2020	ketamine hydrochloride	Temperature	12	13	15
11	16-10-2020	ketamine hydrochloride	Pressure	15	16	18
12	17-10-2020	diltiazem hydrochloride	Temperature	16	14	10
13	17-10-2020	diltiazem hydrochloride	Pressure	8	11	14
14	17-10-2020	docetaxel injection	Temperature	17	21	23
15	17-10-2020	docetaxel injection	Pressure	23	28	28
16	17-10-2020	ketamine hydrochloride	Temperature	18	22	24
17	17-10-2020	ketamine hydrochloride	Pressure	12	13	15

Notice that every column which had even a single missing value has been deleted.

But what are the problems with deleting rows/columns?

- loss of valuable data

So instead of dropping, it would be better to **fill the missing values with some data**.

Data Imputation

How can we fill the missing values with some data?

In [19]:

```
data.fillna(0).head()
```

Out[19]:

	Date	Drug_Name	Parameter	1:30:00	2:30:00	3:30:00	4:30:00	5:30:00	6:30:00
0	15-10-2020	diltiazem hydrochloride	Temperature	23.0	22.0	0.0	21.0	21.0	22
1	15-10-2020	diltiazem hydrochloride	Pressure	12.0	13.0	0.0	11.0	13.0	14
2	15-10-2020	docetaxel injection	Temperature	0.0	17.0	18.0	0.0	17.0	18
3	15-10-2020	docetaxel injection	Pressure	0.0	22.0	22.0	0.0	22.0	23
4	15-10-2020	ketamine hydrochloride	Temperature	24.0	0.0	0.0	27.0	0.0	26

What is `fillna(0)` doing?

- It fills all the missing values with 0.

We can do the same on a particular column too.

In [20]:

```
data['2:30:00'].fillna(0)
```

Out[20]:

```
0     22.0
1     13.0
2     17.0
3     22.0
4      0.0
5      0.0
6     35.0
7     19.0
8     47.0
9     24.0
10     9.0
11    12.0
12    19.0
13     4.0
14    13.0
15    22.0
16    14.0
17     9.0
Name: 2:30:00, dtype: float64
```

Note:

Handling missing value completely depends on the business problem.

However, in general practice (assuming you have a large dataset) -

- if the missing values are minimal (\<5% of rows), dropping them is acceptable.
- for substantial missing values (>10% of rows), use a suitable imputation technique.
- if a column has over 50% of null values, drop that column (unless it's very crucial for the analysis).

What other values can we use to fill the missing values?

We can use some kind of estimator too.

- mean (average value)
- median
- mode (most frequently occuring value)

How would you calculate the mean of the column `2:30:00` ?

In [21]:

```
data['2:30:00'].mean()
```

Out[21]:

```
18.8125
```

Now let's fill the `NaN` values with the mean value of the column.

In [22]:

```
data['2:30:00'].fillna(data['2:30:00'].mean())
```

Out[22]:

```
0     22.0000
1     13.0000
2     17.0000
3     22.0000
4     18.8125
5     18.8125
6     35.0000
7     19.0000
8     47.0000
9     24.0000
10     9.0000
11    12.0000
12    19.0000
13     4.0000
14    13.0000
15    22.0000
16    14.0000
17     9.0000
Name: 2:30:00, dtype: float64
```

But this doesn't feel right. What could be wrong with this?

Can we use the mean of all compounds as average for our estimator?

- Different drugs have different characteristics.
- We can't simply do an average and fill the null values.

Then what could be the solution here?

We could fill the null values of respective compounds with their respective means.

How can we form a column with mean temperature of respective compounds?

- We can use `apply()`

Let's first create a function to calculate the mean.

In [23]:

```
def temp_mean(x):
  x['Temperature_avg'] = x['Temperature'].mean()
  return x
```

Now we can form a new column based on the average values of temperature for each drug.

In [25]:

```
data_tidy = data_tidy.groupby(["Drug_Name"]).apply(temp_mean)
data_tidy
```

```
/var/folders/zk/yt14z40j2lb2lz548fqr3v9m0000gn/T/ipykernel_59236/264220330
0.py:1: FutureWarning: Not prepending group keys to the result index of tra
nsform-like apply. In the future, the group keys will be included in the in
dex, regardless of whether the applied function returns a like-indexed obje
ct.
To preserve the previous behavior, use

        >>> .groupby(..., group_keys=False)

To adopt the future behavior and silence this warning, use

        >>> .groupby(..., group_keys=True)
  data_tidy = data_tidy.groupby(["Drug_Name"]).apply(temp_mean)
```

Out[25]:

	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0	30.387097
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0	30.387097
...	...	...	...	...	...	...
103	17-10-2020	8:30:00	docetaxel injection	26.0	19.0	30.387097
104	17-10-2020	8:30:00	ketamine hydrochloride	11.0	20.0	17.709677
105	17-10-2020	9:30:00	diltiazem hydrochloride	9.0	13.0	24.848485
106	17-10-2020	9:30:00	docetaxel injection	27.0	20.0	30.387097
107	17-10-2020	9:30:00	ketamine hydrochloride	12.0	21.0	17.709677

108 rows × 6 columns

In [26]:

```
data_tidy = data_tidy.groupby(["Drug_Name"],group_keys=False).apply(temp_mea
data_tidy
```

Out[26]:

	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0	30.387097
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0	30.387097
...	...	...	...	...	...	...
103	17-10-2020	8:30:00	docetaxel injection	26.0	19.0	30.387097
104	17-10-2020	8:30:00	ketamine hydrochloride	11.0	20.0	17.709677
105	17-10-2020	9:30:00	diltiazem hydrochloride	9.0	13.0	24.848485
106	17-10-2020	9:30:00	docetaxel injection	27.0	20.0	30.387097
107	17-10-2020	9:30:00	ketamine hydrochloride	12.0	21.0	17.709677

108 rows × 6 columns

Now we fill the null values in `Temperature` using this new column.

In [27]:

```
data_tidy['Temperature'].fillna(data_tidy["Temperature_avg"], inplace=True)
data_tidy
```

Out[27]:

	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0	30.387097
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0	30.387097
...	...	...	...	...	...	...
103	17-10-2020	8:30:00	docetaxel injection	26.0	19.0	30.387097
104	17-10-2020	8:30:00	ketamine hydrochloride	11.0	20.0	17.709677
105	17-10-2020	9:30:00	diltiazem hydrochloride	9.0	13.0	24.848485
106	17-10-2020	9:30:00	docetaxel injection	27.0	20.0	30.387097
107	17-10-2020	9:30:00	ketamine hydrochloride	12.0	21.0	17.709677

108 rows × 6 columns

In [28]:

```
data_tidy.isna().sum()
```

Out[28]:

```
Date                0
time                0
Drug_Name           0
Pressure           13
Temperature         0
Temperature_avg     0
dtype: int64
```

Great!

We have removed the null values from our `Temperature` column.

Let's do the same for `Pressure`.

In [29]:

```
def pr_mean(x):
  x['Pressure_avg'] = x['Pressure'].mean()
  return x
data_tidy=data_tidy.groupby(["Drug_Name"]).apply(pr_mean)
data_tidy['Pressure'].fillna(data_tidy["Pressure_avg"], inplace=True)
data_tidy
```

```
/var/folders/zk/yt14z40j2lb2lz548fqr3v9m0000gn/T/ipykernel_59236/258637458
5.py:4: FutureWarning: Not prepending group keys to the result index of tra
nsform-like apply. In the future, the group keys will be included in the in
dex, regardless of whether the applied function returns a like-indexed obje
ct.
To preserve the previous behavior, use

        >>> .groupby(..., group_keys=False)

To adopt the future behavior and silence this warning, use

        >>> .groupby(..., group_keys=True)
  data_tidy=data_tidy.groupby(["Drug_Name"]).apply(pr_mean)
```

Out[29]:

	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg	Pressure_avg
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485	15.424242
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0	30.387097	25.483871
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677	11.935484
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485	15.424242
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0	30.387097	25.483871
...	...	...	...	...	...	...	...
103	17-10-2020	8:30:00	docetaxel injection	26.0	19.0	30.387097	25.483871
104	17-10-2020	8:30:00	ketamine hydrochloride	11.0	20.0	17.709677	11.935484
105	17-10-2020	9:30:00	diltiazem hydrochloride	9.0	13.0	24.848485	15.424242
106	17-10-2020	9:30:00	docetaxel injection	27.0	20.0	30.387097	25.483871
107	17-10-2020	9:30:00	ketamine hydrochloride	12.0	21.0	17.709677	11.935484

108 rows × 7 columns

In [30]:

```
data_tidy.isna().sum()
```

```
Out[30]: Date               0
         time               0
         Drug_Name          0
         Pressure           0
         Temperature        0
         Temperature_avg    0
         Pressure_avg       0
         dtype: int64
```

How to decide if we should impute the missing values with `mean`, `median` or `mode`?

1. `Mean`: Use when dealing with numerical data that is normally distributed and not heavily skewed by outliers.

2. `Median`: Preferable when data is skewed or contains outliers. It's suitable for ordinal or interval data.

3. `Mode`: Suitable for categorical or nominal data where there are distinct categories.

Question

Based on the given DataFrame, which of the following statements regarding data imputation is mostly accurate?

```
|   CustomerID   |  TransactionAmount  |     Gender     |
Age  |  ProductCategory  |
|----------------|---------------------|----------------|----
---|-------------------|
|       101      |        20           |     Male       |
35  |       Apparel     |
|       102      |       NaN           |     Female     |
28  |       NaN         |
|       103      |        15           |     Female     |
NaN  |       Electronics |
|       104      |        30           |     NaN        |
42  |       Electronics |
|       105      |       150           |     Male       |
30  |       Apparel     |

A) Imputing missing values in the "TransactionAmount" column
using the mean of the available values may not be suitable
due to potential skewness caused by outliers.
B) Imputing missing values in the "TransactionAmount" column
using the median of the available values may be suitable to
handle skewness due to outliers.
C) The presence of missing values in the "Gender" column can
be effectively handled by imputing the most frequent category
(mode).
D) All of the above
```

Answer: All of the above

Explanation:

- Option A is correct because imputing missing values in the "TransactionAmount" column with the mean may not be appropriate if the data contains outliers. Outliers can significantly skew the mean, leading to inaccurate imputations.
- Option B is correct because as the data is skewed, the median that is roubst to outliers can better impute the missing data

- Option C is correct because for the "Gender" categorical column, the most frequently occuring category can be used to impute as gender is unlikely to exhibit significant variation in a dataset of customer transactions.

String methods

What kind of questions can we use string methods for?

- Find rows which contains a particular string.

Say,

How you can you filter rows containing "hydrochloride" in their drug name?

In [31]:

```
data_tidy.loc[data_tidy['Drug_Name'].str.contains('hydrochloride')].head()
```

Out[31]:

	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg	Pressure_avg
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485	15.424242
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677	11.935484
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485	15.424242
5	15-10-2020	11:30:00	ketamine hydrochloride	9.0	21.0	17.709677	11.935484
6	15-10-2020	12:30:00	diltiazem hydrochloride	20.0	21.0	24.848485	15.424242

- So in general, we will be using the following format: `Series.str.function()`

- `Series.str` can be used to access the values of the series as strings and apply several methods to it.

Now suppose we want to form a new column based on the year of the experiments?

What can we do form a column containing the year?

In [32]:

```
data_tidy['Date'].str.split('-')
```

```
Out[32]: 0      [15, 10, 2020]
         1      [15, 10, 2020]
         2      [15, 10, 2020]
         3      [15, 10, 2020]
         4      [15, 10, 2020]
                     ...
         103    [17, 10, 2020]
         104    [17, 10, 2020]
         105    [17, 10, 2020]
         106    [17, 10, 2020]
         107    [17, 10, 2020]
         Name: Date, Length: 108, dtype: object
```

To extract the year, we need to select the last element of each list.

```
In [33]: data_tidy['Date'].str.split('-').apply(lambda x:x[2])
```

```
Out[33]: 0      2020
         1      2020
         2      2020
         3      2020
         4      2020
                ...
         103    2020
         104    2020
         105    2020
         106    2020
         107    2020
         Name: Date, Length: 108, dtype: object
```

But there are certain problems with this approach.

- The **dtype of the output is still an object**, we would prefer a number type.
- The date format will always **not be in day-month-year**, it can vary.

Thus, to work with such date-time type of data, we can use a special method from Pandas.

Datetime

How can we handle datetime data types?

- We can use the `to_datetime()` function of Pandas
- It takes as input:
  - Array/Scalars with values having proper date/time format
  - `dayfirst` : Indicating if the day comes first in the date format used
  - `yearfirst` : Indicates if year comes first in the date format used

Let's first merge our `Date` and `Time` columns into a new `timestamp` column.

```
In [34]: data_tidy['timestamp'] = data_tidy['Date'] + " " + data_tidy['time']
```

```
In [35]: data_tidy.head()
```

Out[35]:

	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg	Pressure_avg	ti
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485	15.424242	
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0	30.387097	25.483871	
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677	11.935484	
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485	15.424242	
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0	30.387097	25.483871	

In [36]:

```
data_tidy['timestamp'] = pd.to_datetime(data_tidy['timestamp'])
data_tidy
```

Out[36]:

	Date	time	Drug_Name	Pressure	Temperature	Temperature_avg	Pressure_avg
0	15-10-2020	10:30:00	diltiazem hydrochloride	18.0	20.0	24.848485	15.424242
1	15-10-2020	10:30:00	docetaxel injection	26.0	23.0	30.387097	25.483871
2	15-10-2020	10:30:00	ketamine hydrochloride	9.0	22.0	17.709677	11.935484
3	15-10-2020	11:30:00	diltiazem hydrochloride	19.0	20.0	24.848485	15.424242
4	15-10-2020	11:30:00	docetaxel injection	29.0	25.0	30.387097	25.483871
...	...	...	...	...	...	...	...
103	17-10-2020	8:30:00	docetaxel injection	26.0	19.0	30.387097	25.483871
104	17-10-2020	8:30:00	ketamine hydrochloride	11.0	20.0	17.709677	11.935484
105	17-10-2020	9:30:00	diltiazem hydrochloride	9.0	13.0	24.848485	15.424242
106	17-10-2020	9:30:00	docetaxel injection	27.0	20.0	30.387097	25.483871
107	17-10-2020	9:30:00	ketamine hydrochloride	12.0	21.0	17.709677	11.935484

108 rows × 8 columns

In [37]:

```
data_tidy.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 108 entries, 0 to 107
Data columns (total 8 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Date             108 non-null    object
 1   time             108 non-null    object
 2   Drug_Name        108 non-null    object
 3   Pressure         108 non-null    float64
 4   Temperature      108 non-null    float64
 5   Temperature_avg  108 non-null    float64
 6   Pressure_avg     108 non-null    float64
 7   timestamp        108 non-null    datetime64[ns]
dtypes: datetime64[ns](1), float64(4), object(3)
memory usage: 11.7+ KB
```

The type of `timestamp` column has been changed from `object` to `datetime`.

Now, let's look at a single timestamp using Pandas.

How can we extract information from a single timestamp using Pandas?

```
In [38]: ts = data_tidy['timestamp'][0]
         ts
```

```
Out[38]: Timestamp('2020-10-15 10:30:00')
```

```
In [39]: ts.year, ts.month, ts.day, ts.month_name()
```

```
Out[39]: (2020, 10, 15, 'October')
```

```
In [40]: ts.hour, ts.minute, ts.second
```

```
Out[40]: (10, 30, 0)
```

This data parsing from `string` to `datetime` makes it easier to work with such data.

We can use this data from the columns as a whole using `.dt` object.

```
In [41]: data_tidy['timestamp'].dt
```

```
Out[41]: <pandas.core.indexes.accessors.DatetimeProperties object at 0x12206b950>
```

- `dt` gives properties of values in a column.
- From this `DatetimeProperties` of column `'end'`, we can extract `year`.

```
In [42]: data_tidy['timestamp'].dt.year
```

```
Out[42]: 0      2020
         1      2020
         2      2020
         3      2020
         4      2020
                ...
         103    2020
         104    2020
         105    2020
         106    2020
         107    2020
         Name: timestamp, Length: 108, dtype: int64
```

We can use `strfttime` (**short for stringformat time**), to modify our datetime format.

Let's learn this with the help of few examples.

```
In [43]: data_tidy['timestamp'][0]
```

```
Out[43]: Timestamp('2020-10-15 10:30:00')
```

```
In [44]: print(data_tidy['timestamp'][0].strftime('%Y')) # formatter for year
```

```
2020
```

Similarly we can combine the format types to modify the datetime format as per our convinience.

A comprehensive list of other formats can be found here:
https://pandas.pydata.org/docs/reference/api/pandas.Period.strftime.html

```
In [45]: data_tidy['timestamp'][0].strftime('%m-%d')
Out[45]: '10-15'
```

Writing to a file

How can we write our dataframe to a CSV file?

- We have to provide the `path` and `file_name` in which we want to store the data.

```
In [46]: data_tidy.to_csv('pfizer_tidy.csv', sep=",", index=False)
```

Setting `index=False` will not inlcude the index column while writing.

```
In [ ]:
```