Data Type

Data

- Data
  - ✓ Literal: 프로그램 상에서 개발자가 직접 입력하는 값
  - ✓ Variable(변수): 값을 저장할 수 있는 메모리 상의 공간에 붙여놓은 이름으로 값을 저장하고 있는 공간에 붙여놓은 이름
  - ✓ 변수의 선언(생성)

    변수명 = 데이터 또는 연산식 또는 함수호출
  - ✓ DataType(자료형)은 변수가 가리키는 공간에 저장할 수 있는 데이터의 종류
  - ✓ DataType(자료형)은 데이터를 저장하기 위한 방법과 크기
  - ✓ 변수명은 저장 공간을 구분하기 위한 이름으로 변수를 생성하면 프로그래밍 언어는 테이블을 만들어서 참조할 위치와 변수 이름을 매핑
  - ✓ 변수의 이름은 영역 내에서 구분하기 위한 이름이므로 동일한 영역에 동일한 이름의 변수를 2번 이상 만들면 데이터를 수정

Data

- 데이터 분류
  - ✓ 데이터 개수에 따른 분류
    - Scala Data : 1개의 데이터
    - Vector Data(Collection): 0개 이상의 데이터
  - ✓ 데이터 형태에 따른 분류
    - 정형 데이터(Structured Data): 관계형 데이터베이스 시스템의 테이블과 같이 고정된 컬럼에 저장되는 데이터와 파일, 그리고 지정된 행과 열에 의해 데이터의 속성이 구별되는 스프레드시트 형태의 데이터
    - 비정형 데이터(Unstructured-Data): 데이터 세트가 아닌 하나의 데이터가 수집 데이터로 객체화 된 형태로 텍스트 데이터나 이미지, 동영상 같은 멀티미디어 데이터가 대표적인 비정형 데이터
    - 반정형 데이터: 데이터 내부에 정형데이터의 스키마에 해당되는 메타데이터를 갖고 있는 경우가 많은데 스마트 기기에서 센서 데이터 등이 여기에 속함

Data Type

❖ Python 의 기본 자료형

bool	True 또는 False를 저장
int, float, complex	정수, 실수, 복소수
str	유니코드 문자열 – 변경할 수 없는 문자열
bytes	0-255 사이의 코드 모임
list	순서가 있는 데이터의 집합 : 내용 변경 가능
tuple	순서가 있는 데이터의 집합 : 내용 변경 불가능
set	순서가 없는 데이터의 집합
dict	키와 값을 쌍으로 만들어진 데이터의 집합

Scala Data - Direct

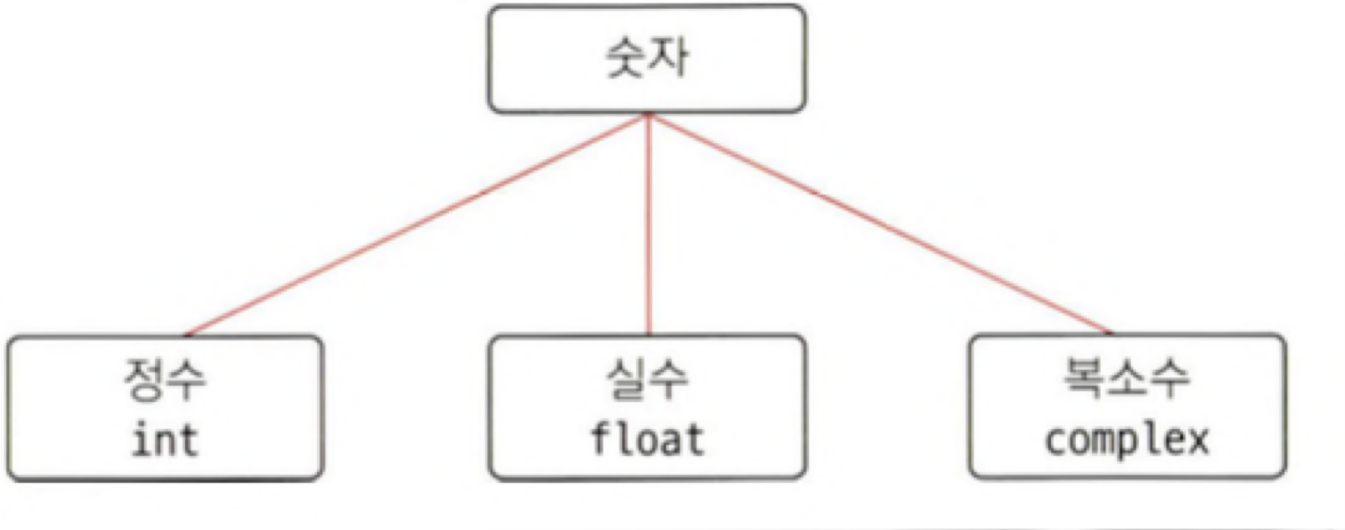
- bool
  - ✓ True 와 False 중 하나를 저장할 수 있는 자료형
  - ✓ 숫자 자료형의 0 이나 데이터가 존재하지 않는 컬렉션의 경우는 False로 간주
  - ✓ 숫자 자료형의 0 이 아닌 값이나 데이터가 존재하는 컬렉션의 경우는 True로 간주
  - ✓ bool 자료형끼리 산술 연산 가능 – False는 0으로 True는 1로 간주하고 산술 연산
  - ✓ 식이나 연산의 결과를 bool 타입으로 변경하고자 하는 경우에는 bool(값이나 연산식)

```
#부울 자료형 확인
print(bool([]))
print(bool(3))
print(bool(3+4))
```

False
True
True

Scala Data - Direct

- 수치 데이터
  - ✓ 정수는 영어로 integer이므로 줄여서 int 라고 하고 실수는 floating—point(부동 소수점)를 줄여서 float 라고 하며 복소수는 complex number를 줄여서 complex



Scala Data - Direct

- 수치 데이터
  - ✓ 정수형
    - 음의 정수, 0, 양의 정수
    - python에서는 메모리가 허용하는 한 무한대의 정수를 다룰 수 있음
    - 일반적인 정수는 10진 정수로 간주
    - 문자열을 정수로 변환할 때는 int(문자열) 또는 int(문자열, 진법)를 이용
    - int() 안에 문자열 대신에 실수를 대입하면 소수를 버림
    - 실수로 된 문자열이나 복소수는 정수로 변환이 안됨

Scala Data - Direct

- 수치 데이터
  - ✓ 정수형

```
#정수 자료형 저장 방법
a = 10
print("a =", a)
#정수 형변환
a = int('10')
print("a =", a)
#5진수로 변환
a = int('20', 5)
print("a =", a)
#실수를 정수로 변한 - 소수가 버려짐
a = int(10.8)
print("a =", a)
```

```
a = 10
a = 10
a = 10
a = 10
```

Scala Data - Direct

- ❖ 수치 데이터
  - ✓ 실수형
    - 파이썬에서는 실수를 지원하기 위해 부동 소수형을 제공
    - 부동 소수형은 소수점을 움직여서 소수를 표현하는 자료형
    - 부동 소수형은 8바이트를 이용해서 표현
    - 한정된 범위의 수만 표현
    - 디지털 방식으로 소수를 표현해야 하기 때문에 정밀도의 한계가 있음
    - 소수점을 표현하면 실수(1.2)
    - 지수(e)를 포함해도 실수(3e3, -0.2e-4)
    - 실수의 가장 큰 값과 작은 값은 sys 모듈의 float_info 또는 info.max, info.min으로 확인 가능
    - 무한대의 수는 float('inf' | '-inf') – 0으로 나누는 경우
    - is_integer 메서드를 통해서 정수로 변환시의 오차 문제 확인 가능
    - 실수를 정수로 변경할 때 사용할 때 사용할 수 있는 함수

      int, round

Scala Data - Direct

- 수치 데이터
  - ✓ 실수형

```
import sys
print("실수 정보:", sys.float_info)
a = 1.2
b = 3e3
c=-0.2e-4
print(a, b, c)
a = int(1.7)
print("a:", a)
a = round(1.7)
print("a:", a)
```

```
실수 정보: sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
    max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-
    307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
1.2 3000.0 -2e-05
a: 1
a: 2
a: 2
```

Scala Data - Direct

- ❖ 수치 데이터
  - ✓ 실수형
    - 실수의 저장 오류: 소수 중에는 정확하게 2진수로 표현할 수 없는 수가 있으므로 실수 연산은 오차가 발생할 수 있음

```
print('0.2 == (1.0-0.8):', (0.2 == (1.0-0.8)))
sum = 0
for i in range(0,1000):
    sum += 1
print("정수 1을 1000번 더한 결과:" , sum)

sum = 0.0
for i in range(0,1000):
    sum += 0.1
print("실수 0.1을 1000번 더한 결과:" , sum)
```

```
0.2 == (1.0-0.8): False
정수 1을 1000번 더한 결과: 1000
실수 0.1을 1000번 더한 결과: 99.9999999999986
```

Scala Data - Direct

- 수치 데이터
  - ✓ 복소수
    - 두 개의 실수를 가지고 복소수를 만들고자 할 때는 complex(실수부, 허수부)
    - 수학에서는 허수부를 i를 사용하지만 공학에서는 j를 사용
    - conjugate()는 켤레 복소수를 만들어서 리턴

```
print(1.2 + 1.3j)
print(complex(1.2, 1.3))

(1.2+1.3j)
(1.2+1.3j)
```

Scala Data - Direct

- ❖ 수치 데이터
  - ✓ Math 모듈
    - 수학에 관련된 함수를 구현한 모듈
    - import math 를 상단에 추가해서 math.함수명(매개변수...) 으로 사용 가능

```
import math

print(math.pow(3, 2))
print(math.sqrt(3))
```

```
9.0
1.7320508075688772
```

Scala Data - Direct

- ❖ 수치 데이터
  - ✓ Math 모듈
    - 표현 함수
      - ceil(x) 올림
      - floor(x) 내림
      - trunc(x) 절사
    - 삼각 함수 – 라디안 값으로 반환
      - cos(x) 코사인
      - sin(x) 사인
      - tan(x) 탄젠트
      - acos(x) 아크코사인
      - asin(x) 아크사인
      - atan(x) 아크탄젠트
      - atan2(x, y) x/y 아크탄젠트

Scala Data - Direct

- 수치 데이터
  - ✓ Math 모듈
    - 하이퍼볼릭 함수 – 라디안 값으로 반환
      - cosh(x) 하이퍼볼릭 코사인
      - sinh(x) 하이퍼볼릭 사인
      - tanh(x) 하이퍼볼릭 탄젠트
      - acosh(x) 하이퍼볼릭 아크코사인
      - asinh(x) 하이퍼볼릭 아크사인
      - atanh(x) 하이퍼볼릭 아크탄젠트
    - 각도 변환
      - degrees(x) 60분법으로 변환
      - radians(x) 호도법으로 변환

Scala Data - Direct

- ❖ 수치 데이터
  - ✓ Math 모듈
    - ● 논리 함수 – 라디안 값으로 반환
      - ◆ isclose(x, y, rel_tol=z) x와 y가 (z*1e+02)% 내외로 가까우면 True, 아니면 False
      - ◆ isinf(x) x가 inf이면 True, 아니면 False
      - ◆ isfinite(x) x가 inf, nan이면 False, 아니면 True
      - ◆ isnan(x) x가 nan이면 True, 아니면 False
        - ▪ isclose(x, y, rel_tol=z) 에서 rel_tol=z를 미입력시 기본값은 1e-09로 계산하는데 두 값의 차이가 5% 이내라면 z=0.05를 사용
    - ● 로그 함수
      - ◆ log(x, y) y를 밑으로 하는 x 로그
      - ◆ log10(x) 10을 밑으로 하는 x로그
      - ◆ log1p(x) e를 밑으로 하는 x+1로그
      - ◆ log2(x) 2를 밑으로 하는 x로그
        - ▪ log(x, y)에서 y를 미입력 시 밑을 e로 사용하여 자연 로그로 이용

Scala Data - Direct

- ❖ 수치 데이터
  - ✓ Math 모듈
    - ● 연산 함수
      - ◆ pow(x, y) — x의 y승
      - ◆ sqrt(x) — 루트 x
      - ◆ erf(x) — 오차함수
      - ◆ erfc(x) — 여오차함수
      - ◆ exp(x) — e의 x승
      - ◆ expm1 — e의 x-1승
      - ◆ frexp(x) — x를 (가수부, 지수부)로 반환
      - ◆ ldexp(x, y) — x*(2^y)
      - ◆ gamma(x) — 감마 함수
      - ◆ lgamma(x) — 감마 함수의 자연로그
      - ◆ factorial(x) — 팩토리얼
      - ◆ fsum([x, y, z, ...]) — 리스트의 합
      - ◆ fmod(x, y) — x를 y로 나눈 나머지
      - ◆ fabs(x) — 절대값
      - ◆ gcd(x, y) — x와 y의 최대 공약수

Scala Data - Direct

- ❖ 수치 데이터
  - ✓ Math 모듈
    - ● 연산 함수
      - ◆ hypot(x, y) 유클리드 놈을 반환
      - ◆ modf(x) x를 (소수부, 정수부)로 반환
      - ◆ copysign(x, y) y의 부호를 사용하는 x를 반환

Scala Data - Direct

- 수치 데이터
  - ✓ Math 모듈
    - 상수
      - e e
      - pi π
      - tau τ
      - inf ∞
      - nan Not a Number

Scala Data - Direct

- ❖ 수치 데이터
  - ✓ Math 모듈
    - ● 함수 수식

erf(x) : 오차함수

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

erfc(x) : 여오차함수

$$erfc(x) = 1 - erf(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$$

frexp(x) : (가수부, 지수부) 반환

$$frexp(x) = 가수부 \times 2^{지수부}$$

Scala Data - Direct

- 수치 데이터
  - ✓ Math 모듈
    - 함수 수식

gamma(x) : 감마함수

$$gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt$$

hypot(x, y) : 유클리드 노름

$$hypot(x, y) = \sqrt{x^2 + y^2}$$

Scala Data - Direct

- 수치 데이터
  - ✓ decimal.Decimal
    - float 자료형 보다 정확한 숫자 표현 모듈
    - 인스턴스 생성
      - Decimal(정수 | 실수 | 실수 문자열 | 튜플 | Decimal인스턴스)
    - Decimal 인스턴스는 Decimal 인스턴스나 정수 데이터와 연산은 가능하지만 실수 데이터 와의 연산은 허용되지 않음
    - 메서드
      - sqrt(): 제곱근 리턴
      - exp(): 지수 함수- e**x 의 값을 반환
      - ln(): Decimal의 자연 로그
      - compare(other): 크기 비교(-1, 0, 1 리턴)
      - copy_abs(): 절대값을 같는 인스턴스 리턴
      - copy_negate(): 원본의 음수 값을 갖는 인스턴스 리턴
      - copy_sign(other): 원본 값에 인자의 부호를 갖는 인스턴스 리턴

Scala Data - Direct

- 수치 데이터
  - ✓ decimal.Decimal

```
from decimal import Decimal
#부정확한 실수 연산
print((1234.567 + 45.67844) + 0.0004)
print(1234.567 + (45.67844 + 0.0004))
#정확한 실수 연산
print((Decimal('1234.567') + Decimal('45.67844')) + Decimal('0.0004'))
```

```
1280.2458399999998
1280.24584
1280.24584
```

Scala Data - Direct

- 수치 데이터
  - ✓ random 모듈
    - 파이썬에서 난수(random number)를 사용하기 위해서는 random 모듈을 사용
    - 메서드
      - seed(x): 시드를 설정하는 것으로 생략하거나 None을 대입하면 현재 시스템 시간을 이용
      - random(): 0 부터 1 사이의 부동소수점(float) 숫자를 리턴
      - randint(최소, 최대): 입력 파라미터인 최소부터 최대까지 중 임의의 정수를 리턴
      - uniform(최소, 최대): 입력 파라미터인 최소부터 최대까지 중 임의의 부동소수점(float) 숫자를 리턴
      - randrange(시작,끝[,간격]): 입력 파라미터인 시작부터 끝값까지 (지정된 간격으로 나열된) 숫자 중 임의의 정수를 리턴
      - gauss(m, sb): 가우스 분포의 난수를 리턴
      - shuffle(시퀀스 인스턴스): 시퀀스 인스턴스를 섞음
      - choice(시퀀스 인스턴스): 시퀀스 인스턴스의 임의의 아이템 리턴 – 복원 추출
      - sample(시퀀스 인스턴스, k): 시퀀스 인스턴스에서 k개 추출 – 비복원 추출

Scala Data - Direct

- 수치 데이터
  - ✓ random 모듈

```
import random

i = random.randint(1, 100)  # 1부터 100 사이의 임의의 정수
print(i)
f = random.random()   # 0부터 1 사이의 임의의 float
print(f)
f = random.uniform(1.0, 36.5)   # 1부터 36.5 사이의 임의의 float
print(f)
i = random.randrange(1, 101, 2) # 1부터 100 사이의 임의의 짝수
print(i)
i = random.randrange(10)  # 0부터 9 사이의 임의의 정수
print(i)
```

```
34
0.04875206312447189
16.621942404498753
77
1
```

Vector Data

- Sequential Type
  - ✓ 객체를 순서대로 저장하는 자료형
  - ✓ 파이썬에서 제공하는 다양한 시퀀스를 이해하면 코드를 새로 구현할 필요가 없으며 시퀀스의 공통 인터페이스를 따라 기존 혹은 향후에 구현될 시퀀스 자료형을 적절히 지원하고 활용할 수 있게 API를 정의할 수 있음
  - ✓ 컨테이너 시퀀스 와 균일 시퀀스
    - 컨테이너 시퀀스
      - 서로 다른 자료형의 항목들을 담을 수 있는 list, tuple, collections.deque 형
      - 객체에 대한 참조를 담고 있으며 객체는 어떠한 자료형도 될 수 있음
    - 균일 시퀀스
      - 하나의 자료형만 담을 수 있는 str, bytes, bytearray, memoryview. array.array 형
      - 객체에 대한 참조 대신 자신의 메모리 공간에 각 항목의 값을 직접 담기 때문에 균일 시퀀스가 메모리를 더 적게 사용하지만 문자, 바이트, 숫자 등 기본적인 자료형만 저장할 수 있음
  - ✓ 가변 시퀀스 와 불변 시퀀스
    - 가변 시퀀스: list, bytearray, array.array, collections.deque, memoryview 형
    - 불변 시퀀스: tuple, str, bytes 형
  - ✓ range를 이용해서 일정한 패턴을 갖는 숫자 Sequence 생성 가능

Vector Data

- Sequential Type
  - ✓ Sequential Type의 공통 연산
    - indexing(특정 번째에 해당하는 데이터를 가져오는 것): 변수명[인덱스] – 인덱스 번째 데이터를 리턴하는데 실제로는 __getitem__ 메서드를 호출하며 왼쪽의 첫번째 값은 0으로 오른쪽으로 갈 때 마다 1씩 증가하고 가장 마지막은 -1을 사용할 수 있으며 앞으로 갈때 마다 1씩 감소하는 형태로 지정 가능하고 인덱스가 범위를 벗어나면 index out of range 예외가 발생
    - 연결하기: + 연산자로 가능한데 새로운 sequence를 리턴
    - 반복하기: * 연산자를 이용해서 수행하는데 새로운 sequence를 리턴
    - 멤버 검사: in 연산자
    - 데이터 개수: len(SequentialType의 데이터)

Vector Data

- Sequential Type
  - ✓ Sequential Type의 공통 연산
    - slicing(특정 범위에 해당하는 데이터를 가져오는 것)
      - [start:end:interval] – start 번째부터 end 번째 바로 앞 까지 interval 간격으로 가져오는 기능을 수행
      - end가 생략되면 start 이후 전체를 가져오고 interval를 생략하면 순서대로 하나씩
      - 내부적으로는 __getitem__ 함수를 호출해서 수행
      - 데이터의 일부분을 대체하거나 삭제하고자 하는 경우에도 사용 가능

Vector Data

- Sequential Type
  - ✓ Sequential Type의 공통 연산

```
msg = "Korea"
print(msg[0]) #0번째 글자
print(msg[-2]) #뒤에서 2번째 글자
print(msg[1:3]) #1번째부터 3번째 글자 앞까지
print(msg[0:5:2]) #0부터 5번째 글자 앞까지 2개씩 건너뛰면서
print(msg[:-1])#-1 번째까지 가져오기
print(msg[::-1])#반대로 출력

msg = "Hello" + "World"
print(msg)

msg = msg * 4
print(msg)
```

```
K
e
or
Kra
Kore
aeroK
HelloWorld
HelloWorldHelloWorldHelloWorldHelloWorld
```

Vector Data

- str
  - ✓ 문자열 자료형 – 0개 이상 문자의 집합
  - ✓ 한 줄 문자열: ' ' 또는 " " 안에 기재하는데 단일 인용 부호 안에 이중 인용 부호를 이용해서 문자열을 대입할 수 있고 이중 인용 부호 안에 단일 인용 부호를 사용 가능
  - ✓ 여러 줄 문자열: ''' ''' 또는 """ """ 안에 기재
  - ✓ 문자열은 내부 데이터를 변경할 수 없음
  - ✓ ==, !=, >, >=, <, <= 연산 가능 – 코드 값을 가지고 앞에서부터 비교하는데 소문자가 대문자보다 코드 값이 32가 더 큼
  - ✓ 문자열은 작업을 할 때 복사해서 수행

```
msg = "hello"
msg[0] = 'f' #이런 문장은 에러
```

TypeError: 'str' object does not support item assignment

Vector Data

- str
  - ✓ 문자열을 추가하거나 삭제할 때는 슬라이싱과 연결하기를 이용

```
msg = "Hardware"
print("현재 msg의 id:", id(msg))

msg = msg + " and Shell"
print("현재 msg의 id:", id(msg))

msg = msg[:8] + ' Kernel' + msg[8:]
print(msg)
print("현재 msg의 id:", id(msg))

msg = msg[:15] + msg[20:]
print(msg)
```

```
현재 msg의 id: 140464219718896
현재 msg의 id: 140464219667472
Hardware Kernel and Shell
현재 msg의 id: 140464219664672
Hardware KernelShell
```

Vector Data

- str
  - ✓ ESCAPE code(제어 문자): 이스케이프 코드란 프로그래밍할 때 사용할 수 있도록 미리 정의해 둔 문자 조합 으로 ₩ 다음에 하나의 문자를 추가한 형태로 사용

코드	설명
₩n	개행 (줄바꿈)
₩t	수평 탭
₩₩	문자 "₩"
₩'	단일 인용부호(')
₩"	이중 인용부호(")
₩r	캐리지 리턴
₩f	폼 피드
₩a	벨 소리
₩b	백 스페이스
₩000	널문자

    - 이 중에서 활용 빈도가 높은 것은 ₩n, ₩t, ₩₩, ₩', ₩"
    - 문자열을 만들 때 r을 앞에 붙이면 raw 문자열로 생성할 수 있는데 이 때는 ESCAPE 문자가 무시됨

Vector Data

- str
  - ✓ ESCAPE code(제어 문자)

```
print('''인생은 생각할 수록 아름답고
역사는 앞으로 발전한다''')
print('₩t탭₩n다음 줄')
print(r'₩t탭₩n다음 줄')
```

```
인생은 생각할 수록 아름답고
역사는 앞으로 발전한다
    탭
다음 줄
₩t탭₩n다음 줄
```

Vector Data

- str
  - ✓ 문자열의 서식 지정: "%포맷형식"%(데이터)의 형태로 포맷을 구성할 수 있음
    - 문자열 안에 서식을 설정하고 이어서 %(데이터 나열)를 추가 설정
      - %s 문자열 (String)
      - %c 문자 1개(character)
      - %d 정수 (Integer)
      - %f 부동소수 (Floating-point)
      - %o 8진수
      - %x 16진수
      - %% Literal % (문자 % 자체)
    - %c 는 상수를 입력할 경우 ASCII Code 로 판단하여 출력
    - 서식은 % 다음에 정수를 추가해서 전체 자릿수를 확보해서 생성할 수 있는데 데이터의 길이보다 작은 숫자를 설정하면 데이터 전체를 출력
    - 실수의 경우는 전체 자릿수와 소수 자릿수를 .으로 구분해서 설정할 수 있는데 소수는 그 이하 자리에서 반올림한 결과를 출력하는데 전체 자릿수는 생략 가능
    - 모든 데이터는 %s로 서식 설정 가능

Vector Data

- str
  - ✓ 문자열의 서식 지정: 문자열의 format 메서드를 이용
    - 문자열 안에 {}를 사용하고 .format(데이터 나열)을 이용하면 나열된 데이터가 순차적으로 대입
    - {인덱스}를 이용하면 format에서 인덱스 번째 데이터가 대입
    - {인덱스:서식}을 이용하면 인덱스 번째의 데이터를 서식에 맞추어 생성

```
data = [1,3,4]
print('최대값:{0:5d}₩t최소값:{1:5d}'.format(max(data), min(data)))
```

최대값: 4 최소값: 1

    - 서식에 사용할 수 있는 보조 서식 문자
      - <: 왼쪽 맞춤
      - >: 오른쪽 맞춤
      - 공백: 양수 일 때 앞에 공백 출력
      - +: 양수 일 때 + 출력

Vector Data

- str
  - ✓ 문자열의 서식 지정: format 메서드 대신에 f.'{데이터:서식}' 도 가능한데 이를 f-string 이라고 함

Vector Data

- str
  - ✓ 문자열의 서식 지정

```
# 문자열 맨 앞에 f를 붙이고, 출력할 변수, 값을 중괄호 안에 넣습니다.
s = 'korean_vodka'
n = 5

result1 = f'저는 {s}를 좋아합니다. 하루 {n}잔 마셔요.'
print(result1)

s1 = 'left'
result1 = f'|{s1:<10}|'
print(result1)
# f-string 가운데 정렬
s2 = 'mid'
result2 = f'|{s2:^10}|'
print(result2)
# f-string 오른쪽 정렬
s3 = 'right'
result3 = f'|{s3:>10}|'
print(result3)

result = f'my  amount of alcohol {{{n}}}, {{n}}'
print(result)
```

Vector Data

- str
  - ✓ 문자열의 서식 지정

```
저는 korean_vodka를 좋아합니다. 하루 5잔 마셔요.
|left       |
|   mid     |
|      right|
my  amount of alcohol {5}, {n}
```

Vector Data

- str
  - ✓ 문자열의 대소문자 변환 메서드
    - upper(): 대문자로 변환
    - lower(): 소문자로 변환
    - swapcase(): 대소문자 스위치
    - capitalize(): 첫 글자만 대문자로 변환
    - title(): 각 단어의 첫 글자를 대문자로 변환

Vector Data

- str
  - ✓ 문자열의 검색 관련 메서드
    - count(문자열): 문자열의 횟수
    - find(문자열): 문자열이 있을 때 오프셋 반환
    - find(문자열, 숫자): 숫자 위치부터 문자열을 검색해서 오프셋을 반환하는데 찾는 문자열이 없을 때는 -1을 반환
    - rfind(문자열): 뒤에서부터 검색
    - index(문자열): 문자열의 위치를 찾아서 오프셋을 반환하는데 문자열이 없을 경우 예외 발생
    - rindex(문자열): 문자열을 뒤에서부터 검색하고 없으면 예외 발생
    - startswith(문자열): 문자열로 시작하는지 여부를 검사해서 bool로 리턴
    - endswith(문자열): 문자열로 끝나는지 여부를 검사
    - startswith(문자열, 숫자): 숫자에 해당하는 위치가 문자열로 시작하는지 여부를 검사해서 bool로 리턴
    - endswith(문자열, 숫자1, 숫자2): 숫자1부터 숫자2까지의 문자열이 문자열로 끝나는지 여부를 검사

Vector Data

- str
  - ✓ 문자열의 편집과 치환 메서드
    - strip(): 좌우 공백 제거
    - rstrip(): 오른쪽 공백 제거
    - lstrip(): 왼쪽 공백 제거
    - strip('문자열'): 좌우의 문자열을 제거
    - replace(문자열1, 문자열2): 앞의 문자열을 뒤의 문자열로 치환
  - ✓ 문자열의 분리와 결합 관련 메서드
    - split(): 공백으로 문자열을 분리해서 리스트로 반환
    - split('문자열'):문자열로 분리해서 리스트로 반환
    - split('문자열', 숫자):문자열로 분리하는데 숫자만큼만 분리해서 리스트로 반환
    - join(문자열 리스트): 문자열 리스트를 문자열을 각 요소 별로 문자열을 추가해서 결합
    - splitlines(): 줄 단위로 분리

Vector Data

- str
  - ✓ 맞춤 관련 메서드
    - center(숫자): 숫자 만큼의 자리를 확보하고 가운데 맞춤
    - ljust(숫자): 왼쪽 맞춤
    - rjust(숫자): 오른쪽 맞춤
    - center(숫자, 문자열): 숫자 만큼의 자리를 확보하고 가운데 맞춤한 후 빈자리는 문자열로 채움

Vector Data

- str
  - ✓ 문자열 메서드

```
#join() 메서드는 문자열을 결합하는데 사용되는 Separator를 join 메서드 앞에 사용
s = ','.join(['가나','다라','마바'])
print(s)
# 출력: 가나,다라,마바

s = ''.join(['가나','다라','마바'])
print(s)
# 출력 : 가나다라마바

#partition() 메서드는 문자열을 partition() 메서드의 첫번째 파라미터로 분리하여
#그 앞부분(prefix), partition 분리자(separator), 뒷부분 (suffix) 등 3개의 값을 Tuple로
    리턴
departure, _, arrival = "Seattle-Seoul".partition('-')
print(departure)
```

Vector Data

- str
  - ✓ 메서드

```
msg = "c:₩₩data₩₩data.txt"
#test의 존재여부
print(msg.find('test'))
#위의 문자열에서 파일의 확장자만 추출하기
ar = msg.split(".")
print("확장자:", ar[len(ar)-1])
print(msg.replace('data', 'python'))
```

```
가나,다라,마바
가나다라마바
Seattle
-1
확장자: txt
c:₩python₩python.txt
```

Vector Data

- str
  - ✓ 탭 문자 변환 메서드
    - expandtabs(숫자): 숫자를 생략하면 탭을 8자 공백으로 변경하고 숫자를 대입하면 숫자 만큼의 공백으로 변경
  - ✓ 문자열 확인 메서드 – is로 시작하는 메서드는 특별한 경우를 제외하고는 리턴 타입이 bool
    - isdigit()
    - isnumeric()
    - isdecimal()
    - isalpha()
    - isalnum()
    - islower()
    - isupper()
    - isspace()
    - istitle()
    - isidentifier()
    - isprintable()

Vector Data

- str
  - ✓ 메서드

```
msg = 'ggangpae1@gmail.com'
#위의 문자열에서 영문을 제외한 모든 글자 삭제
alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
result = ''
for ch in msg:
    if ch in alphabet:
        result = result + ch
print(result)
```

ggangpaegmailcom

Vector Data

- str
  - ✓ 메서드

```
msg = 'ggangpae1@gmail.com'
result = ''
for ch in msg:
    if ch.isalpha():
        result = result + ch
print(result)
```

ggangpaegmailcom

Vector Data

- str
  - ✓ 문자열 매핑 메서드 - maketrans()와 translate()를 이용하면 문자열 매핑 가능한데 maketrans()를 이용해서 치환할 문자열을 만들고 translate의 매개변수로 대입

```
instr = 'abcdef'
outstr = '123456'
trans = str.maketrans(instr, outstr)
msg = 'hello world'
print(msg.translate(trans))
```

h5llo worl4

Vector Data

- ❖ str
  - ✓ 문자열 인코딩
    - 파이썬의 문자열은 기본적으로 utf-8 인코딩을 사용하는데 기본 아스키 코드는 1바이트를 사용하고 나머지는 2바이트를 사용하는 방식의 인코딩
    - 문자열을 바이트 코드로 변환할 때는 encode()를 이용하고 특정 코드로 변환하고자 하는 경우에는 encode('인코딩방식') 메서드를 이용
    - 바이트 코드는 문자열과 직접 연산은 불가능
    - 바이트 코드를 문자열로 변환할 때는 decode('인코딩방식')을 이용해서 문자열로 변환하는데 동일한 인코딩 방식으로 디코딩 해야 하는데 그렇지 않으면 예외가 발생
    - ord('문자'): 문자를 코드 값으로 변환
    - chr(유니코드): 유니코드를 문자로 변환
    - 현재 시스템의 인코딩 확인: sys 모듈의 sys.stdin.encoding 과 sys.stdout.encoding

Vector Data

- ❖ str
  - ✓ 문자열 인코딩
    - ● 대표적인 인코딩
      - ◆ utf-8: 8비트 인코딩을 위해 가장 널리 사용되는 인코딩 방식
      - ◆ cp949(ms949): 한국어판 Microsoft Windows의 기본 인코딩
      - ◆ euc-kr: 웹에서 사용하던 한글 인코딩 방식
      - ◆ latin1(iso8859_1): 서유럽 라틴어 인코딩 방식

Vector Data

- str
  - ✓ 문자열 인코딩

```
import sys
print("현재 시스템의 입력 인코딩:", sys.stdin.encoding)
print("현재 시스템의 출력 인코딩:", sys.stdout.encoding)
b = '파이썬'.encode('utf-8')
print(b)
msg = b.decode('utf-8')
print(msg)
b = '파이썬'.encode('ms949')
print(b)
msg = b.decode('ms949')
print(msg)
```

```
현재 시스템의 입력 인코딩: utf-8
현재 시스템의 출력 인코딩: UTF-8
b'₩xed₩x8c₩x8c₩xec₩x9d₩xb4₩xec₩x8d₩xac'
파이썬
b'₩xc6₩xc4₩xc0₩xcc₩xbd₩xe3'
파이썬
```

Vector Data

- str
  - ✓ 유니코드에서 한글 자소 분리 및 자소를 글자로 변환
    - 한글 유니코드는 조성 19개, 중성 21개, 종성 28개로 구성
    - 한글 유니코드는 0xAC00에서 시작
    - 한글 유니코드 구하기: 0xAC00 + ((초성순서 * 21)+중성순서)*28 + 종성순서

Vector Data

- str
  - ✓ 유니코드에서 한글 자소 분리 및 자소를 글자로 변환: 한글 한 글자의 초성 중성, 종성의 위치 찾아내기

```
c = '한'
code = ord(c) - 0xAC00
chosung = code // (21*28)
jungsung = (code - chosung*21*28)//28
jongsung = (code - chosung*21*28 - jungsung*28)
print('초성:', chosung, "번째 글자")
print('중성:', jungsung, "번째 글자")
print('종성:', jongsung, "번째 글자")

print("==========================")
c = '대'
code = ord(c) - 0xAC00
chosung = code // (21*28)
jungsung = (code - chosung*21*28)//28
jongsung = (code - chosung*21*28 - jungsung*28)
print('초성:', chosung, "번째 글자")
print('중성:', jungsung, "번째 글자")
print('종성:', jongsung, "번째 글자")
```

Vector Data

- str
  - ✓ 유니코드에서 한글 자소 분리 및 자소를 글자로 변환: 한글 한 글자의 초성 중성, 종성의 위치 찾아내기

```
초성: 18 번째 글자
중성: 0 번째 글자
종성: 4 번째 글자
==========================
초성: 3 번째 글자
중성: 1 번째 글자
종성: 0 번째 글자
```

Vector Data

- Regular Expression – 정규 표현식
  - ✓ 특정한 규칙을 가진 문자열의 집합을 표현하는데 사용하는 형식 언어
  - ✓ Programming Language 나 Text Editor 등 에서 문자열의 검색과 치환을 위한 용도로 사용
  - ✓ 입력한 문자열에서 특정한 조건을 표현할 경우 일반적인 조건문으로는 다소 복잡할 수도 있지만 정규 표현식을 이용하면 간단하게 표현
  - ✓ 코드는 간단하지만 가독성이 떨어져서 표현식을 숙지하지 않으면 이해하기 힘들다는 문제점
  - ✓ 파이썬에서 정규식 지원은 re 모듈이 제공

Vector Data

- Regular Expression – 정규 표현식
  - ✓ 문자나 문자 패턴의 반복을 표현하는 메타 문자

메타 문자	설명	예
.	줄 바꿈 문자를 제외한 문자 1개	c.t는 cat, cbt 와 매칭
^	문자열의 시작과 매칭	^cat는 cat으로 시작
	[]안에서는 반대 문자열을 의미 되지 않음	c[^a]t 는 cbt와는 매칭되지만 cat와는 매칭
$	끝나는 문자열	
	[ ]안에서는 그냥 $	
[ ]	문자 집합을 의미	[abc]는 a, b, c 중 하나
		[0-9]는 0 부터 9 까지
		[0-9a-za-Z]는 영문자나 숫자
|	또는	ca|b 는 ca 또는 cb
( )	정규식을 그룹으로 묶기	

Vector Data

- Regular Expression – 정규 표현식
  - ✓ 문자나 문자 패턴의 반복을 표현하는 메타 문자

메타 문자	설명	예
*	0회 이상의 반복	ca*t는 c와 t 사이에 a를 반복할 수 있는데 생략 가능
+	1회 이상의 반복	ca*t는 c와 t 사이에 a를 반복할 수 있는데 생략 불가능
?	0이나 1회 반복	ca? 는 ct, cat 와 매칭
{m}	m회 반복을 허용	ca{2}는 a를 2회 반복
{m,n}	m회 부터 n회까지	ca{2,4}는 caat, caaat, caaaat와 매칭
{m,}	m 회 이상	

Vector Data

- Regular Expression – 정규 표현식
  - ✓ re 객체
    - re 모듈의 match 메서드는 문자열의 시작부터 정규식에 매칭된 경우 Match 인스턴스를 반환
    - search 메서드는 부분적으로 일치하는 문자열로 검사
    - 매칭이 되지 않은 경우에는 아무것도 리턴하지 않음
  - ✓ Match 클래스의 메서드
    - group(): 매칭된 문자열 반환
    - groups(): 매칭된 전체 그룹 문자열을 튜플 형식으로 반환
    - start(): 매칭된 문자열의 시작 위치 리턴
    - end(): 매칭된 문자열의 마지막 위치 리턴
    - span: 매칭된 문자열의 (시작, 끝) 위치를 리턴

Vector Data

- Regular Expression – 정규 표현식
  - ✓ re 객체

```
import re
match = re.match('[0-9]', '1234')
print(match.group())

match = re.match('[0-9]', 'abc')
print(match)

match = re.match('[0-9]+', '1234')
print(match.group())
```

```
1
None
1234
```

Vector Data

- Regular Expression – 정규 표현식
  - ✓ re 객체: 맨 앞에 공백이 오는 경우에는 ₩s를 이용해야 함

```
import re
match = re.match('[0-9]+', ' 1234')
print(match)
match = re.match('₩s[0-9]+', ' 1234')
print(match)
match = re.search('[0-9]+', ' 1234')
print(match)
```

```
None
<re.Match object; span=(0, 5), match=' 1234'>
<re.Match object; span=(1, 5), match='1234'>
```

Vector Data

- Regular Expression – 정규 표현식
  - ✓ re 정규식 표현

특수문자	설명
\\	백 슬래시 문자
\d	모든 숫자
\D	숫자가 아닌 문자
\s	화이트 스페이스와 매칭
\S	화이트 스페이스가 아닌 문자와 매칭
\w	숫자 또는 문자
\W	숫자 또는 문자가 아닌 것
\b	단어의 경계
\B	단어의 경계가 아님

Vector Data

- Regular Expression – 정규 표현식
  - ✓ re 정규식 표현

플래그	설명
re.I	대소문자를 구분하지 않고 매칭
re.M	여러 줄에 걸쳐서 매칭

Vector Data

- Regular Expression – 정규 표현식
  - ✓ re 클래스의 메서드
    - complie(pattern[.flags]): 정규식 인스턴스를 생성
    - search(pattern, string[. flags])
    - match(pattern, string[. flags])
    - split(pattern, string[. maxsplit=0]): pattern을 기준으로 분리
    - findall(pattern, string): pattern을 만족하는 모든 문자열을 추출
    - sub(pattern, repl, string[, count=0]): pattern을 찾아서 repl로 치환하는데 count는 치환 횟수를 제한

Vector Data

- Regular Expression – 정규 표현식
  - ✓ re 클래스의 메서드

```
import re
p = re.compile(r'[_a-zA-Z]₩w*')
m = p.search('123 abc 123 def')
print(m.group())
m = p.findall('123 abc 123 def')
print(m)
p = re.compile('the')
print(p.findall('The cat was hungry, They were scared because of the cat'))
p = re.compile('the', re.I)
print(p.findall('The cat was hungry, They were scared because of the cat'))
```

```
abc
['abc', 'def']
['the']
['The', 'The', 'the']
```

Vector Data

- Regular Expression – 정규 표현식
  - ✓ re 클래스의 메서드

```
import re
#주민등록번호 정규식 인스턴스 만들기
p = re.compile('(₩d{6})-?(₩d{7})')
num = '100000-2000000'
if p.search(num) != None:
    print("올바른 주민번호 형식입니다.")
else:
    print("올바른 주민번호 형식이 아닙니다.")
num = '10000-2000000'
if p.search(num) != None:
    print("올바른 주민번호 형식입니다.")
else:
    print("올바른 주민번호 형식이 아닙니다.")
```

```
올바른 주민번호 형식입니다.
올바른 주민번호 형식이 아닙니다.
```

Vector Data

- Regular Expression – 정규 표현식
  - ✓ re 클래스의 메서드

```
import re
#문자열 분리
result = re.split('[:. ]+', 'apple banana:orange tomato')
print(result)
#문자열 찾기
result = re.findall("app₩w*", 'application orange apple banana')
print(result)
#문자열 치환
result = re.sub("-","", "901225-1234567")
print(result)
result= re.sub("[:,|]", ",", "apple:orange banana|tomato")
print(result)
```

```
['apple', 'banana', 'orange', 'tomato']
['application', 'apple']
9012251234567
apple,orange banana,tomato
```

Vector Data

- Regular Expression – 정규 표현식
  - ✓ re 클래스의 메서드

```
import re
p = re.compile('^[a-zA-Z0-9+-_.]+@[a-zA-Z0-9-]+₩.[a-zA-Z0-9-.]+$')
emails = ['python@mail.example.com', 'python+kr@example.com', # 올바른 형식
          'python-dojang@example.co.kr', 'python_10@example.info', # 올바른 형식
          'python.dojang@e-xample.com',                          # 올바른 형식
          '@example.com', 'python@example', 'python@example-com']#잘못된형식

for email in emails:
    print(p.match(email) != None, end=' ')
```

True True True True True False False False

Vector Data

- bytes
  - ✓ bytes 클래스는 바이트의 모임을 표현하는 클래스로서 bytes는 한번 설정되면 다시 변경할 수 없는 Immutable 타입
  - ✓ 문자열을 바이트로 표현하기 위해서는 b'문자열' 와 같이 접두어 b를 앞에 추가
  - ✓ 문자열을 바이트들로 변경할 때는 encode()를 사용
  - ✓ 바이트들을 문자열로 변경할 때는 decode()를 사용
  - ✓ 문자열 안에서 유니코드 값을 사용하려면 ₩u 에 이어 유니코드 값을 작성
  - ✓ 네트워크를 이용한 전송이나 이미지 프로세싱 에서 많이 이용

Vector Data

- bytes

```
txt = b'Hello'
for b in txt:
    print(b, end="₩t")
print()
s1 = 'A₩u0041'
print(s1)
b = '파이썬'.encode('utf-8')
print(b)
msg = b.decode('utf-8')
print(msg)
```

```
72    101        108        108        111
AA
b'₩xed₩x8c₩x8c₩xec₩x9d₩xb4₩xec₩x8d₩xac'
파이썬
```

Vector Data

- list
  - ✓ 순서를 갖는 객체의 집합으로 대괄호 [ ]로 표현
  - ✓ 내부 데이터를 변경할 수 있는 자료형
  - ✓ 시퀀스 자료형의 특성을 지원하며 크기를 동적으로 변경 가능
  - ✓ 객체 생성
    - [ ]: 비어있는 리스트
    - [데이터 나열]: 나열된 데이터를 갖는 리스트
    - list(__iter__ 가 구현된 객체)
  - ✓ 항목 교체
    - 리스트[인덱스] = 값
    - 리스트[시작위치:종료위치] = [데이터 나열]
  - ✓ 데이터 삭제
    - 리스트[시작위치:종료위치] = [ ]
    - del 리스트[인덱스]

Vector Data

- list

```
li = [1,2,3]
print ("자료형:", type(li))
print ("길이:" , len(li))
print ("두번째 데이터:", li[1])
print ("뒤에서 두번째:", li[-1])
print ("두번째 부터 4번째 앞 까지:",li[1:3])
print ("리스트 결합:", li + li)
print ("리스트 반복:",li * 3)
```

```
자료형: <class 'list'>
길이: 3
두번째 데이터: 2
뒤에서 두번째: 3
두번째 부터 4번째 앞 까지: [2, 3]
리스트 결합: [1, 2, 3, 1, 2, 3]
리스트 반복: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Vector Data

- list

```
li = list(range(4)) #range를 이용한 list 생성
print(li)
del li[::2]#짝수번째 데이터 삭제
print(li)
#리스트 내의 모든 객체 순회
for i in li:
    print(i, end="₩t")
```

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

Vector Data

- list
  - ✓ 중첩 list
    - 리스트 안에 다른 리스트가 포함된 경우
    - 리스트는 다른 객체를 직접 저장하지 않고 객체들의 참조 만을 저장
    - 다른 리스트를 참조하는 경우 참조된 리스트의 내용이 변경되면 포함하고 있는 리스트의 내용도 변경됨

```
innerlist = list(range(4))
outerlist = ['begin', innerlist, 'end']
print(outerlist)
innerlist[0] = 200
print(outerlist)
```

```
['begin', [0, 1, 2, 3], 'end']
['begin', [200, 1, 2, 3], 'end']
```

Vector Data

- list
  - ✓ 메서드
    - append(값): 마지막에 데이터 추가
    - insert(인덱스, 데이터): 인덱스 위치에 데이터를 삽입
    - index(데이터): 데이터의 위치를 검색
    - count(값): 값의 개수 리턴
    - sort(key=적용할 함수에 대한 참조, reverse=False): 리스트의 데이터를 정렬
    - reverse(): 리스트의 순서를 변경
    - remove(데이터): 데이터의 첫번째 것을 삭제
    - pop(인덱스): 인덱스를 생략하면 가장 마지막 데이터를 삭제하고 리턴하고 인덱스를 대입하면 인덱스 번째를 삭제하고 리턴
    - extend(리스트): 리스트를 추가

Vector Data

- list
  - ✓ 정렬
    - sort()를 이용하면 데이터를 내부적으로 정렬 – Timsort(합병 정렬 과 삽입 정렬을 이용)
    - 값을 리턴하지 않음
    - key에 데이터를 비교하기 위한 변환 함수 설정 가능
    - 매개변수로 reverse=True를 대입하면 내림차순 가능

```
data = [30,50,10,40,20]
data.sort()
print("오름차순:", data)
data.sort(reverse=True)
print("내림차순:",data)
```

```
오름차순: [10, 20, 30, 40, 50]
내림차순: [50, 40, 30, 20, 10]
```

Vector Data

- list
  - ✓ 정렬
    - 문자열을 정렬할 때 대소문자 구분 없이 정렬 가능

```
data = ['Morning', 'Afternoon', 'evening' ,'Night' ]
data.sort()
print("대소문자 구분해서 정렬:", data)
data.sort(reverse=True)
print("내림차순 정렬:", data)
data.sort(key=str.lower, reverse=True)
print("내림차순 정렬:", data)
```

```
대소문자 구분해서 정렬: ['Afternoon', 'Morning', 'Night', 'evening']
내림차순 정렬: ['evening', 'Night', 'Morning', 'Afternoon']
내림차순 정렬: ['Night', 'Morning', 'evening', 'Afternoon']
```

Vector Data

- list
  - ✓ 정렬
    - 원본을 정렬하지 않고 원본을 복사해서 정렬한 결과를 리턴하는 함수는 sorted()
    - key와 reverse 키워드를 동일하게 지원

```
data = [30,50,10,40,20]
print("오름차순:", sorted(data))
print("내림차순:",sorted(data, reverse=True))
```

```
data = [30,50,10,40,20]
data = sorted(data)
print("오름차순:", data)
data.reverse()
print("내림차순:",data)
```

```
오름차순: [10, 20, 30, 40, 50]
내림차순: [50, 40, 30, 20, 10]
오름차순: [10, 20, 30, 40, 50]
내림차순: [50, 40, 30, 20, 10]
```

Vector Data

- list
  - ✓ list 대안
    - 리스트는 융통성 있고 사용하기 편하지만 세부 요구 사항에 따라 더 나은 자료 구조도 있음
    - 실수를 천만 개 저장해야 할 때는 배열이 훨씬 더 효율적인데 배열은 모든 기능을 갖춘 float 객체 대신 C 언어의 배열과 마찬가지로 기계가 사용하는 형태로 표현된 바이트 값만 저장하기 때문
    - 리스트의 양쪽 끝에 항목을 계속 추가하거나 삭제하면서 FIFO나 LIFO 데이터 구조를 구현할 때는 deque(양쪽을 사용하는 큐)가 더 빠름
    - NumPy와 SciPy가 제공하는 고급 배열 및 행렬 연산 덕분에 파이썬이 과학 계산 애플리케이션에서 널리 사용되게 되었는데 NumPy는 숫자 뿐 만 아니라 사용자 정의 레코드도 저장할 수 있 는 다차원 동형 배열 및 행렬을 구현하고 요소 단위에서 효율적으로 연산할 수 있게 해 줌
    - SciPy는 NumPy를 기반으로 작성된 라이브러리로서 선형 대수학, 수치 해석, 통계학에 나오는 여러 과학 계산 알고리즘을 제공하는데 SciPy는 Netlib 리포지토리 (http://www.netlib. org)가 제공하는 C 및 포트란 코드 기반을 활용함으로써 빠르고 신뢰성이 높아서 SciPy 는 C 와 포트란에서 최적화되고 업계에서 입증된 수치 계산 함수를 대화형 고급 파이썬 API를 통해 과학자들에게 제공

Vector Data

- list
  - ✓ list 대안
    - queue
      - queue 모듈에서는 동기화된(즉, 스레드 안전한) Queue. LifoQueue. PriorityQueue 클래스를 제공하는데 이 클래스들은 스레드 간에 안전하게 통신하기 위해 사용
      - 0보다 큰 maxsize 인수를 생성자에 전달해서 바인딩할 수 있는데 공간이 꽉 찼을 때 항목을 버리지 않고 대신 새로운 항목의 추가를 블로킹하고 다른 스레드에서 큐 안의 항목을 제거해서 공간을 확보해줄 때까지 기다리기 때문에 활성화된 스레드 수를 조절하기 좋음
    - multiprocessing
      - 멀티프로세싱 모듈은 queue.Queue와 비슷하지만 프로세스 간 통신을 지원하기 위해 설계된 고유한 Queue 클래스를 구현
      - 태스크 관리에 특화된 multiprocessing .JoinableQueue 클래스도 제공
    - asyncio
      - asyncio 모듈은 queue 및 multiprocessing 모듈에 포함된 클래스로 부터 영감을 얻은 Queue, LifoQueue. PriorityQueue, JoinableQueue 클래스를 제공하지만 비동기 프로그래밍 환경에서 작업을 관리하는 데 주안점을 두고 있음
    - heapq
      - heapq는 queue 클래스를 구현하지는 않지만 가변 시퀀스를 힙 큐나 우선순위 큐로 사용할 수 있게 해주는 heappush() 와 heappop() 등의 함수를 제공

Vector Data

- list
  - ✓ list 대안
    - ● 배열

```
#array
from array import array
from random import random
from time import time

start = time()
floats = array('d', (random() for i in range(10 ** 7)))
end = time()
print(end - start)


start = time()
ar = []
for i in range(10 ** 7):
    ar.append(random())
end = time()
print(end - start)
```

1.4166338443756104
1.4865012168884277

Vector Data

- list
  - ✓ list 대안
    - deque
      - 큐(queue)는 선입선출(FIFO) 방식으로 작동하는 반면 데크(deque)는 앞, 뒤 양쪽 방향에서 엘리먼트(element)를 추가하거나 제거할 수 있음
      - 데크는 양 끝 엘리먼트의 append와 pop이 압도적으로 빠름
      - 컨테이너(container)의 양끝 엘리먼트(element)에 접근하여 삽입 또는 제거를 할 경우 일반적인 리스트(list)가 이러한 연산에 O(n)이 소요되는 데 반해 데크(deque)는 O(1)로 접근 가능
      - collections.deque 에 존재하는 메서드
        - deque.append(item): item을 데크의 오른쪽 끝에 삽입
        - deque.appendleft(item): item을 데크의 왼쪽 끝에 삽입
        - deque.pop(): 데크의 오른쪽 끝 엘리먼트를 가져오는 동시에 데크에서 삭제
        - deque.popleft(): 데크의 왼쪽 끝 엘리먼트를 가져오는 동시에 데크에서 삭제

Vector Data

- list
  - ✓ list 대안
    - deque

```
#deque
from collections import deque
deq = deque()
for i in range(10 ** 7):
    deq.append(1)
```

Vector Data

- collections 모듈
  - ✓ deque - 최근의 데이터만 저장 가능

```
from collections import deque

history = deque(maxlen=3)

data = ['두란', '레너드', '헌즈','해글러']

for boxer in data:
    history.append(boxer)

for boxer in history:
    print(boxer)
```

레너드
헌즈
해글러

Vector Data

- tuple
  - ✓ 임의 객체들이 순서를 가지는 모임으로 리스트와 유사하며 접근 방법도 동일
  - ✓ 내부 데이터의 변경이 불가능
  - ✓ 튜플은 시퀀스 자료형이므로 인덱싱과 슬라이싱, 연결하기, 반복하기, 길이 정보 등의 연산 가능
  - ✓ 튜플은 ( )안에 표현
  - ✓ ( )를 사용하지 않고 ,로 데이터를 구분해서 데이터를 나열해도 튜플로 처리
  - ✓ ( )를 하는 경우 데이터가 1개 일 때는 ,를 마지막에 추가
  - ✓ list()와 tuple()를 이용해서 서로 간에 변환 가능하며 tuple 함수에 range 사용 가능
  - ✓ 포함된 데이터 개수를 리턴 해주는 count 메서드 소유
  - ✓ 포함된 데이터 위치를 리턴 해주는 index 메서드 소유

```
t = (1,2,3,2,2,3)
print("2의 개수:", t.count(2))
print("2의 위치:", t.index(2))
print("2의 위치:", t.index(2,3))
```

```
2의 개수: 3
2의 위치: 1
2의 위치: 3
```

Vector Data

- tuple
  - ✓ list 와 tuple은 데이터를 나누어 저장하는 것이 가능 – unpacking이라 하는데 반대되는 작업을 packing 이라고 함

```
#튜플의 unpacking
x,y=1,2
x,y=y,x
print(x, y)

a, b, *rest = range(5)
print(a, b, rest)

a, b, *rest = range(3)
print(a, b, rest)

a, *body, c, d = range(5)
print(a, body, c, d)
```

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

Vector Data

- tuple
  - ✓ 튜플을 이용해서 조건을 설정해서 if 처럼 사용하는 것이 가능한데 (거짓일 때 수식, 참일 때 수식)[조건] 의 형태

```
a = 5
if a > 5:
    a = a*2
    print(a)
else:
    a = a-4
    print(a)

a = 5
b = (a-4, a*2) [a>5]
print(b)
```

```
1
1
```

Vector Data

- tuple
  - ✓ 튜플은 레코드를 담고 있는데 튜플의 각 항목은 레코드의 필드 하나를 의미하며 항목의 위치가 의미를 결정
  - ✓ 튜플을 단지 불변 리스트로 생각한다면 경우에 따라 항목의 크기와 순서가 중요할 수도 있고 그렇지 않을 수도 있지만 튜플을 필드의 집합으로 사용하는 경우에는 항목 수가 고정되어 있고 항목의 순서가 중요

Vector Data

- tuple
  - ✓ 레코드로서의 튜플

```
#레코드로서의 튜플
traveler_ids = [('USA', '31195855'), ('BRA', 'CE342567'), ('ESP', 'XDA205856')]

for passport in traveler_ids:
    print('%s/%s' % passport)
print()

for passport in sorted(traveler_ids):
    print('%s/%s' % passport)
```

```
USA/31195855
BRA/CE342567
ESP/XDA205856

BRA/CE342567
ESP/XDA205856
USA/31195855
```

Vector Data

- tuple
  - ✓ 컬럼에 이름을 사용하는 튜플
    - collections.namedtuple( ) 함수는 필드명과 클래스명을 추가한 튜플의 서브클래스를 생성하는 팩토리 함수로서 디버깅할 때 유용
    - 필드명이 클래스에 저장되므로 namedtuple()로 생성한 객체는 튜플과 동일한 크기의 메모리만 사용
    - 속성을 객체마다 존재하는 __dict__에 저장하지 않으므로 일반적인 객체보다 메모리를 적게 사용

```
from collections import namedtuple
City = namedtuple('City', 'name country population')
seoul = City('Seoul', 'Korea', 973.3)
print(seoul.name, seoul.country, seoul.population)
```

```
Seoul Korea 973.3
```

Vector Data

- set
  - ✓ 데이터를 순서와 상관없이 중복 없이 모아놓은 자료형으로 set과 frozenset(객체의 내용을 변경할 수 없는 set) 두 가지 자료형을 제공
  - ✓ 빈 객체 생성은 set()
  - ✓ 처음부터 데이터를 가지고 있는 경우에는 {데이터 나열}
  - ✓ 데이터가 없는 경우에는 { } 대신 set()으로 생성하는 이유는 디셔너리 와의 혼동의 문제
  - ✓ 튜플, 문자열, 리스트 및 디셔너리 로부터 생성이 가능

Vector Data

- set

```
hashset = set()
print(hashset)
hashset = {1,2,3}
print(hashset)
hashset = set((1,2,3,2))
print(hashset)
hashset = set('hello world')
print(hashset)
hashset = set([1,3,2])
print(hashset)
```

```
set()
{1, 2, 3}
{1, 2, 3}
{'l', 'o', 'w', 'd', 'h', 'e', 'r', ' '}
{1, 2, 3}
```

Vector Data

- set
  - ✓ 메서드
    - 데이터의 추가는 add(데이터), update([데이터 나열])
    - 데이터의 복사는 copy()
    - 데이터의 전체 제거는 clear()
    - 데이터의 한 개 제거는 discard(데이터) – 없으면 그냥 통과
    - 데이터의 한 개 제거는 remove(데이터) – 없으면 예외
    - pop(): 1개 제거하면서 리턴
  - ✓ set의 집합 연산
    - union(다른 set), intersection(다른 set), difference(다른 set), symmetric_difference(다른 set)의 메서드가 제공되는데 결과를 리턴
    - update(), intersection_update(), difference_update(), symmetric_difference_update()는 호출하는 객체의 데이터를 변경

Vector Data

- set
  - ✓ 포함 관계 연산자
    - 데이터 in set : 앞의 요소가 포함되어 있는지 리턴
    - 데이터 not in set: 앞의 요소가 포함되어 있지 않은지 리턴
    - isupperset(다른 set): 다른 set을 포함하고 있는지 여부 리턴
    - issubset(다른 set): 다른 set의 서브셋인지 여부 리턴
    - isdisjoint(다른 set): 교집합이 공집합인지 여부 리턴

```
a = {1,2,3,4,5}
b = {4,5,6,7,8}
print(a.union(b))
print(a.intersection(b))
print(a.difference(b))
print(a.symmetric_difference(b))
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
{4, 5}
{1, 2, 3}
{1, 2, 3, 6, 7, 8}
```

Vector Data

- dict
  - ✓ Key와 Value를 쌍으로 저장하는 자료형
  - ✓ {키:데이터, 키:값....}로 생성 가능하고 fromkeys([키 나열], 기본값)을 이용해서 key에 기본값을 저장하는 dict를 생성하는 것이 가능
  - ✓ Key의 순서는 없으며 Key의 값은 중복될 수 없음
  - ✓ Key의 자료형에 제한은 없지만 거의 str
  - ✓ Key 값에 해당하는 데이터를 가져올 때는 디셔너리[키]
  - ✓ 디셔너리[키] = 데이터 를 이용하면 키의 데이터가 없으면 삽입이 되고 있으면 수정
  - ✓ 없는 키의 값을 호출하면 에러
  - ✓ len(디셔너리)는 키의 개수
  - ✓ 여러 개의 데이터를 묶을 때 주로 사용

Vector Data

- dict

```
member = {'baseball':9, 'soccer':11,"volleyball":6}
print(member)
print(member['baseball'])
print(member['basketball'])

{'baseball': 9, 'soccer': 11, 'volleyball': 6}
9
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/var/folders/42/fxl3_n31121f16yj69fbrb_m0000gn/T/ipykernel_15769/2298363704.py in
      <module>
       2 print(member)
       3 print(member['baseball'])
----> 4 print(member['basketball'])

KeyError: 'basketball'
```

Vector Data

- dict
  - ✓ 생성하는 다른 방법은 dict(키=데이터, 키=데이터)로 가능
  - ✓ 키의 리스트나 값의 리스트가 존재하는 경우 dict(zip(키의 리스트, 값의 리스트))로 생성 가능
  - ✓ 사전을 for에 사용하면 키의 값들이 순서대로 대입됨

```
keys = ['one', 'two', 'three']
values = (1,2,3)
dic = dict(zip(keys, values))
for key in dic:
    print(key,":", dic[key])
```

```
one : 1
two : 2
three : 3
```

Vector Data

- dict
  - ✓ keys(): 모든 key를 리턴 – for를 이용해서 데이터를 순서대로 접근 가능
  - ✓ values(): 모든 value를 리턴 – for를 이용해서 데이터를 순서대로 접근 가능
  - ✓ items(): 모든 데이터 리턴 – 키와 값을 tuple로 묶어서 리턴
  - ✓ copy(): 복제
  - ✓ get(key [,x]): key에 값이 존재하면 값을 리턴하고 없으면 x 리턴하는데 적절히 이용하면 if의 형태로 사용하는 것이 가능
  - ✓ setdefault(key [,x]):get 과 동일하지만 값이 존재하지 않으면 x로 설정
  - ✓ update(key=value): 디셔너리의 key에 대한 value를 수정하는데 key=value 대신에 다른 dict를 대입하면 다른 매개변수로 대입된 dict를 병합 – dict[key] = value를 이용해서도 수정은 가능
  - ✓ del 디셔너리[키]는 키의 데이터 삭제
  - ✓ popitem(): 마지막 하나의 튜플을 반환하고 제거
  - ✓ pop(key): key에 해당하는 데이터를 반환하고 제거
  - ✓ clear(): 모두 삭제

Vector Data

- dict
  - ✓ get을 이용한 if

```
a = 5
if a == 1:
    print("일")
elif a == 2:
    print("이")
elif a == 3:
    print("삼")
else:
    print("알 수 없음")

data = {1 : "일", 2 : "이", 3 : "삼"}
b = data.get(a, "알 수 없음")

print(b)
```

```
알 수 없음
알 수 없음
```

Vector Data

- dict
  - ✓ 레코드를 dict로 사용

```
#데이터를 생성하는 부분
class Person:
    def __init__(self):
        self.name = ""
        self.age = 0

person1 = Person()
person1.name = '아담'
person1.age = 25

person2 = {'name':"류시아", "age":24}

#데이터를 출력하는 부분
print('이름:', person1.name, '₩t나이:', person1.age)

for key in person2:
    print(key,":", person2[key], end='₩t')

#클래스의 속성을 name 에서 nick 으로 변경 하고 dict 의 name 도 nick으로 변경한 후
    실행
```

이름: 아담 나이: 25
name : 류시아 age : 24

Vector Data

- dict
  - ✓ 레코드를 dict로 사용

```
table = []

person1 = {'name':"사이다", "age":23}
person2 = {'name':"아담", "age":25}
person3 = {'name':"류시아", "age":24}

table.append(person1)
table.append(person2)
table.append(person3)

print('{0:10s}{1:10s}'.format('이름','나이'))
for person in table:
    print('{0:10s}{1:10s}'.format(person['name'],str(person['age'])))

print()
table.sort(key = lambda d : d['name'])

print('{0:10s}{1:10s}'.format('이름','나이'))
for person in table:
    print('{0:10s}{1:10s}'.format(person['name'],str(person['age'])))
```

Vector Data

- dict
  - ✓ 레코드를 dict로 사용

이름	나이
사이다	23
아담	25
류시아	24

이름	나이
류시아	24
사이다	23
아담	25

Vector Data

- dict
  - ✓ 레코드를 dict로 사용

```
players = []
players.append({'name':'권투', 'player':['마빈 해글러', '슈거레이 레너드', '토마스 헌즈']})
players.append({'name':'야구', 'player':['조계현', '최향남', '이대진']})
players.append({'name':'축구', 'player':['가린샤', '에우제비오', '차범근']})
players.append({'name':'배구', 'player':['배유나', '한유미']})
players.append({'name':'농구', 'player':['래리 존슨', '알론조 모닝', '레지 밀러', '스카티
    피펜']})
players.append({'name':'당구', 'player':['브롬달', '이상천', '세이기너', '차유람']})

for sport in players:
    print('{0:5s}'.format(sport['name']))
    print('₩t', end='')
    for player in sport['player']:
        print('{0:10s}'.format(player), end='')
    print('₩n')
```

Vector Data

- dict
  - ✓ 레코드를 dict로 사용

권투
마빈 해글러 슈거레이 레너드 토마스 헌즈

야구
조계현 최향남 이대진

축구
가린샤 에우제비오 차범근

배구
배유나 한유미

농구
래리 존슨 알론조 모닝 레지 밀러 스카티 피펜

당구
브룸달 이상천 세이기너 차유람

Vector Data

- enumerate
  - ✓ 인덱스(index)와 원소를 동시에 접근하면서 루프를 돌릴 수 있도록 해주는 내장 함수
  - ✓ list, tuple, set, dict 에서 사용 가능
  - ✓ dict는 key를 순차적으로 접근

```
for idx, element in enumerate(['A', 'B', 'C']):
    print(idx, element)

for idx, key in enumerate({'name':'adam', 'address':'mokdong'}):
    print(idx, key)
```

```
0 A
1 B
2 C
0 name
1 address
```

Vector Data

- Comprehension
  - ✓ 반복 가능한 객체(iterable)들을 이용해서 새로운 반복 가능한 객체를 축약한 형태로 생성하는 방법
  - ✓ 대표적으로 리스트(List), 튜플(Tuple), 집합(Set) 사전(Dict), 생성자(Generator) 등이 있음
  - ✓ 반복 가능한 객체들을 간소화하게 되면 코드 구성이 깔끔해져 더 읽기 쉬운 코드를 구성할 수 있음
  - ✓ 조건문 등을 추가하여 코드를 간략화 할 수 있음

```
L = [i ** 2 for i in range(10)]
print(L)
```

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Vector Data

- Comprehension
  - ✓ sequence 객체에 연산을 적용해서 새로운 sequence를 얻고자 하는 경우 <연산식> for 임시변수 in 시퀀스객체 (if 조건식)

```
#0부터 4까지 3승을 해서 결과를 가져오기
li1 = [i ** 3 for i in range(5)]
print(li1)

li2 = ['태연', '아이린', '재경']
#li2에서 글자수가 3이상인 데이터만 추출하기
li3 = [i for i in li2 if len(i) > 2]
for imsi in li3:
    print(imsi)

li4 = [1,2,3]
li5 = [4,5,6]
result = [x*y for x in li4 for y in li5]
print(result)
```

[0, 1, 8, 27, 64]
아이린
[4, 5, 6, 8, 10, 12, 12, 15, 18]

Vector Data

- Comprehension
  - ✓ 중첩해서 사용하는 것이 가능

```
L = [['a', 'b', 'c'], ['d', 'e', 'f']]
flatten = [j for i in L for j in i]
print(flatten)
extend = [[j + j for j in i] for i in L]
print(extend)
```

['a', 'b', 'c', 'd', 'e', 'f']
[['aa', 'bb', 'cc'], ['dd', 'ee', 'ff']]

Vector Data

- Comprehension
  - ✓ tuple, set, dict 도 생성 가능

```
data = [0, 7, 6, 9, 2, 3]
T = tuple((i for i in data))
print(T)

text = "Adam"
S1 = set([i for i in text])
print(S1)
S2 = {i for i in text}
print(S2)

text = "Adam"
D = {i : text.count(i) for i in text}
print(D)
```

(0, 7, 6, 9, 2, 3)

{'m', 'a', 'd', 'A'}
{'m', 'a', 'd', 'A'}

{'A': 1, 'd': 1, 'a': 1, 'm': 1}

Vector Data

- Comprehension
  - ✓ for 와 if를 같이 이용
    - for문과 if 조건문이 한 개일 때 작성 형식: [식 for 변수 in 리스트 if 조건식]
    - for문과 if 조건문이 여러 개일 때 작성 형식: [식 for 변수1 in 리스트1 if 조건식1 ... for 변수N in 리스트N if 조건식N]
    - for문, if .. else .. 조건문 작성 형식: [식1 if 조건식 else 식2 for 변수 in 리스트]

Vector Data

- Comprehension
  - ✓ for 와 if를 같이 이용

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
squares = [[x**2 for x in row] for row in matrix]
print(squares)

a = [n**2 for n in range(1, 11) if n % 2 == 0]
print(a)

b = [x for x in range(1, 11) if x > 4 if x % 2 == 0]
c = [x for x in range(1, 11) if x > 4 and x % 2 == 0]
print(b)
print(c)

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
filtered = [[x for x in row if x % 3 == 0] for row in matrix if sum(row) >= 10]
print(filtered)

a = [n**2 if n % 2 == 0 else n ** 3 for n in range(1, 11)]
print(a)
```

Vector Data

- Comprehension
  - ✓ for 와 if를 같이 이용

```
[[1, 4, 9], [16, 25, 36], [49, 64, 81]]
[4, 16, 36, 64, 100]
[6, 8, 10]
[6, 8, 10]
[[6], [9]]
[1, 4, 27, 16, 125, 36, 343, 64, 729, 100]
```

Vector Data

- queue
  - ✓ 파이썬에서는 queue모듈에서 큐(Queue), 우선 순위 큐(Priority Queue), 스택(LIFO Queue)을 제공
  - ✓ 큐 모듈은 멀티 스레드 환경을 고려하여 작성되었기에 여러 스레드가 동시에 queue 모듈에 데이터를 저장하고 데이터를 꺼내도 정상적으로 작동하는 것을 보장
  - ✓ 인스턴스 생성시 인자로 전달되는 maxsize는 해당 인스턴스가 저장할 수 있는 최대 아이템 개수
  - ✓ 명시적으로 아이템의 개수를 지정하지 않거나 0보다 같거나 작은 경우 저장할 수 있는 큐의 크기는 메모리가 수용하는 한 무한히 늘어날 수 있음
  - ✓ queue.Queue(maxsize): 선입 선출(FIFO First-In, First-Out) 큐 인스턴스를 생성
  - ✓ queue.LifoQueue(maxsize): 일반적으로 스택(Stack)이라 불리는 후입 선출(LIFO Last-In, First-Out) 큐 인스턴스를 생성
  - ✓ queue.PriorityQueue(maxsize): 우선 순위 큐 인스턴스를 생성하는데 입력되는 아이템의 형식은 (순위, 아이템)의 튜플로 입력되며 우선 순위는 작을수록 높은 순위를 갖음

Vector Data

- queue
  - ✓ 3개의 클래스는 여러 메서드를 동일하게 소유
  - ✓ 메서드의 의미는 동일하지만 해당 클래스의 정렬 방식에 따라서 GET 계열 메서드의 결과가 달라짐
    - Qsize(): 큐 인스턴스에 입력된 아이템의 개수를 반환
    - put(item[, block[, timeout]]): 큐 인스턴스에 아이템을 입력
    - put_nowait(item): 블로킹(blocking)없이 큐 인스턴스에 아이템을 입력
    - 큐 인스턴스가 꽉 차 있는 경우에는 queue.Full 예외 발생
    - get([block[, timeout]]): 생성된 큐 인스턴스 특성에 맞추어, 아이템 1개를 반환
    - get_nowait(): 블로킹(blocking)없이 큐 인스턴스에 들어있는 아이템을 반환
    - 큐 인스턴스에 아이템이 없는 경우에는 queue.Empty 예외 발생
  - ✓ 발생할 수 있는 예외
    - queue.Empty: 큐 인스턴스에 아이템이 없는 경우 발생
    - queue.Full: 큐 인스턴스에 아이템이 꽉 찬 경우에 발생

Vector Data

- queue

```
import queue
li=['태연','아이린','재경']
singer=queue.Queue()
#singer=queue.LifoQueue()
for x in li:
    singer.put(x)

idx = 0
size = singer.qsize()
while idx < size:
    print(singer.get(idx))
    idx = idx + 1
print()
singer=queue.PriorityQueue()
singer.put((30, '태연'))
singer.put((25, '수지'))
singer.put((28, '아이린'))

idx = 0
size = singer.qsize()
while idx < size:
    print(singer.get(idx))
    idx = idx + 1
```

Vector Data

- queue

  태연
  아이린
  재경

  (25, '수지')
  (28, '아이린')
  (30, '태연')

Vector Data

- queue
  - ✓ 큐 인스턴스 생성시 저장할 수 있는 아이템 크기를 명시적으로 지정할 수 있음
  - ✓ 이러한 경우 큐 인스턴스에 저장된 아이템이 없는 상태에서 get()이 호출되거나 반대로 아이템이 저장할 수 있는 사이즈를 꽉 채운 상태에서 put()이 호출되는 경우 큐 인스턴스는 블록킹(Blocking) 됨

```
import queue
q=queue.Queue(2) #아이템이 2개만 저장가능하도록 설정
q.put('태연')
q.put('수지') #큐의 저장 한계
q.put('아이린') #다른 스레드가 아이템을 가지고 갈때까지 무한 대기
```

Vector Data

- queue
  - ✓ 무한대기하는 상태를 피하고자 put_nowait()과 get_nowait()메서드를 지원
  - ✓ 두 메서드는 큐 인스턴스의 상태에 상관없이 블록킹 되지 않고 즉시 결과를 리턴
  - ✓ 대신 큐 인스턴스가 꽉 찬 경우에는 queue.Full 예외를 큐 인스턴스가 빈 경우에는 queue.Empty 예외를 발생하기 때문에 해당 상태에 맞추어서 적절한 처리를 할 수 있음

```
import queue
q=queue.Queue(2) #아이템이 2개만 저장가능하도록 설정
q.put_nowait('태연')
q.put_nowait('수지') #큐의 저장 한계
q.put_nowait('아이린')

q.get_nowait()
q.get_nowait()
q.get_nowait() #큐 인스턴스가 빈 경우
```

Vector Data

- collections 모듈
  - ✓ Counter 클래스를 이용하면 데이터의 개수 나 집계를 하는 것이 편리

```
from collections import Counter

counter = Counter(['red', 'blue', 'red', 'blue', 'green', 'red'])

print(counter['blue'])
print(counter['red'])
print(dict(counter))
```

```
2
3
{'red': 3, 'blue': 2, 'green': 1}
```

Vector Data

- collections 모듈
  - ✓ Counter 클래스를 이용하면 데이터의 개수 나 집계를 하는 것이 편리

```
portfolio = [
    ('GOOG', 100, 490.1),
    ('IBM', 50, 91.1),
    ('CAT', 150, 83.44),
    ('IBM', 100, 45.23),
    ('GOOG', 75, 572.45),
    ('AA', 50, 23.15)
]

from collections import Counter
total_shares = Counter()

# 데이터 개수 세기
for name, shares, price in portfolio:
    total_shares[name] += 1

print(total_shares)
print()

for x in total_shares:
    print(total_shares[x])
```

Vector Data

- collections 모듈
  - ✓ Counter 클래스를 이용하면 데이터의 개수 나 집계를 하는 것이 편리

```
# price 의 합계 구하기
print()
total_shares = Counter()
for name, shares, price in portfolio:
    total_shares[name] += price
print(total_shares)

#상위 2개만 추출
print(total_shares.most_common(3))
```

```
Counter({'GOOG': 2, 'IBM': 2, 'CAT': 1, 'AA': 1})

2
2
1
1

Counter({'GOOG': 1062.5500000000002, 'IBM': 136.32999999999998, 'CAT': 83.44,
     'AA': 23.15})
[('GOOG', 1062.5500000000002), ('IBM', 136.32999999999998), ('CAT', 83.44)]
```

Vector Data

- collections 모듈
  - ✓ defaultdict 클래스를 이용하면 하나의 key 에 여러 개의 데이터 설정 가능

```
portfolio = [
    ('GOOG', 100, 490.1),
    ('IBM', 50, 91.1),
    ('CAT', 150, 83.44),
    ('IBM', 100, 45.23),
    ('GOOG', 75, 572.45),
    ('AA', 50, 23.15)
]

from collections import defaultdict
holdings = defaultdict(list)
for name, shares, price in portfolio:
    holdings[name].append((shares, price))

for x in holdings:
    print(x , " : ", holdings[x])
```

```
GOOG  :  [(100, 490.1), (75, 572.45)]
IBM  :  [(50, 91.1), (100, 45.23)]
CAT  :  [(150, 83.44)]
AA  :  [(50, 23.15)]
```

Vector Data

- collections 모듈
  - ✓ OrderedDict
    - 키의 순서를 유지하는 디셔너리를 만들고자 할 때는 collections 모듈의 OrderedDict를 사용
    - dict와 동일하게 동작하지만 데이터가 추가된 순서를 기억해서 데이터를 순서대로 처리할 수 있도록 해주는 자료형

```
from collections import OrderedDict
keys = ['one', 'two', 'three']
values = (1,2,3)
dic = dict(zip(keys, values))
for key in dic:
    print(key,":", dic[key])
print("======================")
dic = OrderedDict(zip(keys, values))
for key in dic:
    print(key,":", dic[key])
```

```
one : 1
two : 2
three : 3
======================
one : 1
two : 2
three : 3
```

Vector Data

- itertools 모듈
  - ✓ 효율적인 루핑을 위한 이터레이터를 만드는 함수 패키지
  - ✓ https://docs.python.org/ko/3/library/itertools.html
  - ✓ 무한 이터레이터

이터레이터	인자	결과	예
count()	start, [step]	start, start+step, start+2*step, ...	count(10) --> 10 11 12 13 14 ...
cycle()	p	p0, p1, ... plast, p0, p1, ...	cycle('ABCD') --> A B C D A B C D ...
repeat()	elem [,n]	elem, elem, elem, ... 끝없이 또는 최대 n 번	repeat(10, 3) --> 10 10 10

Vector Data

- itertools 모듈
  - ✓ 가장 짧은 입력 시퀀스에서 종료되는 이터레이터

이터레이터	인자	결과	예
`accumulate()`	p [,func]	p0, p0+p1, p0+p1+p2, …	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
`chain()`	p, q, …	p0, p1, … plast, q0, q1, …	`chain('ABC', 'DEF') --> A B C D E F`
`chain.from_iterable()`	iterable	p0, p1, … plast, q0, q1, …	`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
`compress()`	data, selectors	(d[0] if s[0]), (d[1] if s[1]), …	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
`dropwhile()`	pred, seq	seq[n], seq[n+1], pred 가 실패할 때 시작	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
`filterfalse()`	pred, seq	pred(elem)이 거짓인 seq 의 요소들	`filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8`
`groupby()`	iterable[, key]	key(v)의 값으로 그룹화된 서브 이터레이터들	
`islice()`	seq, [start,] stop [, step]	seq[start:stop:step]의 요소들	`islice('ABCDEFG', 2, None) --> C D E F G`

Vector Data

- itertools 모듈
  - ✓ 가장 짧은 입력 시퀀스에서 종료되는 이터레이터

이터레이터	인자	결과	예
pairwise()	iterable	(p[0], p[1]), (p[1], p[2])	pairwise('ABCDEFG') --> AB BC CD DE EF FG
starmap()	func, seq	func(*seq[0]), func(*seq[1]), ...	starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
takewhile()	pred, seq	seq[0], seq[1], pred가 실패할 때까지	takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
tee()	it, n	it1, it2, ... itn 하나의 이터레이터를 n개의 이터레이터로 나눕니다	
zip_longest()	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-

Vector Data

- itertools 모듈
  - ✓ 효율적인 루핑을 위한 이터레이터를 만드는 함수 패키지
  - ✓ 조합형 이터레이터 함수
    - combinations(iterable, r) : iterable에서 원소 개수가 r개인 조합 뽑기
    - combinations_with_replacement(iterable,r): iterable에서 원소 개수가 r개인 중복 조합 뽑기
    - product(*iterables, repeat=1): 여러 iterable의 데카르트곱 리턴
    - permutations(iterable,r=None): iterable에서 원소 개수가 r개인 순열 뽑기

Vector Data

- itertools 모듈
  - ✓ 사용

```
from itertools import combinations
from itertools import combinations_with_replacement
from itertools import permutations
from itertools import product

l = [1,2,3]
for i in combinations(l,2):
    print(i)

l = ['A', 'B', 'C']
for i in combinations_with_replacement(l,2):
    print(i)

l = ['A', 'B', 'C']
for i in permutations(l): #r을 지정하지 않거나 r=None으로 하면 최대 길이의 순열이
    리턴된다!
    print(i)
```

Vector Data

- itertools 모듈
  - ✓ 사용

```
l1 = ['A', 'B']
l2 = ['1', '2']
for i in product(l1,l2,repeat=1): #l1과 l2의 모든 쌍을 지어 리턴한다
    print(i)

for i in product(l1,repeat=3): #product(l1,l1,l1,repeat=1)과 동일한 출력
    print(i)
```

Vector Data

- itertools 모듈
  - ✓ 사용

```
(1, 2)
(1, 3)
(2, 3)
('A', 'A')
('A', 'B')
('A', 'C')
('B', 'B')
('B', 'C')
('C', 'C')
('A', 'B', 'C')
('A', 'C', 'B')
('B', 'A', 'C')
('B', 'C', 'A')
('C', 'A', 'B')
('C', 'B', 'A')
('A', '1')
('A', '2')
('B', '1')
('B', '2')
```

Vector Data

- itertools 모듈
  - ✓ 사용

```
('A', 'A', 'A')
('A', 'A', 'B')
('A', 'B', 'A')
('A', 'B', 'B')
('B', 'A', 'A')
('B', 'A', 'B')
('B', 'B', 'A')
('B', 'B', 'B')
```