Zen of python

The ways to code in python, the rules.

eXploring data

```
sales
      store type  department        date  weekly_sales  is_holiday  temperature_c  fuel_price_usd_per_l  unemployment
0         1    A           1  2010-02-05      24924.50       False          5.728                 0.679         8.106
1         1    A           1  2010-03-05      21827.90       False          8.056                 0.693         8.106
2         1    A           1  2010-04-02      57258.43       False         16.817                 0.718         7.808
3         1    A           1  2010-05-07      17413.94       False         22.528                 0.749         7.808
4         1    A           1  2010-06-04      17558.09       False         27.050                 0.715         7.808
...     ...  ...         ...         ...           ...         ...            ...                   ...           ...
```

to find how many unique values are in a column

```
print(sales["type"].unique())
```

The output: `['A' 'B']`

to find how many unique values are in every column

Use .agg()

```
for col in data.columns:
    un=data[col].unique()
    print("\n\nUnique Values in {}:\n{}".format(col,un))
```

Transforming DataFrame

1. .head()
2. .info()

Name of columns and whether there are any missing value
shows information on each of the columns, such as the data type and number of missing values.

3. .describe()

Mean, min , median, count std...

4. .values()

Data values in two dimension array

```
>>> df
   age  height  weight
0    3      94      31
1   29     170     115
>>> df.values
array([[  3,  94,  31],
       [ 29, 170, 115]])
```

5. To access row and columns
    a. .columns() for columns
    b. .index() for rows

Sorting and subsetting

```
homelessness


                region                 state  individuals  family_members  state_pop
0   East South Central               Alabama       2570.0           864.0    4887681
1              Pacific                Alaska       1434.0           582.0     735139
2             Mountain               Arizona       7259.0          2606.0    7158024
3   West South Central              Arkansas       2280.0           432.0    3009733
4              Pacific            California     109008.0         20964.0   39461588
5             Mountain              Colorado       7607.0          3250.0    5691287
6          New England           Connecticut       2280.0          1696.0    3571520
7       South Atlantic              Delaware        708.0           374.0     965479
8       South Atlantic  District of Columbia       3770.0          3134.0     701547
```

1. **.sort_values** (columns name,ascending =false/true)
   a. #Sort homelessness first by region (ascending), and then by number of family members (descending). Save this as homelessness_reg_fam.
      i.
```
homelessness_reg_fam =  homelessness.sort_values(
["region","family_members"],ascending=[True,False])
```
2. .sort_values([columns name in list] ,ascending =false/true)
3. .isin
   a. Filter homelessness for cases where the USA census state is in the list of Mojave states, canu, assigning to mojave_homelessness
      i.
```
mojave_homelessness = homelessness[homelessness["state"].isin
(["California", "Arizona", "Nevada", "Utah"])]
```

Accessing new column

```
      name        breed  color  height_cm  weight_kg date_of_birth  height_m
0    Bella     Labrador  Brown         56         24    2013-07-01      0.56
1  Charlie       Poodle  Black         43         24    2016-09-16      0.43
2     Lucy    Chow Chow  Brown         46         24    2014-08-25      0.46
3   Cooper    Schnauzer   Gray         49         17    2011-12-11      0.49
4      Max     Labrador  Black         59         29    2017-01-20      0.59
5   Stella    Chihuahua    Tan         18          2    2015-04-20      0.18
6   Bernie  St. Bernard  White         77         74    2018-02-27      0.77
```

1. Adding a new column
   a.
```
dogs["height_m"] = dogs["height_cm"] / 100
```
2. Add a column to homelessness, indiv_per_10k, containing the number of homeless individuals per ten thousand people in each state.
   a.
```
homelessness["indiv_per_10k"] = 10000 * homelessness
["individuals"] / homelessness['state_pop']
```
3. Sort high_homelessness by descending indiv_per_10k, assigning to high_homelessness_srt.

a.
```
high_homelessness_srt = high_homelessness.sort_values
('indiv_per_10k', ascending = False)
```

Aggregating DataFrames

```
sales
       store type  department        date  weekly_sales  is_holiday  temperature_c  fuel_price_usd_per_l  unemployment
0          1    A           1  2010-02-05      24924.50       False          5.728                 0.679         8.106
1          1    A           1  2010-03-05      21827.90       False          8.056                 0.693         8.106
2          1    A           1  2010-04-02      57258.43       False         16.817                 0.718         7.808
3          1    A           1  2010-05-07      17413.94       False         22.528                 0.749         7.808
4          1    A           1  2010-06-04      17558.09       False         27.050                 0.715         7.808
...      ...  ...         ...         ...           ...         ...            ...                   ...           ...
```

# Print the head of the sales DataFrame
print(sales.head)

# Print the info about the sales DataFrame
print(sales.info())

# Print the mean of weekly_sales
print(sales.mean())

# Print the median of weekly_sales
print(sales.median())

# Print the maximum of the date column
print(sales["date"].max())

# Print the minimum of the date column
print(sales["date"].min)

The .agg() method

```
df = pd.DataFrame({'X':[78,85,96,80,86], 'Y':[84,94,89,83,86],'Z':[86,97,96,72,83]});
```

df dataframe

```
    X   Y   Z
0  78  84  86
1  85  94  97
2  96  89  96
3  80  83  72
4  86  86  83
```

- allows compute custom summary statistics.

```
def iqr(column):
    return column.quantile(0)
```

```
def iqr(column):
    return column.quantile(1)
```

- In the custom function for this exercise, "IQR" is short for inter-quartile range, which

is the 75th percentile minus the 25th percentile. It's an alternative to standard deviation that is helpful if your data contains outliers.
- These functions computes 00 and 100 percentage of parameter "column" and returns the expected outcome.

```
def iqr(column):
    return column.quantile(0)

print(df["X"].agg(iqr))
```

For this the output will be : 78.0

```
def iqr(column):
    return column.quantile(1)

print(df["X"].agg(iqr))
```

For this the output will be : 96.0

```
def iqr(column):
    return column.quantile(.75) - column.
quantile(.25)

print(df["X"].agg(iqr))
```

Output of this will be 86-80=6

1. Multiple aggregate function

```
dogs["weight_kg"].agg([pct30, pct40])
```

```
pct30    22.6
pct40    24.0
Name: weight_kg, dtype: float64
```

2. Summaries on multiple columns

```
dogs[["weight_kg", "height_cm"]].agg(pct30)
```

```
weight_kg    22.6
height_cm    45.4
dtype: float64
```

Cumulative statistics

# Sort sales_1_1 by date
sales_1_1 = sales_1_1.sort_values("date",ascending=True)

Get the cumulative sum of weekly_sales, add as cum_weekly_sales col
sales_1_1["cum_weekly_sales"] = sales_1_1["weekly_sales"].cumsum()

Get the cumulative max of weekly_sales, add as cum_max_sales col
sales_1_1["cum_max_sales"] = sales_1_1["weekly_sales"].cummax()

# See the columns you calculated
print(sales_1_1 [["date", "weekly_sales", "cum_weekly_sales", "cum_max_sales"]])

Drop duplicate store/type combinations

```
store_types = sales.drop_duplicates(["store","type"])
print(store_types.head())
```

#Count the proportion of different departments in store_depts, sorting the proportions in descending order.

```
dept_props_sorted = store_depts["department"].
value_counts(sort=True, normalize=True)
```

Grouped summary statistics

```
sales
      store type  department        date  weekly_sales  is_holiday  temperature_c  fuel_price_usd_per_l  unemployment
0         1    A           1  2010-02-05      24924.50       False          5.728                 0.679         8.106
1         1    A           1  2010-03-05      21827.90       False          8.056                 0.693         8.106
2         1    A           1  2010-04-02      57258.43       False         16.817                 0.718         7.808
3         1    A           1  2010-05-07      17413.94       False         22.528                 0.749         7.808
4         1    A           1  2010-06-04      17558.09       False         27.050                 0.715         7.808
...     ...  ...         ...         ...           ...         ...            ...                   ...           ...
```

#Group sales by "type", take the sum of "weekly_sales", and store as sales_by_type.
To do this we have to do it in two part the 1st part is grouping, and the 2nd part is showing sum of
The individuals
1st part: `sales.groupby("type")`
2nd part: `["weekly_sales"].sum()`

```
sales_by_type = sales.groupby("type")
["weekly_sales"].sum()
```

#get proportion of each type

1. Group sales by "type" and "is_holiday", take the sum of weekly_sales, and store as Sales_by_type_is_holiday.

```
sales_by_type_is_holiday = sales.groupby(["type",
"is_holiday"])["weekly_sales"].sum()
print(sales_by_type_is_holiday)
```

```
type  is_holiday
A     False         2.337e+08
      True          2.360e+04
B     False         2.318e+07
      True          1.621e+03
Name: weekly_sales, dtype: float64
```

2. Calculate the proportion of sales at each store type by dividing by the sum of sales_by_type. Assign to sales_propn_by_type.

```
sales_propn_by_type = sales_by_type[["A", "B"]] /
sales_by_type.sum()
```

#use of .groupby()

1. Get the min, max, mean, and median of unemployment and fuel_price_usd_per_l for each store type. Store this as unemp_fuel_stats.

```
unemp_fuel_stats = sales.groupby("type")
["unemployment","fuel_price_usd_per_l"].agg([np.
min,np.max,np.mean,np.median])
```

```
        unemployment                          fuel_price_usd_per_l
                amin   amax   mean median                     amin   amax   mean median
type
A              3.879  8.992  7.973  8.067                    0.664  1.107  0.745  0.735
B              7.170  9.765  9.279  9.199                    0.760  1.108  0.806  0.803
```

2. Get the min, max, mean, and median of weekly_sales for each store type using .groupby() and .agg(). Store this as sales_stats. Make sure to use numpy functions!

```
sales_stats = sales.groupby("type")
["weekly_sales"].agg([np.min,np.max,np.mean,np.
median])
```

```
         amin       amax       mean    median
type
A     -1098.0  293966.05  23674.667  11943.92
B      -798.0  232558.51  25696.678  13336.08
```

#use of pivot | alternative of groupby()

1. `dogs.groupby("color")["weight_kg"].mean()`

   Instead of using this command, in pivot we will use

```
dogs.pivot_table(values="weight_kg",
                 index="color")
```

   Both will have same ans

```
        weight_kg
color
Black        26.5
Brown        24.0
Gray         17.0
Tan           2.0
White        74.0
```

2. Summing the pivot table

   If we set the margins argument to True, the last row and last column of the pivot table contain the mean of all the values in the column or row.

```
dogs.pivot_table(values="weight_kg", index="color", columns="breed",
                 fill_value=0, margins=True)
```

breed	Chihuahua	Chow Chow	Labrador	Poodle	Schnauzer	St. Bernard	All
color							
Black	0	0	29	24	0	0	26.500000
Brown	0	24	24	0	0	0	24.000000
Gray	0	0	0	0	17	0	17.000000
Tan	2	0	0	0	0	0	2.000000
White	0	0	0	0	0	74	74.000000
All	2	24	26	24	17	74	27.714286

3.
```
dogs.pivot_table(values="weight_kg", index="color", aggfunc=[np.mean, np.median])
```
To get multiple summary statistics at a time, we can pass a list of functions to the aggfunc argument. Here, we get the mean and median for each dog color.

	mean	median
	weight_kg	weight_kg
color		
Black	26.5	26.5
Brown	24.0	24.0
Gray	17.0	17.0
Tan	2.0	2.0
White	74.0	74.0

Example:

1. Get the mean and median (using NumPy functions) of `weekly_sales` by `type` using `.pivot_table()` and store as `mean_med_sales_by_type`.
```
mean_med_sales_by_type = sales.pivot_table
(index="type",values="weekly_sales",aggfunc=[np.
mean,np.median])
```

	mean	median
	weekly_sales	weekly_sales
type		
A	23674.667	11943.92
B	25696.678	13336.08

2. Get the mean of `weekly_sales` by `type` and `is_holiday` using `.pivot_table()` and store as `mean_sales_by_type_holiday`.
```
mean_sales_by_type_holiday = sales.pivot_table
(index=["type","is_holiday"],
values="weekly_sales")
```

		weekly_sales
type	is_holiday	
A	False	23768.584
	True	590.045
B	False	25751.981
	True	810.705

Or
```
mean_sales_by_type_holiday = sales.pivot_table
(index="type",columns="is_holiday",
values="weekly_sales")
```

is_holiday	False	True
type		
A	23768.584	590.045
B	25751.981	810.705

3. Print the mean `weekly_sales` by `department` and `type`, filling in any missing values with `0` and summing all rows and columns.

```python
print(sales.pivot_table(values="weekly_sales",
index="department",fill_value=0, columns="type",
margins=True))
```

```
type                 A           B         All
department
1            30961.725    44050.627   32052.467
2            67600.159   112958.527   71380.023
3            17160.003    30580.655   18278.391
4            44285.399    51219.654   44863.254
5            34821.011    63236.875   37189.000
...                ...          ...         ...
96           21367.043     9528.538   20337.608
97           28471.267     5828.873   26584.401
98           12875.423      217.428   11820.590
99             379.124        0.000     379.124
```

Slicing and indexing DataFrames

Indexes

Indexes are controversial. Although they simplify subsetting code, there are some downsides. Index values are just data. Storing data in multiple forms makes it harder to think about. There is a concept called "tidy data," where data is stored in tabular form - like a DataFrame. Each row contains a single observation, and each variable is stored in its own column. Indexes violate the last rule since index values don't get their own column. In pandas, the syntax for working with indexes is different from the syntax for working with columns. By using two syntaxes, your code is more complicated, which can result in more bugs. If you decide you don't want to use indexes, that's perfectly reasonable. However, it's useful to know how they work for cases when you need to read other people's code.

DataFrames are composed of three parts: a NumPy array for the data, and two indexes to store the row and column details.

```
print(temperatures)

            date     city        country  avg_temp_c
0     2000-01-01  Abidjan  Côte D'Ivoire      27.293
1     2000-02-01  Abidjan  Côte D'Ivoire      27.685
2     2000-03-01  Abidjan  Côte D'Ivoire      29.061
3     2000-04-01  Abidjan  Côte D'Ivoire      28.162
4     2000-05-01  Abidjan  Côte D'Ivoire      27.547
...          ...      ...            ...         ...
16495 2013-05-01     Xian          China      18.979
16496 2013-06-01     Xian          China      23.522
```

#setting a column as index

```python
dogs_ind = dogs.set_index("name")
print(dogs_ind)
```

```
         breed    color  height_cm  weight_kg
name
Bella    Labrador  Brown        56         25
Charlie    Poodle  Black        43         23
Lucy    Chow Chow  Brown        46         22
Cooper  Schnauzer   Grey        49         17
Max      Labrador  Black        59         29
Stella  Chihuahua    Tan        18          2
Bernie St. Bernard White        77         74
```

#to reset/revert this change use

```
dogs_ind.reset_index()
```

```
      name        breed  color  height_cm  weight_kg
0    Bella     Labrador  Brown         56         25
1  Charlie       Poodle  Black         43         23
2     Lucy    Chow Chow  Brown         46         22
3   Cooper    Schnauzer   Grey         49         17
4      Max     Labrador  Black         59         29
5   Stella    Chihuahua    Tan         18          2
6   Bernie  St. Bernard  White         77         74
```

#setting multiple column as index

```
temperatures_ind = temperatures.set_index(["country","city"])
```

```
temperatures.loc[("Brazil","Rio De Janeiro"),("Pakistan", "Lahore")]
```

#sorting

```
dogs_srt = dogs.set_index(["breed", "color"]).sort_index()
```

```
                        name  height_cm  weight_kg
breed       color
Chihuahua   Tan       Stella         18          2
Chow Chow   Brown       Lucy         46         22
Labrador    Black        Max         59         29
            Brown      Bella         56         25
Poodle      Black    Charlie         43         23
Schnauzer   Grey      Cooper         49         17
St. Bernard White     Bernie         77         74
```

Sort `temperatures_ind` by the index values.

```
print(temperatures_ind.sort_index())
```

Sort `temperatures_ind` by the index values at the `"city"` level.

```
print(temperatures_ind.sort_index(level=["city"]))
```

Sort `temperatures_ind` by ascending country then descending city.

```
print(temperatures_ind.sort_index(level=["country", "city"], ascending=[True, False]))
```

#time series in

#pivot table

dt can be used to access the values of the series as datetimelike and return several properties

`dataframe["column"].dt.year`

`dataframe["column"].dt.month`

Feature Engineering

There are two type of feature engineering

1. Nominal encoding
    a. One hot encoding
    b. One hot encoding with many categorical variable
    c. Mean encoding
    - And many more.
2. Ordinal encoding
    a. Label encoding
    b. Target guided ordinal encoding.

Nominal encoding

Ordinal encoding

This is more about ranking before encoding.