11장 데이터 분석을 위한 패키지

11.1 배열 데이터를 효과적으로 다루는 NumPy

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

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

시퀀스 데이터로부터 배열 생성

- 시퀀스 데이터(seqjata)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

- 리스트로부터 NumPy의 1차원 배열을 생성

```
import numpy as np

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

```
array([0, 1, 2, 3, 4, 5])
```

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

- 정수와 실수가 혼합된 리스트 데이터로 NumPy 배열

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

```
array([ 0.1,  5. ,  4. , 12. ,  0.5])
```

- NumPy에서 인자 로 정수와 실수가 혼합돼 있을 때 모두 실수로 변환

- NumPy 배열의 속성을 표현하려면 'ndarray.속성' 같이 작성
- ndarray는 NumPy의 배열 객체

- 배열 객체의 타입을 확인

```
a1.dtype
```

```
dtype('int64')
```

```
a2.dtype
```

```
dtype('float64')
```

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

- 리스트 data1과 data2를 NumPy array()의 인자로 넣어서 NumPy의 배열

- anray()에 리스트 데이터를 직접 넣어서 배열 객체를 생성

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

```
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]])
```

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

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

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 배열을 생성
- step 이 1 인 경우에는 생략할 수 있어서 'numpy.arange(start, stop)'처럼
- start가 0인 경우에는 start도 생략하고 'numpy.arange(stop)'처럼 시용

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

```
array([0, 2, 4, 6, 8])
```

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

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

```
np.arange(5)
```

```
array([0, 1, 2, 3, 4])
```

- NumPy 배열의 arange()를 이용해 생성한 1차원 배열에 '.reshape(m,n)'을 추가하면 m x n 형태의 2차원 배열(행렬)로 변경

- m x n 행렬의 구조
- NumPy 배열에서 행과 열의 위치는 각각 0부터 시작

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

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

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

- arange()로 생성되는 배열의 원소 개수와 reshape(m,n)의 m x n 개수는 같아야 함

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

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

```
(4, 3)
```

- b1에는 'reshape(4,3)'을 통해 만들어진 4 x 3 행렬이 할당
- m x n 행렬(2차원 배 열)의 경우에는 'ndarray.shape'를 수행하면 (m,n)이 출력

- 4x3 행렬인 b1의 경우 'b1.shape'을 수행하면 (4, 3)이 출력

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

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

```
(5,)
```

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

- 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
- num을 지정하지 않으면 50으로 간주

- 1부터 10까지 10개의 데이터를 생성

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

```
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

- 지정된 범위(1~10)에 해당하는 10개의 데이터가 잘 생성

- 0부터 파이까지 동일한 간격으로 나눈 20개의 데이터를 생성

```
np.linspace(0, np.pi, 20 )
```

```
array([0.        , 0.16534698, 0.33069396, 0.49604095, 0.66138793,
       0.82673491, 0.99208189, 1.15742887, 1.32277585, 1.48812284,
       1.65346982, 1.8188168 , 1.98416378, 2.14951076, 2.31485774,
       2.48020473, 2.64555171, 2.81089869, 2.97624567, 3.14159265])
```

- 사용한 'np.pi'는 NumPy메서 파이를 입력할 때 이용

특별한 형태의 배열 생성

- 특별한 형태의 배열을 생성하는 방법

- 모든 원소가 0 혹은 1인 다차 원 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

- n개의 원소가 모두 0을 갖는 1차원 배열을 만들려면 np.zeros(n)
- 모든 원소가 0인 m x n 형태의 2차원 배열(행렬)을 생성하려면 np.zeros((m, n))을 이용

- nes()도 마찬가지로, np.ones(n)과 np.ones((m,n))을 이용

- zero() 함수로 원소의 개수가 10개(n=10)인 1차원 배열을 생성

```
np.zeros(10)
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

- zero()를 이용해 3 x 4의 2차원 배열을 생성

```
np.zeros((3,4))
```

```
array([[0., 0., 0., 0.],
```

```
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

- ones()으로 1차원 배열과 2차원 배열을 생성

```
np.ones(5)
```

```
array([1., 1., 1., 1., 1.])
```

```
np.ones((3,5))
```

```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

- 단위행렬(Identity matrix)을 생성하는 방법

- 단위행렬은 n x n인 정사 형 행렬에서 주 대각선이 모두 1 이고 나머지는 0인 행렬을 의미
- arr_I = np.eye(n)
- n x n의 단위행렬을 갖는 2차원 배열 객체를 생성

3 X 3 단위행렬을 생성

```
np.eye(3)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

배열의 데이터 타입 변환

- NumPy의 배열은 숫자뿐만 아니라 문자열도 원소로 가질 수 있음

- 문자열이 원소인 NumPy 배열을 생성

```
np.array(['1.5', '0.62', '2', '3.14', '3.141592'])
```

```
array(['1.5', '0.62', '2', '3.14', '3.141592'], dtype='<U8')
```

- dtype은 NumPy 데이터의 형식(타입)을 표시
- 'rdtype= <U8'의 의미는 NumPy 데이터의 형식이 유니코드이며 문자의 수는 최대 8개라는 것

- NumPy 배열이 문자열(숫자 표시)로 돼 있다면 연산을 위해서는 문자열을 숫자{정수나 실수)로 변환

- NumPy 배열의 형 변환은 astype()으로 할 수 있음
- num_arr = str_arr.astype (dtype)

- NumPy 배열의 모든 요소를 지정한 데이터 타입(dtype)으로 변환

- 문자열로 구성된 NumPy 배열의 모든 원소를 정수나 실수로 변환하거나, 정수를 실수로 혹은 실수를 정수로 변환
- 정수나 실수를 문자열로 변환할 수도 있음
- 정수로 바꾸려면 dtype에 int를 입력하고, 실수로 바꾸려면 float를 입력하며, 문자열로 바꾸려면 str을 입력

- 실수가 입력된 문자열을 원소로 갖는 NumPy 배열을 실수 타입으로 변환

```
str_a1 = np.array(['1.567', '0.123', '5.123', '9', '8'])
num_a1 = str_a1.astype(float)
num_a1
```

```
array([1.567, 0.123, 5.123, 9.   , 8.   ])
```

- 'nanray.dtype'을 이용해 NumPy의 데이터 타입을 확인
- 변환 전후의 NumPy 배열의 데이터 타입을 확인

```
str_a1.dtype
```

```
dtype('<U5')
```

```
num_a1.dtype
```

```
dtype('float64')
```

- 변수 str_a1 은 dtype이 유니코드(U)였는데 변수 num_a1 은 dtype이 실수(float)로 변경된 것

- 정수를 문자열 원소로 갖는 NumPy 배열을 정수로 변환

```
str_a2 = np.array(['1', '3', '5', '7', '9'])
num_a2 = str_a2.astype(int)
```

```
num_a2
```

```
array([1, 3, 5, 7, 9])
```

- 'ndarray.dtype'으로 NumPy 배열의 데이터 타입을 확인

```
str_a2.dtype
```

```
dtype('<U1')
```

```
num_a2.dtype
```

```
dtype('int64')
```

- 배열의 데이터 타입이 유니코드(U)에서 정수 타입(int)으로 변경된 것을 알 수 있음

- 실수를 원소로 갖는 NumPy 배열을 정수로 변환

```
num_f1 = np.array([10, 21, 0.549, 4.75, 5.98])
num_i1 = num_f1.astype(int)
num_i1
```

```
array([10, 21,  0,  4,  5])
```

- 변환 전후의 NumPy 배열의 데이터 타입을 확인

```
num_f1.dtype
```

```
dtype('float64')
```

```
num_i1.dtype
```

```
dtype('int64')
```

- NumPy 배열의 데이터 타입이 실수(float)에서 정수 타입(int)으로 바뀐 것을 볼 수 있음

난수 배열의 생성

- random 모듈을 이용해 임의의 숫자인 난수(random number)를 생성
- NumPy에서도 난수를 발생시킬 수 있는 다양한 함수

- 그중 rand() 함수를 이용하면 실수 난수를 요소로 갖는 NumPy 배열을 생성할 수 있으며,
- randint() 함수를 이용하면 정수 난수를 요소로 갖는 NumPy 배열을 생성

- rand_num = np.random.rand([d0, d1,... dn])
- rand_num = np.random.randint([low,] high [,size])

- rand() 함수는 (0,1)의 실수 난수를 갖는 NumPy 배열을 생성
- (a, b)의 표현은 값의 범위가 a 이상이고 b 미만이라는 의미
- a는 범위에 포함되고 b는 포함되지 않음

- randint()함수는 (low, high) 사의의 정수 난수를 갖는 배열을 생성
- size는 배열의 형태를 지정하는 것으로 (d0, d1.....dn) 형식으로 입력
- low가 입력되지 않으면 0으로 간주하고, size가 입력되지 않으면 1로 간주합

- NumPy에서 난수를 생성

- 실수로 난수 배열을 생성하는 rand()

```
np.random.rand(2,3)
```

```
array([[0.76522847, 0.21738569, 0.85221167],
       [0.26094067, 0.83876583, 0.35160303]])
```

```
np.random.rand()
```

```
0.38526113115236005
```

```
np.random.rand(2,3,4)
```

```
array([[[0.35475222, 0.03063516, 0.45292572, 0.59612515],
        [0.64649559, 0.53502614, 0.76803366, 0.43924129],
        [0.38903127, 0.47275749, 0.64927646, 0.27300071]],

       [[0.75874428, 0.51341843, 0.2333768 , 0.85849203],
        [0.82121703, 0.34413019, 0.50638462, 0.42162077],
        [0.75948111, 0.25263743, 0.53600177, 0.55165934]]])
```

- 지정한 범위에 해당하는 정수로 난수 배열을 생성하는 randint()

```
np.random.randint(10, size=(3, 4))
```

```
array([[7, 9, 7, 9],
       [8, 6, 8, 1],
       [6, 8, 4, 3]])
```

```
np.random.randint(1, 30)
```

```
20
```

배열의 연산

기본 연산

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

```
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 x n 행렬에서 두 행렬의 m과 n이 각각 같은 경우

- 두 배열의 합은 각 배열의 같은 위치의 원소끼리 더함

- 두 NumPy 배열의 합을 구하는 예

```
arr1 + arr2
```

```
array([11, 22, 33, 44])
```

- 두 배열의 합과 마찬가지로 두 배열의 차는 같은 위치의 원소끼리 뺌

```
arr1 - arr2
```

```
array([ 9, 18, 27, 36])
```

- 배열에 상수를 곱할 수도 있음
- 배열에 상수를 곱하면 각 원소에 상수를 곱함

```
arr2 * 2
```

```
array([2, 4, 6, 8])
```

- 배열의 거듭제곱도 할 수 있음
- 배열의 거듭제곱은 배열의 각 원소에 거듭제곱을 함

```
arr2 ** 2
```

```
array([ 1,  4,  9, 16])
```

- 두 배열끼리의 곱셈은 각 원소끼리 곱함

```
arr1 * arr2
```

```
array([ 10,  40,  90, 160])
```

- 두 배열의 나눗셈은 각 원소끼리 나눔

```
arr1 / arr2
```

```
array([10., 10., 10., 10.])
```

- 배열의 복합 연산도 할 수 있음

```
arr1 / (arr2 ** 2)
```

```
array([10.        ,  5.        ,  3.33333333,  2.5       ])
```

- 배열은 비교 연산도 할 수 있음
- 원소별로 조건과 일치하는지 검사한 후 일치하면 True룰,
- 그렇지 않으면 False를 반환

```
arr1 > 20
```

```
array([False, False,  True,  True])
```

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()

- 배열을 생성

```
arr3 = np.arange(5)
arr3
```

```
array([0, 1, 2, 3, 4])
```

- 배열의 합(sum())과 평균(meant))을 구해보기로 함

```
[arr3.sum(), arr3.mean()]
```

```
[10, 2.0]
```

- 배열의 표준 편차(std())와 분산(var())을 구하겠음

```
[arr3.std(), arr3.var()]
```

```
[1.4142135623730951, 2.0]
```

- 배열의 최솟값(min())과 최댓값(Max())을 구함

```
[arr3.min(), arr3.max()]
```

```
[0, 4]
```

- 누적 합(cumsum())과 누적 곱(cumprod())의 예
- arr3의 경우에는 배열의 원소에 0이 있어서 누적 곱이 모두 0이 되므로 새로운 배열을 생성해서 적용

```
arr4 = np.arange(1,5)
arr4
```

```
array([1, 2, 3, 4])
```

- arr4를 이용해 누적 합(cumsum())과 누적 곱{cumprod())을 구함

```
arr4.cumsum()
```

```
array([ 1,  3,  6, 10])
```

```
arr4.cumprod()
```

```
array([ 1,  2,  6, 24])
```

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

- 행렬 연산을 하기 위해 2 x 2 행렬 A와 B를 만들겠음

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

```
array([[0, 1],
       [2, 3]])
```

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

```
array([[3, 2],
       [0, 1]])
```

- 행렬 A와 B를 이용한 행렬 곱의 예

```
A.dot(B)
```

```
array([[0, 1],
       [6, 7]])
```

```
np.dot(A,B)
```

```
array([[0, 1],
       [6, 7]])
```

- 행렬 A의 전치 행렬을 구하는 예

```
np.transpose(A)
```

```
array([[0, 2],
       [1, 3]])
```

```
A.transpose()
```

```
array([[0, 2],
       [1, 3]])
```

- 행렬 A의 역행렬을 구하기

```
np.linalg.inv(A)
```

```
array([[-1.5,  0.5],
       [ 1. ,  0. ]])
```

- 행렬 A의 행렬식을 구하기

```
np.linalg.det(A)
```

```
-2.0
```

배열의 인덱싱과 슬라이싱

- NumPy에서는 배열의 위치, 조건, 범 위를 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)

배열의 인덱싱

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

- 배열의 인덱싱 예를 살펴보기 위해 1차원 배열을 생성

```
a1 = np.array([0, 10, 20, 30, 40, 50])
a1
```

```
array([ 0, 10, 20, 30, 40, 50])
```

- 위의 배열(a1)에서 위치 0의 원소를 선택해서 가져오려면 a1[0]을 입력

```
a1[0]
```

```
0
```

- 배열(a1)에서 위치 4의 원소를 선택하려면 다음처럼 a[4]라고 입력

```
a1[4]
```

```
40
```

- 배열의 원소를 가져올 수 있을 뿐더러 변경할 수도 있음

```
a1[5] = 70
a1
```

```
array([ 0, 10, 20, 30, 40, 70])
```

- i차원 배열에서 여러 개의 원소를 선택하려면 다음과 같이 지정
- 배열명[[위치1, 위치2, ···, 위치n]]

- 1 차원 배열 a1 에서 1, 3, 4의 위치에 있는 원소 10, 30, 40을 가져옴

```
a1[[1,3,4]]
```

```
array([10, 30, 40])
```

- 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 배열에서 인덱싱으로 특정 원소를 선택해서 가져오는 예
- 2 차원 배열을 생성

```
a2 = np.arange(10, 100, 10).reshape(3,3)
a2
```

```
array([[10, 20, 30],
       [40, 50, 60],
       [70, 80, 90]])
```

- 2차원 배열 a2에서 '행_위치'가 0이고 '열_위치'가 2인 원소를 선택해서 가져오기

```
a2[0, 2]
```

- 2차원 배열의 행과 열의 위치를 지정해서 원소를 선택한 후 값을 변경

```
a2[2, 2] = 95
a2
```

```
array([[10, 20, 30],
       [40, 50, 60],
       [70, 80, 95]])
```

- 2차원 배열에서 '행_위치'룰 지정해 행 전체를 가져오기

```
a2[1]
```

```
array([40, 50, 60])
```

- 2차원 배열의 특정 행을 지정해서 행 전체를 변경
- 새로운 값을 입력할 때 'np.array(seq_data)'를 이용해도 되고 리스트를 이용해도 됨

- 특정 행을 지정해 행 전체의 값을 변경하는 예

```
a2[1] = np.array([45, 55, 65])
a2
```

```
array([[10, 20, 30],
       [45, 55, 65],
       [70, 80, 95]])
```

```
a2[1] = [47, 57, 67]
a2
```

```
array([[10, 20, 30],
       [47, 57, 67],
       [70, 80, 95]])
```

- 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

- 2차원 배열에서 행과 열의 위치를 지정해 여러 원소를 선택해서 가져오기

```
a2[[0, 2], [0, 1]]
```

```
array([10, 80])
```

- 2차원 배열 a2에서 (0, 0) 위치의 원소 10과 (2, 1) 위치의 원소 80을 선택해 가져옴

- 배열에 조건을 지정해 조건을 만족하는 배열을 선택할 수도 있음
- 배열명 [조건]
- 배열에서 조건을 만족하는 원소만 선택

- 조건을 지정해 조건에 맞는 배열의 원소만 선택하는 예

```
a = np.array([1, 2, 3, 4, 5, 6])
a[a > 3]
```

```
array([4, 5, 6])
```

- 배열 a에서 'a > 3' 조건을 만족하는 원소만 가져옴

- '(a % 2) = 0' 조건을 이용해 배열 a에서 짝수인 원소만 선택하는 예

```
a[(a % 2) == 0]
```

```
array([2, 4, 6])
```

배열의 슬라이싱

- 범위를 지정해 배열 의 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 '룰 지정하지 않으면 '끝_위치'는 배열의 길이가 되어 범위는 '시작_위치 ~ 배열의 끝'이 됨

- 1차원 배열에서 •시작_위치 '와 '끝_위치를 지정해서 슬라이싱하는 예

```
b1 = np.array([0, 10, 20, 30, 40, 50])
```

```
b1[1:4]
```

```
array([10, 20, 30])
```

- 배열 비에서 '시작_위치'는 1로 '끝_위치'는 4로 지정
- '1~3(시작_위치 ~ 끝_위치-1)'의 배열 원소가 반환

- 1차원 배열에서"시작_위치'와 '끝_위치'룰 지정하지 않고 슬라이싱하는 예

```
b1[:3]
```

```
array([ 0, 10, 20])
```

```
b1[2:]
```

```
array([20, 30, 40, 50])
```

- 슬라이싱을 이용해 원소를 가져올 수 있을 뿐만 아니라 원소를 변경도 가능

```
b1[2:5] = np.array([25, 35, 45])
b1
```

```
array([ 0, 10, 25, 35, 45, 50])
```

- b1[2 : 5]를 이용해 b1 배열의 '시작_위치'(2)부터 '끝_위치-1'(4)까지 3개의 원소를 새로운 값으로 변경

- 여러 원소의 값을 같은 값으로 변경하려면 하나의 값을 지정해 변경할 수 있음

```
b1[3:6] = 60
b1
```

```
array([ 0, 10, 25, 60, 60, 60])
```

- 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 ~ 열끝위치-1'이 됨
- '행시작위치'나 '열시작위치'를 생략하면 행과 열의 시작 위치는 0
- 행끝위치를 생략하면 원소의 행 범위는 '행시작_위치'부터 행의 끝까지 지정되고,
- '열끝위치'를 생략하면 원소의 열 범위는 '열시작위치'부터 열의 끝까지 지정

- 특정 행을 선택한 후 열을 슬라이싱하려면 다음과 같이 행의 위치를 지정하고 열의 시작과 끝 위치를 지정
- 배 열 [행위치] [열시작위치: 열끝위치]

- 2차원 배열의 슬라이싱을 위해 2차원 배열을 생성

```
b2 = np.arange(10, 100, 10).reshape(3,3)
b2
```

```
array([[10, 20, 30],
       [40, 50, 60],
       [70, 80, 90]])
```

- 2차원 배열에서 '행시작위치:행끝위치, 열시작위치:열끝위치'를 지정해 슬라이싱하는 예

```
b2[1:3, 1:3]
```

```
array([[50, 60],
       [80, 90]])
```

- 2차원 배열을 슬라이싱할 때 '행_시작_위치', '행 끝위치', '열_시작_위치', '열 끝위치'중 일부를 생략한 예

```
b2[:3, 1:]
```

```
array([[20, 30],
       [50, 60],
       [80, 90]])
```

- 2차원 배열에서 행을 지정하고 열을 슬라이싱하는 예

```
b2[1][0:2]
```

```
array([40, 50])
```

- 1 차원 배열에서와 마찬가지로 다음과 같이 2차원 배열에서도 슬라이싱된 배열에 값을 지정

```
b2[0:2, 1:3] = np.array([[25, 35], [55, 65]])
b2
```

```
array([[10, 25, 35],
       [40, 55, 65],
       [70, 80, 90]])
```

11.2 구조적 데이터 표시와 처리에 강한 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 데이터 생성하기

Series를 활용한 데이터 생성

- NumPy처럼 pandas도 보통 다음과 같이 'import ~ as ~ ' 형식으로 불러옴

```
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

- Series()를 이용해 Series 형식의 데이터를 생성하는 방법
- s = pd.Series(seq_data)
- Series()의 인자로는 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

- Series 데이터의 구조

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

```
0    10
```

```
1    20
2    30
3    40
4    50
dtype: int64
```

- Series 데이터를 출력하면 데이터 앞에 index가 함께 표시
- index는Series 데이터 생성 시 자동으로 만들어진 것으로 데이터를 처리할 때 이용

- Series 데이터는 index와 values를 분리해서 가져올 수 있음
- Series 데이터를 s라고 할 때 index 는 ’s.index’로 values는 ‘s.values’로 가져올 수 있음

- Series 데이터에서 index를 가져온 예

```
s1.index
print(s1.index)
```

```
RangeIndex(start=0, stop=5, step=1)
```

- RangeIndex는 index를 범위로 표시했음을 의미
- index의 범위는 'start ~ stop-1'이며 간격은 step만큼씩 증가

- Series 데이터에서 values를 가져온 예

```
s1.values
```

```
array([10, 20, 30, 40, 50])
```

- 결과가 NumPy의 배열과 형식이 같음

- 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
```

```
0    a
1    b
2    c
3    1
4    2
5    3
dtype: object
```

- 데이터가 없으면 NumPy를 임포트한 후에 np.nan으로 데이터가 없다고 표시할 수도 있음
- np.nan를 이용해서 Series 데이터에 특정 원소가 없음을 표시한 예

```
import numpy as np

s3 = pd.Series([np.nan,10,30])
s3
```

```
0     NaN
1    10.0
2    30.0
dtype: float64
```

- 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_Seq가 들어가게 됨
- index_Seq도 리스트와 튜플 타입의 데이터를 모두 사용할 수 있지만 주로 리스트 데이터를 이용
- seq_data의 항목 개수와 index_seq의 항목 개수는 같아야 한다는 것

- 어느 가게의 날짜별 판매량을 pandas의 Series 형식으로 입력
- 하루는 판매량 데이터가 없어서 np.nan를 입력

```
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
```

```
2018-08-07    200.0
2018-10-08    195.0
2018-10-09      NaN
2018-10-10    205.0
dtype: float64
```

- index에는 입력 인자 index의 리스트 데이터가 지정됐음을 확인

- 리스트 형식의 데이터와 index를 따로 입력했지만 파이썬의 딕셔너리를 이용하면 데이터와 index를 함께 입력할 수 있음
- s = pd.Series(dict_data)

- 입력 인자로 딕셔너리 데이터를 입력하면 딕셔너리 데이터의 키(keys)와값(values)이 각각 Series 데이터의 index와 values로 들어감

- Series 데이터의 index는 '국어', '영어', '수학'으로 지정됐고 values는 100, 90, 95로 지정된 것을 확인

```
s5 = pd.Series({'국어': 100, '영어': 95, '수학': 90})
s5
```

```
국어     100
영어      95
수학      90
dtype: int64
```

▾ 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 제공하 는 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

- 시작 날짜와 끝 날짜를 지정해 날짜 데이터를 생성

```
import pandas as pd

pd.date_range(start='2019-01-01',end='2019-01-07')
```

```
DatetimeIndex(['2019-01-01', '2019-01-02', '2019-01-03', '2019-01-04',
               '2019-01-05', '2019-01-06', '2019-01-07'],
              dtype='datetime64[ns]', freq='D')
```

- 시작 날짜{start)에서 끝 날짜(end)까지 하루씩 증가한 날짜 데이터가 생성

- 생성된 날짜 데이터의 형식은 모두 yyyy-mm-dd

```
pd.date_range(start='2019/01/01',end='2019.01.07')
```

```
DatetimeIndex(['2019-01-01', '2019-01-02', '2019-01-03', '2019-01-04',
               '2019-01-05', '2019-01-06', '2019-01-07'],
              dtype='datetime64[ns]', freq='D')
```

```
pd.date_range(start='01-01-2019',end='01/07/2019')
```

```
DatetimeIndex(['2019-01-01', '2019-01-02', '2019-01-03', '2019-01-04',
               '2019-01-05', '2019-01-06', '2019-01-07'],
              dtype='datetime64[ns]', freq='D')
```

```
pd.date_range(start='2019-01-01',end='01.07.2019')
```

```
DatetimeIndex(['2019-01-01', '2019-01-02', '2019-01-03', '2019-01-04',
               '2019-01-05', '2019-01-06', '2019-01-07'],
              dtype='datetime64[ns]', freq='D')
```

- 시작 날짜와 끝 날짜를 지정해 날짜 데이터를 생성했는데 끝 날짜를 지정하지 않고 periods 를 입력해서 날짜를 생성
- 시작 날짜로부터 7개의 날짜가 하루씩 증가해 생성
- freq 옵션을 쓰지 않으면 기본적으로 'D'가 지정되어 하루씩 날짜가 증가

```
pd.date_range(start='2019-01-01', periods = 7)
```

```
DatetimeIndex(['2019-01-01', '2019-01-02', '2019-01-03', '2019-01-04',
               '2019-01-05', '2019-01-06', '2019-01-07'],
              dtype='datetime64[ns]', freq='D')
```

- 2일씩 증가하는 날짜를 생성

```
pd.date_range(start='2019-01-01', periods = 4, freq = '2D')
```

```
DatetimeIndex(['2019-01-01', '2019-01-03', '2019-01-05', '2019-01-07'],
dtype='datetime64[ns]', freq='2D')
```

- 달력의 요일을 기준으로 일주일씩 증가하는 날짜를 생성

```
pd.date_range(start='2019-01-01', periods = 4, freq = 'W')
```

```
DatetimeIndex(['2019-01-06', '2019-01-13', '2019-01-20', '2019-01-27'],
dtype='datetime64[ns]', freq='W-SUN')
```

- 업무일 기준 2개월 월말 주기로 12개의 날짜를 생성한 예

```
pd.date_range(start='2019-01-01', periods = 12, freq = '2BM')
```

```
DatetimeIndex(['2019-01-31', '2019-03-29', '2019-05-31', '2019-07-31',
               '2019-09-30', '2019-11-29', '2020-01-31', '2020-03-31',
               '2020-05-29', '2020-07-31', '2020-09-30', '2020-11-30'],
              dtype='datetime64[ns]', freq='2BM')
```

- 분기 시작일을 기준으로 4개의 날짜를 생성한 예

```
pd.date_range(start='2019-01-01', periods = 4, freq = 'QS')
```

```
DatetimeIndex(['2019-01-01', '2019-04-01', '2019-07-01', '2019-10-01'],
dtype='datetime64[ns]', freq='QS-JAN')
```

- 연도의 첫날을 기준으로 1년 주기로 3개의 날짜를 생성한 예

```
pd.date_range(start='2019-01-01', periods = 3, freq = 'AS')
```

```
DatetimeIndex(['2019-01-01', '2020-01-01', '2021-01-01'], dtype='datetime64[ns]', freq='AS-
JAN')
```

- date_range()를 이용해 날짜뿐만 아니라 시간을 생성하는 방법

- 1시간 주기로 10개의 시간을 생성한 예

```
pd.date_range(start = '2019-01-01 08:00', periods = 10, freq='H')
```

```
DatetimeIndex(['2019-01-01 08:00:00', '2019-01-01 09:00:00',
               '2019-01-01 10:00:00', '2019-01-01 11:00:00',
               '2019-01-01 12:00:00', '2019-01-01 13:00:00',
               '2019-01-01 14:00:00', '2019-01-01 15:00:00',
               '2019-01-01 16:00:00', '2019-01-01 17:00:00'],
              dtype='datetime64[ns]', freq='H')
```

- 업무 시간을 기준으로 1 시간 주기로 10개의 시간을 생성하는 예

```
pd.date_range(start = '2019-01-01 08:00', periods = 10, freq='BH')
```

```
DatetimeIndex(['2019-01-01 09:00:00', '2019-01-01 10:00:00',
               '2019-01-01 11:00:00', '2019-01-01 12:00:00',
               '2019-01-01 13:00:00', '2019-01-01 14:00:00',
               '2019-01-01 15:00:00', '2019-01-01 16:00:00',
               '2019-01-02 09:00:00', '2019-01-02 10:00:00'],
              dtype='datetime64[ns]', freq='BH')
```

- 30분 단위로 4개의 시간을 생성한 예

```
pd.date_range(start = '2019-01-01 10:00', periods = 4, freq='30min')
```

```
DatetimeIndex(['2019-01-01 10:00:00', '2019-01-01 10:30:00',
               '2019-01-01 11:00:00', '2019-01-01 11:30:00'],
              dtype='datetime64[ns]', freq='30T')
```

- 분 단위를 'T'로 입력해도 결과는 같음

```
pd.date_range(start = '2019-01-01 10:00', periods = 4, freq='30T')
```

```
DatetimeIndex(['2019-01-01 10:00:00', '2019-01-01 10:30:00',
               '2019-01-01 11:00:00', '2019-01-01 11:30:00'],
              dtype='datetime64[ns]', freq='30T')
```

- 10초 단위로 증가하면서 4개의 시간을 생성

```
pd.date_range(start = '2019-01-01 10:00:00', periods = 4, freq='10S')
```

```
DatetimeIndex(['2019-01-01 10:00:00', '2019-01-01 10:00:10',
               '2019-01-01 10:00:20', '2019-01-01 10:00:30'],
              dtype='datetime64[ns]', freq='10S')
```

- date_range()를 이용해 Series의 index를 지정한 예
- 특정 날짜부터 5일간의 판매량을 표시하려면 다음과 같이 입력

```
index_date = pd.date_range(start = '2019-03-01', periods = 5, freq='D
pd.Series([51, 62, 55, 49, 58], index = index_date )
```

```
2019-03-01    51
2019-03-02    62
2019-03-03    55
2019-03-04    49
2019-03-05    58
Freq: D, dtype: int64
```

- pandas의 date_range()를 이용해 날짜와 시간을 자동으로 생성하는 방법

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

- DataFrame을 이용해 데이터를 생성하는 방법
- df = pd.DataFrame(data [, index = index—data, columns = columns_data])
- DataFrameO의 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

- DataFrame의 세로축 라벨을 index라고 하고 가로축 라벨을 columns라고 함
- 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

- 리스트를 이용해 DataFrame의 데이터를 생성하는 예

```
import pandas as pd

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

	0	1	2
0	1	2	3
1	4	5	6
2	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

- NumPy의 배열 데이터를 입력해 생성한 DataFrame 데이터의 예

```
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)
```

	0	1	2
0	10	20	30
1	40	50	60
2	70	80	90

- 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)
```

	A	B	C
2019-09-01	1	2	3
2019-09-02	4	5	6
2019-09-03	7	8	9
2019-09-04	10	11	12

- index와 columns에 지정한 값이 들어간 것
- index에는 date_range()로 생성한 날짜를, columns에는 리스트 데이터([*A*, *B*,'C'])를 입력

- 딕셔너리 타입으로 2차원 데이터를 입력한 예

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

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

- 만든 딕셔너리 데이터를 이용해 DataFrame 형식의 데이터를 생성

```
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

- index는 X]동으로 생성됐고 딕셔너리 데이터의 키(keys)는 DataFrame에서 columns 로 지정돼
- 표에서 각 열의 제목{고객 수 , 연도, 지사)처럼 들어간 것

- DataFrame 데이터의 열은 입력한 데이터의 순서대로 생성되지 않았음
- 딕셔너리 데이 터가 키(keys)에 따라서 자동으로 정렬됐기 때문
- 한글은 가나다순으로 영어는 알파벳순으로 정렬

- 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
```

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

- DataFrame 데이터에서 index, columns, values는
- 각각 DataFrame_data.index, DataFrame_data.columns, DataFrame—data.values로 확인할 수 있음

- DataFrame 데이터에서 index, columns, values를 각각 구한 예

```
df.index
```

```
RangeIndex(start=0, stop=5, step=1)
```

```
df.columns
```

```
Index(['연도', '지사', '고객 수'], dtype='object')
```

```
df.values
```

```
array([[2015, '한국', 200],
       [2016, '한국', 250],
       [2016, '미국', 450],
       [2017, '한국', 300],
       [2017, '미국', 500]], dtype=object)
```

데이터 연산

- pandas의 Series()와 DataFrame()으로 생성한 데이터끼리는 사칙 연산을 할 수 있음
- Series()로 생성한 Series 데이터의 예

```
s1 = pd.Series([1, 2, 3, 4, 5])
s2 = pd.Series([10, 20, 30, 40, 50])
s1 + s2
```

```
0    11
1    22
2    33
3    44
4    55
dtype: int64
```

```
s2 - s1
```

```
0     9
1    18
2    27
3    36
4    45
dtype: int64
```

```
s1 * s2
```

```
0      10
1      40
2      90
3     160
4     250
dtype: int64
```

```
s2 / s1
```

```
0    10.0
1    10.0
2    10.0
3    10.0
4    10.0
dtype: float64
```

- 파이씬의 리스트와 NumPy의 배열과 달리 pandas의 데이터끼리는 서로 크기가 달라도 연산할 수 있음

- 연산을 할 수 있는 항목만 연산을 수행

```
s3 = pd.Series([1, 2, 3, 4])
s4 = pd.Series([10, 20, 30, 40, 50])
s3 + s4
```

```
0    11.0
1    22.0
2    33.0
3    44.0
4     NaN
dtype: float64
```

```
s4 - s3
```

```
0     9.0
1    18.0
2    27.0
3    36.0
4     NaN
dtype: float64
```

```
s3 * s4
```

```
0     10.0
1     40.0
2     90.0
3    160.0
4      NaN
dtype: float64
```

```
s4/s3
```

```
0    10.0
1    10.0
2    10.0
3    10.0
```

```
4    NaN
dtype: float64
```

- s3와 S4의 데이터 크기는 다름
- 이 경우 연산할 수 있는 부분만 연산해 결과를 보여주고 연산할 수 없는 부분은 NaN으로 표시

- OataFrame()으로 생성한 DataFrame 데이터끼리도 사칙 연산을 할 수 있음

```
table_data1 = {'A': [1, 2, 3, 4, 5],
               'B': [10, 20, 30, 40, 50],
               'C': [100, 200, 300, 400, 500]}
df1 = pd.DataFrame(table_data1)
df1
```

	A	B	C
0	1	10	100
1	2	20	200
2	3	30	300
3	4	40	400
4	5	50	500

```
table_data2 = {'A': [6, 7, 8],
               'B': [60, 70, 80],
               'C': [600, 700, 800]}
df2 = pd.DataFrame(table_data2)
df2
```

	A	B	C
0	6	60	600
1	7	70	700
2	8	80	800

- 개의 DataFrame 데이터 df1 과 df2는 길이가 같지 않음
- 두 데이터의 길이가 같지 않더라도 앞의 Series 데이터처럼 연산할 수 있음

```
df1 + df2
```

	A	B	C
0	7.0	70.0	700.0
1	9.0	90.0	900.0
2	11.0	110.0	1100.0
3	NaN	NaN	NaN
4	NaN	NaN	NaN

- Series 데이터의 경우와 마찬가지로 DadaFrame 데이터의 경우도
- 연산할 수 있는 항목끼리만 연산하 고 그렇지 못한 항목은 빼으로 표시

- 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
df3
```

	봄	여름	가을	겨울
2012	256.5	770.6	363.5	139.3
2013	264.3	567.5	231.2	59.9
2014	215.9	599.8	293.1	76.9
2015	223.2	387.1	247.7	109.1
2016	312.8	446.2	381.6	108.1

- DataFrame 데이터(df3)를 이용해 통계
- pandas에서 제공하는 통계 메서드는 원소의 합을 구하는 sun(),
- 평균을 구하는 meanO, 표준 편차를 구하는 std(),
- 분산을 구하는 var(), 최솟값을 구하는 min(),
- 최댓값을 구하는 max(), 각 원소의 누적 합을 구하는 cumsun(),

- 각 원소의 누적 곱을 구하는 cumprod() 등이 있음

```
df3.mean()
```

```
봄      254.54
여름     554.24
가을     303.42
겨울      98.66
dtype: float64
```

```
df3.std()
```

```
봄       38.628267
여름     148.888895
가을      67.358496
겨울      30.925523
dtype: float64
```

- mean()과 std()는 연산의 방향 설정하기 위해 axis 인자를 추가할 수 있음

- 인자 axis가 0이면 DataFrame의 values에서 열별로 연산을 수행하고 , 1이면 행별로 연산을 수행
- axis 인자를 설정하지 않으면 기본값으로 0이 설정
- 연산의 방향을 행 방향으로 설정

```
df3.mean(axis=1)
```

```
2012     382.475
2013     280.725
2014     296.425
2015     241.775
2016     312.175
dtype: float64
```

```
df3.std(axis=1)
```

```
2012     274.472128
2013     211.128782
2014     221.150739
2015     114.166760
2016     146.548658
dtype: float64
```

- describe()를 이용하면 평균, 표준 편 차, 최솟값과 최댓값 등을 한 번에 구할 수도 있음

```
df3.describe()
```

	봄	여름	가을	겨울
count	5.000000	5.000000	5.000000	5.000000
mean	254.540000	554.240000	303.420000	98.660000
std	38.628267	148.888895	67.358496	30.925523
min	215.900000	387.100000	231.200000	59.900000
25%	223.200000	446.200000	247.700000	76.900000
50%	256.500000	567.500000	293.100000	108.100000
75%	264.300000	599.800000	363.500000	109.100000
max	312.800000	770.600000	381.600000	139.300000

데이터를 원하는 대로 선택하기

- pandas의 DataFrame 데이터를 원본 훼손 없이 원하는 부분만 선택하는 방법
- 데이터를 이용해 다음과 같이 DataFrame 데이터를 생성

```
import pandas as pd
import numpy as np

KTX_data = {'경부선 KTX': [39060, 39896, 42005, 43621, 41702, 41266,
            '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,
col_list = ['경부선 KTX','호남선 KTX','경전선 KTX','전라선 KTX','동해
index_list = ['2011', '2012', '2013', '2014', '2015', '2016', '2017']

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

	경부선 KTX	호남선 KTX	경전선 KTX	전라선 KTX	동해선 KTX
2011	39060	7313	3627	309	NaN

- DataFrame 데이터의 index, columns, values를 확인하려면 다음과 같이 작성

2013	42005	6873	4088	1954	NaN

```
df_KTX.index
```

```
Index(['2011', '2012', '2013', '2014', '2015', '2016', '2017'], dtype='object')
```

2016	41266	10622	4984	3945	3786.0

```
df_KTX.columns
```

```
Index(['경부선 KTX', '호남선 KTX', '경전선 KTX', '전라선 KTX', '동해선 KTX'],
dtype='object')
```

```
df_KTX.values
```

```
array([[39060.,  7313.,  3627.,   309.,    nan],
       [39896.,  6967.,  4168.,  1771.,    nan],
       [42005.,  6873.,  4088.,  1954.,    nan],
       [43621.,  6626.,  4424.,  2244.,    nan],
       [41702.,  8675.,  4606.,  3146.,  2395.],
       [41266., 10622.,  4984.,  3945.,  3786.],
       [32427.,  9228.,  5570.,  5766.,  6667.]])
```

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

- 인자 n 없이 Dal:aFrame_da1:a.head()와 DataFrame_data.tail()을수행

```
df_KTX.head()
```

	경부선 KTX	호남선 KTX	경전선 KTX	전라선 KTX	동해선 KTX
2011	39060	7313	3627	309	NaN
2012	39896	6967	4168	1771	NaN
2013	42005	6873	4088	1954	NaN
2014	43621	6626	4424	2244	NaN
2015	41702	8675	4606	3146	2395.0

```
df_KTX.tail()
```

	경부선 KTX	호남선 KTX	경전선 KTX	전라선 KTX	동해선 KTX
2013	42005	6873	4088	1954	NaN
2014	43621	6626	4424	2244	NaN
2015	41702	8675	4606	3146	2395.0
2016	41266	10622	4984	3945	3786.0
2017	32427	9228	5570	5766	6667.0

- n개의 처음과 마지막 행 데이터를 반환하려면 다음 과 같이 인자 n을 지정해서 head(n)와 tail(n)을 실행

```
df_KTX.head(3)
```

	경부선 KTX	호남선 KTX	경전선 KTX	전라선 KTX	동해선 KTX
2011	39060	7313	3627	309	NaN
2012	39896	6967	4168	1771	NaN
2013	42005	6873	4088	1954	NaN

```
df_KTX.tail(2)
```

	경부선 KTX	호남선 KTX	경전선 KTX	전라선 KTX	동해선 KTX
2016	41266	10622	4984	3945	3786.0
2017	32427	9228	5570	5766	6667.0

- DataFrame 데이터에서 연속된 구간의 행 데이터를 선택하려면 다음과 같이 '행 시작 위치'와 '끝 위치'를 지정
- DataFrame_data [행-시작_위 치: 행끝위 치]
- DataFrame 데이터(DataFrame_data) 중에서 '행_시작_위치 ~ 행끝위치-1'까지의 행 데이터를 반환
- 행의 위치는 0부터 시작

- df_KTX에서 행 위치 1 의 행 데이터 하나를 선택

```
df_KTX[1:2]
```

	경부선 KTX	호남선 KTX	경전선 KTX	전라선 KTX	동해선 KTX
2012	39896	6967	4168	1771	NaN

- 변수 df_KTX에서 행 위치 2에서 4까지의 행 데이터를 선택

```
df_KTX[2:5]
```

	경부선 KTX	호남선 KTX	경전선 KTX	전라선 KTX	동해선 KTX
2013	42005	6873	4088	1954	NaN
2014	43621	6626	4424	2244	NaN
2015	41702	8675	4606	3146	2395.0

- DataFrame 데이터를 생성할 때 index를 지정했다면 다음과 같이 index 항목 이름을 지정해 행을 선택 할 수도 있음
- DataFrame_data.loc[index_name]
- DataFrame_data의 데이터에서 index가 index_name인 행 데이터를 반환

- df_KTX에서 index로 지정한 항목 중 2011 년 데이터만 선택한 예

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

```
경부선 KTX    39060.0
호남선 KTX     7313.0
경전선 KTX     3627.0
전라선 KTX      309.0
동해선 KTX        NaN
Name: 2011, dtype: float64
```

- 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 선택한 예

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

	경부선 KTX	호남선 KTX	경전선 KTX	전라선 KTX	동해선 KTX
2013	42005	6873	4088	1954	NaN
2014	43621	6626	4424	2244	NaN

- 데이터에서 하나의 열만 선택하려면 다음과 같이 하나의 columns 항목 이름을 지정
- DataFrame_data[column_name]
- DataFrame_data에서 column_name으로 지정한 열이 선택

- df_KTX에서 columns의 항목 중 '경부선 KTX'를 지정해 하나의 열 데이터만 선택한 예

```
df_KTX['경부선 KTX']
```

```
2011    39060
2012    39896
2013    42005
2014    43621
2015    41702
2016    41266
2017    32427
Name: 경부선 KTX, dtype: int64
```

- DataFrame 데이터에서 하나의 열을 선택한 후 index의 범위를 지정해 원하는 데이터만 선택할 수도 있음
- DataFrame_data[column_naine][start_index_naine : end_index_nanie]
- DataFrame_data[column_name][start_index_pos : end_index_pos]
- column_name으로 하나의 열을 선택한 후 'start_index_name : end_index_name'으로
- index의 이름을 지정해 index의 범위를 선택할 수도 있고 ,
- 'start_index_pos:end_index_pos'로 index의 위치를 지정해 index의 범위를 선택할 수도 있음
- index의 위치는 0부터 시작

-df_KTX에서 ' 경부선 KTX'로 열을 선택한 후 2012년에서 2014년까지 index의 범위를 index의 이름으로 지정한 예

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

```
2012    39896
2013    42005
2014    43621
Name: 경부선 KTX, dtype: int64
```

- index의 위치로 범위를 지정한 예

```
df_KTX['경부선 KTX'][2:5]
```

```
2013    42005
2014    43621
2015    41702
Name: 경부선 KTX, dtype: int64
```

- DataFrame 데이터 중 하나의 원소만 선택하려면 다음 방법 중 하나를 이용
- 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_KTX에서 2016년의 '호남선 KTX_ 의 이용자 수를 선택하려면 다음과 같이 여러 방법으로 수행

```
df_KTX.loc['2016']['호남선 KTX']
```

```
10622.0
```

```
df_KTX.loc['2016','호남선 KTX']
```

```
10622
```

```
df_KTX['호남선 KTX']['2016']
```

```
10622
```

```
df_KTX['호남선 KTX'][5]
```

```
10622
```

```
df_KTX['호남선 KTX'].loc['2016']
```

```
10622
```

- DataFrame 데이터의 행과 열을 바 꾸는 방법
- 행렬에서 행과 열을 바꾸는 것을 전치(transpose)
- pandas 에서는 다음과 같은 방법으로 DataFrame 데이터의 전치를 구할 수 있음
- DataFrame_data.T

- df_KTX의 전치를 구하는 예

```
df_KTX.T
```

	2011	2012	2013	2014	2015	2016	2017
경부선 KTX	39060.0	39896.0	42005.0	43621.0	41702.0	41266.0	32427.0
호남선 KTX	7313.0	6967.0	6873.0	6626.0	8675.0	10622.0	9228.0
경전선 KTX	3627.0	4168.0	4088.0	4424.0	4606.0	4984.0	5570.0
전라선 KTX	309.0	1771.0	1954.0	2244.0	3146.0	3945.0	5766.0
동해선 KTX	NaN	NaN	NaN	NaN	2395.0	3786.0	6667.0

- DataFrame 데이터는 열의 항목 이름을 지정해 열의 순서를 지정할 수 있음

```
df_KTX
```

	경부선 KTX	호남선 KTX	경전선 KTX	전라선 KTX	동해선 KTX
2011	39060	7313	3627	309	NaN
2012	39896	6967	4168	1771	NaN
2013	42005	6873	4088	1954	NaN
2014	43621	6626	4424	2244	NaN
2015	41702	8675	4606	3146	2395.0
2016	41266	10622	4984	3945	3786.0
2017	32427	9228	5570	5766	6667.0

- DataFrame 데이터 변수 df_KTX에 열의 항목을 지정해 열의 순서를 다음과 같이 변경

```
df_KTX[['동해선 KTX', '전라선 KTX', '경전선 KTX', '호남선 KTX', '경부
```

	동해선 KTX	전라선 KTX	경전선 KTX	호남선 KTX	경부선 KTX
2011	NaN	309	3627	7313	39060
2012	NaN	1771	4168	6967	39896
2013	NaN	1954	4088	6873	42005
2014	NaN	2244	4424	6626	43621
2015	2395.0	3146	4606	8675	41702
2016	3786.0	3945	4984	10622	41266
2017	6667.0	5766	5570	9228	32427

데이터 통합하기

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

세로 방향으로 통합하기

- DataFrame에서 columns가 같은 두 데이터를 세로 방향(index 증가 방향)으로 합하려면
- 'append()' 를 이용
- DataFrame_data1.append(DataFrame_data2 [,ignore_index=True])
- 세로 방향으로 DataFrame_data1 다음에 DataFrame_data2가 추가돼서 Da切Frame 데이터로 반환
- 'ignore_index=True'를 입력하지 않으면 생성된 DataFrame 데이터에는 기존의 데이터의 index가 그대로 유지되고
- 'ignore_index=True'를 입력하면 생성된 Da切Frame 데이터에는 데이터 순서대로 새로운 index가 할당

- 두 학급의 시험 점수가 담긴 DataFrame 데이터(df1)를 다음과 같이 생성

```
import pandas as pd
import numpy as np

df1 = pd.DataFrame({'Class1': [95, 92, 98, 100],
                    'Class2': [91, 93, 97, 99]})
df1
```

	Class1	Class2
0	95	91
1	92	93
2	98	97
3	100	99

- 각각 두 명의 학생이 전학을 와서 다음과 같이 DataFrame 데이터(df2)를 추가로 생성

```
df2 = pd.DataFrame({'Class1': [87, 89],
                    'Class2': [85, 90]})
df2
```

	Class1	Class2
0	87	85
1	89	90

- 'append()'이용해 df1에 df2를 추가해서 데이터를 하나로

```
df1.append(df2)
```

	Class1	Class2
0	95	91
1	92	93
2	98	97
3	100	99
0	87	85
1	89	90

- 기존 데이터 df1 에 세로 방향으로 df2가 추가됐는데 생성된 데이터의 inde>(가 기존의 index와 같은 것을 볼 수 있음

- 생성된 데이터에서 순차적으로 index가 중가하게 하려면 옵션으로 'ignore, index=True'를 입력

```
df1.append(df2, ignore_index=True)
```

	Class1	Class2
0	95	91
1	92	93
2	98	97
3	100	99
4	87	85
5	89	90

- columns가 같지 않은 DataFrame 데이터를 'append()'를 이용해 추가한다면 데이터가 없는 부분은 NaN으로 채워짐

```
df3 = pd.DataFrame({'Class1': [96, 83]})
```

```
df3
```

	Class1
0	96
1	83

- 열이 두 개인 DataFrame 데이터(df2)에 열이 하나인 Da切Frame 데이터(df3)를 추가

```
df2.append(df3, ignore_index=True)
```

	Class1	Class2
0	87	85.0
1	89	90.0
2	96	NaN
3	83	NaN

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 다음과 같이 'join() '을 이용
- dataFrame_data1.join(DataFrame_data2)
- DataFrame_data1 다음에 가로 방향으로 DataFrame_data2가 추가돼서 DataFrame 데이터로 반환

-'join()'을 이용

- df1 과 index 방향으로 크기가 같은 DataFrame 데이터 하나를 생성

```
df4 = pd.DataFrame({'Class3': [93, 91, 95, 98]})
df4
```

	Class3
0	93
2	95

join()을 이용해 df1에 df4를 가로 방향으로 추가

```
df1.join(df4)
```

	Class1	Class2	Class3
0	95	91	93
1	92	93	91
2	98	97	95
3	100	99	98

index 라벨을 지정한 DataFrame의 데이터의 경우에도 index가 같으면 'join()'을 이용 해 가로 방향으로 데이터를 추가 할 수 있음

```
index_label = ['a','b','c','d']
df1a = pd.DataFrame({'Class1': [95, 92, 98, 100],
                     'Class2': [91, 93, 97, 99]}, index= index_label)
df4a = pd.DataFrame({'Class3': [93, 91, 95, 98]}, index=index_label)

df1a.join(df4a)
```

	Class1	Class2	Class3
a	95	91	93
b	92	93	91
c	98	97	95
d	100	99	98

- index의 크기가 다른 DataFrame 데이터를 'join()'을 이용해 추가한다면 데이터가 없는 부분은 NaN으로 채워짐

```
df5 = pd.DataFrame({'Class4': [82, 92]})
df5
```

- index의 크기가 2인 DataFrame 데이터(df5)를 'join()'을 이용해 index 크기가 4인 DataFrame 데이터(df1)에 추가

```
df1.join(df5)
```

	Class1	Class2	Class4
0	95	91	82.0
1	92	93	92.0
2	98	97	NaN
3	100	99	NaN

- index의 크기가 작은 DataFrame 데이터에서 원소가 없는 부분은 NaN으로 채워짐

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

- 우선 두 개의 DataFrame 데이터를 생성

```
df_A_B = pd.DataFrame({'판매월': ['1월', '2월', '3월', '4월'],
                       '제품A': [100, 150, 200, 130],
                       '제품B': [90, 110, 140, 170]})
df_A_B
```

	판매월	제품A	제품B
0	1월	100	90
1	2월	150	110
2	3월	200	140
3	4월	130	170

```
df_C_D = pd.DataFrame({'판매월': ['1월', '2월', '3월', '4월'],
                       '제품C': [112, 141, 203, 134],
                       '제품D': [90, 110, 140, 170]})
df_C_D
```

	판매월	제품C	제품D
0	1월	112	90
1	2월	141	110
2	3월	203	140
3	4월	134	170

- DataFrame 데이터 df_A_B와 df_C_D에 모두 있는 것이 '판매월'인 열 데이터이므로 이를 중심으로 두 DataFrame 데이터를 통합

```
df_A_B.merge(df_C_D)
```

	판매월	제품A	제품B	제품C	제품D
0	1월	100	90	112	90
1	2월	150	110	141	110
2	3월	200	140	203	140
3	4월	130	170	134	170

- 두 데이터가 '판매월'을 기준으로 통합
- 특정 열을 기준으로 두 DataFrame 데 이터가 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 인 자를 입력하지 않으면 자동으로 두 데이터에서 공통적으로 포함된 열이 선택
- how 인자에는 지정 된 특정 열(key)을 기준으로 통합 방법(mergejnethod)을 지정

- 두 개의 DataFrame 데이터를 생성

```
df_left = pd.DataFrame({'key':['A','B','C'], 'left': [1, 2, 3]})
df_left
```

	key	left
0	A	1
1	B	2
2	C	3

```
df_right = pd.DataFrame({'key':['A','B','D'], 'right': [4, 5, 6]})
df_right
```

	key	right
0	A	4
1	B	5
2	D	6

- 두 개의 데이터(dfjeft와 dfjright)에서 특정 열(key)의 일부('A'와'B') 데이터는 공통으로 있고 나머지('C'와 는 각각 한쪽에만 있음

```
df_left.merge(df_right, how='left', on = 'key')
```

	key	left	right
0	A	1	4.0
1	B	2	5.0
2	C	3	NaN

```
df_left.merge(df_right, how='right', on = 'key')
```

	key	left	right
0	A	1.0	4
1	B	2.0	5
2	D	NaN	6

```
df_left.merge(df_right, how='outer', on = 'key')
```

```
  key  left  right
```

```
df_left.merge(df_right, how='inner', on = 'key')
```

	key	left	right
0	A	1	4
1	B	2	5

- 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

데이터 파일을 읽고 쓰기

- pandas는 표 형식의 데이터 파일을 DataFrame 형식의 데이터로 읽어오는 방법과
- DataFrame 형식의 데이터를 표 형식으로 파일로 저장하는 편리한 방법을 제공

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

- 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
```

```
Overwriting sea_rain1.csv
```

- pandas의 read_csv()로 위의 csv 파일을 읽어옴

```
import pandas as pd

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

	연도	동해	남해	서해	전체
0	1996	17.4629	17.2288	14.4360	15.9067
1	1997	17.4116	17.4092	14.8248	16.1526
2	1998	17.5944	18.0110	15.2512	16.6044
3	1999	18.1495	18.3175	14.8979	16.6284
4	2000	17.9288	18.1766	15.0504	16.6178

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

	연도	동해	남해	서해	전체
0	1996	17.4629	17.2288	14.4360	15.9067
1	1997	17.4116	17.4092	14.8248	16.1526
2	1998	17.5944	18.0110	15.2512	16.6044
3	1999	18.1495	18.3175	14.8979	16.6284
4	2000	17.9288	18.1766	15.0504	16.6178

- 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
```

- read_csv()에서 읽고자 하는 데이터 파일의 구분자가 콤마가 아닌 경우에는
- 'sep=구분자' 옵션을 주가해야 텍스트 파일에서 pandas의 DataFrame 형식으로 데이터를 제대로 읽어올 있음

- 텍스트 파일에서 각 데이터 필드가 공백으로 구분돼 있으면 read_csv()에 sep=" "옵션을 추가해서 구분자가 공백임을 지정

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

	연도	동해	남해	서해	전체
0	1996	17.4629	17.2288	14.4360	15.9067
1	1997	17.4116	17.4092	14.8248	16.1526
2	1998	17.5944	18.0110	15.2512	16.6044
3	1999	18.1495	18.3175	14.8979	16.6284
4	2000	17.9288	18.1766	15.0504	16.6178

- 텍스트 파일의 확장자가 CSV가 아니더라도 read_csv()를 이용해 텍스트 데이터를 DataFrame 데이터로 읽어올 수 있음
- pandas에서 제공하는 read_csv(>로 텍스트 파일을 읽어오면 index가 자동으로 지정
- 자동 으로 생성된 index 말고 데이터 파일에서 특정 열(column)을 index로 선택하려면 옵션에 'index_col= 열 이름'을 추가

- csv 데이터 파일 .sea_rain1.csv_에서 '연도' 열을 index로 선택해 DataFrame 형식으로 데이터를 불러온 예

```
pd.read_csv('sea_rain1.csv', index_col="연도" )
```

연도	동해	남해	서해	전체
1996	17.4629	17.2288	14.4360	15.9067
1997	17.4116	17.4092	14.8248	16.1526
1998	17.5944	18.0110	15.2512	16.6044
1999	18.1495	18.3175	14.8979	16.6284
2000	17.9288	18.1766	15.0504	16.6178

- 줄력된 결과를 보면 DataFrame 데이터에는 자동으로 생성된 index 대신 index_col에 지정한 '연도'의 열 데이터가 index로 설정

▾ 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_name은 텍스트 파일 이름으로 경로를 포함
- 선택사항인 options에는 구분자와 문 자의 인코딩 방식 등을 지정할 수 있는데 지정하지 않으면 구분자는 콤마가 되고 문자의 인코딩 방식은 'utf-8’

- 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
```

User	Weight	Height
ID_1	62	165
ID_2	67	177
ID_3	55	160
ID_4	74	180

- 파일로 저장하기 전에 몸무게와 키를 이용해 체질량 지수(BMI)를 구해서 dtWH에 추가
- df_WH의 몸무게와 키 데이터를 이용해 체질량 지수(BMI)를 구함
- 키의 경우 입력한 데이터는 cm 단위여서 m 단위로 변경하기 위해 100으로 나눔

```
bmi = df_WH['Weight']/(df_WH['Height']/100)**2
bmi
```

```
User
ID_1    22.773186
ID_2    21.385936
ID_3    21.484375
ID_4    22.839506
dtype: float64
```

- DataFrame 데이터(df)에 'df['column_name']=colurnn_data'로 새로운 열 데이터를 추가
- 체질량 지수(BMI)를 df_WH에 추가

```
df_WH['BMI'] = bmi
df_WH
```

User	Weight	Height	BMI
ID_1	62	165	22.773186
ID_2	67	177	21.385936
ID_3	55	160	21.484375
ID_4	74	180	22.839506

- DataFrame 데이터 df_WH를 csv 파일로 저장

```
df_WH.to_csv('save_DataFrame.csv')
```

- DataFrame 데이터가 csv 파일로 잘 저장됐는지는 확인

```
!cat save_DataFrame.csv
```

```
User,Weight,Height,BMI
ID_1,62,165,22.77318640955005
ID_2,67,177,21.38593635289987
ID_3,55,160,21.484374999999996
ID_4,74,180,22.839506172839506
```

- DataFrame 데이터를 파일로 저장할 때 옵션을 지정하는 예제
- 한 회 사의 제품별 판매 가격과 판매량 정보가 있는 DataFrame 데이터를 생성

```
df_pr = pd.DataFrame({'판매가격':[2000, 3000, 5000, 10000],
                      '판매량':[32, 53, 40, 25]},
                      index=['P1001', 'P1002', 'P1003', 'P1004'])
df_pr.index.name = '제품번호'
df_pr
```

제품번호	판매가격	판매량

- DataFrame 데이터 df_pr를 텍스트 파일로 저장

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

- 저장된 파일을 확인

```
file_name = 'save_DataFrame_cp.txt'
df_pr.to_csv(file_name, sep=" ")
```

```
!cat save_DataFrame_cp949.txt
```

```
"","판매가격","판매량"
P1001 2000 32
P1002 3000 53
P1003 5000 40
P1004 10000 25
```

11.3 정리

- 데이터 분석에 아주 유용하게 활용할 수 있는 NumPy와 pandas
- 파이썬 프로그램을 작성할 때는 파이 썬의 기본 기능과 외부 패키지나 라이브러리를 함께 이용

- Python Package Index 홈페이지(https://pypi.org)

✓ 0초 오전 7:21에 완료됨