

Pandas

April 2, 2024

- PANDAS comes from the term “panel data”
- index -> axis=0
- columns -> axis=1
- “Hierarchical columns” look like this:

	baz			zoo		
bar	A	B	C	A	B	C
foo						
one	1	2	3	x	y	z
two	4	5	6	q	w	t

1 Initialisation

```
[1]: import pandas as pd
import numpy as np
from IPython.display import HTML, Markdown, Latex

def display_df(tp_df=None, index=False):
    # tp_df = tp_df if isinstance(tp_df, pd.DataFrame) else df
    tp_df = tp_df if tp_df is not None else df
    display(Markdown(tp_df.to_markdown(index=index)))

# def display_df(tp_df=None, index=False):
#     tp_df = tp_df if isinstance(tp_df, pd.DataFrame) else df
#     display(HTML(tp_df.to_html(index=index)))
```

2 Creating a dataframe

```
[2]: data = {
    'age': [10, 22, 13, 21, 12, 11, 17],
    'section': ['A', 'B', 'C', 'B', 'B', 'A', 'A'],
```

```

    'city':          □
    ↪['Gurgaon','Delhi','Mumbai','Delhi','Mumbai','Delhi','Mumbai'],
    'gender':        ['M','F','F','M','M','M','F'],
    'favorite_color': ['red','black','yellow','pink','black','green','red']
}

data_csv = [
    ['age', 'section', 'city', 'gender', 'favorite_color'],
    [10, 'A', 'Gurgaon', 'M', 'red'],
    [22, 'B', 'Delhi', 'F', 'black'],
    [13, 'C', 'Mumbai', 'F', 'yellow'],
    [21, 'B', 'Delhi', 'M', 'pink'],
    [12, 'B', 'Mumbai', 'M', 'black'],
    [11, 'A', 'Delhi', 'M', 'green'],
    [17, 'A', 'Mumbai', 'F', 'red']
]

data_dict = [
    {'age': 10, 'section': 'A', 'city': 'Gurgaon', 'gender': 'M', □
    ↪'favorite_color': 'red'},
    {'age': 22, 'section': 'B', 'city': 'Delhi', 'gender': 'F', □
    ↪'favorite_color': 'black'},
    {'age': 13, 'section': 'C', 'city': 'Mumbai', 'gender': 'F', □
    ↪'favorite_color': 'yellow'},
    {'age': 21, 'section': 'B', 'city': 'Delhi', 'gender': 'M', □
    ↪'favorite_color': 'pink'},
    {'age': 12, 'section': 'B', 'city': 'Mumbai', 'gender': 'M', □
    ↪'favorite_color': 'black'},
    {'age': 11, 'section': 'A', 'city': 'Delhi', 'gender': 'M', □
    ↪'favorite_color': 'green'},
    {'age': 17, 'section': 'A', 'city': 'Mumbai', 'gender': 'F', □
    ↪'favorite_color': 'red'}
]

df = pd.DataFrame(data)
display_df()

df = pd.DataFrame(data_csv[1:], columns=data_csv[0])
display_df()

df = pd.DataFrame(data_dict)
display_df()

display_df(index=True)

```

age	section	city	gender	favorite_color
10	A	Gurgaon	M	red
22	B	Delhi	F	black
13	C	Mumbai	F	yellow
21	B	Delhi	M	pink
12	B	Mumbai	M	black
11	A	Delhi	M	green
17	A	Mumbai	F	red

age	section	city	gender	favorite_color
10	A	Gurgaon	M	red
22	B	Delhi	F	black
13	C	Mumbai	F	yellow
21	B	Delhi	M	pink
12	B	Mumbai	M	black
11	A	Delhi	M	green
17	A	Mumbai	F	red

age	section	city	gender	favorite_color
10	A	Gurgaon	M	red
22	B	Delhi	F	black
13	C	Mumbai	F	yellow
21	B	Delhi	M	pink
12	B	Mumbai	M	black
11	A	Delhi	M	green
17	A	Mumbai	F	red

	age	section	city	gender	favorite_color
0	10	A	Gurgaon	M	red
1	22	B	Delhi	F	black
2	13	C	Mumbai	F	yellow
3	21	B	Delhi	M	pink
4	12	B	Mumbai	M	black
5	11	A	Delhi	M	green
6	17	A	Mumbai	F	red

3 Meta stuff

```
[3]: data = {
      'age':      [10,22,13,21,12,11,17],
      'section':  ['A','B','C','B','B','A','A'],
```

```

    'city':          □
    ↪ ['Gurgaon', 'Delhi', 'Mumbai', 'Delhi', 'Mumbai', 'Delhi', 'Mumbai'],
    'gender':        ['M', 'F', 'F', 'M', 'M', 'M', 'F'],
    'favorite_color': ['red', 'black', 'yellow', 'pink', 'black', 'green', 'red']
}

df = pd.DataFrame(df)

print(f'df.empty:    {df.empty}')
print(f'df.shape:    {df.shape}')
print(f'df.index:    {df.index}')
print(f'df.columns:  {df.columns}')

print('\ndf.describe():')
display_df(df.describe(), index=True)

print('\ndf.info():')
df.info() # This automatically prints stuff to stdout

```

```

df.empty:    False
df.shape:    (7, 5)
df.index:    RangeIndex(start=0, stop=7, step=1)
df.columns:  Index(['age', 'section', 'city', 'gender', 'favorite_color'],
dtype='object')

```

```
df.describe():
```

	age
count	7
mean	15.1429
std	4.8795
min	10
25%	11.5
50%	13
75%	19
max	22

```

df.info():
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7 entries, 0 to 6
Data columns (total 5 columns):
#   Column      Non-Null Count  Dtype
---  -
0   age         7 non-null     int64
1   section     7 non-null     object
2   city        7 non-null     object

```

```

3   gender          7 non-null    object
4   favorite_color  7 non-null    object
dtypes: int64(1), object(4)
memory usage: 412.0+ bytes

```

4 .head() and .tail()

Both take a single optional argument, ie `n`, which is an integer representing the number of records to show. By default, it is 5. `.head()` shows the `n` top most records and `.tail()` shows the `n` bottom most records

```
[4]: display_df( df.head(3) , index=True )
      display_df( df.tail(3) , index=True )
```

	age	section	city	gender	favorite_color
0	10	A	Gurgaon	M	red
1	22	B	Delhi	F	black
2	13	C	Mumbai	F	yellow

	age	section	city	gender	favorite_color
4	12	B	Mumbai	M	black
5	11	A	Delhi	M	green
6	17	A	Mumbai	F	red

5 .iloc[]

Format is something like `DataFrame.iloc[row_indexer, column_indexer]`. Here `column_indexer` is optional. If it is not given, all columns will be printed

- `row_indexer`: This can be a slice (like `1:9:2` or `:`), or a list of the indexes, like `[1,4,5]`
- `column_indexer`: This can be a slice (like `1:9:2` or `:`), or a list of the indexes of the columns, like `[1,4,5]`

```
[5]: data = {
      'age':          [10,22,13,21,12,11,17],
      'section':      ['A','B','C','B','B','A','A'],
      'city':          ['Gurgaon','Delhi','Mumbai','Delhi','Mumbai','Delhi','Mumbai'],
      'gender':        ['M','F','F','M','M','M','F'],
      'favorite_color': ['red','black','yellow','pink','black','green','red']
    }
df = pd.DataFrame(data)

print('\nOriginal data:')
display_df()
```

```
print('\nRecords of index 1 & 3')
display_df( df.iloc[ [1,3] , : ] )
```

Original data:

age	section	city	gender	favorite_color
10	A	Gurgaon	M	red
22	B	Delhi	F	black
13	C	Mumbai	F	yellow
21	B	Delhi	M	pink
12	B	Mumbai	M	black
11	A	Delhi	M	green
17	A	Mumbai	F	red

Records of index 1 & 3

age	section	city	gender	favorite_color
22	B	Delhi	F	black
21	B	Delhi	M	pink

6 .loc[]

Just like `.iloc[]`, but uses names (strings) rather than indexes, and the slicing is end-inclusive (unlike the slicing we have seen till now)

7 .rename()

It by default doesn't do the renaming inplace, and instead returns a copy

(Here the `mapper`, `index`, and `columns` arguments accept a dict-like object or a function)

Arguments:

- **mapper**: Dict-like or function transformations to apply to that axis' values
- **axis**: `int` | `str` = 0: Axis to target with mapper. Can be 0/`index` or 1/`columns`
- **index**: Alternative to specifying axis (`mapper,axis=0` is equivalent to `index=mapper`)
- **columns**: Alternative to specifying axis (`mapper,axis=1` is equivalent to `columns=mapper`)
- **inplace**: `bool` = `False`: Whether to modify the DataFrame rather than creating a new one
- **errors**: `str`: Can be "raise" or "ignore". If "raise", then raise a `KeyError` when a dict-like mapper, index, or columns contains labels that are not present in the Index being transformed. If "ignore", existing keys will be renamed and extra keys will be ignored

```
[6]: df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
```

```
df.rename(columns={'A': 'New A', 'B': 'New B'}, inplace=False)
df.rename(index=lambda x: f'Row {x}', inplace=True)
display_df(index=True)
```

	A	B
Row 0	1	4
Row 1	2	5
Row 2	3	6

8 .query()

Can do SQL-python like queries. Returns a dataframe

The format is `DataFrame.query(expr:str, inplace:bool)`. If `inplace` is `True`, `None` is returned and the original `df` is replaced by the `df` which would had been returned if `inplace` was `False`

```
[7]: data = {
    'age':          [10,22,13,21,12,11,17],
    'section':      ['A','B','C','B','B','A','A'],
    'city':         ['Gurgaon','Delhi','Mumbai','Delhi','Mumbai','Delhi','Mumbai'],
    'gender':       ['M','F','F','M','M','M','F'],
    'favorite_color': ['red','black','yellow','pink','black','green','red']
}
df = pd.DataFrame(data)

print('\nOriginal data:')
display_df()

print('\nRecords where age >= 15:')
display_df( df.query('age >= 15') )

print('\nRecords where age >= 12 and gender = Male:')
display_df( df.query('age >= 12 and gender == "M"') )

print('\nCity and gender of people with age >= 12:')
display_df( df.query('age >= 12')[['city','gender']] )

# Use of `@` and ```
```

Original data:

age	section	city	gender	favorite_color
10	A	Gurgaon	M	red
22	B	Delhi	F	black

age	section	city	gender	favorite_color
13	C	Mumbai	F	yellow
21	B	Delhi	M	pink
12	B	Mumbai	M	black
11	A	Delhi	M	green
17	A	Mumbai	F	red

Records where age >= 15:

age	section	city	gender	favorite_color
22	B	Delhi	F	black
21	B	Delhi	M	pink
17	A	Mumbai	F	red

Records where age >= 12 and gender = Male:

age	section	city	gender	favorite_color
21	B	Delhi	M	pink
12	B	Mumbai	M	black

City and gender of people with age >= 12:

city	gender
Delhi	F
Mumbai	F
Delhi	M
Mumbai	M
Mumbai	F

9 `.sort_values()`

Arguments:

- **by:** str | list[str]: Name or list of names to sort by
- **axis:** int | str = 0: Axis to be sorted. Can be 0/index or 1/columns
- **ascending:** bool | list[bool] = True: Self explanatory. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by
- **inplace:** bool = False: If True, perform operation in-place
- **na_position:** str = "last": Puts NaNs at the beginning if **first**, and at the end if **last**
- **ignore_index:** bool = False: If True, the resulting axis will be labeled 0,1,...,n-1


```
[8]: data = {
    'age':          [10,22,13,21,12,11,17],
    'section':      ['A','B','C','B','B','A','A'],
    'city':         ['Gurgaon','Delhi','Mumbai','Delhi','Mumbai','Delhi','Mumbai'],
    'gender':       ['M','F','F','M','M','M','F'],
    'favorite_color': ['red','black','yellow','pink','black','green','red']
}
df = pd.DataFrame(data)

print('\nOriginal data:')
display_df()

print('\nSorted by age (descending):')
display_df( df.sort_values(by='age',ascending=False).head(3) , index=True )
```

Original data:

	age	section	city	gender	favorite_color
	10	A	Gurgaon	M	red
	22	B	Delhi	F	black
	13	C	Mumbai	F	yellow
	21	B	Delhi	M	pink
	12	B	Mumbai	M	black
	11	A	Delhi	M	green
	17	A	Mumbai	F	red

Sorted by age (descending):

	age	section	city	gender	favorite_color
1	22	B	Delhi	F	black
3	21	B	Delhi	M	pink
6	17	A	Mumbai	F	red

10 .count(), .sum(), min(), max(), .mean(), median(), and mode()

All of these return a `pd.core.series.Series`, except `.mode()`. Mode returns a `df` cause there might be many values that are the mode, and different rows contains these different values

10.0.1 .count()

- `axis: int | str = 0`: If 0 or “index” counts are generated for each column. If 1 or “columns”, counts are generated for each row
- `numeric_only: bool = False`: Include only float, int or boolean data

10.0.2 .sum()

- axis: int | str = 0: Axis for the function to be applied on
- numeric_only: bool = False: Include only float, int or boolean data
- skipna: bool = True: Exclude NA/null values when computing the result
- min_count: int = 0: The required number of valid values to perform the operation. If fewer non-NA values are present, the result will be NA

10.0.3 .min(), .max(), .mean(), .median(), & .mode()

- axis: int | str = 0: Axis for the function to be applied on
- numeric_only: bool = False: Include only float, int or boolean data
- skipna: bool = True: Exclude NA/null values when computing the result

```
[9]: data = {
      "Person": ["John", "Myla", "Lewis", "John", "Myla"],
      "Age":    [24, np.nan, 21, 33, 26],
      "Single": [False, True, True, True, False]
    }
    df = pd.DataFrame(data)

    print('Original df:')
    display_df()

    print('\n1) Count:')
    display_df( df.count(), index=True)
    display_df( df.count(numeric_only=True), index=True)

    print('\n2) Sum:')
    display_df( df.sum(), index=True)

    print('\n3) Min:')
    display_df( df.min(), index=True)

    print('\n4) Max:')
    display_df( df.max(), index=True)

    print('\n5) Mean:')
    display_df( df.mean(numeric_only=True), index=True)

    print('\n6) Median:')
    display_df( df.median(numeric_only=True), index=True)

    print('\n7) Mode:')
    display_df( df.mode(numeric_only=True), index=True)
```

Original df:

Person	Age	Single
John	24	False
Myla	nan	True
Lewis	21	True
John	33	True
Myla	26	False

1) Count:

	0
Person	5
Age	4
Single	5

	0
Age	4
Single	5

2) Sum:

	0
Person	JohnMylaLewisJohnMyla
Age	104.0
Single	3

3) Min:

	0
Person	John
Age	21.0
Single	False

4) Max:

	0
Person	Myla
Age	33.0

	0
Single	True

5) Mean:

	0
Age	26
Single	0.6

6) Median:

	0
Age	25
Single	1

7) Mode:

	Age	Single
0	21	1
1	24	nan
2	26	nan
3	33	nan

11 .groupby()

Returns a `pandas.core.groupby.generic.DataFrameGroupBy` object

This is not supposed to be printed or displayed, but we can apply various aggregation methods to it like `mean`, `count`, `median`, `sum`, etc. These will return a dataframe

Note: `mode` is not there

```
[10]: # Create a DataFrame
data = {
    'Category': ['A', 'B', 'A', 'B', 'A', 'B'],
    'Value':    [10, 20, 15, 25, 20, 30]
}
df = pd.DataFrame(data)
print('Original df:')
display_df()

# Group the DataFrame by 'Category' column
```

```
grouped = df.groupby('Category')

# Perform aggregation operations on the groups
mean_age = grouped.mean()
total_count = grouped.count()

# Print the results
print('Mean:')
display_df(mean_age, index=True)
print('Total:')
display_df(total_count, index=True)
```

Original df:

Category	Value
A	10
B	20
A	15
B	25
A	20
B	30

Mean:

Category	Value
A	15
B	25

Total:

Category	Value
A	3
B	3

12 `.select_dtypes()`

Returns a dataframe containing only the columns which satisfy the inclusive the exclusive conditions. It takes in the following arguments:

- **includes:** scalar or list-like = None: If a scalar value is passed, the columns must be of that datatype. Else if a list-like object is passed, the columns must be one of those datatypes
- **excludes:** scalar or list-like = None: Similar logic as **includes**

Will raise a `ValueError` if both **include** and **exclude** are empty, contain overlapping elems, or contain a string datatype

To select all numeric datatypes, do `np.number` or `"number"`

```
[11]: df = pd.DataFrame({
        'Integers': [1, 2, 3],
        'Floats': [4.0, 5.0, 6.0],
        'Strings (Objects)': ['a', 'b', 'c'],
        'Booleans': [True, False, True]
    })

    print("include='int'")
    display_df(df.select_dtypes(include='int'))

    print("include='float'")
    display_df(df.select_dtypes(include='float'))

    print("include=[bool, 'object']")
    display_df(df.select_dtypes(include=[bool, 'object']))

    print("include='number'")
    display_df(df.select_dtypes(include='number'))
```

include='int'

Integers
1
2
3

include='float'

Floats
4
5
6

include=[bool, 'object']

Strings (Objects)	Booleans
a	True
b	False
c	True

include='number'

Integers	Floats
1	4
2	5
3	6

13 .duplicated() and drop_duplicates()

13.0.1 .duplicated()

Returns a boolean Series denoting duplicate rows. Arguments:

- **subset:** column label or sequence of labels = None: Only consider certain columns for identifying duplicates. Duplicates will have the same value in ALL these columns. By default use all of the columns
- **keep:** {'first','last',False} = 'first': Determines which duplicates (if any) to mark as False. **first** marks all duplicates as True except for the first occurrence. **last** marks all duplicates as True except for the last occurrence. **False** marks all duplicates as True

13.0.2 .drop_duplicates

- **subset:** column label or sequence of labels = None: Only consider certain columns for identifying duplicates. Duplicates will have the same value in ALL these columns. By default use all of the columns
- **keep:** {'first','last',False} = 'first': Determines which duplicates (if any) to keep. **first** drops all duplicates except for the first occurrence. **last** drops all duplicates except for the last occurrence. **False** drops all duplicates
- **inplace:** bool = False: Whether to modify the DataFrame rather than creating a new one
- **ignore_index:** bool = False: If True, the resulting axis will be labeled 0,1,...,n-1. Else, the indexes shall be presevered inshallah

```
[12]: df = pd.DataFrame({
    'Name': ['John', 'Doe', 'John', 'Doe', 'Anna'],
    'City': ['New York', 'Los Angeles', 'New York', 'Los Angeles', 'Chicago'],
    'Age': [28, 35, 28, 35, 29],
})

df['Duplicate'] = df.duplicated()
print('Original df:')
display_df(df, index=True)

df = df.drop_duplicates(subset=['Name','City'])
print('\nAfter dropping duplicates:')
display_df(df, index=True)
```

Original df:

	Name	City	Age	Duplicate
0	John	New York	28	False
1	Doe	Los Angeles	35	False
2	John	New York	28	True
3	Doe	Los Angeles	35	True
4	Anna	Chicago	29	False

After dropping duplicates:

	Name	City	Age	Duplicate
0	John	New York	28	False
1	Doe	Los Angeles	35	False
4	Anna	Chicago	29	False

14 pd.concat() and .append()

14.0.1 pd.concat

- **objs**: List of the dataframes to combine/concat
- **axis**: {0, 'index', 1, 'column'} = 0: 0 or index concatenates the dfs vertically, ie on top of each other. 1 or column concatenates the df horizontally, ie side by side
- **ignore_index**: bool = False: If True, do not use the index values along the concatenation axis. The resulting axis will be labeled 0,1,...,n-1. This is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information. Note the index values on the other axes are still respected in the join
- **copy**: bool = True: If False, do not copy data unnecessarily

14.0.2 .append()

IT HAS BEEN DEPRECIATED SINCE 1.4.0. The current version when writing this is 2.2.1, so if your examiner asks about this, call them a boomer

```
[13]: # Joining two dfs vertically
df1 = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})
df2 = pd.DataFrame({
    'B': [7, 8, 9],
    'C': [4, 5, 6]
})

new_df = pd.concat([df1, df2], axis=0)
display_df(new_df, index=True)
```



```

# Joining two dfs horizontally
df1 = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [11, 22, 33]
})

df2 = pd.DataFrame({
    'C': [111, 222, 333],
    'D': [1111, 2222, 3333]
})

new_df = pd.concat([df1, df2], axis=1)
display_df(new_df)

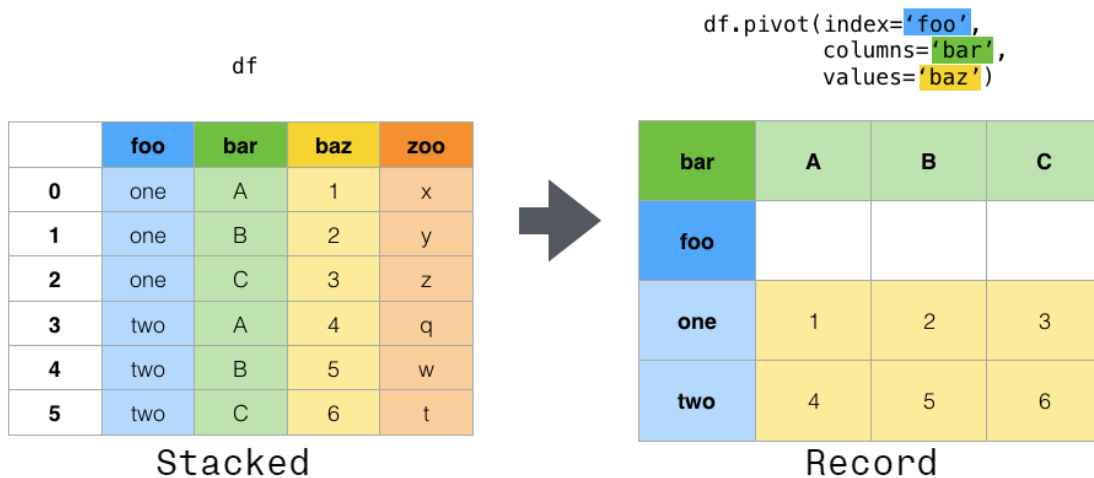
```

	A	B	C
0	1	4	nan
1	2	5	nan
2	3	6	nan
0	nan	7	4
1	nan	8	5
2	nan	9	6

A	B	C	D
1	11	111	1111
2	22	222	2222
3	33	333	3333

15 .pivot() and .pivot_table

Pivot



A `ValueError` will be thrown in `.pivot` when there are any index+column combinations with multiple values. In such cases, `.pivot_table` will be used (ie when you need to aggregate)

15.0.1 .pivot()

- `index: str | list[str] = _NoDefault.no_default`: Column whose values will be the new df's index. The index-col-name will also be changed to this col name. If not given, existing index is used
- `columns: str | list[str]`: Column whose values will be the new df's columns
- `values: str | list[str] = _NoDefault.no_default`: Column(s) to use for populating new frame's values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns

15.0.2 .pivot_table

- [Same args as `.pivot()`]
- `aggfunc: str | function | list[function] | dict = "mean"`: If a list of functions is passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves). If a dict is passed, the key is column to aggregate and the value is function or list of functions
- `fill_value: scalar = None`: Value to replace missing values with (in the resulting pivot table, after aggregation)

```
[14]: df = pd.DataFrame({
    'foo': ['one', 'one', 'one', 'two', 'two', 'two'],
    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
    'baz': [1, 2, 3, 4, 5, 6],
    'zoo': ['x', 'y', 'z', 'q', 'w', 't']
})
```

```

})

pivoted_df1 = df.pivot(index='foo', columns='bar', values='baz')
pivoted_df2 = df.pivot(index='foo', columns='bar')

print('Original df:')
display_df(df)
print('\n.n.pivot() dfs:')
display_df(pivoted_df1, index=True)
print(pivoted_df2)

print('\n\n')

df = pd.DataFrame({
    'Day':      ['Monday', 'Monday', 'Monday', 'Tuesday', 'Tuesday',
    ↪ 'Tuesday'],
    'City':     ['Delhi', 'Delhi', 'Mumbai', 'Delhi', 'Mumbai', 'Mumbai'],
    'Temperature': [32, 33, 36, 33, 36, 37],
})
pivot_table_df = df.pivot_table(index='Day', columns='City',
    ↪ values='Temperature', aggfunc='count')

print('\nOriginal df: ')
display_df(df)
print('\n.n.pivot_table() df: ')
display_df(pivot_table_df, index=True)

```

Original df:

foo	bar	baz	zoo
one	A	1	x
one	B	2	y
one	C	3	z
two	A	4	q
two	B	5	w
two	C	6	t

.pivot() dfs:

foo	A	B	C
one	1	2	3
two	4	5	6

```

      baz      zoo
bar   A  B  C   A  B  C
foo
one    1  2  3   x  y  z
two    4  5  6   q  w  t

```

Original df:

Day	City	Temperature
Monday	Delhi	32
Monday	Delhi	33
Monday	Mumbai	36
Tuesday	Delhi	33
Tuesday	Mumbai	36
Tuesday	Mumbai	37

`.pivot_table()` df:

Day	Delhi	Mumbai
Monday	2	1
Tuesday	1	2

16 `.value_counts()`

Basically the frequency of each unique record/value

- **subset:** `str | list[str] = None`: Columns to use when counting unique combinations. If `None` (default), use all columns
- **normalize:** `bool = False`: If `True`, return proportions (as in floats) rather than frequencies
- **sort:** `bool = True`: Sort by frequencies when `True`. Sort by DataFrame column values when `False`
- **ascending:** `bool = False`: If `True`, sort in ascending order
- **dropna:** `bool = True`: If `True`, don't include counts of rows that contain NA values

```

[15]: df = pd.DataFrame({
      'Name': ['John', 'Doe', 'John', 'Doe', 'Anna'],
      'Age':  [28, 35, 28, 35, 29],
      'City': ['New York', 'Los Angeles', 'New York', 'Los Angeles', 'Chicago']
    })

counts = df.value_counts()
display_df(counts, index=True)

```

```
print('\n')
counts = df.value_counts(subset='City')
display_df(counts, index=True)
counts = df.value_counts(subset=['City'], normalize=True)
display_df(counts, index=True)
```

	count
('Doe', 35, 'Los Angeles')	2
('John', 28, 'New York')	2
('Anna', 29, 'Chicago')	1

City	count
Los Angeles	2
New York	2
Chicago	1

	proportion
('Los Angeles',)	0.4
('New York',)	0.4
('Chicago',)	0.2

17 .get_dummies()

18 .quantile()

When a float is passed into `q`, a series is passed where the indexes are the col names and the col name (of the only “column”) is the float passed. When a list of floats is passed, a dataframe is returned with the indexes being the floats passed and the column names are the columns of

- `q`: float | array-like = 0.5: Value between $0 \leq q \leq 1$, the quantile(s) to compute
- `axis`: {0, 'index', 1, 'columns'} = 0: 0/'index' for row-wise, 1/'columns' for column-wise
- `numeric_only`: bool = False: Include only float, int or boolean columns
- `interpolation`: {'linear', 'lower', 'higher', 'midpoint', 'nearest'} = 'linear': This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points `i` and `j`:
 - `linear`: $i + (j-i) * \text{fraction}$, where fraction is the fractional part of the index surrounded by `i` and `j`
 - `lower`: `i`
 - `higher`: `j`

- nearest: i or j whichever is nearest
- midpoint: (i + j) / 2

```
[16]: df = pd.DataFrame({
      'A': [1,2,3,4,5,6,7,8,9,10]
    })

display_df(df.quantile(0.25), index=True)
display_df(df.quantile([0.25, 0.5, 0.75]), index=True)
```

	0.25
A	3.25

	A
0.25	3.25
0.5	5.5
0.75	7.75

19 .fillna()

- **value:** scalar | dict | Series | DataFrame: Value to use to fill holes . Alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). Values not in the dict/Series/DataFrame will not be filled. This value cannot be a list
- **inplace:** bool = False: If True, fill inplace
- **limit:** int | None = None: If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None

20 .var() / .std()

For **.var()**, return (unbiased) variance over requested axis. For **.std()**, return sample standard deviation over requested axis

- **axis:** {0,1} = 0: 0 - variance of each column. 1 - variance of each row/index. For series this parameter is unused and defaults to 0
- **skipna:** bool = True: Exclude NA/null values. If an entire row/column is NA, the result will be NA
- **numeric_only:** bool = False: Include only float, int, boolean columns. Not implemented for series

Remember, the formula for variance is

$$\sigma^2 = \frac{\sum_{i=1}^n (x_i - \mu)^2}{n}$$

where - x_i represents the i-th value - μ represents the population mean - n represents the total number of values in the population

21 `.reset_index()`

Reset the index of the dataframe, and use the default one instead

- `inplace: bool = False`: Whether to modify the dataframe rather than creating a new one

22 Series

`.isin()` `.astype()` `.apply()`