main

Pandas Review¶

Introduction¶

Pandas is a Python library that plays a pivotal role in data science. It is used both for data wrangling and for calculations, and merges well with machine learning libraries, too. It has plenty of applications, covering a lot of the lost ground that Python had versus R in the past. You need to install it with the following command in the terminal.

```
python3 -m pip install pandas
```

Pandas uses *numpy* under the hood. *numpy* is a numerical library that enhances Python's computational capabilities. Pandas is well thought so that we do not need to explicitly invoke *numpy* often, but it will nevertheless appear every now and then.

We will mentioned *arrays* sometimes, and we will be referring to numpy's *ndarray* object. You may think of it loosely as a homogeneus list of numbers.

In [ ]:

```
from pprint import pprint
import pandas as pd
```

Series¶

Series and *DataFrame* are the two workhorses of pandas. *Series* is a one-dimensional object containing a sequence of values and an associated array of data labels called *index*.

Let's define our first *series* (the *pprint* is not necessary).

In [ ]:

```
from pprint import pprint
import pandas as pd
obj = pd.Series([1, 10, 5, 2])
pprint(obj)
```

```
0     1
```

```
1    10
2     5
3     2
dtype: int64
```

We can access the *array* and *index* attributes easily.

In [ ]:

```
obj.array
obj.index
```

Out[ ]:

```
RangeIndex(start=0, stop=4, step=1)
```

Sometimes we want the *index* to consist of labels instead of integers. The labels can be used then to access single values or sets of values.

In [ ]:

```
obj_ = pd.Series([1, 10, 5, 2], index=['a', 'b', 'c', 'd'])
obj_
obj_['a']
obj_[['a', 'c']]
```

Out[ ]:

```
a    1
c    5
dtype: int64
```

Note that we need to put the indexes in a list. We can filter a *Series* and apply numerical operations to it.

In [ ]:

```
obj_[obj_ > 3]
obj_ * 3
import numpy as np
np.log(obj_)
```

Out[ ]:

```
a    0.000000
b    2.302585
c    1.609438
d    0.693147
dtype: float64
```

You may think of a *Series* as an ordered dictionary, meaning that we can apply a similar syntax to the one used in dictionaries.

In [ ]:

```
'a' in obj_
'other_index' in obj_
```

Out[ ]:

```
False
```

Indeed, we can easily create a Series from a dictionary.

In [ ]:

```
savings = {'Ann': 10, 'Bob': 20, 'Charlie':15, 'Diane': 5}
obj_s = pd.Series(savings)
```

We can enforce a particular order for the indexes, but if we push one that does not have a value, a `NA` will appear. `NA` is a missing value and can be detected with the Python functions `isna()` and `notna()`.

In [ ]:

```
people = ('Ann', 'Bob', 'Charlie', 'Diane', 'Eddie')
obj_n = pd.Series(savings, index=people)
obj_n
```

Out[ ]:

```
Ann        10.0
Bob        20.0
Charlie    15.0
Diane       5.0
Eddie       NaN
dtype: float64
```

In [ ]:

```
pd.isna(obj_n)
pd.notna(obj_n)
obj_n.isna()
```

Out[ ]:

```
Ann        False
Bob        False
Charlie    False
Diane      False
Eddie       True
dtype: bool
```

Indexes are useful because they align the Series automatically when operating with them.

In [ ]:

```
lump = {'Ann':2, 'Bob': 3, 'Charlie':0, 'Eddie':2}
obj_l = pd.Series(lump)
obj_n + obj_l
```

Out[ ]:

```
Ann        12.0
Bob        23.0
Charlie    15.0
Diane       NaN
Eddie       NaN
dtype: float64
```

Note that the `NaN` is "contagious", affecting the operations where it is involved.

Both the Series object and its index have a name that can be modified. The index can also be altered by assignment.

In [ ]:

```
obj_n.name = 'savings'
obj_n.index.name = 'people'
obj_n
```

Out[ ]:

```
people
Ann        10.0
Bob        20.0
Charlie    15.0
Diane       5.0
Eddie       NaN
Name: savings, dtype: float64
```

Dataframes¶

A DataFrame is a rectangular table of data that contains a ordered, named collection of columns. One common way to build a dataframe is with a dictionary of lists or arrays.

In [ ]:

```
companies = {'name': ['Tesla', 'Berkshire', 'Nvidia', 'Tencent'],
             'country': ['US', 'US', 'US', 'China'],
             'market_cap': [350, 400, 600, 550],
}
df = pd.DataFrame(companies)
print(df)
```

```
        name country  market_cap
0      Tesla      US         350
```

```
1  Berkshire      US         400
2     Nvidia      US         600
3    Tencent   China         550
```

The order of the columns can be specified.

In [ ]:

```
df = pd.DataFrame(companies, columns=['country', 'name', 'market_cap'])
print(df)
```

```
  country       name  market_cap
0      US      Tesla         350
1      US  Berkshire         400
2      US     Nvidia         600
3   China    Tencent         550
```

Columns that are not contained in the constructor will be full of `NaN`s.

In [ ]:

```
df = pd.DataFrame(companies, columns=['country', 'name', 'market_cap', 'revenues'])
print(df)
```

```
  country       name  market_cap revenues
0      US      Tesla         350      NaN
1      US  Berkshire         400      NaN
2      US     Nvidia         600      NaN
3   China    Tencent         550      NaN
```

Columns can be accessed either as keys or as attributes.

In [ ]:

```
df.name
df['name']
```

Out[ ]:

```
0        Tesla
1    Berkshire
2       Nvidia
3      Tencent
Name: name, dtype: object
```

Rows can be accessed with the `loc` attribute and their index.

In [ ]:

```
df.loc[1]
```

Out[ ]:

```
country              US
name          Berkshire
```

```
market_cap          400
revenues            NaN
Name: 1, dtype: object
```

Columns can be modified by assignment of a scalar value or of an array or list of values.

In [ ]:

```
df.earnings = 100
df.earnings = [400, 300, 200, 100]
```

In [ ]:

```
df
```

Out[ ]:

	country	name	market_cap	revenues
0	US	Tesla	350	NaN
1	US	Berkshire	400	NaN
2	US	Nvidia	600	NaN
3	China	Tencent	550	NaN

In this assignment, either the value's length is the same as that of the DataFrame, or it is embedded in a Series with matching indexes. Columns can be deleted with the `del` command.

In [ ]:

```
earnings = pd.Series([20, 50], index=['two', 'four'])
df['earnings'] = earnings
del df['country']
df
```

Out[ ]:

	name	market_cap	revenues	earnings
0	Tesla	350	NaN	NaN
1	Berkshire	400	NaN	NaN
2	Nvidia	600	NaN	NaN
3	Tencent	550	NaN	NaN

Note that columns returned from indexing are a *view* of the dataframe, and therefore in-place modifications will be reflected in the underlying DataFrame. To prevent this, we can use the *copy* method.

Indexes are immutable, which makes them safer to share between DataFrames.

In [ ]:

```
obj = pd.Series(range(3), index=['first', 'second', 'third'])
index_ = obj.index
index_
# index_[0] = 'primero' # this will fail
obj2 = pd.Series(range(10, 13), index=index_)
obj2
```

Out[ ]:

```
first     10
second    11
third     12
dtype: int64
```

Basic operations¶

Introduction¶ We will now walk through the most usual Python functionalities.

Reindexing¶ `reindex` rearranges the data according to a new index.

In [ ]:

```
obj = pd.Series([10, 20, 25, 40, 30], index=['Annie', 'Bob', 'Charlie', 'Doug', 'Elaine'])
obj.reindex(['Fred', 'Elaine', 'Doug', 'Charlie', 'Bob', 'Annie'])
df = pd.DataFrame({'manufacturer': ['Tesla', 'Ford', 'Toyota']}, index=['Model 3', 'Mondeo',
df.reindex(['Corolla', 'Model 3', 'Mondeo', 'Golf'])
```

Out[ ]:

	manufacturer
Corolla	Toyota
Model 3	Tesla
Mondeo	Ford
Golf	NaN

Columns of a DataFrame can also be reindexed with the `reindex` method or, alternatively, with `loc`, which we will cover shortly.

In [ ]:

```
df.reindex(columns=['manufacturer', 'model'])
```

Out[ ]:

	manufacturer	model
Model 3	Tesla	NaN
Mondeo	Ford	NaN
Corolla	Toyota	NaN

In [ ]:

```
df = pd.DataFrame({'model':['Model 3', 'Mondeo', 'Corolla'],'manufacturer': ['Tesla', 'Ford'
df
```

Out[ ]:

	model	manufacturer
0	Model 3	Tesla
1	Mondeo	Ford
2	Corolla	Toyota

In [ ]:

```
df.loc[:, ['manufacturer', 'model']]
```

Out[ ]:

	manufacturer	model
0	Tesla	Model 3
1	Ford	Mondeo
2	Toyota	Corolla

Dropping entries¶ Dropping entries (i.e. rows) can be done with the `drop` method.

In [ ]:

```
df = pd.DataFrame({'model': ['Tesla', 'Ford', 'Toyota']}, index=['Model 3', 'Mondeo', 'Corol
df.drop('Mondeo')
```

Out[ ]:

	model
Model 3	Tesla
Corolla	Toyota

Dropping columns requires to pass the argument `axis`.

In [ ]:

```
df.drop('model', axis='columns')
```

Out[ ]:

Model 3
Mondeo
Corolla

Note that the changes are not permanent unless we set the argument `inplace` equal to `True`.

In [ ]:

```
df
```

Out[ ]:

	model
Model 3	Tesla
Mondeo	Ford
Corolla	Toyota

In [ ]:

```
df.drop('model', axis='columns', inplace=True)
```

In [ ]:

```
df
```

Out[ ]:

Model 3
Mondeo
Corolla

Indexing, selection, and filtering¶

Indexing Series¶ Indexing bears similarities with dictionaries. Let's take a look at some examples.

In [ ]:

```
obj = pd.Series([5, 10, 0], index=['bonds', 'stocks', 'cash'])
obj['bonds']
```

```
obj[0]
```

Out[ ]:

```
5
```

Ranges of entries and filters can also be applied.

In [ ]:

```
obj[0:2]
obj[['cash', 'bonds']]
obj[[1, 0]]
obj[obj < 5]
```

Out[ ]:

```
cash    0
dtype: int64
```

Note how the index can be accessed through an integer regardless of whether the index is an integer. This is error-prone and, to avoid it, the recommended way to index is with the `loc` and `iloc` operators, for label and integer indexes respectively.

In [ ]:

```
obj.loc[['cash', 'bonds']]
obj.iloc[[2, 0]]
```

Out[ ]:

```
cash     0
bonds    5
dtype: int64
```

Slicing with labels is similar to normal Python slicing, but includes the endpoints.

In [ ]:

```
obj.loc['stocks':'cash']
obj.loc['stocks':'cash'] = 7
obj
```

Out[ ]:

```
bonds     5
stocks    7
cash      7
dtype: int64
```

Indexing DataFrames¶ Like in Series, we can select a subset of the rows and columns of a DataFrame with `loc` and `iloc`. These are a few examples.

In [ ]:

```
df = pd.DataFrame({'manufacturer': ['Tesla', 'Ford', 'Toyota']}, index=['Model 3', 'Mondeo',
df.loc[['Model 3', 'Corolla'], 'manufacturer']
```

Out[ ]:

```
Model 3     Tesla
Corolla    Toyota
Name: manufacturer, dtype: object
```

In [ ]:

```
df.iloc[[0, 2], 0]
```

Out[ ]:

```
Model 3     Tesla
Corolla    Toyota
Name: manufacturer, dtype: object
```