

# 공간정보 딥러닝 응용

## - 1장. 파이썬 -

- 1) 파이썬이란?
- 2) 파이썬 설치
- 3) 파이썬 인터프리터
- 4) 파이썬 스크립트 파일
- 5) NumPy & matplotlib 라이브러리

서울시립대학교 공간정보공학과  
정 형 섭

# 1) 파이썬이란?

## • 파이썬

→ Python은 현재 가장 인기 있는 프로그래밍 언어.

→ 딥러닝 시스템의 구현은 대개 python을 기반으로 개발되어 있어 딥러닝을 배우기 위해서는 기본이 되는 언어임.

→ 파이썬은 간단하고 배우기 쉬운 프로그래밍 언어로 알려져 있음.

→ 장점:

- 1) 오픈소스라 무료로 자유롭게 이용가능
- 2) 영어와 유사한 문법 구조로 영어문법을 구사할 수 있으면 쉽게 이용가능
- 3) 불편한 컴파일 과정 불필요
- 4) 읽기 쉽고, 성능도 뛰어남
- 5) 데이터가 많은 경우에도 처리가능
- 6) 초보자와 전문가 모두 활용

## 파이썬이란?

### • 파이썬

- 과학분야, 특히 기계학습(machine learning)과 데이터 과학분야에서 널리 활용됨.
- NumPy와 SciPy와 같은 수치계산과 통계처리를 다루는 성능이 좋은 라이브러리를 활용가능
- 딥러닝 프레임워크에도 파이썬을 사용: Caffe, TensorFlow, Chainer, Theano 등의 딥러닝 프레임워크들이 Python용 API를 제공함.
- API: [ application programming interface ]  
운영체제와 응용프로그램 사이의 통신에 사용되는 언어나 메시지 형식

API는 응용 프로그램이 운영체제나 데이터베이스 관리 시스템과 같은 시스템 프로그램과 통신할 때 사용되는 언어나 메시지 형식을 가지며, API는 프로그램 내에서 실행을 위해 특정 서브루틴에 연결을 제공하는 함수를 호출하는 것으로 구현된다. 그러므로 하나의 API는 함수의 호출에 의해 요청되는 작업을 수행하기 위해 이미 존재하거나 또는 연결되어야 하는 몇 개의 프로그램 모듈이나 루틴을 가진다. (두산백과)

## 2) 파이썬 설치

### • 파이썬 설치시 유의사항

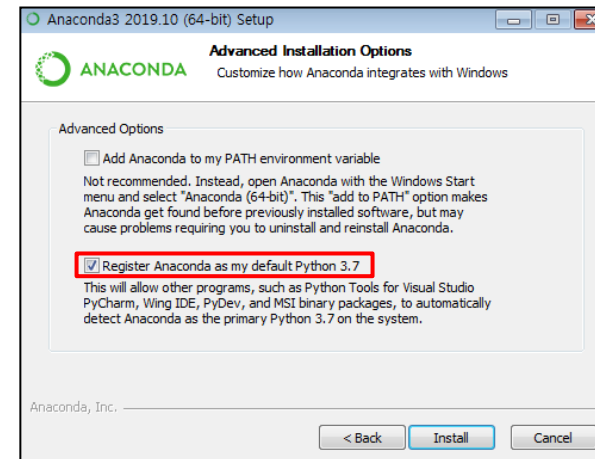
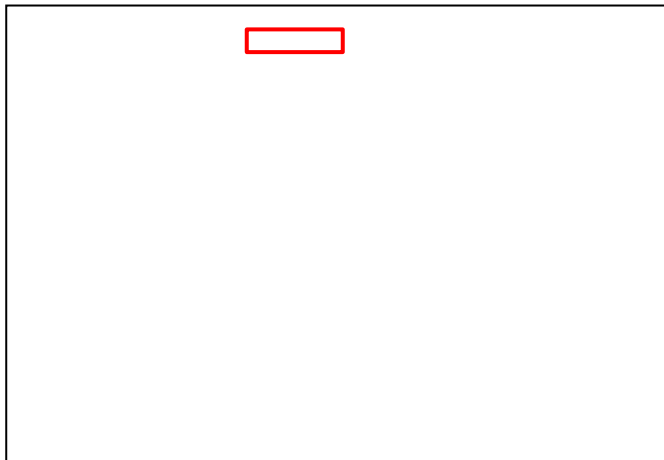
- 파이썬은 버전이 2와 3으로 구분되는데, 쉽게 두 버전간의 상관성이 아주 낮다고 생각하면 됨.
- 파이썬 3로 짠 프로그램은 파이썬 2에서는 실행되지 않으므로 파이썬 3를 설치하는 것이 좋음.
- 딥러닝을 구현하기 위하여 꼭 필요한 라이브러리는 NumPy와 matplotlib임.
  - 1) NumPy: 수치계산용 라이브러리. 수학알고리즘과 행렬연산을 위한 함수를 포함
  - 2) matplotlib: 실험결과를 시각화하거나 실행과정의 중간데이터의 그래프를 그려주는 라이브러리
- 아나콘다(Anaconda) 배포판  
배포판이란 사용자가 설치를 한 번에 수행할 수 있도록 필요한 라이브러리 등을 정리한 것으로 파이썬 설치시 '아나콘다'라는 배포판을 이용하여 설치하는 것이 편리함.  
아나콘다에는 NumPy와 matplotlib가 포함되어있음.  
아나콘다 버전 3을 설치하여 활용가능

## 파이썬 설치

### • 아나콘다(Anaconda) 설치

→ 아나콘다 사이트에서 상단의 Products-Anaconda Distribution으로 들어간 후  
아나콘다 파이썬 3.7 버전 다운로드하기 (<https://www.anaconda.com/distribution/>)

→ 설치 과정에서 Advanced Options에서는 Register Anaconda as my default  
Python 3.7 옵션 체크 (기본 옵션)



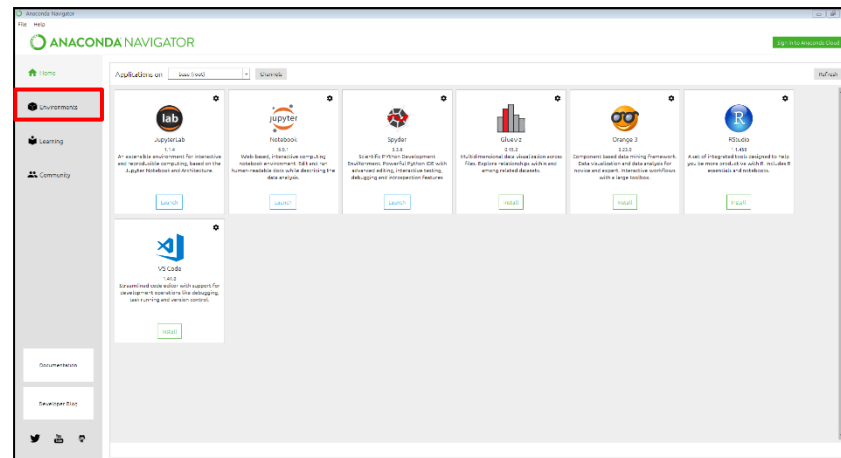
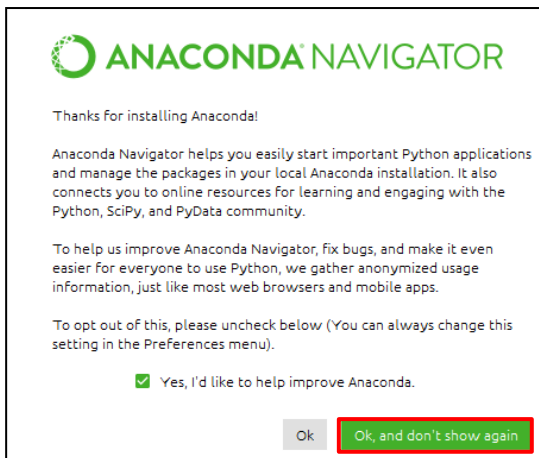
# 파이썬 설치

## • 설치된 라이브러리 확인

→ 아나콘다 설치 후 아나콘다 네비게이터(Anaconda Navigator) 실행

→ 첫 실행 시 뜨는 팝업 창에서는 Ok, and don't show again 클릭

→ 네비게이터에서 좌측의 Environments 탭 클릭



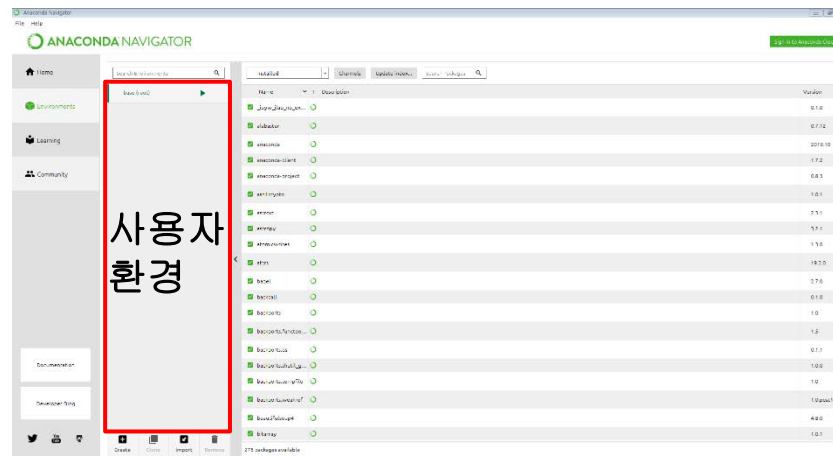
# 파이썬 설치

## • 설치된 라이브러리 확인

→ Search Environments 탭은 아나콘다 플랫폼 사용자 환경을 나타냄

→ 각 환경마다 컴퓨터 내부에 별도의 폴더를 생성하여 관리  
(A 환경에서 설치한 라이브러리를 B 환경에서 사용할 수 없음)

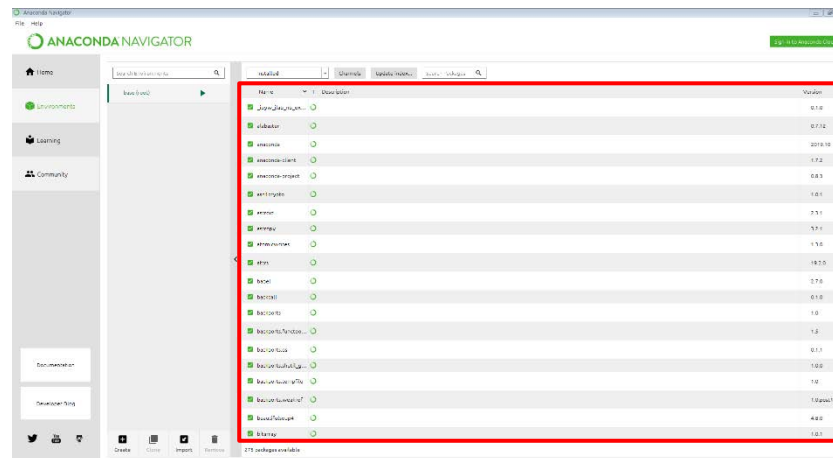
→ 기본적으로 base(root) 를 이용



## 파이썬 설치

## ● 설치된 라이브러리 확인

- Search Packages 탭은 각 사용자 환경에서 설치된 라이브러리를 확인할 수 있음
- 라이브러리 이름과 버전을 확인할 수 있으며, 사용하지 않는 라이브러리는 삭제 가능함
- Search Packages 박스에서 라이브러리를 검색할 수 있음





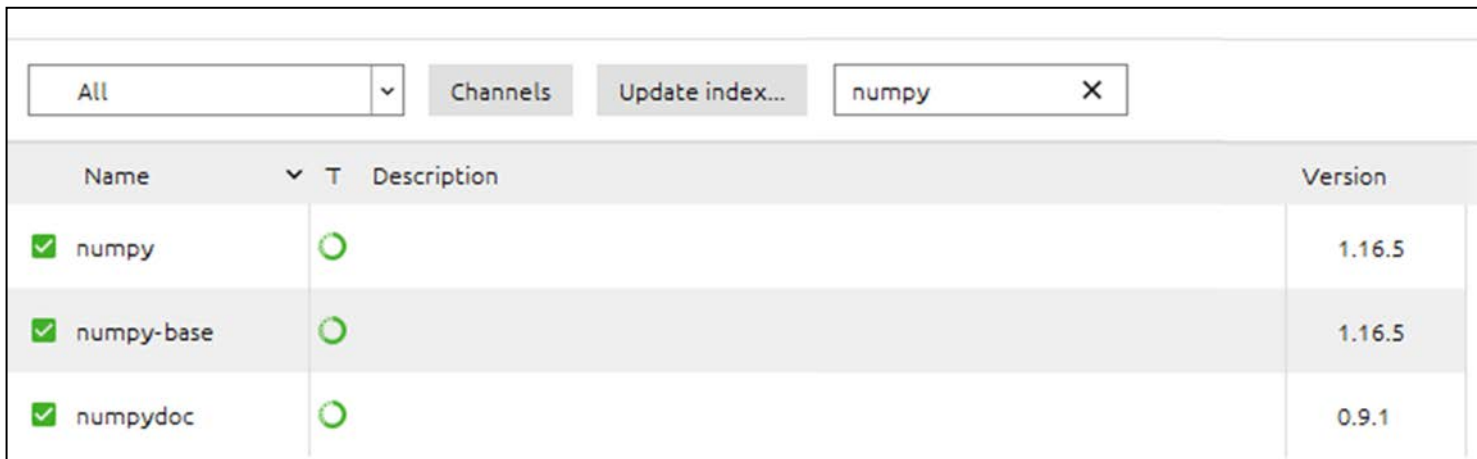
## 파이썬 설치

- 설치된 라이브러리 확인

→ Search Packages 박스에서 numpy와 matplotlib 검색

→ 만약 라이브러리가 검색되지 않을 경우, 해당 라이브러리는 설치되지 않았음을 의미함

→ 새로운 라이브러리는 Anaconda Prompt 에서 설치할 수 있음



The screenshot shows the Anaconda Search Packages interface. At the top, there is a search bar with 'numpy' entered, a 'Channels' button, and an 'Update index...' button. Below the search bar is a table with columns: Name, T, Description, and Version. The table lists three packages: numpy (version 1.16.5), numpy-base (version 1.16.5), and numpydoc (version 0.9.1). Each package has a green checkmark in the 'Name' column and a green circle in the 'T' column.

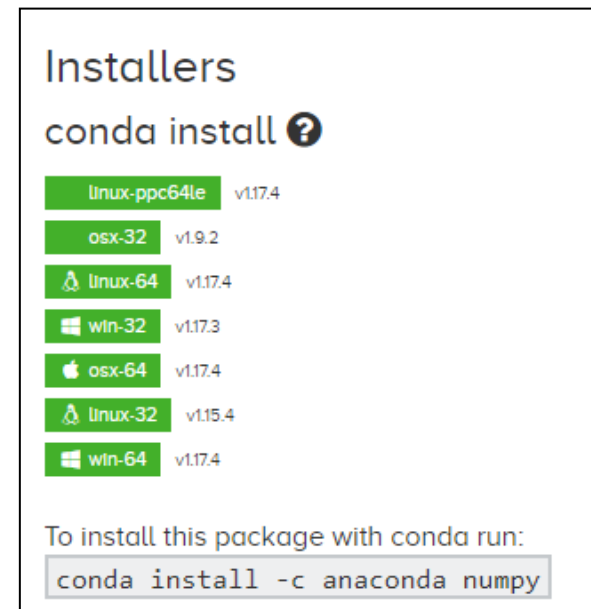
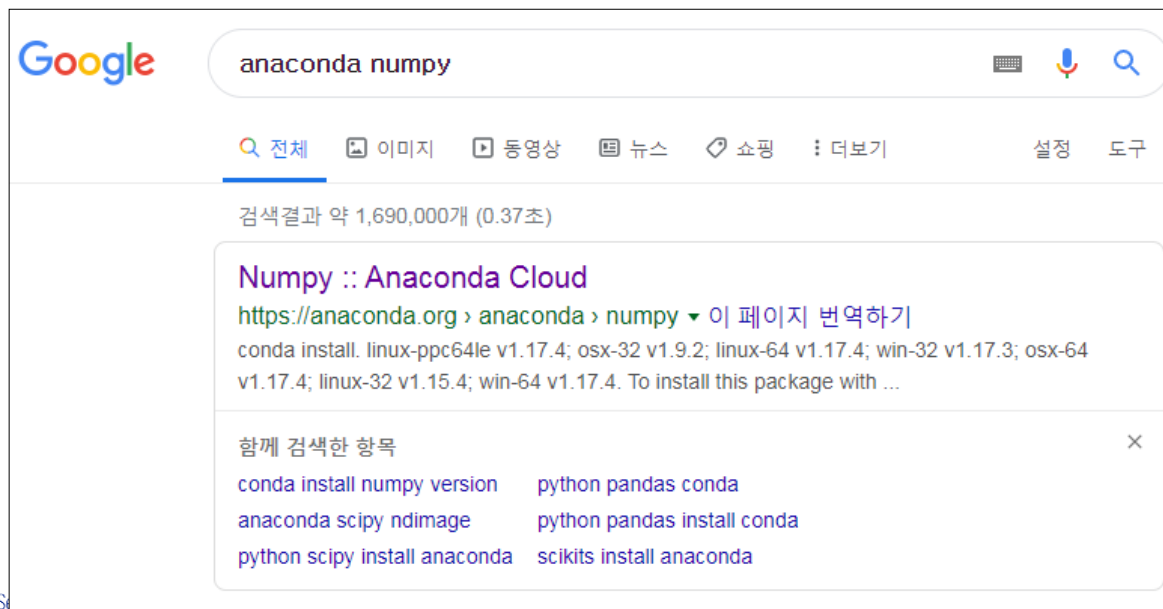
Name	T	Description	Version
✓ numpy	○		1.16.5
✓ numpy-base	○		1.16.5
✓ numpydoc	○		0.9.1

# 파이썬 설치

## • 라이브러리 설치

→ 구글에서 “anaconda [설치할 라이브러리]” 검색  
(ex: anaconda numpy)

→ 검색 결과로 나온 Anaconda Cloud 홈페이지에서 설치 명령어 확인



## 파이썬 설치

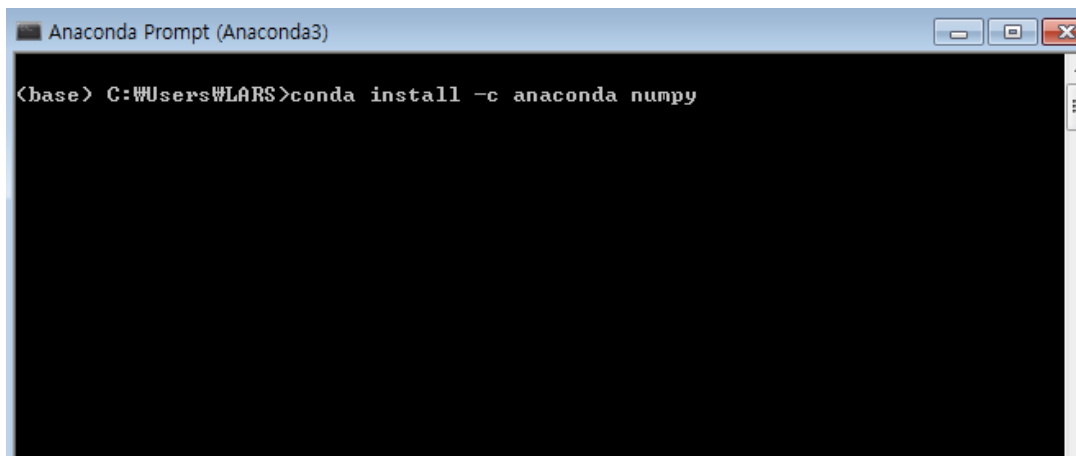
- 라이브러리 설치

→ Anaconda Prompt 실행

→ 라이브러리 설치 명령어 입력 (이미 설치가 되어있는 경우 할 필요 없음)

numpy : `conda install -c anaconda numpy`

matplotlib : `conda install -c conda-forge matplotlib`



```
Anaconda Prompt (Anaconda3)

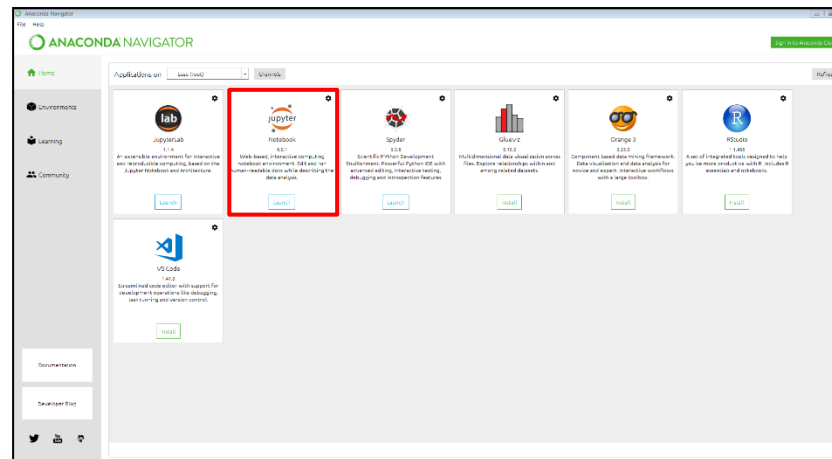
(base) C:\Users\W\ARS>conda install -c anaconda numpy
```

## 파이썬 설치

### • 주피터 노트북(Jupyter Notebook) 실행

→ 파이썬 명령어의 결과를 시각적으로 확인하기 위해서는 주피터 노트북을 이용하는 것이 편리함

→ 아나콘다 네비게이터에서 주피터 노트북 Launch를 클릭해 실행할 수 있음

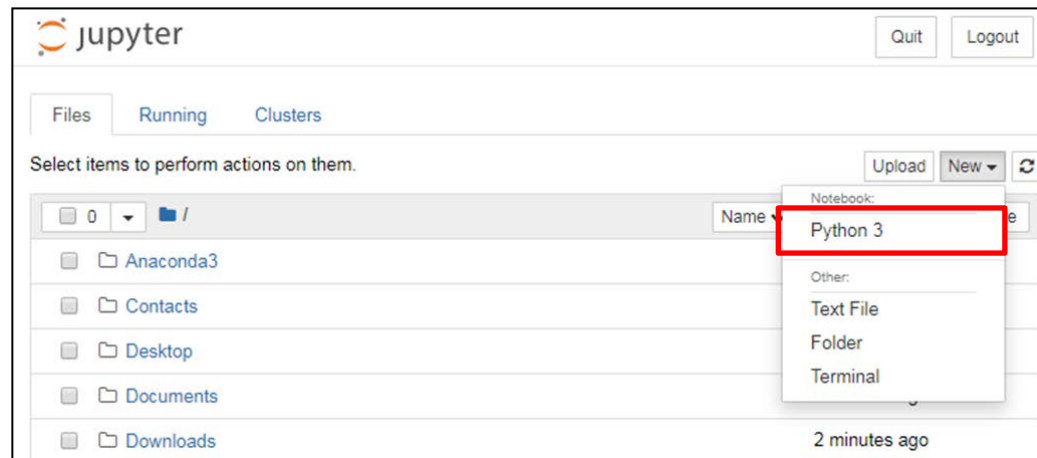


## 파이썬 설치

- 주피터 노트북(Jupyter Notebook) 실행

→ 주피터 노트북은 구글 크롬 브라우저에서 실행됨

→ 주피터 노트북을 생성할 폴더 위치로 이동한 후, 좌상단의 New 탭에서 Python 3 을 클릭하여 주피터 노트북을 생성할 수 있음



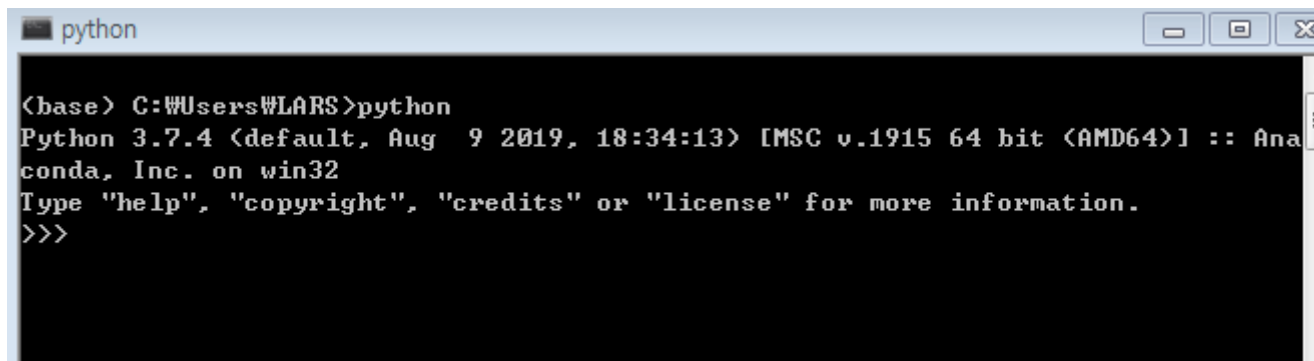
## 파이썬 설치

- **아나콘다 프롬프트(Anaconda Prompt) 실행**

→ 간단한 프로그래밍을 할 경우 아나콘다 프롬프트에서 하는 것이 편리함

→ 아나콘다 프롬프트에서 파이썬 프로그래밍을 하기 위해서는 python 이라는 명령어를 입력해야 함

→ Pytho 명령어를 입력하여 >>> 표시가 뜨면 실행이 된 것임



```
python

(base) C:\Users\W\ARS>python
Python 3.7.4 <default, Aug  9 2019, 18:34:13> [MSC v.1915 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

### 3) 파이썬 인터프리터 (Interpreter)

- 파이썬 인터프리터

→ 파이썬 버전 확인: 'python -version'

“Python 3.x.x”이면, 파이썬 버전 3가 제대로 설치된 상태

→ 파이썬 인터프리터는 ‘대화모드’라고 하며, 개발자와 파이썬이 대화하듯 프로그래밍이 가능함. 예를 들면 “1+2는?” 이라고 물으면, 파이썬 인터프리터가 “3”이라고 대답한다는 의미임.

Ex) >>> 1+2

## 파이썬 인터프리터

- 산술연산

→ 덧셈, 곱셈, 나누기 및 제곱승 연산

Ex)

```
>>> 1-2
```

```
>>> 4*5
```

```
>>> 7/5
```

```
>>> 3**2
```



## 파이썬 인터프리터

- 자료형 (data type)

→ 정수, 실수, 문자열

→ 파이썬에서는 `type()` 함수로 데이터의 자료형을 알 수 있음.

Ex)

```
>>> type(10)
```

```
>>> type(3.14)
```

```
>>> type("hello")
```

# 파이썬 인터프리터

## • 변수 (Variable)

- 알파벳을 이용하여 x, y, z 등과 같이 변수를 정의할 수 있음.
- 변수를 사용하여 다른 값을 대입할 수도 있음.

Ex)

```
>>> x = 10 #초기화
```

```
>>> print(x)
```

```
>>> x = 100 # 변수에 값 대입
```

```
>>> print(x)
```

```
>>> y = 3.14
```

```
>>> x*y
```

```
>>> type(x*y)
```

# 파이썬 인터프리터

- 리스트 (List)

→ 여러 데이터를 리스트로 정리.

Ex)

```
>>> a = [1,2,3,4,5] #list 생성
```

```
>>> print(a)
```

```
>>> len(a)
```

```
>>> a[0]
```

```
>>> a[4]
```

```
>>> a[4] = 99
```

```
>>> print(a)
```

## 파이썬 인터프리터

- 슬라이싱 (slicing)

→ 슬라이싱을 이용하면 범위를 지정해 원하는 부분의 리스트를 얻을 수 있음.

Ex)

```
>>> a = [1,2,3,4,5] #list 생성
```

```
>>> print(a)
```

```
>>> a[0:2] # 끝 숫자 불포함.
```

```
>>> a[1:]
```

```
>>> a[:3] # 끝 숫자 불포함
```

```
>>> a[:-1]
```

```
>>> a[:-2]
```

## 파이썬 인터프리터

- **딕셔너리 (dictionary)**

→ 리스트는 인덱스 번호로 값을 저장하는 반면, 딕셔너리는 키(key)와 값(value)를 한쌍으로 저장

Ex)

```
>>> you = {'height':180}
```

```
>>> you['height']
```

```
>>> you['weight'] = 70
```

```
>>> print(you)
```

## 파이썬 인터프리터

- **불 (bool)**

→ bool이라는 자료형은 참과 거짓을 취합니다. 참은 True이고, 거짓은 False  
→ Bool형은 and, or, not 연산자를 사용할 수 있음.

Ex)

```
>>> hungry = True
```

```
>>> sleepy = False
```

```
>>> type(hungry)
```

```
>>> not sleepy
```

```
>>> hungry and sleepy
```

```
>>> hungry or sleepy
```

## 파이썬 인터프리터

- 조건문 (if)

→ 조건에 따라 달리 처리할때는 if/else 문을 사용합니다.

Ex)

```
>>> hungry = True
```

```
>>> if hungry:
```

```
...     print ("I am hungry") # 공백 4칸 또는 탭 활용. 대개는 공백을 사용함.
```

```
...
```

```
>>> if not hungry:
```

```
...     print("I am hungry")
```

```
...else:
```

```
...     print("I am not hungry")
```

```
...     print("I am so sleepy")
```

```
...
```

파이썬에서는 공백문자가 중요한 의미를 지니기 때문에 잘 사용해야 함.

## 파이썬 인터프리터

- 루프문 (for)

→ 반복처리에는 for문을 사용함.

Ex)

```
>>> tlist = ['one','two','three']
```

```
>>> for i in tlist:
```

```
...     print (i)
```

```
...
```

```
>>> for i in range(1,11):
```

```
...     add = add + i
```

```
...
```

```
>>> print(add)
```

\* range는 숫자리스트를 자동으로 만들어주는 함수. range(1,10)이면,  
1,2,3,4,5,6,7,8,9의 값을 지니는 리스트를 만들어줌. range(10)=range(0,10)과  
동일함.



# 파이썬 인터프리터

## • 함수

→ 특정기능을 수행하는 일련의 명령들을 묶어서 하나의 함수로 정의가능  
Ex)

```
>>> def hello():  
...     print("Hello World!")  
...
```

```
>>> hello()
```

→ 입력변수를 취할 수도 있음.

```
>>> def hello(object):  
...     print("Hello" + object + "!")  
...
```

```
>>> hello("cat")
```

\* 파이썬 인터프리터의 종료는 윈도우의 경우, 컨트롤키와 Z키를 함께 누른다. 리눅스나 맥은 Z대신 D를 이용한다.

## 4) 파이썬 스크립트 파일

### • 스크립트 파일

→ 파이썬 인터프리터는 간단한 코드의 결과를 빠르게 실험하기 좋지만, 긴 작업을 수행해야 한다면, 매번 코드를 입력해야하는 불편함이 있음.

→ 파이썬에서는 스크립트 파일을 제공함.

→ 파일의 확장자는 .py임.

→ 텍스트 편집기를 열고 아래의 한줄을 입력한 후, hungry.py라는 파일이름으로 저장함.  
`print("I am very hungry!! I didn't have lunch today")`

→ 파일이 있는 디렉토리로 이동한 후, 아래를 실행함.

`$ python hungry.py`

## 파이썬 스크립트 파일

### • 클래스

- 파이썬에서는 클래스를 정의할 수 있음.
- 클래스가 정의되면, 1) 독자적인 자료형을 만들 수 있고, 2) 정의된 클래스만의 전용함수 (method)와 속성을 정의할 수 있음.
- 파이썬에서는 class라는 키워드로 클래스를 정의함.

---

```
class 클래스 이름:
    def __init__(self, 인수, ...):      # 생성자
        ...
    def 메서드 이름 1(self, 인수, ...):  # 메서드 1
        ...
    def 메서드 이름 2(self, 인수, ...):  # 메서드 2
        ...
```

---

## 파이썬 스크립트 파일

- 클래스

→ `__init__` 라는 메소드(생성자)가 주어지는데 클래스를 초기화하는 방법 정의.

→ `__init__`는 클래스의 인스턴스가 만들어질 때 딱 한번 실행됨.

→ 파이썬에서는 메서드의 첫 번째인수로 자신(자기 인스턴스)를 나타내는 `self`를 명시적으로 사용함.

→ 사람을 만났을 때 'Hello'라고 인사하고, 헤어질때 'Bye'라고 인사하는 클래스를 만들어보자. 그리고, 이 파일을 `person.py`로 저장하자. 그리고, 실행해보자

Ex)

```
$ python person.py
```

## 파이썬 스크립트 파일

- 클래스 (계속)

Ex)

Class Person:

```
def __init__(self, name):  
    self.name = name  
    print("Initialized!")
```

```
def hello(self):  
    print("Hello " + self.name + "!")
```

```
def bye(self):  
    print("Bye " + self.name + "!")
```

```
p = Person("Jung")
```

```
p.hello()
```

```
p.bye()
```

## 5) NumPy & matplotlib 라이브러리

### • NumPy 가져오기

→ 딥러닝 구현시 행렬 연산이 필수임.

→ NumPy의 경우, `numpy.array`에 행렬연산에 필요한 함수가 많이 준비되어 있어 딥러닝 구현시 필수적임.

→ NumPy는 외부 라이브러리기 때문에 라이브러리를 쓸 수 있도록 import해야 함.

```
>>> import numpy as np
```

→ “numpy를 np라는 이름으로 가져와라”는 의미. 이렇게 해두면 NumPy가 제공하는 method를 np를 통하여 참조할 수 있게 됨.

# NumPy & matplotlib 라이브러리

## • NumPy 배열

- NumPy 배열생성은 NumPy라이브러리의 `array()`라는 함수를 사용함
- `array`는 파이썬의 리스트를 인수로 받아서 NumPy라이브러리가 필요로하는 형태의 배열을 반환함.
- 이때 데이터의 타입은 `numpy.ndarray`임.

---

```
>>> x = np.array([1.0, 2.0, 3.0])
>>> print(x)
[1. 2. 3.]
>>> type(x)
<class 'numpy.ndarray'>
```

---

# NumPy & matplotlib 라이브러리

## • NumPy 산술연산

→ NumPy 배열의 연산은 원소별로 연산을 수행하기 때문에 기본적으로 배열의 크기가 같아야만 함.

```
>>> x = np.array([1.0, 2.0, 3.0])
>>> y = np.array([2.0, 4.0, 6.0])
>>> x + y # 원소별 덧셈
array([ 3.,  6.,  9.])
>>> x - y
array([ -1., -2., -3.])
>>> x * y # 원소별 곱셈
array([ 2.,  8., 18.])
>>> x / y
array([ 0.5,  0.5,  0.5])
```

### 배열과 스칼라 연산

```
>>> x = np.array([1.0, 2.0, 3.0])
>>> x / 2.0
array([ 0.5,  1.,  1.5])
```



## NumPy & matplotlib 라이브러리

- NumPy 다차원 배열

- NumPy 는 N차원 배열도 작성 가능
- shape는 행렬의 크기를 표현하고
- dtype은 데이터타입을 표현함.

---

```
>>> A = np.array([[1, 2], [3, 4]])  
>>> print(A)  
[[1 2]  
 [3 4]]  
>>> A.shape  
(2, 2)  
>>> A.dtype  
dtype('int64')
```

---

## NumPy & matplotlib 라이브러리

- NumPy N차원 배열 (계속)

---

```
>>> B = np.array([[3, 0], [0, 6]])  
>>> A + B  
array([[ 4,  2],  
       [ 3, 10]])  
>>> A * B  
array([[ 3,  0],  
       [ 0, 24]])
```

---

---

```
>>> print(A)  
[[1 2]  
 [3 4]]  
>>> A * 10  
array([[ 10, 20],  
       [ 30, 40]])
```

---

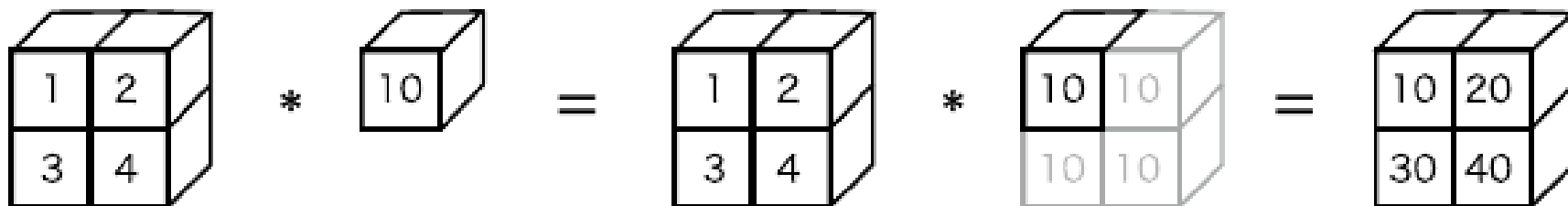
## NumPy & matplotlib 라이브러리

### • NumPy 브로드캐스트

→ NumPy에서 배열의 크기가 다른 경우에 확장하여 연산하는 것을 브로드캐스트라고 함.

```
>>> A * 10
array([[ 10, 20],
       [ 30, 40]])
```

그림 1-1 브로드캐스트의 예 : 스칼라값인 10이 2×2 행렬로 확대된다.

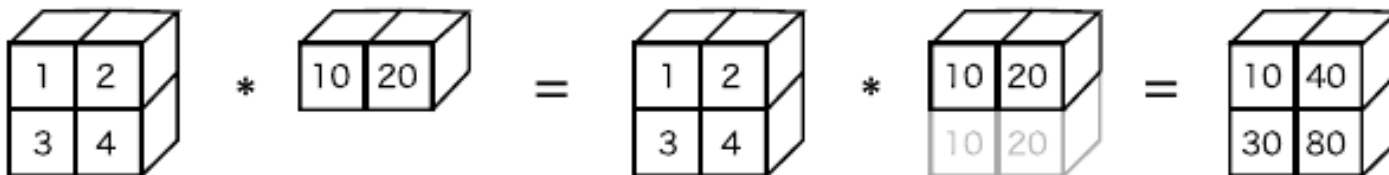


## NumPy & matplotlib 라이브러리

### • NumPy 브로드캐스트 (계속)

```
>>> A = np.array([[1, 2], [3, 4]])
>>> B = np.array([10, 20])
>>> A * B
array([[ 10, 40],
       [ 30, 80]])
```

그림 1-2 브로드캐스트의 예 2



이처럼 넘파이가 제공하는 브로드캐스트 기능 덕분에 형상이 다른 배열끼리의 연산을 스마트 함.  
하게 할 수 있습니다.

# NumPy & matplotlib 라이브러리

- NumPy 원소접근

→ 원소의 접근은 0부터 시작.

---

```
>>> X = np.array([[51, 55], [14, 19], [0, 4]])
>>> print(X)
[[51 55]
 [14 19]
 [ 0  4]]
>>> X[0]          # 0행
array([51, 55])
>>> X[0][1]       # (0, 1) 위치의 원소
55
```

---

## NumPy & matplotlib 라이브러리

- NumPy 원소접근 (계속)

→ for문으로 각 원소에 접근

```
>>> X = np.array([[51, 55], [14, 19], [0, 4]])  
>>> for row in X:  
...     print(row)  
...  
[51 55]  
[14 19]  
[0 4]
```

```
>>> for i in X:  
...     for j in i:  
...         print(j)
```

## NumPy & matplotlib 라이브러리

- NumPy 원소접근 (계속)

- 인덱스를 배열로 지정해 한번에 여러 요소에 접근 가능
- flatten함수는 배열을 1차원 배열로 만들어줌.

---

```
>>> X = np.array([[51, 55], [14, 19], [0, 4]])
>>> X = X.flatten()           # X를 1차원 배열로 변환(평탄화)
>>> print(X)
[51 55 14 19  0  4]
>>> X[np.array([0, 2, 4])]    # 인덱스가 0, 2, 4인 원소 얻기
array([51, 14,  0])
```

---

## NumPy & matplotlib 라이브러리

- NumPy 원소접근 (계속)

- 특정 조건을 만족하는 원소만을 추출.
- 배열 X에서 15이상인 원소 추출

---

```
>>> X = np.array([[51, 55], [14, 19], [0, 4]])  
>>> X = X.flatten()           # X를 1차원 배열로 변환(평탄화)  
>>> X > 15  
array([ True,  True, False,  True, False, False], dtype=bool)  
>>> X[X>15]  
array([51, 55, 19])
```

---

- bool배열을 이용하여 배열 X에서 True에 해당하는 원소만을 추출

\*



## NumPy & matplotlib 라이브러리

### • NumPy 원소접근 (계속)

→ 파이썬은 쉽지만, 속도가 늦어질 수 있어 빠른 성능이 요구되는 경우에는 해당부분을 C/C++로 구현함.

→ NumPy도 주된 처리는 C/C++로 구현되어 빠른 속도가 보장되면서도 파이썬의 편리한 문법을 사용할 수 있음.

**NOTE\_** 파이썬 같은 동적 언어는 C나 C++ 같은 정적 언어(컴파일 언어)보다 처리 속도가 느다고 합니다. 실제로 무거운 작업을 할 때는 C/C++로 작성한 프로그램을 쓰는 편이 좋습니다. 그래서 파이썬에서 빠른 성능이 요구될 경우 해당 부분을 C/C++로 구현하곤 합니다. 그때 파이썬은 C/C++로 쓰인 프로그램을 호출해주는, 이른바 '중개자' 같은 역할을 합니다. 넘파이도 주된 처리는 C와 C++로 구현했습니다. 그래서 성능을 해치지 않으면서 파이썬의 편리한 문법을 사용할 수 있는 것이죠.

## NumPy & matplotlib 라이브러리

- matplotlib로 그래프 그리기

→ 데이터를 시각화해주고, 그래프를 그려주는 라이브러리

→ 그래프를 그리기 위해서는 matplotlib의 pyplot 모듈을 이용함.

→ NumPy의 arrange 함수를 이용하여 변수 x에 [0,0.1,0.2,...5.9]라는 데이터를 생성

→ NumPy의 sin함수를 이용하여 변수 x에 대한 sin함수 값 y를 생성

→ matplotlib의 pyplot 모듈내의 plot이라고 하는 함수를 이용하여 그래프를 생성

→ show라는 함수를 이용하여 그래프를 화면에 출력

## NumPy & matplotlib 라이브러리

- matplotlib로 그래프 그리기 (계속)

→ NumPy와 matplotlib의 pyplot을 import하여 그래프를 작성.

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

```
# 데이터 준비
```

```
x = np.arange(0, 6, 0.1) # 0에서 6까지 0.1 간격으로 생성
```

```
y = np.sin(x)
```

```
# 그래프 그리기
```

```
plt.plot(x, y)
```

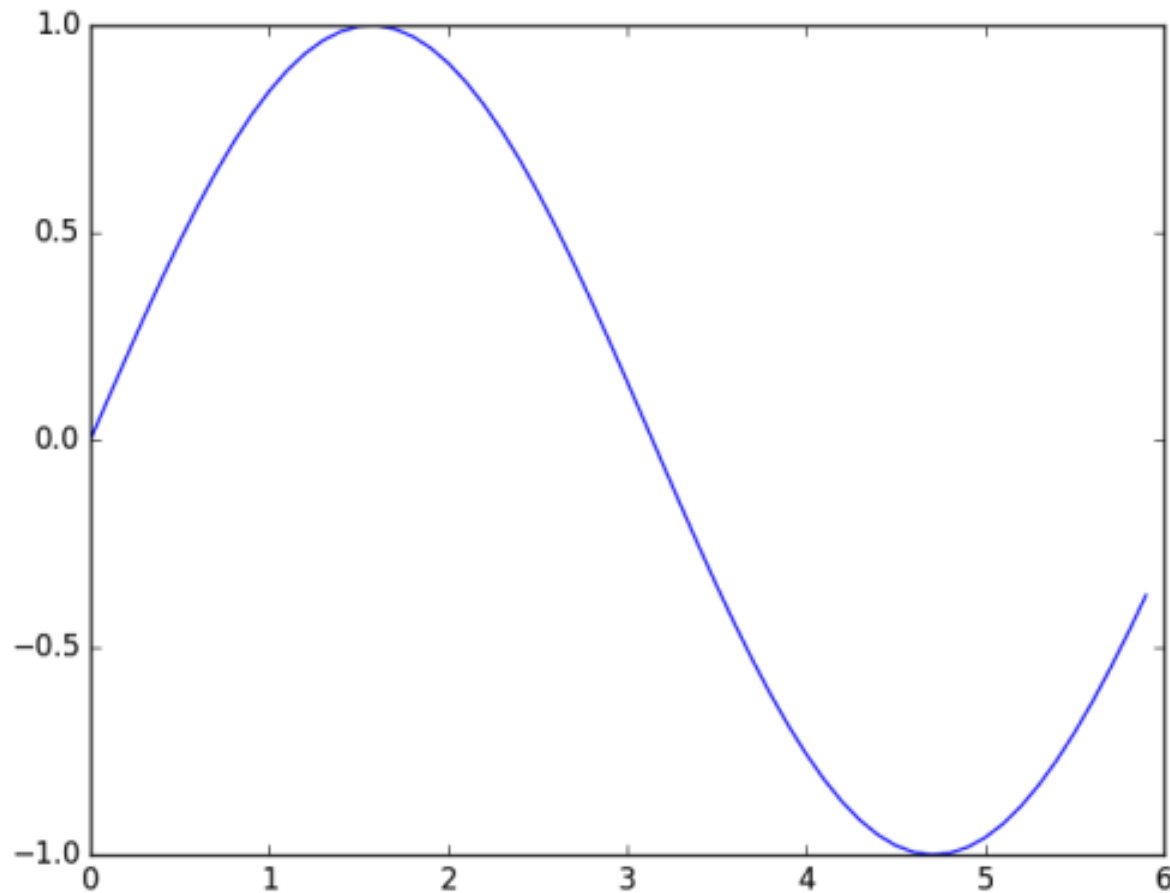
```
plt.show()
```

Ex) sin 함수 그래프 그리기

## NumPy & matplotlib 라이브러리

- matplotlib로 그래프 그리기 (계속)

그림 1-3 sin 함수 그래프



# NumPy & matplotlib 라이브러리

## • matplotlib로 그래프 그리기 (계속)

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

# 데이터 준비
x = np.arange(0, 6, 0.1) # 0에서 6까지 0.1 간격으로 생성
y1 = np.sin(x)
y2 = np.cos(x)

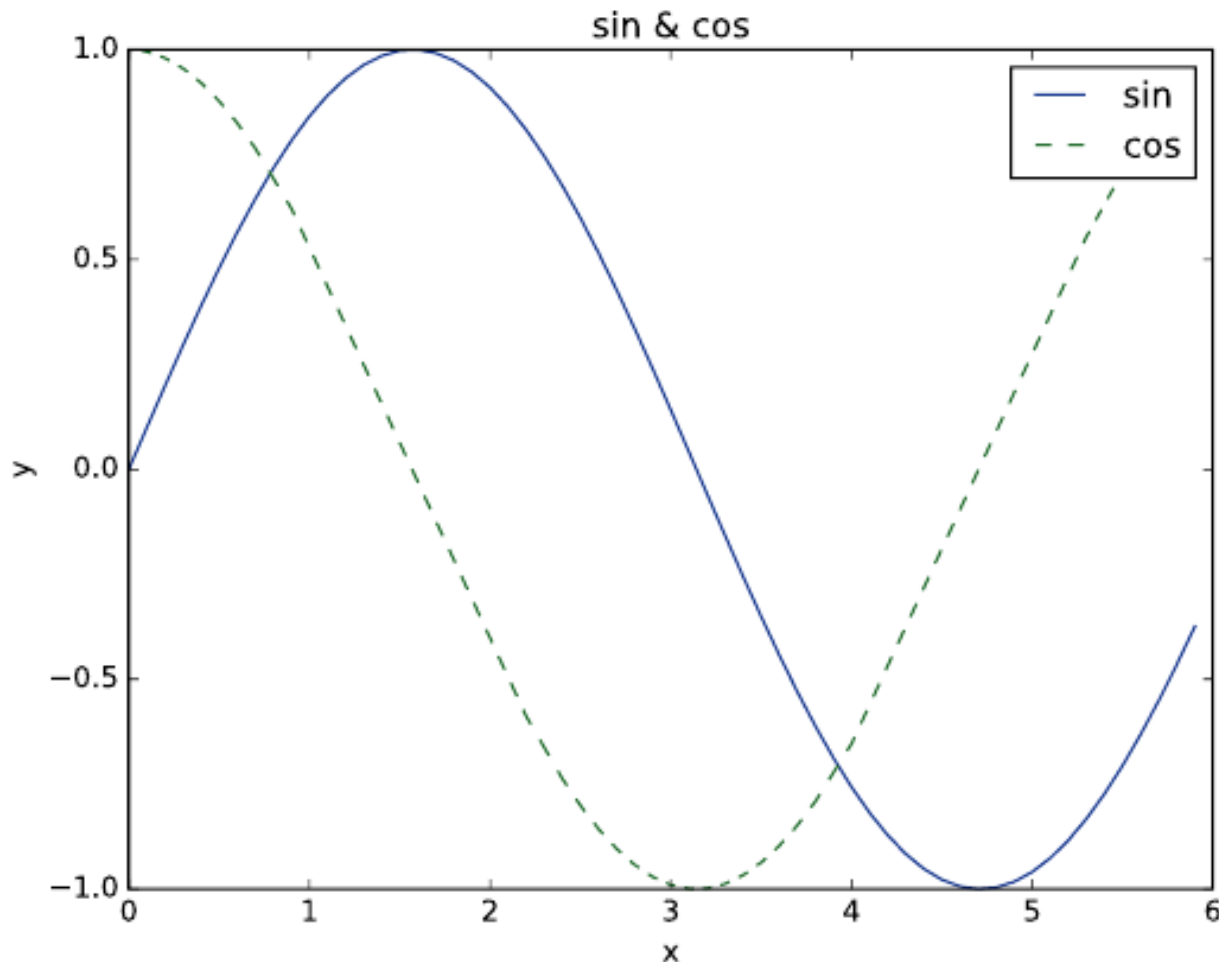
# 그래프 그리기
plt.plot(x, y1, label="sin")
plt.plot(x, y2, linestyle="--", label="cos") # cos 함수는 점선으로 그리기
plt.xlabel("x") # x축 이름
plt.ylabel("y") # y축 이름
plt.title('sin & cos') # 제목
plt.legend()
plt.show()
```

Ex) sin 및 cos 함수 그래프 그리기

# NumPy & matplotlib 라이브러리

## • matplotlib로 그래프 그리기 (계속)

그림 1-4 sin 함수와 cos 함수 그래프



## NumPy & matplotlib 라이브러리

- matplotlib로 이미지 표시하기

- pyplot 모듈에는 imshow라는 함수가 있으며, 이 함수는 이미지를 도시하는데 활용
- Matplotlib의 image 모듈의 imread 함수를 이용하여 영상을 읽어들이고, imshow함수로 이미지를 표현하고, show함수로 화면에 출력함.

---

```
import matplotlib.pyplot as plt
from matplotlib.image import imread

img = imread('lena.png') # 이미지 읽어오기(적절한 경로를 설정하세요!)

plt.imshow(img)
plt.show()
```

---

# NumPy & matplotlib 라이브러리

- matplotlib로 이미지 표시하기

그림 1-5 이미지 표시하기

