PyCon Korea 2019

뚱뚱하고 굼뜬 판다(Pandas)를 위한 효과적인 다이어트 전략

오성우

발표자 소개

오성우

이메일: david.oh0126@gmail.com

블로그: sacko.tistory.com

I Love Python & Data!

파이썬과 데이터를 통해 새롭게 세상을 바라보고 이해하는 즐거움을 맛보고 있습니다

데이터 분석 및 컨설팅, 머신러닝, 딥러닝 모델 개발

Pandas

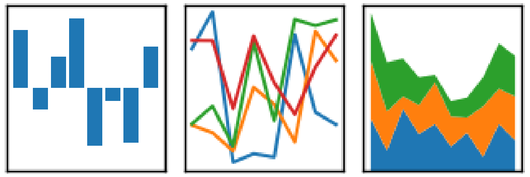
Pandas는 “관계형“ 데이터를 빠르고, 유연하게 그리고 효율적으로 처리하기 위해 개발된 자료구조 입니다. 데이터 처리와 분석에 많이 활용되고 있습니다.

지난 7월 0.25.0 버전 released (0.x 의 마지막 업데이트)

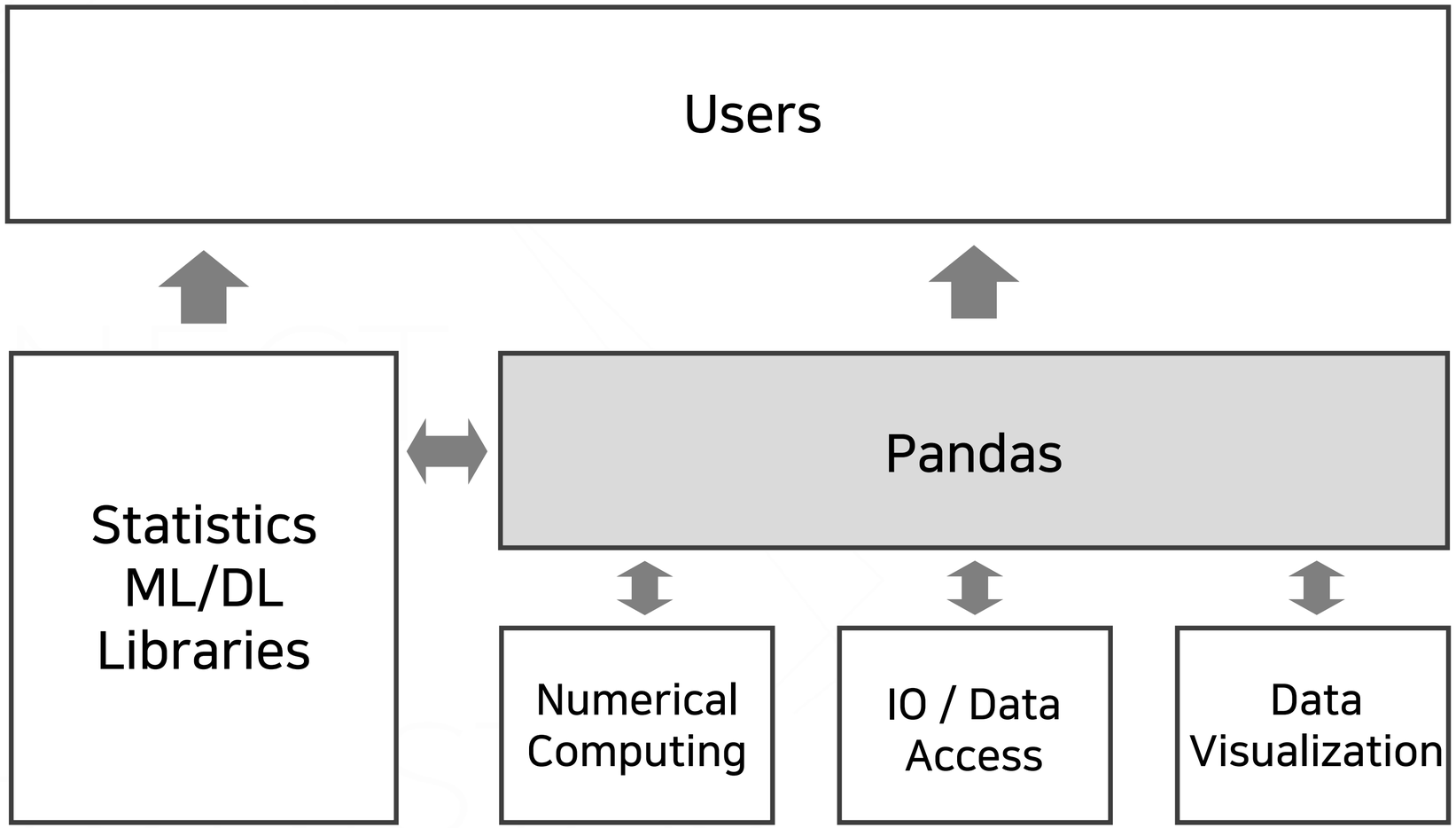
- Tabular Data (SQL, Excel)
- Time-series Data
- Matrix Data
- Any Other Statistical dataset

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$

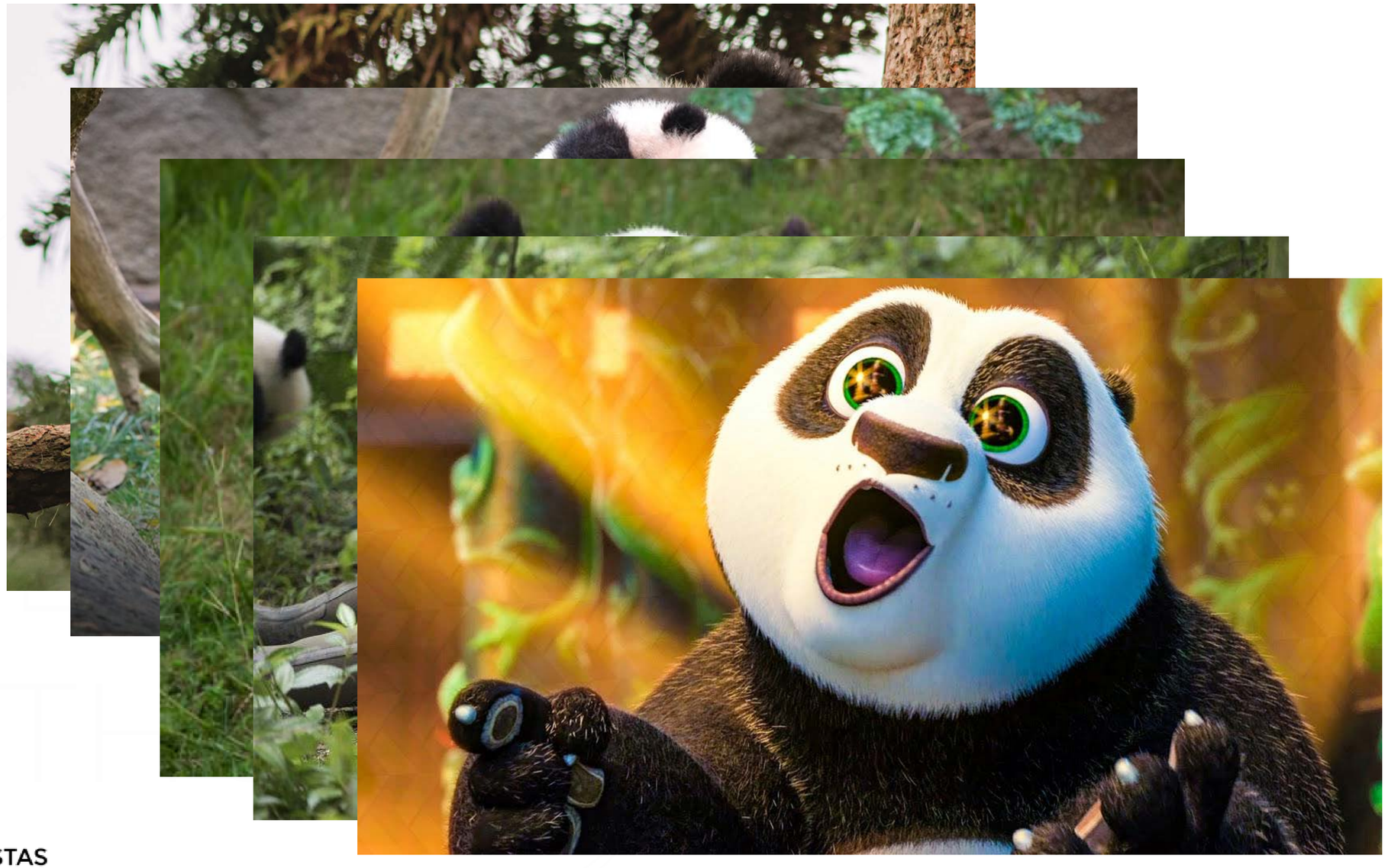


Pandas' role in the Python Data Ecosystem



Source: Jeff Redback, What is the Future of Pandas, PyData New York City 2017

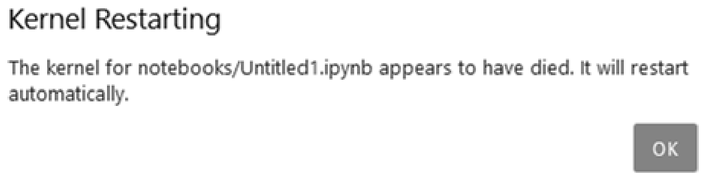
Pandas 하면 떠오르는 이미지는???



흔히 맞닥뜨리는 문제

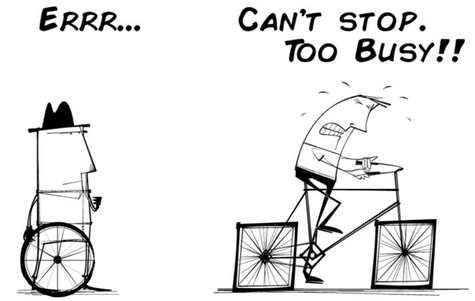
빠르고, 효율적이며, 편리한 API 형태라고 하지만 막상 사용해보면

데이터를 읽지도 못하고



시간도 오래 걸리고

작성하는 코드 방식도 제각각



기술부채 Technical Debt…

원인은??

Pandas Memory mapping issue

Python Auto Garbage collecting

Re-assign unnecessary data frame

Data processing in memory at once

Using wrong syntax

Using one CPU process

Pythonic way? Pandas way!

Python을 사용하는 올바른 방법이 있듯이
Pandas도 마찬가지로 올바른 방법이 있습니다

3가지
다이어트 전략

1. 식사량 조절
2. 식이요법
3. 생활습관

다이어트 컨셉

Pandas가 실제로 빠르고, 유연하고, 효율적이기 위해서는

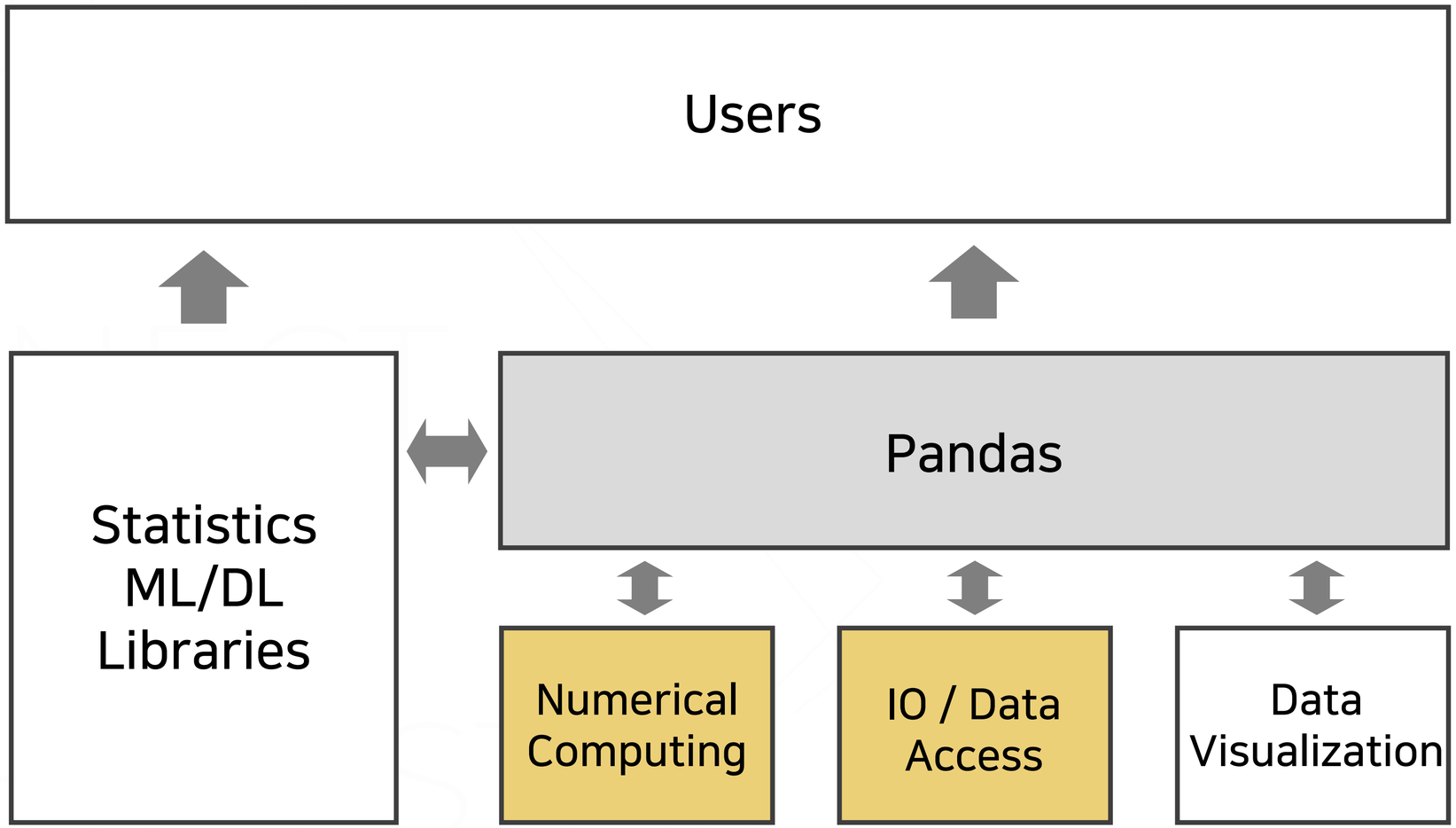
다음의 3가지 전략을 잘 지키시는 것이 좋습니다

전략1
식사량 조절
Memory
Optimization

전략2
식이요법
Enhancing
Performance

전략3
생활습관
Adopting
Convention

다이어트 공략 부위



Source: Jeff Redback, What is the Future of Pandas, PyData New York City 2017

다이어트 검증을 위한 데이터 소개

국민건강보험공단(NHIS)에서 제공하는 공공데이터를 활용하여 전략의 효과 검증

전국민의 2%인 100만명에 대한 국민건강정보 4억 건 중 일부를 전처리하여 사용

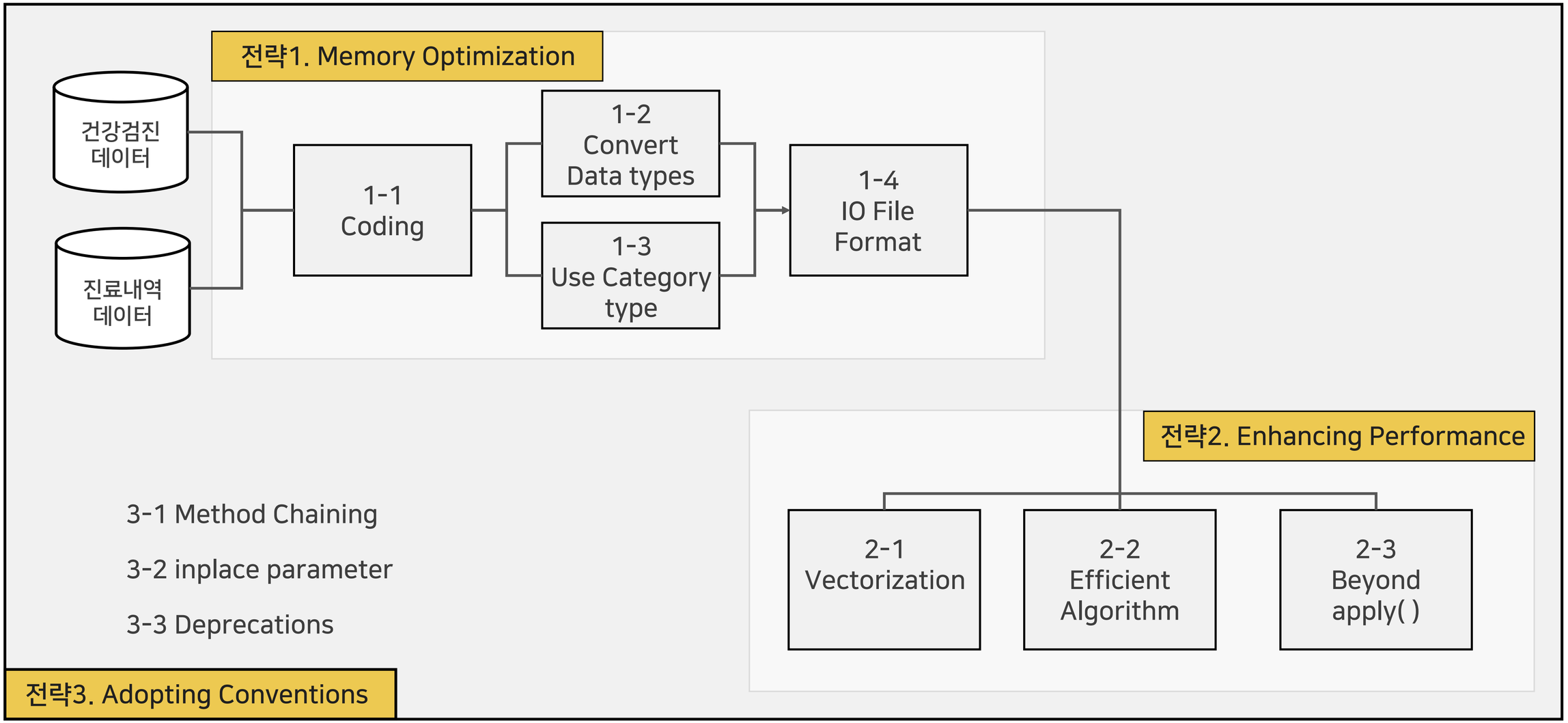
건강검진 데이터

3년치 300만 건, 0.38GB
검진자의 성별, 연령, 신장, 혈압, 허리둘레,
시력 등의 건강검진 정보 포함

진료내역 데이터

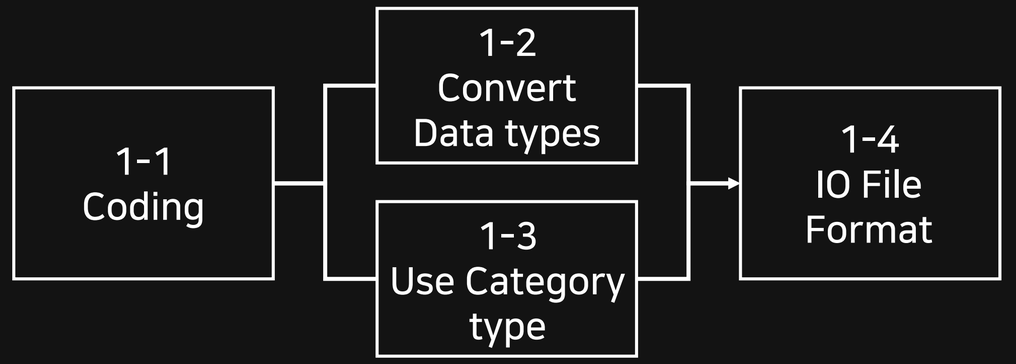
3년치 약 4,000만 건, 3.1GB
병원에 방문한 환자의 진료과목, 질병코드,
입원 등의 진료내역 정보 포함

다이어트 전략 검증 과정


전략1. Memory Optimization
건강검진 데이터
진료내역 데이터
1-1 Coding
1-2 Convert Data types
1-3 Use Category type
1-4 IO File Format
전략2. Enhancing Performance
2-1 Vectorization
2-2 Efficient Algorithm
2-3 Beyond apply( )
3-1 Method Chaining
3-2 inplace parameter
3-3 Deprecations
전략3. Adopting Conventions

전략1

식사량 조절
Memory Optimization



첫 번째 전략 개요

"Pandas rule of thumb: have 5 to 10 times as much RAM as the size of your dataset"

- *Wes McKinney, Creator of Pandas*

- 목표

  큰 사이즈 데이터의 메모리 사용을 최적화

- 세부 내용

  1-1. 코드화

  1-2. 데이터 형식 변환

  1-3. Category type 사용

  1-4. 파일 저장 포맷 변경

1-1 코드화

문자열로 된 데이터를 숫자 / 영어로 변환하여 데이터 크기 축소

- 남자 → 0
  여자 → 1
- 서울특별시 → 11
  대구광역시 → 45
- 정상 → 0
  비정상 → 1

1-1 코드화 결과

한글 문자열로 된 범주 값을 숫자형태로 변환하는 작업을 진행하였더니

4.49GB의 Memory 사용을 보였던 건강검진 데이터는 1.79GB로 크게 감소

	YEAR	PATIENT_ID	SEX	AGE	SIDO	HEIGHT	WEIGHT	WAIST	SIGHT_LEFT	SIGHT_RIGHT	HEAR_LEFT	HEAR_RIGHT	...	SMOKE	DRINK
0	2015	141216	F	14	전라북도	125.0	25.0	56.0	1.2	0.9	정상	정상	...	비흡연	0.0
1	2015	393051	M	14	전라북도	125.0	25.0	54.0	1.0	1.0	정상	정상	...	비흡연	0.0
2	2015	78703	M	14	광주광역시	125.0	30.0	71.0	1.0	1.2	정상	정상	...	비흡연	0.0

	YEAR	PATIENT_ID	SEX	AGE	SIDO	HEIGHT	WEIGHT	WAIST	SIGHT_LEFT	SIGHT_RIGHT	HEAR_LEFT	HEAR_RIGHT	...	SMOKE	DRINK
0	2015	141216	1	14	44	125.0	25.0	56.0	1.2	0.9	0	0	...	0	0.0
1	2015	393051	0	14	44	125.0	25.0	54.0	1.0	1.0	0	0	...	0	0.0
2	2015	78703	1	14	46	125.0	30.0	71.0	1.0	1.2	0	0	...	0	0.0

1-2 데이터 형식 변환

데이터 형식에 따라서 표현하는 값의 범위와 사용하는 메모리 크기가 달라집니다

컬럼마다 고정된 크기(Fixed-length)로 할당하기 때문에
크기가 작은 데이터 형식을 사용하면 메모리 사용량을 크게 줄일 수 있습니다

`Object` > `complex` > `datetime64, float64, int64` > `float32, int32` > …

Data Type	Size	Value range	Memory Usage (10,000rows)
`bool`	1 byte	True / False	9.7 KB
`int8`	1 byte	-128 to 127	9.7 KB
`int16`	2 bytes	-32768 to 32767	19.5 KB
`Int32`	4 bytes	-2**16 to 2**16-1	39.0 KB
`int64`	8 bytes	-2**32 to 2**32-1	78.1 KB
`float32`	4 bytes		39.0 KB
`float64`	8 bytes		78.1 KB
`complex128`	16 bytes		156.3 KB
`datetime64[ns]`	8 bytes		78.1 KB
`object`	variable		656.6 KB

데이터 형식 변환 방법

1. 각 컬럼의 데이터 형식을 아는 경우

```
data_types = {‘col1’: ‘int8’, ‘col2’: ‘float32’}
df = pd.read_csv(data_path, dtype=data_types}
```

2. 데이터를 불러왔지만 크기를 줄이고 싶은 경우

```
df = pd.read_csv(data_path)
data_types = check_dtypes(df)
df = df.astype(data_types)
```

3. 데이터가 커서 불러오지도 못하는 경우

```
columns = pd.read_csv(data_path, rows=0).columns
data_types = {col: check_dtypes(df[col]) for col in columns}
df = pd.read_csv(data_path, dtype=data_types)
```

check_dtypes??

사용자 함수: check_dtypes()

각 컬럼 내 데이터의 최소, 최대 범위를 계산해
적절한 Data type을 찾아내는 함수

각 컬럼의 데이터 형식을 모를 때 자동으로
체크하여 사용하고 데이터를 불러올 때
체크된 데이터 형식으로 데이터를 불러옵니다

형식을 지정하지 않으면 가장 메모리를 많이 차지하는
방식으로 데이터를 불러오기 때문에
형식을 지정해서 데이터를 불러오면 메모리를 줄여
큰 데이터도 불러올 수 있습니다

```python
def check_dtypes(file_path):
    print(file_path)
    tmp = pd.read_csv(file_path, nrows=0)
    col_dtypes = {}
    for col in tmp.columns:
        df = pd.read_csv(file_path, usecols=[col])
        dtype = df[col].dtype

        if dtype == 'int' or dtype == 'float':
            c_min = df[col].min()
            c_max = df[col].max()
        elif dtype == 'object':
            n_unique = df[col].nunique()
            threshold = n_unique / df.shape[0]

        if dtype == 'int':
            if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8).max:
                col_dtype = 'int8'
            elif c_min > np.iinfo(np.uint8).min and c_max < np.iinfo(np.uint8).max:
                col_dtype = 'uint8'
            elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.int16).max:
                col_dtype = 'int16'
            elif c_min > np.iinfo(np.uint16).min and c_max < np.iinfo(np.uint16).max:
                col_dtype = 'uint16'
            elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.int32).max:
                col_dtype = 'int32'
            elif c_min > np.iinfo(np.uint32).min and c_max < np.iinfo(np.uint32).max:
                col_dtype = 'uint32'
            elif c_min > np.iinfo(np.int64).min and c_max < np.iinfo(np.int64).max:
                col_dtype = 'int64'
            elif c_min > np.iinfo(np.uint64).min and c_max < np.iinfo(np.uint64).max:
                col_dtype = 'uint64'

        elif dtype == 'float':
#             # ERROR occured When using float32 in feather, parquet
#             if c_min > np.finfo(np.float16).min and c_max < np.finfo(np.float16).max:
#                 col_dtype = 'float16'
            if c_min > np.finfo(np.float32).min and c_max < np.finfo(np.float32).max:
                col_dtype = 'float32'
            else:
                col_dtype = 'float64'

        elif dtype == 'object':
            if threshold > 0.7:
                col_dtype = 'object'
            else:
                col_dtype = 'category'

        col_dtypes[col] = col_dtype

    return col_dtypes
```

Referenced: https://www.mikulskibartosz.name/how-to-reduce-memory-usage-in-pandas/

1-3. Category 데이터 형식 사용

- 특징
  - Finite Value에 적합한 Data type으로 Unique value가 반복해서 나타나는 경우 사용
  - Int8로 데이터를 저장하고, Ordered과 Unordered 방식 모두 가능

- 장점
  - Object를 Category로 변환했을 때 memory 사용량 크게 감소
  - 데이터 처리 및 분석 속도 향상
  - 시각화 등에서 편의성 향상

- 한계
  - 범주가 무수히 많은 경우에는 object 보다 비효율적일 수 있음

Category 사용 시 분석, 처리속도 향상

진료내역 정보에서 가장 많이 진단된 최빈도 5개 질병에 대한 분석

Object 형식을 사용했을 때보다 Category 형식을 사용했을 8배 가까운 향상

Object: 2.3초 , Category: 0.3초

Object

```
t20['MAIN_SICK'] = t20['MAIN_SICK'].astype('object')
```

```
%%time
t20['MAIN_SICK'].value_counts().nlargest()
```

```
CPU times: user 2.23 s, sys: 70 ms, total: 2.3 s
Wall time: 2.32 s
J209    3038222
I109    1902664
F_       645754
E119     548867
J304     529549
Name: MAIN_SICK, dtype: int64
```

Category

```
t20['MAIN_SICK'] = t20['MAIN_SICK'].astype('category')
```

```
%%time
t20['MAIN_SICK'].value_counts().nlargest()
```

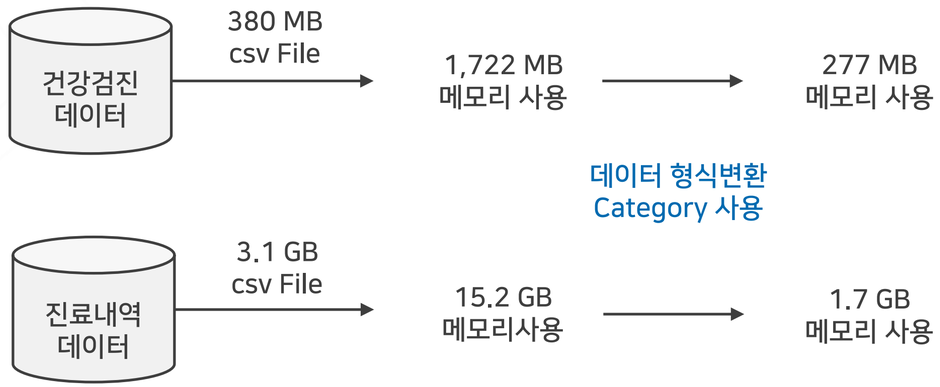
```
CPU times: user 210 ms, sys: 90 ms, total: 300 ms
Wall time: 299 ms
J209    3038222
I109    1902664
F_       645754
E119     548867
J304     529549
Name: MAIN_SICK, dtype: int64
```

1-2 데이터 형식 변환, 1-3 Category 사용 결과

건강검진 데이터 메모리 최적화 결과
: 메모리 사용량 84% 감소

처음 건강검진 데이터(380MB)의 csv 파일을 불러왔을 때 1,722MB의 메모리를 사용했지만, 최적화 이후 277MB 까지 감소하였습니다. (-84%)

크기가 커서 불러오지도 못했던 진료내역 데이터(3.1GB)는 최적화 이후 1.7GB의 메모리로 불러올 수 있었습니다



1-4 파일 저장 포맷 변경

데이터 분석을 하다 보면 처리된 데이터를 따로 저장하거나 다른 사람에게 공유해야 될 때가 많습니다

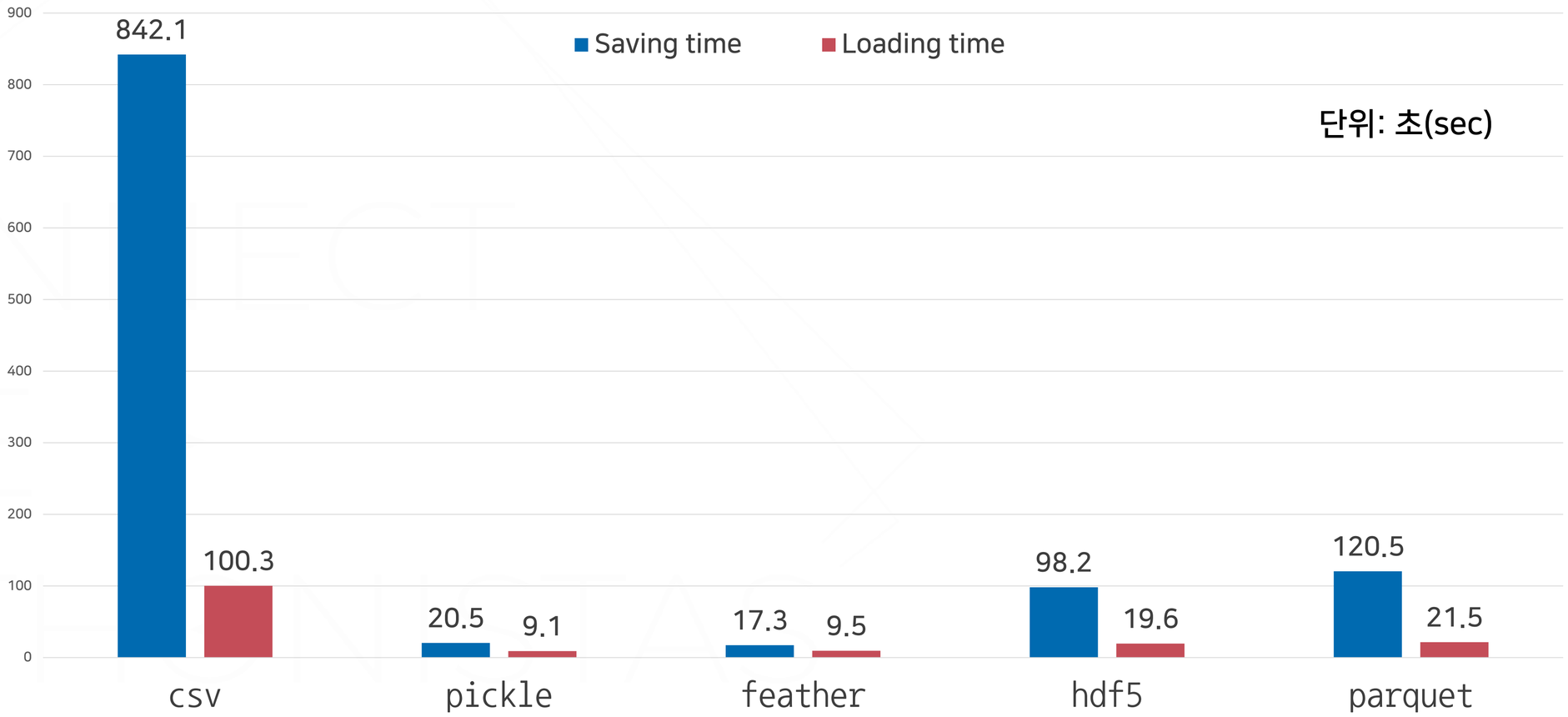
일반적으로 사용하는 CSV 포맷의 단점

- string 기반으로 IO 효율이 떨어짐
- Meta data가 없어 각 컬럼의 데이터 형식 등에 대해 연속성을 가지고 사용 불가

최근 많이 사용되고 있는 hdf5, parquet 포맷과 pickle, feather 같은 다른 포맷과
1) 데이터를 읽고 쓰는 시간, 2) 메모리 사용량, 3) 저장된 파일의 크기를 비교할 것입니다.

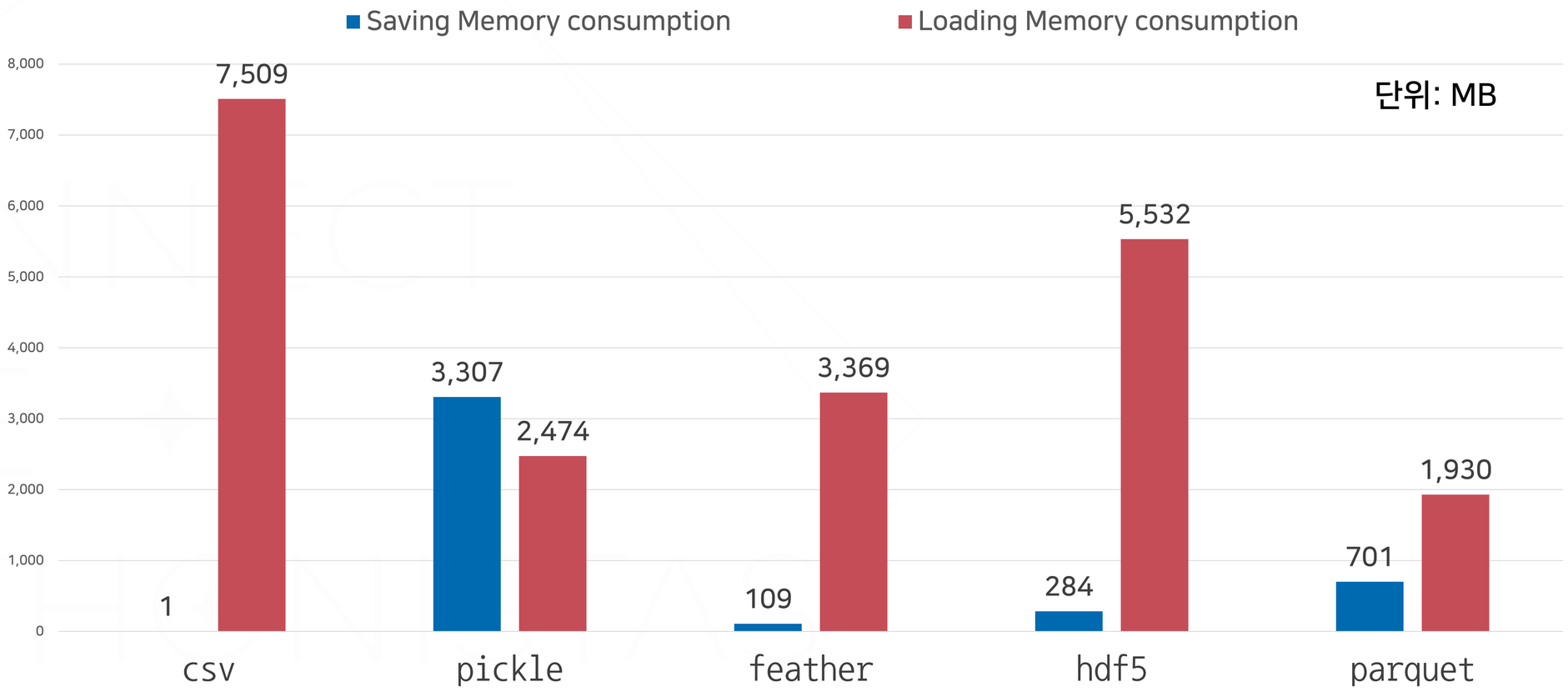
파일형식에 따른 데이터 저장, 로드 시간 비교

pickle, feather < hdf5, parquet <<< csv



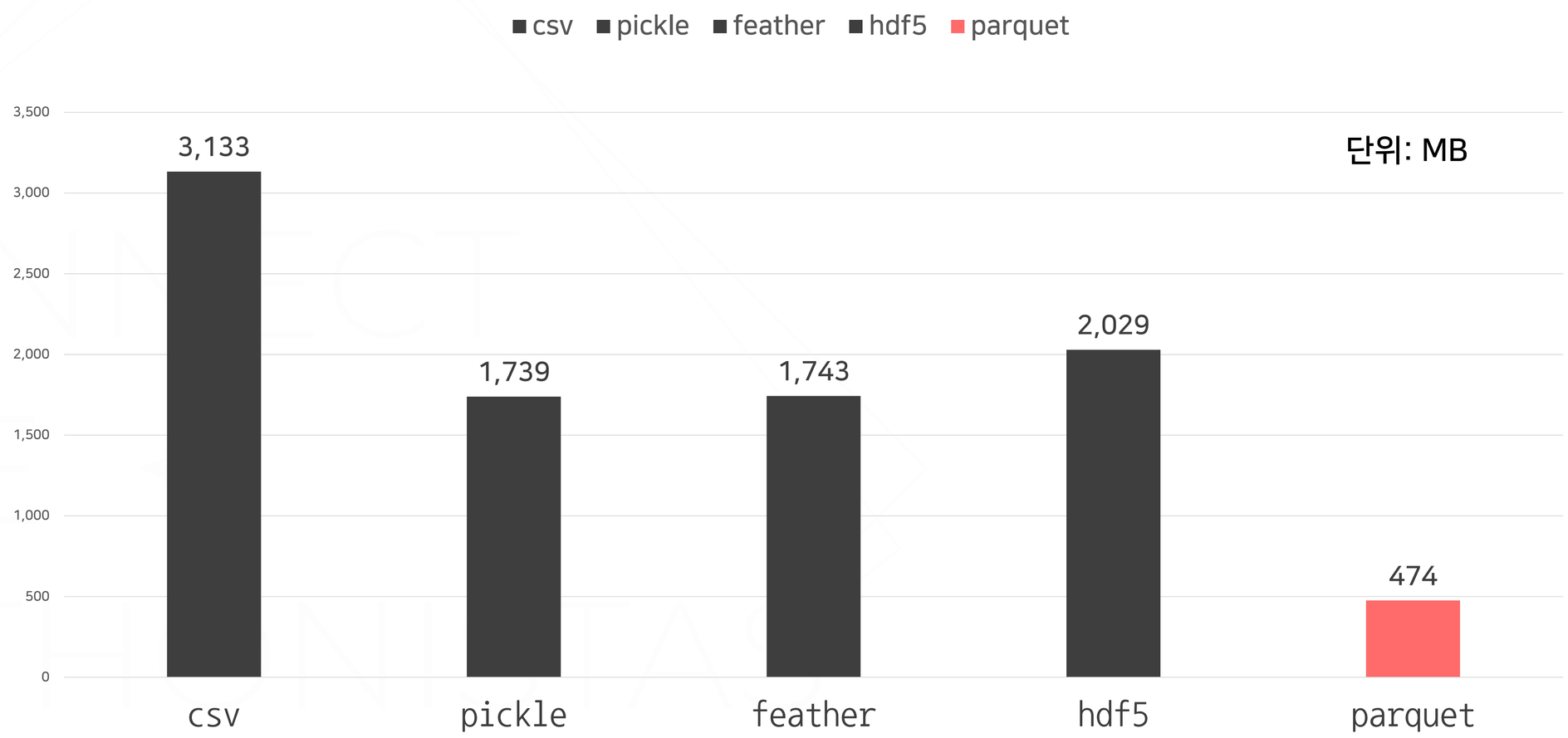
파일 형식에 따른 IO 시의 메모리 사용량

parquet < pickle, feather < hdf5 < csv



파일 형식에 따른 저장되는 파일의 크기

parquet < pickle < feather < hdf5 < csv



추천하는 파일 포맷 형식

개인적 사용 목적: feather, pickle

프로젝트, 협업 목적: parquet, hdf5

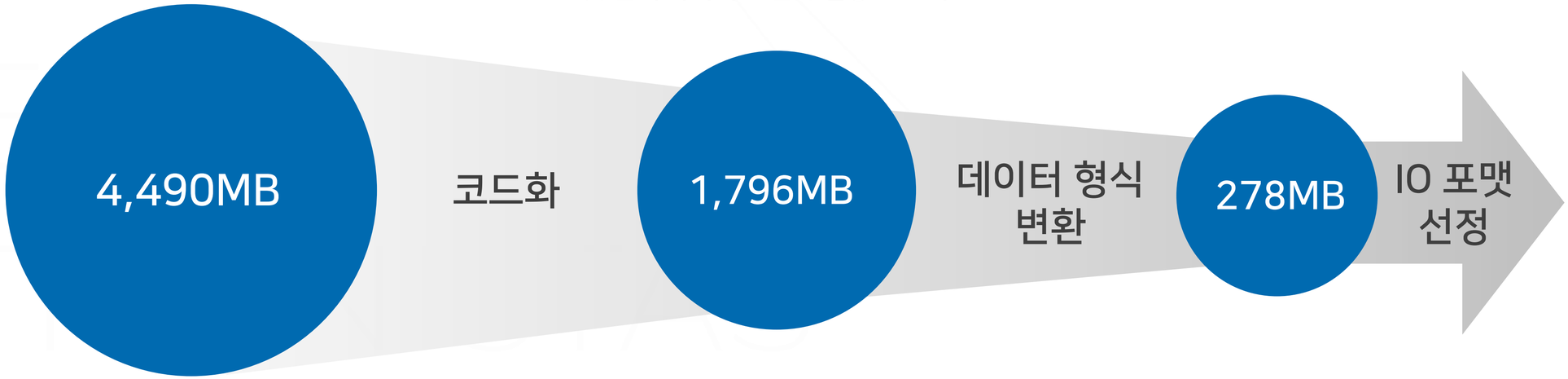
- Parquet is way to go
  - Spark, Hive, Impala, AWS services, BigQuery와의 연계가 용이
  - PySpark, Dask와 같은 프레임워크와 함께 사용할 때 편리
  - AWS의 S3, Athena에 parquet 데이터를 바로 저장하거나 불러오기도 함

- HDF5 is also recommended
  - Pandas에서 저장한 hdf 포맷 그대로 h5py 라이브러리로 사용 가능
  - 기타 Hadoop service와도 잘 맞음

첫 번째 전략 요약

1. 데이터를 불러올 때 데이터 형식을 파악해서 불러오면 90% 가까이 메모리를 줄일 수 있습니다
2. 심지어 불러오지 못했던 파일도 불러올 수도 있습니다
3. 적절한 파일 포맷을 사용하면 데이터를 불러오고 저장하는 시간과 메모리를 아낄 수 있습니다

건강검진 데이터
첫 번째 다이어트 전략 검증 프로세스

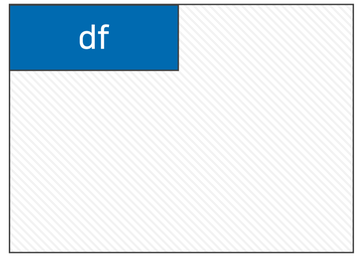


별첨. 데이터 일부분만 불러오기

read.csv 의 parameter를 활용

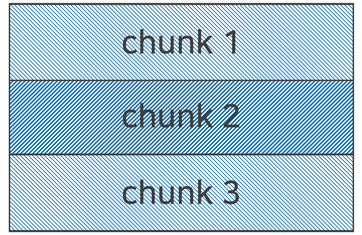
1) 필요한 열과 행 만 가져오는 방법

```
pd.read_csv(file_name, usecols=[ ‘ A ’ ,  ‘ B ’ ], nrows=1000)
```



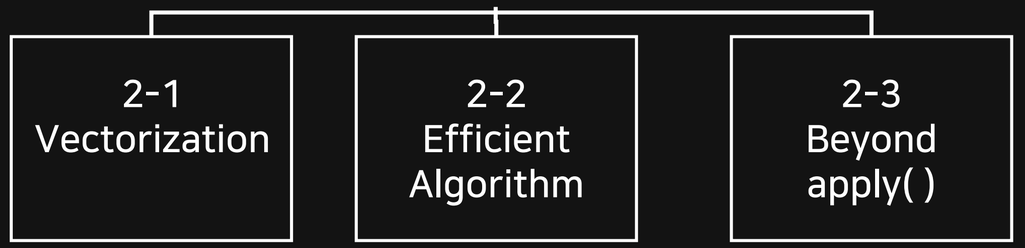
2) 전체 데이터를 정해진 크기로 잘라 Generator로 불러오는 방법

```
chunks = pd.read_csv(file_name, chunksize=1000)
df = pd.DataFrame(columns=[ ‘ A ’ ,  ‘ B ’ , ⋯ ])
for chunk in chunks:
    tmp = do_something(chunk)
    df.append(tmp, ignore_index=True)
```



전략2

식이습관
Enhancing Performance



두 번째 전략 개요

- 목표

  Enhancing performance when manipulating, analyzing data

- 세부 내용

  2-1. Vectorization을 사용하세요

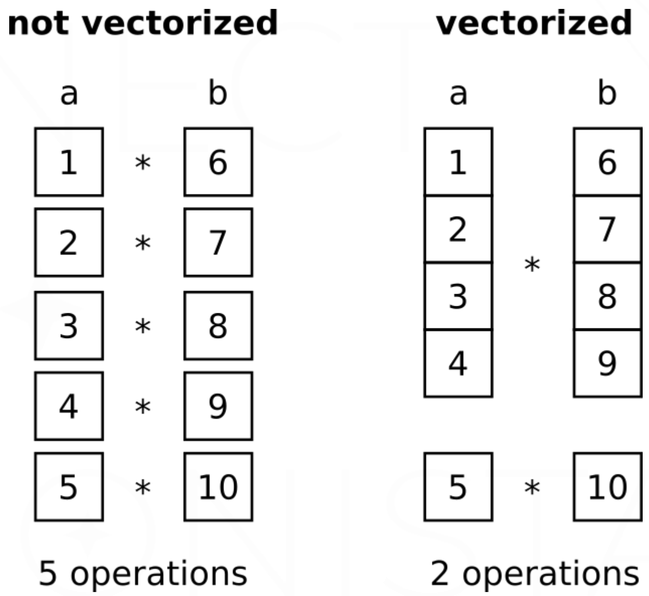
  2-2. 효율적인 Algorithm을 고려하세요

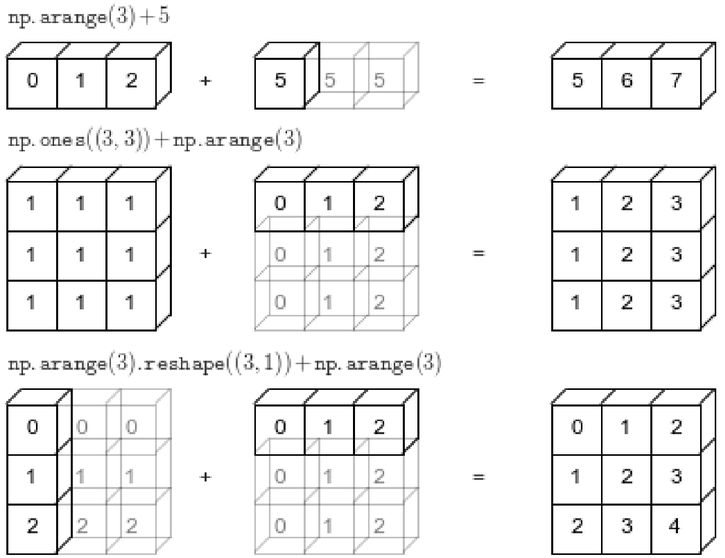
  2-3. pd.apply()는 만능키가 아닙니다

2-1. Vectorization?

벡터화(Vectorization) 연산을 사용하면 명시적으로 반복문을 사용하지 않고도
배열의 모든 원소에 대해 반복연산이 가능합니다

실제 건강검진 데이터를 분석하면서 Performance를 비교하겠습니다





https://www.johncanessa.com/2018/12/24/numpy-vectorization/

2-1 건강위험지수 산출하기

2017년 한 해 건강검진을 받은 100만 명의 검진자들의 건강위험지수 계산

위험 score가 높을수록 건강 위험성이 높음 (0점 ~ 10점)

*scoring_health : 건강검진 결과로 건강위험지수 산출을 위한 지표를 산출하는 함수
각 항목별로 1과 0으로 정상, 위험으로 구분하여 계산

1. BMI
2. 허리둘레
3. 공복혈당
4. 총콜레스테롤
5. 트리글리세라이드
6. 혈색소
7. 혈청크레아티닌
8. 혈청지오티
9. 흡연여부
10. 음주여부

```
def scoring_health(patient):
    bmi = ((patient[ ' Weight ' ] / (patient[ ' Height ' ]/100**2) >= 30) * 1
    waist = (patient[ ' WAIST ' ] >= 90) * 1
    blds = (patient[ ' BLDS ' ] >= 125) * 1
    chole = (patient[ ' TOT_CHOLE ' ] >= 130) * 1
    trigly = (patient[ ' TRIGLYCERIDE ' ] >= 150) * 1
    hmg = (patient[ ' HMG ' ] < 12) * 1
    crea = (patient[ ' CREATININE ' ] > 1.7) * 1
    sg = ((patient[ ' SGOT_AST ' ] >= 40) | (patient[ ' SGPT_ALT ' ] >= 40)) * 1
    smoke = (patient[ ' SMOKE ' ] == 3) * 1
    drink = (patient[ ' DRINK ' ] == 1) * 1

    patient_score = np.sum([bmi, waist, blds, trigly, hmg, crea, sg, smoke, drink], axis=0)

    return patient_score
```

산출을 위한 방법 비교

- Iteration by indexing with `len(df)`
- Iteration by using `.iterrows()`
- Use Pandas `.apply()`
- Vectorization with Pandas Series
- Vectorization with Numpy Array

방법별로 건강위험지수 산출 수행 시간 비교

반복문을 사용하는 것보다는 Vectorization을 사용했을 때 훨씬 빠른 속도를 보여주었습니다

방법	코드	수행 시간 (100만명)

Iteration by indexing with len(df)	`scores = []`
	`for i in range(len(df)):`
	`  patient = df.iloc[i]`
	`  scores.append(scoring_health(patient))`
		7분 23초

Iteration by using .iterrows()	`scores = []`
	`for patient in df.iterrows():`
	`  scores.append(scoring_health(patient))`
		4분 28초

Use Pandas .apply()	`scores = df.apply(scoring_health, axis=1)`	2분 33초
Vectorization with Pandas Series	`scores = scoring_health(df)`	2.3초
Vectorization with Numpy Array	`scores = scoring_health_np(df)`	0.7초

632배 속도 증가

기타. np.vectorize

Custom 함수의 vectorization을 쉽게 도와주는 Numpy 함수

tf.function 과 역할이 비슷

```
@np.vectorize
def func(a, b):
    return a * b
```

```
def func(a, b):
    return a * b
np_func = np.vectorize(func)
```

2-2 Considering Efficient Algorithm

어떤 방법을 사용하는지에 따라서 데이터의 처리와 분석 속도가 천차만별

같은 결과를 가져올 수 있는 함수가 여러 개 있지만 상황에 따라 전혀 다른 실행 성능을 보여줍니다

Series 는 325개의 methods/attributes를,
DataFrame은 224개의 methods/attributes 제공하기 때문에
적절하게 조합하여 사용하면 성능의 향상을 기대할 수 있습니다

병원 비용이 가장 높았던 진료내역서 추출

진료내역 데이터(T20)에는 2015년부터 2017년까지의 100만명의 환자가 진료를 받고 청구한 내역데이터가 약 4,000만 건이 있습니다

여기서 요양 급여비용이 높았던 진료내역 상위 5개를 추출해보았습니다

같은 결과를 주는 2가지 방법을 비교했을 때 10배가 넘는 차이가 나타났습니다

```
%%time
tmp = t20['EDEC_TRAMT'].sort_values(ascending=False).head(5)
```

```
CPU times: user 7.21 s, sys: 610 ms, total: 7.82 s
Wall time: 7.81 s
```

```
%%time
tmp = t20['EDEC_TRAMT'].nlargest(5)
```

```
CPU times: user 290 ms, sys: 390 ms, total: 680 ms
Wall time: 683 ms
```

2-3 Pandas .apply() ???

많은 경우에 Custom 함수를 만들어서 apply를 사용해 데이터 처리 또는 분석을 합니다.
(ex. 조건 추출, 그룹 함수 등)

대부분의 경우에 pandas의 apply는 원하는 대로 작동합니다.
Lambda 함수와 함께 많이 사용됩니다.

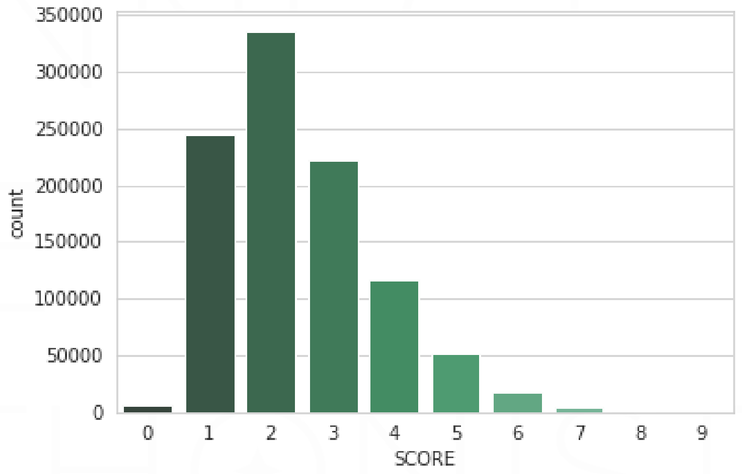
하지만 더욱 효율적인 방법들을 생각해야 합니다.

건강위험지수가 높은 사람들의 진료내역 추출

2-1 Vectorization 실험에서 위험지수를 산출했습니다

위험지수 6 이상 받은 사람들의 ID 번호로 진료내역을 추출하는 것이 이번 목표입니다

어떤 방식을 사용했을 때 가장 효과적이었는지 살펴보도록 하겠습니다



진료내역 데이터 샘플

	YEAR	PATIENT_ID	SEQ	SEX	AGE	SIDO	FORM	DSBJT	MAIN_SICK	SUB_SICK	VSCN	RECN	EDEC_ADD	EDEC_TRAMT
0	2015	970070	1	1	6	29	2	4	K358	NaN	5	9	0.0	1621240
1	2015	372596	2	1	8	41	2	4	K358	NaN	6	13	0.0	2545350
2	2015	363860	3	2	9	41	2	4	K358	NaN	4	11	0.0	2344790
3	2015	192334	4	2	6	47	2	10	O342	O601	7	14	0.0	2314100
4	2015	960084	5	1	1	47	2	13	J353	H659	4	9	0.0	1121030

진료내역 추출을 위한 방법 비교

- List Comprehension
- pd.Series.apply()
- pd.DataFrame.isin
- pd.DataFrame.query
- np.isin
- pd.DataFrame.merge

방법 별 특정 환자의 진료내역 추출 소요 시간

데이터의 형태와 조건에 따라 적절한 방법을 선택하는 것이 좋습니다

방법	코드	수행 시간 (4,000만 건)
List Comprehension	`df = df[[x in patients for x in df[ ' PATIENT_ID ' ]]]`	6분 21초
pd.Series.apply()	`df = df[df['PATIENT_ID'].apply(lambda x: x in patients)`	6분 26초
pd.DataFrame.isin()	`df = df[df.isin({ ' PATIENT_ID ' : patients})[ ' PATIENT_ID ' ]]`	11.3초
pd.DataFrame.query()	`df = df.query( ' PATIENT_ID in @patients ' )`	6.2초
np.isin()	`df = df[np.isin(df[ ' PATIENT_ID ' ], patients)]`	5.1초
pd.DataFrame.merge()	`df = df.merge(patients, how= ' inner ' , on= ' PATIENT_ID ' )`	3.6초

105배 속도 증가

두 번째 전략 요약

1. 반복문 사용은 피하고 Vectorization을 사용합니다
2. 수행 시간을 더 빠르게 하고 싶다면 좀 더 효율적인 알고리즘을 고려해봅니다
3. Custom 함수를 사용할 때 우선은 apply를 적용하고
   수행 시간이 오래 걸리면 적절한 Pandas built-in 함수를 찾아 조합하여 사용합니다

전략3

생활습관
Adopting Conventions

3-1 Method Chaining

3-2 inplace parameter

3-3 Deprecations

세 번째 전략 개요

- 목표

  Pandas에도 pep8처럼 coding convention들이 있음을 소개

  버전 1.0을 향해 나아가고 있는 Pandas의 변화 방향 소개

- 세부 내용

  3.1 Method Chaining

  3.2 inplace parameter

  3.3 Deprecations will be removed in 1.0

3-1. Method Chaining

- 특징
  - 가독성이 좋음 (Readability)
  - 성능이 좋음 (Performance)

- 한계
  - DataFrame의 중간 체크가 어려움
  - 진짜 성능이 좋아지는지는 물음표

일반적인 방법

```
jack_jill = JackAndJill()
on_hill = went_up(jack_jill, 'hill')
with_water = fetch(on_hill, 'water')
fallen = fell_down(with_water, 'jack')
broken = broke(fallen, 'jack')
after = tmple_after(broken, 'jill')
```

Method Chaining

```
jack_jill = JackAndJill()
after = (
         jack_jill
             .went_up("hill")
             .fetch("water")
             .fell_down("jack")
             .broke("crown")
             .tumble_after("jill")
        )
```

3-2. inplace parameter

“inplace” parameter는 Pandas DataFrame에서 작업을 하다가 결과를 바로 해당 DataFrame에 덮어씌우고 싶을 때 많이 사용합니다.

```
df = pd.read_csv( ‘ titanic.csv ’ )

# 일반적인 방법

df = df.sort_values( ‘ SEX ’ )

# inplace를 사용하는 방법

df.sort_values( ‘ SEX ’ , inplace=True)
```

inplace를 왜 사용하는가, 왜 사용하지 말아야 하는가

inplace를 선호하는 사용자들의 여러 의견은 다음과 같습니다

- 속도가 더 빠르다
- 메모리를 더 효율적으로 사용한다

이에 Pandas Core 개발자들은 반대를 하고 있으며,
이미 inplace parameter에 대한 deprecation 및 삭제를 논의하고 있고,
Method Chaining 방식을 적극 권장하고 있습니다

inplace 실행 이후에도 메모리에 데이터가 남아있는 문제가 자주 있기 때문입니다

Referenced: https://github.com/pandas-dev/pandas/issues/16529

어떤식으로 구분해서 사용하는 것이 좋은가

- Method Chaining
  - 결과 DataFrame의 전체를 생성하고 재할당(reassign)하는 특징
  - chaining 과정에서 데이터가 크기가 줄어들 때 메모리 효율적
  - 따라서 .drop(), .astype() 등을 우선적으로 사용하는 것이 좋음

- inplace parameter
  - 추상화된 함수 내부에서 DataFrame의 일부만 생성되어 재할당(reassign)되는 특징
  - 큰 작업 단위에서 Method Changing 보다 성능, 메모리 사용에서 종종 우위를 보임
  - 따라서 .set_index(), .rename() 과 같이 일부 내용만 변경하는 경우에 효율적

3-3. Deprecations

Pandas Core 개발자인 Marc Garcia가 2018년 London에서 열린 PyData meetup에서 Pandas 1.0의 청사진을 이야기하며

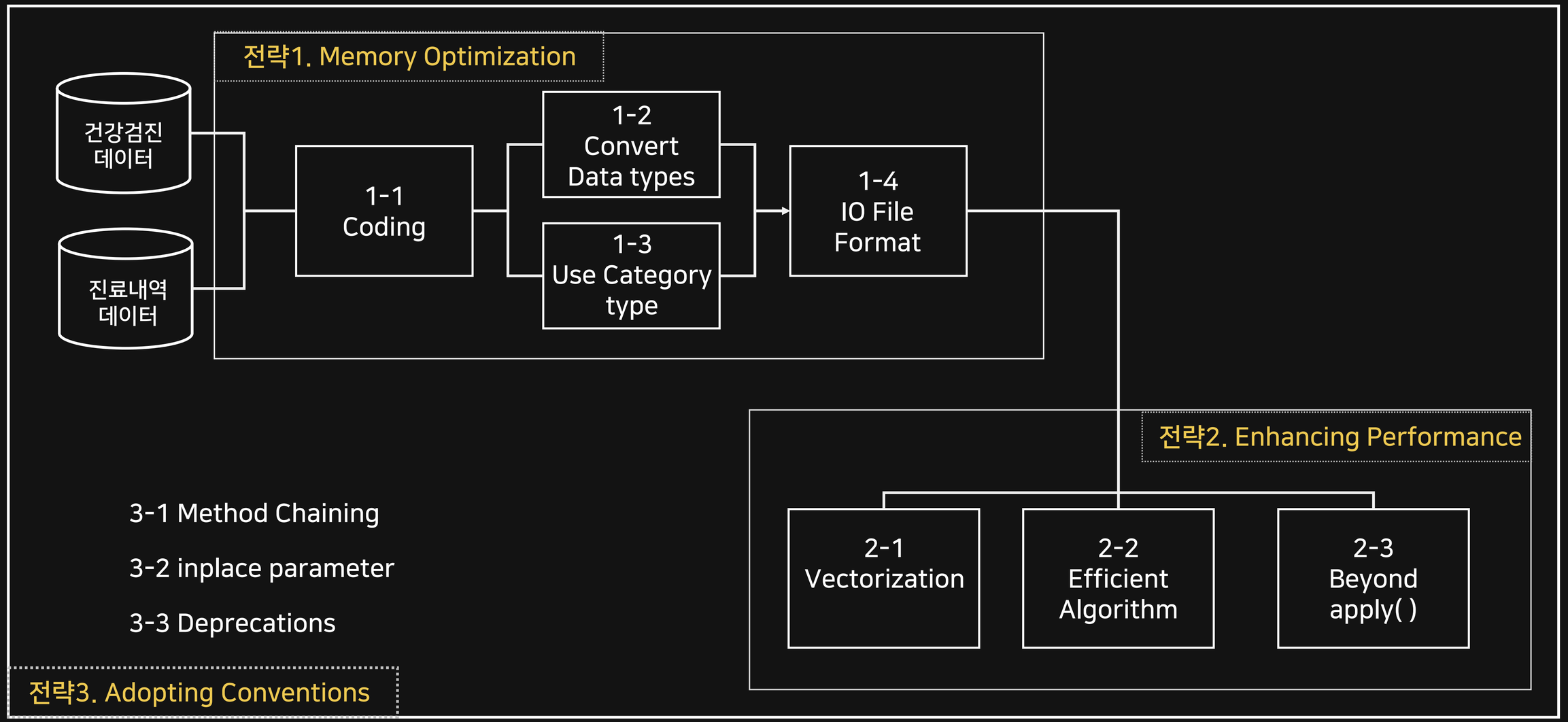
.ix 와 같은 오래전부터 deprecation으로 등록되어 있는 것들은
Pandas 1.0에서는 모두 삭제될 것이라고 밝혔습니다.

Pandas 0.25.x 까지는 사용 가능
0.25 버전부터는 Python 2 지원 제외

https://www.youtube.com/watch?v=hK6o_TDXXN8

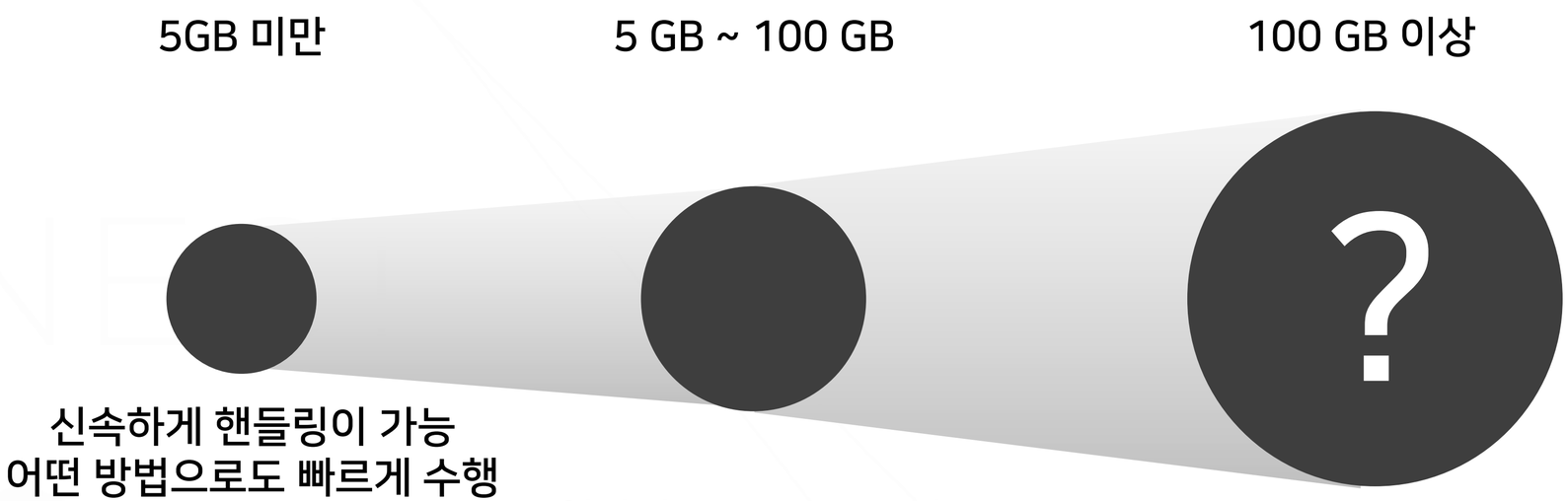
3-4. 기타 유의사항

- Case by Case
- Garbage collection
- Not designed for Big Data?


전략1. Memory Optimization
건강검진 데이터
진료내역 데이터
1-1 Coding
1-2 Convert Data types
1-3 Use Category type
1-4 IO File Format
전략2. Enhancing Performance
2-1 Vectorization
2-2 Efficient Algorithm
2-3 Beyond apply( )
3-1 Method Chaining
3-2 inplace parameter
3-3 Deprecations
전략3. Adopting Conventions

적절한 데이터 크기는?

빅데이터는 어디서부터 인지… Pandas로 적합한 건 어디까지 인지…



Referenced: Jeff Redback, What is the Future of Pandas, PyData New York City 2017

How Scaling?

1. Buy more RAM...
2. Other Dataframe frameworks
   - Dask, Modin
3. GPU
   - Numba, cuDF
4. Parallelization module
   - multiprocessing
5. DBMS, Spark ...

“ 모든 인간은 본성상 알기를 원하고 앎을 통해 즐거움을 느낀다 ”

- 아리스토텔레스

감사합니다

Thank you for your time and consideration