Pandas 1

Let's say I have data as a table (again, using the car parts example):

Part name	Number of units	Price per unit	Total unit price
Wheels	4	500	2000
Doors	4	200	800
Windows	4	100	400
Engine	1	2000	2000
Body	1	5000	5000

There is no way to represent this entire table nicely using the data structures we've seen so far.

Pandas allows us to play with such tables easily.

The first step in using Pandas is to import the module. We will use two statements at the beginning of every pandas code:

In [1]:
```
from pandas import Series, DataFrame
import pandas as pd
```

This does two things:

1. Recall how we used the square-root function from the math module? We used math.sqrt. With the "import pandas as pd", we can call pandas functions using pd.function-name (e.g., pd.read_csv)
2. Two particular data structures will be used very often: Series and DataFrame. Instead of having to say pd.Series, we import these specifically, so we can now just call Series. Ditto for DataFrame.

This lecture will focus on these two structures:

- Series, and
- DataFrame

Series

A Series is a list-like object, but with a few differences.

In [2]:
```
unit_prices = [500, 200, 100, 2000, 5000]
obj = Series(unit_prices) # initializing a Series from a list
obj
```

Out[2]:
```
0     500
1     200
2     100
3    2000
4    5000
dtype: int64
```

A Series consists of an *index* (the first column) and *values* (the second column). Let's enumerate the difference from a vanilla list.

1. The values in a Series are all of the same type (in this case, int). Recall that a list can combine items of arbitrary types.
2. The Series has an *index* (here, 0, 1, 2, 3). We can access individual elements of the Series with this index. Lists also have this implicit index, but with a Series, this index can be arbitrary. Example, the index could be 'car part name'.

In [3]:
```
obj.values  # gives the values in a Series
```

Out[3]:
```
array([ 500,  200,  100, 2000, 5000], dtype=int64)
```

In [4]:
```
obj.index
```

Out[4]:
```
RangeIndex(start=0, stop=5, step=1)
```

Both the *index* and *values* are special kinds of objects that look like lists, but are a bit different. We won't dive too deeply here.

Let's create a more interesting Series.

In [5]:
```
part_names = ['Wheels', 'Doors', 'Windows', 'Engine', 'Body']
unit_price_series = Series(unit_prices, index=part_names)
unit_price_series
```

Out[5]:
```
Wheels      500
Doors       200
Windows     100
Engine     2000
Body       5000
dtype: int64
```

Instead of creating a Series from two lists (the values, and the indices), we can also create it from a dictionary.

In [6]:
```
obj2 = Series({'Wheels':500, 'Doors':200, 'Windows':100, 'Engin
e':2000, 'Body':5000})
obj2
```

Out[6]:
```
Wheels      500
Doors       200
Windows     100
Engine     2000
Body       5000
dtype: int64
```

Accessing elements

Series combine properties of lists and dictionaries:

- The Series values are in a list-like form, and can be accessed just like a list.
- The Series index provides *keys* to access the corresponding values, just like a dictionary.

Thus, a Series allows us to use both list-like and dictionary-like access.

Dictionary-style access to Series

In [7]:
```
# Dictionary-style access
unit_price_series['Windows']
```

Out[7]:
```
100
```

In [8]:
```
unit_price_series[['Body', 'Doors', 'Windows']]
```

Out[8]:
```
Body       5000
Doors       200
Windows     100
dtype: int64
```

In [9]:
```
# we can search within the index, just like for dictionary keys
'Body' in unit_price_series
```

Out[9]:
```
True
```

List-style access

In [10]:
```
unit_price_series
```

Out[10]:
```
Wheels      500
Doors       200
Windows     100
Engine     2000
Body       5000
dtype: int64
```

In [11]:
```
# List-style access
unit_price_series[1]
```

Out[11]:
```
200
```

In [12]:
```
unit_price_series[3:]
```

Out[12]:
```
Engine    2000
Body      5000
dtype: int64
```

Finally, we can combine dictionary-style access with list-like slicing.

In [13]:
```
unit_price_series['Wheels':'Windows']
```

Out[13]:
```
Wheels     500
Doors      200
Windows    100
dtype: int64
```

Difference from dictionary

There are two main differences from a dictionary.

- In a Python dictionary, there is no ordering on the keys.
  - You cannot say, dict[key1:key5].
  - However, list-like slicing on the index is allowed for Series.
  - That is why *the order of the indices matter*.

- In a dictionary, all the keys have to be distinct; you can only have one value per key.
  - However, that is not so for Series. Key不需要be distinct

In [14]:
```
labels_copy = ['Wheels'] * 5  # recall: the '*'-operator repeats list items
labels_copy
```

Out[14]: ['Wheels', 'Wheels', 'Wheels', 'Wheels', 'Wheels']

In [15]:
```
obj3 = Series(unit_prices, index=labels_copy)
obj3
```

Out[15]:
```
Wheels     500
Wheels     200
Wheels     100
Wheels    2000
Wheels    5000
dtype: int64
```

The index now has repeated items, so there are multiple values for the same index.

In [16]:
```
obj3['Wheels']  # Returns a Series; not just one value like for dictionaries!
```

Out[16]:
```
Wheels     500
Wheels     200
Wheels     100
Wheels    2000
Wheels    5000
dtype: int64
```

Filtering a Series

One of the important functions that can be performed on a Series is filtering. Suppose we want all units priced less than some amount, say, 400. How do we do it?

In [17]:
```
mask = (unit_price_series < 400)
mask
```

Out[17]:
```
Wheels     False
Doors       True
Windows     True
Engine     False
Body       False
dtype: bool
```

This gives a Boolean Series where we have the same index, but the values are True (if value < 400) or False (value >= 400). This is often called a boolean **mask**.

The mask can be used to select out items from a Series.

In [18]:
```
unit_price_series[mask]
```

Out[18]:
```
Doors      200
Windows    100
dtype: int64
```

In [19]:
```
unit_price_series  # We already have the series of unit prices
```

Out[19]:
```
Wheels      500
Doors       200
Windows     100
Engine     2000
Body       5000
dtype: int64
```

In [20]:
```
# Let's create another Series of number-of-units for each car pa
rt
num_units_series = Series({'Wheels':4, 'Doors':4, 'Windows':4,
'Engine':1, 'Body':1})
num_units_series
```

Out[20]:
```
Wheels     4
Doors      4
Windows    4
Engine     1
Body       1
dtype: int64
```

Example: Find the unit prices of all car parts of which we only need 1 unit.

In [21]:
```
mask = (num_units_series == 1)  # Recall: == is equality conditi
on
mask
```

Out[21]:
```
Wheels     False
Doors      False
Windows    False
Engine      True
Body        True
dtype: bool
```

In [22]:
```
unit_price_series[mask]
```

Out[22]:
```
Engine    2000
Body      5000
dtype: int64
```

NOTE: The *order* of parts in unit_price_series and num_units_series are different! However, this is where the index is useful; pandas doesn't use the ordering, it uses the index to figure out how to apply the mask to unit_price_series.

Operations on Series

Obvious things work.

In [23]:
```
# Increase unit prices by 3% for inflation
unit_price_series * 1.03
```

Out[23]:
```
Wheels      515.0
Doors       206.0
Windows     103.0
Engine     2060.0
Body       5150.0
dtype: float64
```

If you want to apply some function to the Series, use the map() method of Series.

In [24]:
```
# Find square-roots of all unit prices
import math
unit_price_series.map(math.sqrt)
```

Out[24]:
```
Wheels     22.360680
Doors      14.142136
Windows    10.000000
Engine     44.721360
Body       70.710678
dtype: float64
```

We can also get aggregate statistics of a Series.

In [25]:
```
print('Mean =', unit_price_series.mean())  # Average unit price
print('Variance =', unit_price_series.var()) # Variance of unit
 prices
print('Max =', unit_price_series.max(), ' for car part =', unit_
price_series.idxmax())
```

```
Mean = 1560.0
Variance = 4283000.0
Max = 5000  for car part = Body
```

Example: Find all car parts whose unit price is at least 10% of the priciest part.

In [26]:
```
unit_price_series[unit_price_series >= 0.1 * unit_price_series.m
ax()]
```

Out[26]:
```
Wheels     500
Engine    2000
Body      5000
dtype: int64
```

We can also combine two Series in obvious ways.

In [27]:

```
unit_prices_second_car = Series({'Wheels':600, 'Doors': 400, 'Wi
ndows':100, 'Engine':5000, 'Body':10000})
print('Second car:')
print(unit_prices_second_car)
print
print('First car:')
print(unit_price_series)
```

```
Second car:
Wheels       600
Doors        400
Windows      100
Engine      5000
Body       10000
dtype: int64
First car:
Wheels      500
Doors       200
Windows     100
Engine     2000
Body       5000
dtype: int64
```

Example: Find the average unit price for each car part.

In [28]:

```
# Average unit price for each car part
(unit_price_series + unit_prices_second_car) / 2
```

Out[28]:

```
Wheels      550.0
Doors       300.0
Windows     100.0
Engine     3500.0
Body       7500.0
dtype: float64
```

Missing values

Real-world data is often full of missing or incorrect values. One of the advantages of pandas is that it makes dealing with missing values relatively painless.

In [29]:
```
# Let's ask for a missing car part
# Alternately: unit_price_series.reindex(['Engine', 'Transmissio
n', 'Body'])

missing_series = unit_price_series[['Engine', 'Transmission', 'B
ody']]
missing_series
```

```
C:\Users\deepay\Miniconda\lib\site-packages\pandas\core\serie
s.py:1155: FutureWarning:
Passing list-likes to .loc or [] with any missing label will
raise
KeyError in the future, you can use .reindex() as an alternat
ive.

See the documentation here:
https://pandas.pydata.org/pandas-docs/stable/user_guide/index
ing.html#deprecate-loc-reindex-listlike
  return self.loc[key]
```

Out[29]:
```
Engine          2000.0
Transmission       NaN
Body            5000.0
dtype: float64
```

The 'NaN' stands for "Not A Number", and this is how pandas denotes missing values.

Another common situation is when we process two series with mismatched indices.

There are three types of operations we can do with missing values:

- find the items with missing values,
- drop them from our Series, or
- fill the missing values with a value of our choice.

In [30]:
```
# Find missing elements
mask = missing_series.isnull()
mask
```

Out[30]:
```
Engine          False
Transmission     True
Body            False
dtype: bool
```

In [31]:
```
missing_series[mask]
```

Out[31]:
```
Transmission   NaN
dtype: float64
```

In [32]:
```
# Drop missing elements
missing_series.dropna()
```

Out[32]:
```
Engine    2000.0
Body      5000.0
dtype: float64
```

In [33]:
```
# Fill missing values
missing_series.fillna(-1)
```

Out[33]:
```
Engine          2000.0
Transmission      -1.0
Body            5000.0
dtype: float64
```

Example: Replace missing values with the mean.

How do we do it?

In [34]:
```
missing_series.fillna(missing_series.mean())
```

Out[34]:
```
Engine          2000.0
Transmission    3500.0
Body            5000.0
dtype: float64
```

Summary

A Series allows us to attach an index to a list. This has several benefits:

- The index allows dictionary-like access to the list items, in addition to the usual list-like access.
- Pandas lets us combine two Series by "matching up" their indices.
- Finally, there are lots of helper functions to modify values, deal with missing values, compute statistics and such.

However, it still leaves much to be desired.

- We still cannot represent the entire car parts table using just a series
  - We need multiple series

A DataFrame is just that.

DataFrame

Roughly, DataFrame = combination of Series sharing the same index.

For instance, our Car Parts table can be thought of as three series (unit price, number of units, and total unit price) on the same index (car part name).

In [35]:

```
data = {'unit price': [500, 200, 100, 2000, 5000], 'number of units':[4, 4, 4, 1, 1]}
print('data =', data)
print('part_names =', part_names)

car_table = DataFrame(data, index=part_names)
car_table
```

```
data = {'unit price': [500, 200, 100, 2000, 5000], 'number of units': [4, 4, 4, 1, 1]}
part_names = ['Wheels', 'Doors', 'Windows', 'Engine', 'Body']
```

Out[35]:

	unit price	number of units
Wheels	500	4
Doors	200	4
Windows	100	4
Engine	2000	1
Body	5000	1

Thus, each column of the DataFrame is a Series, and all the series share the same index.

Accessing elements

We can easily get the individual series that form this DataFrame.

In [36]:
```
car_table['number of units']
```

Out[36]:
```
Wheels     4
Doors      4
Windows    4
Engine     1
Body       1
Name: number of units, dtype: int64
```

We can also add new columns.

In [37]:
```
car_table['Total unit price'] = car_table['number of units'] * car_table['unit price']
car_table
```

Out[37]:

	unit price	number of units	Total unit price
Wheels	500	4	2000
Doors	200	4	800
Windows	100	4	400
Engine	2000	1	2000
Body	5000	1	5000

Accessing *rows* is a little different.

In [38]:
```
# Get all information about Windows
car_table.loc['Windows']
```

Out[38]:
```
unit price          100
number of units       4
Total unit price    400
Name: Windows, dtype: int64
```

Notice that this also gives us a Series; it is just that row written out as a Series.

What happens if you want two rows?

In [39]:
```
# Get two rows
car_table.loc[['Engine', 'Body']]
```

Out[39]:

	unit price	number of units	Total unit price
Engine	2000	1	2000
Body	5000	1	5000

In [40]:
```
# Get total unit price of just Wheels and Doors
car_table.loc[['Wheels', 'Doors'], ['Total unit price']]
```
row column

Out[40]:

	Total unit price
Wheels	2000
Doors	800

We can also use list-like indexing for the rows

In [41]:
```
# First two rows and columns
car_table.iloc[:2, :2]
```

Out[41]:

	unit price	number of units
Wheels	500	4
Doors	200	4

We can again use masks.

In [42]:
```
# Let's add a second car.
car_table['car-2 unit price'] = [300, 400, 500, 3000, 4000]
car_table['car-2 Total unit price'] = car_table['car-2 unit pric
e'] * car_table['number of units']
car_table
```

Out[42]:

	unit price	number of units	Total unit price	car-2 unit price	car-2 Total unit price
Wheels	500	4	2000	300	1200
Doors	200	4	800	400	1600
Windows	100	4	400	500	2000
Engine	2000	1	2000	3000	3000
Body	5000	1	5000	4000	4000

Example: Find units for which car-2 is pricier than the first car.

In [43]:
```
mask = (car_table['car-2 unit price'] > car_table['unit price'])
car_table[mask]
```

Out[43]:

	unit price	number of units	Total unit price	car-2 unit price	car-2 Total unit price
Doors	200	4	800	400	1600
Windows	100	4	400	500	2000
Engine	2000	1	2000	3000	3000

In [44]:
```
# To flip the index and columns
car_table.T    # T is short-form for "transpose", which flips row
s and columns of a matrix
```

Out[44]:

	Wheels	Doors	Windows	Engine	Body
unit price	500	200	100	2000	5000
number of units	4	4	4	1	1
Total unit price	2000	800	400	2000	5000
car-2 unit price	300	400	500	3000	4000
car-2 Total unit price	1200	1600	2000	3000	4000

Reading from CSV files

Most often, you will have data in a tabular form somewhere and you'll read from it. Pandas allows us to easily build DataFrames from CSV files.

In [45]:
```
!cat Intro_4_data/CarParts.csv
```

```
Part name,Number of units,Price per unit,Total price
Wheels,4,500,2000
Doors,4,200,800
Windows,4,100,400
Engine,1,2000,2000
Body,1,5000,5000
```

In [46]:
```
df = pd.read_csv('Intro_4_Data/CarParts.csv')
df
```

Out[46]:

	Part name	Number of units	Price per unit	Total price
0	Wheels	4	500	2000
1	Doors	4	200	800
2	Windows	4	100	400
3	Engine	1	2000	2000
4	Body	1	5000	5000

This creates a data frame as desired, but the index is the *default* index.

In [47]:
```
print(df.index)
```

```
RangeIndex(start=0, stop=5, step=1)
```

We want to set the 'Part name' to be the index. We do this via set_index().

In [48]:
```
df.set_index('Part name', inplace=True)
df
```

Out[48]:

Part name	Number of units	Price per unit	Total price
Wheels	4	500	2000
Doors	4	200	800
Windows	4	100	400
Engine	1	2000	2000
Body	1	5000	5000

In [49]:
```
print(df.index)
```

```
Index(['Wheels', 'Doors', 'Windows', 'Engine', 'Body'], dtype
='object', name='Part name')
```

In [50]:
```
print(df.index.values)
```

```
['Wheels' 'Doors' 'Windows' 'Engine' 'Body']
```

Operations on a DataFrame

It is easy to select a Series, and apply a formula to that Series.

In [51]:
```
# mean unit price of car parts
df['Price per unit'].mean()
```

Out[51]:
```
1560.0
```

We can also apply the same function to all columns.

Example: Find the range of values (max - min) for each of the columns.

In [52]:

```
def get_column_range(x):
    # x here is a Series
    return x.max() - x.min()

# "Apply" this range function to each column of the DataFrame
df.apply(get_column_range)
```

Out[52]:

```
Number of units       3
Price per unit     4900
Total price        4600
dtype: int64
```

Another common operation is sorting the entire DataFrame. There are two methods for this:

- *sort_index()*, and
- *sort_values()*

In [53]:

```
# Sort the DataFrame by its index.
df.sort_index()
```

Out[53]:

Part name	Number of units	Price per unit	Total price
Body	1	5000	5000
Doors	4	200	800
Engine	1	2000	2000
Wheels	4	500	2000
Windows	4	100	400

In [54]:
```
# Sort the DataFrame by price per unit
df.sort_values(by='Price per unit')
```

Out[54]:

Part name	Number of units	Price per unit	Total price
Windows	4	100	400
Doors	4	200	800
Wheels	4	500	2000
Engine	1	2000	2000
Body	1	5000	5000

Summary

A DataFrame helps organize several Series together. Each Series becomes a column of a table, and they are all linked via the same index.

- Read in a table using pd.read_csv (or pd.read_table(); do help(pd.read_table)!)
- Access a column by df['Number of units']
- Access a row by df.loc['Windows'] or df.iloc[0]
- Change the index using df.set_index('Price per unit', inplace=True)
- Apply arbitrary functions using apply()
- In general, use Series methods after selecting out a column of the DataFrame.