1. Intoduction to Pandas

1.1 Importing pandas package

Numpy는 대수학 연산을 수행하는데 유용한 연산을 제공하고 빠른 수행시간을 가지고 있다. 하지만 Numpy의 ndarray 객체는 numerical data만 저장할 수 있으며 동일한 데이터 타입만 하나의 ndarray에 저장할 수 있다. Pandas package는 서로 다른 데이터 타입에 대해서 데이터프레임이라는 객체를 생성할 수 있다.

```
import pandas as pd
```

1.2 Read csv file using pandas

Pandas는 CSV파일을 읽기 위한 pandas.read_csv() 함수를 제공한다. 또한 내부의 다양한 parameter는 CSV파일을 다양하게 읽을 수 있게 한다.

- sep : specify separator
- encoding : specify different encoding

```
import pandas as pd
cars = pd.read_csv('cars.csv')
```

1.3 DataFrame

Pandas 패키지는 Numpy 패키지를 작성되어 ndarray를 기반으로한 DataFrame 객체가 존재한다. DataFrame은 2-dimensional numpy array와 비슷하며 많은 메소드와 속성을 보유하고 있다.

- shape : attribute shows a tuple with the number of rows and columns of the dataframe
- df.head() : returns the first five rows
- df.tail() : returns the last five rows

```
cars_shape = cars.shape
first_six_rows = cars.head(6)
last_four_rows = cars.tail(4)
```

1.4 Indexing DataFrame

iloc

DataFrame은 DataFrame.iloc 속성을 가지고 있다. iloc 속성은 행과 열의 인덱스를 통해 접근할 수 있게 해주며 integer location을 상징한다. ndarray에서 사용했던 start:end:step의 인덱스 방식 또한 사용할 수 있다.

iat

iat 속성은 iloc와는 다르게 범위를 지정할 수 없다.

```
cars_odd = cars.iloc[1::2, :3]
fifth_odd_car_name = cars_odd.iat[4,0]
last_four = cars_odd.tail(4)
```

loc

Pandas는 인덱스 번호로만 데이터를 검색하는 것 뿐만아니라 인덱스, 컬럼의 이름을 통해 검색할 수 있다. DataFrame.loc 속성은 행과 열의 이름을 통해 데이터를 검색할 수 있게 해준다. 행중 하나를 인덱스로 설정하는 방법은 다음과 같다.

- pd.read_csv(..., index_col = n)
- DataFrame.set_index([col], inplace = True)

```
cars.set_index('Name', inplace = True)
weight_torino = cars.loc['Ford Torino', 'Weight']
```

설정한 인덱스를 데이터프레임으로 행으로 다시 전환하려면 다음 메소드를 사용한다.

```
honda_civic_hp = cars.loc['Honda Civic', 'Horsepower']
print(honda_civic_hp)
cars.reset_index(inplace = True)
```

ndarray에서 리스트를 통해 특정 행을 인덱싱한 것과 동일하게, 복수의 컬럼을 인덱싱하기 위해서 리스트 내부에 행과 열 이름을 담아 인덱싱을 할 수 있다.

```
numeric_data = cars.loc[:, 'MPG':'Acceleration']
weights = cars['Weight']
name_origin_0_and_3 = cars.loc[[0,3], ['Name', 'Origin']]
```

1.5 DataFrame Index

데이터프레임의 인덱스 객체는 default로 0부터 시작하는 정수로 저장되어 있다. 만약 새로운 인덱스를 지정하고 싶다면 pd.Index() 메소드를 통해 새로운 인덱스 객체를 정의할 수 있다.

```
num_rows = cars.shape[0]
one_index = pd.Index(range(1,num_rows+1))
cars.set_index(one_index, inplace = True)

cars_100 = cars.loc[100]
cars_2_to_10 = cars.loc[2:10]
```

2. Calculating With Pandas

2.1 Series

Pandas의 Series 객체는 1-dimensional data를 저장하는 데이터 구조이다. Series 객체는 1-dimensional ndarray와 동일한 구조를 가지고 있으며 데이터프레임으로부터 파생될때 데이터프레임의 인덱스와 행을 가지고 나온다.

- Series.values : return values of Series
- Series.max() : return max value of Series
- Seires.min() : return min value of Series

2.2 Useful methods

- Series.mean() : return the average value of a Series
- Series.value_counts() : count of many of each value a series contains
- Series.to_dict() : covert series into a dictionary

```
origin_counts = cars['Origin'].value_counts()
origin_counts_dict = origin_counts.to_dict()
print(origin_counts_dict)
```

2.3 Boolean Indexing

Numpy에서 조건을 만족하는 행과 열을 찾기 위해 boolean mask를 사용하였다. Pandas도 이와 마찬가지로 True/False value를 반환하는 boolean mask를 사용하여 조건에 맞는 데이터프레임을 조회할 수 있다. loc 속성을 사용하면 조건을 만족하는 특정 열만을 추출할 수 있다.

```
european_cars = cars[cars['Origin'] == 'Europe']
non_us_cars = cars[~(cars['Origin'] == 'US')]
low_mpg_horsepower = cars[(cars['MPG'] > 0) & (cars['MPG'] < 10) & (cars['Horsepower'] > 150)]
light_or_fast = cars[(cars['Weight'] <= 2000) | (cars['Acceleration'] >= 30)]

name_and_origin = cars.loc[(cars['MPG'] > 0) & (cars['MPG'] < 12) & (cars['Horsepower'] >= 200), ['Name', 'Origin']]
```

2.4 Make new columns

기존에 존재하는 컬럼을 사용해서 연산을 진행한 값을 새로운 컬럼에 할당하는 것은 데이터프레임의 새로운 변수에 그대로 할당하면 된다.

```
cars['PW_ratio'] = cars['Horsepower'] / cars['Weight']
max_pw_ratio = PW_ratio.max()
```

DataFrame은 복수의 Series가 열을 기준으로 결합한 것과 같다. 따라서 미리 생성된 Series를 DataFrame에 할당할 수도 있다.

```
mpg_l100_constant = 235.214583
mpg_non_zero = cars.loc[cars['MPG'] > 0, 'MPG']
L100 = mpg_l100_constant / mpg_non_zero
cars['L/100'] = L100
```

3. Optimizing DataFrame Memory Footprint

3.1 Estimate the amount of memory

DataFrame.info() 메소드는 데이터프레임의 Non-Null Count, dtype, memory usage에 대한 정보를 제공한다. memory_usage = 'deep' 키워드는 좀 더 정확한 메모리 사용을 확인할 수 있다.

```
import pandas as pd
moma = pd.read_csv('moma.csv')
moma.info()
```

BlockManager

Pandas의 BlockManager Class는 데이터를 타입별로 최적화하여 분리하여 저장한다. BlockManager Class는 API처럼 행동하며 우리가 값을 처리할 때 데이터프레임을 BlockManager와 상호작용하여 처리한다. 각각의 블록은 Numpy ndarray로 저장되어 있어 속도가 빠르다.

```
print(moma._data)

# Blockmanager stored in _data
BlockManager
Items: Index(['ExhibitionID', 'ExhibitionNumber', 'ExhibitionTitle',
       'ExhibitionCitationDate', 'ExhibitionBeginDate', 'ExhibitionEndDate',
       'ExhibitionSortOrder', 'ExhibitionURL', 'ExhibitionRole',
       'ConstituentID', 'ConstituentType', 'DisplayName', 'AlphaSort',
       'FirstName', 'MiddleName', 'LastName', 'Suffix', 'Institution',
       'Nationality', 'ConstituentBeginDate', 'ConstituentEndDate',
       'ArtistBio', 'Gender', 'VIAFID', 'WikidataID', 'ULANID',
       'ConstituentURL'],
      dtype='object')
Axis 1: RangeIndex(start=0, stop=34558, step=1)
FloatBlock: [0, 6, 9, 19, 20, 23, 25], 7 x 34558, dtype: float64
ObjectBlock: [1, 2, 3, 4, 5, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 21, 22, 24, 26], 20 x 34558, dtype: object
```

Float Columns

float64 데이터 타입은 64bits(8bytes)를 사용하는 소수값으로 데이터 프레임 내부에 34,588 행이 있기 때문에 float64 column은 데이터프레임 내부에서 276,464 bytes를 사용하게 된다.

Object Columns

object 데이터 타입은 문자열 데이터를 저장하는 데이터 타입이다. Python은 high-level, interpered 언어이기 때문에 저장 메모리를 더 소모하고 속도도 느려지게 된다. Python list내부의 데이터는 실제 값을 저장하는게 아닌 값이 저장되어 있는 주소를 저장하기 때문이다.

object type은 즉 연결된 값을 검사하는 것이 아닌 저장되어 있는 주소 데이터(8 bytes)만 저장해서 데이터프레임 메모리를 추정하게 된다.

3.2 Calculate the amount of memory

DataFrame.size 속성은 데이터프레임 내부에 저장되어 있는 값의 개수를 반환한다. 문자열 데이터의 크기까지 고려한 데이터프레임의 메모리 크기는 info(memory_usage = "deep")를 통해 확인할 수 있다. 즉, 해당 키워드를 사용하지 않은 메모리의 크기가 7.1MB이고, 키워드를 사용한 메모리의 크기가 45.6MB라면, 문자열 데이터의 크기는 38.5MB임을 추정할 수 있다.

정확한 데이터 크기를 측정하는 방법은 memory_usage() 메소드를 통해 확인할 수 있다.

- DataFrame.select_dtypes(include = ['object']) : get a dataframe containing only the columns with the datatype
- DataFrame.memory_usage(deep = True) : return the amount of memory each column consumes

```
obj_cols = moma.select_dtypes(include=['object'])
obj_cols_mem = obj_cols.memory_usage(deep = True)
obj_cols_sum = obj_cols_mem.sum() / (2**20)
```

3.3 Optimizing numerical data

숫자형 자료는 숫자 데이터가 가지고 있는 bit의 길이를 고정함으로써 최적화할 수 있다. numpy.iinfo() 메소드는 각각의 자료형이 가지고 있는 최소와 최대 정수를 반환한다.

```
def change_to_int(df, col_name):
    # Get the minimum and maximum values
    col_max = df[col_name].max()
    col_min = df[col_name].min()
    # Find the datatype
    for dtype_name in ['int8', 'int16', 'int32', 'int64']:
        # Check if this datatype can hold all values
        if col_max <  np.iinfo(dtype_name).max and col_min > np.iinfo(dtype_name).min:
            df[col_name] = df[col_name].astype(dtype_name)
            break

# Add you code below
float_moma = moma.select_dtypes(include = ['float64'])
print(float_moma.isnull().sum())

# Column ExhibitionSortOrder don't have NaN values
change_to_int(moma, 'ExhibitionSortOrder')
print(moma['ExhibitionSortOrder'].dtype)
```

Check nan

데이터프레임 내부에 존재하는 결측값은 np.NaN으로 저장되어 있다. DataFrame.isnull() 메소드는 결측값의 여부에 따른 True/False 데이터프레임을 반환한다. 이를 활용하여 DataFrame.isnull().sum()을 활용해 각 컬럼에 따른 결측값의 개수를 확인할 수 있다.

3.4 Using to_numeric() method

pandas.to_numeric() 메소드는 일반적인 자료형을 바꿀 뿐만 아니라 downcast 키워드에 입력한 자료형의 최소 자료형으로 데이터를 자동 변경한다. 즉, 앞서 작성했던 change_to_int() 함수의 built-in Version이다.

```
float_cols = moma.select_dtypes(include=['float']).columns

# Write you code below

for col in float_cols :
    moma[col] = pd.to_numeric(moma[col], downcast = 'float')
print(moma.dtypes)
```

3.5 Optimizing in datetime

Pandas에는 datetime64 라는 날짜형 데이터를 표현하는 데이터 타입이 존재한다. pd.to_datetime()은 컬럼의 데이터를 datetime 자료형으로 변경한다.

```
moma['ExhibitionBeginDate'] = pd.to_datetime(moma['ExhibitionBeginDate'])
moma['ExhibitionEndDate'] = pd.to_datetime(moma['ExhibitionEndDate'])
moma[['ExhibitionBeginDate', 'ExhibitionEndDate']].memory_usage()
```

3.6 Optimizing in category

SQLite에서 활용했던 열거형(enumerated) 타입처럼, Pandas 내부에는 category 데이터 타입이 존재한다. category type은 각 데이터를 정수에 맵핑시켜놓은 자료형으로, 수가 적은 데이터의 집합을 나타내는데 매우 효율적이다. 하지만 고유값이 전체 데이터의 50%를 넘어가는 경우에는 category type을 저장하기 위한 저장소 문제가 발생하기 때문에 비효율적이게 된다.

```
print(moma['ConstituentType'].memory_usage(deep=True))

moma['ConstituentType'] = moma['ConstituentType'].astype('category')
print(moma['ConstituentType'].memory_usage(deep=True))

# return the integer values the category type uses to represent each value
print(moma['ConstituentType'].cat.codes)


cat_col = [fea for fea in moma.select_dtypes(include = ['object']) if moma[fea].nunique() < (len(moma)/2)]

for col in cat_col :
    moma[col] = moma[col].astype('category')
print(moma.info(memory_usage = 'deep'))
```

3.7 Optimizing when read data

만약 데이터프레임을 저장공간의 문제 때문에 열지못할 경우 데이터프레임을 불러오면서 데이터 타입을 지정해야할 필요가 있다. pandas.read_csv() 메소드의 dtype 키워드는 데이터프레임을 불러올 때 데이터 타입을 지정함으로써 데이터프레임을 최적화 할 수 있다.

- dtype : accept a dictionary that has column names as the key and Numpy type object as the values
- parse_dates : accepts a list of strings containing the nmae of columns we want to parse as datetime
- usecols : specify which columns we want to include

```
keep_cols = ['ExhibitionID', 'ExhibitionNumber', 'ExhibitionBeginDate', 'ExhibitionEndDate', 'ExhibitionSortOrder', 'ExhibitionRole', 'ConstituentType', 'DisplayName', 'Institution', 'Nationality', 'Gender']

import pandas as pd
moma = pd.read_csv("moma.csv", parse_dates = ['ExhibitionBeginDate', 'ExhibitionEndDate'], usecols = keep_cols)
moma.head()
```

4. Processing DataFrames in Chunks

4.1 Chunks

데이터프레임의 데이터 타입을 최적화하고 적절한 열을 선택한 후에도 메모리에 데이터셋의 크기가 적합하지 않을 때가 있다. 이때 전체 데이터프레임을 메모리에 로드하는 것보다 **Chunk** 단위로 처리하는게 효율적이다. 주어진 시간동안 전체 행의 일부분만 메모리에서 사용되어야 한다. 즉, 우리는 데이터의 일부분만 사용해서 작업을 처리하고 결과를 즉시 결합한 다음, 최종적으로 다시 합쳐야 한다.

pandas.read_csv() 메소드의 chunksize 키워드에 값을 입력하면 pandas.io.parser.TextFileReadert 라는 객체를 생성한다. 해당 데이터프레임 청크는 입력받은 chunksize의 행을 저장하고 있다.

예를들어 34,558개의 행을 가지고 45MB의 메모리를 필요로 하는 데이터프레임이 존재하고 해당 데이터프레임에 대해서 1MB의 여유 메모리를 가지고 있다면 어떻게 chunksize를 설정해야 하는지 대략적으로 추론할 수 있다. 약 250개의 행이 0.3MB를 가지고 있기 때문에 chunksize를 250으로 설정하면 여유 메모리 내에서 데이터프레임을 처리할 수 있다.

```
import pandas as pd
import matplotlib.pyplot as plt

chunk_iter = pd.read_csv('moma.csv', chunksize = 250)
memmory_footprints = []

for chunk in chunk_iter :
    memory_footprints.append(chunk.memory_usage(deep = True).sum()/(2**20))

plt.hist(memory_footprints)
plt.show()
```

4.2 Batch processing in pandas

```
num_rows = 0
chunk_iter = pd.read_csv('moma.csv', chunksize = 250)
for chunk in chunk_iter :
    num_rows += len(chunk)
    print(num_rows)
```

다음 코드의 결과 num_rows의 값은 250, 500, 750, ... , 34558로 250개의 행이 chunk 내부에 저장되어 있음을 확인할 수 있다. **Batch processing**에서는 chunk를 분할하고, 서로 다른 chunk를 독립적으로 처리한 뒤, 다시 합치는 작업이 매우 중요하다.

Python 내부에서 각각의 청크에서 연산을 수행하고 다시 합치는 매우 소모적이다. Python 객체는 메모리를 재할당해서 저장하기 때문이다. 다만 이 작업은 Pandas 객체의 최적화로 인해 가능하다. pandas.concat() 메소드는 서로 다른 Series를 행을 기주능로 쌓을 수 있다.

```
import pandas as pd

dtypes = {'ConstituentBeginDate' : 'float', 'ConstituentEndDate' : 'float'}
chunk_iter = pd.read_csv('moma.csv', chunksize = 250, dtype = dtypes)
lifespans = []
for chunk in chunk_iter :
    diff = chunk['ConstituentEndDate'] - chunk['ConstituentBeginDate']
    lifespans.append(diff)
lifespans_dist = pd.concat(lifespans)
```

4.3 value_counts() in chunk

```
chunk_iter = pd.read_csv("moma.csv", chunksize=250, usecols=['Gender'])

overall_vc = []
for chunk in chunk_iter :
    chunk_vc = chunk['Gender'].value_counts()
    overall_vc.append(chunk_vc)
combined_vc = pd.concat(overall_vc)
print(combined_vc)
```

Series.value_counts() 메소드의 실행결과 각 청크별로 남/녀의 고유값의 수를 확인만 하고 전체 데이터프레임의 남/녀 고유값의 수는 확인하지 못한다. 각각의 고유값의 인덱스에 따른 값을 더하기 위해선 저장된 동일한 인덱스 별로 값을 더할 필요가 있다.

DataFrame.groupby() 메소드는 pandas.GroupBy 객체를 생성하면서 고유값의 인덱스를 저장한다. 이후 sum() 메소드를 통해 고유값 별로 데이터를 처리할 수 있다.

```
chunk_iter = pd.read_csv("moma.csv", chunksize=250, usecols=['Gender'])
overall_vc = []

for chunk in chunk_iter :
    chunk_vc = chunk['Gender'].value_counts()
    overall_vc.append(chunk_vc)
combined_vc = pd.concat(overall_vc)
final_vc = combined_vc.groupby(combined_vc.index).sum()
```

5. Augmenting Pandas with SQLite

5.1 SQLite with Pandas

Pandas는 데이터를 메모리 내부에 저장하고 작업한다. 하지만 SQLite와 같은 데이터베이스 툴은 데이터를 디스크 내부에서 처리한다. 즉, Pandas가 이용가능한 메모리 내부에서 데이터를 처리한다면, SQLite는 이용 가능한 디스크 공간 내부에서 데이터를 처리하게 된다. 이를 확장하여 클라우드 내부의 서버에서 데이터를 처리하면 메모리보다 훨씬 더 효율적인 공간을 가지게 된다.

대부분의 경우 SQlite에 저장되어 있는 데이터베이스의 테이블에서 데이터를 추출하여 Pandas에서 데이터를 분석, 탐색, 시각화한다. CSV파일 데이터를 SQLite 데이터베이스 내부로 옮기는 방법은 수동으로 각 문장을 분리하고 INSERT 문을 사용하는 방법이 있다. 추가로 각 파일의 각 청크를 데이터프레임으로써 DataFrame.to_sql() 메소드를 사용해 테이블로 로드하는 방법이 존재한다.

5.2 Load file on database table

DataFrame.to_sql() 메소드를 사용하기 위해선 데이터베이스 내부에 존재하는 테이블에 데이터를 로드하면 안된다. 따라서 cunked 된 데이터프레임을 존재하는 테이블에 로드하기 위해선 if_exists = 'append' argument를 사용해야 한다. 추가적으로 index 옵션을 False로 입력하여 데이터프레임의 인덱스 값이 SQLite 테이블에 추가되지 않도록 해야한다.

```
import sqlite3
import pandas as pd
conn = sqlite3.connect('moma.db')

moma_iter = pd.read_csv('moma.csv', chunksize = 1000)
for chunk in moma_iter :
    chunk.to_sql('exhibitions', conn, if_exists = 'append', index = False)
```

5.3 Retrieve rows using pandas

DataFrame.to_sql(0 메소드를 사용하면 SQLite는 자동으로 데이터프레임에 저장되어 있는 데이터 타입을 SQLite에 맞게 변환하여 저장하게 된다. Pandas.read_sql()은 SQLite에 query할 수 있는 메소드이다.

```
results_df = pd.read_sql('PRAGMA table_info(exhibitions);', conn)
print(results_df)
```

5.4 Optimize datatype on database table

앞서 Pandas 메소드를 통해 최적화를 진행했던 방법을 사용하여 데이터프레임 내부에서 최적화를 진행한 뒤 SQL테이블에 업로드 할 수 있다.

```
moma_iter = pd.read_csv('moma.csv', chunksize = 1000)
for chunk in moma_iter :
    chunk['ExhibitionSortOrder'] = chunk['ExhibitionSortOrder'].astype('int16')
    chunk.to_sql("exhibitions", conn, if_exists='append', index = False)
results_df = pd.read_sql('PRAGMA table_info(exhibitions);', conn)
print(results_df)
```

5.5 Workflow of generating a pandas dataframe

Pandas DataFrame 객체를 생성하는 두 workflow는 다음과 같다.

1. SQL 내부에서 연산을 진행한 뒤 결과를 dataframe으로 파싱한다.
2. SQL 내부에서 데이터를 추출한 뒤 Pandas를 통해 연산을 진행한다.

Pandas가 SQLite보다 좋은점은 데이터프레임을 효율적으로 다루는 다양한 함수, 메소드, 연산자가 있고, memory 내부에서 처리하는 속도가 빠르다.

case1

```
import pandas as pd
import sqlite3
conn = sqlite3.connect('moma.db')

eid_counts = pd.read_sql("""
    SELECT exhibitionid , COUNT(*) AS counts
        FROM exhibitions
    GROUP BY exhibitionid
    ORDER BY counts DESC;
""", conn)
eid_counts.head(10)
```

case2

```
import pandas as pd
import sqlite3
conn = sqlite3.connect('moma.db')

eid_df = pd.read_sql('SELECT exhibitionid FROM exhibitions;', conn)
eid_pandas_counts = eid_df['ExhibitionID'].value_counts()
```

5.6 Read table in chunk

데이터베이스에서 테이블을 읽어올 때 메모리의 문제나 테이블의 크기가 너무 클 경우, read_sql의 chunksize keyword를 통해 데이터프레임을 chunk로 분해해서 작업한 것과 같이 다룰 수 있다.

```
q = 'select exhibitionid from exhibitions;'
chunk_iter = pd.read_sql(q, conn, chunksize=100)
for chunk in chunk_iter:
    # Process each chunk.
```