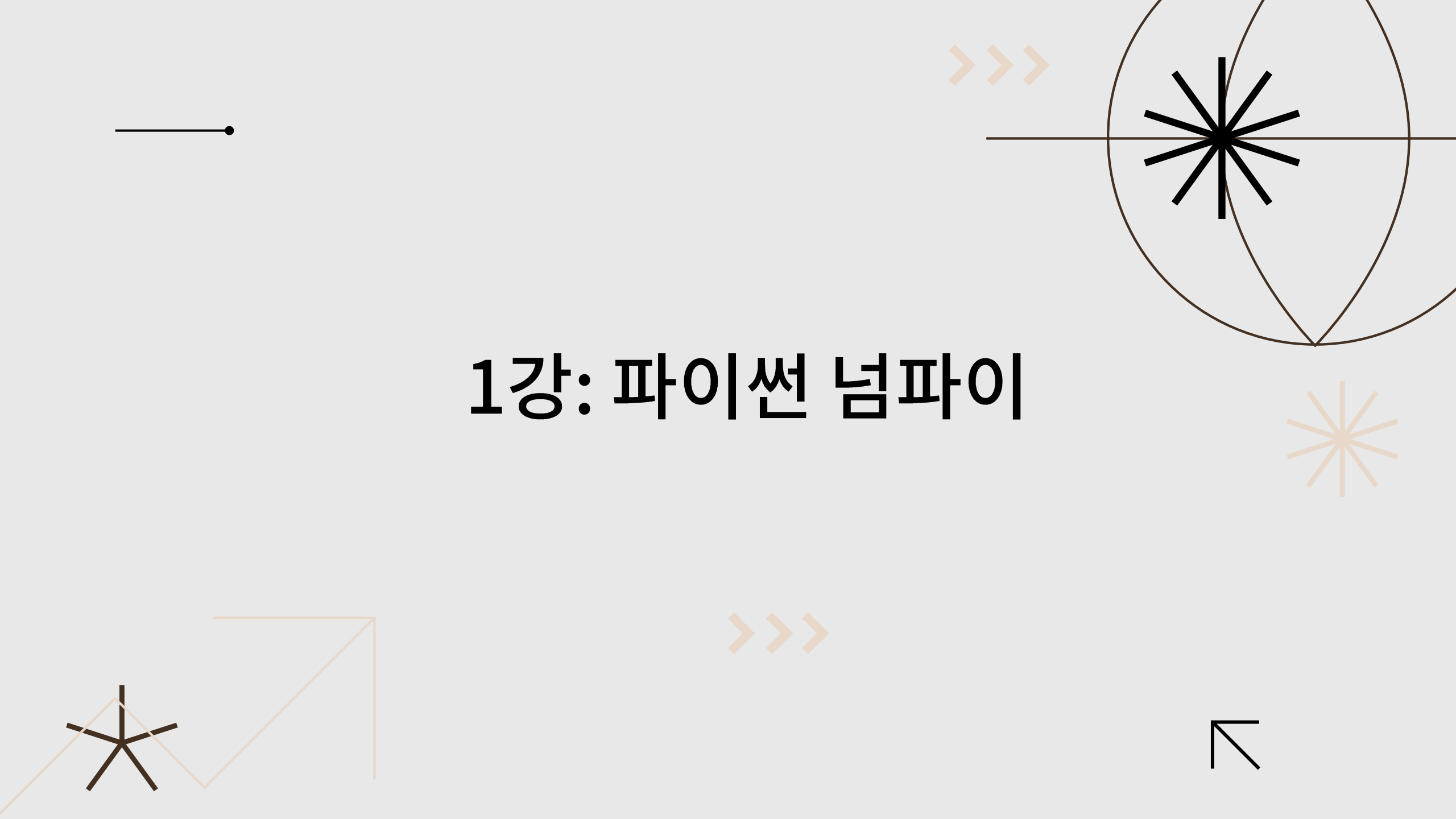
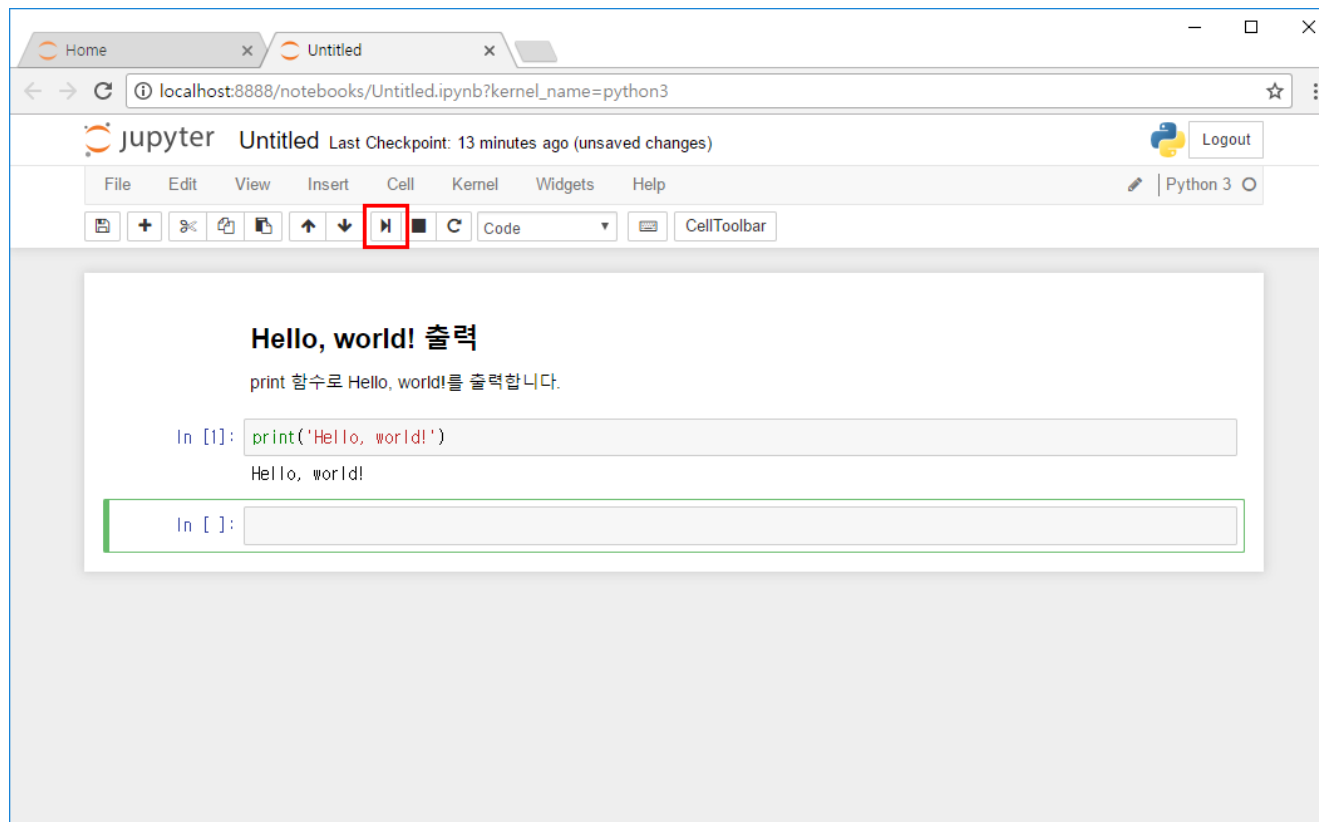


1강: 파이썬 넘파이



- 주피터 노트북(Jupyter notebook)
 - 웹 브라우저 내에서 로컬로 파이썬 코드 작성 및 실행 가능
 - 코드를 쉽게 수정하고 조각조각 실행할 수 있어 널리 사용됨



- 코랩(Colab)
 - 구글이 제공하는 주피터 노트북의 한 버전
 - 기계학습과 데이터 분석에 특히 적합하며 클라우드에서 실행됨
 - 무료이고 설정이 필요 없으며, 많은 패키지가 기본적으로 설치됨
 - GPU, TPU 등의 하드웨어 가속기에 액세스 가능



코랩에서 튜토리얼 실행

- <https://colab.research.google.com/github/cs231n/cs231n.github.io/blob/master/python-colab.ipynb>
- 위 코랩 노트북에서 파이썬/넘파이 튜토리얼 코드 실행 가능(권장 방법)

파이썬

- 파이썬은 고수준(high-level), 동적 타입의 다중 패러다임 프로그래밍 언어
- 읽기 쉬워서 종종 의사 코드와 거의 비슷하다고 여겨짐
- 이 수업에서는 모든 코드가 파이썬 3.7을 사용

파이썬 데이터 타입

- 파이썬에는 정수, 실수, 논리형(Boolean), 문자열과 같은 여러 기본 타입을 지원
- 이 데이터 타입들은 다른 프로그래밍 언어에서 익힌 방식과 유사하게 동작

숫자

```
x = 3
print(type(x)) # Prints "<class 'int'>"
print(x)       # Prints "3"
print(x + 1)    # Addition; prints "4"
print(x - 1)    # Subtraction; prints "2"
print(x * 2)    # Multiplication; prints "6"
print(x ** 2)   # Exponentiation; prints "9"
x += 1
print(x)        # Prints "4"
x *= 2
print(x)        # Prints "8"
y = 2.5
print(type(y))  # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

논리형(Boolean)

- 논리형은 일반적인 논리 연산자를 모두 지원하지만, 심볼(&&, || 등) 대신 영어 단어를 사용

```
t = True
f = False
print(type(t)) # Prints "<class 'bool'>"
print(t and f) # Logical AND; prints "False"
print(t or f)  # Logical OR; prints "True"
print(not t)   # Logical NOT; prints "False"
print(t != f)  # Logical XOR; prints "True"
```


문자열

```
hello = 'hello'      # String literals can use single quotes
world = "world"      # or double quotes; it does not matter.
print(hello)         # Prints "hello"
print(len(hello))    # String length; prints "5"
hw = hello + ' ' + world # String concatenation
print(hw)            # prints "hello world"
hw12 = '%s %s %d' % (hello, world, 12) # sprintf style string formatting
print(hw12)          # prints "hello world 12"
```

문자열

- 문자열 객체는 유용한 여러 메서드를 제공

```
s = "hello"
print(s.capitalize())  # Capitalize a string; prints "Hello"
print(s.upper())        # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))       # Right-justify a string, padding with spaces; prints "  hello"
print(s.center(7))      # Center a string, padding with spaces; prints " hello "
print(s.replace('l', '(ell)')) # Replace all instances of one substring with another;
                                # prints "he(ell)(ell)o"
print('  world '.strip()) # Strip leading and trailing whitespace; prints "world"
```

컨테이너

- 컨테이너(container)
 - 여러 개의 데이터를 하나의 단위로 관리하고 조작할 수 있도록 하는 객체
 - 대표적인 컨테이너
 - 리스트(list)
 - 딕셔너리(dictionary)
 - 세트(set)
 - 튜플(tuple)

리스트

- 배열에 상응하는 개념
- 크기 조절 가능하며 다양한 타입 요소 포함 가능

```
xs = [3, 1, 2]      # Create a list
print(xs, xs[2])    # Prints "[3, 1, 2] 2"
print(xs[-1])       # Negative indices count from the end of the list; prints "2"
xs[2] = 'foo'       # Lists can contain elements of different types
print(xs)           # Prints "[3, 1, 'foo']"
xs.append('bar')     # Add a new element to the end of the list
print(xs)           # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()         # Remove and return the last element of the list
print(x, xs)        # Prints "bar [3, 1, 'foo']"
```

리스트

- 슬라이싱을 통해 **부분 리스트**에 접근할 수 있음

```
nums = list(range(5))      # range is a built-in function that creates a list of integers
print(nums)               # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])          # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
print(nums[2:])            # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])           # Get a slice from the start to index 2 (exclusive); prints "[0, 1]"
print(nums[:])            # Get a slice of the whole list; prints "[0, 1, 2, 3, 4]"
print(nums[:-1])          # Slice indices can be negative; prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]        # Assign a new sublist to a slice
print(nums)               # Prints "[0, 1, 8, 9, 4]"
```

리스트 순회

- 리스트의 요소들에 대해 순회 가능

```
animals = ['cat', 'dog', 'monkey']  
for animal in animals:  
    print(animal)  
  
# Prints "cat", "dog", "monkey", each on its own line.
```

- 각 요소 인덱스에 접근하려면, 내장 함수 enumerate를 사용

```
animals = ['cat', 'dog', 'monkey']  
for idx, animal in enumerate(animals):  
    print('#%d: %s' % (idx + 1, animal))  
  
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

리스트 컴프리헨션

- 리스트 컴프리헨션은 리스트를 생성하는 간결하고 읽기 쉬운 방법
- 반복문과 조건문을 사용해 한 줄로 리스트 생성 가능
- 예제)

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)    # Prints [0, 1, 4, 9, 16]
```

리스트 컴프리헨션

- 리스트 컴프리헨션을 통해 더 간단하게 생성 가능

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)    # Prints [0, 1, 4, 9, 16]
```

- 리스트 컴프리헨션은 조건문도 포함 가능

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)    # Prints "[0, 4, 16]"
```


딕셔너리

- 딕셔너리(dictionary)는 (키, 값) 쌍을 저장

```
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
print(d['cat'])                      # Get an entry from a dictionary; prints "cute"
print('cat' in d)                    # Check if a dictionary has a given key; prints "True"
d['fish'] = 'wet'                    # Set an entry in a dictionary
print(d['fish'])                      # Prints "wet"
# print(d['monkey']) # KeyError: 'monkey' not a key of d
print(d.get('monkey', 'N/A'))        # Get an element with a default; prints "N/A"
print(d.get('fish', 'N/A'))          # Get an element with a default; prints "wet"
del d['fish']                         # Remove an element from a dictionary
print(d.get('fish', 'N/A'))          # "fish" is no longer a key; prints "N/A"
```

딕셔너리 키 순회

- 딕셔너리의 키 순회

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

- 키 뿐만 아니라 값에도 접근하려면 items 메서드 사용

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

딕셔너리 컴프리헨션

- 딕셔너리 컴프리헨션(dictionary comprehension)은 딕셔너리를 쉽게 구성할 수 있게 함

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)  # Prints "{0: 0, 2: 4, 4: 16}"
```

셋

- 셋(set)은 중복을 허용하지 않는, 순서가 없는 컬렉션

```
animals = {'cat', 'dog'}
print('cat' in animals)    # Check if an element is in a set; prints "True"
print('fish' in animals)   # prints "False"
animals.add('fish')        # Add an element to a set
print('fish' in animals)   # Prints "True"
print(len(animals))        # Number of elements in a set; prints "3"
animals.add('cat')         # Adding an element that is already in the set does nothing
print(len(animals))        # Prints "3"
animals.remove('cat')      # Remove an element from a set
print(len(animals))        # Prints "2"
```

셋 순회

- 셋 순회는 리스트 순회와 동일

```
animals = {'cat', 'dog', 'fish'}  
for animal in animals:  
    print(animal)
```

```
animals = {'cat', 'dog', 'fish'}  
for idx, animal in enumerate(animals):  
    print('#%d: %s' % (idx + 1, animal))  
# Prints "#1: fish", "#2: dog", "#3: cat"
```

- 셋은 순서가 없으므로, 셋의 요소를 어떤 순서로 방문할지에 대한 가정은 할 수 없음

셋 컴프리헨션

- 셋 컴프리헨션(set comprehension)을 통해 쉽게 셋 구성 가능

```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums)    # Prints "{0, 1, 2, 3, 4, 5}"
```

튜플

- 튜플(tuple)은 값의 정렬된 목록
- 튜플은 리스트와 달리 값의 추가/수정/삭제가 불가능
- 튜플은 딕셔너리의 키나 셋의 요소로 사용될 수 있는 반면 리스트는 그렇게 할 수 없음

```
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
t = (5, 6)                             # Create a tuple
print(type(t))                         # Prints "<class 'tuple'>"
print(d[t])                            # Prints "5"
print(d[(1, 2)])                       # Prints "1"
```

- 파이썬 함수는 def 키워드를 사용하여 정의됨

```
def sign(x):  
    if x > 0:  
        return 'positive'  
    elif x < 0:  
        return 'negative'  
    else:  
        return 'zero'  
  
for x in [-1, 0, 1]:  
    print(sign(x))  
  
# Prints "negative", "zero", "positive"
```


- 매개변수에 디폴트값 설정 가능
- 매개변수에 하나 덜 넣으면 디폴트 값을 사용

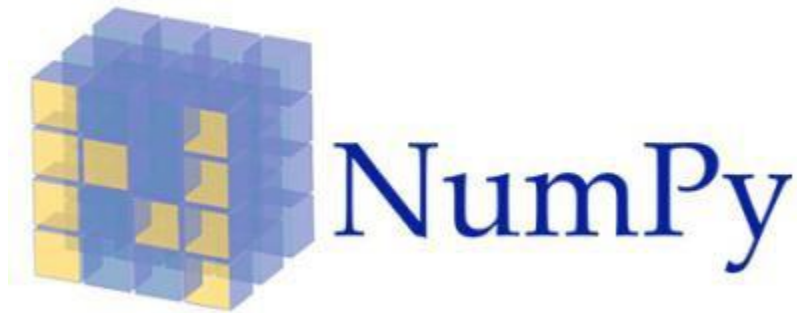
```
def hello(name, loud=False):  
    if loud:  
        print('HELLO, %s!' % name.upper())  
    else:  
        print('Hello, %s' % name)  
  
hello('Bob') # Prints "Hello, Bob"  
hello('Fred', loud=True) # Prints "HELLO, FRED!"
```

- 파이썬 클래스는 class 키워드를 사용하여 정의됨

```
class Greeter(object):  
  
    # Constructor  
    def __init__(self, name):  
        self.name = name # Create an instance variable  
  
    # Instance method  
    def greet(self, loud=False):  
        if loud:  
            print('HELLO, %s!' % self.name.upper())  
        else:  
            print('Hello, %s' % self.name)  
  
g = Greeter('Fred') # Construct an instance of the Greeter class  
g.greet() # Call an instance method; prints "Hello, Fred"  
g.greet(loud=True) # Call an instance method; prints "HELLO, FRED!"
```

넘파이(Numpy)

- 넘파이(Numpy)는 파이썬에서 과학 계산을 위한 핵심 라이브러리
- 고성능의 다차원 배열 객체와 이 배열을 다루는 도구를 제공함



넘파이 배열(array)

- 넘파이 배열(array)는 같은 타입의 값들로 이루어진 고차원 배열 데이터 타입
- 음이 아닌 정수 순서쌍에 의해 인덱싱 됨
- 배열의 차원 수를 랭크(rank)라고 하며 배열의 형태(shape)는 각 차원을 따라 배열의 크기를 나타내는 정수 튜플
- 중첩된 파이썬 리스트로부터 고차원 배열 초기화 가능
- 대괄호를 사용하여 요소에 접근 가능

넘파이 배열(array)

```
import numpy as np

a = np.array([1, 2, 3])    # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]]) # Create a rank 2 array
print(b.shape)            # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0]) # Prints "1 2 4"
```

넘파이 배열 생성 함수

- 넘파이는 배열 생성을 위한 여러 함수를 제공

```
import numpy as np

a = np.zeros((2,2))    # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #           [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)              # Prints "[[ 7.  7.]
                      #           [ 7.  7.]]"

d = np.eye(2)          # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #           [ 0.  1.]]"

e = np.random.random((2,2)) # Create an array filled with random values
print(e)              # Might print "[[ 0.91940167  0.08143941]
                      #           [ 0.68744134  0.87236687]]"
```

배열 인덱싱

- 넘파이는 배열에 인덱싱하는 몇 가지 방법을 제공함
 - 슬라이싱(slicing)
 - 정수 배열 인덱싱(integer array indexing)
 - 논리형 배열 인덱싱(Boolean array indexing)

슬라이싱

- 파이썬 리스트와 유사하게 넘파일 배열도 슬라이싱 가능
- 다차원 배열의 경우 각 차원에 대한 슬라이스를 명시해야함

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])    # Prints "2"
b[0, 0] = 77      # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])    # Prints "77"
```


슬라이싱

- 넘파이 배열의 슬라이스는 **원본 배열과 메모리를 공유함**
- 따라서 **슬라이스를 수정하면 원본 배열도 수정됨**
- 이를 **뷰(view)**라고 부르며, 이 특성 때문에 넘파이는 큰 데이터 세트를 효율적으로 처리할 수 있지만, 부주의한 사용은 예기치 않은 결과를 초래함

배열 a (형태: 4x5)

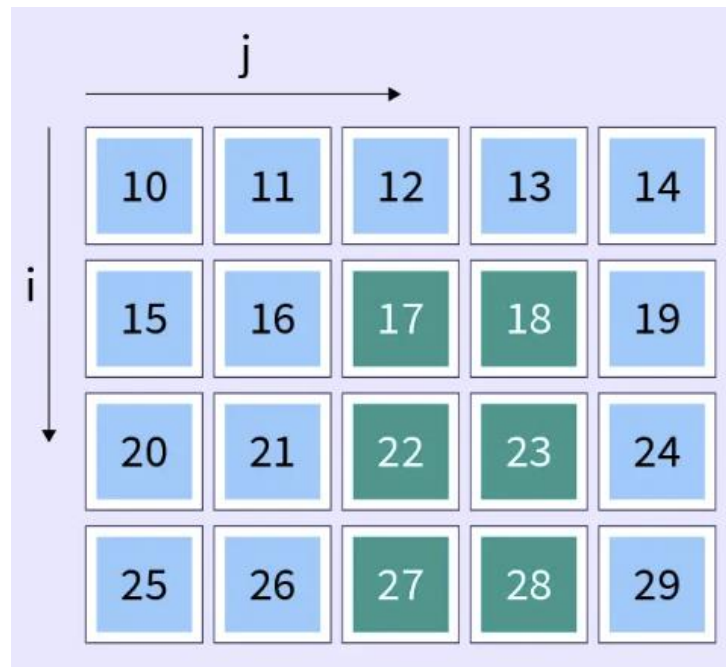
배열 b = a[1:4, 2:4]

```
# A slice of an array is a view into the same data, so modifying it  
# will modify the original array.
```

```
print(a[0, 1])    # Prints "2"
```

```
b[0, 0] = 77      # b[0, 0] is the same piece of data as a[0, 1]
```

```
print(a[0, 1])    # Prints "77"
```



슬라이싱

- 정수 인덱싱과 슬라이스 인덱싱을 혼합하여 사용하는 것도 가능
- 이렇게 할 경우 원래 배열보다 랭크가 낮은 배열이 생성됨

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"
```

슬라이싱

```
# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape) # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape) # Prints "[[ 2
                                #           [ 6]
                                #           [10]] (3, 1)"
```

정수 배열 인덱싱

- 슬라이싱을 사용하는 경우와 달리 정수 인덱싱만 사용하는 경우 원본 배열과 메모리를 공유하지 않음
- 정수 배열 인덱싱을 사용하는 경우에도 원본 배열과 메모리를 공유하지 않고 새로운 배열을 생성함

```
import numpy as np

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

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]]) # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]])) # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]]) # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]])) # Prints "[2 2]"
```

정수 배열 인덱싱

- 정수 배열 인덱싱의 유용한 트릭 중 하나는 행렬의 각 행에서 하나의 요소를 선택하거나 변경하는 것

```
import numpy as np

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a) # prints "array([[ 1,  2,  3],
#           [ 4,  5,  6],
#           [ 7,  8,  9],
#           [10, 11, 12]])"

# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b]) # Prints "[ 1  6  7 11]"

# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10

print(a) # prints "array([[11,  2,  3],
#           [ 4,  5, 16],
#           [17,  8,  9],
#           [10, 21, 12]])"
```

정수 배열 인덱싱

- 정수 배열 인덱싱 사용 시 리스트나 넘파이 배열 모두 사용 가능

```
import numpy as np

# 원본 넘파이 배열
a = np.array([0, 1, 2, 3, 4, 5])

# 파이썬 리스트를 사용한 정수 배열 인덱싱
b = a[[1, 3, 4]] # b는 [1, 3, 4]

# 넘파이 배열을 사용한 정수 배열 인덱싱
indices = np.array([1, 3, 4])
c = a[indices] # c는 [1, 3, 4]

print(b) # 출력: [1 3 4]
print(c) # 출력: [1 3 4]
```

논리 연산자

- 배열에 논리 연산자를 사용하면 각 성분에 대해 논리연산자를 적용한 배열을 반환

```
import numpy as np

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

bool_idx = (a > 2)    # Find the elements of a that are bigger than 2;
                       # this returns a numpy array of Booleans of the same
                       # shape as a, where each slot of bool_idx tells
                       # whether that element of a is > 2.

print(bool_idx)        # Prints "[[False False]
                       #           [ True  True]
                       #           [ True  True]]"
```

논리형 배열 인덱싱

- 논리형 배열 인덱싱은 배열의 임의의 요소를 선택하게 함
- 어떤 조건을 만족하는 요소를 선택하는데 사용됨

```
import numpy as np

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

bool_idx = (a > 2)  # Find the elements of a that are bigger than 2;
                    # this returns a numpy array of Booleans of the same
                    # shape as a, where each slot of bool_idx tells
                    # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                    #          [ True  True]
                    #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```


배열 데이터 타입

- 모든 넘파이 배열은 **같은 타입의 요소**로 이루어진 그리드
- 넘파이 배열은 다양한 숫자 데이터 타입을 제공

```
import numpy as np

x = np.array([1, 2])    # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0]) # Let numpy choose the datatype
print(x.dtype)         # Prints "float64"

x = np.array([1, 2], dtype=np.int64) # Force a particular datatype
print(x.dtype)         # Prints "int64"
```

배열 수학

- 기본 수학 함수는 **배열에 요소별로 작동**
- 넘파이 모듈 내의 함수 및 기본 연산자 모두 사용 가능

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))
```

```
# Elementwise product; both produce the array  
# [[ 5.0 12.0]  
# [21.0 32.0]]  
print(x * y)  
print(np.multiply(x, y))  
  
# Elementwise division; both produce the array  
# [[ 0.2          0.33333333]  
# [ 0.42857143  0.5         ]]  
print(x / y)  
print(np.divide(x, y))  
  
# Elementwise square root; produces the array  
# [[ 1.          1.41421356]  
# [ 1.73205081  2.         ]]  
print(np.sqrt(x))
```

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

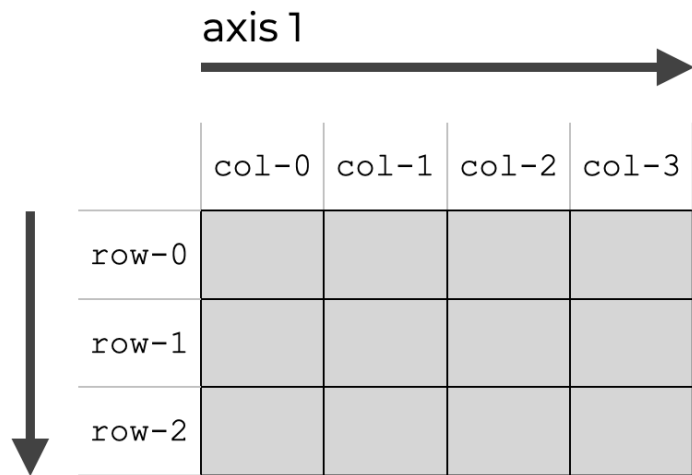
# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

배열 수학

- sum 함수는 요소들의 합을 계산
- 인자 axis는 어떤 축을 따라 더할 것인지 결정



The diagram shows a 2D array grid with 3 rows and 4 columns. A horizontal arrow at the top points to the right, labeled 'axis 1'. A vertical arrow on the left points downwards, labeled 'axis 0'. The columns are labeled 'col-0', 'col-1', 'col-2', and 'col-3' at the top. The rows are labeled 'row-0', 'row-1', and 'row-2' on the left. The grid cells are shaded gray.

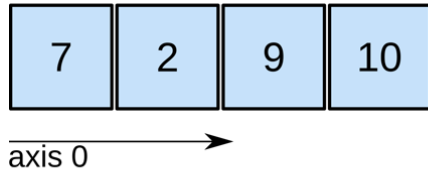
	col-0	col-1	col-2	col-3
row-0				
row-1				
row-2				

```
import numpy as np

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

print(np.sum(x))    # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

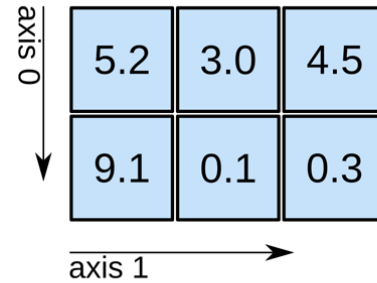
1D array



shape: (4,)

`np.sum(x, axis=0)`

2D array

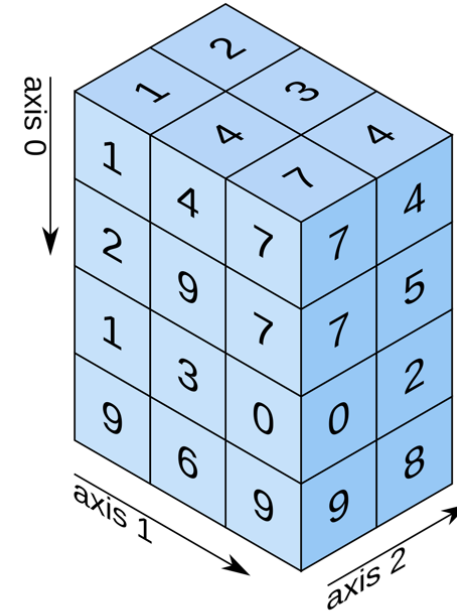


shape: (2, 3)

`np.sum(x, axis=0)`

`np.sum(x, axis=1)`

3D array



shape: (4, 3, 2)

`np.sum(x, axis=0)`

`np.sum(x, axis=1)`

`np.sum(x, axis=2)`

축 번호

- 중첩 리스트를 넘파이 배열로 바꾸는 경우, 가장 안쪽에 있는 대괄호는 가장 높은 축(axis)에 해당되며, 바깥쪽으로 나갈 수록 축의 번호가 줄어듦

```
import numpy as np

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

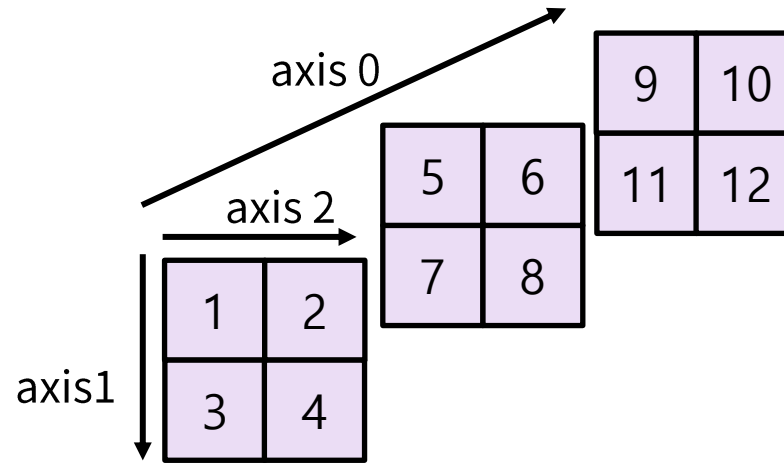
- 가장 안쪽 대괄호 안 원소들 [1,2], [3,4] 등은 axis 2를 따라 나열됨
- 다음 단계 대괄호 안 원소들 [[1,2], [3,4]], [[5,6], [7,8]] 등은 axis 1을 따라 나열됨
- 가장 바깥쪽 대괄호 안 원소들 [[[1, 2], [3, 4]], [[5, 6], [7, 8]], [[9, 10], [11, 12]]] 은 axis 0을 따라 나열됨

축 번호

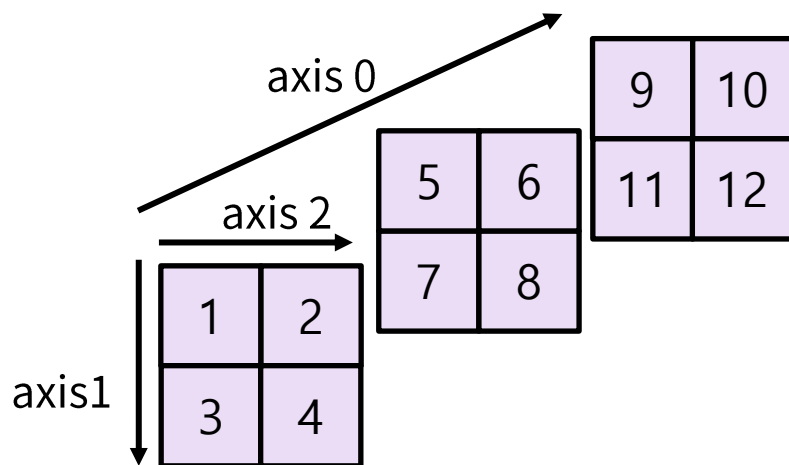
```
import numpy as np
```

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

- axis 0: 깊이
- axis 1: 세로
- axis 2: 가로



축 번호



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

axis=0: 첫 번째 축을 따라 원소들을 더함

```
result_axis0 = np.sum(c, axis=0)
```

```
print(result_axis0)
```

출력:

```
# [[15 18]
```

```
# [21 24]]
```

axis=1: 두 번째 축을 따라 원소들을 더함

```
result_axis1 = np.sum(c, axis=1)
```

```
print(result_axis1)
```

출력:

```
# [[ 4  6]
```

```
# [12 14]
```

```
# [20 22]]
```

axis=2: 세 번째 축을 따라 원소들을 더함

```
result_axis2 = np.sum(c, axis=2)
```

```
print(result_axis2)
```

출력:

```
# [[ 3  7]
```

```
# [11 15]
```

```
# [19 23]]
```

- 행렬의 전치(transpose)는 객체의 T 속성을 사용 가능

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)      # Prints "[[1 2]
               #           [3 4]]"
print(x.T)    # Prints "[[1 3]
               #           [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)      # Prints "[1 2 3]"
print(v.T)    # Prints "[1 2 3]"
```

브로드캐스팅(broadcasting)

- 브로드캐스팅은 연산 시 **다양한 형태의 배열로 작업하게 하는 강력한 메커니즘**
- 종종 작은 배열과 큰 배열을 연산할 때, 작은 배열을 반복하여 큰 배열을 만든 후 연산을 수행

브로드캐스팅(broadcasting)

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x) # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

브로드캐스팅(broadcasting)

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1)) # Stack 4 copies of v on top of each other
print(vv)                # Prints "[[1 0 1]
                          #           [1 0 1]
                          #           [1 0 1]
                          #           [1 0 1]]"

y = x + vv # Add x and vv elementwise
print(y) # Prints "[[ 2  2  4
          #           [ 5  5  7]
          #           [ 8  8 10]
          #           [11 11 13]]"
```

브로드캐스팅(broadcasting)

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v # Add v to each row of x using broadcasting
print(y) # Prints "[[ 2  2  4]
          #           [ 5  5  7]
          #           [ 8  8 10]
          #           [11 11 13]]"
```

브로드캐스팅(broadcasting)

- 브로드캐스팅 규칙
 - 배열의 랭크(차원)가 같지 않으면, 낮은 배열의 형태에 1을 추가하여 랭크가 같도록 함
 - ex) $(4,3)$ 배열에 $(3,)$ 배열을 더할 때 $(3,) \Rightarrow (1,3)$
 - 두 배열은 어떤 차원에서 크기가 같거나, 그 차원에서 하나의 배열이 크기 1을 가질 경우 그 차원에서 호환 가능하다고 함
 - ex) $(4,3)$ 과 $(1,3)$ 은 두 차원 모두에서 호환 가능

브로드캐스팅(broadcasting)

- 브로드캐스팅 규칙
 - 모든 차원에서 호환 가능하면 두 배열은 브로드캐스팅 가능
 - ex) $(4,1,3)$ 형태 배열과 $(4,5,3)$ 형태 배열은 브로드캐스팅 가능
 - 브로드캐스팅 후, 각 배열은 마치 두 입력 배열 형태의 성분별 최대값과 동일한 형태를 가짐
 - ex) $(4,1,3)$ 형태 배열과 $(4,5,1)$ 형태 배열을 더하면 $(4,5,3)$ 형태
 - 어떤 차원에서 하나의 배열의 크기가 1이고 다른 배열 크기가 1보다 큰 경우, 첫 번째 배열은 그 차원을 따라 복사됨

브로드캐스팅 사례

```
import numpy as np

# Compute outer product of vectors
v = np.array([1,2,3]) # v has shape (3,)
w = np.array([4,5])   # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6]
#  [5 7 9]]
print(x + v)
```

브로드캐스팅 사례

- reshape 함수의 기능

- 차원 추가

```
import numpy as np

A = np.random.rand(500, 3072) # 예제 데이터

reshaped_A = A.reshape(500, 1, 3072)
```

- 재배열

- 원래 배열의 원소를 변경하지 않고 주어진 새로운 형태에 따라 원소를 재배열

```
import numpy as np

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

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

브로드캐스팅 사례

```
import numpy as np

# Compute outer product of vectors
v = np.array([1,2,3]) # v has shape (3,)
w = np.array([4,5])   # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6]
#  [5 7 9]]
print(x + v)
```

브로드캐스팅 주의사항

- 두 배열의 차원 수가 다르면, 더 적은 수의 차원을 갖는 배열의 형태의 앞쪽을 1로 채워 차원 크기가 동일하도록 함
- 아래 예는 (4,3) 형태와 (4,) 형태의 배열을 더하는데 오류가 발생

```
import numpy as np

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

```
C = A + B
```



SciPy

이미지 작업

- SciPy는 이미지 작업을 위한 몇 가지 기본 함수 제공
- 예시)
 - 디스크에서 이미지를 넘파이 배열로 읽어들이는 함수
 - 넘파이 배열을 디스크에 이미지로 쓰는 함수
 - 이미지의 크기를 조정하는 함수

이미지 작업

```
from scipy.misc import imread, imsave, imresize

# Read an JPEG image into a numpy array
img = imread('assets/cat.jpg')
print(img.dtype, img.shape) # Prints "uint8 (400, 248, 3)"

# We can tint the image by scaling each of the color channels
# by a different scalar constant. The image has shape (400, 248, 3);
# we multiply it by the array [1, 0.95, 0.9] of shape (3,);
# numpy broadcasting means that this leaves the red channel unchanged,
# and multiplies the green and blue channels by 0.95 and 0.9
# respectively.
img_tinted = img * [1, 0.95, 0.9]

# Resize the tinted image to be 300 by 300 pixels.
img_tinted = imresize(img_tinted, (300, 300))

# Write the tinted image back to disk
imsave('assets/cat_tinted.jpg', img_tinted)
```



이미지 작업 - 색조 변경

```
# We can tint the image by scaling each of the color channels  
# by a different scalar constant. The image has shape (400, 248, 3);  
# we multiply it by the array [1, 0.95, 0.9] of shape (3,);  
# numpy broadcasting means that this leaves the red channel unchanged,  
# and multiplies the green and blue channels by 0.95 and 0.9  
# respectively.  
img_tinted = img * [1, 0.95, 0.9]
```

이미지 작업 - 색조 변경

- 넘파이 배열과 (중첩) 리스트 간에서도 +, -, * 등의 연산자를 통해 성분별 연산 수행 가능

```
import numpy as np

# 넘파이 배열 생성
x = np.array([[1, 2], [3, 4]])

# 중첩 리스트 생성
y = [[2, 3], [4, 5]]

# 요소별 곱셈 수행
result = x * y
print(result)
```

```
[[ 2  6]
 [12 20]]
```

```
import numpy as np

# 넘파이 배열 생성
x = np.array([[1, 2], [3, 4]])

# 중첩 리스트 생성
y = [[2, 3], [4, 5]]

# 요소별 덧셈
result_add = x + y
print("덧셈 결과:")
print(result_add)

# 요소별 뺄셈
result_sub = x - y
print("\n뺄셈 결과:")
print(result_sub)
```

```
덧셈 결과:
[[3 5]
 [7 9]]
```

```
뺄셈 결과:
[[-1 -1]
 [-1 -1]]
```

이미지 작업 - 색조 변경

```
# We can tint the image by scaling each of the color channels  
# by a different scalar constant. The image has shape (400, 248, 3);  
# we multiply it by the array [1, 0.95, 0.9] of shape (3,);  
# numpy broadcasting means that this leaves the red channel unchanged,  
# and multiplies the green and blue channels by 0.95 and 0.9  
# respectively.  
img_tinted = img * [1, 0.95, 0.9]
```

이미지 작업 - 크기 조정

```
# Resize the tinted image to be 300 by 300 pixels.
```

```
img_tinted = imresize(img_tinted, (300, 300))
```

```
# Write the tinted image back to disk
```

```
imsave('assets/cat_tinted.jpg', img_tinted)
```



Matlab 파일

- 함수 `scipy.io.loadmat`와 `scipy.io.savemat`는 Matlab 파일을 읽고 쓸 수 있게 함

점 간의 거리

- SciPy는 점 집합 간의 거리 계산을 위한 유용한 함수들을 제공
- 함수 `scipy.spatial.distance.pdist`는 주어진 집합에서 모든 점 쌍 간의 거리를 계산

```
import numpy as np
from scipy.spatial.distance import pdist, squareform

# Create the following array where each row is a point in 2D space:
# [[0 1]
#  [1 0]
#  [2 0]]
x = np.array([[0, 1], [1, 0], [2, 0]])
print(x)

# Compute the Euclidean distance between all rows of x.
# d[i, j] is the Euclidean distance between x[i, :] and x[j, :],
# and d is the following array:
# [[ 0.          1.41421356  2.23606798]
#  [ 1.41421356  0.          1.          ]
#  [ 2.23606798  1.          0.          ]]
d = squareform(pdist(x, 'euclidean'))
print(d)
```



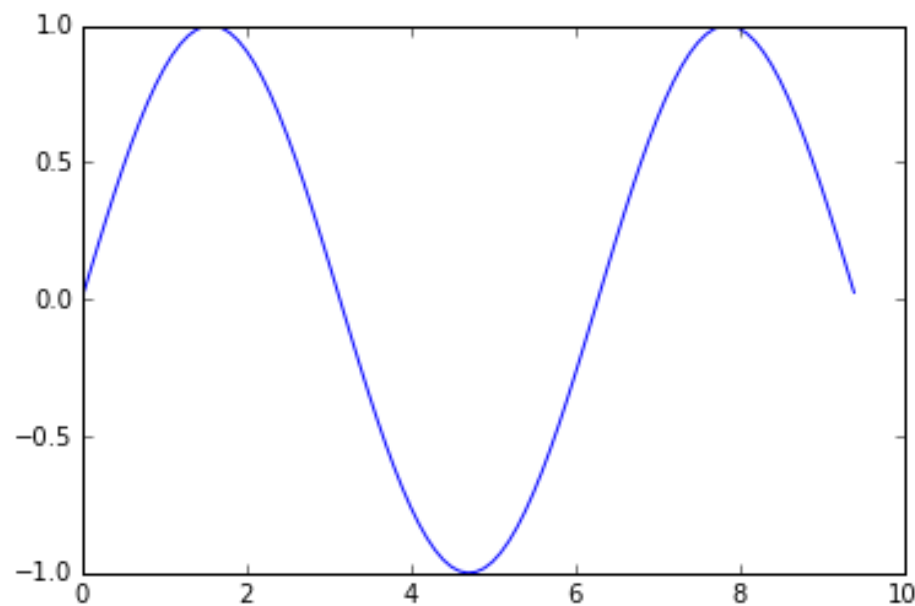
그래프 그리기

- Matplotlib에서 가장 중요한 함수는 2D 데이터를 그릴 수 있게 해 주는 plot 함수

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

# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show() # You must call plt.show() to make graphics appear.
```

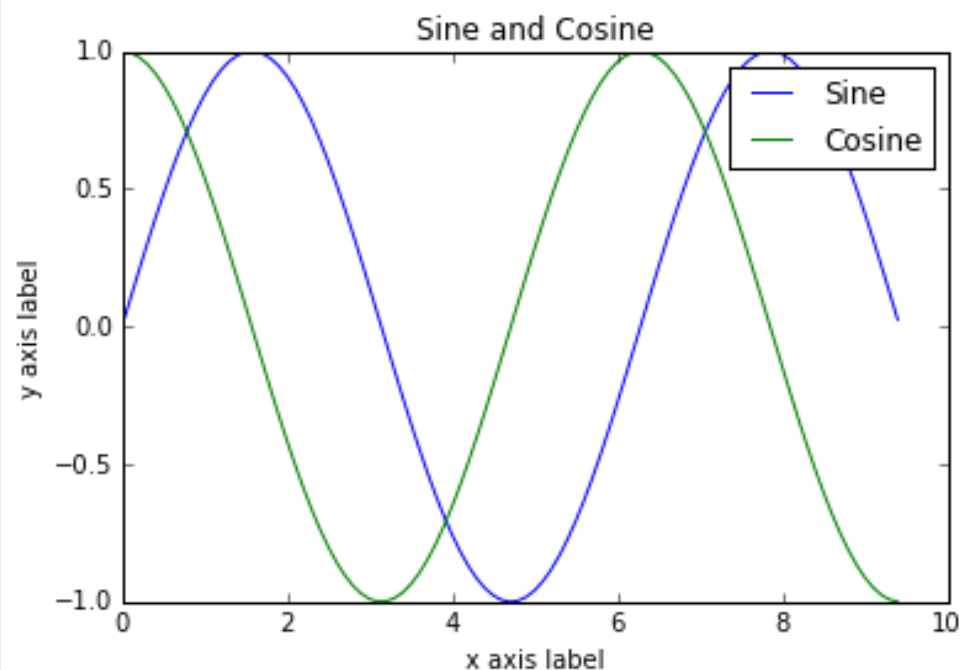


그래프 그리기

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

# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```



서브플롯(Subplot)

- subplot 함수를 통해 다양한 그래프를 한번에 그릴 수 있음

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

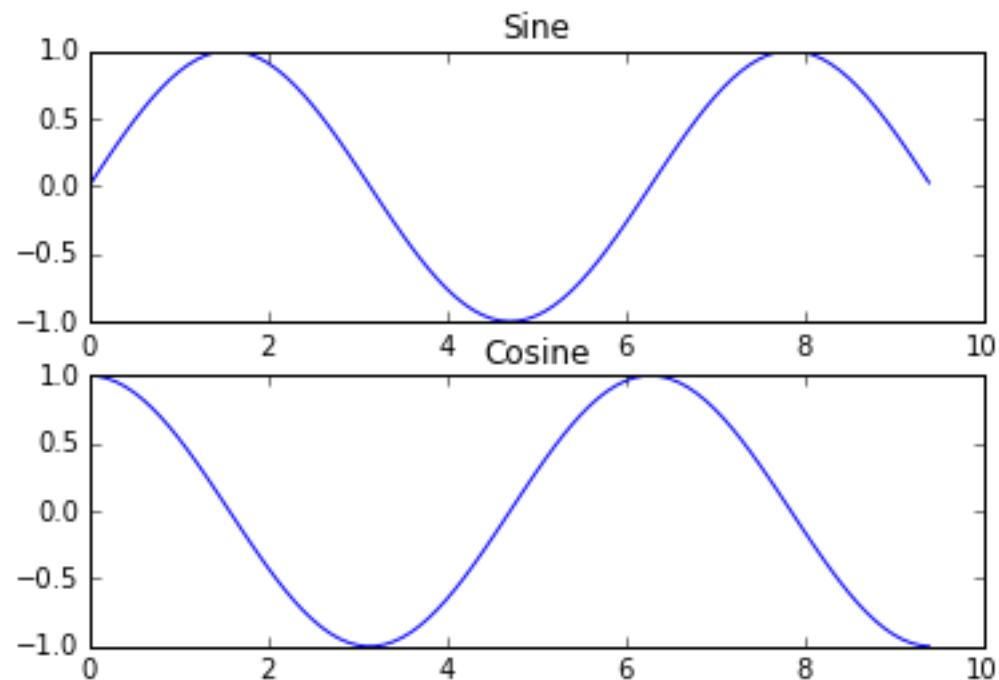
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```



이미지 보기

- 이미지를 보여주기 위해 imshow 함수 사용 가능

```
import numpy as np
from scipy.misc import imread, imresize
import matplotlib.pyplot as plt
```

```
img = imread('assets/cat.jpg')
img_tinted = img * [1, 0.95, 0.9]
```

Show the original image

```
plt.subplot(1, 2, 1)
plt.imshow(img)
```

Show the tinted image

```
plt.subplot(1, 2, 2)
```

*# A slight gotcha with imshow is that it might give strange results
if presented with data that is not uint8. To work around this, we
explicitly cast the image to uint8 before displaying it.*

```
plt.imshow(np.uint8(img_tinted))
plt.show()
```

