부록 A. numpy 라이브러리: 파이썬의 데이터 표현과 가공

인공지능을 구현하는 주요 기술인 기계 학습에서는 수학을 많이 사용하는데 선형 대수linear algebra, 확률 통계probability and statistics, 최적화 이론optimization theory을 주로 사용한다. 선형 대수는 데이터를 표현하고 변환하는데 사용하고 확률 통계는 데이터가 지닌 불확실성을 처리하는데 사용하며 최적화는 손실 함수의 최저점을 찾아야 하는 학습 과정에서 주로 사용한다. 이러한 수학적 개념은 학습에 부담일 수 있는데, 수학 때문에 주눅이 들거나 지레 포기할 필요는 없다. 깊이 있는 수학 개념을 필요로 하는 사람은 새로운 기계 학습 알고리즘을 개발하거나 기계 학습의 현상을 새로운 시각으로 해석하는 연구자다. 이 책이 겨냥하는 독자층은 이미 습득한 수학 실력을 상기하고 프로그래밍을 통해 기계 학습에 적용하는 정도면 된다.

먼저 데이터를 간결하게 표현하고 다른 형태로 변환하는 연산을 지원하는 선형 대수를 설명한다. 선형 대수는 대학의 한 학기 강의 분량의 결코 가볍지 않은 내용을 담고 있는데, 이 책은 벡터와 행렬, 합과 곱, 전치 행렬 정도의 아주 낮은 수준의 내용만 다룬다. 다행스럽게도 이 정도만 이해해도 이 책을 끝까지 공부하는데 지장이 없다. 중요한 점은 다차원 배열, 즉 텐서의 구조를 이해하고 프로그램으로 구현할 수 있는 실력을 갖추는 것이다. 따라서 행렬과 텐서의 기본 개념과 연산을 간략히 설명한 다음 파이썬으로 프로그래밍하는 연습을 집중적으로 한다.

A.1 벡터와 행렬, 텐서

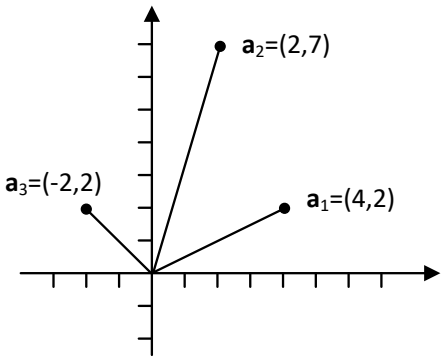
선형 대수는 벡터vector와 행렬matrix을 중요하게 다룬다. 먼저 벡터와 행렬을 어떻게 표현하는지 소개하고 벡터와 행렬에 적용하는 연산을 설명한다. 또한 벡터와 행렬을 텐서tensor로 확장하고, 텐서가 기계 학습이 처리하는 데이터와 어떤 연관성이 있는지 설명한다.

벡터와 행렬의 표현

벡터 $\mathbf{a}$는 식 (A.1)로 표현한다. 벡터는 굵은 체로 표기하고, 벡터를 구성하는 요소는 이탤릭 체로 표기하며 아래 첨자를 붙여 요소끼리 구분한다. d는 벡터를 구성하는 요소의 개수인데, 벡터의 차원dimension이라 부른다. 수학에서는 벡터가 d차원의 한 점이라는 뜻에서 $\mathbf{a} \in \mathbb{R}^d$로 표현한다. $\mathbb{R}^d$은 축이 d개인 실수 공간이다.

$$\mathbf{a} = (a_1 \;\; a_2 \;\; \cdots \;\; a_d) \qquad \text{(A.1)}$$

[그림 A-1]은 d=2인 경우의 예를 보여준다. 벡터가 여러 개면 벡터에 아래 첨자를 붙여 $\mathbf{a}_1$과 $\mathbf{a}_2$처럼 구분하거나 $\mathbf{a}$와 $\mathbf{b}$처럼 구분한다. 상황에 따라 편리한 방식을 사용하면 된다.



[그림 A-1] 2차원 벡터

행렬은 여러 개의 벡터를 한꺼번에 담는다. [그림 A-1]에 있는 세 개의 2차원 벡터를 행렬에 담아 표현하면 $\mathbf{A} = \begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{pmatrix} = \begin{pmatrix} 4 & 2 \\ 2 & 7 \\ -2 & 2 \end{pmatrix}$이다.

식 (A.2)는 행렬을 표현한다. 행렬은 대문자로 표기하여 벡터와 구분한다. 행렬은 행row과 열column로 구성되는데, r은 행의 개수이고 c는 열의 개수이다. $\mathbf{A}$는 r*c 행렬이라고 말한다. 행렬 $\mathbf{A}$의 요소 a_{ij}는 i번째 행의 j번째 열에 있는 값이다. i번째 행 $\mathbf{a}_i = (a_{i1}a_{i2}\cdots a_{ic})$는 행렬을 구성하는 행 벡터row vector이다.

[TIP] r*c는 r by c라고 읽는다.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1c} \\ a_{21} & a_{22} & \cdots & a_{2c} \\ \vdots & \vdots & \ddots & \vdots \\ a_{r1} & a_{r1} & \cdots & a_{rc} \end{pmatrix} = \begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \vdots \\ \mathbf{a}_r \end{pmatrix} \quad \text{(A.2)}$$

기계 학습에서 벡터와 행렬

기계 학습은 데이터를 다루는 학문이다. 현대 인공지능은 기계 학습으로 구현되기 때문에 인공지능도 데이터를 다루는 학문이라고 말할 수 있다. 가장 단순한 축에 속하는 데이터인 iris를 가지고 행렬 표현을 설명한다. 세 줄로 구성된 다음 코드를 실행해보자.[1]

```
> from sklearn import datasets
> d=datasets.load_iris()
> print(d.data)
[[5.5 3.5 1.4 0.2]
 [4.9 3.   1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.   3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.   3.4 1.5 0.2]
…
…
 [5.9 3.0 5.1 1.8]]
```

iris 데이터셋은 150개 샘플로 구성되는데 각각의 샘플은 4개 특징으로 표현된다. 다시 말해 iris는 4차원 특징 벡터 150개로 구성된다. 앞의 코드가 출력한 샘플을 식 (A.1)에 맞추어 표현하면 다음과 같다.

$$\begin{aligned} &\text{iris의 첫번째 샘플의 특징 벡터: } \mathbf{a}_1 = (5.5 \quad 3.5 \quad 1.4 \quad 0.2) \\ &\text{iris의 두번째 샘플의 특징 벡터: } \mathbf{a}_2 = (4.9 \quad 3.0 \quad 1.4 \quad 0.2) \\ &\text{iris의 세번째 샘플의 특징 벡터: } \mathbf{a}_3 = (4.7 \quad 3.2 \quad 1.3 \quad 0.2) \\ &\cdots \\ &\text{iris의 150번째 샘플의 특징 벡터: } \mathbf{a}_{150} = (5.9 \quad 3.0 \quad 5.1 \quad 1.8) \end{aligned}$$

전체 샘플을 행렬에 담으면 행의 개수는 150, 열의 개수는 4인 150*4 행렬이 된다.

[1] 책 3.1.1항을 참조한다.

$$\mathbf{A} = \begin{pmatrix} 5.5 & 3.5 & 1.4 & 0.2 \\ 4.9 & 3.0 & 1.4 & 0.2 \\ 4.7 & 3.2 & 1.3 & 0.2 \\ \vdots & \vdots & \vdots & \vdots \\ 5.9 & 3.0 & 5.1 & 1.8 \end{pmatrix} = \begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \\ \vdots \\ \mathbf{a}_{150} \end{pmatrix}$$

연산

스칼라에 덧셈과 곱셈, 역수 등의 연산을 수행하는 것처럼 벡터와 행렬에도 비슷한 연산을 적용할 수 있다. [그림 A-1]의 벡터 $\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{a}_3$으로 벡터 연산을 설명한다.

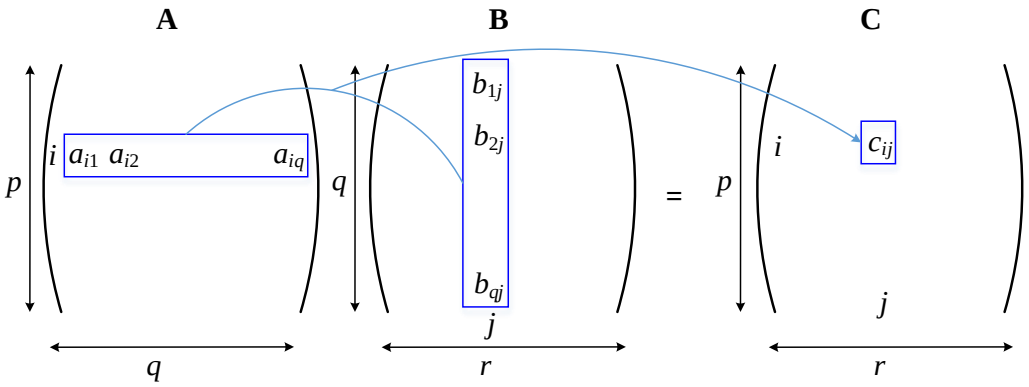
- 벡터에 스칼라 곱: $3 * \mathbf{a}_1 = 3 * (4 \quad 2) = (12 \quad 6)$
- 벡터 덧셈: $\mathbf{a}_1 + \mathbf{a}_2 = (4 \quad 2) + (2 \quad 7) = (6 \quad 9)$
- 벡터의 요소별 곱: $\mathbf{a}_1 \cdot \mathbf{a}_2 = (4 \quad 2) \cdot (2 \quad 7) = (8 \quad 14)$

행렬도 여러 유용한 연산을 제공한다. 행렬에 스칼라를 곱하는 연산은 행렬 요소 각각에 스칼라를 곱하면 된다. 행렬의 덧셈은 두 행렬의 크기가 같을 때만 가능하며 같은 위치에 있는 요소끼리 더하면 된다.

행렬의 전치transpose는 열과 행의 위치를 바꾸는 연산이다. **A**의 i번째 행의 j번째 열에 있는 요소가 a_{ij}라면 **A**의 전치 행렬에서는 a_{ij}가 j번째 행의 i번째 열에 위치한다. **A**가 $p*q$라면 **A**의 전치 행렬은 $q*p$이고 $\mathbf{A}^{\mathrm{T}}$로 표기한다.

행렬의 곱셈은 조금 복잡하다. 행렬 **A**와 행렬 **B**를 곱하는 연산 **AB**는 **A**의 열의 개수와 **B**의 행의 개수가 같은 경우에만 가능하다. 다시 말해 **A**가 $p*q$이고 **B**가 $q*r$이면 두 행렬을 곱할 수 있다. 예를 들어 3*2 행렬과 2*3 행렬은 곱할 수 있지만 3*2 행렬과 3*2 행렬은 곱할 수 없다. [그림 A-2]는 행렬 **A**와 **B**를 곱하여 결과를 **C**에 저장하는 과정을 보여준다. **A**는 $p*q$이고 **B**는 $q*r$이라 가정하고 [그림 A-2]는 **C**의 요소 c_{ij}를 계산하는 과정을 설명한다. c_{ij}는 **A**의 i번째 행과 **B**의 j번째 열을 요소끼리 곱하고 결과를 더한 값이다. 이를 공식으로 쓰면 식 (A.3)이다.

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{iq}b_{qj} \quad \text{(A.3)}$$



[그림 A-2] 행렬의 곱셈

아래에 있는 세 행렬 **A**, **B**, **C**를 가지고 행렬 연산을 설명한다. **A**는 3*2, **B**는 3*2, **C**는 2*3 행렬이다.

$$\mathbf{A} = \begin{pmatrix} 4 & 2 \\ 2 & 7 \\ -2 & 1 \end{pmatrix}, \mathbf{B} = \begin{pmatrix} 0 & 1 \\ 2 & 1 \\ 3 & -2 \end{pmatrix}, \mathbf{C} = \begin{pmatrix} 1 & -2 & 3 \\ 5 & 0 & 2 \end{pmatrix}$$

- 행렬에 스칼라 곱: $3 * \mathbf{A} = 3 * \begin{pmatrix} 4 & 2 \\ 2 & 7 \\ -2 & 1 \end{pmatrix} = \begin{pmatrix} 12 & 6 \\ 6 & 21 \\ -6 & 3 \end{pmatrix}$
- 행렬의 덧셈: $\mathbf{A} + \mathbf{B} = \begin{pmatrix} 4 & 2 \\ 2 & 7 \\ -2 & 1 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ 2 & 1 \\ 3 & -2 \end{pmatrix} = \begin{pmatrix} 4 & 3 \\ 4 & 8 \\ 1 & -1 \end{pmatrix}$
- 행렬의 덧셈: $\mathbf{A} + \mathbf{C}$는 크기가 달라 덧셈 불가함
- 전치 행렬: **A**의 전치 행렬 $\mathbf{A}^{\mathrm{T}} = \begin{pmatrix} 4 & 2 & -2 \\ 2 & 7 & 1 \end{pmatrix}$
- 행렬의 곱셈: $\mathbf{AB} = \begin{pmatrix} 4 & 2 \\ 2 & 7 \\ -2 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 2 & 1 \\ 3 & -2 \end{pmatrix}$는 크기가 맞지 않아 곱셈 불가함
- 행렬의 곱셈: $\mathbf{AC} = \begin{pmatrix} 4 & 2 \\ 2 & 7 \\ -2 & 1 \end{pmatrix} \begin{pmatrix} 1 & -2 & 3 \\ 5 & 0 & 2 \end{pmatrix} = \begin{pmatrix} 14 & -8 & 16 \\ 37 & -4 & 20 \\ 3 & 4 & -4 \end{pmatrix}$

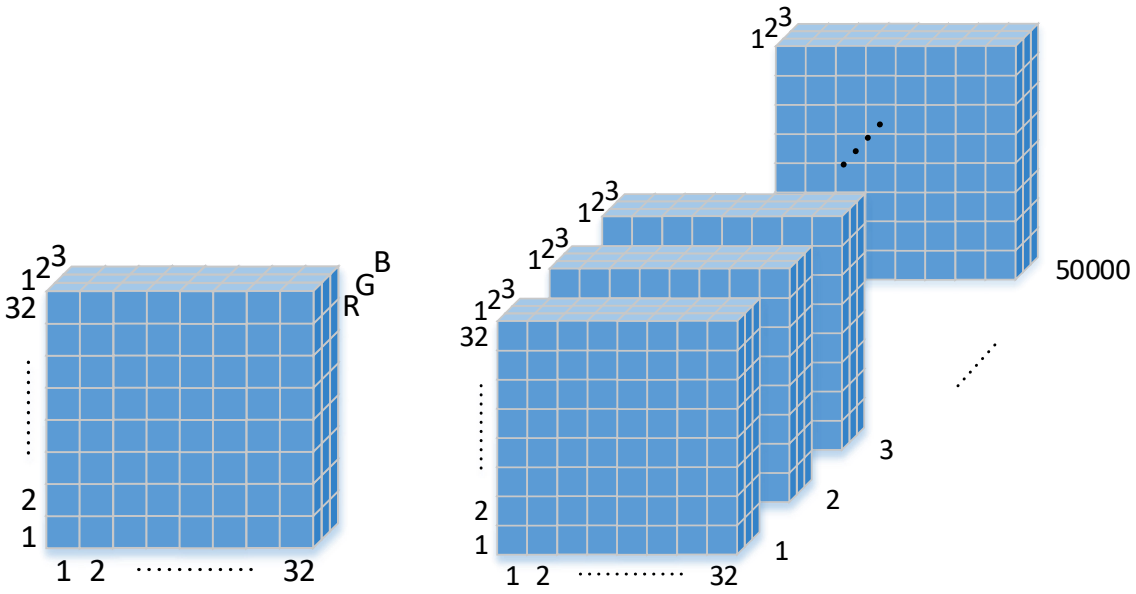
텐서

선형 대수는 벡터와 행렬에 대한 더욱 복잡한 연산을 다룬다. 행렬의 역행렬inverse matrix을 구하고 행렬을 분해하는 연산이 대표적이다. 이 책은 이들 연산이 필요하지 않아 다루지 않는다. 대신에 수가 한 줄로 이어진 1차원 구조의 벡터와 행과 열로 확장한 2차원 구조의 행렬을 넘어서 3차원 구조, 4차원 구조, 5차원 구조 등으로 확장한 텐서를 소개한다.

텐서tensor는 다차원 구조의 배열이다. 벡터는 1차원 구조의 텐서, 행렬은 2차원 구조의 텐서로 간주한다. 기계 학습이 다루는 데이터는 높은 차원의 구조를 가진 텐서이다. 앞서

소개한 iris 데이터에서 샘플 하나는 1차원 구조의 텐서이고, 150개 샘플로 구성된 전체 데이터는 2차원 구조의 텐서이다.

하지만 영상으로 구성된 데이터는 차원이 더 높다. 하나의 영상은 화소가 2차원 구조로 배치되어 있어 2차원 텐서이다. 여러 장의 영상으로 구성된 영상 데이터는 3차원 텐서이다. 그런데 컬러 영상은 2차원 구조의 R, G, B맵 세 개로 구성되므로 영상 자체가 3차원 텐서이고, 여러 영상으로 구성된 영상 데이터는 4차원 텐서이다. [그림 A-3]은 32*32 크기의 RGB 영상 50,000장으로 구성된 CIFAR-10 데이터의 텐서 구조를 보여준다. 보다 자세한 설명은 텐서플로로 CIFAR-10을 처리하는 5.2.2항을 참조한다.



(a) 컬러 영상 한 장(3차원 구조 텐서) (b) 50,000장의 컬러 영상(4차원 구조 텐서)

[그림 A-3] CIFAR-10 데이터셋

A.2 numpy 라이브러리

인공지능 시스템을 만들려면 앞서 공부한 텐서를 프로그래밍해야 한다. 파이썬이 제공하는 numpy 라이브러리는 ndarray라는 데이터형을 제공하는데 ndarray를 가지고 텐서를 구현한다. 이 절에서는 numpy 라이브러리를 설명하고 다음의 A.3절에서는 파이썬이 기본으로 제공하는 리스트, 튜플, 딕셔너리, 셋 자료구조를 소개한다.

[TIP] numpy는 numerical python의 약어로서 넘파이라고 읽는다.

데이터를 다루는 일은 목공 작업과 유사하다. 원시 데이터는 잡음도 있고, 손실 값도 있고, 아웃라이어도 많다. 이런 원시 데이터를 가공하여 핵심이 되는 패턴을 추출하고 패턴을 조합하고 변환하여 원하는 정보를 추출한다. [그림 A-4]가 보여주는 바와 같이 목공에서는 벌레 먹고, 파이고, 갈라진 원목을 자르고, 붙이고, 대패질하여 판재를 만들고, 판재를 조립하여 멋진 피크닉 테이블을 만든다. 목공을 잘 하려면 연장을 아주 능숙하게 쓸 수 있어야 한다. 마찬가지로 기계 학습을 잘 하려면 데이터를 가공하는 도구를 능숙하게 쓸 수 있어야 하다. 기계 학습에서 도구란 파이썬의 라이브러리가 제공하는 각종 함수들이다. 이 절에서는 numpy 라이브러리가 제공하는 함수를 사용하는 연습을 한다.

지금부터 도구를 쓰는 연습을 시작한다. 공식 사이트인 https://numpy.org/에 접속하여 [Documentation]에서 제공하는 『Numpy User Guide』를 3장까지 공부하면 된다. 도구에 능숙해지는 유일한 길은 반복 학습이다. 공식 문서에서 제공하는 예제를 변형하여 반복 학습을 충실히 하기 바란다.

[TIP] 『Numpy User Guide』는 https://numpy.org/doc/stable/numpy-user.pdf에서 다운로드할 수 있다.

[그림 A-4] 데이터를 다루는 일과 비슷한 목공 작업 과정

ndarray란

numpy는 텐서를 다차원 배열multi-dimensional array이라고 부른다. numpy는 다차원 배열을 ndarray 데이터형의 객체에 저장한다. ndarray는 몇 가지 특성이 있다.

- 처음 생성될 때 메모리 크기가 정해진다. 배열이 커지면 원래 메모리를 반환하고 새로 생성한 큰 메모리를 사용한다.

·모든 요소가 같은 데이터형을 가지며 같은 수의 바이트를 점유한다. 따라서 i번째 요소의 메모리 주소를 아주 빨리 계산할 수 있다. ndarray로 표현한 데이터를 처리하는 시간이 빠른 이유이다.

·A.1절에서 공부한 벡터와 행렬 연산을 포함하여 유용한 함수를 아주 많이 제공한다. 이들 함수는 계산 시간 측면에서 최적화되어 있어 기계 학습에 활용하기에 적합하다. 이런 이유 때문에 기계 학습에 관련된 대부분 라이브러리가 자신의 데이터를 ndarray로 변환하여 처리한다. 이 책에서 많이 사용한 tensorflow 라이브러리도 ndarray를 많이 사용하며, 자신에게 고유한 tensor 데이터형은 ndarray와 호환된다.

아래 예제 코드를 실행해보자. 01행은 numpy 라이브러리를 불러오고 02행은 5개 요소를 가진 ndarray를 만들어 객체 a에 저장한다. print문으로 출력한 결과를 보면 지정한 대로 값이 저장되었음을 확인할 수 있다. 03행은 객체 a의 멤버 함수 sort를 호출한다.[2] sort 함수는 배열에 있는 값을 오름차순으로 정렬하고 결과를 배열에 다시 저장한다. print문으로 출력한 결과를 보면 제대로 정렬되었음을 확인할 수 있다.

```
> import numpy as np                    01
> a=np.array([12,8,20,17,15])           02
> print(a)
[12 8 20 17 15]
> a.sort()                              03
> print(a)
[ 8 12 15 17 20]
```

dir과 type, shape 명령어

파이썬 프로그래밍을 하면서 자주 쓰는 명령어로 dir과 type, shape이 있다. 앞의 예제에서 객체 a의 멤버 함수 sort를 사용했는데, 사용 가능한 함수 목록을 알고자 할 때 dir을 사용한다. 아래 코드에서 04행을 실행하면 객체 a가 가진 멤버 함수의 목록을 알 수 있다.

```
> dir(a)                                04
['T',
 '__abs__',
 '__add__',
 '__and__',
…
 'shape',
 'size',
 'sort',
```

[2] 파이썬은 객체 지향 언어다. 모든 객체는 멤버 변수와 멤버 함수를 가지며, 멤버 함수를 사용할 때는 객체 이름 뒤에 .을 찍고 함수 이름을 적으면 된다. 책 2.7.2항에 객체 지향에 대한 간단한 설명이 있다.

…]

기계 학습 프로그래밍을 하다 보면 아주 많은 객체를 다루게 된다. 이런 상황에서는 객체가 어떤 데이터형을 가지는지 확인할 필요가 종종 생기는데 이때 쓰는 명령어가 type이다. 또한 객체의 모양을 확인할 필요가 발생하는데 이때 쓰는 명령어가 shape이다.

다음 코드에서 05행의 type 명령어는 객체 a가 ndarray형임을 알려주고 06행의 shape 명령어는 a가 5개 요소로 구성된 1차원 구조임을 알려준다.

```
> type(a)                                    05
numpy.ndarray
> a.shape                                    06
(5,)
```

ndarray 배열 만들기

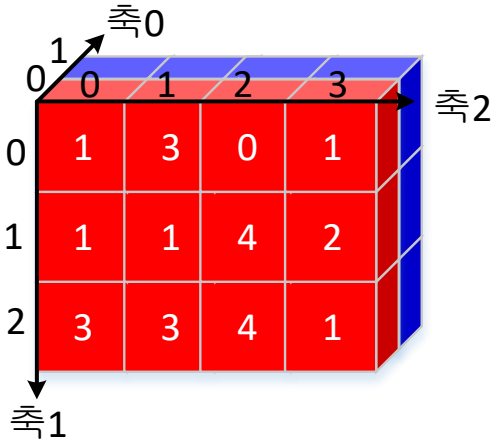
새 배열을 만드는 방법은 여러 가지가 있다. 앞의 코드에서 02행은 array 함수를 사용하여 1차원 구조의 배열을 만들었다. 다음 코드에서 07행도 array 함수로 배열을 만드는데 앞의 02행과 다른 점이 두 가지 있다. 첫째는 2차원 구조의 배열을 만든다는 점이고 둘째는 요소 중에 4.0이라는 실수가 들어있다는 점이다. 배열은 생성될 때 요소를 보고 데이터형을 결정하는데, b는 요소 중에 실수가 있기 때문에 64비트 실수인 float64형이 된다. ndarray 객체는 배열이 몇 차원의 구조인지 알려주는 ndim, 요소의 데이터형을 알려주는 dtype, 모양을 알려주는 shape 등의 멤버 변수를 가진다. 08, 09, 10행은 이들 멤버 변수를 출력하여 객체 b의 정보를 알아낸다.

```
> b=np.array([[12,3,4.0],[1,4,5]])           07
> print(b)
[[12.   3.   4.]
 [ 1.   4.   5.]]
> b.ndim                                     08
2
> b.shape                                    09
(2,3)
> b.dtype                                    10
dtype('float64')
```

11행은 3차원 구조의 배열 c을 생성한다. numpy에서는 n차원 구조에서 각각의 차원을 축$_{axis}$이라 부른다. c는 3개의 축을 가지는데 축 0은 두 요소를 가진다. 두 요소를 구분하기 위해 빨간색과 파란색으로 표시하였다.

축 1은 3개의 요소, 축 2는 4개의 요소를 가진다. 요소의 개수를 길이라고 부르기도 하는데 축 0, 1, 2의 길이는 각각 2, 3, 4이다. [그림 A-5]는 이런 구조를 설명한다. 뒤에서 축을 이용하여 배열을 다른 형태로 변환하는 방법을 공부한다. 따라서 [그림 A-4]를 보고 c의 구조를 잘 이해해 두기 바란다.

```
> c=np.array([[[1,3,0,1],[1,1,4,2],[3,3,4,1]], [[2,1,2,1] ,[1,0,1,0],[1,5,6,2]]])   11
> print(c)
  [[[1 3 0 1]
    [1 1 4 2]
    [3 3 4 1]]
   [[2 1 2 1]
    [1 0 1 0]
    [1 5 6 2]]]
  > c.shape
  (2, 3, 4)
```



[그림 A-5] 세 개의 축을 가진 3차원 구조의 배열(11행이 생성하는 배열 c)

zeros 함수는 0으로 채운 배열을 만들고 ones 함수는 1로 채운 배열을 만든다. 12행은 두 개의 축을 가진 2차원 배열 d를 생성한다. 축 0은 길이가 2, 축 1은 길이가 3이다. d.dtype으로 확인한 결과는 float64이다. numpy는 기본으로 64비트 실수형을 사용하기 때문이다. 마지막 명령어가 보여주듯이 numpy는 random 함수를 사용하여 난수로 구성된 배열을 만들 수 있다. print문의 결과에서 볼 수 있듯이 random 함수는 [0,1] 범위에서 실수 난수를 생성해준다.

```
> d=np.zeros([2,3])                    12
> print(d)
[[0. 0. 0.]
 [0. 0. 0.]]
> d.dtype
dtype('float64')
> e=np.random.random([2,5])
> print(e)
[[0.86997882 0.32528632 0.13021243 0.48668632 0.24810257]
 [0.13114461 0.28725987 0.49369103 0.01422569 0.2509103 ]]
```

배열을 만드는 또 다른 방법은 일정한 간격의 수를 만들어주는 arange 함수를 쓰는

것이다. 다음 코드의 13행은 1에서 시작하여 2.5씩 증가시키면 20미만까지 등간격 수를 만들어 배열에 담아준다.

```
> f=np.arange(1,20,2.5)                    13
> print(f)
[ 1.    3.5   6.    8.5 11.   13.5 16.   18.5]
```

기계 학습 데이터로 ndarray 만들기

기계 학습에서는 공개된 데이터를 읽어 ndarray 배열을 생성하는 경우가 많다. 이 책이 제시한 프로그램 대부분은 이렇게 배열을 생성한다. 다음 코드는 CIFAR-10 데이터를 읽어 cifar10_data라는 ndarray 객체를 생성하고 특성을 확인하는 것이다.

```
> import tensorflow.keras.datasets as ds                    14
> (x_train, y_train), (x_test, y_test) = ds.cifar10.load_data()    15
> print(x_train)
[[[[ 59   62   63]
   [ 43   46   45]
   [ 50   48   43]
   ...
   [158 132 108]
   [152 125 102]
   [148 124 103]]

  [[ 16   20   20]
   [  0    0    0]
   [ 18    8    0]
   ...
> x_train.shape
(50000, 32, 32, 3)
> d.dtype
dtype('uint8')
> d.nbytes
153600000
```

14행은 tensorflow 라이브러리가 설치되어 있어야 실행이 가능하다. tensorflow 설치에 대해서는 책 2.3.1항을 참조한다. 15행은 tensorflow가 기본으로 제공하는 CIFAR-10 데이터를 읽어 훈련 집합과 테스트 집합으로 나누어 저장한다. 여기서는 훈련 집합인 x_train을 살펴본다. x_train을 print문으로 출력하면 [[[[…로 표시된 것으로 보아 4차원 구조의 배열임을 알 수 있다. x_train.shape으로 확인해보니 50000*32*32*3 구조의 배열이라는 사실을 확인할 수 있다. [그림 A-3(b)]는 이 구조를 설명한다. 32*32 해상도의 RGB 영상이 50,000장 담겨 있다고 해석하면 된다. x_train.dtype으로 요소의 데이터형을 확인해보니 부호 없는 1바이트 정수형을 뜻하는 uint8이다. 데이터의 메모리 용량을 알아보려면 nbytes 속성을 출력하면 된다. x_train의 경우 15,360,000바이트임을 확인했다. 대략 15메가 바이트 정도이다. nbytes는 요소의 수를 알려주는 size와 개별 요소가 차지하는 바이트 수를 알려주는 itemsize를 곱한 것과 같다. 따라서 x_train.nbytes는

x_train.size*x_train.itemsize 또는 np.prod(x_train.shape)*x_train.itemsize와 같다.

대입 연산

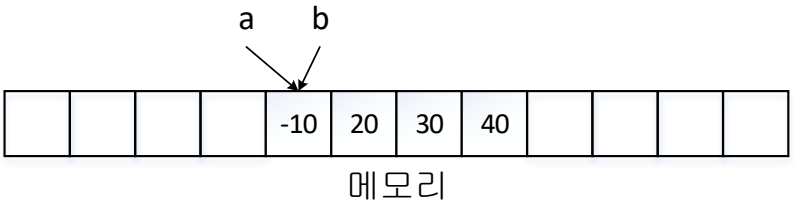
아래 코드는 스칼라 변수 x를 y에 대입한 뒤 y 값을 변경한 다음 x와 y 값을 출력한다. 당연히 x는 원래 값 10을 가지고 있고 y는 변경된 값 12를 갖는다. x와 y는 서로 다른 메모리 주소에 저장되어 있다.

```
> x=10
> y=x
> y=12
> print(x, y)
10 12
```

비슷한 과정을 ndarray로 만든 배열에 적용해 보자. 배열 a를 만든 다음, a를 b에 대입한다. b의 첫번째 요소 b[0]를 -10으로 변경한 후 a와 b를 출력한다. 놀랍게도 b의 첫번째 요소를 바꾸었는데 a도 따라서 바뀌었다. 이러한 사실과 그 이유를 꼭 기억해야 한다. 파이썬에서는 x, y와 같은 스칼라 변수는 다른 메모리 공간을 사용하고, y=x라는 명령어을 실행하면 x에 해당하는 메모리에서 y에 해당하는 메모리로 값을 복사한다. x와 y는 이름도 다르고 메모리 주소도 다르다. 하지만 배열에서는 같은 메모리 공간을 공유한다. [그림 A-6(b)]는 이런 원리를 설명한다. b=a 명령어로 a를 b에 대입하면 b를 위해 새로운 메모리 공간을 확보하고 내용을 복사하는 것이 아니라, 단지 a가 차지하고 있는 메모리 공간을 b가 가리키게 하여 메모리 공간을 공유한다.

```
> a=np.array([10,20,30,40])
> b=a
> b[0]=-10
> print(a,b)
[-10   20   30   40] [-10   20   30   40]
```

(a) 서로 다른 스칼라 변수는 다른 메모리 공간을 사용



(b) 배열에서는 같은 메모리 공간을 공유

[그림 A-6] 파이썬에서 대입 명령어에 따른 효과

다른 메모리 공간에 실제 복사를 하려면 16행과 같이 copy 함수를 사용해야 된다. 아래 코드는 copy 함수를 사용하면 a와 c가 다른 메모리 공간을 차지하게 됨을 확인해준다.

```
> c=a.copy()                          16
> c[1]=-20
> print(a,c)
[-10   20   30   40] [-10 -20   30   40]
```

모양 바꾸기

기계 학습에서는 배열을 다른 모양으로 바꾸어야 하는 경우가 많다. 예를 들어 2차원 구조의 배열로 표현되는 영상을 다층 퍼셉트론에 입력하려면 1차원 구조로 바꾸어야 한다. 반대로 1차원 구조를 2차원으로 변환해야 할 경우도 있다. 이때 쓰는 함수가 reshape이다. 17행은 1차원 배열에 reshape(2,3) 함수를 적용하여 2*3 모양으로 바꾼다. 원래 배열 a와 새로운 배열 b를 출력한 결과, a는 그대로 1차원을 유지하고 b는 2차원이 되었음을 확인할 수 있다. 유념할 점은 배열 a는 원래 모양을 그대로 유지한다는 사실이다.

```
> a=np.array([1,2,3,4,5,6])
> b=a.reshape([2,3])                  17
> print(a)
[1 2 3 4 5 6]
> print(b)
[[1 2 3]
 [4 5 6]]
```

또 하나 유념할 점이 있다. 배열 a와 b가 모양이 달라 서로 다른 메모리 공간을 차지할 것이라 생각할 수 있는데 그렇지 않다. [그림 A-6(b)]의 상황을 그대로 유지한다. 단지 a와 b는 모양만 달라진다. numpy에서는 a와 b의 뷰$_{view}$는 다르고 내용은 같다라고 말한다. a와 b는 뷰 정보는 각자 갖지만 내용은 공유한다. 다음 코드는 a와 b가 메모리 공간을 여전히 공유한다는 사실을 확인한다.

```
> a[4]=-5
> print(b)
[[ 1   2   3]
 [ 4 -5   6]]
```

때때로 데이터의 축의 위치를 교환할 필요가 있다. 축의 위치를 교환하는 T, swapaxes, transpose를 실험해보자. 18행은 2*3 배열 a에 T를 적용하여 3*2 배열 b에 저장한다. 즉 축 0과 축 1을 서로 교환한 셈이다. 축에 대해서는 [그림 A-5]를 참조한다. T는 전치 행렬을 구해주는데 그 이상의 기능을 바로 이어 확인해 본다.

```
> a=np.array([[1,2,3],[4,5,6]])
> b=a.T                               18
> print(b)
```

```
[[1 4]
 [2 5]
 [3 6]]
```

이제 좀더 복잡한 CIFAR-10 데이터에 대해 축을 교환하는 연산을 적용해 보자. x_train은 [그림 A-3(b)]의 50000*32*32*3 구조이다. x_train.T를 적용하면 축의 순서가 뒤집어져 3*32*32*50000이 된다. 즉 축 0, 1, 2, 3이 축 3, 2, 1, 0이 된다. 19행에서는 이전과 다르게 x_train.T.shape으로 간결하게 썼는데, 헷갈리면 이전처럼 x1=x_train.T와 x1.shape의 두 줄로 실행하면 된다. swapaxes 함수는 교환하려는 두 축을 매개변수로 주면 된다. 20행은 0번 축과 3번 축을 교환한다. transpose 함수는 모든 축에 대해 새로운 위치를 지정하면 그에 따라 축의 순서가 바뀐다. 21행의 x_train.transpose(1,2,3,0)은 원래 0,1,2,3이었던 축을 1,2,3,0으로 바꾸라는 지시이다. 다시 한번 유념할 점은 원래 배열인 x_train은 원래대로 유지되며, 변환된 배열은 뷰만 다르고 내용은 원래 배열과 같다는 사실이다.

```
> import tensorflow.keras.datasets as ds
> (x_train, y_train), (x_test, y_test) = ds.cifar10.load_data()
> x_train.shape
(50000, 32, 32, 3)
> x_train.T.shape                          19
(3, 32, 32, 50000)
> x_train.swapaxes(2,3).shape              20
(50000, 32, 3, 32)
> x_train.transpose(1,2,3,0).shape         21
(32, 32, 3, 50000)
```

인덱싱과 슬라이싱

배열을 구성하는 특정 요소에 접근하여 값을 읽거나 변경할 때 요소의 위치를 지정하는 것을 인덱싱indexing이라 한다. c[1]=-20은 인덱싱 사례로서, 1차원 구조의 배열 c의 주소 1에 있는 요소에 -20을 대입하라는 명령이다. 아래 코드는 3*5 크기의 a를 가지고 인덱싱을 설명한다. a[1,2]는 축 0은 1, 축 1은 2 위치의 요소를 가리킨다. 파이썬에서는 배열의 인덱스를 0, 1, 2, …처럼 0부터 시작한다. 따라서 a[1,2]는 print문의 결과에서 빨간색으로 표시한 요소를 가리킨다. 인덱스 -1은 축의 마지막 인덱스를 가리킨다. 축 0의 마지막 인덱스는 2이므로 a[-1,3]은 a[2,3]과 같아 주황색의 요소를 가리킨다. 인덱스 -2는 뒤에서 두번째 인덱스이므로 a[0,-2]는 a[0,3]과 같아 파란색으로 표시된 요소를 가리킨다.

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

이제 배열의 일부를 잘라내는 슬라이싱slicing을 실험해본다. 바로 이전 코드에서 설정한

배열 a를 그대로 사용한다. 아래 코드는 축을 생략하는 방법을 예시한다. 첫번째 명령문에 있는 a[2]는 축 1을 생략한다. 이 경우에 축 0의 2번 인덱스만 잘라내므로 [0 1 -2 -3 2]가 출력된다. 두번째 명령문에 있는 a[-2]는 축 0에 대해 뒤에서 두번째를 가리키므로 a[1]과 같다. 축 0을 생략하려면, 세번째 명령문처럼 축 0에 :을 지정하면 된다. a[:,3]은 축 0을 생략하고 축 1의 위치 3을 가리키므로 [0 5 -3]이 된다.

```
> print(a[2])
[ 1   1 -2 -3   2]
> print(a[-2])
[ 2 -3   4   5   2]
> print(a[:,3])
[ 0   5 -3]
```

특정 축에서 일부 범위를 지정하여 슬라이싱할 수 있다. 아래 코드의 첫번째 명령어는 a[1,1:3]으로 슬라이싱하는데, 축 0은 인덱스 1만 취하고 축 1의 1:3은 인덱스 1과 2를 취한다. 일반적으로 p:q는 p, p+1, …, q-1을 뜻한다. q는 빠진다는 사실을 눈 여겨 봐야 한다. 두번째 명령어는 a[:,1:]으로 슬라이싱한다. 축 0이 지정한 :은 처음부터 끝까지 취하라는 뜻이다. 축 1의 1:은 q가 생략되어 있는 셈인데, 1부터 끝까지 취하라는 뜻이다. 따라서 축 0은 전체를, 축 1은 1:부터 마지막 인덱스까지 취하라는 뜻이어서 축 1에서 1,2,3,4를 잘라낸다. 세번째 명령어는 p가 생략된 :3 형태인데 0:3과 같다.

```
> print(a[1,1:3])
[-3   4]
> print(a[:,1:])
[[ 2 -2   0   1]
 [-3   4   5   2]
 [ 1 -2 -3   2]]
> print(a[:,:3])
[[ 3   2 -2]
 [ 2 -3   4]
 [ 0   1 -2]]
```

지금까지 인덱싱하고 슬라이싱하는 방법을 설명하였는데, 인덱싱하는 또다른 방법을 소개한다. 앞에서 a[1,3]과 같이 인덱싱하였는데 a[1][3]처럼 해도 동작한다. 다음 코드의 첫번째 줄은 둘이 같은 곳을 가리킨다는 사실을 보여준다. 하지만 둘의 내부적인 동작은 크게 다르다. a[1,3]은 인덱스 값 1과 3을 가지고 메모리 주소를 계산하여 바로 접근하는 반면, a[1][3]은 a[1]을 실행하여 슬라이싱한 다음 그 결과에 [3]을 다시 적용한다.

다시 말해 a[*slice1*][*slice2*]는 a를 *slice1*로 슬라이싱한 다음 그 결과에 *slice2*를 적용한다. 아래 코드의 22행과 23행은 a[1,3]과 a[1][3]을 실행하는 시간을 측정한다. timeit a[1,3]은 a[1,3]을 처리하는데 걸리는 시간을 측정해주는데, 실행 결과를 보면 1천만 번 반복하는 일을 7번 반복한 다음 평균을 구하여 110나노초 가량 걸린다는 사실을 알려준다. 세번째 명령어의 결과를 보면 a[1][3]을 계산하는데 224나노초가 걸려 대략 두 배의 시간이 걸린다는 사실을 확인할 수 있다. 24행은 두 가지 인덱싱 방식이 서로 다른 결과를

출력하는 보다 극적인 사례이다. a[:,2]는 축 0은 다 취하고 축 1은 인덱스 2만 취하여 결과가 [-2 4 -2]가 된다. 하지만 a[:][2]는 먼저 a[:]을 실행하여 축 0을 기준으로 다 취하니 a와 같게 되고 그 결과에 [2]를 적용하니 축 0을 기준으로 인덱스 2를 취하여 [1 1 -2 -3 2]가 된다. 앞으로 배열에서 한 요소를 지정할 때는 a[1,3]과 같은 표현을 쓰기 바란다.

```
> print(a[1,3],a[1][3])
5 5
> timeit a[1,3]                                                    22
110 ns ± 1.12 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
> timeit a[1][3]                                                   23
224 ns ± 1.15 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
> print(a[:,2],a[:][2])                                            24
[-2  4 -2] [ 1  1 -2 -3  2]
```

벡터화를 통한 사칙 연산과 관계 연산

파이썬은 벡터화vectorization되어 있다고 말한다. 벡터화의 장점은 표기의 간결성과 효율적인 계산에 있다. 먼저 표기에 대해 살펴보자. 아래 코드가 보여주듯이 두 개의 벡터 a와 b를 더할 때 파이썬은 for i…와 같은 반복 표현을 사용하지 않고 코딩을 한다. 단지 a+b와 같이 코딩하면, 파이썬은 a와 b가 벡터라는 사실을 알고 있으므로 실제 연산을 반복문으로 변환하여 요소 하나씩 순차적으로 계산을 수행한다. +, -, *, /의 사칙 연산자는 요소별로 연산을 적용한다. **는 요소별 제곱을 하는 연산이다. 나눗셈 연산에서는 첫번째 요소가 0이라 Inf가 출력되고 경고 메시지가 출력되었다. 마지막 연산인 b<0은 요소별로 0보다 작은지 검사하여 참이면 True, 거짓이면 False가 된다. 벡터화는 병렬 처리를 지원하여 GPU와 같은 병렬 처리 하드웨어를 사용하는 경우 모든 요소를 동시에 계산하여 계산 속도를 높여준다.

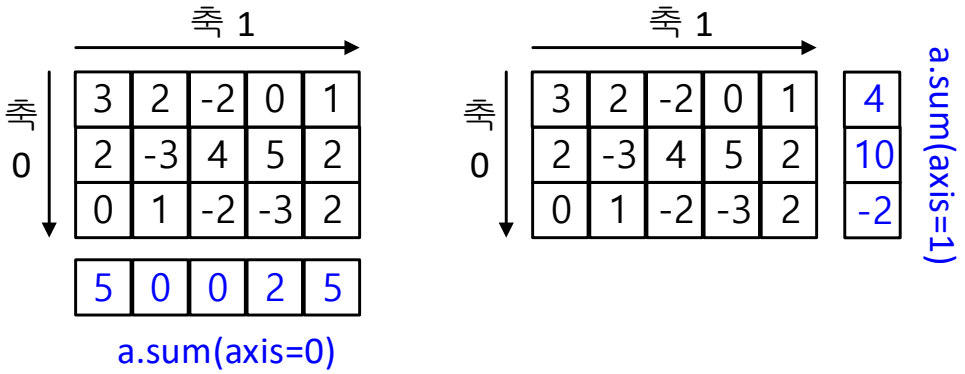
```
> a=np.array([1,2,3,4,5])
> b=np.array([0,-1,2,6,1])
> print(3*a)
[ 3  6  9 12 15]
> print(a**2)
[ 1 4 9 16 25]
> print(a+b)
[ 1  1  5 10  6]
> print(a-b)
[ 1  3  1 -2  4]
> print(a*b)
[ 0 -2  6 24  5]
> print(a/b)
[        inf -2.         1.5        0.66666667 5.        ]
__main__:1: RuntimeWarning: divide by zero encountered in true_divide
> print(b<0)
[False  True False False False]
```

유용한 함수들

numpy 라이브러리는 유용한 함수를 많이 제공한다. 아래 코드는 몇가지 활용 예제를 보여준다. sum 함수는 배열 원소의 합을 계산하는 것으로 a.sum()은 모든 원소에 대해 합을 구한다. a.sum(axis=0)은 축 0을 기준으로 합을 구하며, a.sum(axis=1)은 축 1을 기준으로 합을 구한다. 축의 기준으로 합을 구한다는 말의 의미를 [그림 A-7]이 설명한다.

a.cumsum(axis=0)은 축 0을 기준으로 누적 합을 계산한다. a.max(axis=0)는 축 0을 기준으로 최대값을 계산하며, a.argmax(axis=0)는 축 0을 기준으로 최대값을 가진 인덱스를 찾아준다. 예제 코드에서는 sum, cumsum, max, argmax를 사용했는데, 추가로 평균을 계산하는 mean, 표준편차를 계산하는 std, 정렬해주는 sort 등이 있다.

```
> a=np.array([[3,2,-2,0,1],[2,-3,4,5,2],[0,1,-2,-3,2]])
> print(a)
[[ 3   2 -2   0   1]
 [ 2 -3   4   5   2]
 [ 0   1 -2 -3   2]]
> print(a.sum())
12
> print(a.sum(axis=0))
[5 0 0 2 5]
> print(a.sum(axis=1))
[ 4 10 -2]
> print(a.cumsum(axis=0))
[[ 3   2 -2   0   1]
 [ 5 -1   2   5   3]
 [ 5   0   0   2   5]]
> print(a.max(axis=0))
[3 2 4 5 2]
> print(a.argmax(axis=0))
[0 0 1 1 1]
```



[그림 A-7] a.sum(axis=0)과 a.sum(axis=1)

때때로 조건을 만족하는 요소만 골라 연산을 적용할 필요가 있다. 다음 예제는 0보다 큰

요소만 골라 합을 구하는 코드이다.

```
> positive=a>0
> print(positive)
[[ True  True False False  True]
 [ True False  True  True  True]
 [False  True False False  True]]
> b=a[positive]
> print(b)
[3 2 1 2 4 5 2 1 2]
> b.sum()
22
```

앞의 코드는 다음과 같이 하나의 명령어로 간결하게 쓸 수 있다.

```
> a[a>0].sum()
22
```

배열 붙이기

hstack과 vstack은 각각 배열을 가로 방향과 세로 방향으로 이어 붙이는데 쓰는 함수이다. 25행의 명령어는 a와 b를 수직 방향으로 쌓고 26행도 수직 방향으로 쌓는다. 27행은 수평 방향으로 이어 붙인다. 28행은 1차원 구조의 배열과 2차원 구조의 배열을 수평 방향으로 이어 붙일 수 없어 오류가 발생한다.

```
> a=np.array([1,2,3])
> b=np.array([4,5,6])
> c=np.array([[7,8,9],[1,4,7]])
> x=np.vstack([a,b])                    25
> print(x)
[[1 2 3]
 [4 5 6]]
> y=np.vstack([a,c])                    26
> print(y)
[[1 2 3]
 [7 8 9]
 [1 4 7]]
> z=np.hstack([a,b])                    27
> print(z)
[1 2 3 4 5 6]
> zz=np.hstack([a,c])                   28
ValueError: all the input arrays must have same number of dimensions, but the array at index 0 has 1 dimension(s) and the array at index 1 has 2 dimension(s)
```

유니버설 함수와 브로드캐스팅

요소별 연산을 지원하는 함수를 유니버설 함수라 부른다. 앞에서 수행한 사칙 연산과

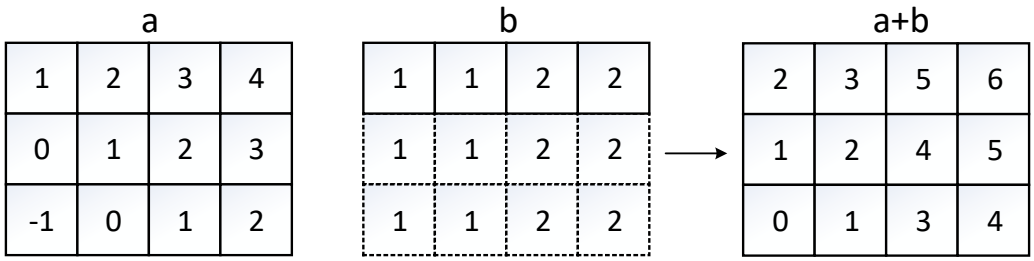
관계 연산은 요소별로 계산하므로 유니버설 함수에 해당한다. 유니버설 함수는 사칙, 관계, 로그, 제곱근, 삼각함수, 비트, 최대와 최소 등의 다양한 연산을 제공한다. 다음 코드는 이들 연산을 예시한다.

add는 유니버설 함수로서 덧셈을 수행한다. log10은 밑이 10인 로그, sin은 사인 함수, left_shift는 왼쪽 시프트, greater는 큰지 비교, maximum은 최대값, round는 반올림을 계산해 준다. round(2)는 소수점 2자리에서 반올림하라는 뜻이다. 아래 코드에서 pi는 원주율 3.141592…이다.
[TIP] 유니버설 함수 목록을 보려면 https://numpy.org/doc/stable/reference/ufuncs.html을 참조한다.

```
> a=np.array([1,2,3,4,5])
> b=np.array([0,-1,2,6,1])
> c=np.array([np.pi/2,np.pi,np.pi*2])
> print(c)
[1.57079633 3.14159265 6.28318531]
> print(np.add(a,b))
[1 1 5 10 6]
> print(np.log10(a))
[0.    0.30103    0.47712125 0.60205999    0.69897]
> print(np.sin(c))
[ 1.0000000e+00    1.2246468e-16 -2.4492936e-16]
> print(np.left_shift(a,1))
[ 2    4    6    8 10]
> print(np.greater(a,b))
[ True    True    True False    True]
> print(np.maximum(a,b))
[1 2 3 6 5]
> print(c.round(2))
[1.05 3.14 6.28]
```

유니버설 함수는 모양이 다른 배열 사이의 연산을 가능하게 해주는 브로드캐스팅broadcasting 기능과 결합하여 더욱 강력하다. [그림 A-8]은 브로드캐스팅 예를 보여준다. 브로드캐스팅은 해당하는 축의 길이가 같거나 둘 중 하나가 1이어야 적용이 가능하다. 아래 코드에서 축 0에 대해서 a는 길이가 3이고 b는 1이다. 축 1에 대해서는 a와 b는 모두 길이가 4이다. 따라서 [그림 A-8]이 설명하는 바와 같이 b를 확장하여 모양을 맞춘 다음에 연산을 실행한다.

```
> a=np.array([[1,2,3,4],[0,1,2,3],[-1,0,1,2]])  # shape: (3,4)
> b=np.array([[1,1,2,2]])                        # shape: (1,4)
> print(a+b)                                     # shape: (3,4)
[[2 3 5 6]
 [1 2 4 5]
 [0 1 3 4]]
```



[그림 A-8] 브로드캐스팅

아래 예시를 추가로 살펴보자. 첫번째 예제에서 a의 축 0과 b의 축 0은 각각 길이가 6과 2인데 길이가 다르고 둘 중 하나가 1이 아니므로 적용이 불가능하다. 나머지 예시는 모두 브로드캐스팅이 가능하다. 두번째 예시는 [그림 A-8]에 해당한다. 세번째 예시에서 b(0)은 b는 스칼라임을 뜻하는데, 이 예시는 np.array([1,2,3,4])+5와 같이 1차원 배열에 스칼라를 더하는 경우이다. 이때 [1,2,3,4]+[5,5,5,5]로 브로드캐스팅 되어 [6,7,8,9]가 된다. 네번째는 브로드캐스팅 규칙에 따라 길이가 1인 축이 더 큰 길이로 확장된다. 다섯 번째 예시에서 b(1)은 b=[5]와 같은 경우이다. 이 예시는 축의 개수가 다른 경우인데 이 경우 손실된 축을 길이가 1인 축으로 확장한 다음에 브로드캐스팅을 적용하여 모양을 맞춘다.

```
a(6,4), b(2,4)          → 불가능
a(3,4), b(1,4)          → c(3,4)
a(4), b(0)              → c(4)
a(5,1,4,1), b(1,6,4,5)  → c(5,6,4,5)
a(3,4,2), b(1)          → c(3,4,2)
```

행렬의 곱셈

[그림 A-2]의 행렬 곱셈은 다음 예시처럼 matmul 함수로 수행한다.

```
> a=np.array([[4,2],[2,7],[-2,1]])     # 3*2 행렬
> c=np.array([[1,-2,3],[5,0,2]])       # 2*3 행렬
> res=np.matmul(a,c)
> print(res)
[[14 -8 16]
 [37 -4 20]
 [ 3   4 -4]]
```

A.3 리스트, 튜플, 딕셔너리, 셋

리스트는 여러 개의 값을 순서대로 저장한다는 점에서 A.2절에서 설명한 numpy의 ndarray와 같지만 쓰임새와 동작 원리가 사뭇 다르다. 따라서 특성을 잘 이해하고 상황에 맞는 자료구조를 선택하는 것이 중요하다. 리스트$_{list}$와 유사한 자료구조로 튜플$_{tuple}$, 딕셔너리$_{dictionary}$, 셋$_{set}$이 제공된다. 리스트와 달리 튜플은 생성하고 나면 값을 변경할 수 없다. 따라서 한번 만들면 수정이 필요 없는 상황에 주로 사용한다. 딕셔너리의 요소는 키$_{key}$와 값$_{value}$의 쌍으로 구성되는데 키를 통해 값을 검색하는 응용에 주로 사용한다. 셋은 집합을 표현하기 때문에 중복된 요소를 허용하지 않는다. 이들 자료구조는 모두 파이썬이 기본으로 제공하므로 라이브러리를 임포트하지 않고 바로 사용할 수 있다.

리스트, 튜플, 딕셔너리, 셋 만들기와 인덱싱, 슬라이싱

리스트는 […], 튜플은 (…), 딕셔너리는 {…}라는 괄호를 사용하여 만들어 서로 구별한다. 집합은 set이라는 함수로 만든다. 다음 코드의 앞 네 줄은 순서대로 리스트, 튜플, 딕셔너리, 집합을 생성한다. type 함수로 데이터형을 알아본 결과 순서대로 list형, tuple형, dict형, set형이라는 사실을 확인할 수 있다. 객체 a, b, c, d를 출력한 결과, 집합을 표현하는 d에서는 두 번 나타난 2를 하나 제거하여 한번만 나타났음을 알 수 있다. dict형의 c에서 키로 정수와 실수, 문자열을 섞어 쓸 수 있다는 사실을 확인할 수 있다. 마지막 명령어는 튜플의 요소를 변경하려 시도하는데 TypeError가 발생하여 변경이 불가함을 알 수 있다.
[TIP] print문에서 sep='\n'은 줄을 바꿔 객체를 출력하라는 뜻이다.

```
> a=[5,2,3,8,2]
> b=(5,2,3,8,2)
> c={1: 'book', 5: 'notebook', 3: 'pencil', -3: 'eraser', 'as': 120, 12.2: 50}
> d=set([5,2,3,8,2])
> print(type(a),type(b),type(c),type(d))
<class 'list'> <class 'tuple'> <class 'dict'> <class 'set'>
> print(a,b,c,d,sep='\n')
[5, 2, 3, 8, 2]
(5, 2, 3, 8, 2)
{1: 'book', 5: 'notebook', 3: 'pencil', -3: 'eraser', 'as': 120, 12.2: 50}{8, 2, 3, 5}
> b[2]=-3
TypeError: 'tuple' object does not support item assignment
```

numpy의 ndarray 배열에서는 모든 요소가 같은 데이터형이어야 한다. 반면에 리스트는 서로 다른 데이터형을 허용한다. 다음 코드를 살펴보자. 리스트 객체 a는 5개 요소를 가지는데 앞의 3개는 정수이고 네번째 요소는 길이가 4인 리스트이며 다섯 번째 요소는 문자열이다. 인덱스 2, 3, 4에 있는 요소를 출력하면 해당 요소 값이 제대로 출력된다. a[3][2]는 a의 3번 요소인 [3,4,-2,2]의 2번 요소를 가리키므로 -2가 출력된다. a[4][1]은 a의 4번 요소인 'book'의 1번 요소를 가리키므로 문자 'o'가 출력된다. 리스트를 인덱싱하는

방법은 ndarray 배열과 개념이 비슷하다. 리스트에서는 a[3][2]처럼 인덱싱한다. numpy의 ndarray가 사용하는 a[3,2] 방식은 허용되지 않는다.

```
> a=[5,2,3,[ 'pencil',4,-2,2], 'book']
> print(a[2],a[3],a[4])
3 ['pencil', 4, -2, 2] book
> print(a[3][2],a[4][1])
-2 o
```

리스트의 슬라이싱은 ndarray의 슬라이싱과 비슷하다. 다음 코드는 몇가지 슬라이싱 예제를 보여준다. A.2절의 ndarray 슬라이싱에서 설명한 바와 같이 p:q는 p, p+1, …, q-1을 뜻한다. p가 생략되면 0:q를 뜻하고 q를 생략하면 p부터 마지막에 위치한 요소까지 포함한다.

```
> print(a[1:4],a[:-1],a[2:],sep='\n')
[2, 3, ['pencil', 4, -2, 2]]
[5, 2, 3, ['pencil', 4, -2, 2]]
[3, ['pencil', 4, -2, 2], 'book']
> print(a[3][1:],a[4][:3])
[4, -2, 2] boo
```

리스트와 튜플의 인덱싱은 0,1,2,…의 숫자 인덱스로 이루어지는 반면, 딕셔너리의 인덱싱은 키를 통해 이루어진다. 다음 예제는 딕셔너리 객체인 c를 인덱싱하는 예제이다. -3과 5를 키로 사용해서 인덱싱한 결과 해당하는 값 eraser와 notebook이 출력되었다. 하지만 2를 키로 사용해서 c[2]와 같이 인덱싱하면 2라는 키가 없기 때문에 KeyError가 발생한다.

```
> print(c[-3],c[5], c['as'], c[12.2])
eraser notebook 120 50> print(c[2])
KeyError: 2
```

+와 * 연산자

리스트에서 +는 덧셈을 뜻하는 것이 아니라 두 리스트를 접합하는 연산을 수행한다. *는 곱셈이 아니라 지정된 수만큼 반복한다. 다음 코드를 살펴보자. +와 * 연산자는 리스트와 튜플에만 적용할 수 있고 딕셔너리와 집합에는 적용할 수 없다.

```
> x=[1,2,'pen']
> y=[-2,1]
> print(3*x)
[1, 2, 'pen', 1, 2, 'pen', 1, 2, 'pen']
> print(y+x)
[-2, 1, 1, 2, 'pen']
```

추가와 삭제 연산

리스트에 요소를 추가하거나 삭제하는 연산은 리스트 클래스의 멤버 함수로 제공된다. append 함수는 뒤에 요소를 추가해주고, insert 함수는 지정한 위치에 요소를 추가한다. remove는 지정된 값을 가진 요소를 삭제하며 pop은 지정된 위치의 요소를 삭제한다. 추가와 삭제 연산 이외에도 정렬하는 sort, 뒤집는 reverse 등 유용한 함수를 많이 제공하니 튜토리얼 문서를 참조하기 바란다. 아래 예제 코드는 리스트 z에 append, insert, remove, pop 함수를 적용한 사례를 보여준다.

[TIP] 리스트의 다양한 연산은 https://docs.python.org/3/tutorial/datastructures.html를 참조한다.

```
> z=[1,2,3,4]
> z.append(-7)
> print(z)
[1, 2, 3, 4, -7]
> z.insert(1,-7)
> print(z)
[1, -7, 2, 3, 4, -7]
> z.remove(-7)
> print(z)
[1, 2, 3, 4, -7]
> v=z.pop(2)
> print(v,z)
3 [1, 2, 4, -7]
```

리스트 내포

리스트 내포list comprehension를 이용하면 간결하게 리스트를 만들 수 있다. 다음 코드는 i를 0,1,2,…,9로 변화시키면서 i*i를 계산한 리스트를 만들어준다.

```
> x2=[i*i for i in range(10)]
> print(x2)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

위 코드는 아래의 세 줄짜리 코드를 한 줄로 간결하게 쓴 셈이다. 이와 같이 리스트 내포는 어떤 식을 반복적으로 계산하여 리스트를 만들 수 있는 간결한 방법이다.

```
> x2=[ ]
> for i in range(10):
>       x2.append(i*i)
```

다음 코드는 중첩된 리스트 내포의 예제로 리스트 내포를 이용하여 리스트의 리스트를 만들 수 있다는 사실을 보여준다. x4는 리스트 내포에 조건문을 적용하여 조건이 맞는 경우에만 값을 생성하게 할 수 있다는 사실을 보여준다.

```
> x3=[[i,j*2] for i in [10,2,3,1] for j in [2,4]]
> print(x3)
[[10, 4], [10, 8], [2, 4], [2, 8], [3, 4], [3, 8], [1, 4], [1, 8]]
> x4=[[i,j*2] for i in [10,2,3,1] for j in [2,4] if i!=j]
```

```
> print(x4)
[[10, 4], [10, 8], [2, 8], [3, 4], [3, 8], [1, 4], [1, 8]]
```

부록 B. matplotlib 라이브러리: 파이썬의 데이터 시각화

인공지능은 데이터를 다루는 학문이다. 대부분 데이터는 아주 많은 양의 숫자와 기호로 구성되므로 그 자체를 보고 특성을 파악하거나 패턴을 발견하는 일은 거의 불가능하다. 하지만 그래프를 이용하여 시각화하면 데이터의 특성을 한 눈에 파악할 가능성이 생긴다.

이 책의 본문에서는 다양한 유형의 데이터를 시각화하여 독자가 데이터의 특성을 쉽게 이해하도록 배려한다. 부록 B는 독자가 책의 본문에 있는 프로그램을 이해할 수 있도록 matplotlib 라이브러리 사용법을 소개한다.

[TIP] matplotlib은 맷플롯립이라고 부른다.

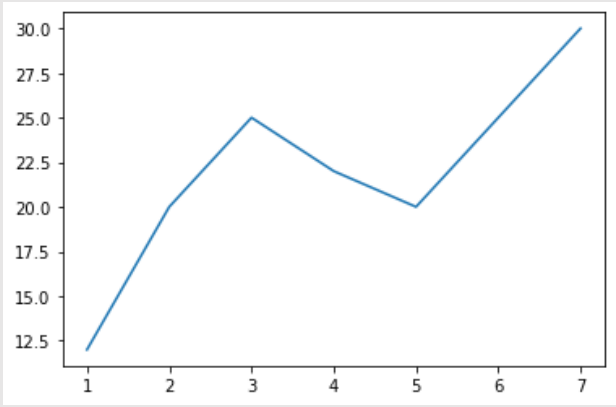
matplotlib은 파이썬 프로그래밍에서 그래프를 그릴 때 가장 많이 사용하는 라이브러리다. 아래 코드의 첫번째 줄은 matplotlib을 불러오고 두번째 줄은 부록 A에서 다룬 numpy 라이브러리를 불러온다. 앞으로 제시할 코드는 모두 이들 라이브러리를 임포트한 상태에서 실행한다고 가정한다.

```
> import matplotlib.pyplot as plt
> import numpy as np
```

B.1 그래프 그리기

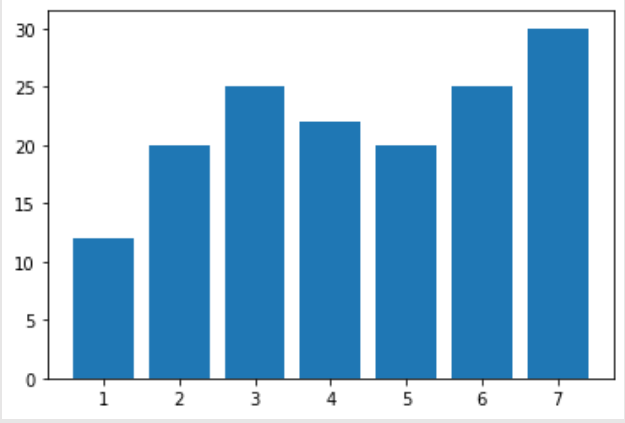
아주 단순한 그래프를 그려보자. 예를 들어 일주일의 과일 판매량을 그래프로 그린다고 가정한다. 7일을 1, 2, 3, …으로 표현하여 x에 저장하고 판매량을 y 변수에 저장한다. 그런 다음 plt.plot(x,y)를 실행하면 멋진 꺾은선 그래프가 생성된다. x는 가로축을 위한 값으로 쓰이고 y는 세로축의 값으로 쓰인다.

```
> x=[1,2,3,4,5,6,7]                # 일주일
> y=[12,20,25,22,20,25,30]         # 과일 판매량
> plt.plot(x,y)
```

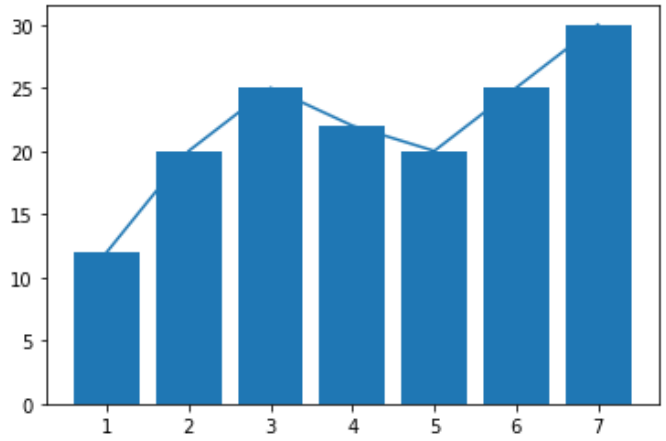


막대 그래프를 그리고 싶으면 plt.bar(x,y)를 실행하면 된다.

```
> plt.bar(x,y)
```



앞의 예제처럼 plt.plot()과 plt.bar()를 따로 실행하는 경우 별개의 그림이 생성된다. 하지만 두 명령어를 한꺼번에 실행하면 아래처럼 하나의 그림에 두 그래프가 겹쳐서 나타난다. matplotlib은 이어서 등장하는 명령어를 하나의 그림에 계속 적용하기 때문이다.

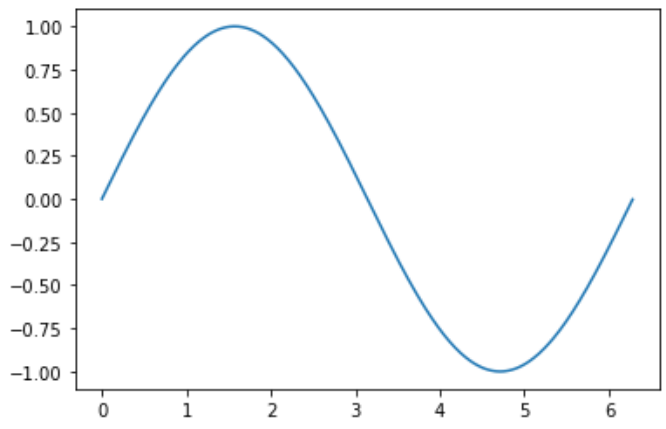


show() 함수가 나타나면 그때 하나의 그림을 생성한 다음, 이후 명령어는 새로운 그림에 적용한다. 예를 들어 아래 코드와 같이 plt.plot() 명령어 다음에 plt.show() 함수를 두면 이 코드를 한꺼번에 실행하더라도 두 개의 그래프를 따로따로 생성해준다.

```
…
plt.plot(x,y)
plt.show()
plt.bar(x,y)
plt.show()
…
```

이제 약간 복잡한 sin 그래프를 그려보자. 가로축은 0도부터 2π도까지 0.05만큼씩 증가시킨다. 아래 코드로 쉽게 멋진 sin 그래프를 그렸다.

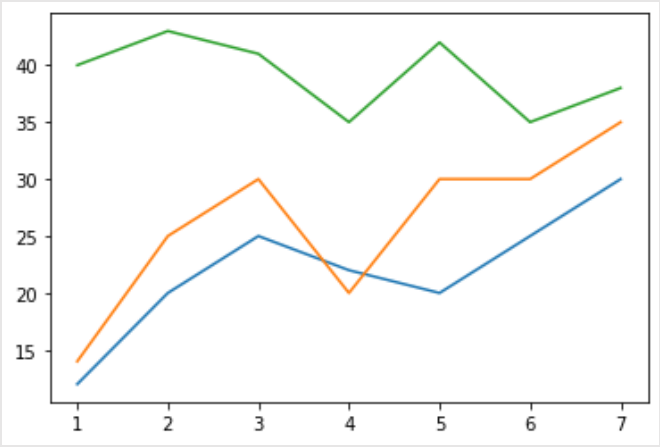
```
> x=np.arange(0,2*np.pi,0.05)
> y=np.sin(x)
> plt.plot(x,y)
```



앞에서는 한 개 품목의 판매량을 시각화했는데, 이제 세 개 품목으로 확장해보자. 두번째

행은 판매량을 저장하는 y를 2차원 구조로 확장한다. 그래프를 그리는 명령어는 앞에서와 똑같이 plt.plot(x,y)이다. 단지 y 변수만 1차원 구조가 2차원 구조로 바뀌었다. matplotlib은 세 개 품목의 데이터를 색깔로 구분하여 보기 좋게 그려준다.

```
> x=np.array([1,2,3,4,5,6,7])
> y=np.array([[12,14,40],[20,25,43],[25,30,41],[22,20,35],[20,30,42],[25,30,35],[30,35,38]])
> plt.plot(x,y)
```

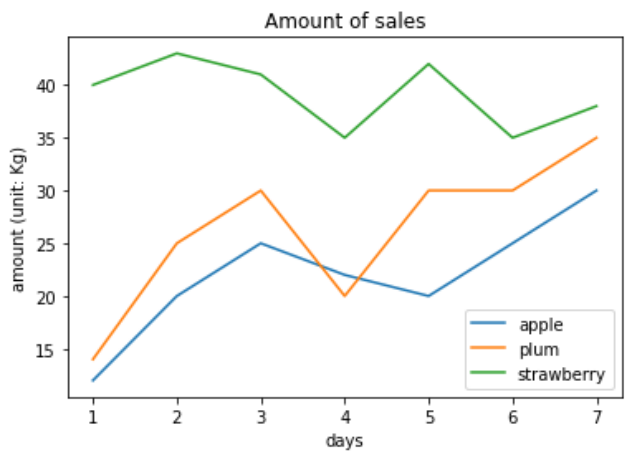


B.2 그래프 꾸미기

앞 절에서는 간단한 코드로 멋진 그래프를 얻었다. 그런데 이들 그래프는 여러 측면에서 부족함이 있다. 축이 무엇을 뜻하는지 설명이 없고 세 품목이 무엇인지 표시되어 있지 않다. 선의 색깔이나 굵기도 기본값을 사용하였기 때문에 마음에 들지 않을 수 있다. 이제 그래프를 꾸미는 프로그래밍 실습을 해본다.

다음 코드는 그래프에 여러 가지 유용한 장식을 덧붙인다. title 함수는 그래프에 이름을 붙였고, xlabel과 ylabel은 가로축과 세로축에 이름을 붙여준다. legend 함수의 첫번째 매개변수는 품목 세 개의 설명문이며 두번째 매개변수 loc은 설명문이 나타날 위치를 지정한다. loc 매개변수를 생략하면 기본값인 'best'로 설정되어 네 구석 중 가장 여유로운 곳을 찾아 자동으로 위치를 잡아준다. 마지막 행의 show 함수는 앞에서 설명한 바와 같이 호출되는 순간 그래프를 생성하고, 이후에 나오는 명령어는 새로운 그림을 시작한다.

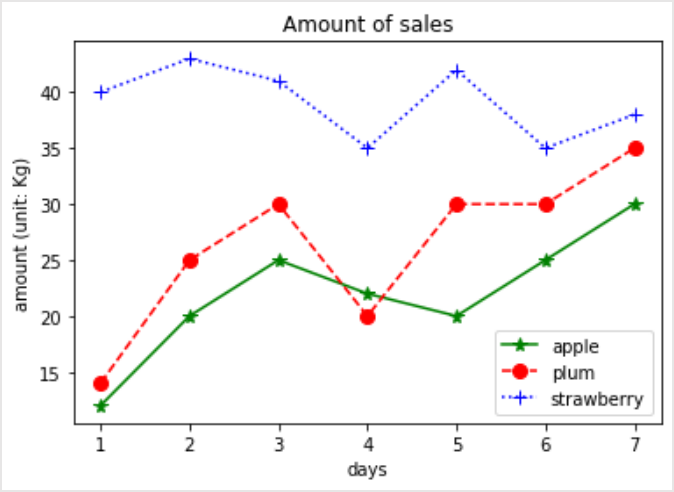
```
> x=np.array([1,2,3,4,5,6,7])
> y=np.array([[12,14,40],[20,25,43],[25,30,41],[22,20,35],[20,30,42],[25,30,35],[30,35,38]])
> plt.plot(x,y)                                    ①
> plt.title('Amount of sales')
> plt.xlabel('days')
> plt.ylabel('amount (unit: Kg)')
> plt.legend(['apple','plum','strawberry'],loc='lower right')
> plt.show()
```



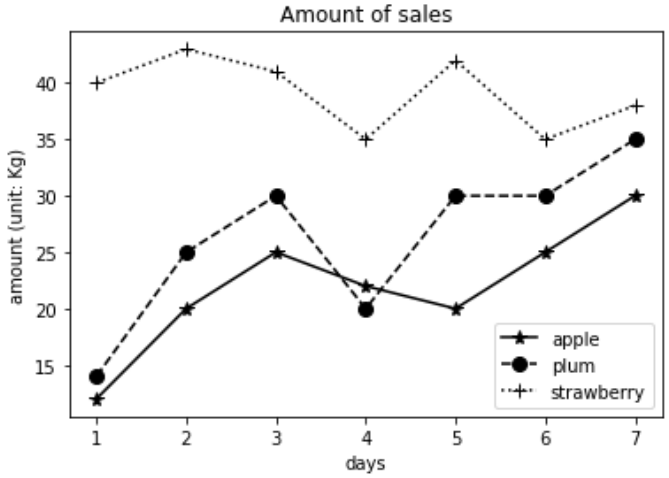
이제 품목을 나타내는 세 개의 선분을 꾸며보자. 위에 있는 코드에서 ①행을 아래와 같이 세 행으로 바꾸면 된다. 이들 행은 세 개의 품목을 저장한 y를 슬라이싱하여 각각 첫번째 품목 y[:,0], 두번째 품목 y[:,1], 세번째 품목 y[:,2]를 사용한다. 'g*-', 'ro—',

'b+ :'에서 g(녹색), r(빨간색), b(파란색)는 색깔을 나타내며, *, o, + 는 점을 강조할 마커를 나타내며, - (실선), -- (파선), : (점선)은 선분의 스타일을 나타낸다. linewidth 매개변수는 선분의 굵기, markersize는 마커의 크기를 나타낸다.

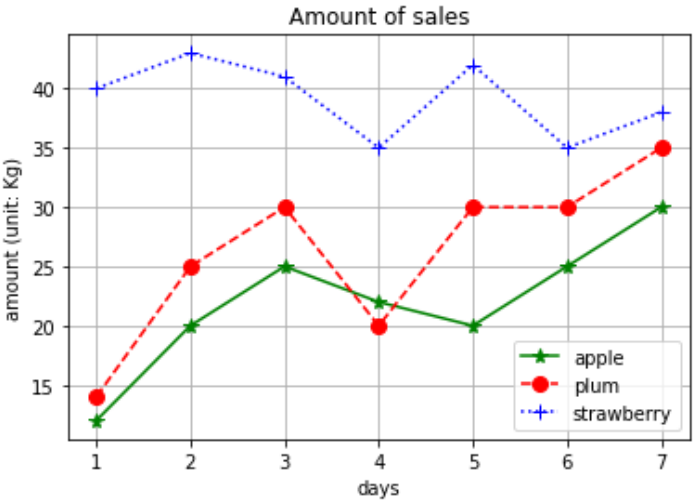
```
plt.plot(x,y[:,0],'g*-',linewidth=1.5,markersize=8)
plt.plot(x,y[:,1],'ro--',linewidth=1.5,markersize=8)
plt.plot(x,y[:,2],'b+:',linewidth=1.5,markersize=8)
```



컬러를 쓸 수 없는 상황에서는 마커 또는 선분 모양이 매우 중요하다. 아래 그림은 위의 코드에서 색상을 나타내는 g, r, b를 모두 k로 바꾸고 실행한 결과이다.



격자를 넣어 값을 쉽게 알아볼 수 있게 하려면 plt.grid()라는 명령을 추가하면 된다. 위의 코드에서 plt.plot()과 plt.show() 사이에 아무 곳에나 추가하면 된다. 다음은 격자를 추가한 그림으로 점의 위치를 보다 쉽게 알아볼 수 있다.


Amount of sales
amount (unit: Kg)
days
apple
plum
strawberry
40
35
30
25
20
15
1
2
3
4
5
6
7

B.3 그림 하나에 여러 그래프 배치하기

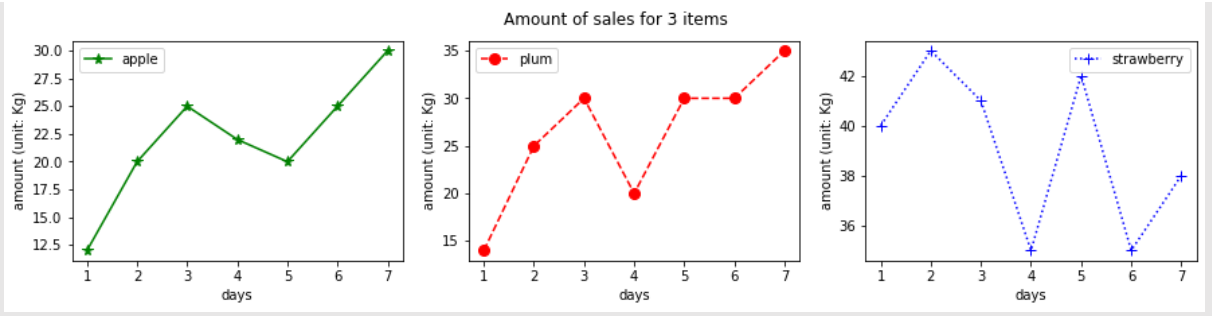
때때로 여러 개의 그래프를 하나의 그림에 담을 필요가 있다. 예를 들어 세 품목을 따로 그린 세 개의 그래프를 하나의 그림에 그리는 프로그래밍을 해보자. 이런 용도로 사용하는 함수가 subplot이다.

다음 코드는 subplot 함수를 이용하여 세 개의 그래프를 한 줄에 그린다. 1-1행은 그림의 크기를 지정한다. figsize=(15,3)은 가로 방향 길이를 15, 세로 방향 길이를 3으로 설정한다. 이 매개변수를 다르게 설정하여 그림 크기가 어떻게 달라지는지 실험해 보기 바란다. 1-2행의 suptitle 함수는 그래프 전체의 제목을 위쪽에 붙인다.
[TIP] 2, 3, 4행은 파란색으로 표시한 부분만 다르고 같은 코드를 반복한다.

2-1행의 subplot(1,3,1)은 1*3 배치(한 행에 3열 배치)에서 1번 위치에 그리라는 뜻이고, 그 뒤에 따라오는 plot, xlabel, ylabel, legend 함수는 1번 위치에 적용된다.

3-1행의 subplot(1,3,2)는 1*3 배치에서 2번 위치에 그리라는 뜻이고, 그 뒤에 따라오는 plot, xlabel, ylabel, legend 함수는 2번 위치에 적용된다. 같은 과정이 코드 4행에도 적용된다. 마지막 명령어인 show 함수는 그래프를 완성하고 생성해준다. legend 함수에서 loc 매개변수를 생략하여 기본값 'best'를 사용하기 때문에 세 개 그래프의 설명문이 자동으로 가장 여유로운 곳에 배치되었다.

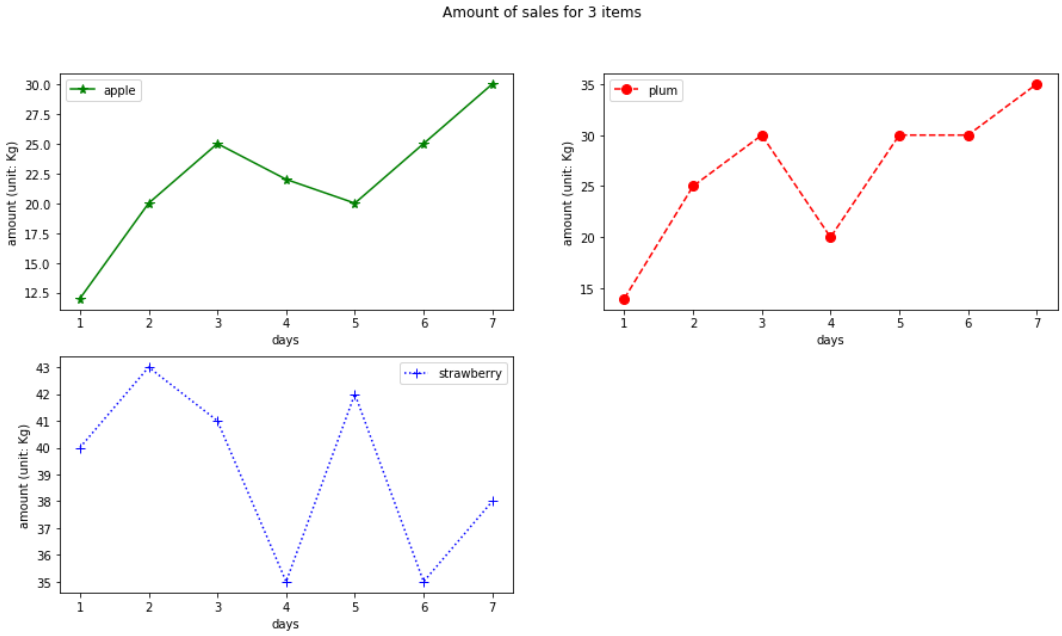
```
> x=np.array([1,2,3,4,5,6,7])
> y=np.array([[12,14,40],[20,25,43],[25,30,41],[22,20,35],[20,30,42],[25,30,35],[30,35,38]])
> plt.figure(figsize=(15,3))                                    1-1
> plt.suptitle('Amount of sales for 3 items')                   1-2
> plt.subplot(1,3,1)                                            2-1
> plt.plot(x,y[:,0],'g*-',linewidth=1.5,markersize=8)            2-2
> plt.xlabel('days')                                            2-3
> plt.ylabel('amount (unit: Kg)')                               2-4
> plt.legend(['apple'])                                         2-5
> plt.subplot(1,3,2)                                            3-1
> plt.plot(x,y[:,1],'ro--',linewidth=1.5,markersize=8)           3-2
> plt.xlabel('days')                                            3-3
> plt.ylabel('amount (unit: Kg)')                               3-4
> plt.legend(['plum'])                                          3-5
> plt.subplot(1,3,3)                                            4-1
> plt.plot(x,y[:,2],'b+:',linewidth=1.5,markersize=8)            4-2
> plt.xlabel('days')                                            4-3
> plt.ylabel('amount (unit: Kg)')                               4-4
> plt.legend(['strawberry'])                                    4-5
> plt.show()
```



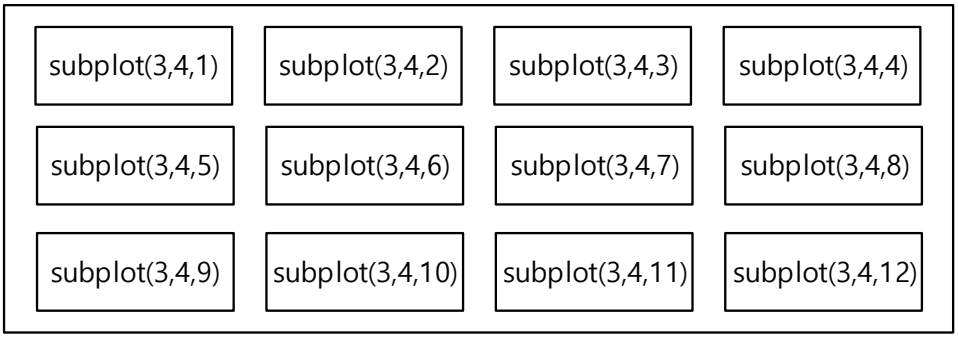
위의 프로그램은 같은 코드 패턴을 세 번 반복한다. 이런 프로그래밍은 바람직하지 않다. 아래 코드는 중복성을 제거한 깔끔한 코드이다. 위의 코드에서 파란색으로 표시한 부분, 즉 반복할 때마다 바뀌는 부분만 반복문의 인덱스 i로 적절히 대치한 프로그램이다.

```
> x=np.array([1,2,3,4,5,6,7])
> y=np.array([[12,14,40],[20,25,43],[25,30,41],[22,20,35],[20,30,42],[25,30,35],[30,35,38]])
> plt.figure(figsize=(15,3))
> plt.suptitle('Amount of sales for 3 items')
> decoration=['g*-','ro--','b+:']                    # 선분 스타일
> legends=['apple','plum','strawberry']              # 세 품목의 설명문
> for i in range(3):
>       plt.subplot(1,3,i+1)
>       plt.plot(x,y[:,i],decoration[i],linewidth=1.5,markersize=8)
>       plt.xlabel('days')
>       plt.ylabel('amount (unit: Kg)')
>       plt.legend([legends[i]])
> plt.show()
```

앞의 코드는 한 줄에 세 개, 즉 1*3 형태로 그래프를 배치하였다. 한 줄에 두 개씩 두 줄에 걸쳐 그리고 싶으면, 위 코드에서 plt.subplot(1,3,i+1)을 plt.subplot(2,2,i+1)로 바꾸기만 하면 된다. 위 코드의 세번째 행은 figure 함수로 그림의 크기를 지정하는데, figure(figsize(15,3))을 그대로 두면 위아래로 찌그러져 보이는데 figure(figsize(15,8))로 바꾸면 적절한 크기가 된다. 아래 그림은 이렇게 얻은 그래프다.



[그림 B-1]은 subplot(r,c,i)가 그래프의 위치를 지정하는 방법을 설명한다. 앞의 두 매개변수 r과 c는 배치 모양을 나타내는데, [그림 B-1]에서는 r=3이고 c=4이어서 3*4 모양으로 배치한다. 모든 subplot 함수 호출에서 앞의 두 매개변수의 값은 3과 4이어야 한다. 세번째 매개변수 i는 3*4 배치의 12곳 위치를 지정하는데 [그림 B-1]이 보여주는 바와 같이 행 우선으로 순서를 매긴다. 즉 첫번째 행은 1,2,3,4이고 두번째 행은 5,6,7,8이고 세번째 행은 9,10,11,12이다.



[그림 B-1] 여러 그래프를 한꺼번에 그리는 subplot(3,4,i) 함수 사용 예제