

파이썬을 이용한 데이터수집 및 스마트공장 견학

Python의 기초

2023년 1월 10일
안재관

금일 목표

- 제어문과 반복문을 활용하여 구구단 출력
- 함수를 활용하여 구구단 출력 기능을 재사용 및 보강

식별자(Identifier) Recap

● 식별자(Identifier)의 value와 type

- **Value**는 문자나 숫자와 같이 프로그램이 작동하는 기본적인 것 중 하나이다. 지금까지 우리가 본 몇몇 **Value**들은 2, 42.0, 그리고 'Hello World!'이다.
- 이 **Value**들은 다른 **Type**에 속한다: 2는 **Integer**(정수), 42.0은 **Float**(실수), 그리고 'Hello, World!'는 **String**(문자열)이다.

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<class 'str'>
```

파이썬에서의 Type

Text Type: **str**

Numeric Types: **int**, **float**, complex

Sequence Types: **list**, tuple, range

Mapping Type: **dict**

Set Types: set, frozenset

Boolean Type: **bool**

Binary Types: bytes, bytearray, memoryview

자료형 – 수치형(Numeric)

● 정수형(integer), 실수형(float), 복소수형(complex)

```

example = 5
print(type(example), example)

example = 5.
print(type(example), example)

example = 5.23
print(type(example), example)

example = 5.23e2
print(type(example), example)

example = 5.23e-2
print(type(example), example)

example = 5j
print(type(example), example)

```

▶

```

<class 'int'> 5
<class 'float'> 5.0
<class 'float'> 5.23
<class 'float'> 523.0
<class 'float'> 0.0523
<class 'complex'> 5j

```

● 산술 연산자: / 와 //와 % 용례 구분 주의

Name	Meaning	Example	Result
+	Addition	34 + 1	35
-	Subtraction	34.0 - 0.1	33.9
*	Multiplication	300 * 30	9000
/	Float Division	1 / 2	0.5
//	Integer Division	1 // 2	0
**	Exponentiation	4 ** 0.5	2.0
%	Remainder	20 % 3	2

▶

```

print(34 + 1)
print(34.0 - 0.1)
print(300 * 30)
print(1 / 2)
print(2 // 3)
print(9 ** 0.5)
print(20 % 3)

```

```

35
33.9
9000
0.5
0
3.0
2

```

자료형 – 수치형(Numeric)

● 데이터 타입 변환

- `int()`: 문자열, 실수형 데이터를 정수형으로 변환 – 기본적으로 실수형은 소수점 이하 무시
- `float()`: 문자열, 정수형 데이터를 실수형으로 변환
- `str()`: 정수형, 실수형 데이터를 문자열로 변환
- `round()`: 실수형 데이터에 대해 반올림 기능
 기본적으로 정수형으로 변환
 소수점 자리 지정 가능

<code>print(int(1.3))</code>	1
<code>print(int(-1.7))</code>	-1
<code>print(int("5"))</code>	5
<code>print(float(5))</code>	5.0
<code>print(float("1.3"))</code>	1.3
<code>print(str(3))</code>	3
<code>print(type(str(3)))</code>	<class 'str'>
<code>print(str(-1.3e3))</code>	-1300.0
<code>print(round(1.3))</code>	1
<code>print(round(-1.7))</code>	-2
<code>print(round(1.37, 1))</code>	1.4

자료형 – 문자열(String)

● Immutable type; 불변형 자료형

● Escape code

- 특정 문자를 문자열에 포함시키기 위해 사용하는 방법
- 따옴표, 줄바꿈, 탭 등의 문자를 위해서 많이 사용됨
- \n: 줄바꿈, \t: 탭, \' : 작은 따옴표, \": 큰 따옴표, \\: 백슬래시

● Formatting

- 문자열 내에 특정 값을 대입할 수 있게 하는 기능
- 다른 문자열이나 숫자를 문자열로 변환 후, concatenation 하는 것과 같은 기능
- 포맷 코드: 문자열 내에 대입하고자 하는 부분을 표기, 변수형에 따라 바뀜.
- 혹은 {} 와 format을 이용해서 역시 formatting 가능

코드	설명
%s	문자열 (String)
%c	문자 1개(character)
%d	정수 (Integer)
%f	부동소수 (floating-point)
%o	8진수
%x	16진수
%%	Literal % (문자 % 자체)

```

number = 3
day = "five"

sentence = "I ate %d apples. So I was sick for %s days." % (number, day)
print(sentence)

sentence = "I ate {} apples. So I was sick for {} days".format(number, day)
print(sentence)

sentence = "I ate %.1f apples. So I was sick for %s days" % (number, day)
print(sentence)

sentence = "\nI ate %d apples.\nSo I was sick for %s days." % (number, day)
print(sentence)

I ate 3 apples. So I was sick for five days.
I ate 3 apples. So I was sick for five days
I ate 3.0 apples. So I was sick for five days

I ate 3 apples.
So I was sick for five days.

```

자료형 – 문자열(String)

● 문자열 연산

- Concatenation : + 연산자를 통해 두 문자열을 이어 붙일 수 있음
- 반복 연산 : * 연산자를 통해 문자열을 반복시킬 수 있음

```
dma = "Data Management"

dma += " and Analysis"
dma

'Data Management and Analysis'

dma*6

'Data Management and AnalysisData Management and AnalysisData Management and AnalysisData Management and AnalysisData Management and AnalysisData Management and Analysis'
```

● 인덱싱 및 슬라이싱

- 문자열의 개별 문자에 접근(Indexing) 혹은 특정 구간 추출(Slicing)
- 정방향 및 역방향 인덱스 모두 활용 가능
- 인덱싱은 되지만 Immutable type이므로 특정 index 통해 item assignment 불가

```
dma[0:3]

'Dat'

dma[: -3]

'Data Management and Analy'
```

```
dma[3] = 'd'
dma

-----
TypeError                                Traceback (most recent call last)
<ipython-input-13-4859cf3dd44c> in <module>
----> 1 dma[3] = 'd'
      2 dma

TypeError: 'str' object does not support item assignment
```

자료형 – 문자열(String)

● 유용한 문자열 내장 함수

- count() : 괄호 내의 문자열이 몇 번 나오는지 반환
- find(): 괄호 내의 문자열이 어느 위치에 있는지 처음 위치 반환
cf> 등장하는 모든 위치를 찾고 싶다면 별도의 함수 이용 (re.finditer)
- upper(), lower(): 대소문자 변환
- strip(): 좌우의 공백 제거
- replace(): 특정 문자열을 다른 문자열로 대체
- split(): 괄호 내의 문자열을 기준으로 쪼개서 리스트로 반환, 입력 값이 없을 경우 공백을 기준으로 나눔
- join(): 괄호 내의 문자열 사이에 해당 문자열을 삽입

```
dma.count("a")
```

```
6
```

```
dma.find("a")
```

```
1
```

```
dma.find_all("a")
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-16-569764dfeca6> in <module>
----> 1 dma.find_all("a")
```

```
AttributeError: 'str' object has no attribute 'find_all'
```

```
# 참고용
import re
```

```
[m.start() for m in re.finditer('a', dma)]
```

```
[1, 3, 6, 8, 16, 22]
```

```
dma.upper()
```

```
'DATA MANAGEMENT AND ANALYSIS'
```

```
dma.lower()
```

```
'data management and analysis'
```

```
dma.replace("Data", "Life")
```

```
'Life Management and Analysis'
```

```
dma.split()
```

```
['Data', 'Management', 'and', 'Analysis']
```

```
dma.split("a")
```

```
['D', 't', ' M', 'n', 'gement ', 'nd An', 'lysis']
```

```
dma.join(",")
```

```
','
```

```
(",").join(dma)
```

```
'D,a,t,a, ,M,a,n,a,g,e,m,e,n,t, ,a,n,d, ,A,n,a,l,y,s,i,s'
```


자료형 – 리스트(List)

● []으로 둘러 쌓인 데이터의 sequence

- 하나의 자료형에 대한 리스트를 선언하는 Java와는 달리, 여러 자료형이 섞일 수 있음
- 리스트 내에 리스트가 들어갈 수 있음
- 아무것도 기입하지 않고 [] 만으로 빈 리스트 생성 가능

```
list1 = []  
list2 = [1, 2, "list"]  
list3 = [[1,2], "list"]  
  
print(list2)  
print(list3)  
  
[1, 2, 'list']  
[[1, 2], 'list']
```

● 리스트 연산

- 문자열과 같은 방법으로 concatenation(+)과 반복(*) 연산 수행 가능
- 문자열과 같은 방식으로 indexing과 slicing 가능
- Immutable type이 아니기에 index를 통해 element 수정 가능

```
a = [1,2,3]  
b = [4,5,6]  
  
print(a+b)  
  
[1, 2, 3, 4, 5, 6]
```

```
a*2  
  
[1, 2, 3, [4, 5, 6], 7, 8, 1, 2, 3, [4, 5, 6], 7, 8]
```

```
a[0:5]  
  
[1, 2, 3, [4, 5, 6], 7]
```

자료형 – 리스트(List)

● 리스트 관련 함수

- len(): 리스트의 길이 반환
- sum(): 정수형, 실수형으로 구성된 리스트의 합계
- max(), min(): 최댓값, 최솟값(문자열이 포함된 리스트도 가능)

```
len(a)
```

```
7
```

```
c = [1, 2, 3, 4]
```

```
sum(c)
```

```
10
```

```
min(c)
```

```
1
```

```
max(c)
```

```
4
```

```
d = ["a", "b", "c", "A"]
```

```
sum(d)
```

```
TypeError                                Traceback (most recent call last)
<ipython-input-40-efc3ec6806b8> in <module>
      1 d = ["a", "b", "c", "A"]
      2
----> 3 sum(d)
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
min(d)
```

```
'A'
```

```
max(d)
```

```
'c'
```

자료형 – 리스트(List)

● 리스트 관련 함수

- `append(item)`: 해당 아이템을 리스트의 맨 뒤에 추가
- `insert(index, item)`: 리스트의 지정된 인덱스에 아이템 삽입
- `extend(list)`: 해당 리스트를 리스트의 뒤에 이어 붙임
- `pop(index)`: 지정된 인덱스의 아이템을 반환하고 리스트에서 제거
- `index(item)`: 해당 아이템이 처음으로 등장한 인덱스 반환

```
a = [1,2,3]
```

```
a.append([4,5,6])  
a
```

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

```
a.insert(0, 0)  
a
```

```
[0, 1, 2, 3, [4, 5, 6]]
```

```
a.extend([4,5,6])  
a
```

```
[0, 1, 2, 3, [4, 5, 6], 4, 5, 6]
```

```
a.pop(0)  
a
```

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

```
a.index([4,5,6])
```

```
3
```

자료형 – 리스트(List)

● 리스트 관련 함수

- `remove(item)`: 처음으로 등장한 지정된 아이템을 삭제
- `count(item)`: 해당 아이템의 등장 횟수 반환
- `sort()`: 오름차순 정렬
- `reverse()`: 리스트의 순서를 반대로 바꿈

```
a.remove([4,5,6])  
a
```

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

```
a.count(2)
```

```
1
```

```
a.sort()  
a
```

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

```
a.reverse()  
a
```

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

반복문

● 시퀀스 데이터 타입

- 여러 개의 자료들의 배열을 나타내고 순서가 정해진 타입
- 종류
 - 문자열 : 문자들의 시퀀스
 - 리스트 : 임의의 자료형을 갖는 데이터들의 시퀀스
- 내장 함수 예시
 - len() : 시퀀스의 길이를 알려줌
 - 리스트.index(item) : 리스트 내에 item이 등장하는 첫 인덱스를 알려줌
 - 리스트.count(item) : 리스트 내에 item이 몇 회 등장하는지 알려줌

```
list = [2, 3, "abcd", "defg", 2, 2]
print(list[0])           2
print(list[2])           abcd
print(len(list))         6
print(list.index(3))     1
print(list.count(2))     3

str = "example"
print(str[0])            e
print(str[2])            a
print(len(str))          7
print(str.index("e"))    0
print(str.count("e"))    2
```

자료형 – 튜플(Tuple) & 집합(set)

● 튜플

- ()로 둘러 쌓인 Immutable type의 sequence 자료형
- 리스트와 거의 유사하며 튜플 역시 인덱싱과 슬라이싱 가능

```
list_a = [1,2,3]
list_a[0] = 3
list_a
```

```
[3, 2, 3]
```

```
tuple_a = (1,2,3)
tuple_a[0]
```

```
1
```

```
tuple_a[0] = 3
```

```
TypeError                                Traceback (most recent call last)
<ipython-input-50-aa9c85090299> in <module>
----> 1 tuple_a[0] = 3
```

```
TypeError: 'tuple' object does not support item assignment
```

● 집합

- 순서가 없음, 중복 허용 하지 않음
- set(sequence) 통해 시퀀스를 집합으로 변환 가능
- list(set) 통해 집합을 리스트로 변환 가능
- 집합 관련 함수
 - add(item): 원소 추가
 - update(sequence): 시퀀스 내 값들 추가
 - remove(item): 원소 제거
- 기타 집합 연산 모두 가능
 - 교집합(&, intersection)
 - 합집합(|, union)
 - 차집합(-, difference)

```
set([1,2,3,3,4,4,4])
{1, 2, 3, 4}
```

```
set1 = set([1,2,3,4])
set2 = set([3,4,5,6])
set1 & set2
{3, 4}
```

```
set1.intersection(set2)
{3, 4}
```

```
set1 | set2
{1, 2, 3, 4, 5, 6}
```

```
set1.union(set2)
{1, 2, 3, 4, 5, 6}
```

```
set1 - set2
{1, 2}
```

```
set1.difference(set2)
{1, 2}
```

```
set1.add(5)
set1
{1, 2, 3, 4, 5}
```

```
set1.update([5,6,7])
set1
{1, 2, 3, 4, 5, 6, 7}
```

```
set1.remove(4)
set1
{1, 2, 3, 5, 6, 7}
```

```
list(set1)
[1, 2, 3, 5, 6, 7]
```

자료형 – 딕셔너리(Dictionary)

● Key – value의 쌍을 저장하는 자료 구조

- 순서가 아닌 key를 통해 접근
- Key의 중복은 허용되지 않음
Key를 중복하여 삽입할 경우 마지막에 삽입된 value로 덮어 씌움
- {key1: value1, key2: value2, ...} 또는 dict()로 생성
- Value에는 리스트와 같은 자료형이 들어갈 수 있음 (key는 불가)
- Key를 통해 value 찾기 가능

● 관련 함수들

- keys() : 딕셔너리의 모든 key들을 반환
- values() : 딕셔너리의 모든 value들을 반환
- items() : 딕셔너리의 모든 key-value 쌍을 반환(tuple 형태로 반환)
- clear() : 딕셔너리를 비움
- in : 해당 key가 딕셔너리에 있는지 조사

```
dic = {'a': 1, 'b': 2, 3: 4}
```

```
dic
```

```
{'a': 1, 'b': 2, 3: 4}
```

```
dic['a'] = [1,2,3]
```

```
dic
```

```
{'a': [1, 2, 3], 'b': 2, 3: 4}
```

```
del dic['a']
```

```
dic
```

```
{'b': 2, 3: 4}
```

```
dic['c'] = 3
```

```
dic
```

```
{'b': 2, 3: 4, 'c': 3}
```

```
'a' in dic
```

```
False
```

```
'b' in dic
```

```
True
```

```
dic.keys()
```

```
dict_keys(['b', 3, 'c'])
```

```
dic.values()
```

```
dict_values([2, 4, 3])
```

```
dic.items()
```

```
dict_items([('b', 2), (3, 4), ('c', 3)])
```

자료형 – 불리언(Boolean)

● True, False의 값 만을 가지는 이진 자료형

```
2>1
```

```
True
```

● 불 자료형으로의 변환

- bool() 함수를 써서 다른 자료형을 불 자료형으로 변환
- 정수형, 실수형 : 0일 경우 False, 나머지는 True
- 문자열 : 빈 문자열의 경우 False, 나머지는 True
- 리스트, 집합, 딕셔너리, 튜플 : 비어있을 경우 False, 나머지는 True

```
2<1
```

```
False
```

```
bool(1)
```

```
True
```

```
bool(0)
```

```
False
```

```
bool('False')
```

```
True
```

```
bool('')
```

```
False
```

```
bool([])
```

```
False
```

```
bool([False])
```

```
True
```

● 불 연산

- and, or, not: 기본적인 논리 연산

```
True and False
```

```
False
```

```
True or False
```

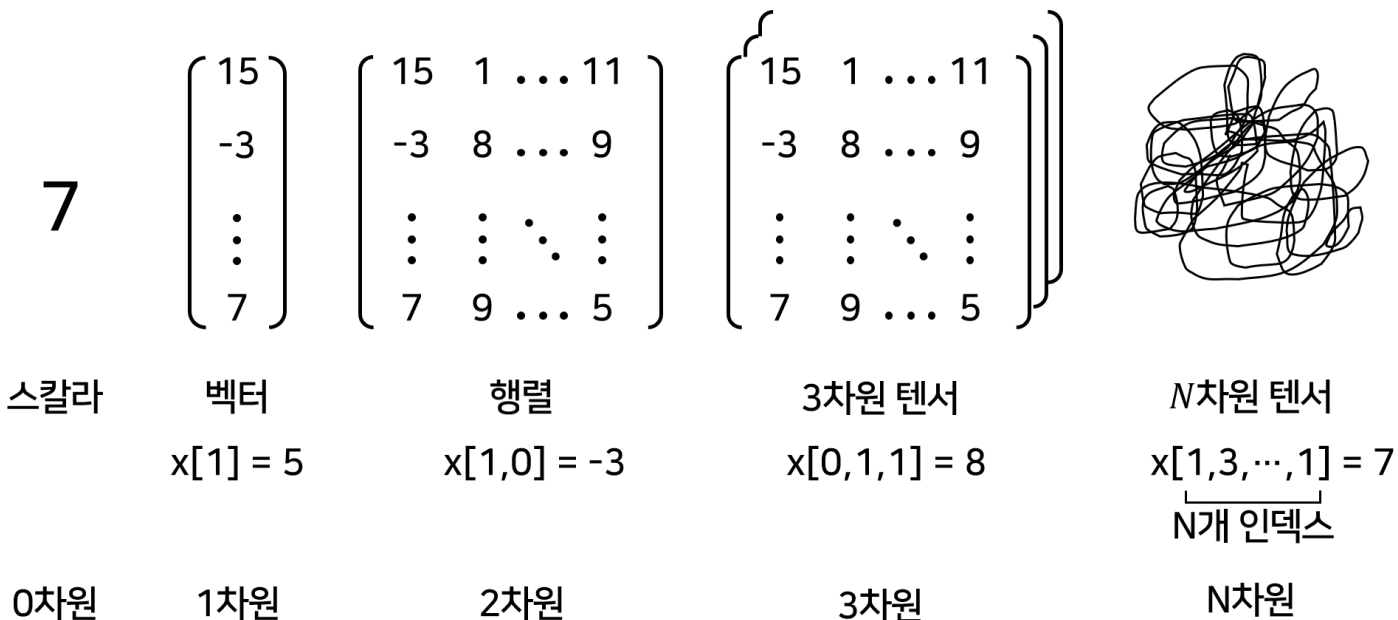
```
True
```

```
not True
```

```
False
```





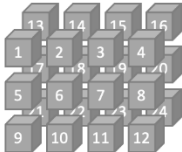

자료형 – 스칼라, 벡터, 행렬, 텐서

- 스칼라는 0차원, 벡터는 1차원, 행렬은 2차원의 Array형 자료
- 텐서란 N 차원의 배열(Array)



자료형 – 스칼라, 벡터, 행렬, 텐서

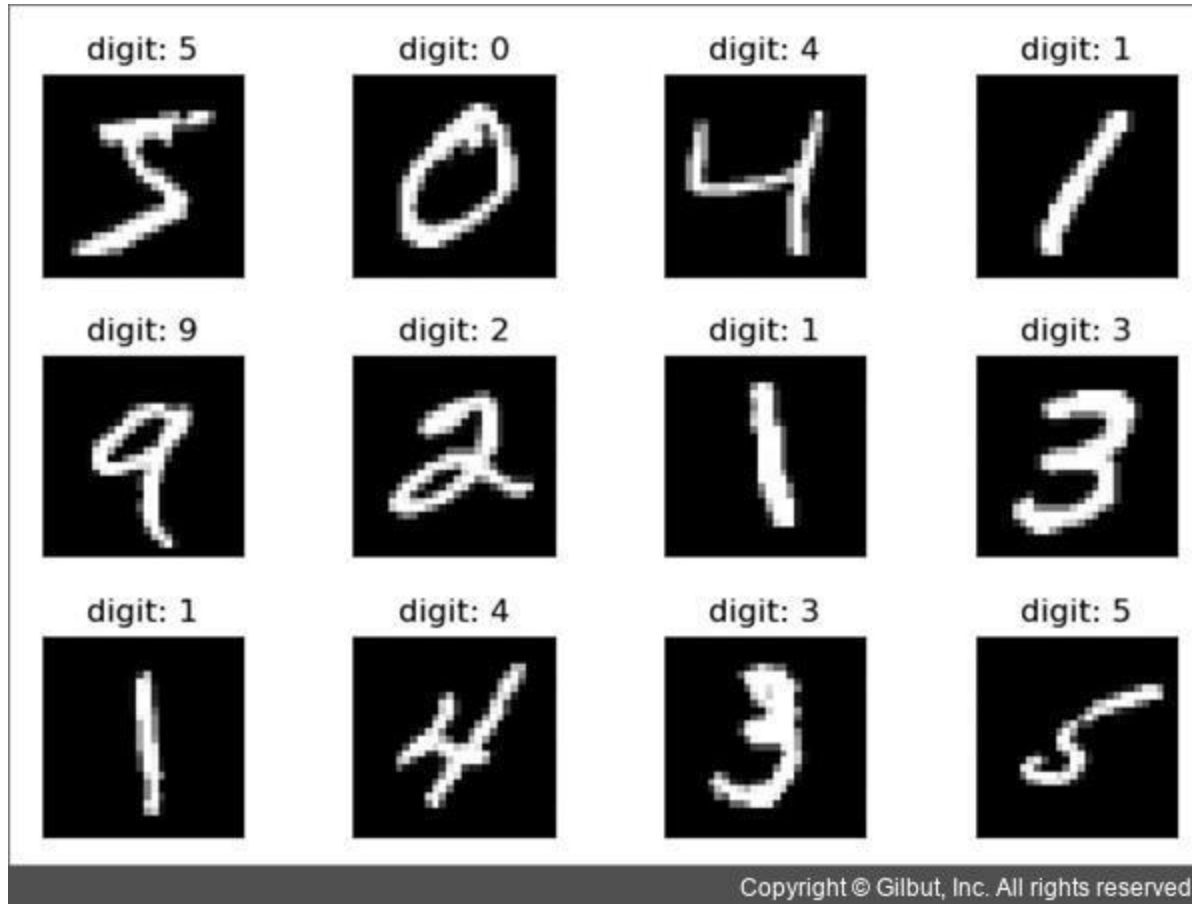
- 스칼라는 0차원, 벡터는 1차원, 행렬은 2차원의 Array형 자료
- 텐서란 N 차원의 배열(Array)

 텐서플로 자료 구조 (tf.Tensor data types)				
	스칼라 (Scalar)	벡터 (Vector)	행렬 (Matrix)	텐서 (Tensor)
rank	0	1	2	3
shape	()	(4,)	(3, 4)	(2, 3, 4)
				
	<pre><tf.Tensor: shape=(), dtype=int32, numpy=1></pre>	<pre><tf.Tensor: shape=(4,), dtype=int32, numpy=array([1, 2, 3, 4], dtype=int32)></pre>	<pre><tf.Tensor: shape=(3, 4), dtype=int32, numpy= array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]], dtype=int32)></pre>	<pre><tf.Tensor: shape=(2, 3, 4), dtype=int32, numpy= array([[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]], [[13, 14, 15, 16], [17, 18, 19, 20], [21, 22, 23, 24]]], dtype=int32)></pre>

[R, Python 분석과 프로그래밍의 친구] <https://rfriend.tistory.com>

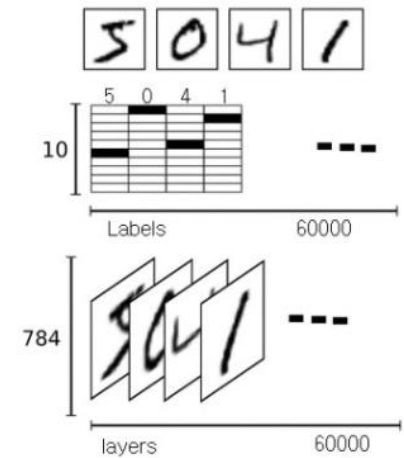
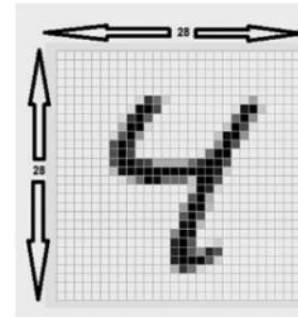
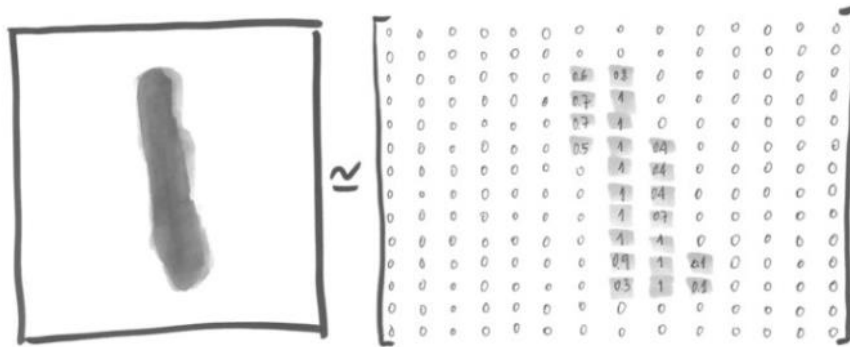
2017년 이후는 빅데이터 시대

- 비정형데이터에 대한 처리: 대표적인 이미지 데이터 MNIST



2017년 이후는 빅데이터 시대

● 비정형데이터에 대한 처리: 대표적인 이미지 데이터 MNIST



제어문

● if문

- 주어진 조건을 만족했을 때 특정 명령을 수행함
- 사용법

if 조건:
 명령



- 조건문 이후에는 콜론(:)이 있어야 함
- 조건문에 해당하는 수행 명령어는 들여쓰기로 구분되어 있어야 함
- 최소 한 줄 이상의 수행 명령어가 있어야 함
아무런 행동을 하지 않을 때에는 pass 명령어

- 예시

```
a = 20
if a > 15:
    print("a는 15보다 큼니다.")
print("=====")
```

```
a는 15보다 큼니다.
=====
```

```
a = 10
if a > 15:
    print("a는 15보다 큼니다.")
print("=====")
```

```
=====
```

- 명령 앞에 들여쓰기 필수

제어문

● else문

- if문이 거짓일 경우에만 특정 명령을 수행함
- 사용법

```
if 조건:
    명령
else:
    명령
```

- 예시

```
a = 10
if a > 15:
    print("a는 15보다 큽니다.")
else:
    print("a는 15보다 작습니다.")
print("=====")
```

a는 15보다 작습니다.

=====

- 명령 앞에 들여쓰기 필수
- else 단독으로 쓸 수 없고 if와 함께 쓰여야 함
- 짝을 이루는 if와 else는 동일한 수준의 들여쓰기 필수
- else에는 if와 달리 조건이 붙지 않음

제어문

● elif문

- if문이 거짓이고 직전의 elif까지 전부 거짓일 경우에만 특정 명령을 수행함
- 사용법

```

if 조건:
    명령
elif 조건:
    명령
elif 조건:
    명령
else:
    명령

```

- 예시

```

a = 18
if a > 20:
    print("a는 20보다 큼니다.")
elif a > 18:
    print("a는 20이하 18초과입니다.")
elif a > 16:
    print("a는 18이하 16초과입니다.")
else:
    print("a는 15보다 작습니다.")
print("=====")

```

```

a는 18이하 16초과입니다.
=====

```

- 명령 앞에 들여쓰기 필수
- elif 단독으로 쓸 수 없고 if/else와 함께 쓰여야 함
- 짝을 이루는 if, elif, else는 동일한 수준의 들여쓰기 필수
- 하나만 써야하는 if와 else와 달리 다수의 elif 사용 가능

제어문

● 주의사항

- 제어문 내에서 정의된 변수는 제어문 내에서만 활용하는 것이 좋음
조건문이 거짓일 경우 변수가 선언되지 않음
이후 제어문 밖에서 해당 변수 사용시, 선언되어 있지 않기에 오류
- 제어문 사용 시, 들여쓰기 주의
- 제어문의 조건문에도 불 연산이 포함될 수 있음

```
condition = 0
```

```
if condition > 0:
    print("positive")
elif condition < 0:
    print("negative")
else:
    print("zero")
```

```
zero
```

```
condition1 = True
condition2 = False
```

```
if condition1 and condition2:
    print("Both are True")
else:
    print("No way")
```

```
No way
```

```
condition = 0
```

```
if condition > 0:
    print("positive")
elif condition < 0:
    print("negative")
else:
    print("zero")
```

```
File "<ipython-input-90-dfa65cd3e81b>", line 5
elif condition < 0:
    ^
```

```
SyntaxError: invalid syntax
```

```
condition = 0
```

```
if condition >= 0:
    print("Non-negative")
elif condition < 0:
    print("negative")
```

```
Non-negative
```


반복문

● for문

- 주어진 시퀀스의 요소 하나하나를 사용하여 특정 명령을 수행함
- 사용법

```
for 요소 in 시퀀스:  
    명령
```

- 예시

```
for c in "sleepy":  
    print(c)  
list = [1, 2, 3, 4]  
for i in list:  
    print(i + 2)
```

- 명령 앞에 들여쓰기 필수

반복문

● range()

- 정수들의 시퀀스를 제공하는 함수
- 사용법 : range(시작인덱스, 끝인덱스, 건너뛰는크기)
 - 시작인덱스 : 생략가능, 생략시 0
 - 끝인덱스 : 필수
 - 건너뛰는크기 : 생략가능, 생략시 1
 - 예시
 - 다음 세 표현 모두 [0, 1, 2]를 나타냄
 - range(3)
 - range(0, 3)
 - range(0, 3, 1)
- 예시

```
for i in range(2, 5):  
    print(i)
```

2
3
4

```
for i in range(0, 10, 2):  
    print(i)
```

0
2
4
6
8

반복문의 흐름 변경하기

● break

- 반복문 내에서 break 실행시 해당 반복문이 전체가 종료됨
- 예시

```
for i in range(2, 100):  
    if(i > 9):  
        break  
    print("구구단", i, "단입니다.")  
print("구구단 출력을 마칩니다.")
```

구구단 2 단입니다.
구구단 3 단입니다.
구구단 4 단입니다.
구구단 5 단입니다.
구구단 6 단입니다.
구구단 7 단입니다.
구구단 8 단입니다.
구구단 9 단입니다.
구구단 출력을 마칩니다.

● continue

- 반복문 내에서 continue 실행시 현재 요소의 반복문 실행을 중지하고 다음 요소로 넘어가서 반복문 계속 실행
- 예시

```
for i in range(2, 10):  
    if(i == 5):  
        print("5단은 모르겠습니다.")  
        continue  
    print("구구단", i, "단입니다.")
```

구구단 2 단입니다.
구구단 3 단입니다.
구구단 4 단입니다.
5단은 모르겠습니다.
구구단 6 단입니다.
구구단 7 단입니다.
구구단 8 단입니다.
구구단 9 단입니다.

반복문 활용하기

● 구구단에 for문 활용하기

- 하나의 단 내에서의 반복 줄이기

```
dan = 2
print("구구단", dan, "단을 출력합니다.")
print(dan, "x 2 =", dan * 2)
print(dan, "x 3 =", dan * 3)
print(dan, "x 4 =", dan * 4)
print(dan, "x 5 =", dan * 5)
print(dan, "x 6 =", dan * 6)
print(dan, "x 7 =", dan * 7)
print(dan, "x 8 =", dan * 8)
print(dan, "x 9 =", dan * 9)
```



```
dan = 2
print("구구단", dan, "단을 출력합니다.")
for i in range(2, 10):
    print(dan, "x", i, "=", dan * i)
```

```
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
```

반복문 활용하기

● 구구단에 for문 활용하기

- 여러 단 내에서의 반복 줄이기

```
dan = 2
print("구구단", dan, "단을 출력합니다.")
for i in range(2, 10):
    print(dan, "x", i, "=", dan * i)
```

```
dan = 3
print("구구단", dan, "단을 출력합니다.")
for i in range(2, 10):
    print(dan, "x", i, "=", dan * i)
```

• • • •

```
dan = 9
print("구구단", dan, "단을 출력합니다.")
for i in range(2, 10):
    print(dan, "x", i, "=", dan * i)
```



```
for dan in range(2, 10):
    print("구구단", dan, "단을 출력합니다.")
    for i in range(2, 10):
        print(dan, "x", i, "=", dan * i)
```

구구단 2 단을 출력합니다.

```
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
```

구구단 3 단을 출력합니다.

```
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
```

• • • •

구구단 9 단을 출력합니다.

```
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
9 x 8 = 72
9 x 9 = 81
```

반복문 활용하기

● DIY 별 만들기

- 힌트 : `print("'" * 숫자)` 를 사용하면 별이 숫자만큼 찍힘

- 1.

```
*
**
***
****
*****
```

- 2.

```
*
***
*****
*****
*****
```

- 3.

```
*****
****
***
**
*
```

- 4.

```
      *
     ***
    *****
   *****
  *****
 *****
```

- 5.

```
12345
 1234
  123
   12
    1
```

- 6.

```
123454321
 1234321
  12321
   121
    1
```

함수

● 함수

- 자주 사용될 기능을 함수로 선언하면 나중에도 재사용 가능
- 함수 정의

```
def 함수명(인자1, 인자2, ...):  
    명령
```

- 함수 사용

```
함수명(인자1, 인자2, ...)
```

- 예시

```
def printDan(dan):  
    print("구구단", dan, "단을 출력합니다.")  
    for i in range(2, 10):  
        print(dan, "x", i, "=", dan * i)  
  
printDan(15)
```

구구단 15 단을 출력합니다.

```
15 x 2 = 30  
15 x 3 = 45  
15 x 4 = 60  
15 x 5 = 75  
15 x 6 = 90  
15 x 7 = 105  
15 x 8 = 120  
15 x 9 = 135
```

● DIY 7

- printDan 함수는 원래 x 9 까지만 출력하지만, 어디까지 출력할지 정할 수 있도록 printDan 함수에 인자를 추가해보기 (인자가 총 2개)

함수

● 함수(Function)

- 여러 변수가 있을 경우, return 을 맨 마지막에 넣음으로써 원하는 값을 뱉게 만들 수 있음
- return 값을 tuple 형태를 통해 여러 개를 리턴 가능
- 입력 인수의 개수를 모를 때 *args로 정의 가능

```

1 def summation(num):
2     sum = 0
3     for i in range(num):
4         sum += i + 1
5     return sum
6
7 summation(5)

```

15

```

1 def sum_mul(num):
2     sum = 0
3     mul = 1
4     for i in range(num):
5         sum += i + 1
6         mul *= i + 1
7     return sum, mul
8
9 sum_mul(5)

```

(15, 120)

```

1 def sum_many(*args):
2     sum = 0
3     for i in args:
4         sum += i
5     return sum
6
7 print(sum_many(1,2,3))
8 print(sum_many(1,3,5,7,9))
9
10

```

6
25

```

1 def sum_many2(base = 0, *args):
2     for i in args:
3         base += i
4     return base
5
6 print(sum_many2(10,1,2,3))

```

16

함수

● Lambda 함수(Lambda Function)

- 간단한 함수를 한 줄로 표현하여 사용 가능

Lambda 인자: 표현식

Ex) `lambda(x, y: x+y)(1,2) → 3`

- Map, filter function과의 활용
 - Map 함수는 리스트의 요소를 지정된 함수로 처리해주는 함수
 - Filter 함수는 특정 조건으로 걸러서 걸러진 요소들로 처리해주는 함수

● Lambda 함수는 def으로 정의 된 함수와 함께 사용 가능

```
def text_preprocessing(text):
    ex2=text.lower() #대소문자
    ex3 = re.sub(r'[\.,\!\@\#\$\%\&\*\*\+\-=>\/\(\)\#\[\]]', '', ex2)
    ex4=WordPunctTokenizer().tokenize(ex3)
    return final_NN_words
```

```
tqdm.pandas()
output_updated=output.drop(["video_id", "view_count"], axis=1)
output_updated["preprocessed_title"]=output['video_title'].progress_apply(lambda x: text_preprocessing(x))
output_updated["preprocessed_description"]=output['description'].progress_apply(lambda x: text_preprocessing(x))
```

```
1 (lambda x,y: x+y)(1,2)
```

```
3
```

```
1 list(map(lambda x: x**2, range(5)))
```

```
[0, 1, 4, 9, 16]
```

```
1 a = [1,2, 2.5, 3.7, 4.6]
2 for i in range(len(a)):
3     a[i] = int(a[i])
4 a
```

```
[1, 2, 3, 4]
```

```
1 a = [1,2, 2.5, 3.7, 4.6]
2 a = list(map(int, a))
3 a
```

```
[1, 2, 3, 4]
```

```
1 list(filter(lambda x: x<3, range(5)))
```

```
[0, 1, 2]
```