대규모 다차원 배열/수학 연산 처리

Numpy

AI

AI

Numpy의 특징

- 과학계산에 적합
- 고정 타입 배열
- 다차원 배열(N-dimensional array)
- C 로 최적화한 매우 효율적인 라이브러리
- UFunc(유니버설 함수)
- Broadcasting
- AI 프레임워크인 Pytorch와 호환성

AI

리스트로 부터 배열 생성

› 정수 배열 생성

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

› NumPy는 배열의 모든 요소가 같은 타입 이여야함, 타입이 일치 하지 가능한 경우 상위 타입을 취함

```
np.array([3.14, 4, 2, 3])
# array([3.14, 4. , 2. , 3. ])
```

AI

리스트로 부터 배열 생성

› 명시적으로 타입을 설정하려면 dtype 키워드를 사용

```
np.array([1, 2, 3, 4], dtype='float32')
# array([1., 2., 3., 4.], dtype=float32)
```

› 리스트를 중첩하면 다차원 배열을 만들 수 있음

```
np.array([range(i, i + 3) for i in [2, 4, 6]])
# array([[2, 3, 4],
#        [4, 5, 6],
#        [6, 7, 8]])
```

AI

직접 배열 생성

› 0으로 채운 길이 10의 정수 배열 생성

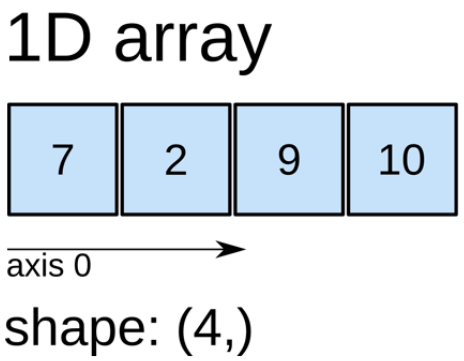
```
np.zeros(10, dtype=int)
# array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

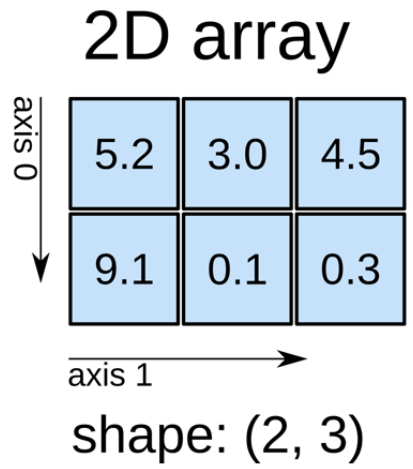
› 1로 채운 3x5 부동 소수점 배열 생성

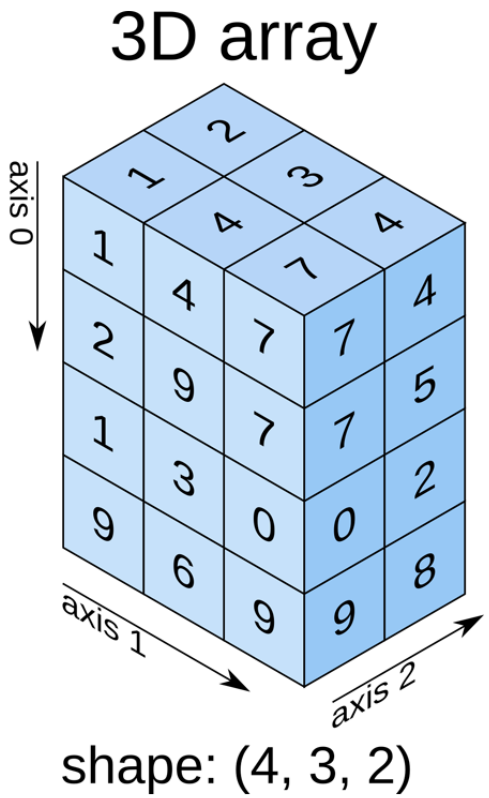
```
np.ones((3, 5), dtype=float)
# array([[1., 1., 1., 1., 1.],
#        [1., 1., 1., 1., 1.],
#        [1., 1., 1., 1., 1.]])
```

AI

다차원 배열과 차원 축







- 차원의 수에 따라 n-D array 로 분류
- 각 차원의 크기는 shape (n, m, ...)로 표시
- 각 요소는 모두 동일한 데이터 타입
- 2-D array 이상에서의 축 방향에 주의

AI

배열 속성

```
x3 = np.random.randint(10, size=(3, 4, 5)) # 3차원 배열
print("ndim", x3.ndim)    # ndim(랭크, 차원의 개수)
# ndim 3
print("shape", x3.shape) # shape(형상, 각 차원의 크기)
# shape (3, 4, 5)
print("size", x3.size)        # size(전체 배열 크기)
# size 60
print("dtype", x3.dtype) # dtype(배열의 데이터 타입)
# dtype int32
```

AI

배열 생성

함수	설명
np.array(list)	파이썬 리스트로 부터 배열 생성
np.zeros(shape)	0으로 초기화된 배열 생성
np.ones(shape)	1으로 초기화된 배열 생성
np.full(shape, value)	특정 값으로 초기화된 배열 생성
np.arange(v1, v2, step)	특정 범위 값을 가지는 배열 생성
np.linspace(v1, v2, split)	특정 범위 값을 가지는 배열 생성
np.empty(shape)	초기화되지 않은 배열 생성

AI

배열 생성

함수	설명
np.random .random(shape)	0과 1 사이의 난수로 초기화된 배열 생성
np.random .normal(mean, dev, shape)	정규 분포의 난수로 초기화된 배열 생성
np.random .randint(s, e, size=shape)	[s, e] 구간의 임의의 정수로 초기화된 배열 생성
np.eye(n)	n x n 크기의 단위 행렬 생성

AI

데이터 타입

타입(dtype)	설명
np.bool_	1바이트로 저장된 불 값(참 또는 거짓)
np.int16	-32,768 ~ 32,767
np.int32	-2,147,483,648 ~ 2,147,483,647
np.uin16	0 ~ 65,535
np.uint32	0 ~ 4,294,967,295
np.float32	3.4E+/-38(7개의 자릿수)
np.float63	1.7E+/-308(15개의 자릿수)
np.complex64	복소수, 두개의 float32으로 표현

AI

1-D 배열 인덱싱

› 파이썬 리스트와 마찬가지로 1차원 배열에 꺾쇠괄호로 인덱스를 지정 (인덱스는 0부터 시작)
끝에서 부터 인덱싱하려면 음수 인덱스를 사용

```
› x1 = np.random.randint(10,size=6)
  # [3 4 6 9 6 6]
  print(x1[0])
  # 3
  print(x1[-1])
  # 6
```

AI

n-D 배열 인덱싱

› 다차원 배열에서는 콤마로 구분된 인덱스를 사용.
파이썬 리스트와 달리 NumPy 배열을 고정 타입을
가지므로 다른 타입의 데이터 할당시 형변환에 주의

```
› x2 = np.random.randint(10, size=(3, 4))
  x2[0, 0] = 3.14159
  print(x2)
  # [[3 1 0 5]
  #  [3 7 0 5]
  #  [5 5 7 3]]
```

AI

배열 슬라이싱: 하위 배열에 접근

› 부분 배열에 접근하는 슬라이싱 구문은 파이썬과 동일
arr[start = 0 : stop = 차원 크기 : step=1]

```
› print(x[:5])  # 첫 다섯 개 요소로 구성된 배열
  print(x[5:])  # 인덱스 5 다음 요소로 구성된 배열
  print(x[4:7]) # 중간 하위 배열
  print(x[::2])  # 하나 걸러 하나씩의 요소로 구성된 배열
  print(x[1::2]) # 인덱스 1에서 시작해 하나 걸러 하나씩
  요소로 구성된 배열
```

AI

다차원 배열 슬라이싱: 하위 배열에 접근

› 다차원 슬라이싱도 콤마로 구분된 다중 슬라이스를 사용해 똑같은 방식으로 동작함

```
› print(x2)
  # [[9 0 2 0] [0 6 0 1] [3 9 1 3]]
  print(x2[:2, :3]) # 두 개의 행, 세 개의 열
  # [[9 0 2] [0 6 0]]
  print(x2[:3, ::2]) # 모든 행, 한 열 걸러 하나씩
  # [[9 2] [0 0] [3 1]]
```

AI

다차원 배열 슬라이싱: 하위 배열에 접근

› 배열의 행과 열의 접근은 인덱싱과 빈 슬라이싱을 결합하여 가능함, 행에 접근하는 경우 마지막 빈 슬라이스는 생략 가능

```
› print(x2)
  #[[2 4 4 5] [7 3 3 6] [3 3 7 8]]
  print(x2[:, 0])
  # [7 3 3 6]
  print(x2[0, : ]) # x2[0]와 동일
  #[2 4 4 5]
```

AI

슬라이싱: 사본과 뷰

› 슬라이싱하면 사본(copy)가 아닌 뷰(view)를 반환하여 불필요한 오버헤드를 줄임, 사본이 필요시 명시적으로 .copy() 메서드를 이용

```
› x2 = np.array([[6, 4, 1, 0], [8, 8, 1, 3], [5, 9, 1, 1]])
  # [[6 4 1 0] [ 8 8 1 3] [ 5 9 1 1]]
  x2_sub = x2[:2, :2]
  x2_sub[0, 0] = 99 # 뷰에 대한 수정은 원래 배열의 변경
  print(x2_sub) # [[99 4] [ 8 8]]
  print(x2)       # [[99 4 1 0] [ 8 8 1 3] [ 5 9 1 1]]
```

AI

배열 재구조화

› 배열의 차원 변경은 reshape() 또는 슬라이스 내에 newaxis 키워드를 사용함, 가령 3x3 그리드에서 숫자 1~9를 넣고자 한다면 다음과 같음

```
› grid = np.arange(1, 10).reshape((3, 3))
  print(grid)
  # [[1 2 3] [4 5 6] [7 8 9]]

› column = np.array([1, 2, 3])[ : , np.newaxis] # 열 벡터
  print(column)
  # [[1] [2] [3]]
```

AI

배열 연결

› np.concatenate를 이용해 동일한 차원의 배열들을 연결, 지정한 Axis 이외에 동일한 형상이여야함

```
› x, y = np.array([1, 2, 3]), np.array([4, 5, 6]) shape=(3,)
  print(np.concatenate([x, y] , axis=0))
  # [1 2 3 4 5 6] shape=(6,)
```

› np.concatenate는 2차원 배열에서도 사용할 수 있다.

```
› grid = np.array([[1, 2, 3], [4, 5, 6]]) shape=(2, 3)
  print(np.concatenate([grid, grid], axis=1))
  #[[1 2 3 1 2 3] [4 5 6 4 5 6]] shape=(2, 6)
```

AI

배열 연결

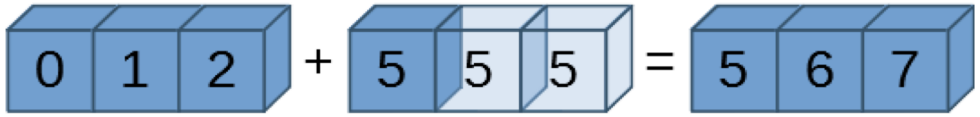
› 3d-array 이하의 혼합된 차원의 배열들을연결할때는 np.vstack, hstack ,dstack이 더 명확하고 간편함

```
› grid = np.array([[1, 2, 3], [4, 5, 6]]) #shape=(2, 3)
  v = np.array([7, 8, 9])                  #shape=(3,)
  print(np.vstack([v, grid]))
  # [[7 8 9] [1 2 3] [4 5 6]] shape=(3, 3)
  print(np.concatenate([v[np.newaxis, :], grid], axis=0))
  # [[7 8 9] [1 2 3] [4 5 6]] shape=(3, 3)
```

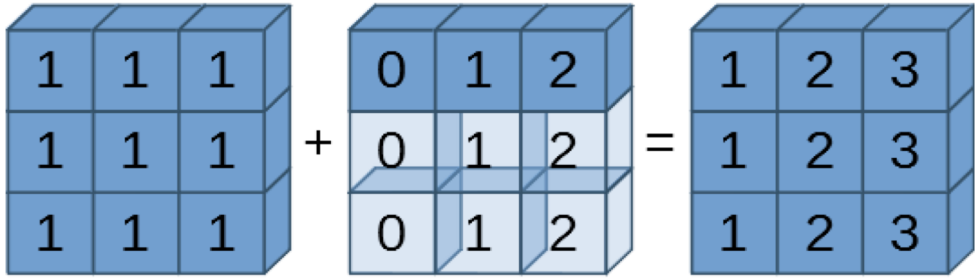
AI

브로드캐스팅

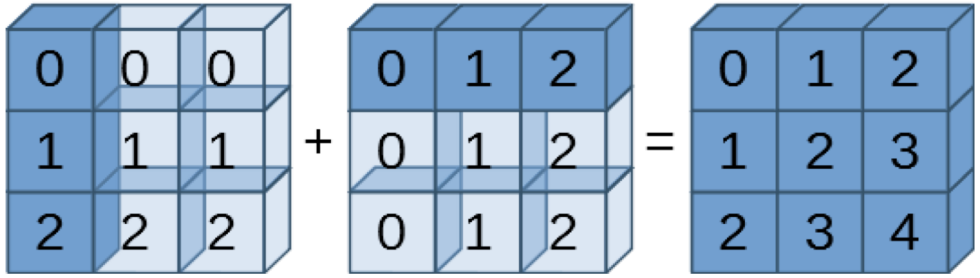
np.arange(3) + 5



np.ones((3, 3)) + np.arange(3)



np.ones((3, 1)) + np.arange(3)



- 규칙 1: 차원 수가 다르면 작은 쪽 배열의 왼쪽 차원을 1로 채워서 맞춘다
- 규칙 2: 각 차원의 크기(형상)가 다르면 차원의 크기가 1인 차원의 크기를 일치하도록 늘린다
- 규칙 3: 임의의 차원에서 크기가 일치하지 않고 1도 아니라면 오류를 발생한다

AI

UFuncs: 유니버설 함수

표준 연산자	대응 UFunc
+	np.add
-	np.subtract
-(음수)	np.negative
*	np.multiply
/	np.divide
//	np.floor_divide
**	np.power
%	np.mod

- 유니버설 함수는 배열의 요소에 대한 반복되는 작업을 NumPy의 기저를 이루는 컴파일된 계층으로 밀어넣음으로써 훨씬 빠르게 실행되도록 설계됨
- UFuncs에는 파이썬의 표준 연산자를 오버라이딩하여 구현

AI

UFuncs: 유니버설 함수

연산자	대응 UFunc
&	np.biwise_and
|	np.biwise_or
^	np.biwise_xor
~	np.biwise_not

연산자	대응 UFunc
==	np.equal
!=	np.not_equal
<	np.less
<=	np.less_equal
>	np.greater
>=	np.greater_equal

AI

UFuncs: 유니버설 함수

함수	설명
np.abs(x)	절댓값
np.sin(x)	삼각함수 sin
np.cos(x)	삼각함수 cos
np.tan(x)	삼각함수 tan
np.arcsin(x)	역삼각함수 sin
np.arccos(x)	역삼각함수 cos
np.arctan(x)	역삼각함수 tan

함수	설명
np.exp(x)	지수 e^x
np.exp2(x)	지수 2^x
np.power(n, x)	지수 n^x
np.log(x)	자연 로그
np.log2(x)	base 2 로그
np.log10(x)	상용 로그

AI

UFuncs: 유니버설 함수

함수	설명
np.sum	요소의 합
np.prod	요소의 곱
np.mean	평균
np.std	표준 편차
np.var	분산
np.min	최솟값
np.max	최댓값

함수	설명
np.argmin	최솟값의 인덱스
np.argmax	최댓값의 인덱스
np.median	중앙값
np.percentile	순위 기반 백분위 수
np.any	True 존재 ?
np.all	모두 True ?

AI

UFuncs: 시간 성능 비교

```
› def compute_reciprocals(values):
      output = np.empty(len(values))
      for i in range(len(values)):
          output[i] = 1.0 / values[i]
      return output

› big_array = np.random.randint(1, 100, size=1000000)

› timeit(lambda : compute_reciprocals(big_array))
  # 수행 시간 2.35006285 secs
  timeit(lambda : 1.0 / big_array)
  # 수행 시간 0.01562691 secs
```

AI

UFuncs: 표준 연산 함수

› 유니버설 함수는 배열의 각 요소에 반복적으로 적용

```
› x = np.arange(4)
› print(x)       # [0 1 2 3]
› print(x + 5) # [5 6 7 8]
› print(x - 5)  # [-5 -4 -3 -2]
› print(x * 2)  # [0 2 4 6]
› print(x / 2)  # [0. 0.5 1. 1.5]
› print(x // 2) # [0 0 1 1]
```

AI

UFuncs: 연산 출력지정

› 함수의 out 인수를 사용해 출력을 지정할 수 있음
임시 배열의 생성과 복사로 인한 오버헤드 줄임

```
› x, y = np.arange(5), np.empty(5)
  np.multiply(x, 10, out=y)
  print(y)  # [ 0. 10. 20. 30. 40.]
```

```
› y = np.zeros(10)
  np.power(2, x, out=y[::2]) # 배열 뷰 가능
  print(y)  # [ 1. 0. 2. 0. 4. 0. 8. 0. 16. 0.]
```

AI

UFuncs: 축 방향 연산

› 일부 함수는 연산을 수행할 축(axis)을 지정하여
해당 축을 축소하는 방향으로 연산을 수행

```
› M = np.random.randint(10, size=(3, 4))
  # 전체 요소 [[5 1 6 9] [5 6 6 6] [0 7 9 6]]
› print(M.sum())          # 전체 요소의 합 66
› print(M.min(axis=0)) # 각 열의 최소 값 [0 1 6 6]
› print(M.max(axis=1)) # 각 행의 최댓 값 [9 6 9]
```

AI

마스킹 연산

› 조건 연산의 결과인 부울 배열을 마스크를 이용하여 True에 해당되는 값들을 1차원 배열로 구성

```
› x = np.random.randint(10, size=(3, 4))
› print(x)          # [[8 5 0 5] [2 8 0 1] [6 5 8 6]]
› print(x < 5)      # [[F F T F] [ T F T T] [F F F F]]
› print(x[x < 5]) # [0 2 0 1]
```

AI

팬시 인덱싱

› 단일 스칼라 대신 인덱스 배열을 전달하여 접근
결과의 형상은 인덱스 배열의 형상을 반영

```
› print([x[3], x[7], x[4], x[5]])
  # [99, 43, 15, 83]

› ind = [3, 7, 4, 5]
  print(x[ind])  # [99 43 15 83]

› ind = np.array([[3, 7], [4, 5]])
  print(x[ind])  # array([[99, 43], [15, 83]])
```

AI

팬시 인덱싱

› 팬시 인덱싱도 여러 차원을 지정 할 수 있으며(행, 열, ...)
이때 인덱스 쌍을 만드는 것은 브로드캐스팅 규칙을 따름

```
› x = np.arange(12).reshape((3, 4))
  print(x) # [[ 0 1 2 3] [ 4 5 6 7] [ 8 9 10 11]]

› row, col = np.array([0, 1, 2]), np.array([2, 1, 3])
  print(x[row, col])
  # [ 2 5 11]

› print(x[row[:, np.newaxis], col])
  # [[ 2 1 3] [ 6 5 7] [10 9 11]]
```