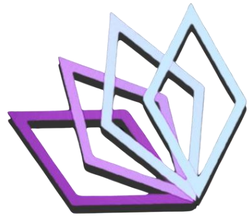
데이터 분석 스터디



2주차

목 차

2주차 복습

```
import numpy as np

# 점수는 0~100 까지

np.random._____(100) # 난수 고정

# A반 | 행 : 학생별(총 3명) / 열 : 과목별(국, 수, 영, 과 순)
Class_A = np.random._____________________________

# B반 | 행 : 과목별(국, 수, 영, 과 순) / 열 : 학생별(총 3명)
Class_B = np.random._____________________________

print(Class_A)
print(Class_B)
```

```
[[ 8 24 67 87]
 [79 48 10 94]
 [52 98 53 66]]
[[ 98  14  34]
 [ 24  15 100]
 [ 60  58  16]
 [  9  93  86]]
```

```
print("A반의 과목별 평균")
print(_____________________)
print('A반에서 가장 잘하는 과목 평균')
print(_____________________________________)
```

```
print("B반의 학생별 평균")
print(_____________________)
print("B반에서 가장 잘하는 학생의 평균")
print(_____________________________________)
```

```
A반의 과목별 평균
[46.33333333 56.66666667 43.33333333 82.33333333]
A반에서 가장 잘하는 과목 평균
82.33333333333333
B반의 학생별 평균
[47.75 45.   59.  ]
B반에서 가장 잘하는 학생의 평균
59.0
```

```
Class_B = _____________
Class_A_B = np._____________________________________

print(Class_A_B)
```

```
[[  8  24  67  87]
 [ 79  48  10  94]
 [ 52  98  53  66]
 [ 98  24  60   9]
 [ 14  15  58  93]
 [ 34 100  16  86]]
```

DATA ANALYSIS

```
print('A반과 B반의 전체 평균')
print(_____________________)

print("A반과 B반 학생들의 각각 점수 총합")
print(_____________________________)

print("A반과 B반의 점수 총합 중 가장 낮은 학생의 점수")
print(_____________________________)

print("A반과 B반의 점수 총합 중 가장 높은 학생의 점수")
print(_____________________________)
```

```
A반과 B반의 전체 평균
53.875
A반과 B반 학생들의 각각 점수 총합
[186 231 269 191 180 236]
A반과 B반의 점수 총합 중 가장 낮은 학생의 점수
180
A반과 B반의 점수 총합 중 가장 높은 학생의 점수
269
```

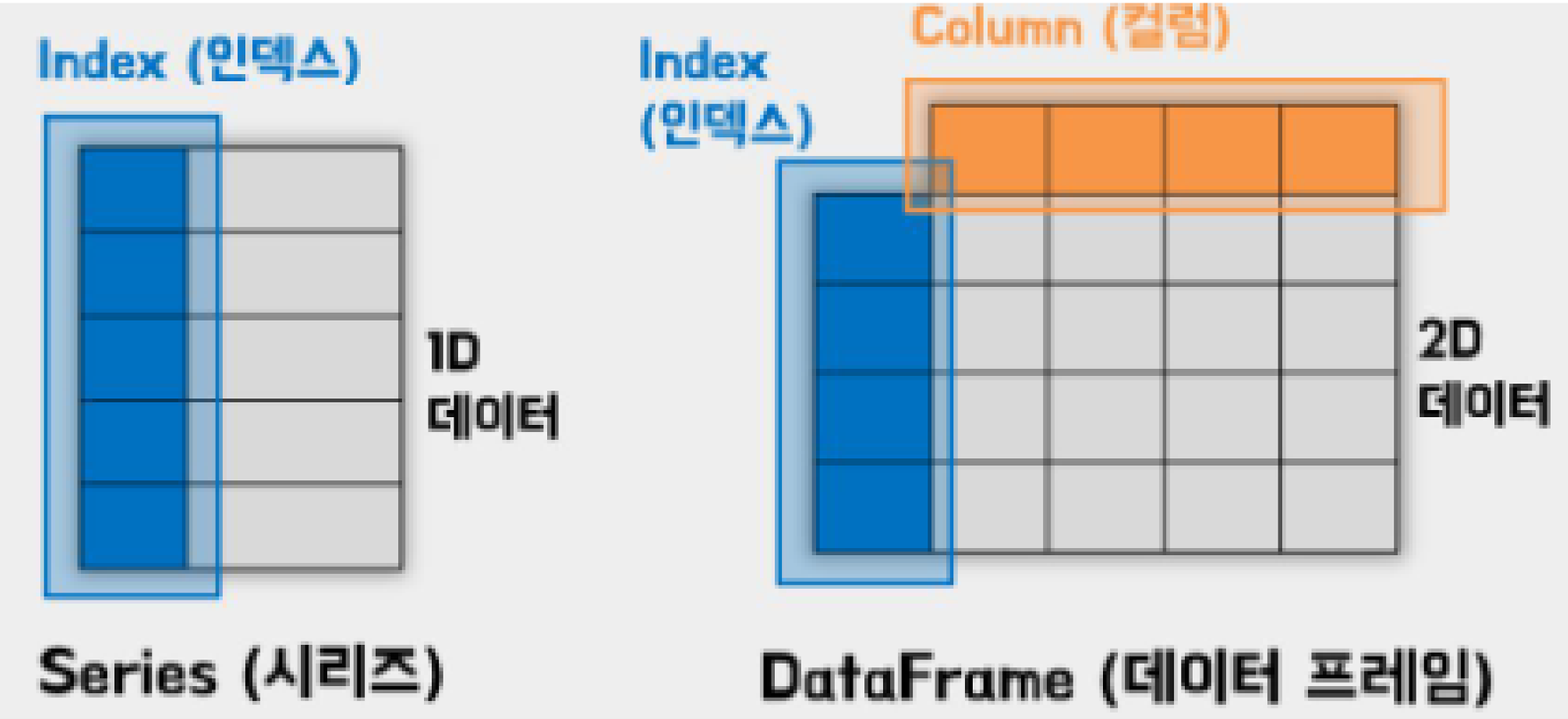
3.0
판다스 인트로

DATA ANALYSIS

판다스 인트로

1. 넘파이 기반이라 연산,실행속도 빠름
2. 데이터프레임을 통한 통계적 수치 계산 용이
3. 결측치 처리도 가능

4. 자료 구조 : 1차원 시리즈 / 2차원 데이터 프레임



3.1
시리즈 생성과 정렬

3.1.1 시리즈 생성과 정렬

pd.Series(data, index, dtype, name)

```
import pandas as pd
s1 = pd.Series([20,21,23])          #리스트로 Series 생성
s2 = pd.Series(('남','여','남'))     #튜플로 Series 생성
s3 = pd.Series({'가':'이순신','나':'이영희','다':'김철수'})   #딕셔너리로 생성
```

0	20
1	21
2	23

0	남
1	여
2	남

가	이순신
나	이영희
다	김철수

시리즈 속성

```
print(s1.dtype)    #시리즈 원소의 자료형
print(s1.shape)    #시리즈의 차원
print(s1.size)     #시리즈 원소의 개수
print(s1.ndim)     #시리즈의 차원(당연히 1차원)
```

3.1.2 시리즈 인덱싱,슬라이싱

넘파이와 동일한 방법

+ 문자형으로 인덱싱 가능

```
house_price_data.head()
```

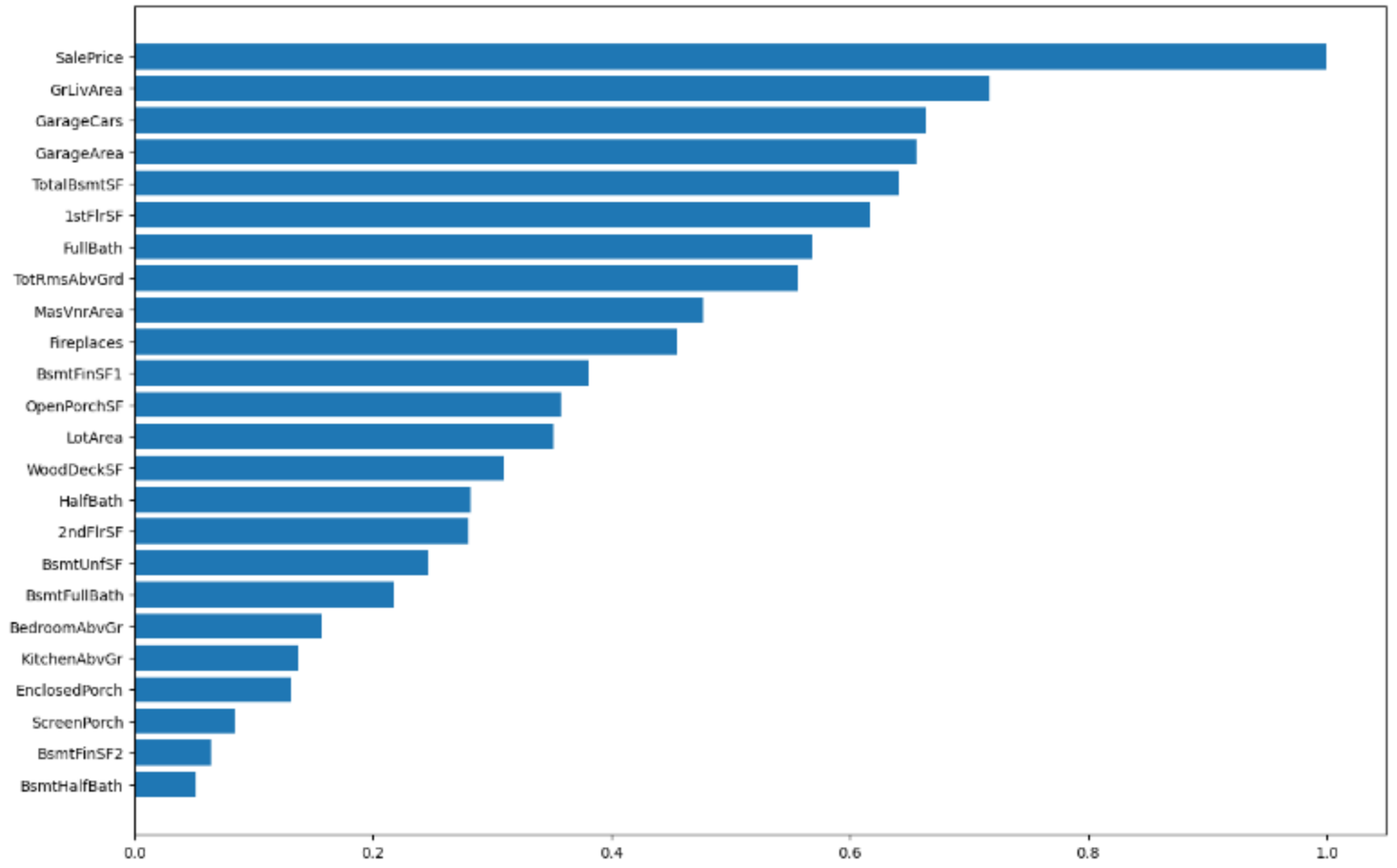
]:

Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	La
1	60	RL	65.0	8450	Pave	NaN	Reg	Lv
2	20	RL	80.0	9600	Pave	NaN	Reg	Lv
3	60	RL	68.0	11250	Pave	NaN	IR1	Lv
4	70	RL	60.0	9550	Pave	NaN	IR1	Lv
5	60	RL	84.0	14260	Pave	NaN	IR1	Lv

5 rows × 80 columns

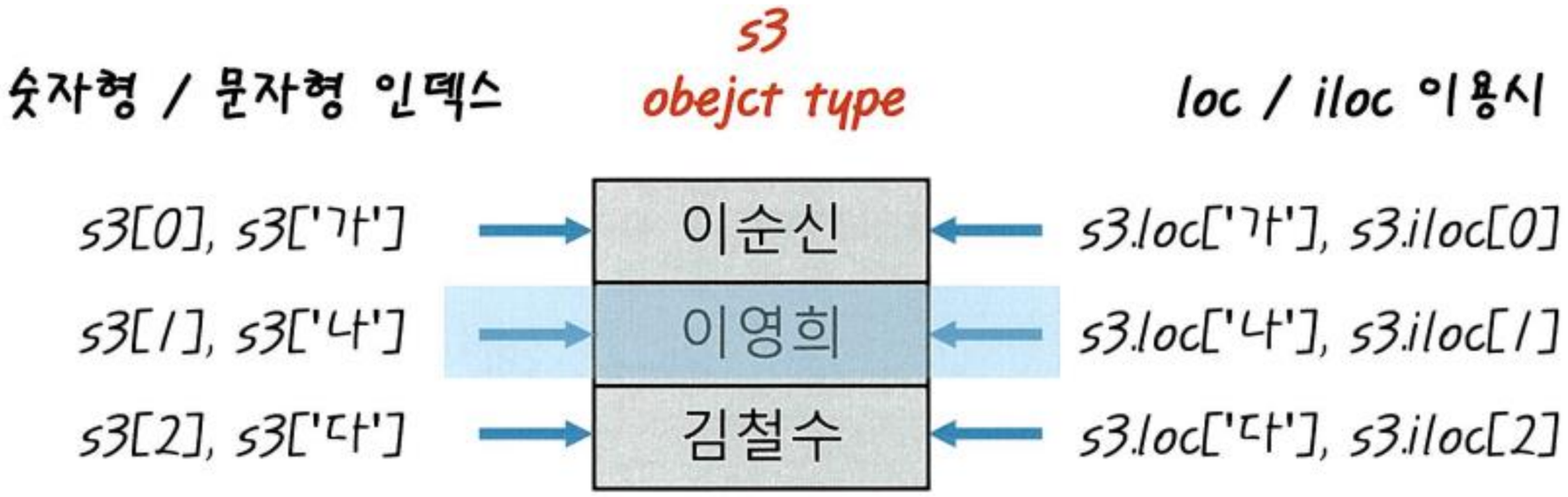
```
house_price_corr = np.abs(house_price_data[Heatmap_column].c
orr()).sort_values("SalePrice",ascending=True)
plt.figure(figsize=(15,10))
plt.barh(y = house_price_corr["SalePrice"].index, width=hous
e_price_corr["SalePrice"])
plt.grid
plt.show()
```

문자형으로 인덱싱해서 데이터의 열들을 시각화한 모습



loc / iloc 인덱서를 통한 데이터 접근

iloc = integer location -> 정수로만 인덱싱 가능



s3.loc['나'] == s3.iloc[1] == s3['나'] == s3[1]=='이영희'

[‘가’ : ‘다’] : 정수형 슬라이싱과 달리 끝 부분 포함

```
print(s1[0:2])          #0번, 1번 인덱스 선택, 2번은 포함되지 않음
print(s2[1:])           #1번 인덱스 후 모두 선택
print(s3['가':'다'])     #'가','나','다' 인덱스 모두 선택 (‘다’ 포함)
print(s3.iloc[0:3])     #0번, 1번, 2번 인덱스 선택   (3 미포함)
print(s3.loc['가':'나']) #‘가’,‘나’ 인덱스 선택
```

인덱스 지정과 이름 추가

```
s_age = pd.Series([20, 21, 23], index = ['이순신', '이영희', '김철수'], name = 'age')
```

3.1.3 시리즈 value 변경,추가

인덱싱과 슬라이싱을 통해 값 변경 가능

없던 인덱스에 접근해서 값 추가 가능

A과자	1.2
B과자	1.5
C과자	2.3
D과자	0.9
E과자	2500.0

```
s_snack.loc['E과자'] = 2.5
s_snack
```

결과		
	A과자	1.2
	B과자	1.5
	C과자	2.3
	D과자	0.9
	E과자	2.5

```
s_snack.loc['F과자'] = 3.5
s_snack
```

결과		
	A과자	1.2
	B과자	1.5
	C과자	2.3
	D과자	0.9
	E과자	2.5
	F과자	3.5

3.1.4 시리즈 index와 value

시리즈.index : 시리즈의 모든 인덱스 반환
시리즈.values : 시리즈의 모든 원소 값 반환

문제1

1에서 45사이의 랜덤한 6개의 숫자를 value로 가지고, 인덱스는 1부터 6까지를 가지는 시리즈를 생성해 보자. 이때 시리즈의 dtype은 int8이 되도록 한다.

```
s_4 = pd.Series(np.random.randint(1, 46, 6),index=range(1,7), dtype = 'int8')
print(s_4)
print(s_4.index)
print(s_4.values)
```

```
1    12
2    34
3    39
4    18
5    14
6    13
dtype: int8
RangeIndex(start=1, stop=7, step=1)
[12 34 39 18 14 13]
```

시리즈의 인덱스 변경

```
s_5 = pd.Series([15, 18, 22, 21.5])
# 기존 인덱스 = [ 0, 1, 2, 3 ]
s_5.index = ['강릉','서울','부산','대구']
```

조건 :

인덱스 수 = 시리즈의 원소 수

3.1.5 시리즈 index 재설정

index와 value의 순서를 같이 변경할 때 reindex( )사용

```
s6 = s5.reindex(['강릉','서울','부산','대구'])
```

강릉	15.0
서울	18.0
부산	22.0
대구	21.5

강릉	15.0
대구	21.5
부산	22.0
서울	18.0

reindex가 기존
시리즈 자체를 변경 X
-> 변경된 결과 저장해야 함

3.1.6 시리즈 정렬

index 기준 정렬
시리즈.sort_index(ascending= ,inplace=)

```
s7.sort_index(ascending=True, inplace=True)
```

1001	강감찬
1003	이순신
1002	권율
1005	김종서
1004	맥아더

1001	강감찬
1002	권율
1003	이순신
1004	맥아더
1005	김종서

ascending :
오름차순 여부

inplace :
결과 저장 여부

value 기준 정렬

시리즈.sort_values(ascending= ,inplace=)

```
s7.sort_values(ascending=True, inplace=True)
```

1001	강감찬
1003	이순신
1002	권율
1005	김종서
1004	맥아더

1001	강감찬
1002	권율
1005	김종서
1004	맥아더
1003	이순신

3.2
시리즈의 주요 메서드

3.2.1 head()와 tail()

시리즈.head(n) : 상단 n개의 데이터 확인

시리즈.tail(n) : 하단 n개의 데이터 확인

```
1  s7.head(3)
2  s7.tail(3)
```

```
1003    이순신
1004    맥아더
1005    김종서
dtype: object
```

```
1005    김종서
1002     권율
1001    강감찬
dtype: object
```

3.2.2 unique().nunique() 그리고 value_counts()

시리즈.unique() : 유니크한 value들 반환
시리즈.nunique() : 유니크한 value의 개수 반환

```
name = ['이순신','이순신','강감찬','권율','김종서','이순신']
name = pd.Series(name)
print('유니크한 values :', name.unique())
print('유니크한 value의 수 :', name.nunique())
```

```
유니크한 values : ['이순신' '강감찬' '권율' '김종서']
유니크한 value의 수 : 4
```

시리즈.value_counts(normalize= , sort= , ascending= , bins= , dropna=)

```
name = ['이순신', '이순신', '강감찬', '권율', '김종서', '이순신', '강감찬']
s_name = pd.Series(name)
s_name.value_counts()
```

```
이순신    3
강감찬    2
권율      1
김종서    1
dtype: int64
```

파라미터 기본 값

normalize = False　　bins = None

sort = True　　dropna = True

ascending = False

normalize : 각 value가 차지하는 비율 알려줌
(전체 비율 = 1)

sort : False로 설정하면 무작위로 출력

ascending : 오름차순, 내림차순 여부 (value 개수 기준)

dropna : 출력 값에 NaN을 포함할 지 여부

bins : 값을 구간화 시켜서 나타냄

normalize 예시

```
s_name.value_counts(ascending=True, normalize=True) * 100
```

결과

```
권율      14.285714
김종서     14.285714
강감찬     28.571429
이순신     42.857143
dtype: float64
```

normalize를 True로 설정하고
출력 값에 100을 곱해서
각 value들이 차지하는 %를 알 수 있다

bins 예시

```
s1 = pd.Series([80, 91, 75, 88, 89, 90, 92, 91, 82, 80])

s1.value_counts(bins = 3)
#출력
> (74.982, 80.667] 3
  (80.667, 86.333] 1
  (86.333, 92.0]   6
```

구간을 3개로 나눠서 출력

```
s1.value_counts(bins = [70, 75, 85, 90, 100])
#출력
> (70, 75]  1
  (75, 85]  3
  (85, 90]  3
  (90, 100] 3
```

구간을 원하는 대로 입력 가능

질의응답

팀별 문제 풀이/해설

문제 교환

마무리

과제 설명

발표 준비 : 2팀!

ch.4 판다스 데이터프레임

9월 30일에 봐요!

2주차 마치겠습니다

수고하셨습니다!