NumPy

안 화 수

numpy

numpy

1. numpy는 Numerical Python의 줄임말로 고성능의 과학계산 컴퓨팅과 데이터 분석에 필요한 패키지 입니다.
2. numpy 는 수치 계산을 효율적으로 하기 위한 모듈로서, 다차원 배열과 고수준의 수학 함수를 제공합니다.
3. numpy / pandas 두 라이브러리는 C언어로 작성돼 있으므로, 파이썬으로 만들어진 라이브러리 보다 처리 속도가 빠릅니다.
4. numpy 를 사용하려면, 표준 모듈이 아니므로 따로 설치해야 합니다. pip명령으로 설치하면 되는데, Anaconda 를 사용한다면 기본적으로 설치 되어 있습니다.

```
c:₩> pip  install  numpy
```

배열 생성

시퀀스 데이터로부터 배열 생성

- 배열(Array) 이란 순서가 있는 같은 종류의 데이터가 저장된 집합을 말한다.
- 시퀀스 데이터(seq_data)를 인자로 받아 numpy 의 배열 객체를 생성 해보자.
- 시퀀스 데이터(seq_data)로 리스트와 튜플 타입의 데이터를 모두 사용할 수 있지만 주로 리스트 데이터를 사용한다.

```
import numpy as np

arr_obj = np.array(seq_data)
```

배열 생성

- **시퀀스 데이터로부터 배열 생성**

array01.py

```
import numpy as np

data1 = [0, 1, 2, 3, 4, 5]
a1 = np.array(data1)
print(a1)                                   # [0 1 2 3 4 5]
print(a1.dtype)                             # int32

data2 = [0.1, 5, 4, 12, 0.5]
a2 = np.array(data2)
print(a2)                                   # [ 0.1  5.  4. 12. 0.5]
print(a2.dtype)                             # float64

a3 = np.array([0.5, 2, 0.01, 8])
print(a3)                                   # [0.5  2. 0.01 8. ]
print(a3.dtype)                             # float64

# 2차원 배열
a4 = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(a4)
print(a4.dtype)
```

배열 생성

범위를 지정해서 배열 생성

- numpy의 arange() 함수를 이용해서 범위를 지정해서 배열을 생성해보자.
- start부터 stop-1 까지 step 만큼 증가된 numpy 배열을 생성한다.

  arr_obj = np.arange([start, stop, step])

- numpy 배열의 arange()를 이용해 생성된 1차원 배열에, reshape(m행, n열) 함수를 추가하면 m x n 형태의 2차원 행렬(matrix)로 변경 할 수 있다.

  arr_obj = np.arange(12) . reshape(4, 3)

- linspace()를 이용해서 시작과 끝을 지정하고 데이터 개수를 지정해 numpy배열을 생성한다.

  arr_obj = np.linspace(start, stop, num)

배열 생성

- **범위를 지정해서 배열 생성**

array02.py

```
import numpy as np

# np.arange(start, stop, step)
a1 = np.arange(0, 10, 2)
print(a1)                                   # [0 2 4 6 8]

# np.arange(start, stop)
a2 = np.arange(1, 10)
print(a2)                                   # [1 2 3 4 5 6 7 8 9]

# np.arange(stop)
a3 = np.arange(5)
print(a3)                                    # [0 1 2 3 4]

# arange(12)로 12개의 숫자 생성후 reshape(4,3)으로 4x3 행렬을 만든다.
a4 = np.arange(12).reshape(4, 3)
print(a4)                                   # [[ 0  1  2]
                                            #  [ 3  4  5]
                                            #  [ 6  7  8]
                                            #  [ 9 10 11]]
print(a4.shape)                             #  (4, 3)   4행 3열 행렬
```

배열 생성

- **범위를 지정해서 배열 생성**

```
# linspace(start, stop, num)
# 1부터 10까지 10개의 데이터 생성
a5 = np.linspace(1, 10, 10)
print(a5)                                   # [ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]

# 1부터 10까지 3개의 데이터 생성
a6 = np.linspace(1, 10, 3)
print(a6)                                   # [ 1. 5.5  10. ]

# 0부터 pie 까지 동일한 간격으로 나눈 20개의 데이터를 생성
a7 = np.linspace(0, np.pi, 20)
print(a7)

# [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]
```

reshape()함수

- **reshape 함수 : 1차원 배열을 2차원 배열로 변환**
  reshape(행의 수, 열의 수)

array02_1.py

```
import numpy as np

array1 = np.arange(10)                  # 0 ~ 9 까지 원소를 가진 1차원 배열
print('array1:□n', array1)
# [0 1 2 3 4 5 6 7 8 9]

array2 = array1.reshape(2,5)            # 2행 5열
print('array2:□n', array2)
# [[0 1 2 3 4]
#  [5 6 7 8 9]]

array3 = array1.reshape(5,2)            # 5행 2열
print('array3:□n', array3)
# [[0 1]
#  [2 3]
#  [4 5]
#  [6 7]
#  [8 9]]
```

reshape()함수

```
array4 = np.arange(10)                  # 0 ~ 9까지 원소를 가진 1차원 배열
print(array4)                           # [0 1 2 3 4 5 6 7 8 9]

# reshape(-1, 5) : 열은 5열로 만들고,  -1은 행의 수를 남은 배열의 길이와
#                  남은 차원으로 추정해서 행을 지정하라는 의미
array5 = array4.reshape(-1,5)
print('array5 shape:', array5.shape)      #  (2, 5)
print(array5)
# [[0 1 2 3 4]
#  [5 6 7 8 9]]

# reshape(5,-1) : 행은 5행으로 만들고, -1은 열의 수를 남은 배열의 길이와
#                 남은 차원으로 추정해서 열을 지정하라는 의미
array6 = array4.reshape(5,-1)
print('array6 shape:', array6.shape)      # (5, 2)
print(array6)
# [[0 1]
#  [2 3]
#  [4 5]
#  [6 7]
#  [8 9]]

# 배열의 원소의 수와 차원의 수가 맞지 않으면 오류 발생한다.
array7 = np.arange(10)                  # 0 ~ 9까지 원소를 가진 1차원 배열
# array8 = array7.reshape(-1,4)         # 오류발생
```

reshape()함수

```
array9 = np.arange(8)                    # 0 ~ 7까지 원소를 가진 1차원 배열

# 1차원 배열을 2차원 배열로 변환
array10 = array9.reshape(-1,1)
print('array10:₩n', array10.tolist())    # [[0], [1], [2], [3], [4], [5], [6], [7]]
print('array10 shape:', array10.shape)   # (8,1)

# 1차원 배열을 3차원 배열로 변환
array3d = array9.reshape((2,2,2))        # 3차원 배열
print('array3d:₩n', array3d.tolist())    # [[[0, 1], [2, 3]], [[4, 5], [6, 7]]]
print('array3d shape:', array3d.shape)   # (2, 2, 2)

# 3차원 배열을 2차원 배열로 변환
array11 = array3d.reshape(-1,1)
print('array11:₩n', array11.tolist())    # [[0], [1], [2], [3], [4], [5], [6], [7]]
print('array11 shape:', array11.shape)   # (8,1)
```

특별한 형태의 배열

❖ **특별한 형태의 배열 생성**

zeros()함수는 모든 원소가 0인 다차원 배열을 생성
ones()함수는 모든 원소가 1인 다차원 배열을 생성

- np.zeros(n) : n개의 원소가 모두 0인 1차원 배열
- np.zeros((m, n)) : 모든 원소가 0인 m x n 형태의 2차원 배열(행렬)
- np.ones(n) : n개의 원소가 모두 1인 1차원 배열
- np.ones((m, n)) : 모든 원소가 1인 m x n 형태의 2차원 배열(행렬)
- np.eye(n) : n x n 단위행렬을 갖는 2차원 배열(행렬)

단위행렬(identity matrix)은 n x n 인 정사각형 행렬에서 주 대각선이 모두 1이고 나머지는 0인 행렬을 의미한다.

특별한 형태의 배열

- **특별한 형태의 배열 생성**

array03.py

```
import numpy as np

# zeros()함수로 원소의 갯수가 10개인 1차원 배열 생성
a1 = np.zeros(10)
print(a1)                       # [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

# zeros()함수를 이용해 3 x 4의 2차원 배열을 생성
a2 = np.zeros((3, 4))
print(a2)                       # [[0. 0. 0. 0.]
                                #  [0. 0. 0. 0.]
                                #  [0. 0. 0. 0.]]
```

특별한 형태의 배열

- **특별한 형태의 배열 생성**

```
# ones() 함수로 원소의 갯수가 5인 1차원 배열 생성
a3 = np.ones(5)
print(a3)                          # [1. 1. 1. 1. 1.]

# ones() 함수로 3 x 5 인 2차원 배열 생성
a4 = np.ones((3, 5))
print(a4)                          # [[1. 1. 1. 1. 1.]
                                   #  [1. 1. 1. 1. 1.]
                                   #  [1. 1. 1. 1. 1.]]

# 3 x 3 단위 행렬 생성
a5 = np.eye(3)
print(a5)                          # [[1. 0. 0.]
                                   #  [0. 1. 0.]
                                   #  [0. 0. 1.]]
```

배열의 데이터 타입변환

배열의 데이터 타입 변환

- numpy 배열은 숫자뿐만 아니라 문자열도 원소로 가질 수 있다.

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

- numpy 배열이 문자열로 되어 있다면, 연산을 하기 위해서는 숫자(정수, 실수)로 형 변환을 해야한다.

- numpy 배열을 형변환 하기 위해서는 astype() 함수를 사용한다.

  num_arr = str_arr . astype(dtype)

배열의 데이터 타입변환

- 배열의 데이터 타입 변환

array04.py

```
import numpy as np

# 1.문자를 원소로 갖는 numpy 배열 생성
str_a1 = np.array(['1.567','0.123','5.123','9','8'])
print(str_a1)                    # ['1.567' '0.123' '5.123' '9' '8']
print(str_a1.dtype)              # <U5     유니코드 5자리(문자 5자리)

# astype()함수로 문자를 실수형으로 형변환
num_a1 = str_a1.astype(float)
print(num_a1)                    # [1.567 0.123 5.123 9.    8.   ]
print(num_a1.dtype)              # float64
```

배열의 데이터 타입변환

- **배열의 데이터 타입 변환**

```
# 2.문자를 원소로 갖는 numpy 배열 생성
str_a2 = np.array(['1','3','5','7','9'])
print(str_a2)                                   # ['1' '3' '5' '7' '9']
print(str_a2.dtype)                             # <U1     유니코드 1자리(문자 1자리)

# astype()함수로 문자를 정수형으로 형변환
num_a2 = str_a2.astype(int)
print(num_a2)                                    # [1 3 5 7 9]
print(num_a2.dtype)                              # int32

# 3.실수를 원소로 갖는 numpy 배열 생성
num_f1 = np.array([10, 21, 0.549, 4.75, 5.98])
print(num_f1)                                    # [10.    21.     0.549  4.75   5.98 ]
print(num_f1.dtype)                              # float64

# astype()함수로 실수를 정수형으로 형변환
num_i1 = num_f1.astype(int)
print(num_i1)                                   # [10 21  0  4  5]
print(num_i1.dtype)                             # int32
```

난수 배열

난수 배열의 생성

- Python 에서 제공되는 random 모듈을 이용해서 난수 발생

  random.random() : 0.0 <= 실수 <1.0 사이의 실수 형태의 난수 발생
  random.randint(1, 10) : 1<= 정수 <=10 사이의 정수 형태의 난수 발생

- Numpy 모듈에서 제공되는 rand(), randint() 함수를 이용해서 난수 발생

  np.random.rand() : 0이상 1미만 사이의 실수 형태의 난수 발생
  np.random.rand(2, 3) : 0이상 1미만 사이의 실수 형태의 2행 3열 난수 발생

  np.random.randint([low], high,[size]) : low이상 high미만의 정수 형태의 난수 발생
  np.random.randn() : 표준편차가 1이고, 평균값이 0인 정규분포에서 표본추출

난수 배열

❖ **난수 배열의 생성**

array05.py

```
import numpy as np

# 0.0 <= r1 < 1.0 사이의 실수형태의 난수 발생
r1 = np.random.rand()
print(r1)

# 0.0 <= r2 < 1.0 사이의 실수형태의 2행 3열 난수 발생
r2 = np.random.rand(2,3)
print(r2)

# 1 <= r3 < 30 사이의 정수형태의 난수 발생
r3 = np.random.randint(1,30)
print(r3)

# 0 <= r4 < 10 사이의 정수형태의 3행 4열 난수 발생
r4 = np.random.randint(10, size=(3, 4))
print(r4)
```

배열의 산술연산

배열의 산술 연산

- 배열의 형태(shape)가 같은 경우에 덧셈, 뺄셈, 곱셈, 나눗셈을 할 수 있다.
- 배열의 각 원소 끼리 산술 연산을 수행한다.

배열의 산술연산

❖ **배열의 산술 연산**

array06.py

```
import numpy as np

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

a1 = arr1 + arr2                    # 배열 더하기 : 각 원소끼리 더하기
print(a1)                           # [11 22 33 44]

a2 = arr1 - arr2                    # 배열 빼기: 각 원소끼리 빼기
print(a2)                           # [ 9 18 27 36]

a3 = arr2 * 2                       # 배열에 상수 곱하기
print(a3)                           # [2 4 6 8]

a4 = arr2 ** 2                      # 배열에 거듭제곱
print(a4)                           # [ 1  4  9 16]
```

배열의 산술연산

- **배열의 산술 연산**

```
a5 = arr1 * arr2                    # 배열 곱하기 : 각 원소끼리 곱함
print(a5)                           # [ 10  40  90 160]

a6 = arr1 / arr2                    # 배열 나누기 : 각 원소끼리 나눔
print(a6)                           # [10. 10. 10. 10.]

a7 = arr1 / (arr2 ** 2)             # 복합 연산
print(a7)                           # [ 10. 5. 3.33333333  2.5 ]

# 비교연산 : 각 원소와 비교해서 참이면 True, 거짓이면 False 리턴
a8 = arr1 > 20
print(a8)                           # [False  False  True  True]
```

numpy의 통계분석 함수

numpy의 통계분석 함수

- numpy 에서는 통계에서 자주 사용하는 함수들을 지원한다

sum() : 원소의 합
mean() : 평균
var() : 분산
std() : 표준편차
max() : 최대값
min() : 최소값
cumsum() : 각 원소의 누적 합
cumprod() : 각 원소의 누적 곱

numpy의 통계분석 함수

❖ **numpy의 통계분석 함수**

array07.py

```
import numpy as np

arr3 = np.arange(5)
print(arr3)                    # [0 1 2 3 4]

sum = arr3.sum()               # 배열 각 원소의 합
print(sum)                     # 10

mean = arr3.mean()             # 배열 원소의 평균
print(mean)                    # 2.0

var = arr3.var()               # 분산
print(var)                     # 2.0

std = arr3.std()               # 표준편차
print(std)                     # 1.4142135623730951
```

numpy의 통계분석 함수

- **numpy의 통계분석 함수**

```
max = arr3.max()                    # 최대값
print(max)                          # 4

min = arr3.min()                    # 최소값
print(min)                          # 0

arr4 = np.arange(1, 5)
print(arr4)                         # [1 2 3 4]

cumsum = arr4.cumsum()              # 각 원소들의 누적합
print(cumsum)                       # [ 1  3  6 10]

cumprod = arr4.cumprod()            # 각 원소들의 누적곱
print(cumprod)                      # [ 1  2  6  24]
```

행렬연산

행렬 연산

- Numpy 에서는 배열의 단순 연산뿐만 아니라, 선행대수(Linear Algebra)를 위한 행렬(2차원 배열) 연산도 지원한다.

- 다양한 기능 중에서 행렬곱, 전치행렬, 역행렬, 행렬식을 구하는 방법을 알아보자

행렬곱(matrix product)	A.dot(B) 혹은 np.dot(A,B)
전치행렬(transpose matrix)	A.transpose() 혹은 np.transpose(A)
역행렬(inverse matrix)	np.linalg.inv(A)
행렬식(determinant)	np.linalg.det(A)

행렬연산

행렬 연산

array08.py

```
import numpy as np

# 2 x 2 행렬 A와 B 생성
A = np.array([0,1,2,3]).reshape(2,2)
B = np.array([3,2,0,1]).reshape(2,2)

# 행렬 곱
print(A.dot(B))
print(np.dot(A,B))
# [[0 1]
#  [6 7]]
```

행렬연산

행렬 연산

```
# 행렬 A의 전치 행렬
print(A.transpose())
print(np.transpose(A))
# [[0 2]
#  [1 3]]

# 행렬 A의 역행렬
print(np.linalg.inv(A))
# [[-1.5  0.5]
#  [ 1.   0. ]]

# 행렬 A의 행렬식
print(np.linalg.det(A))                    # -2.0
```

배열의 인덱싱

배열의 인덱싱

- 배열에서 선택된 원소의 값을 가져오거나 변경 할 수 있다.
  배열의 위치나 조건을 지정해 배열의 원소를 선택하는 것을 인덱싱(indexing)이라고 한다.

- 1차원 배열 인덱싱
  배열명[index번호] : index번호의 원소1개를 인덱싱
  배열명[[index번호, index번호,…., index번호]] : 여러 개의 원소를 인덱싱

- 2차원 배열 인덱싱
  배열명[행 위치, 열 위치]
  배열명[[행 위치1,행 위치2,…,행 위치n] , [열 위치1,열 위치2,…..,열 위치n]]
  배열명[조건]

배열의 인덱싱

- **1차원 배열의 인덱싱**

array09.py

```
import numpy as np

# 1차원 배열 정의
a1 = np.array([0, 10, 20, 30, 40, 50])
print(a1)                                   # [ 0 10 20 30 40 50]

# index 번호 1번 위치의 원소
print(a1[1])                                # 10

# index 번호 4번 위치의 원소
print(a1[4])                                # 40

# index 번호 5번 위치의 원소값 50 -> 70으로 수정
a1[5] = 70
print(a1[5])                                # 70

# 1차원 배열에서 여러개의 원소 구하기
print(a1[[1,3,4]])                          # [10 30 40]
```

배열의 인덱싱

- **2차원 배열의 인덱싱**

array10.py

```
import numpy as np


# 2차원 배열 정의 : 10 ~ 90까지 10씩 증가된 배열
a2 = np.arange(10, 100, 10).reshape(3, 3)
print(a2)                                   # [[10 20 30]
                                            #  [40 50 60]
                                            #  [70 80 90]]

# 배열명[행위치, 열위치] : 0행 2열의 원소를 구함
print(a2[0, 2])                             # 30

# 2행 2열의 값을 90 -> 95 로 변경
a2[2, 2] = 95
print(a2)                                   # [[10 20 30]
                                            #  [40 50 60]
                                            #  [70 80 95]]
```

배열의 인덱싱

- **2차원 배열의 인덱싱**

```
# 2차원 배열의 1행 전체를 변경:[40,50,60]->[45,55,65]
a2[1] = np.array([45, 55, 65])
print(a2)                                   # [[10 20 30]
                                            #  [45 55 65]
                                            #  [70 80 95]]

# 2차원 배열의 행과 열의 위치를 지정해서 여러 원소 구하기
# 배열명[ [행위치1,행위치2],[열위치1,열위치2] ]
# (0,0)위치의 원소 10, (2,1)위치의 원소 80 을 구함
print(a2[[0, 2], [0, 1]])                   # [10 80]

# 2차원 배열에서 조건을 만족하는 원소 구하기
# 배열명[조건] : 조건에 맞는 원소만 구함
a = np.array([1, 2, 3, 4, 5, 6])
print(a[a > 3])                             # [4 5 6]
```

배열의 슬라이싱

배열의 슬라이싱(Slicing)

- 배열에서 선택된 원소의 값을 가져오거나 변경 할 수 있다.
  배열의 범위를 지정해서 배열의 원소를 선택하는 것을 슬라이싱(Slicing)
  이라고 한다.

- 1차원 배열 슬라이싱
  배열명[시작위치 : 끝위치] : 시작위치 ~ 끝위치 -1 번 원소 슬라이싱
  배열명[시작위치 :] : 시작위치 ~ 끝위치 원소 슬라이싱
  배열명[: 끝위치] : 처음부터 ~ 끝위치 -1 번 원소 슬라이싱

- 2차원 배열 슬라이싱
  배열[행 시작위치 : 행 끝위치 , 열 시작위치 : 열 끝위치]
  배열명[[행 시작위치 : 행 끝위치] , [열 시작위치 : 열 끝위치]]
  배열명[행 위치] [열 시작위치 : 열 끝위치]

배열의 슬라이싱

❖ **1차원 배열의 슬라이싱**

array11.py

```
import numpy as np

# 1차원 배열 정의
b1 = np.array([0, 10, 20, 30, 40, 50])

# 배열[시작위치 : 끝위치]:시작위치 ~ 끝위치-1 까지 슬라이싱
# 인덱스 1 ~ 3번 원소를 슬라이싱
print(b1[1:4])                                  # [10 20 30]

# 배열[시작위치 : ]:시작위치 ~ 끝위치 까지 슬라이싱
# 인덱스 2번 부터 끝까지 슬라이싱
print(b1[2:])                                   # [20 30 40 50]

# 배열[ : 끝위치]:시작위치 ~ 끝위치-1 까지 슬라이싱
# 처음부터 인덱스 2번 까지 슬라이싱
print(b1[:3])                                   # [ 0 10 20]

# 슬라이싱으로 원소의 값 변경
# 인덱스 2~4번 원소의 값을 [25, 35, 45]로 변경
b1[2:5] = np.array([25, 35, 45])
print(b1)                                       # [ 0 10 25 35 45 50]

# 인덱스 3 ~ 5번 원소의 값을 60 으로 변경
b1[3:6] = 60
print(b1)                                       # [ 0 10 25 60 60 60]
```

배열의 슬라이싱

❖ **2차원 배열의 슬라이싱**

array12.py

```
import numpy as np

# 2차원 배열 정의 : 10 ~ 90 까지 10씩 증가된 3행 3열 배열
b2 = np.arange(10, 100, 10).reshape(3,3)
print(b2)                                          # [[10 20 30]
                                                   #  [40 50 60]
                                                   #  [70 80 90]]

# 배열[행 시작위치 : 행 끝위치 , 열 시작위치 : 열 끝위치]
# 1~2행, 1~2열 슬라이싱
print(b2[1:3, 1:3])                                # [[50 60]
                                                   #  [80 90]]

# 0~2행, 1~2열 슬라이싱
print(b2[:3, 1:])                                  # [[20 30]
                                                   #  [50 60]
                                                   #  [80 90]]

# 1행, 0~1열 슬라이싱
print(b2[1][0:2])                                  # [40 50]

# 2차원 배열에서 슬라이싱 된 배열에 값을 지정
# 0~1행, 1~2열 위치에 [25, 35], [55, 65] 값 변경
b2[0:2, 1:3] = np.array([[25, 35],[55, 65]])
print(b2)                                          # [[10 25 35]
                                                   #  [40 55 65]
                                                   #  [70 80 90]]
```