Python 2 - Object Oriented Programming and Pandas

4 Pillars of OOP

- Encapsulation: Group related variables and functions together to reduce complexity and increase reusability
- Data Abstraction: Creating methods to interface with attributes of your class. Show only essentials to reduce complexity
- Inheritance
- Polymorphism

Inheritance

- New classes do not need to be declared from scratch. They may build on existing classes
- When one class inherits from another, it automatically takes on all the attributes and methods of the first class
- Goal: Eliminate redundant code by inheriting attributes and methods from a parent class

In [1]:

```
class Car():
    """A simple attempt to represent a car."""
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def read_odometer(self):
        print("This car has " + str(self.odometer_reading) + " miles on it.")

    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        self.odometer_reading += miles
```

In [2]:

```
class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""
    def __init__(self, make, model, year):
        """Initialize attributes of the parent class."""
        super().__init__(make, model, year)
```

In [3]:

```
my_tesla = ElectricCar('tesla', 'model s', 2016)
print(my_tesla.get_descriptive_name())
```

```
2016 Tesla Model S
```

In [4]:
```
my_tesla.increment_odometer(10)
my_tesla.read_odometer()
```

```
This car has 10 miles on it.
```

Polymorphism

- Because child classes inherit all attributes and methods from their parent class, we may wish to refactor and customize classes to specific use cases.
- Overiding involves the redefining of methods to better suit child classes

In [5]:
```
class GasCar(Car):
    def __init__(self, make, model, year):
        """Initialize attributes of the parent class."""
        super().__init__(make, model, year)

    def get_descriptive_name(self):
        long_name = str(self.year) + ' ' + self.make + ' '\
                    + self.model + " is a gas car"
        return long_name.title()
```

In [6]:
```
my_bmw = GasCar('BMW', 'i8', 2015)
print(my_bmw.get_descriptive_name())
```

```
2015 Bmw I8 Is A Gas Car
```

Pandas

In [7]:
```
import pandas as pd
%matplotlib inline
```

Reading CVS Files

- Function to use in Pandas: read_csv()
- Value passed to read_csv() must be string and the **exact** name of the file
- CSV Files must be in the same directory as the python file/notebook

In [8]:
```
df = pd.read_csv("imports - Sheet1.csv")
#read_excel also an option

#print(df)
```

Basic DataFrame Functions

- head() will display the first 5 values of the DataFrame
- tail() will display the last 5 values of the DataFrame
- shape will display the dimensions of the DataFrame
- columns() will return the columns of the DataFrame as a list
- dtypes will display the types of each column of the DataFrame
- drop() will remove a column from the DataFrame

```
In [9]: df.head()
```

Out[9]:

	year	country_origin_id	country_destination_id	hs92_product_id	export_val	export_val_pct
0	1995	VNM	BFA	ALL	67177.77	0.00%
1	1995	VNM	CAF	ALL	514674.15	0.00%
2	1995	VNM	CIV	ALL	58011.71	0.00%
3	1995	VNM	CMR	ALL	97669.00	0.00%
4	1995	VNM	COG	ALL	24018.39	0.00%

```
In [39]: df.tail()
```

Out[39]:

	year	country_origin_id	country_destination_id	hs92_product_id	export_val	export_val_pct
2425	2015	VNM	ECU	ALL	4412351.39	0.01%
2426	2015	VNM	GUY	ALL	7137466.15	0.02%
2427	2015	VNM	PER	ALL	280650.31	0.00%
2428	2015	VNM	PRY	ALL	16496727.35	0.05%
2429	2015	VNM	URY	ALL	206349.39	0.00%

```
In [40]: df.shape
```

```
Out[40]: (2430, 6)
```

```
In [41]: df.columns
```

```
Out[41]: Index(['year', 'country_origin_id', 'country_destination_id',
                'hs92_product_id', 'export_val', 'export_val_pct'],
               dtype='object')
```

In [28]:

```
df.columns = ["year", "country origin", "country destination",
              "product", "export_val", "export_val_pct"]

df.head()
```

Out[28]:

	year	country origin	country destination	product	export_val	export_val_pct
0	1995	VNM	BFA	ALL	67177.77	0.00%
1	1995	VNM	CAF	ALL	514674.15	0.00%
2	1995	VNM	CIV	ALL	58011.71	0.00%
3	1995	VNM	CMR	ALL	97669.00	0.00%
4	1995	VNM	COG	ALL	24018.39	0.00%

In [14]:

```
df.dtypes
```

Out[14]:

```
year                       int64
country origin            object
country destination       object
product                   object
export_val               float64
export_val_pct            object
dtype: object
```

Indexing and Series Functions

- Columns of a DataFrame can be accessed through the following format: df_name["name_of_column"]
- Columns will be returned as a Series, which have different methods than DataFrames
- A couple useful Series functions: max(), median(), min(), value_counts(), sort_values()

In [15]:

```
df["export_val"]

df["export_val"].max()
```

Out[15]:

```
2718394688.0
```

In [16]:

```
df["export_val"].median()
```

Out[16]:

```
767979.0700000001
```

In [17]:

```
df["export_val"].min()
```

Out[17]:

```
1000.0
```

In [18]:
```
df["year"].value_counts()
```

Out[18]:
```
2007    131
2005    131
2006    129
2008    124
2003    124
2004    124
2009    123
2010    121
2000    120
2002    120
2001    119
2011    116
1999    114
2012    114
2015    109
2014    109
2013    108
1998    108
1997    101
1996     98
1995     87
Name: year, dtype: int64
```

In [19]:
```
df.sort_values(by = "year", ascending = True)
df.head()
```

Out[19]:

	year	country origin	country destination	product	export_val	export_val_pct
0	1995	VNM	BFA	ALL	67177.77	0.00%
1	1995	VNM	CAF	ALL	514674.15	0.00%
2	1995	VNM	CIV	ALL	58011.71	0.00%
3	1995	VNM	CMR	ALL	97669.00	0.00%
4	1995	VNM	COG	ALL	24018.39	0.00%

In [20]:
```
# delete one column
df.drop("export_val_pct", 1).head()
```

Out[20]:

	year	country origin	country destination	product	export_val
0	1995	VNM	BFA	ALL	67177.77
1	1995	VNM	CAF	ALL	514674.15
2	1995	VNM	CIV	ALL	58011.71
3	1995	VNM	CMR	ALL	97669.00
4	1995	VNM	COG	ALL	24018.39

In [21]:
```
# delete multiple columns
df.drop(["export_val_pct", "product"], 1, inplace = True)
```

```
In [22]: df.head()
```

```
Out[22]:
      year  country origin  country destination  export_val
   0  1995             VNM                  BFA    67177.77
   1  1995             VNM                  CAF   514674.15
   2  1995             VNM                  CIV    58011.71
   3  1995             VNM                  CMR    97669.00
   4  1995             VNM                  COG    24018.39
```

Indexing

- Because Pandas will select entries based on column values by default, selecting data based on row values requires the use of the iloc method.
- Allowed inputs are:
    - An integer, e.g. 5.
    - A list or array of integers, e.g. [4, 3, 0].
    - A slice object with ints, e.g. 1:7.

```
In [20]: #Retrieve a couple rows from their index values
         df.iloc[[0]]
         df.iloc[[0, 1]]
```

```
Out[20]:
      year  country_origin_id  country_destination_id  hs92_product_id  export_val  export_val_pct
   0  1995                VNM                     BFA              ALL    67177.77           0.00%
   1  1995                VNM                     CAF              ALL   514674.15           0.00%
```

```
In [22]: #Similar to arrays, we can use splicing to access multiple rows
         df.iloc[:5]
```

```
Out[22]:
      year  country_origin_id  country_destination_id  hs92_product_id  export_val  export_val_pct
   0  1995                VNM                     BFA              ALL    67177.77           0.00%
   1  1995                VNM                     CAF              ALL   514674.15           0.00%
   2  1995                VNM                     CIV              ALL    58011.71           0.00%
   3  1995                VNM                     CMR              ALL    97669.00           0.00%
   4  1995                VNM                     COG              ALL    24018.39           0.00%
```

```
In [23]: #We may also provide specific row/column values to access specific values
         df.iloc[0, 1]
```

```
Out[23]: 'VNM'
```

In [24]:

```
#Multiple rows and specific columns
df.iloc[[0, 2], [1, 3]]
```

Out[24]:

	country_origin_id	hs92_product_id
0	VNM	ALL
2	VNM	ALL

In [25]:

```
#We can also splice multiple rows / columns
df.iloc[1:3, 0:3]
```

Out[25]:

	year	country_origin_id	country_destination_id
1	1995	VNM	CAF
2	1995	VNM	CIV

In [34]:

```
#How to iterate over rows
for index, row in df.iterrows():
    print(f'Export from {row["country origin"]} to {row["country destination"]} of {r
```

```
Export from VNM to BFA of 67177.77
Export from VNM to CAF of 514674.15
Export from VNM to CIV of 58011.71
Export from VNM to CMR of 97669.0
Export from VNM to COG of 24018.39
Export from VNM to DZA of 3045918.0
Export from VNM to EGY of 2004172.01
Export from VNM to ETH of 6721108.07
Export from VNM to GIN of 501237.81
Export from VNM to MDG of 58962.92
Export from VNM to MUS of 1735714.92
Export from VNM to NER of 59760.85
Export from VNM to SDN of 1379844.58
Export from VNM to SYC of 10551.0
Export from VNM to TCD of 63364.31
Export from VNM to TGO of 270465.31
Export from VNM to TUN of 1369375.58
Export from VNM to TZA of 148144.85
Export from VNM to UGA of 1103468.68
Export from VNM to ZAF of 1086686.0
```

Conditional Indexing

- Conditional Operators (>, ==, >=) can be used to return rows based on their values
- Bitwise Operators (|, &) can be used to combine conditonal statements

In [23]:

```
df_1995 = df[df["year"] == 1995]

df_1995.head()
```

Out[23]:

	year	country origin	country destination	export_val
0	1995	VNM	BFA	67177.77
1	1995	VNM	CAF	514674.15
2	1995	VNM	CIV	58011.71
3	1995	VNM	CMR	97669.00
4	1995	VNM	COG	24018.39

In [24]:

```
df_2000s = df[df["year"] > 1999]

df_2000s.head()
```

Out[24]:

	year	country origin	country destination	export_val
508	2000	VNM	BEN	923912.58
509	2000	VNM	BFA	339732.75
510	2000	VNM	CAF	33662.13
511	2000	VNM	CIV	342503.71
512	2000	VNM	CMR	1447.32

In [25]:

```
caf_1995 = df[(df["year"]  == 1995) & (df["country destination"]  == "CAF")]
caf_1995.head()
```

Out[25]:

	year	country origin	country destination	export_val
1	1995	VNM	CAF	514674.15

In [26]:

```
df[(df["year"]  == 1995) | (df["year"]  == 1996)].head()
```

Out[26]:

	year	country origin	country destination	export_val
0	1995	VNM	BFA	67177.77
1	1995	VNM	CAF	514674.15
2	1995	VNM	CIV	58011.71
3	1995	VNM	CMR	97669.00
4	1995	VNM	COG	24018.39

In [27]:
```
# find the exports to CAN in 1995

# find the exports to CAN for years greater than 1999
```

Out[27]:

	year	country origin	country destination	export_val
23	1995	VNM	CHN	5.893655e+07
112	1996	VNM	CHN	6.175346e+07
212	1997	VNM	CHN	1.081749e+08
315	1998	VNM	CHN	4.985120e+07
422	1999	VNM	CHN	3.834332e+07

Formatting Data

- To access and format the string values of a DataFrame, we can access methods within the "str" module of the DataFrame
- We may also format float values using options.display.float_format() in Pandas

In [28]:
```
df["country origin"] = df["country origin"].str.replace("VNM", "Vietnam")
```

In [29]:
```
df.head()
```

Out[29]:

	year	country origin	country destination	export_val
0	1995	Vietnam	BFA	67177.77
1	1995	Vietnam	CAF	514674.15
2	1995	Vietnam	CIV	58011.71
3	1995	Vietnam	CMR	97669.00
4	1995	Vietnam	COG	24018.39

In [30]:
```
pd.options.display.float_format = "{:.2f}".format
df.head()
```

Out[30]:

	year	country origin	country destination	export_val
0	1995	Vietnam	BFA	67177.77
1	1995	Vietnam	CAF	514674.15
2	1995	Vietnam	CIV	58011.71
3	1995	Vietnam	CMR	97669.00
4	1995	Vietnam	COG	24018.39

In [31]:
```
df.to_csv("exports.csv")
#to_excel also an option
```