NUMPY

Numpy

1. 개요

Numarray와 Numeric이라는 오래된 Python 패키지를 계승해서 나온 수학 및 과학 연산을 위한 파이썬 패키지이다. Py는 파이썬을 나타내기 때문에, 일반적으로 넘파이라고 읽는다.

프로그래밍 하기 어려운 C, C++, Fortran[1] 등의 언어에 비하면, NumPy로는 꽤나 편리하게 수치해석을 실행할 수 있다. 게다가 NumPy 내부는 상당부분 C나 포트란으로 작성되어 실행 속도도 꽤 빠른편이다. 기본적으로 ndarray라는 자료를 생성하고 이를 바탕으로 색인, 처리, 연산 등을 하는 기능을 수행한다. 물론 넘파이 자체만으로도 난수생성, 푸리에변환, 행렬연산, 간단한 기술통계 분석 정도는 가능하지만 실제로는 SciPy, Pandas, matplotlib 등 다른 Python 패키지와 함께 쓰이는 경우가 많다.

파이썬으로 수치해석, 통계 관련 기능을 구현한다고 할 때 NumPy는 가장 기본이 되는 모듈이다. 그만큼 NumPy는 수치해석, 통계 관련 작업시 중요한 역할을 하므로, 파이썬으로 관련 분야에 도전하고자 한다면 반드시 이에 대한 기초를 잘 쌓아두고 가자.

하도 기본적으로 쓰이는 모듈이다 보니 아래와 같이 np로 호출하는 것이 관례가 되었다.

```
import numpy as np
```

Numpy

2. 다른 Python 패키지와의 관계

SciPy (싸이파이 = 싸이언스 + 파이썬)

NumPy와 SciPy는 서로 떨어질 수 없을 정도로 밀접한 관계에 있으며 SciPy를 활용할 때에는 상당히 많이 NumPy를 이용하게 된다. 실제로 SciPy에 관한 책을 구매했을 때 책의 앞부분은 SciPy 관련 내용보다는 오히려 NumPy의 기초에 대한 내용 위주로 보게 되는 경우가 많다.

SciPy는 수치해석시 NumPy를 보다 본격적으로 이용할 수 있게 해 준다. 사실 대학 학부 수준의 수치해석 교재에 있는 여러 수치 미분법이나 수치 적분법, 수치미분방정식 해법(룽게-쿠타 방법 등)을 구현하는 데에는 전적으로 Python 기본 메서드와 NumPy만으로 충분하다. 그렇지만 SciPy를 이용하면 NumPy만으로는 길게 코딩해야 하는 기법들을 단 2~3줄에 구현할 수 있다. 게다가 NumPy로 구현하기 막막하거나 사실상 불가능한 것들도 SciPy로는 쉽게 결과를 얻을 수 있다. 따라서 NumPy와 SciPy를 적절하게 혼용하게 되면 MATLAB 부럽지 않은 수치해석 툴을 얻은 셈이 된다.

Numpy

2. 다른 Python 패키지와의 관계

SymPy (심파이 = 심볼릭 + 파이썬)

Python의 대표적인 기호계산 패키지인 SymPy도 NumPy와 잘 연동된다. 예컨대 NumPy만으로 구현하기 까다로운 함수의 경우, SymPy로 원하는 함수를 구하고, 이 함수를 바탕으로 NumPy를 이용하여 그 함수를 기반으로 배열형 자료를 구할 수 있다. 어떤 SymPy 함수를 일단 구하기만 하면, f = sympy.lambdify(정의역 문자, SymPy 함수, 'numpy') 꼴의 간단한 코딩만으로 NumPy에 적용가능한 함수 f를 얻을 수 있다. 여기에서 f(numpy.linspace(0,1, 101))를 입력하면 함수에 대해 0에서 1까지 100등분한 정의역 점들에 대한 함숫값들의 배열을 얻는다. 간혹 NumPy 자체나 SciPy만으로 구하기 어려운 함수도 SymPy로는 간결하게 구할 수 있는 경우도 많으므로 NumPy와 SymPy를 같이 배워두면 여러모로 쓸모가 많다.

Matplotlib

파이썬에서 널리 사용되는 그래픽 패키지인 Matplotlib에서도 NumPy는 자주 애용된다. 물론 파이썬에 기본적으로 내장된 리스트형 자료로도 충분히 수많은 종류의 데이터를 그래프화할 수 있다. 그러나 아무래도 NumPy를 쓸 때보다는 코드가 굉장히 복잡해지고, 그래프를 얻는 속도도 느리다. 실제로 인터넷 상의 Matplotlib 튜토리얼이나 예제들을 봐도 다들 NumPy의 ndarray 자료형을 쓰지 리스트형은 잘 쓰지 않는다. 당연하지만 Matplotlib를 기반으로 좀 더 유려한 그래프와 도표를 얻는 seaborn과 같은 패키지에도 굉장히 유용하게 쓰인다.

Numpy

2. 다른 Python 패키지와의 관계

Pandas

Pandas는 NumPy보다 더 복잡한 형태의 자료(Series, DataFrame)를 다루지만, NumPy를 이용해 만든 array 자료를 이용해서 Series나 DataFrame 자료를 생성하거나 수정할 수 있다.

실제로는 'NumPy + 다른 한 개의 패키지' 조합뿐만 아니라 'NumPy + 여러 개의 패키지' 조합을 쓸 때가 많다. 예컨대 SymPy로 함수의 라플라스 변환을 구하고, NumPy를 이용해서 변환한 함수의 값들에 대한 배열형 자료를 얻고, 이 배열형 자료를 바탕으로 Matplotlib을 써서 그래프를 그릴 수 있다.

NUMPY ARRAY

Numpy

Numpy

많은 숫자 데이터를 하나의 변수에 넣고 관리 할 때 리스트는 속도가 느리고 메모리를 많이 차지하는 단점이 있다. 배열(array)을 사용하면 적은 메모리로 많은 데이터를 빠르게 처리할 수 있다. 배열은 리스트와 비슷하지만 다음과 같은 점에서 다르다.

1. 모든 원소가 같은 자료형이어야 한다.
2. 원소의 갯수를 바꿀 수 없다.

파이썬은 자체적으로 배열 자료형을 제공하지 않는다. 따라서 배열을 구현한 다른 패키지를 임포트해야 한다. 파이썬에서 배열을 사용하기 위한 표준 패키지는 넘파이(NumPy)다.

Numpy Array

ndarray

ndarray는 N-dimensional Array의 약자이다. 이름 그대로 1차원 배열 이외에도 2차원 배열, 3차원 배열 등의 다차원 배열 자료 구조를 지원한다.

넘파이는 수치해석용 파이썬 패키지이다. 다차원의 배열 자료구조 클래스인 ndarray 클래스를 지원하며 벡터와 행렬을 사용하는 선형대수 계산에 주로 사용된다. 내부적으로는 BLAS 라이브러리와 LAPACK 라이브러리를 사용하고 있으며 C로 구현된 CPython에서만 사용할 수 있다.

넘파이의 배열 연산은 C로 구현된 내부 반복문을 사용하기 때문에 파이썬 반복문에 비해 속도가 빠르며 벡터화 연산(vectorized operation)을 이용하여 간단한 코드로도 복잡한 선형 대수 연산을 수행할 수 있다. 또한 배열 인덱싱(array indexing)을 사용한 질의(Query) 기능을 이용하여 간단한 코드로도 복잡한 수식을 계산할 수 있다.

1차원 배열 만들기

넘파이 패키지 임포트

배열을 사용하기 위해서는 우선 다음과 같이 넘파이 패키지를 임포트한다. 넘파이는 np라는 이름으로 임포트하는 것이 관례이다.

```
import numpy as np
```

1차원 배열 만들기

넘파이의 array 함수에 리스트를 넣으면 ndarray 클래스 객체 즉, 배열로 변환해 준다. 따라서 1 차원 배열을 만드는 방법은 다음과 같다.

```
ar = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
ar
```

1차원 배열 만들기

리스트와 비슷해 보이지만 type 명령으로 자료형을 살펴보면 ndarray임을 알 수 있다.

type(ar)

만들어진 ndarray 객체의 표현식(representation)을 보면 바깥쪽에 array()란 것이 붙어 있을 뿐 리스트와 동일한 구조처럼 보인다. 그러나 배열 객체와 리스트 객체는 많은 차이가 있다.

우선 리스트 클래스 객체는 각각의 원소가 다른 자료형이 될 수 있다. 그러나 배열 객체 객체는 C언어의 배열처럼 연속적인 메모리 배치를 가지기 때문에 모든 원소가 같은 자료형이어야 한다. 이러한 제약사항이 있는 대신 원소에 대한 접근과 반복문 실행이 빨라진다.

벡터화 연산(vectorized operation)

배열 객체는 배열의 각 원소에 대한 반복 연산을 하나의 명령어로 처리하는 벡터화 연산(vectorized operation)을 지원한다. 예를 들어 다음처럼 여러 개의 데이터를 모두 2배 해야 하는 경우를 생각하자.

```
data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

for 반복문을 사용하면 다음과 같이 구현할 수 있다.

```
answer = []
for di in data:
    answer.append(2 * di)
answer
```

하지만 벡터화 연산을 사용하면 다음과 같이 for 반복문이 없이 한번의 연산으로 할 수 있다. 계산 속도도 반복문을 사용할 때 보다 훨씬 빠르다.

```
x = np.array(data)
x
2 * x
```

a == 2

벡터화 연산(vectorized operation)

참고로 일반적인 리스트 객체에 정수를 곱하면 객체의 크기가 정수배 만큼으로 증가한다.

```
L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(2 * L)
```

벡터화 연산은 비교 연산과 논리 연산을 포함한 모든 종류의 수학 연산에 대해 적용된다.

```
a = np.array([1, 2, 3])
b = np.array([10, 20, 30])
2 * a + b
a == 2
b > 10
(a == 2) & (b > 10)
```

2차원 배열 만들기

2차원 배열은 행렬(matrix)이라고 하는데 행렬에서는 가로줄을 행(row)이라고 하고 세로줄을 열(column)이라고 부른다.
다음과 같이 리스트의 리스트(list of list)를 이용하면 2차원 배열을 생성할 수 있다. 안쪽 리스트의 길이는 행렬의 열의 수 즉, 가로 크기가 되고 바깥쪽 리스트의 길이는 행렬의 행의 수, 즉 세로 크기가 된다. 예를 들어 2개의 행과 3개의 열을 가지는 2 x 3 배열은 다음과 같이 만든다.

```
c = np.array([[0, 1, 2], [3, 4, 5]])  # 2 x 3 array
C

# 행의 갯수
len(c)

# 열의 갯수
len(c[0])
```

3차원 배열 만들기

리스트의 리스트의 리스트를 이용하면 3차원 배열도 생성할 수 있다. 크기를 나타낼 때는 가장 바깥쪽 리스트의 길이부터 가장 안쪽 리스트 길이의 순서로 표시한다. 예를 들어 2 x 3 x 4 배열은 다음과 같이 만든다.

```
d = np.array([[[1, 2, 3, 4],
               [5, 6, 7, 8],
               [9, 10, 11, 12]],
              [[11, 12, 13, 14],
               [15, 16, 17, 18],
               [19, 20, 21, 22]]])   # 2 x 3 x 4 array

#3차원 배열의 깊이, 행, 열
len(d), len(d[0]), len(d[0][0])
```

배열의 차원과 크기 알아내기

배열의 차원 및 크기를 구하는 더 간단한 방법은 배열의 ndim 속성과 shape 속성을 이용하는 것이다. ndim 속성은 배열의 차원, shape 속성은 배열의 크기를 반환한다.

```
# a = np.array([1, 2, 3])
print(a.ndim)
print(a.shape)

# c = np.array([[0, 1, 2], [3, 4, 5]])
print(c.ndim)
print(c.shape)

print(d.ndim)
print(d.shape)
```

배열의 인덱싱

일차원 배열의 인덱싱은 리스트의 인덱싱과 같다.

```
a = np.array([0, 1, 2, 3, 4])
a[2]
a[-1]
```

다차원 배열일 때는 다음과 같이 콤마(comma ,)를 사용하여 접근할 수 있다. 콤마로 구분된 차원을 축(axis)이라고도 한다. 그래프의 x축과 y축을 떠올리면 될 것이다.

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

a[0, 0]  # 첫번째 행의 첫번째 열

a[0, 1]  # 첫번째 행의 두번째 열

a[-1, -1]  # 마지막 행의 마지막 열
```

배열 슬라이싱

배열 객체로 구현한 다차원 배열의 원소 중 복수 개를 접근하려면 일반적인 파이썬 슬라이싱(slicing)과 comma(,)를 함께 사용하면 된다.

```
a = np.array([[0, 1, 2, 3], [4, 5, 6, 7]])
a

a[0, :]  # 첫번째 행 전체

a[:, 1]  # 두번째 열 전체

a[1, 1:]  # 두번째 행의 두번째 열부터 끝열까지

a[:2, :2]
```

배열 인덱싱

넘파이 배열 객체의 또다른 강력한 기능은 팬시 인덱싱(fancy indexing)이라고도 부르는 배열 인덱싱(array indexing) 방법이다. 인덱싱이라는 이름이 붙었지만 사실은 데이터베이스의 질의(Query) 기능을 수행한다. 배열 인덱싱에서는 대괄호(Bracket, [])안의 인덱스 정보로 숫자나 슬라이스가 아니라 위치 정보를 나타내는 또 다른 ndarray 배열을 받을 수 있다. 여기에서는 이 배열을 편의상 인덱스 배열이라고 부르겠다. 배열 인덱싱의 방식에은 불리언(Boolean) 배열 방식과 정수 배열 방식 두가지가 있다.

먼저 불리안 배열 인덱싱 방식은 인덱스 배열의 원소가 True, False 두 값으로만 구성되며 인덱스 배열의 크기가 원래 ndarray 객체의 크기와 같아야 한다.

예를 들어 다음과 같은 1차원 ndarray에서 짝수인 원소만 골라내려면 짝수인 원소에 대응하는 인덱스 값이 True이고 홀수인 원소에 대응하는 인덱스 값이 False인 인덱스 배열을 넣으면 된다.

```
a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
idx = np.array([True, False, True, False, True,
                False, True, False, True, False])
a[idx]
```

배열 인덱싱

조건문 연산을 사용하면 다음과 같이 간단히 쓸 수 있다.

```
a % 2

a % 2 == 0

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

정수 배열 인덱싱에서는 인덱스 배열의 원소 각각이 원래 ndarray 객체 원소 하나를 가리키는 인덱스 정수이여야 한다. 예를 들어 1차원 배열에서 홀수번째 원소만 골라내는 것은 다음과 같다

```
a = np.array([11, 22, 33, 44, 55, 66, 77, 88, 99])
idx = np.array([0, 2, 4, 6, 8])
a[idx]
```

배열 인덱싱

이 때는 배열 인덱스의 크기가 원래의 배열 크기와 달라도 상관없다. 같은 원소를 반복해서 가리키는 경우에는 배열 인덱스가 원래의 배열보다 더 커지기도 한다.

```
a = np.array([11, 22, 33, 44, 55, 66, 77, 88, 99])
idx = np.array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2])
a[idx]
```

배열 인덱싱은 다차원 배열의 각 차원에 대해서도 할 수 있다.

```
a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
a

a[:, [True, False, False, True]]

a[[2, 0, 1], :]
```

연습문제 1

연습 문제 1

넘파이를 사용하여 다음과 같은 행렬을 만든다.

```
10 20 30 40
50 60 70 80
```

연습문제 2

연습 문제 2

다음 행렬과 같은 행렬이 있다.

```
m = np.array([[ 0,  1,  2,  3,  4],
              [ 5,  6,  7,  8,  9],
              [10, 11, 12, 13, 14]])
```

1. 이 행렬에서 값 7 을 인덱싱한다.
2. 이 행렬에서 값 14 을 인덱싱한다.
3. 이 행렬에서 배열 [6, 7] 을 슬라이싱한다.
4. 이 행렬에서 배열 [7, 12] 을 슬라이싱한다.
5. 이 행렬에서 배열 [[3, 4], [8, 9]] 을 슬라이싱한다.

연습문제 3

연습 문제 3

다음 행렬과 같은 배열이 있다.

```
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
              11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
```

1. 이 배열에서 3의 배수를 찾아라.
2. 이 배열에서 4로 나누면 1이 남는 수를 찾아라.
3. 이 배열에서 3으로 나누면 나누어지고 4로 나누면 1이 남는 수를 찾아라.

넘파이 자료형

넘파이의 배열 즉, ndarray클래스는 원소가 모두 같은 자료형이어야 한다. array 명령으로 배열을 만들 때 자료형을 명시적으로 적용하려면 dtype 인수를 사용한다. 만약 dtype 인수가 없으면 주어진 데이터를 저장할 수 있는 자료형을 스스로 유추한다. 만들어진 배열의 자료형을 알아내려면 dtype 속성을 보면 된다.

```
x = np.array([1, 2, 3])
x.Dtype

x = np.array([1.0, 2.0, 3.0])
x.Dtype

x = np.array([1, 2, 3.0])
x.dtype
```

dtype 인수로 지정할 자료형은 다음 표에 보인것과 같은 "dtype 접두사"로 시작하는 문자열이고 이 글자 뒤에 오는 숫자는 바이트 수 혹은 글자 수를 의미한다. 예를 들어 f8은 8바이트(64비트) 부동소수점 실수를 뜻하고 U4 는 4글자 유니코드 문자열을 뜻한다. 숫자를 생략하면 운영체제에 따라 알맞은 크기를 지정한다.

넘파이 자료형

dtype 접두사	설명	사용 예
b	불리언	b (참 혹은 거짓)
i	정수	i8 (64비트)
u	부호없는 정수	u8 (64비트)
f	부동소수점	f8 (64비트)
c	복수 부동소수점	c16 (128비트)
O	객체	0 (객체에 대한 포인터)
S	바이트 문자열	S24 (24 글자)
U	유니코드 문자열	U24 (24 유니코드 글자)

```
x = np.array([1, 2, 3], dtype='f')
x.Dtype

x[0] + x[1]
```

```
x = np.array([1, 2, 3], dtype='U')
x.Dtype

x[0] + x[1]
```

Inf와 NaN

넘파이에서는 무한대를 표현하기 위한 np.inf(infinity)와 정의할 수 없는 숫자를 나타내는 np.nan(not a number)을 사용할 수 있다. 다음 예와 같이 1을 0으로 나누려고 하거나 0에 대한 로그 값을 계산하면 무한대인 np.inf이 나온다. 0을 0으로 나누려고 시도하면 np.nan이 나온다.

```
np.array([0, 1, -1, 0]) / np.array([1, 0, 0, 0])

np.log(0)

np.exp(-np.inf)
```

배열 생성

NumPy는 몇가지 단순한 배열을 생성하는 명령을 제공한다.

zeros, ones
zeros_like, ones_like
empty
arange
linspace, logspace
zeros

크기가 정해져 있고 모든 값이 0인 배열을 생성하려면 zeros 명령을 사용한다. 인수로는 배열을 크기를 뜻하는 정수를 넣는다.

```
a = np.zeros(5)
a

#크기를 뜻하는 튜플을 입력하면 다차원 배열도 만들 수 있다.
b = np.zeros((2, 3))
b
```

배열 생성

array 명령과 마찬가지로 dtype 인수를 명시하면 해당 자료형 원소를 가진 배열을 만든다.

```
c = np.zeros((5, 2), dtype="i")
c
```

문자열 배열도 가능하지면 모든 원소의 문자열 크기가 같아야 한다. 만약 더 큰 크기의 문자열을 할당하면 잘릴 수 있다.

```
d = np.zeros(5, dtype="U4")
d

d[0] = "abc"
d[1] = "abcd"
d[2] = "ABCDE"
d
```

배열 생성

0이 아닌 1로 초기화된 배열을 생성하려면 ones 명령을 사용한다.

```
e = np.ones((2, 3, 4), dtype="i8")
e
```

만약 크기를 튜플로 명시하지 않고 다른 배열과 같은 크기의 배열을 생성하고 싶다면 ones_like, zeros_like 명령을 사용한다.

```
f = np.ones_like(b, dtype="f")
f
```

배열의 크기가 커지면 배열을 초기화하는데도 시간이 걸린다. 이 시간을 단축하려면 배열을 생성만 하고 특정한 값으로 초기화를 하지 않는 empty 명령을 사용할 수 있다. empty 명령으로 생성된 배열에는 기존에 메모리에 저장되어 있던 값이 있으므로 배열의 원소의 값을 미리 알 수 없다.

```
g = np.empty((4, 3))
g
```

배열 생성

arange 명령은 NumPy 버전의 range 명령이라고 볼 수 있다. 특정한 규칙에 따라 증가하는 수열을 만든다.

```
np.arange(10)  # 0 .. n-1

np.arange(3, 21, 2)  # 시작, 끝(포함하지 않음), 단계
```

linspace 명령이나 logspace 명령은 선형 구간 혹은 로그 구간을 지정한 구간의 수만큼 분할한다.

```
np.linspace(0, 100, 5)  # 시작, 끝(포함), 개수

np.logspace(0.1, 1, 10)
```

전치 연산

2차원 배열의 전치(transpose) 연산은 행과 열을 바꾸는 작업이다. 이는 배열의 T 속성으로 구할 수 있다. 메서드가 아닌 속성이라는 점에 유의 한다.

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


A.T
```

배열의 크기 변경

일단 만들어진 배열의 내부 데이터는 보존한 채로 형태만 바꾸려면 reshape 명령이나 메서드를 사용한다. 예를 들어 12개의 원소를 가진 1차원 행렬은 3x4 형태의 2차원 행렬로 만들 수 있다.

```
a = np.arange(12)
a

b = a.reshape(3, 4)
b
```

일단 만들어진 배열의 내부 데이터는 보존한 채로 형태만 바꾸려면 reshape 명령이나 메서드를 사용한다. 예를 들어 12개의 원소를 가진 1차원 행렬은 3x4 형태의 2차원 행렬로 만들 수 있다.

```
a.reshape(3, -1)

a.reshape(2, 2, -1)

a.reshape(2, -1, 2)
```

배열의 크기 변경

다차원 배열을 무조건 1차원으로 만들기 위해서는 flatten 나 ravel 메서드를 사용한다.

```
a.flatten()

a.ravel()
```

배열 사용에서 주의할 점은 길이가 5인 1차원 배열과 행, 열의 갯수가 (5,1)인 2차원 배열 또는 행, 열의 갯수가 (1, 5)인 2차원 배열은 데이터가 같아도 엄연히 다른 객체라는 점이다.

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

x.reshape(1, 5)

x.reshape(5, 1)
```

이렇게 같은 배열에 대해 차원만 1차원 증가시키는 경우에는 newaxis 명령을 사용하기도 한다.

```
x[:, np.newaxis]
```

배열 연결

행의 수나 열의 수가 같은 두 개 이상의 배열을 연결하여(concatenate) 더 큰 배열을 만들 때는 다음과 같은 명령을 사용한다.

hstack
vstack
dstack
stack
r_
c_
tile
hstack

hstack 명령은 행의 수가 같은 두 개 이상의 배열을 옆으로 연결하여 열의 수가 더 많은 배열을 만든다. 연결할 배열은 하나의 리스트에 담아야 한다.

```
a1 = np.ones((2, 3))
a1
a2 = np.zeros((2, 2))
a2
np.hstack([a1, a2])
```

배열 연결

vstack 명령은 열의 수가 같은 두 개 이상의 배열을 위아래로 연결하여 행의 수가 더 많은 배열을 만든다. 연결할 배열은 마찬가지로 하나의 리스트에 담아야 한다.

```
b1 = np.ones((2, 3))
b1
a2 = np.zeros((2, 2))
a2

np.vstack([b1, b2])
```

dstack 명령은 제3의 축 즉, 행이나 열이 아닌 깊이(depth) 방향으로 배열을 합친다. 가장 안쪽의 원소의 차원이 증가한다. 즉 가장 내부의 숫자 원소가 배열이 된다. shape 정보로 보자면 가장 끝에 값이 2인 차원이 추가되는 것이다.이 예제의 경우에는 shape 변화가 2개의 (3 x 4) -> 1개의 (3 x 4 x 2)가 된다.

```
c1 = np.ones((3, 4))
c1
c2 = np.zeros((3, 4))
c2

np.dstack([c1, c2])
```

배열 연결

stack 명령은 dstack의 기능을 확장한 것으로 dstack처럼 마지막 차원으로 연결하는 것이 아니라 사용자가 지정한 차원(축으로) 배열을 연결한다. axis 인수(디폴트 0)를 사용하여 연결 후의 회전 방향을 정한다. 디폴트 인수 값은 0이고 가장 앞쪽에 차원이 생성된다. 즉, 배열 두 개가 겹치게 되므로 연결하고자 하는 배열들의 크기가 모두 같아야 한다.

다음 예에서는 axis=0 이므로 가장 바깥에 값이 2인 차원이 추가된다. 즉, shape 변화는 2개의 (3 x 4) -> 1개의 (2 x 3 x 4) 이다.

```
c = np.stack([c1, c2])
c
c.shape
```

axis 인수가 1이면 두번째 차원으로 새로운 차원이 삽입된다. 다음 예에서 즉, shape 변화는 2개의 (3 x 4) -> 1개의 (3 x 2 x 4) 이다

```
c = np.stack([c1, c2], axis=1)
c
c.shape
```

배열 연결

r_ 메서드는 hstack 명령과 비슷하게 배열을 좌우로 연결한다. 다만 메서드임에도 불구하고 소괄호(parenthesis, ())를 사용하지 않고 인덱싱과 같이 대괄호(bracket, [])를 사용한다. 이런 특수 메서드를 인덱서(indexer)라고 한다.

```
np.r_[np.array([1, 2, 3]), np.array([4, 5, 6])]
```

c_ 메서드는 배열의 차원을 증가시킨 후 좌우로 연결한다. 만약 1차원 배열을 연결하면 2차원 배열이 된다.

```
np.c_[np.array([1, 2, 3]), np.array([4, 5, 6])]
```

tile 명령은 동일한 배열을 반복하여 연결한다.

```
np.tile(a, (3, 2))
```

연습문제

연습 문제 3.2.1

지금까지 공부한 명령어를 사용하여 다음과 같은 배열을 만들어라.

```
array([[   0.,    0.,    0.,    1.,    1.],
       [   0.,    0.,    0.,    1.,    1.],
       [   0.,    0.,    0.,    1.,    1.],
       [  10.,   20.,   30.,   40.,   50.],
       [  60.,   70.,   80.,   90.,  100.],
       [ 110.,  120.,  130.,  140.,  150.],
       [   0.,    0.,    0.,    1.,    1.],
       [   0.,    0.,    0.,    1.,    1.],
       [   0.,    0.,    0.,    1.,    1.],
       [  10.,   20.,   30.,   40.,   50.],
       [  60.,   70.,   80.,   90.,  100.],
       [ 110.,  120.,  130.,  140.,  150.]])
```

2차원 그리드 포인트 작성

변수가 2개인 2차원 함수의 그래프를 그리거나 표를 작성하려면 2차원 영역에 대한 (x,y) 좌표값 쌍 즉, 그리드 포인트(grid point)를 생성하여 각 좌표에 대한 함수 값을 계산해야 한다. 예를 들어 x, y 라는 두 변수를 가진 함수에서 x가 0부터 2까지, y가 0부터 4까지의 사각형 영역에서 변화하는 과정을 보고 싶다면 이 사각형 영역 안의 다음과 같은 그리드 포인트들에 대해 함수를 계산해야 한다.

(x,y)=(0,0),(0,1),(0,2),(0,3),(0,4),(1,0),…(2,4)

이러한 그리드 포인트를 만드는 과정을 도와주는 것이 meshgrid 명령이다. meshgrid 명령은 사각형 영역을 구성하는 가로축의 점들과 세로축의 점을 나타내는 두 벡터를 인수로 받아서 이 사각형 영역을 이루는 조합을 출력한다. 결과는 그리드 포인트의 x 값만을 표시하는 행렬과 y 값만을 표시하는 행렬 두 개로 분리하여 출력한다.

```
x = np.arange(3)
x
y = np.arange(5)
y
X, Y = np.meshgrid(x, y)
X
Y
[list(zip(x, y)) for x, y in zip(X, Y)]
```

배열의 연산 – 벡터화 연산

앞서 넘파이가 벡터화 연산(vectorized operation)을 지원한다고 이야기하였다. 벡터화 연산을 쓰면 명시적으로 반복문을 사용하지 않고도 배열의 모든 원소에 대해 반복연산을 할 수 있다. 벡터화 연산의 또다른 장점은 선형 대수 공식과 동일한 아주 간단한 파이썬 코드를 작성할 수 있다는 점이다.

예를 들어 선형 대수에서 두 벡터의 합은 다음과 같이 구한다.

일 때, 두 벡터의 합

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ 10000 \end{bmatrix}, \quad y = \begin{bmatrix} 10001 \\ 10002 \\ 10003 \\ \vdots \\ 20000 \end{bmatrix},$$

z=x+y

은 다음과 같이 구한다.

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ 10000 \end{bmatrix} + \begin{bmatrix} 10001 \\ 10002 \\ 10003 \\ \vdots \\ 20000 \end{bmatrix} = \begin{bmatrix} 1+10001 \\ 2+10002 \\ 3+10003 \\ \vdots \\ 10000+20000 \end{bmatrix} = \begin{bmatrix} 10002 \\ 10004 \\ 10006 \\ \vdots \\ 30000 \end{bmatrix}$$

벡터화 연산

만약 벡터화 연산을 사용하지 않는다면 이 연산은 반복문을 사용하여 다음과 같이 만들어야 한다. 이 코드에서 %%time은 셀 코드의 실행시간을 측정하는 아이파이썬(IPython) 매직(magic) 명령이다.

```
x = np.arange(1, 10001)
y = np.arange(10001, 20001)

%%time
z = np.zeros_like(x)
for i in range(10000):
    z[i] = x[i] + y[i]

z[:10]
```

그러나 벡터화 연산을 사용하면 덧셈 연산 하나로 끝난다. 위에서 보인 선형 대수의 벡터 기호를 사용한 연산과 결과가 완전히 동일하다. 연산 속도도 벡터화 연산이 훨씬 빠르다.

```
%%time
z = x + y

z[:10]
```

벡터화 연산

사칙 연산뿐 아니라 비교 연산과 같은 논리 연산도 벡터화 연산이 가능하다.

```
a = np.array([1, 2, 3, 4])
b = np.array([4, 2, 2, 4])
c = np.array([1, 2, 3, 4])

np.all(a == b)
np.all(a == c)
```

지수 함수, 로그 함수 등의 수학 함수도 벡터화 연산을 지원한다.

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

np.exp(a)

10 ** a

np.log(a + 1)
```

스칼라와 벡터/행렬의 곱셈

스칼라와 벡터/행렬의 곱도 선형 대수에서 사용하는 식과 넘파이 코드가 일치한다.

```
x = np.arange(10)
x

100 * x

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

100 * x
```

브로드캐스팅

벡터(또는 행렬)끼리 덧셈 혹은 뺄셈을 하려면 두 벡터(또는 행렬)의 크기가 같아야 한다. 넘파이에서는 서로 다른 크기를 가진 두 배열의 사칙 연산도 지원한다. 이 기능을 브로드캐스팅(broadcasting)이라고 하는데 크기가 작은 배열을 자동으로 반복 확장하여 크기가 큰 배열에 맞추는 방벙이다.

예를 들어 다음과 같이 벡터와 스칼라를 더하는 경우를 생각하자.

$$x = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}, \quad x + 1 = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 1 = ?$$

브로드캐스팅은 다음과 같이 스칼라를 벡터와 같은 크기로 확장시켜서 덧셈 계산을 하는 것이다.

$$\begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 1 = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

브로드캐스팅

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

y = np.ones_like(x)
y

x + y

x + 1
```

브로드캐스팅

브로드캐스팅은 다음처럼 더 차원이 높은 경우에도 적용된다.

$$\begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \\ 4 & 4 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \end{bmatrix} + [0 \quad 1 \quad 2] = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix}$$

```
x = np.vstack([range(7)[i:i + 3] for i in range(5)])
x

y = np.arange(5)[:, np.newaxis]
y
```

```
x + y

y = np.arange(3)
y

x + y
```

차원 축소

행렬의 하나의 행에 있는 원소들을 하나의 데이터 집합으로 보고 그 집합의 평균을 구하면 각 행에 대해 하나의 숫자가 나오게 된다. 예를 들어 10x5 크기의 2차원 배열에 대해 행-평균을 구하면 10개의 숫자를 가진 1차원 벡터가 나오게 된다. 이러한 연산을 차원 축소(dimension reduction) 연산이라고 한다.

넘파이는 다음과 같은 차원 축소 연산 명령 혹은 메서드를 지원한다.

최대/최소: min, max, argmin, argmax
통계: sum, mean, median, std, var
불리언: all, any

```
x = np.array([1, 2, 3, 4])
x
np.sum(x)
x.sum()

x = np.array([1, 3, 2])
x.min()
x.max()
x.argmin()  # 최솟값의 위치
x.argmax()  # 최댓값의 위치
```

차원 축소

```
x = np.array([1, 2, 3, 1])

x.mean()
np.median(x)
np.all([True, True, False])

a = np.zeros((100, 100), dtype=np.int)
a
np.any(a != 0)
np.all(a == a)

a = np.array([1, 2, 3, 2])
b = np.array([2, 2, 3, 2])
c = np.array([6, 4, 4, 5])
((a <= b) & (b <= c)).all()
```

차원 축소

연산의 대상이 2차원 이상인 경우에는 어느 차원으로 계산을 할 지를 axis 인수를 사용하여 지시한다. axis=0인 경우는 열 연산, axis=1인 경우는 행 연산이다. 디폴트 값은 axis=0이다. axis 인수는 대부분의 차원 축소 명령에 적용할 수 있다.

```
x = np.array([[1, 1], [2, 2]])
x


x.sum()
x.sum(axis=0)   # 열 합계
x.sum(axis=1)   # 행 합계
```

연습 문제

연습 문제 3.3.1

실수로 이루어진 5 x 6 형태의 데이터 행렬을 만들고 이 데이터에 대해 다음과 같은 값을 구한다.

1. 전체의 최댓값
2. 각 행의 합
3. 각 행의 최댓값
4. 각 열의 평균
5. 각 열의 최솟값

정렬

sort 함수나 메서드를 사용하여 배열 안의 원소를 크기에 따라 정렬하여 새로운 배열을 만들 수도 있다. 2차원 이상인 경우에는 행이나 열을 각각 따로따로 정렬하는데 axis 인수를 사용하여 행을 정렬할 것인지 열을 정렬한 것인지 결정한다. axis=0이면 각각의 행을 따로따로 정렬하고 axis=1이면 각각의 열을 따로따로 정렬한다. 디폴트 값은 -1 즉 가장 안쪽(나중)의 차원이다.

```
a = np.array([[4,  3,  5,  7],
              [1, 12, 11,  9],
              [2, 15,  1, 14]])
a

np.sort(a)  # axis=-1 또는 axis=1 과 동일

np.sort(a, axis=0)
```

정렬

sort 메서드는 해당 객체의 자료 자체가 변화하는 자체변화(in-place) 메서드이므로 사용할 때 주의를 기울여야 한다.

```
a.sort(axis=1)
a
```

만약 자료를 정렬하는 것이 아니라 순서만 알고 싶다면 argsort 명령을 사용한다.

```
a = np.array([42, 38, 12, 25])
j = np.argsort(a)
j

a[j]

np.sort(a)
```

연습 문제

연습 문제 3.3.2

다음 배열은 첫번째 행(row)에 학번, 두번째 행에 영어 성적, 세번째 행에 수학 성적을 적은 배열이다. 영어 성적을 기준으로 각 열(column)을 재정렬하라.

```
array([[  1,   2,   3,   4],
       [ 46,  99, 100,  71],
       [ 81,  59,  90, 100]])
```

기술 통계

넘파이는 다음과 같은 데이터 집합에 대해 간단한 통계를 계산하는 함수를 제공한다. 이러한 값들을 통틀어 기술 통계(descriptive statistics)라고 한다.

데이터의 개수(count), 평균(mean, average)
분산(variance), 표준 편차(standard deviation)
최댓값(maximum), 최솟값(minimum)
중앙값(median), 사분위수(quartile)

예를 들어 다음과 같은 데이터 x가 있다고 하자.

x={18,5,10,23,19,−8,10,0,0,5,2,15,8,2,5,4,15,−1,4,−7,−24,7,9,−6,23,−13}

x를 이루는 숫자 하나하나를 수학 기호로는 x1,x2,⋯,xN 처럼 표시한다. 위 예에서 x1=18 , x2=5 이다.
넘파이에서는 이러한 데이터를 1차원 배열로 구현한다.

```
x = np.array([18,   5,  10,  23,  19,  -8,  10,   0,   0,   5,   2,  15,   8,
               2,   5,   4,  15,  -1,   4,  -7, -24,   7,   9,  -6,  23, -13])a
```

데이터의 개수

데이터의 개수는 len 명령으로 구할 수 있다.

```
len(x)  # 갯수
```

표본 평균

평균을 통계용어로는 표본 평균(sample average, sample mean)이라고 한다. x 데이터에 대한 표본 평균은 $\bar{x}$ 라고 표시하며 다음과 같이 계산한다. 이 식에서 N 은 데이터의 개수이다.

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

```
np.mean(x)  # 평균
```

표본 분산

표본 분산(sample variance)은 데이터와 표본 평균간의 거리의 제곱의 평균이다. 표본 분산이 작으면 데이터가 모여있는 것이고 크면 흩어져 있는 것이다. 수학 기호로는 s^2이라고 표시하며 다음과 같이 계산한다.

$$s^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

```
np.var(x)   # 분산

np.var(x, ddof=1)   # 비편향 분산
```

표본 표준편차

표본 표준편차(sample standard variance)는 표본 분산의 양의 제곱근 값이다. s라고 표시한다.

$$s = \sqrt{s^2}$$

```
np.std(x)  # 표준 편차
```

최댓값과 최솟값

최댓값(maximum)은 데이터 중에서 가장 큰 값을, 최솟값(minimum)은 가장 작은 값을 의미한다.

```
np.max(x)  # 최댓값

np.min(x)  # 최솟값
```

중앙값

중앙값(median)은 데이터를 크기대로 정렬하였을 때 가장 가운데에 있는 수를 말한다. 만약 데이터의 수가 짝수이면 가장 가운데에 있는 두 수의 평균을 사용한다.

```
np.median(x)  # 중앙값
```

사분위수

사분위수(quartile)는 데이터를 가장 작은 수부터 가장 큰 수까지 크기가 커지는 순서대로 정렬하였을 때 1/4, 2/4, 3/4 위치에 있는 수를 말한다. 각각 1사분위수, 2사분위수, 3사분위수라고 한다. 1/4의 위치란 전체 데이터의 수가 만약 100개이면 25번째 순서, 즉 하위 25%를 말한다. 따라서 2사분위수는 중앙값과 같다.
때로는 위치를 1/100 단위로 나눈 백분위수(percentile)을 사용하기도 한다. 1사분위수는 25% 백분위수와 같다.

```
np.percentile(x, 0)   # 최소값
np.percentile(x, 25)   # 1사분위 수
np.percentile(x, 50)   # 2사분위 수
np.percentile(x, 75)   # 3사분위 수
np.percentile(x, 100)   # 최댓값
```

난수 발생과 카운팅 – 시드 설정하기

파이썬을 이용하여 데이터를 무작위로 섞거나 임의의 수 즉, 난수(random number)를 발생시키는 방법에 대해 알아본다. 이 기능은 주로 NumPy의 random 서브패키지에서 제공한다.
컴퓨터 프로그램에서 발생하는 무작위 수는 사실 엄격한 의미의 무작위 수가 아니다. 어떤 특정한 시작 숫자를 정해 주면 컴퓨터가 정해진 알고리즘에 의해 마치 난수처럼 보이는 수열을 생성한다. 이런 시작 숫자를 시드(seed)라고 한다. 일단 생성된 난수는 다음번 난수 생성을 위한 시드값이 된다. 따라서 시드값은 한 번만 정해주면 된다. 시드는 보통 현재 시각등을 이용하여 자동으로 정해지지만 사람이 수동으로 설정할 수도 있다. 특정한 시드값이 사용되면 그 다음에 만들어지는 난수들은 모두 예측할 수 있다. 이 책에서는 코드의 결과를 재현하기 위해 항상 시드를 설정한다.

파이썬에서 시드를 설정하는 함수는 seed이다. 인수로는 0과 같거나 큰 정수를 넣어준다.

```
np.random.seed(0)
```

이렇게 시드를 설정한 후 넘파이 random 서브패키지에 있는 rand 함수로 5개의 난수를 생성해 보자. rand 함수는 0과 1사이의 난수를 발생시키는 함수로 인수로 받은 숫자 횟수만큼 난수를 발생시킨다.

```
np.random.rand(5)
```

시드 설정하기

몇번 더 난수를 생성해보자. 사람이 예측할 수 없는 무작위 숫자가 나오는 것을 볼 수 있다.

```
np.random.rand(10)
np.random.rand(10)
```

이제 시드를 0으로 재설정하고 다시 난수를 발생시켜 본다.

```
np.random.seed(0)
np.random.rand(5)
np.random.rand(10)
np.random.rand(10)
```

아까와 같은 숫자가 나오는 것을 확인할 수 있다.

데이터의 순서 바꾸기

데이터의 순서를 바꾸려면 shuffle 함수를 사용한다. shuffle 함수도 자체 변환(in-place) 함수로 한 번 사용하면 변수의 값이 바뀌므로 사용에 주의해야 한다.

```
x = np.arange(10)
x

np.random.shuffle(x)
x
```

데이터 샘플링

이미 있는 데이터 집합에서 일부를 무작위로 선택하는 것을 표본선택 혹은 샘플링(sampling)이라고 한다. 샘플링에는 choice 함수를 사용한다. choice 함수는 다음과 같은 인수를 가질 수 있다.

numpy.random.choice(a, size=None, replace=True, p=None)

a : 배열이면 원래의 데이터, 정수이면 arange(a) 명령으로 데이터 생성
size : 정수. 샘플 숫자
replace : 불리언. True이면 한번 선택한 데이터를 다시 선택 가능
p : 배열. 각 데이터가 선택될 수 있는 확률

```
np.random.choice(5, 5, replace=False)  # shuffle 명령과 같다.

np.random.choice(5, 3, replace=False)  # 3개만 선택

np.random.choice(5, 10)  # 반복해서 10개 선택

np.random.choice(5, 10, p=[0.1, 0, 0.3, 0.6, 0])  # 선택 확률을 다르게 해서 10개 선택
```

난수 생성

넘파이의 random 서브패키지는 이외에도 난수를 생성하는 다양한 함수를 제공한다. 그 중 가장 간단하고 많이 사용되는 것은 다음 3가지 함수다.

rand: 0부터 1사이의 균일 분포
randn: 표준 정규 분포
randint: 균일 분포의 정수 난수

rand 함수는 0부터 1사이에서 균일한 확률 분포로 실수 난수를 생성한다. 숫자 인수는 생성할 난수의 크기이다. 여러개의 인수를 넣으면 해당 크기를 가진 행렬을 생성한다.

```
np.random.rand(10)

np.random.rand(3, 5)
```

난수 생성

randn 함수는 기댓값이 0이고 표준편차가 1인 표준 정규 분포(standard normal distribution)를 따르는 난수를 생성한다. 인수 사용법은 rand 명령과 같다.

```
np.random.randn(10)

np.random.randn(3, 5)
```

randint 함수는 다음과 같은 인수를 가진다.

numpy.random.randint(low, high=None, size=None)

만약 high를 입력하지 않으면 0과 low사이의 숫자를, high를 입력하면 low와 high는 사이의 숫자를 출력한다. size는 난수의 숫자이다

```
np.random.randint(10, size=10)
np.random.randint(10, 20, size=10)
np.random.randint(10, 20, size=(3, 5))
```

연습 문제

연습 문제 3.5.1

1. 동전을 10번 던져 앞면(숫자 1)과 뒷면(숫자 0)이 나오는 가상 실험을 파이썬으로 작성한다.
2. 주사위를 100번 던져서 나오는 숫자의 평균을 구하라.

연습 문제 3.5.2

가격이 10,000원인 주식이 있다. 이 주식의 일간 수익률(%)은 기댓값이 0%이고 표준편차가 1%인 표준 정규 분포를 따른다고 하자. 250일 동안의 주가를 무작위로 생성하라

정수 데이터 카운팅

이렇게 발생시킨 난수가 실수값이면 히스토그램 등을 사용하여 분석하면 된다. 히스토그램을 시각화 부분에서 나중에 자세히 설명한다.

만약 난수가 정수값이면 unique 명령이나 bincount 명령으로 데이터 값을 분석할 수 있다.

unique 함수는 데이터에서 중복된 값을 제거하고 중복되지 않는 값의 리스트를 출력한다. return_counts 인수를 True 로 설정하면 각 값을 가진 데이터 갯수도 출력한다.

```
np.unique([11, 11, 2, 2, 34, 34])

a = np.array(['a', 'b', 'b', 'c', 'a'])
index, count = np.unique(a, return_counts=True)

index

count
```

정수 데이터 카운팅

그러나 unique 함수는 데이터에 존재하는 값에 대해서만 갯수를 세므로 데이터 값이 나올 수 있음에도 불구하고 데이터가 하나도 없는 경우에는 정보를 주지 않는다. 예를 들어 주사위를 10번 던졌는데 6이 한 번도 나오지 않으면 이 값을 0으로 세어주지 않는다.

따라서 데이터가 주사위를 던졌을 때 나오는 수처럼 특정 범위안의 수인 경우에는 bincount 함수에 minlength 인수를 설정하여 쓰는 것이 더 편리하다. bincount 함수는 0 부터 minlength - 1 까지의 숫자에 대해 각각 카운트를 한다. 데이터가 없을 경우에는 카운트 값이 0이 된다.

```
np.bincount([1, 1, 2, 2, 2, 3], minlength=6)
```