

Numpy와 Pandas

배열 데이터를 효과적으로 다루는 Numpy

NumPy

- NumPy는 파이썬으로 과학 연산을 쉽고 빠르게 할 수 있게 만든 패키지
- NumPy를 이용하면 파이썬의 기본 데이터 형식과 내장 함수를 이용하는 것보다 다차원 배열 데이터를 효과적으로 처리
- NumPy 홈페이지 (<http://www.numpy.org>)

배열 데이터를 효과적으로 다루는 Numpy

배열 생성하기

- 설치된 NumPy를 이용하려면 먼저 NumPy 패키지를 불러 와야 함
- NumPy 패키지를 불러올 때 'import numpy'라고 작성해도 되지만
`'import ~ as ~'` 형식을 이용해 NumPy를 불러옴

```
[ ] import numpy as np
```

- 배열(Array)이란 순서가 있는 같은 종류의 데이터가 저장된 집합을 말함
- NumPy를 이용해 배열을 처리하기 위해서는 우선 NumPy로 배열을 생성

배열 데이터를 효과적으로 다루는 Numpy

시퀀스 데이터로부터 배열 생성

- 시퀀스 데이터(seqdata)로 NumPy의 배열을 생성하는 방법
- `arr_obj = np.array(seq_data)`
- 시퀀스 데이터(seq_data)를 인자로 받아 NumPy의 배열 객체(array object)를 생성
- 시퀀스 데이터(sec_data) 로 리스트와 튜플 타입의 데이터를 모두 사용할 수 있지만 주로 리스트 데이터를 이용

배열 데이터를 효과적으로 다루는 Numpy

시퀀스 데이터로부터 배열 생성

- 리스트로부터 NumPy의 1차원 배열을 생성

```
[ ] import numpy as np

data1 = [0, 1, 2, 3, 4, 5]
a1 = np.array(data1)
a1
```

- 리스트 데이터(data1)를 NumPy의 array() 인자로 넣어서 NumPy 배열을 만들었음

배열 데이터를 효과적으로 다루는 Numpy

시퀀스 데이터로부터 배열 생성

- 정수와 실수가 혼합된 리스트 데이터로 NumPy 배열

```
[ ] data2 = [0.1, 5, 4, 12, 0.5]
    a2 = np.array(data2)
    a2
```

- NumPy에서 인자 로 정수와 실수가 혼합돼 있을 때 모두 실수로 변환
- NumPy 배열의 속성을 표현하려면 'ndarray.속성' 같이 작성
- ndarray는 NumPy의 배열 객체

배열 데이터를 효과적으로 다루는 Numpy

배열 객체의 타입을 확인

- 정수와 실수가 혼합된 리스트 데이터로 NumPy 배열

```
[ ] a1.dtype
```

```
[ ] a2.dtype
```

- a1는 32비트 정수 타입(int32)이고, a2는 64비트 실수 타입(float64)

배열 데이터를 효과적으로 다루는 Numpy

배열 객체의 타입을 확인

- 리스트 data1과 data2를 NumPy array()의 인자로 넣어서 NumPy의 배열
- array()에 리스트 데이터를 직접 넣어서 배열 객체를 생성

```
[ ] np.array([0.5, 2, 0.01, 8])
```

- 1차원 배열을 생성했는데 NumPy는 다차원 배열도 생성

```
[ ] np.array([[1,2,3], [4,5,6], [7,8,9]])
```

- 2차원 배열을 표시
- NumPy에서는 1차원뿐만 아니라 다차원 배열도 생성

배열 데이터를 효과적으로 다루는 Numpy

배열 객체의 타입을 확인

- 리스트 data1과 data2를 NumPy array()의 인자로 넣어서 NumPy의 배열
- array()에 리스트 데이터를 직접 넣어서 배열 객체를 생성

```
[ ] np.array([0.5, 2, 0.01, 8])
```

- 1차원 배열을 생성했는데 NumPy는 다차원 배열도 생성

```
[ ] np.array([[1,2,3], [4,5,6], [7,8,9]])
```

- 2차원 배열을 표시
- NumPy에서는 1차원뿐만 아니라 다차원 배열도 생성

배열 데이터를 효과적으로 다루는 Numpy

범위를 지정해 배열 생성

- 범위를 지정해 NumPy 배열을 생성하는 방법
- NumPy의 `arange()`를 이용해 NumPy 배열을 생성하는 방법
- `arr_obj = np.arange([start,] stop[, step])`
- start부터 시작해서 stop 전까지 step만큼 계속 더해 NumPy의 배열 생성
- step 이 1 인 경우에는 생략할 수 있어서 'numpy.arange(start, stop)'
- start가 0인 경우에는 start도 생략하고 'numpy.arange(stop)'

```
[ ] np.arange(0, 10, 2)
```

```
[ ] np.arange(1, 10)
```

```
[ ] np.arange(5)
```

배열 데이터를 효과적으로 다루는 Numpy

범위를 지정해 배열 생성

- NumPy 배열의 `arange()`를 이용해 생성한 1차원 배열에 `reshape(m,n)`을 추가하면 $m \times n$ 형태의 2차원 배열(행렬)로 변경
- $m \times n$ 행렬의 구조
- NumPy 배열에서 행과 열의 위치는 각각 0부터 시작

```
[ ] np.arange(12).reshape(4,3)
```

- `arange(12)`로 12개의 숫자를 생성한 후 `reshaped,3)`으로 4×3 행렬

배열 데이터를 효과적으로 다루는 Numpy

범위를 지정해 배열 생성

- o `arange()`로 생성되는 배열의 원소 개수와 `reshape(m,n)`의 $m \times n$ 개수는 같아야 함

- o NumPy 배열의 형태를 알기 위해서는 '`ndarray.shape`'를 실행

```
[ ] b1 = np.arange(12).reshape(4,3)
    b1.shape
```

- o `b1`에는 '`reshape(4,3)`'을 통해 만들어진 4×3 행렬이 할당
- o $m \times n$ 행렬(2차원 배열)의 경우에는 '`ndarray.shape`'를 수행하면 (m,n) 이 출력
- o 4×3 행렬인 `b1`의 경우 '`b1.shape`'을 수행하면 $(4, 3)$ 이 출력

배열 데이터를 효과적으로 다루는 Numpy

범위를 지정해 배열 생성

- n개의 요소를 갖는 1차원 배열의 경우에는 'ndarray.shape'를 수행하면 '(n,)' 처럼 표시

```
[ ] b2 = np.arange(5)
    b2.shape
```

- 생성한 변수 b1에는 5개의 요소를 갖는 1차원 배열이 할당돼 있으므로
- 'b2.Shape'을 수행하면 결과로 (5,)가 나옴

배열 데이터를 효과적으로 다루는 Numpy


범위를 지정해 배열 생성

- 범위의 시작과 끝을 지정하고 데이터의 개수를 지정해 NumPy 배열을 생성하는 `linspace()`
- `arr_obj = np.linspace(start, stop[, num])`
- `linspace()`는 start부터 stop까지 num개의 NumPy 배열을 생성
- `num`을 지정하지 않으면 50으로 간주

```
[ ] np.linspace(1, 10, 10)
```


배열 데이터를 효과적으로 다루는 Numpy

특별한 형태의 배열 생성

- 모든 원소가 0 혹은 1인 다차원 배열을 만들기 위해서는 `zeros()`와 `ones()`를 이용
- `arr_zero_n = np.zeros (n)` 
- `arr_zero_mxn = np.zeros((m,n))`
- `arr_one_n = np.ones(n)`
- `arr_one_mxn = np.ones((m/n))`
- `zeros()`는 모든 원소가 0, `ones()`는 모든 원소가 1 인 다차원 배열을 생성

배열 데이터를 효과적으로 다루는 Numpy

기본 연산

- NumPy 배열은 다양하게 연산할 수 있음 
- 배열의 형태(shape)가 같다면 덧셈과 뺄셈, 곱셈과 나눗셈 연산 가능

```
[ ] arr1 = np.array([10, 20, 30, 40])  
    arr2 = np.array([1, 2, 3, 4])
```

- 배열의 형태가 같은 두 개의 1차원 배열을 생성
- 두 배열의 형태가 같다는 의미는 두 배열의 'ndarray.shape'의 결과가 같다는 의미
- 1차원 배열의 경우에는 두 배열의 원소 개수가 같다면 형태가 같은 배열
- 2차원 배열이라면 $m \times n$ 행렬에서 두 행렬의 m 과 n 이 각각 같은 경우

배열 데이터를 효과적으로 다루는 Numpy

통계를 위한 연산

- NumPy에는 배열의 합, 평균, 표준 편차, 분산, 최솟값과 최댓값, 누적 합과 누적 곱 등 주로 통계 에서 많이 이용하는 메서드
- 이 메서드는 각각 `sum()`, `mean()`, `std()`, `var()`, `min()`, `max()`, `cumsum()`, `cumprod()`

배열 데이터를 효과적으로 다루는 Numpy

행렬 연산

- NumPy는 배열의 단순 연산뿐만 아니라 선형 대수(Linear algebra)를 위한 행렬(2차원 배열) 연산도 지원
- 다양한 기능 중 행렬 곱, 전치 행렬, 역행렬, 행렬식을 구하는 방법

```
[ ] A = np.array([0, 1, 2, 3]).reshape(2,2)  
A
```

```
[ ] B = np.array([3, 2, 0, 1]).reshape(2,2)  
B
```

배열 데이터를 효과적으로 다루는 Numpy

배열의 인덱싱과 슬라이싱

- NumPy에서는 배열의 위치, 조건, 범위를 지정해 배열에서 필요한 원소를 선택
- 배열에서 선택된 원소는 값을 가져오거나 변경
- 배열의 위치나 조건을 지정해 배열의 원소를 선택하는 것을 인덱싱 (Indexing)
- 범위를 지정해 배열의 원소를 선택하는 것을 슬라이싱(Slicing)

배열 데이터를 효과적으로 다루는 Numpy

배열의 인덱싱

- 1차원 배열에서 특정 위치의 원소를 선택하려면 원소의 위치를 지정
- 배열명 [위치]
- 배열 원소의 위치는 0부터 시작

배열 데이터를 효과적으로 다루는 Numpy

배열의 인덱싱

- 2차원 배열에서 특정 위치의 원소를 선택하려면 다음과 같이 행과 열의 위치를 지정
- 배열명[행_위치, 열_위치]
- '열_위치' 없이 '배열명[행_위치]'만 입력하면 지정한 행 전체가 선택
- 2차원 배열의 특정 행을 지정해서 행 전체를 변경
- 새로운 값을 입력할 때 'np.array(seq_data)'를 이용해도 되고 리스트를 이용해도 됨

배열 데이터를 효과적으로 다루는 Numpy

배열의 인덱싱

- 2차원 배열의 여러 원소를 선택하기 위해서는 다음과 같이 지정
- 배열명[[행_위치1, 행_위치2, 행_위치n], [열_위치1, 열_위치2, ..., 열_위치n]]
- (행-위치1, 열-위치1)의 원소, (행-위치2, 열-위치2)의 원소, (행-위치n, 열-위치n)의 원소를 가져옴
- 배열에 조건을 지정해 조건을 만족하는 배열을 선택할 수도 있음
- 배열명 [조건]
- 배열에서 조건을 만족하는 원소만 선택

배열 데이터를 효과적으로 다루는 Numpy

배열의 슬라이싱

- 범위를 지정해 배열의 일부분을 선택하는 슬라이싱
- 1차원 배열의 경우 슬라이싱은 다음과 같이 배열의 시작과 끝 위치를 지정
- 배열 [시작_위치:끝_위치]
- 반환되는 원소의 범위는 '시작_위치 ~ 끝_위치-1'가 됨
- '시작_위치'를 지정하지 않으면 _시작_위치 는 0이 되어 범위는 ~ 끝_위치-1'이 됨
- '끝_위치'를 지정하지 않으면 '끝_위치'는 배열의 길이가 되어 범위는 '시작_위치 ~ 배열의 끝'이 됨

배열 데이터를 효과적으로 다루는 Numpy

배열의 슬라이싱

- 2차원 배열(행렬)의 경우 슬라이싱은 다음과 같이 행과 열의 시작과 끝 위치를 지정
- 배열 [행시작위치: 행끝위치, 열시작위치: 열끝위치]
- 원소의 행 범위는 '행시작위치 ~ 행끝위치-1'이 되고, 열 범위는 '열시작위치 ~ 열 끝 위치-1'이 됨
- '행시작위치'나 '열시작위치'를 생략하면 행과 열의 시작 위치는 0
- '행끝위치'를 생략하면 원소의 행 범위는 '행시작_위치' 부터 행의 끝까지 지정되고,
- '열끝위치'를 생략하면 원소의 열 범위는 '열시작_위치' 부터 열의 끝까지 지정

구조적 데이터표시와 처리에 강한 pandas

pandas

- 파이썬을 이용하면 다량의 데이터를 손쉽게 처리할 수 있음
- 파이썬에서 데이터 분석과 처리를 쉽게 할 수 있게 도와주는 것이 바로 pandas 라이브러리
- pandas는 NumPy를 기반으로 만들어졌지만 좀 더 복잡한 데이터 분석에 특화
- NumPy가 같은 데이터 타입의 배열만 처리할 수 있는 데 반해
- pandas는 데이터 타입이 다양하게 섞여 있을 때도 처리할 수 있음
- pandas 홈페이지(<https://pandas.pydata.org>)

구조적 데이터표시와 처리에 강한 pandas

구조적 데이터 생성하기

- Series를 활용한 데이터 생성

```
[ ] import pandas as pd
```

- pandas를 불러오면 이제 pandas를 이용할 때 pandas 대신 pd를 이용할 수 있음
- pandas에서 데이터를 생성하는 가장 기본적인 방법은 **Series()**를 이용하는 것
- Series()**를 이용 하면 Series 형식의 구조적 데이터(라벨을 갖는 1차원 데이터)를 생성

구조적 데이터표시와 처리에 강한 pandas

구조적 데이터 생성하기

- Series()를 이용해 Series 형식의 데이터를 생성하는 방법
- `s = pd.Series(seq_data)`
- Series의 인자로 시퀀스 데이터(seq_data)가 들어감
- 시퀀스 데이터(seq_data)로는 리스트와 튜플 타입의 데이터를 모두 사용할 수 있지만 주로 리스트 데이터를 이용
- 지정하면 인자로 넣은 시퀀스 데이터(seq_data)에 순서를 표시하는 라벨이 자동으로 부여
- Series 데이터에서는 세로축 라벨을 index라고 하고, 입력한 시퀀스 데이터를 values라고 함

구조적 데이터표시와 처리에 강한 pandas

구조적 데이터 생성하기

- Series 데이터의 구조

```
▶ s1 = pd.Series([10, 20, 30, 40, 50])  
s1
```

- Series 데이터를 출력하면 데이터 앞에 index가 함께 표시
- index는 Series 데이터 생성 시 자동으로 만들어진 것으로 데이터를 처리할 때 이용
- Series 데이터는 index와 values를 분리해서 가져올 수 있음
- Series 데이터를 s라고 할 때 index는 's.index'로 values는 's.values'로 가져올 수 있음

구조적 데이터표시와 처리에 강한 pandas

구조적 데이터 생성하기

- NumPy의 경우 배열의 모든 원소가 데이터 타입이 같아야 했지만 pandas의 경우에는 원소의 데이터 타입이 달라도 됨
- Series()로 데이터를 생성할 때 문자와 숫자가 혼합된 리스트를 인자로 이용

```
[ ] s2 = pd.Series(['a', 'b', 'c', 1, 2, 3])  
s2
```

구조적 데이터표시와 처리에 강한 pandas

구조적 데이터 생성하기

- 데이터가 없으면 NumPy를 임포트한 후에 `np.nan`으로 데이터가 없다고 표시할 수도 있음
- `np.nan`를 이용해서 Series 데이터에 특정 원소가 없음을 표시
- NaN은 데이터가 없다는 것을 의미(데이터를 위한 자리(index)는 있지만 실제 값은 없음)
- Series 데이터를 생성할 때 다음과 같이 인자로 index를 추가할 수 있음
- `s = pd.Series(seq_data, index = index_seq)`
- 인자로 index를 명시적으로 입력하면 Series 변수(s)의 index에는 자동 생성되는 index 대신 `index_Seq`가 들어가게 됨
- `index_Seq`도 리스트와 튜플 타입의 데이터를 모두 사용할 수 있지만 주로 리스트 데이터를 이용
- `seq_data`의 항목 개수와 `index_seq`의 항목 개수는 같아야 함

구조적 데이터표시와 처리에 강한 pandas

구조적 데이터 생성하기

- 어느 가게의 날짜별 판매량을 pandas의 Series 형식으로 입력
- 하루는 판매량 데이터가 없어서 np.nan를 입력
- index에는 입력 인자 index의 리스트 데이터가 지정됐음을 확인

```
index_date = ['2018-08-07', '2018-10-08', '2018-10-09', '2018-10-10']
s4 = pd.Series([200, 195, np.nan, 205], index = index_date)
s4
```
- 리스트 형식의 데이터와 index를 따로 입력했지만 파이썬의 딕셔너리를 이용하면 데이터와 index를 함께 입력할 수 있음
- ```
s = pd.Series(dict_data)
```

```
s5 = pd.Series({'국어': 100, '영어': 95, '수학': 90})
s5
```
- 입력 인자로 딕셔너리 데이터를 입력하면 딕셔너리 데이터의 키(keys)와 값(values)이 각각 Series 데이터의 index와 values로 들어감

# 구조적 데이터표시와 처리에 강한 pandas

## 날짜 자동 생성: date\_range

- index에 날짜를 입력할 때 'index = [ '2018-10-07','2018-10-08' 2018-10-09','2018-10-10' ]처럼 문자열을 하나씩 입력
- pandas에서 제공하는 date\_range()
- 원하는 날짜를 자동으로 생성하므로 날짜 데이터를 입력 할 때 편리
- `pd.date_range(start=None, end=None, periods=None, freq='D')`
- start는 시작 날짜 , end는 끝 날짜 , periods는 날짜 데이터 생성 기간 , freq는 날짜 데이터 생성 주기를 나타냄
- start는 반드시 있어야 하며 end나 periods는 둘 중 하나만 있어도 됨
- freq를 입력하지 않으면 'D' 옵션이 설정돼 달력 날짜 기준으로 하루씩 증가



# 구조적 데이터표시와 처리에 강한 pandas

## DataFrame을 활용한 데이터 생성

- Series를 이용해 1차원 데이터를 생성
- pandas에서는 표(Table)와 같은 2차원 데이터 처리를 위해 DataFrame을 제공
- DataFrame은 이름에서 알 수 있듯이 자료(Data)를 담는 틀(Frame)임
- DataFrame을 이용하면 라벨이 있는 2차원 데이터를 생성하고 처리할 수 있음

# 구조적 데이터표시와 처리에 강한 pandas

## DataFrame을 활용한 데이터 생성

- DataFrame을 이용해 데이터를 생성하는 방법
- `df = pd.DataFrame(data [, index = index_data, columns = columns_data])`
- DataFrame()의 인자인 data에는 리스트와 형태가 유사한 데이터 타입은 모두 사용할 수 있음
- 리스트와 딕셔너리 타입의 데이터, NumPy의 배열 데이터, Series나 DataFrame 타입의 데이터를 입력 할 수 있음

# 구조적 데이터표시와 처리에 강한 pandas

## DataFrame을 활용한 데이터 생성

- DataFrame의 세로축 라벨을 index라고 하고 가로축 라벨을 columns
- index와 columns를 제외한 부분을 values라고 함
- index와 columns에는 1차원 배열과 유사한 데이터 타입(리스트, NumPy의 배열 데이터 , Series 데이터 등)의 데이터를 입력할 수 있음
- DataFrame의 data의 행 개수와 index 요소의 개수, data 열의 개수와 columns 요소의 개수가 일치해야 한다는 것
- index와 columns는 선택 사항이므로 입력하지 않을 수 있는데 그러면 index 와 columns에는 자동으로 0부터 숫자가 생성되어 채워짐

# 구조적 데이터표시와 처리에 강한 pandas

## DataFrame을 활용한 데이터 생성

- 리스트를 이용해 DataFrame의 데이터를 생성

```
[] import pandas as pd

 pd.DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

- values 부분에는 입력한 data가 순서대로 입력돼 있고
- 가장 좌측의 열과 가장 윗줄의 행에는 각각 숫자가 자동으로 생성되어 index와 columns를 구성
- index와 columns를 입력하지 않더라도 자동으로 index와 columns가 생성됨

# 구조적 데이터표시와 처리에 강한 pandas

## DataFrame을 활용한 데이터 생성

- NumPy의 배열 데이터를 입력해 생성한 DataFrame 데이터

```
[] import numpy as np
import pandas as pd

data_list = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
pd.DataFrame(data_list)
```

# 구조적 데이터표시와 처리에 강한 pandas

## DataFrame을 활용한 데이터 생성

- data 뿐만 아니라 index와 columns도 지정

```
[] import numpy as np
import pandas as pd

data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
index_date = pd.date_range('2019-09-01', periods=4)
columns_list = ['A', 'B', 'C']
pd.DataFrame(data, index=index_date, columns=columns_list)
```

- index와 columns에 지정한 값이 들어간 것
- index에는 date\_range()로 생성한 날짜를, columns에는 리스트 데이터를 입력

# 구조적 데이터표시와 처리에 강한 pandas

## DataFrame을 활용한 데이터 생성

- 딕셔너리 타입으로 2차원 데이터를 입력

```
table_data = { '연도': [2015, 2016, 2016, 2017, 2017],
 '지사': ['한국', '한국', '미국', '한국', '미국'],
 '고객 수': [200, 250, 450, 300, 500]}
```

table\_data

```
pd.DataFrame(table_data)
```

|   | 연도   | 지사 | 고객 수 |
|---|------|----|------|
| 0 | 2015 | 한국 | 200  |
| 1 | 2016 | 한국 | 250  |
| 2 | 2016 | 미국 | 450  |
| 3 | 2017 | 한국 | 300  |
| 4 | 2017 | 미국 | 500  |



# 구조적 데이터표시와 처리에 강한 pandas

## DataFrame을 활용한 데이터 생성

- index는 자동으로 생성됐고 딕셔너리 데이터의 키(keys)는 DataFrame에서 columns로 지정
- 표에서 각 열의 제목(고객 수, 연도,지사)처럼 들어간 것
- DataFrame 데이터의 열은 입력한 데이터의 순서대로 생성되지 않았음
- 딕셔너리 데이터가 키(keys)에 따라서 자동으로 정렬됐기 때문
- 한글은 가나다순으로 영어는 알파벳순으로 정렬



# 구조적 데이터표시와 처리에 강한 pandas

## DataFrame을 활용한 데이터 생성

- 데이터의 정렬 순서는 다음과 같이 'columns = columns\_list'를 이용해 키의 순서를 지정할 수 있음

```
df = pd.DataFrame(table_data, columns=['연도', '지사', '고객 수'])
df
```

- DataFrame 데이터에서 index, columns, values는
- 각각 DataFrame\_data.index, DataFrame\_data.columns, DataFrame\_data.values로 확인할 수 있음

```
df.index
```

```
df.columns
```

```
df.values
```

# 구조적 데이터표시와 처리에 강한 pandas

## 데이터 연산

- pandas의 Series()와 DataFrame()으로 생성한 데이터끼리는 사칙 연산을 할 수 있음
- 파이썬의 리스트와 NumPy의 배열과 달리 pandas의 데이터끼리는 서로 크기가 달라도 연산할 수 있음
- 연산을 할 수 있는 항목만 연산을 수행
- Series 데이터의 경우와 마찬가지로 DataFrame 데이터의 경우
  - 연산할 수 있는 항목끼리만 연산하고 그렇지 못한 항목은 NaN으로 표시

# 구조적 데이터표시와 처리에 강한 pandas

## pandas에는 데이터의 통계 분석

- pandas에는 데이터의 통계 분석을 위한 다양한 메서드가 있어서 데이터의 총합, 평균, 표준 편차 등을 쉽게 구할 수 있음
- pandas의 메서드로 통계 분석하는 방법
- 2012년부터 2016년까지 우리나라의 계절별 강수량(단위 mm)

```
[] table_data3 = {'봄': [256.5, 264.3, 215.9, 223.2, 312.8],
 '여름': [770.6, 567.5, 599.8, 387.1, 446.2],
 '가을': [363.5, 231.2, 293.1, 247.7, 381.6],
 '겨울': [139.3, 59.9, 76.9, 109.1, 108.1]}
columns_list = ['봄', '여름', '가을', '겨울']
index_list = ['2012', '2013', '2014', '2015', '2016']

df3 = pd.DataFrame(table_data3, columns = columns_list, index = index_list)
df3
```

# 구조적 데이터표시와 처리에 강한 pandas

## 데이터를 원하는 대로 선택하기

- o pandas의 DataFrame 데이터를 원본 훼손 없이 원하는 부분만 선택하는 방법

```
[] import pandas as pd
import numpy as np

KTX_data = { '경부선 KTX': [39060, 39896, 42005, 43621, 41702, 41266, 32427],
 '호남선 KTX': [7313, 6967, 6873, 6626, 8675, 10622, 9228],
 '경전선 KTX': [3627, 4168, 4088, 4424, 4606, 4984, 5570],
 '전라선 KTX': [309, 1771, 1954, 2244, 3146, 3945, 5766],
 '동해선 KTX': [np.nan, np.nan, np.nan, np.nan, 2395, 3786, 6667]}
col_list = ['경부선 KTX', '호남선 KTX', '경전선 KTX', '전라선 KTX', '동해선 KTX']
index_list = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']

df_KTX = pd.DataFrame(KTX_data, columns = col_list, index = index_list)
df_KTX
```

# 구조적 데이터표시와 처리에 강한 pandas

## 데이터를 원하는 대로 선택하기

- pandas에서는 다음과 같이 `head()`와 `tail()`을 이용해 DataFrame의 전체 데이터 중 처음 일부 분과 끝 일부분만 반환
  - `DataFrame_data.head([n])`
  - `DataFrame_data.tail([n])`
- 인자 `n`을 지정하면 `head(n)`의 경우에는 처음 `n`개의 행의 데이터를 반환
- `tail(n)`의 경우에는 마지막 `n`개의 행 데이터를 반환
- 인자 `n`을 지정하지 않으면 기본값으로 5가 지정

# 구조적 데이터표시와 처리에 강한 pandas

## 데이터를 원하는 대로 선택하기

- DataFrame 데이터에서 연속된 구간의 행 데이터를 선택하려면 다음과 같이 '행 시작 위치'와 '끝 위치'를 지정
- DataFrame\_data [ 행시작\_위치: 행끝\_위치 ]
- DataFrame 데이터(DataFrame\_data) 중에서 '행 시작 위치 ~ 행 끝 위치-1'까지의 행 데이터를 반환
- 행의 위치는 0부터 시작

# 구조적 데이터표시와 처리에 강한 pandas

## 데이터를 원하는 대로 선택하기

- DataFrame 데이터를 생성할 때 index를 지정했다면 다음과 같이 index 항목 이름을 지정해 행을 선택 할 수도 있음
- `DataFrame_data.loc[index_name]`
- `DataFrame_data`의 데이터에서 index가 `index_name`인 행 데이터를 반환
- `df_KTX`에서 index로 지정한 항목 중 2011 년 데이터만 선택

```
df_KTX.loc['2011']
```

# 구조적 데이터표시와 처리에 강한 pandas

## 데이터를 원하는 대로 선택하기

- DataFrame 데이터에서 다음과 같이 index 항목 이름으로 구간을 지정해서 연속된 구간의 행을 선택할 수도 있음
- `DataFrame_data.loc[start_index_name : end_index_name]`
- DataFrame\_data의 데이터 중 index가 start\_index\_name에서 end\_index\_name까지 구간의 행 데이터가 선택
- df\_KTX에서 index로 지정한 항목 중 2013년부터 2016년까지의 행 데이터를 선택  

```
df_KTX.loc['2013': '2016']
```



# 구조적 데이터표시와 처리에 강한 pandas

## 데이터를 원하는 대로 선택하기

- 데이터에서 하나의 열만 선택하려면 다음과 같이 하나의 columns 항목 이름을 지정
- `DataFrame_data[column_name]`
- `DataFrame_data`에서 `column_name`으로 지정한 열이 선택
- `df_KTX`에서 `columns`의 항목 중 '경부선 KTX'를 지정해 하나의 열 데이터만 선택

```
df_KTX['경부선 KTX']
```

# 구조적 데이터표시와 처리에 강한 pandas

## 데이터를 원하는 대로 선택하기

- DataFrame 데이터에서 하나의 열을 선택한 후 index의 범위를 지정해 원하는 데이터만 선택할 수도 있음
- `DataFrame_data[column_name][start_index_name : end_index_name]`
- `DataFrame_data[column_name][start_index_pos : end_index_pos]`
- column\_name으로 하나의 열을 선택한 후 'start\_index\_name : end\_index\_name'
- index의 이름을 지정해 index의 범위를 선택
- 'start\_index\_pos:end\_index\_pos'로 index의 위치를 지정해 index의 범위를 선택
- index의 위치는 0부터 시작

```
df_KTX['경부선 KTX']['2012':'2014']
```

```
df_KTX.loc['2011']
```

# 구조적 데이터표시와 처리에 강한 pandas

## 데이터를 원하는 대로 선택하기

- DataFrame 데이터 중 하나의 원소만 선택
- DataFrame\_data.loc[index\_name][column\_name]
- DataFrame\_data.loc[index\_name, column\_name]
- DataFrame\_data[column\_name][index\_name]
- DataFrame\_data[column\_name][index\_pos]
- DataFrame\_data[column\_name].loc[index\_name]

```
df_KTX.loc['2016']['호남선 KTX']
```

```
df_KTX.loc['2016', '호남선 KTX']
```

```
df_KTX['호남선 KTX']['2016']
```

```
df_KTX['호남선 KTX'][5]
```

```
df_KTX['호남선 KTX'].loc['2016']
```

# 구조적 데이터표시와 처리에 강한 pandas

---

## 데이터를 원하는 대로 선택하기

- DataFrame 데이터의 행과 열을 바꾸는 방법
- 행렬에서 행과 열을 바꾸는 것을 전치(transpose)
- pandas에서는 다음과 같은 방법으로 DataFrame 데이터의 전치
- DataFrame\_data.T

# 구조적 데이터표시와 처리에 강한 pandas

---

## 데이터 통합하기

- 두 개의 데이터를 하나로 통합하는 방법
- 통합 방법에는 세로로 증가하는 방향으로 통합하기,
- 가로로 증가하는 방향으로 통합하기,
- 특정 열을 기준으로 통합하는 방법

# 구조적 데이터표시와 처리에 강한 pandas

## 세로 방향으로 통합하기

- DataFrame에서 columns가 같은 두 데이터를 세로 방향(index 증가 방향)으로 합
- 'append()' 를 이용
- DataFrame\_data1.append(DataFrame\_data2 [,ignore\_index=True])
- 세로 방향으로 DataFrame\_data1 다음에 DataFrame\_data2가 추가돼서 DataFrame 데이터로 반환
- 'ignore\_index=True'를 입력하지 않으면 생성된 DataFrame 데이터에는 기존의 데이터의 index가 그대로 유지
- 'ignore\_index=True'를 입력하면 생성된 DataFrame 데이터에는 데이터 순서대로 새로운 index가 할당

# 구조적 데이터표시와 처리에 강한 pandas

## 가로 방향으로 통합하기

- columns가 같은 두 DataFrame 데이터에 대해 세로 방향(index 증가 방향)으로 데이터를 추가하는 방법
- index가 같은 두 DataFrame 데이터에 대해 가로 방향(columns 증가 방향)에 새로운 데이터를 추가하는 방법
- 두 개의 DataFrame 데이터를 가로 방향으로 합치려면 'join()' 을 이용
- `dataFrame_data1.join(DataFrame_data2)`
- DataFrame\_data1 다음에 가로 방향으로 DataFrame\_data2가 추가되어 DataFrame 데이터로 반환

# 구조적 데이터표시와 처리에 강한 pandas

## 가로 방향으로 통합하기

- index 라벨을 지정한 DataFrame의 데이터의 경우에도 index가 같으면 'join()'을 이용하여 가로 방향으로 데이터를 추가 할 수 있음
- index의 크기가 다른 DataFrame 데이터를 'join()'을 이용해 추가한다면 데이터가 없는 부분은 NaN으로 채워짐
- index의 크기가 작은 DataFrame 데이터에서 원소가 없는 부분은 NaN으로 채워짐



# 구조적 데이터표시와 처리에 강한 pandas

## 특정 열을 기준으로 통합하기

- 두 개의 DataFrame 데이터를 특정 열을 기준으로 통합
- 특정 열을 키(key)라고 함
- 두 개의 DataFrame 데이터에 공통된 열이 있다면 이 열을 기준으로 두 데이터를 다음 과 같은 방법으로 통합
- `DataFrame_left_data.merge(DataFrame_right_data)`
- 왼쪽 데이터(`DataFrame_left_data`)와 오른쪽 데이터 (`DataFrame_right_data`)가 공통된 열 (key)을 중심으로 좌우로 통합

# 구조적 데이터표시와 처리에 강한 pandas

## 특정 열을 기준으로 통합하기

- 특정 열을 기준으로 두 DataFrame 데이터가 모두 값을 갖고 있을 때 특정 열을 기준으로 통합
- 두 개의 DataFrame 데이터가 특정 열을 기준으로 일부만 공통된 값을 갖는 경우에 통합
- 'merge()'에 선택 인자를 지정
- `DataFrame_left_data.merge(DataFrame_right_data, how=merge_method, on=key_label)`
  - on 인자에는 통합하려는 기준이 되는 특정 열(key)의 라벨 이름(key\_label)을 입력
  - on 인자를 입력하지 않으면 자동으로 두 데이터에서 공통적으로 포함된 열이 선택
  - how 인자에는 지정된 특정 열(key)을 기준으로 통합 방법(merge\_method)을 지정

# 구조적 데이터표시와 처리에 강한 pandas

## 특정 열을 기준으로 통합하기

- on 인자의 값은 key로 지정하고 how 인자의 값을 각각 left, right, outer, inner로 변경
- how 인자의 값에 따라 통합 결과가 달라지는 것을 볼 수 있음
- 해당 항목에 데이터가 없는 경우는 NaN이 자동으로 입력

| how 선택 인자 | 설명                                            |
|-----------|-----------------------------------------------|
| left      | 왼쪽 데이터는 모두 선택하고 지정된 열(key)에 값이 있는 오른쪽 데이터를 선택 |
| right     | 오른쪽 데이터는 모두 선택하고 지정된 열(key)에 값이 있는 왼쪽 데이터를 선택 |
| outer     | 지정된 열(key)을 기준으로 왼쪽과 오른쪽 데이터를 모두 선택           |
| inner     | 지정된 열(key)을 기준으로 왼쪽과 오른쪽 데이터 중 공통 항목만 선택(기본값) |

# 구조적 데이터표시와 처리에 강한 pandas

---

## 데이터 파일을 읽고 쓰기

- pandas는 표 형식의 데이터 파일을 DataFrame 형식의 데이터로 읽어오는 방법
- DataFrame 형식의 데이터를 표 형식으로 파일로 저장하는 편리한 방법

# 구조적 데이터표시와 처리에 강한 pandas

## 표 형식의 데이터 파일을 읽기

- `read_csv()`를 이용해 표 형식의 텍스트 데이터 파일을 읽는 방법
- `read_csv()`는 기본적으로 각 데이터 필드가 콤마(,)로 구분된 CSV(comma-separated values) 파일을 읽는데 이용
- 옵션을 지정하면 각 데이터 필드가 콤마 외의 다른 구분자로 구분돼 있어도 데이터를 읽어올 수 있음

# 구조적 데이터표시와 처리에 강한 pandas

## 표 형식의 데이터 파일을 읽기

- read\_csv() 이용하는 방법
- DataFrame\_data = pd.read\_csv(file\_name [, options])
- filename은 텍스트 파일의 이름으로 경로를 포함
- options는 선택 사항

```
[] %%writefile sea_rain1.csv
연도, 동해, 남해, 서해, 전체
1996, 17.4629, 17.2288, 14.436, 15.9067
1997, 17.4116, 17.4092, 14.8248, 16.1526
1998, 17.5944, 18.011, 15.2512, 16.6044
1999, 18.1495, 18.3175, 14.8979, 16.6284
2000, 17.9288, 18.1766, 15.0504, 16.6178
```

# 구조적 데이터표시와 처리에 강한 pandas

## 표 형식의 데이터 파일을 읽기

- o pandas의 read\_csv()로 위의 csv 파일을 읽어옴

```
[] import pandas as pd
```

```
pd.read_csv('sea_rain1.csv')
```

```
[] pd.read_csv('sea_rain1_from_notepad.csv', encoding = "cp949")
```

# 구조적 데이터표시와 처리에 강한 pandas

## 표 형식의 데이터 파일을 읽기

- 데이터 파일처럼 공백(빈칸)으로 구분

```
%%writefile sea_rain1_space.txt
연도 동해 남해 서해 전체
1996 17.4629 17.2288 14.436 15.9067
1997 17.4116 17.4092 14.8248 16.1526
1998 17.5944 18.011 15.2512 16.6044
1999 18.1495 18.3175 14.8979 16.6284
2000 17.9288 18.1766 15.0504 16.6178
```

 Writing sea\_rain1\_space.txt

```
[] pd.read_csv('sea_rain1_space.txt', sep=" ")
```



# 구조적 데이터표시와 처리에 강한 pandas

## 표 형식의 데이터를 파일로 쓰기

- pandas에서 제공하는 `to_csv()`를 이용해 DataFrame 형식의 데이터를 텍스트 파일로 저장하는 방법
- `DataFrame_data.to_csv(file_name [, options])`
- `file_name`은 텍스트 파일 이름으로 경로를 포함
- 선택사항인 `options`에는 구분자와 문자의 인코딩 방식 등을 지정할 수 있는데 지정하지 않으면 구분자는 콤마가 되고 문자의 인코딩 방식은

'utf-8'

```
[] file_name = 'save_DataFrame_cp949.txt'
 df_pr.to_csv(file_name, sep=" ", encoding = "cp949")
```

# 구조적 데이터표시와 처리에 강한 pandas

## 표 형식의 데이터를 파일로 쓰기

- DataFrame 데이터를 파일로 저장하기 위해
- 네 명의 몸무게(Weight, 단위: kg) 와 키(Height, 단위: cm) 데이터를 DataFrame 형식으로 생성
- DataFrame 데이터에서 index에 이름을 추가하고 싶으면  
'df.index.name=문자열'과 같이 작성

```
df_WH = pd.DataFrame({'Weight': [62, 67, 55, 74],
 'Height': [165, 177, 160, 180]},
 index=['ID_1', 'ID_2', 'ID_3', 'ID_4'])
df_WH.index.name = 'User'
df_WH
```

# 구조적 데이터표시와 처리에 강한 pandas

## 표 형식의 데이터를 파일로 쓰기

- 파일로 저장하기 전에 몸무게와 키를 이용해 체질량 지수(BMI)를 구해서 dt\_WH에 추가
- df\_WH의 몸무게와 키 데이터를 이용해 체질량 지수(BMI)를 구함
- 키의 경우 입력한 데이터는 cm 단위여서 m 단위로 변경하기 위해 100으로 나눔
- DataFrame 데이터(df)에 'df['column\_name']=columnn\_data'로 새로운 열 데이터를 추가
- 체질량 지수(BMI)를 df\_WH에 추가
- DataFrame 데이터 df\_WH를 csv 파일로 저장

# 정리

---

## 정리

- 데이터 분석에 아주 유용하게 활용할 수 있는 NumPy와 pandas
- 파이썬 프로그램을 작성할 때는 파이썬의 기본 기능과 외부 패키지나 라이브러리를 함께 이용
- Python Package Index 홈페이지(<https://pypi.org>)