1st week summary

<NumPy: Ndarray>

```
import numpy as np

array1 = np.array([1, 2, 3])
print('array1 array 형태:' array1.shape)
print('array1: {:0}차원'.format(array1.ndim))

'''output
array1 array형태: (3,)
array1: 1차원
'''
```

- **np.array()**: ndarray로 변환
- **.shape**: ndarray의 차원, 형태를 튜플 형태로 나타냄

  ex. 1차원 (3,), 2차원(row, col)
- **.ndim**: array의 차원 확인

Ndarray의 데이터 타입

🚀 **Numeric**
bool, int, unsignded ing(부호 없는 정수형), float, complex(복소수)

Character
string

- **.dtype**: 데이터 타입 확인(ex. int32, float64 etc.)
- **.astype('float64')**: 데이터 형변환

  cf) 여러 데이터가 섞여있는 경우 데이터 크기가 더 큰 타입으로 형변환 일괄 적용

ndarray 생성, 초기화

```
sequence_array = np.arange(10)
print(sequence_array)

zero_array = np.zeros((3, 2), dtype='int32')
print(zero_array)

'''output
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[[0 0] [0 0] [0 0]]
'''
```

- **arange(start, stop)**: range와 같음
- **zeros()**: 모든 값을 0으로 채운 shape ndarray 반환, 형변환 지정 없을 시 float64
- **ones()**: 값을 1로 채운 ndarray 반환

reshape(): 특정 차원 및 크기로 변환

```
array1 = np.arange(10)
array2 = array1.reshape[2, 5]
print(array2)

'''output
[[0 1 2 3 4]
[5 6 7 8 9]]
'''
```

cf) -1을 인자로 가질 경우: 고정된 row/col에 맞는 row/col을 자동으로 생성

(ex. `array1.reshape(-1, 5)` 5 col에 맞는 row를 자동생성)

reshape(-1, 1)은 원본이 어떻든 2차원이고 여러 개의 row를 가지되 반드시 1개의 col을 가진 ndarray로 변환됨을 보장한다.

Indexing

- **단일 값 추출**: 해당 위치의 index 값을 []안에 입력

  index -1은 그룹 맨 끝의 데이터 값을 의미

- **슬라이싱**: ':'을 사용하여 시작부터 종료 인덱스-1의 위치에 있는 데이터 반환
- **팬시 인덱싱**: 인덱스 집합을 지정하면 해당 위치의 데이터를 반환
- **불린 인덱싱**: 조건 필터링과 검색을 동시에 가능

  (ex. `array1[array1>5]`)

행렬 정렬

- **np.sort()**: 원 행렬을 유지한 채 정렬된 행렬 변환
- **ndarray.sort()**: 원 행렬을 정렬한 형태로 변환, 반환값 None

  cf) 내림차순 정렬 `[::-1]` (ex. `np.sort()[::-1]`)
- **np.argsort()**: 정렬 행렬의 원본 행렬 인덱스를 ndarray형으로 반환

```
org_array = np.array([3, 1, 9, 5])
sort_indices = np.argsort(ort_array)
print(sort_indices)

'''output
[1 0 3 2]
'''
```

- **np.dot()**: 행렬 곱 계산
- **np.transpose()**: 전치 행렬

<Pandas; DataFrame>

EDA

- **.info():** 총 데이터 건수, 데이터 타입, Null 건수 확인
- **.describe()**: n-percentile 분포도(Q1, Q2, Q3), mean, max. min 등 확인
- **.value_counts()**: 해당 칼럼값의 유형과 건수 확인, 데이터 분포도를 확인하는데 유용

  고유 칼럼 값을 식별자로 사용할 수 있음(48p)
- **.head()**; 데이터의 앞 5개 출력

- **.tail()**: 데이터의 뒤 5개 출력

DataFrame과 상호 변환

ndarray, list, dict를 DataFrame으로 변환

2차원 이하의 데이터들만 DataFrame으로 변환할 수 있으며, 칼럼명 지정 필요

- **pd.DataFrame(**`list1, columns=col1`**)**

  1개 이상의 칼럼명이 필요한 경우 리스트 생성 `col_name=['c1', 'c2', c3']`

딕셔너리를 DataFrame으로 변환 시 딕셔너리의 Key는 칼럼명, Value는 해당 데이터로 변환

```
dict = {'col1':[1, 11], 'col2':[2, 22], 'col3':[3, 33]}
df_dict = pd.DataFrame(dict)
print(df_dict)

'''output
  col1  col2  col3
0    1     2     3
1   11    22    33
'''
```

DataFrame을 ndarray, list, dict으로 변환

- **.values**: values를 이용한 ndarray로의 변환이 매우 많이 사용됨!
- **tolist(), to_dict()**

```
#DataFrame → list
list3 = df_dict.values.tolist()

#DataFrame → dict
dict3 = df_dict.to_dict('list')
```

칼럼 data set 생성/수정/삭제

- [ ] 연산자를 이용해 일괄적으로 값 할당

  넘파이의 ndarray에 상숫값을 할당하면 모든 ndarrary값에 일괄 적용됨과 동일

```
titanic_df['Age_0'] = 0
titanic_df['Age_0']+100 #update
```

- **drop()**: 데이터(주로 칼럼) 삭제

  DataFrame.drop(`labels=None` , `axis=0` , `index=None, columns=None, level=None, inplace=False` , `errors='raise'`)

```
drop_result = titanic_df.drop(['Age_0', 'Age_10'], axis=1, inplace=TRUE)
```

  cf) `axis=0` row, `axis=1` col

  `inplace` FALSE면 원본 DataFrame의 데이터는 삭제하지 않고 삭제한 DataFrame반환

Index 추출

- **.index**: index 추출
- **.reset_index():**

  새롭게 index를 연속 숫자형으로 할당하고, 기존 index는 새로운 칼럼명으로 추가

  index가 연속된 int형 데이터가 아닐 경우에 쓰임

  Series에 적용하면 DataFrame이 반환(기존 index가 칼럼으로 옮겨져서)

  `drop=TRUE`로 설정하면 기존 index는 칼럼으로 옮기지 않고 삭제됨

Data Selection, Filtering

DataFrame 뒤의 [ ]는 칼럼만 지정할 수 있는 **명칭 지정 연산자**로 이해하는 것이 혼돈을 막아 줌

```
titanic_df['Pclass']
titanic_df[0] #Error
titanic_df[0:2] #처음 2개 데이터(행)을 추출, 그러나 사용하지 않는 것이 좋음
```

명칭(Lable) 기반 인덱싱: 칼럼의 명칭을 기반으로 열 위치 지정

위치(Position) 기반 인덱싱: 0을 출발점으로 하는 가로/세로축 좌표 기반 데이터 지정(input 정수)

- **.ix[ ] 연산(64p)**

  판다스에서 사라질/진 연산

  칼럼 명칭(label)기반 인덱싱, 칼럼 위치(position) 기반 인덱싱 모두 제공

  ```
  titanic_df.ix[0,2] #위치 기반 인덱싱
  titanic_df.ix[0, 'Pclass'] #명칭 기반 인덱싱
  ```

  cf) 인덱스가 integer일 경우 ix[0, 1]은 위치 기반 인덱스로 사용하지 않고 DataFrame의 인덱스값을 사용한다. 인덱스 0이 없다면 오류를 반환한다. (67p)

- **.iloc[ ]**

  위치 기반 인덱싱

  문자열을 행 위치에 입력하면 오류 발생

  불린 인덱싱 제공X

- **.loc[ ]**

  명칭 기반 인덱싱

  cf) 슬라이싱 기호(시작점:종료점)를 적용 시 종료 값-1이 아닌 종료 값까지 포함하여 반환

- **불린 인덱싱**

  개별 조건은( )으로 묶고, 복합 조건 연산자를 사용

```
titanic_df[titanic_df['Age']>60[['Name', 'Age']]]
#조건 필터링 후 두 개 이상의 칼럼을 출력하고 싶을 때 [[ ]]를 사용

#복합 조건 연산
titanic_df[(titanic_df['Age']>60) & (titanic_df['Pclass']==1) &
(titanic_df['Sex']=='female')]
```

정렬, Aggregation 함수, GroupBy 적용

- **.sort_values()**

  주요 입력 파라미터: `by` 특정 칼럼으로 정렬 `ascending=TRUE` 오름차순 `inplace=FALSE`

```
#여러 개의 칼럼으로 정렬하려면 리스트 형식 사용
titanic_sorted = titanic_df.sort_values(by=['Pclass', 'Name'], ascending=False)
```

- **Aggregation 함수**(min, max, sum, count ..)

  DataFrame의 경우 바로 aggregation을 호출할 경우 모든 칼럼에 해당 함수를 적용

  특정 칼럼에 적용하기 위해서는 해당 칼럼만 추출하여 적용

```
taitanic_df[['Age', 'Fare']].mean()
```

- **groupby()**

  파라미터 `by`에 칼럼을 입력하면 해당 칼럼으로 group

  SQL group by와 달리 DataFrame에 groupby()를 호출한 결과에 aggregation함수를 호출하면 groupby() 대상 칼럼을 제외한 모든 칼럼에 함수가 적용됨

  각각 다른 칼럼에 다른 agg를 적용하고 싶을 땐 딕셔너리 형태로 적용

```
titanic_groupby = titanic_df.groupby(by='Pclass')

#count가 Pclass를 제외한 칼럼에 모두 적용
titanic_groupby = titanic_df.groupby('Pclass').count()

#PassengerId, Survived에만 count 적용
titanic_groupby = titanic_df.groupby('Pclass')[['PassengerId', 'Survived']].count()

#agg 적용
titanic_df.groupby('Pclass')['Age'].agg([max, min])

agg_format=['Age':'max', 'SibSp':'sum', 'Fare':'mean']
titanic_df.groupby('Pclass').agg(agg_format)
```

결손 데이터

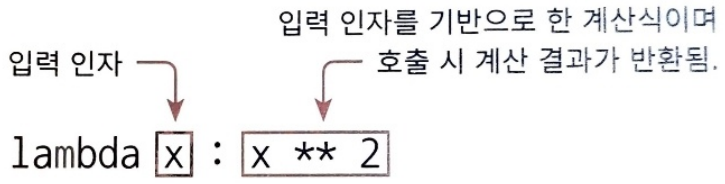
- **isna():** 결손 데이터 TRUE/FALSE로 확인

  결손 데이터의 개수는 isna()결과에 sum() 함수를 추가해 구할 수 있음(0, 1로 변환되어서)

- **fillna()**: 결손 데이터를 다른 값으로 대체할 수 있음 (ex. `.fillna('C000')`)

  inplcae 파라미터를 사용해야 원본 데이터도 변환

Apply lambda 식으로 데이터 가공



lambda: 익명 함수

`map` list 같은 sequence형 데이터를 argument로 받아서 각 원소에 입력 받은 함수를 적용시키고 list로 반환

`apply` map과 다르게 전체 column에 해당 함수를 적용합니다. map이 각 Series 데이터에 적용되는 것만과는 다릅니다. apply는 전체 통계자료를 낼 때 사용할 수 있습니다.

설명 출처

여러 개의 값을 입력 인자로 사용해야 할 경우 `map()` 함수를 결합하여 사용

```
a = [1, 2, 3]
squares = map(lambda x: x**2, a)
list(squares)

'''output
[1, 4, 9]
'''
```

`apply` 에 lambda 식을 적용하여 가공

```
#Name_len이라는 새로운 칼럼 생성
titanic_df['Name_len'] = titanic_df['Name'].apply(lambda x: len(x))

#if else 사용, Child_Adult라는 새로운 칼럼 생성
titanic_df['Child_Adult'] = titanic_df['Age'].apply(lambda x:Child if x<=15 else 'Adult')

#다중 if else
titanic_df['Age_cat'] = titanic_df['Age'].apply(lambda x: 'Child' if x<=15
                                                else('Adult' if x<=60 else 'Elderly'))
```

cf) if절의 경우 **식보다 반환 값을 먼저 기술**해야 함

if, else만 지원하고 나머지는 지원하지 않기 때문에 else절에 ()를 내포하여 다시 if else를 적용