<부록 02. 파이썬 문법 정리>

Python 이란

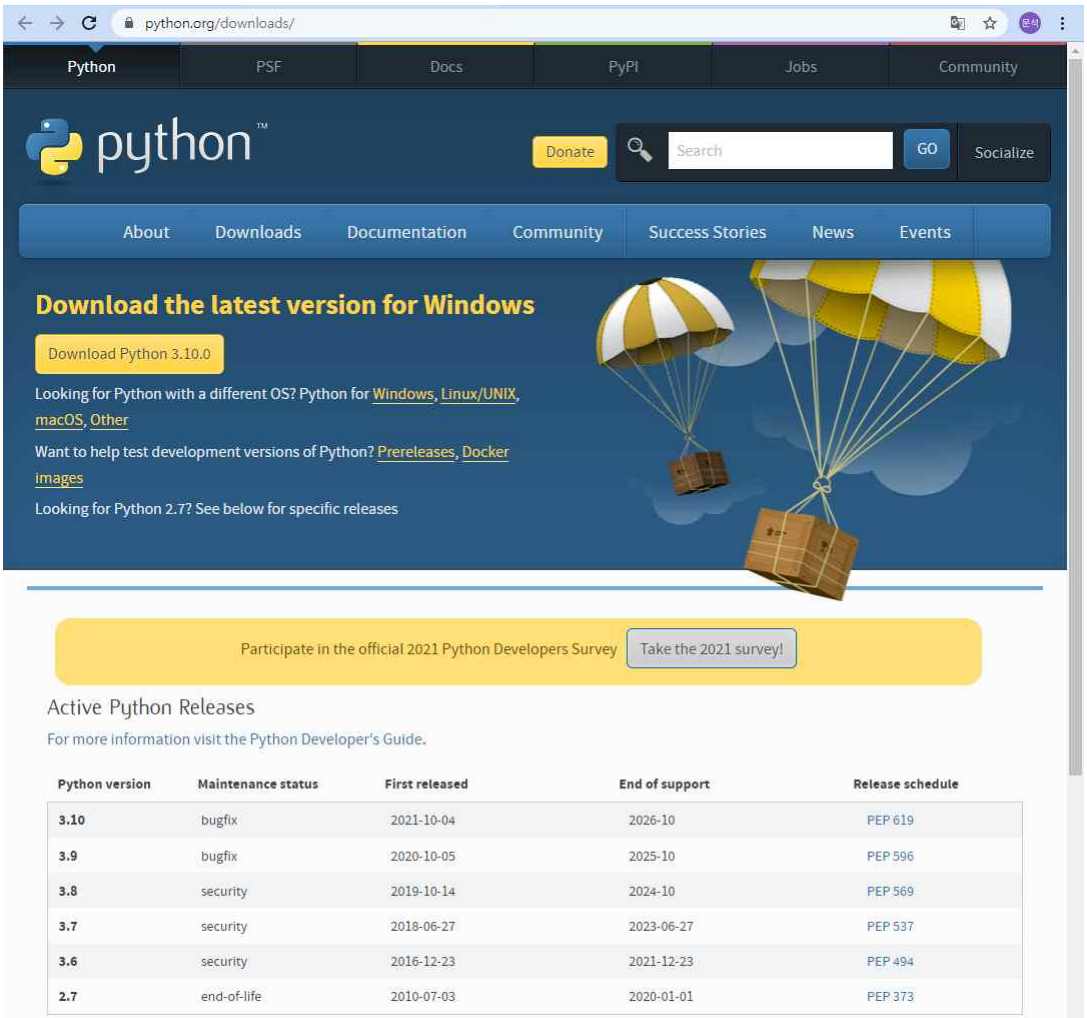
효율적이고 생산성이 높으며 객체 지향 프로그래밍(OOP)입니다.

Compile, Link가 필요가 없으며 자료형을 정의할 필요가 없어 간편하고 유연한 작업이 가능합니다.

다양한 데이터형과 연산이 가능하며 C언어와 결합하여 사용이 가능합니다.

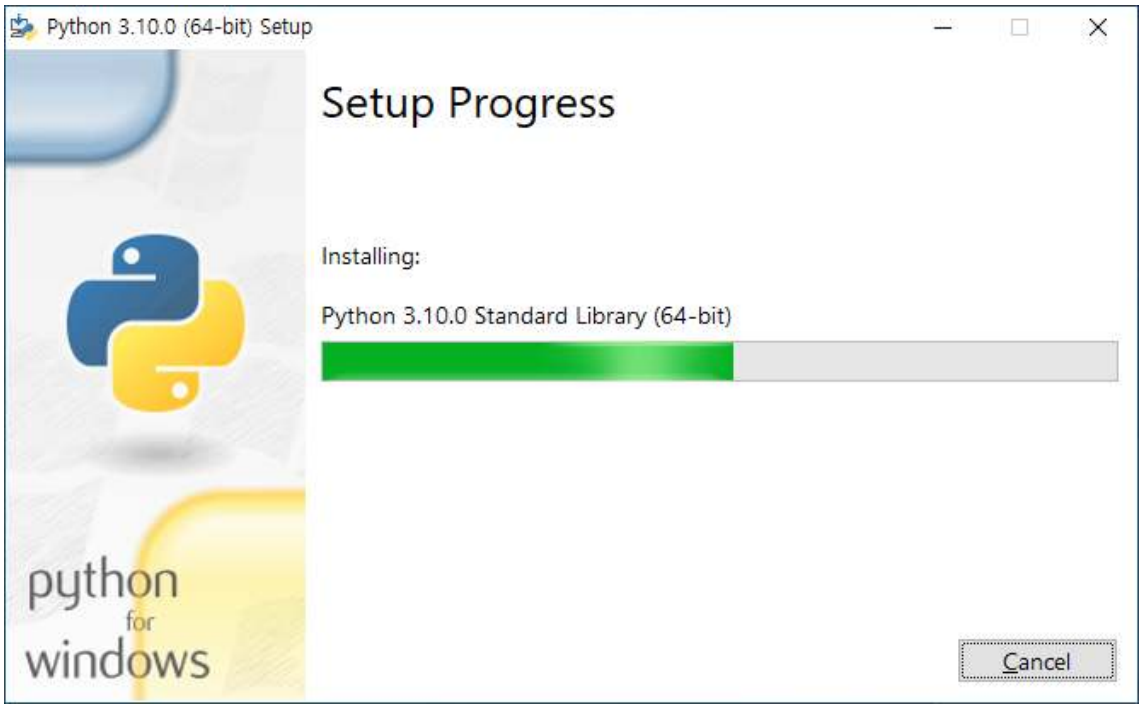
현재 강좌는 Python 3.10.0를 기반으로 진행하겠습니다.

Python 설치 : https://www.python.org/downloads/
Python Software Foundation의 Python 3.10.0에서 무료로 다운받으실 수 있습니다.

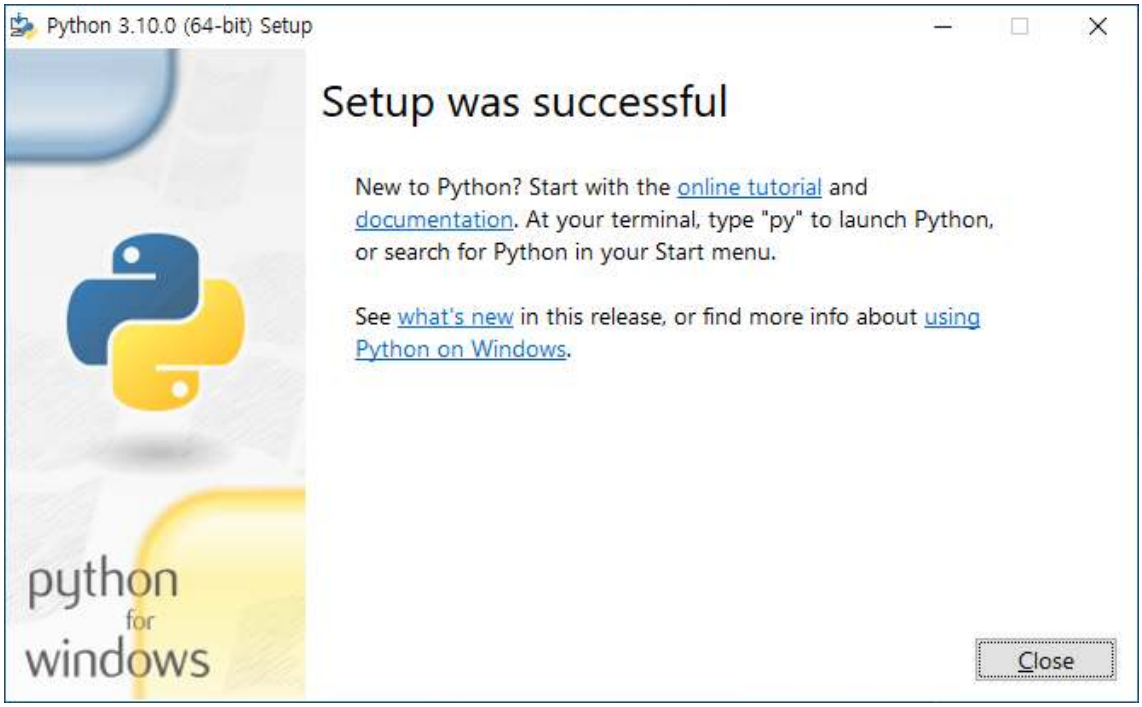


Download Python 3.10.0을 클릭하여 설치합니다.

Install Now 선택하여 설치 시작



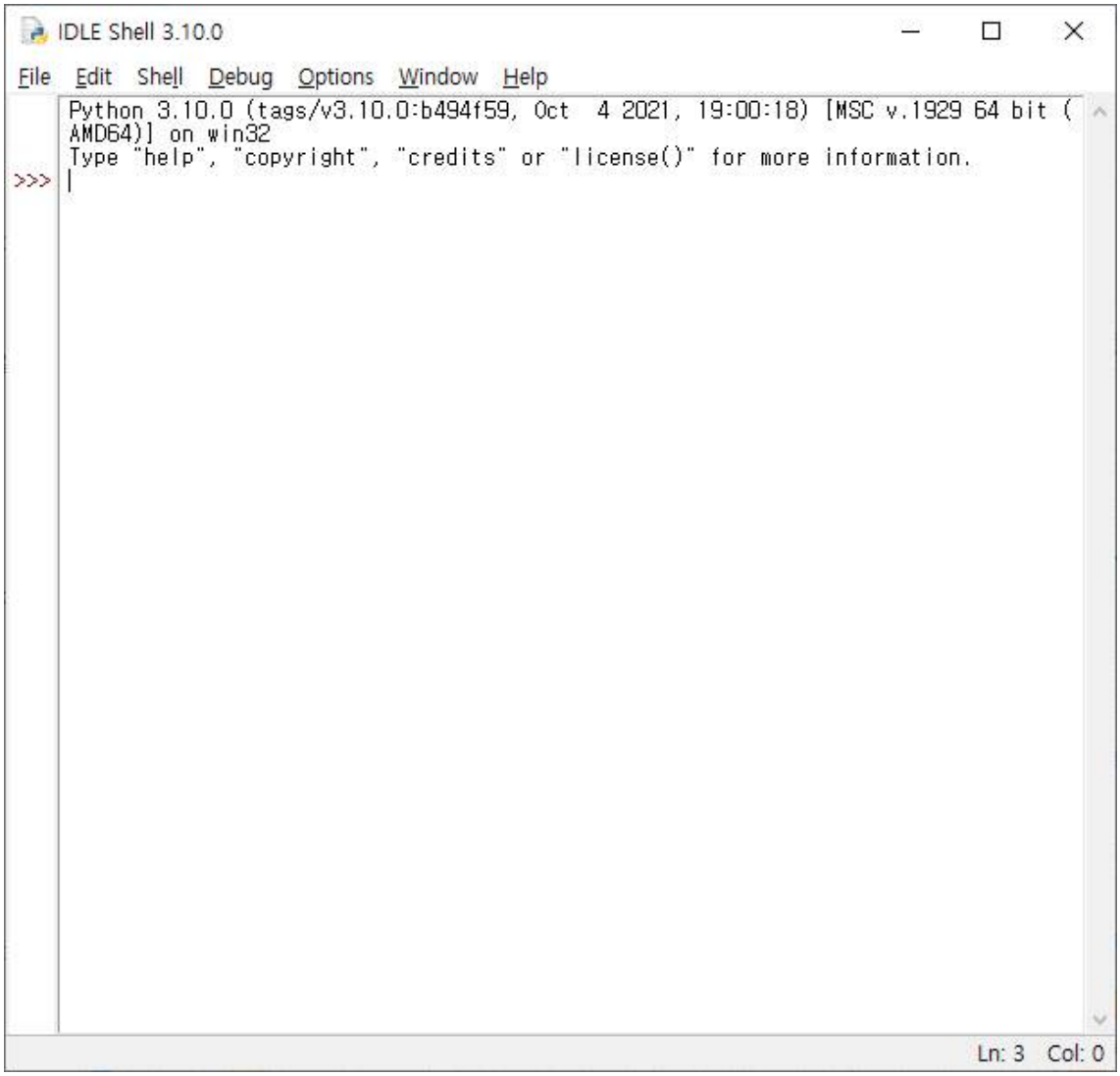
지정된 경로에 자동 설치


Python 3.10.0 (64-bit) Setup
Setup was successful
New to Python? Start with the online tutorial and documentation. At your terminal, type "py" to launch Python, or search for Python in your Start menu.
See what's new in this release, or find more info about using Python on Windows.
python for windows
Close

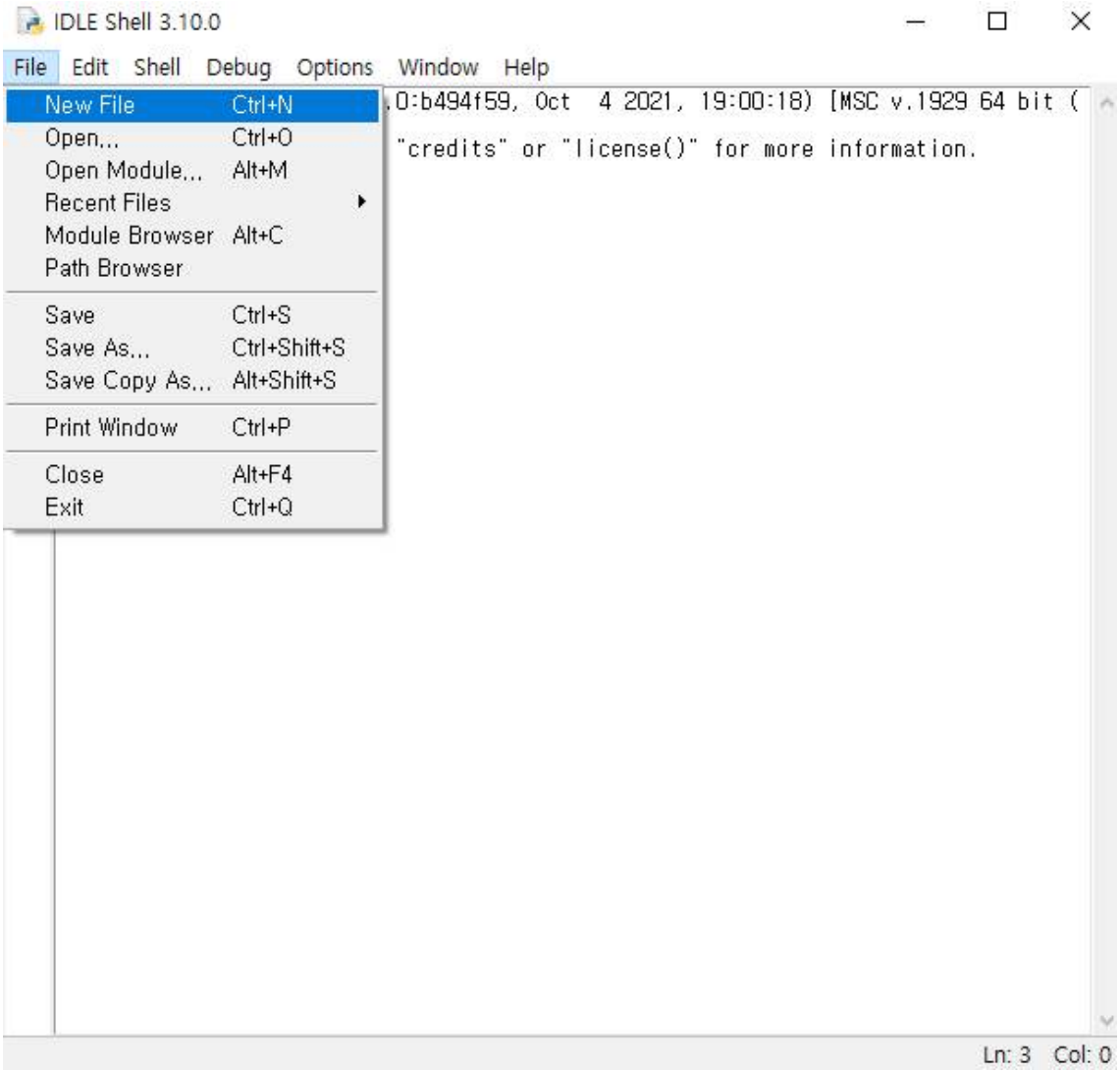
설치 완료


Python 3.10
IDLE (Python 3.10 64-bit)
Python 3.10 (64-bit)
Python 3.10 Manuals (64-bit)
Python 3.10 Module Docs (64-bit)

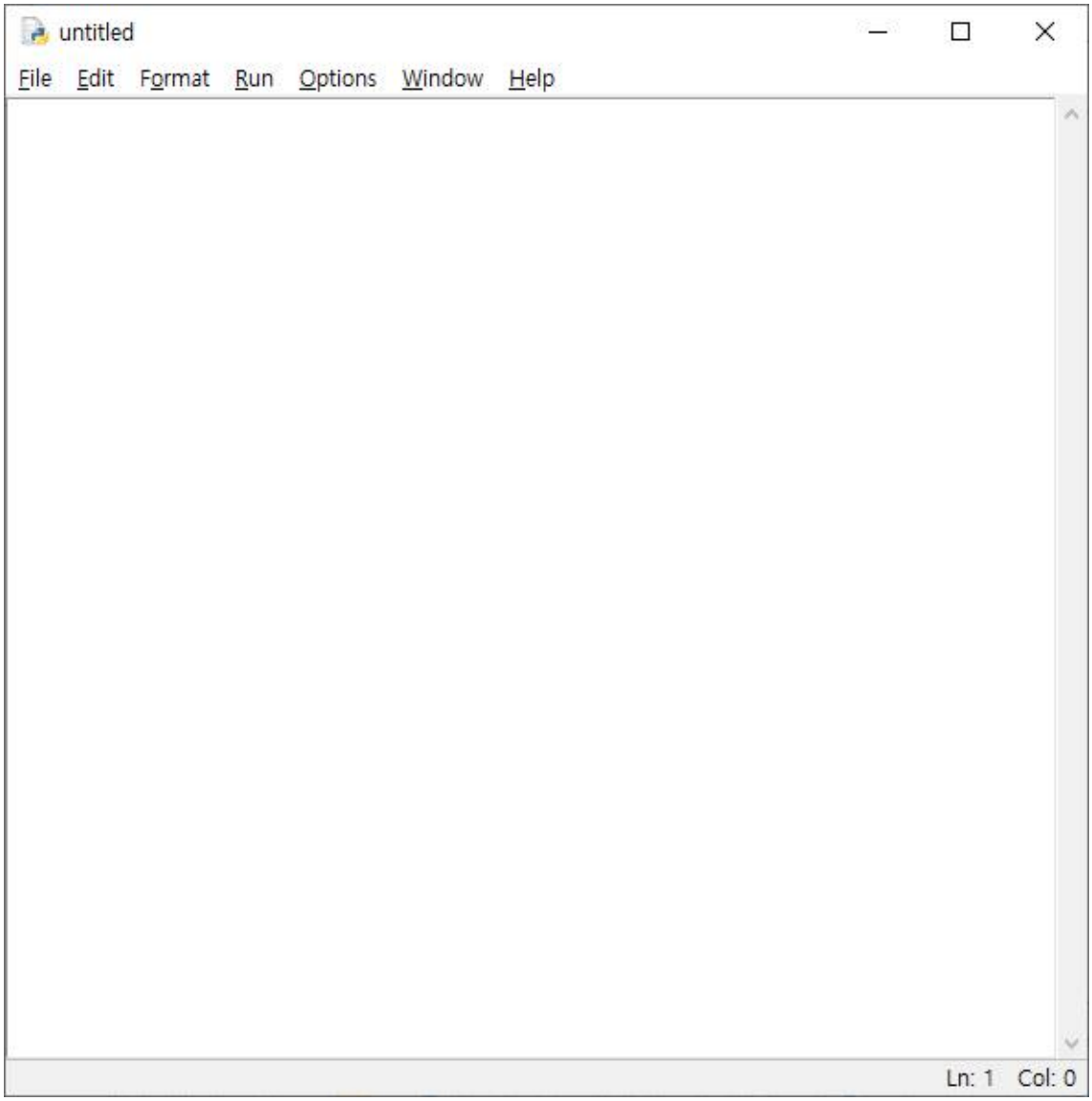
IDLE(Python 3.10-64bit) 실행



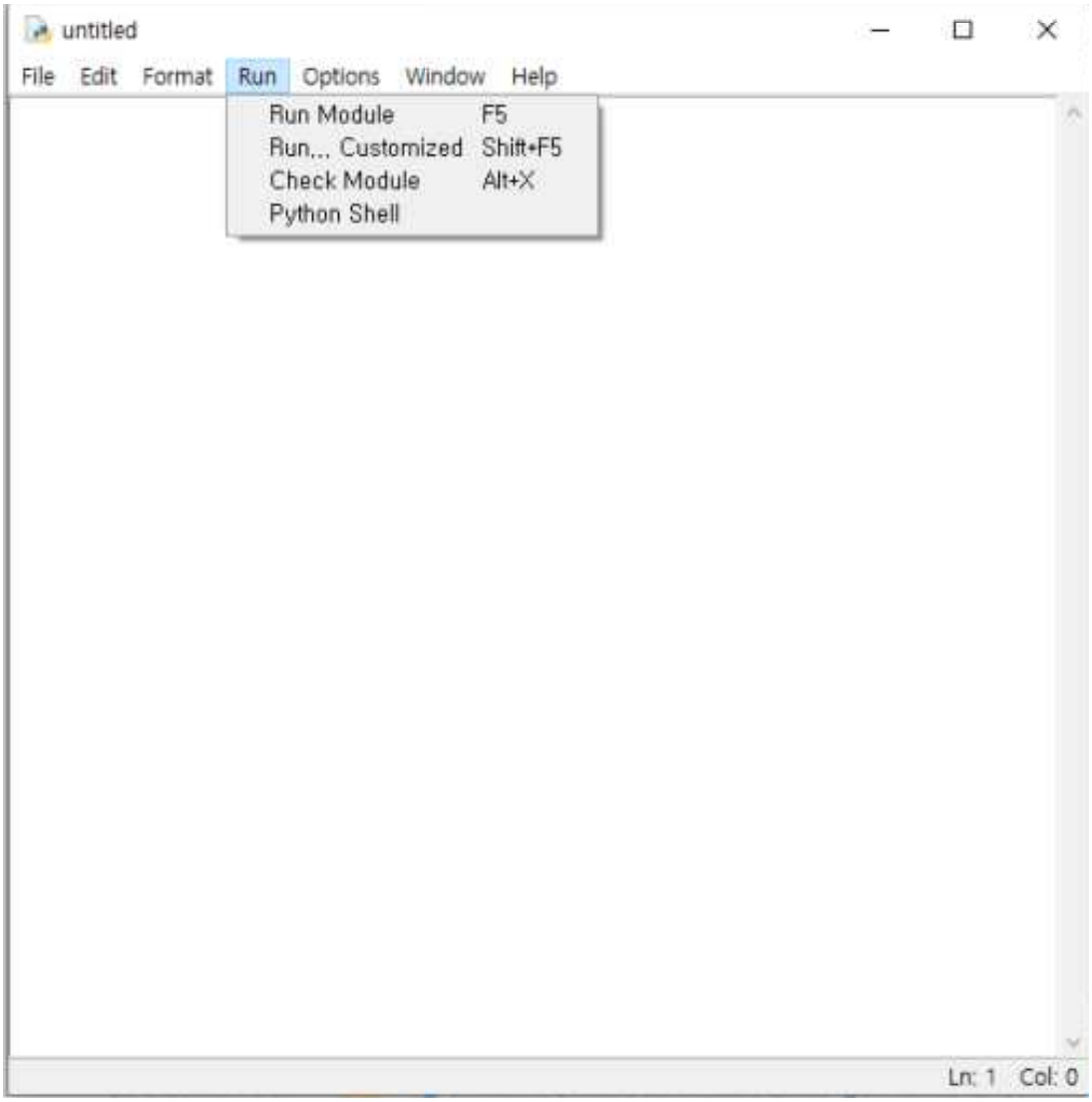
프로젝트 생성



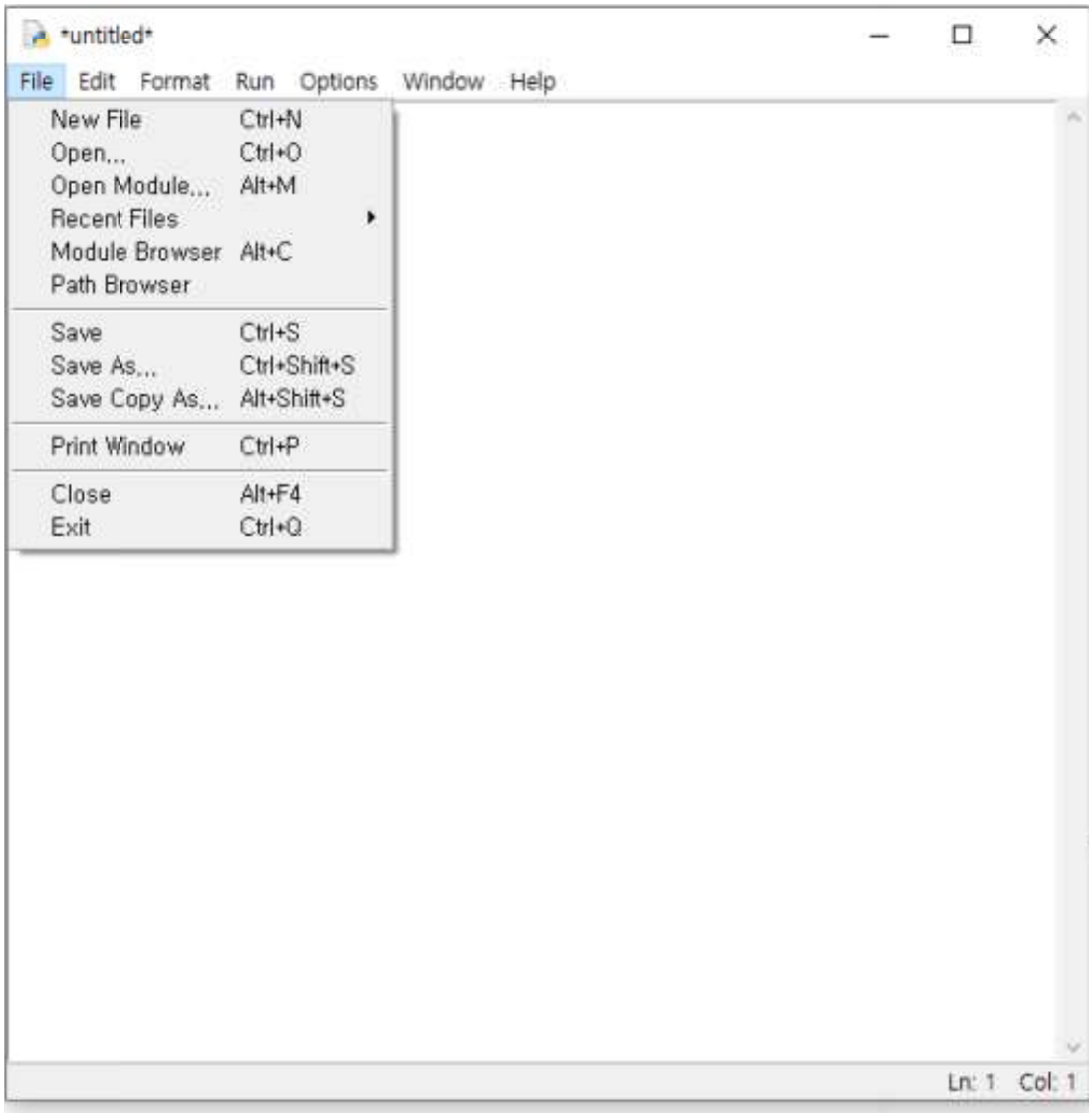
File->New File 실행



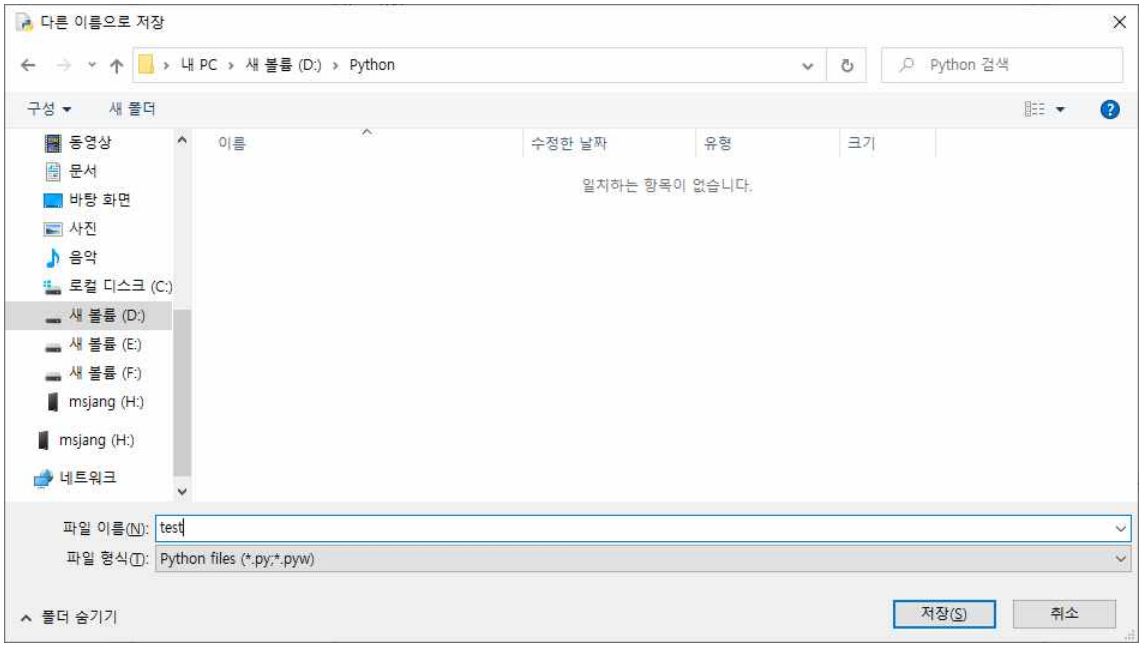
Untitiled 창이 생성되고 코드 구현 가능

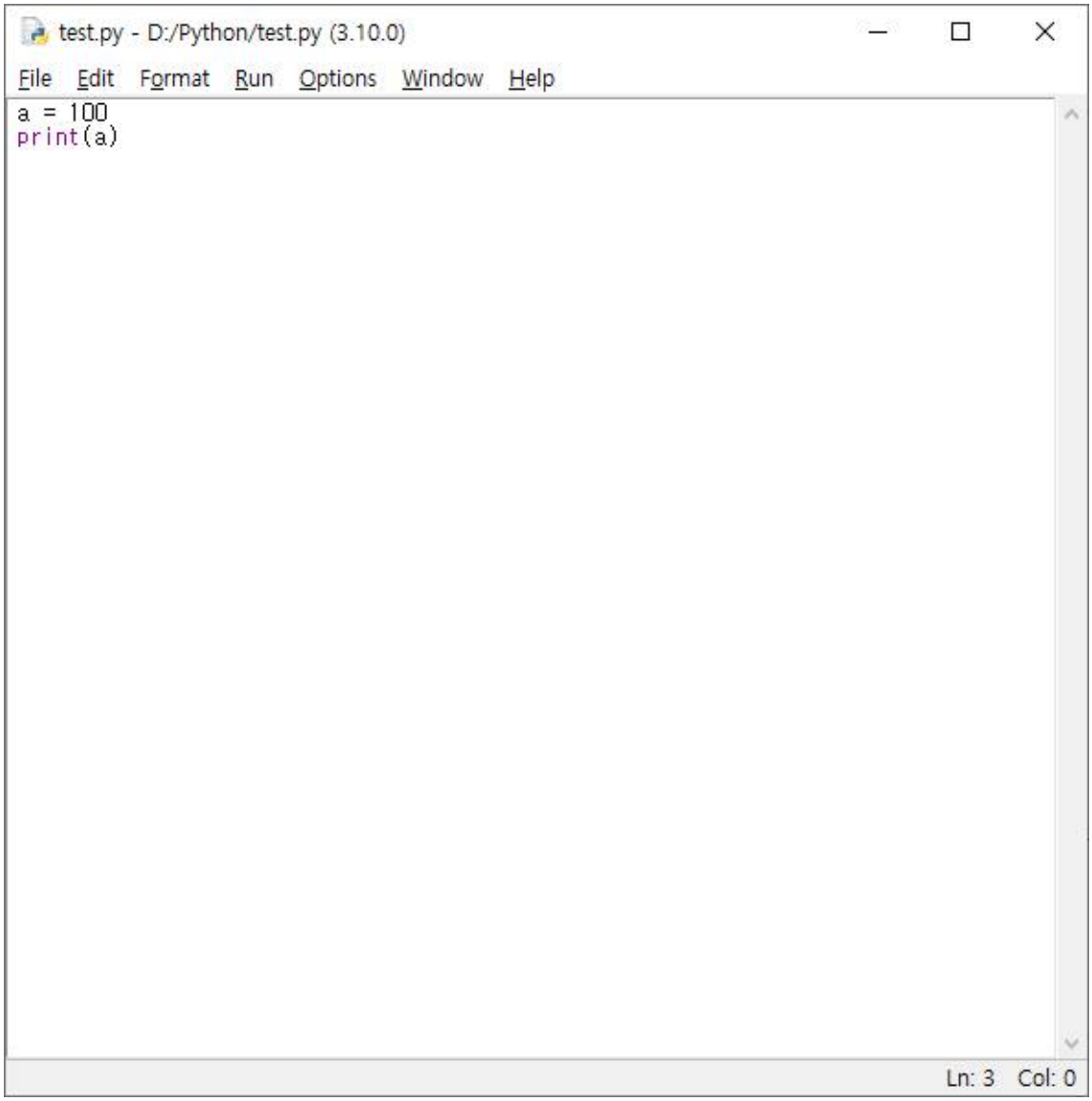


Run->Run Module로 프로젝트 실행

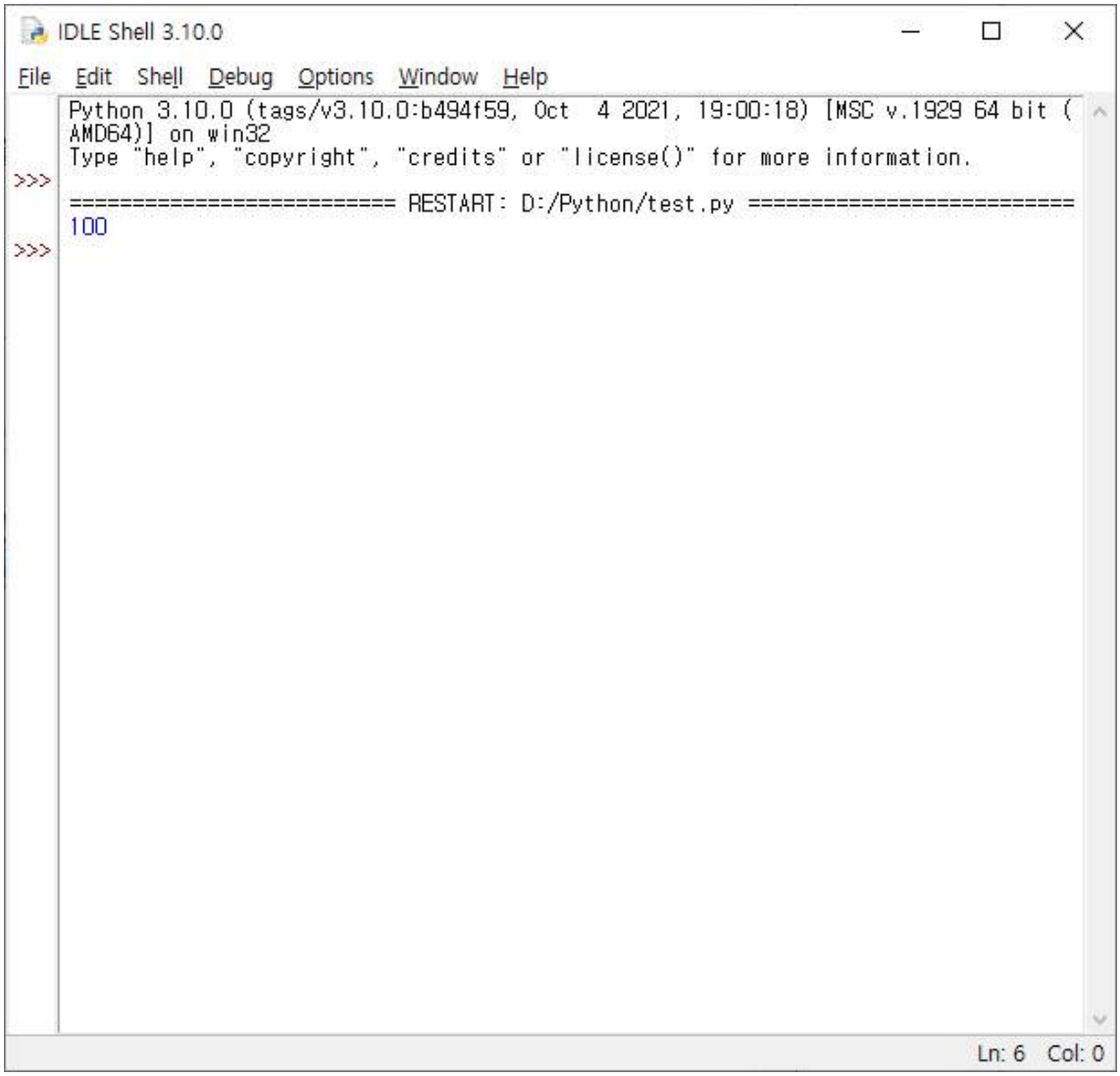


File->Save로 파일 저장

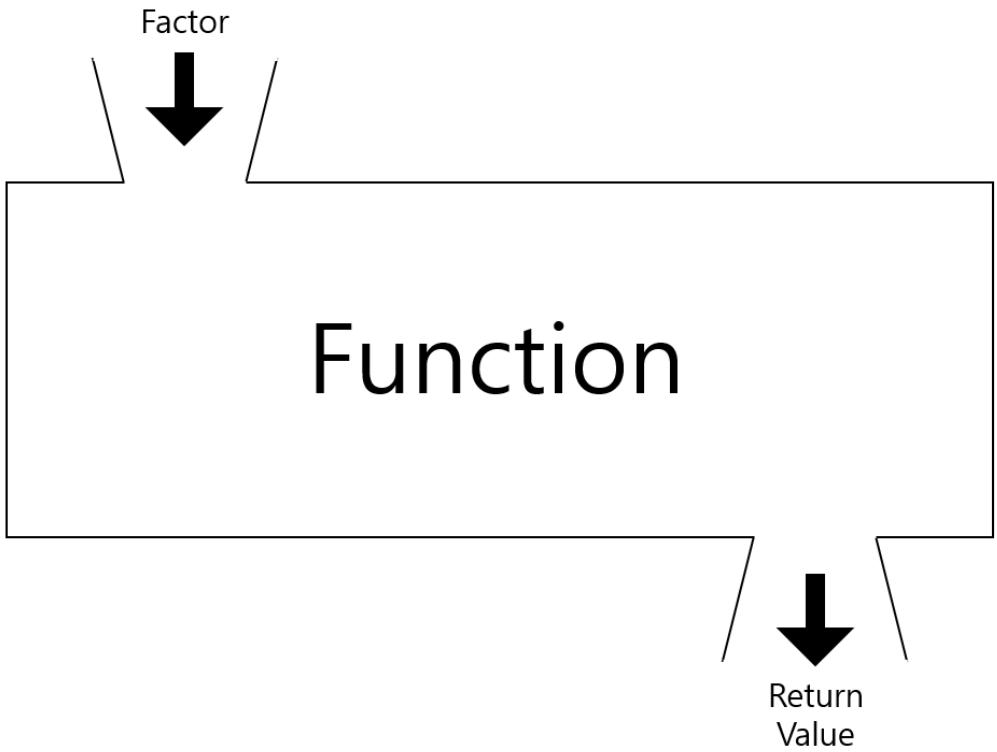

다른 이름으로 저장
내 PC › 새 볼륨 (D:) › Python
Python 검색
구성
새 폴더
동영상
문서
바탕 화면
사진
음악
로컬 디스크 (C:)
새 볼륨 (D:)
새 볼륨 (E:)
새 볼륨 (F:)
msjang (H:)
msjang (H:)
네트워크
이름
수정한 날짜
유형
크기
일치하는 항목이 없습니다.
파일 이름(N): test
파일 형식(T): Python files (*.py;*.pyw)
폴더 숨기기
저장(S)
취소



Run -> Run Module 실행


IDLE Shell 3.10.0
File Edit Shell Debug Options Window Help
Python 3.10.0 (tags/v3.10.0:b494f59, Oct 4 2021, 19:00:18) [MSC v.1929 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: D:/Python/test.py ==========
100
>>>
Ln: 6 Col: 0

함수의 이해



함수란 입력한 데이터를 받아 일련의 과정을 거쳐 결과를 반환하는 것을 의미합니다.

입력 데이터를 인수(Factor), 처리하는 동작을 함수(Function), 처리한 데이터를 반환값(Return Value)라 합니다.

인수가 존재하지 않을 수도 있으며, 물론 반환값이 없는 경우도 있습니다.

$$y = x + 1$$

위와 같은 함수가 있다하면 인수는 x, 함수는 x+1, 결과값은 y가 됩니다. 인수가 1일 경우 함수를 통해서 결과값은 2가 됩니다.

자료형

	자료형	저장 모델	변경 가능성	접근 방법
수치형	int, float, complex	Literal	Immutable	Direct
문자열	str	Container		Sequence
튜플	tuple			
리스트	list		Mutable	
사전	dict			Mapping
집합	set			Set

Python은 C/C++과는 다르게 존재하지 않는 자료형이 있습니다.

대표적인 예로 double이 있습니다.

또한, 다양한 데이터를 다루기 위한 데이터형이 존재합니다.

수치형, 문자열, 튜플, 리스트, 사전, 집합으로 분류할 수 있습니다.

저장 모델
  Literal : 단일 종류
  Container : 종류에 무관
변경 가능성
  Immutable : 변경 불가
  Mutable : 변경 가능
접근 방법
  Direct : 직접 할당
  Sequence : 순서 중시
  Mapping : 순서 무관
  Set : 중복 불가

튜플, 리스트, 사전, 집합은 C/C++ 등 의 배열과 비슷하나 약간 식 다른 차이를 가집니다.

튜플과 리스트의 경우 둘 다 다양한 종류의 데이터를 저장하고 순서를 중요시합니다.

하지만, 튜플의 경우 최초에 입력한 데이터를 변경이 불가하며 리스트의 경우 최초에 입력한 데이터의 변경이 가능합니다.

사전은 순서에 상관 없지만 중복이 가능한 대신, 집합은 중복이 불가능합니다.

대입 연산

```
A = B = C = 3
a, b = 1, 2
C += 1

print(A, B, C, a, b)
```

결과
3 3 4 1 2

Python에서는 세미콜론(;)을 입력하지 않아도 연산이 가능합니다. 또한 대소문자를 구별합니다.

print()를 이용하여 출력할 수 있습니다. 콤마(,)로 변수를 구별합니다.

사칙 연산

연산	결과
x + y	더하기
x - y	빼기
x * y	곱하기
x / y	나누기
x // y	정수 나누기
x % y	나머지
-x	부호 변경
x ** y	제곱

나누기와 정수 나누기의 차이점은 7/3일 경우 결과는 2.333333333333335가 출력되며 7//3일 경우 결과는 2가 출력됩니다.

실수 표현

```
A = float(3)
B = int(3.2)
C = 3.5

print(A, B, C)
```

결과

```
3.0 3 3.5
```

실수 표현은 명시적으로 소수점을 표기하거나 float()을 통해 표현할 수 있습니다.

반대로, 실수여도 int()를 통해 표현할 경우 정수로 표현됩니다.

int()나 float() 등을 통해 데이터 형식을 변경할 수 있습니다.

복소수 표현

연산	결과
x + yj	복소수
complex(x, y)	복소수
a.conjugate()	a의 공액복소수

복소수는 j를 포함하면 복소수로 입력됩니다. 허수 부분에 j를 붙여줍니다.

complex(x, y)는 x + yj로 표현됩니다.

공액복소수는 허수의 부호를 변경합니다.

a.conjugate()는 a에 영향을 미치지 않습니다. 즉, a의 값 자체가 바뀌지는 않습니다.

```
a = complex(3, 2)
b = a.conjugate()
c = 3 + 2j

print(a)
print(b)
print(c)
```

결과

```
(3+2j)
(3-2j)
(3+2j)
```

기본 함수

함수	결과
int(x)	int 형변환
float(x)	float 형변환
pow(x, y)	x의 y승
divmod(x, y)	(몫, 나머지)
abs(x)	절대값
max(x, y, z, …)	최대값
min(x, y, z, …)	최소값
round(x)	정수 반올림
round(x, n)	n번째 소수점 이하 반올림

pow(x, y)는 x**y와 동일합니다.

divmod(x, y)는 (x//y, x%y)와 동일합니다.

abs(x)는 x의 값이 복소수여도 절대값으로 반환합니다.

비교 연산

연산	의미
x >y	크다
x >= y	크거나 같다
x <y	작다
x <= y	작거나 같다
x == y	같다
x != y	같지 않다

비교 연산은 결과값이 True 또는 False로 반환됩니다.

연산이 참일 경우 True, 거짓일 경우 False로 반환합니다.

```
a = int(False)
b = float(True)

print(a)
print(b)
```

결과

```
0
1.0
```

True와 False의 상수 값은 각각 1과 0을 의미합니다.

논리 연산

연산	의미
x or y	논리합
x and y	논리곱
not x	부정

논리 연산은 결과값이 True 또는 False로 반환됩니다.

비교 연산을 혼합하여 논리 연산의 사용이 가능합니다.

```
x = True
a = (0>7) or (6>9)
b = (2>3) and (1>0)
c = not (1>0)
d = (4>3) and x

print(a)
print(b)
print(c)
print(d)
```

결과

```
False
False
False
True
```

논리합의 경우 두 조건 중 하나만 참이라면 True를 반환합니다.

논리곱의 경우 두 조건 모두 참이여야 True를 반환합니다.

부정의 경우 결과값을 반전시킵니다.

논리 연산 자체에 True와 False를 직접 입력하여 사용 할 수 있습니다.

논리 연산의 주의사항

```
a = (7>3) or 0
b = (7<3) or 1
c = (7>3) and 2
d = (7<3) and 3

print(a)
print(b)
print(c)
print(d)
```

결과

```
True
1
2
False
```

논리 연산의 경우 두 가지의 조건에서 앞의 조건을 확인 한 후, 뒤의 조건을 확인합니다.

이때 논리합(or)의 경우 앞의 조건이 참이면 뒤의 조건을 비교하지 않고 True를 반환합니다.

만약 앞의 조건이 거짓일 경우 뒤의 조건을 검사하게 되는데 조건이 아닌 상수일 경우 상수를 대입해버립니다.

논리곱(and)의 경우 논리합(or)과 마찬가지로 앞의 조건을 검사 후, 뒤의 조건을 검사합니다.

이 역시 앞의 조건에서 False가 반환되지 않으면 뒤의 조건에서 상수를 그대로 대입합니다.

비트 연산(Bitwise)

연산	의미
x | y	or
x &y	and
x ^ y	xor
x « y	left shift
x » y	right shift
~x	not

비트 연산을 통하여 2진법으로 구성된 값을 계산할 수 있습니다.

각 자릿수를 서로 비교하여 결과를 반환합니다.

- or : 둘 중 하나의 값이 1일 경우 1을 반환
- and : 둘 다 값이 1일 경우 1을 반환
- xor : 둘 다 값이 다를 경우 1을 반환
- left shift : 좌측으로 y회 비트 밀기
- right shift : 우측으로 y회 비트 밀기
- not : 반전

```
x = 0b0110
y = 0b1010

print(bin(x | y), x | y)
print(bin(x & y), x & y)
print(bin(x ^ y), x ^ y)
print(bin(x << 1), x << 1)
print(bin(y >> 1), y >> 1)
print(bin(~x), ~x)
```

결과

```
0b1110 14
0b10 2
0b1100 12
0b1100 12
0b101 5
-0b111 -7
```

2진법은 0b를 포함하여 0과 1로 구성된 진법입니다. 맨 우측 첫 자릿수부터 2^n-1을 의미합니다. (n=1)

0b0110의 경우 0 / 1 / 1 / 0 이므로 2^2 + 2^1입니다. 즉, 0b0110=6을 의미합니다.

0b1010의 경우 1 / 0 / 1 / 0 이므로 2^3 + 2^1입니다. 즉, 0b1010=10을 의미합니다.

or, and, xor은 각 자릿수들을 비교하여 반환하며 결과에서 확인 가능합니다.

shift는 y회 만큼 방향으로 이동하며 결과에서 확인 가능합니다.

not의 경우 0 → 1로 1 → 0으로 반전시킵니다. 하지만 여기서 -6이 아닌 -7이 나오는데, 2의 보수 표현법을 사용하기 때문입니다.

Tip : 2의 보수란 반전 시킨 값에 +1을 더해주어 음수를 표현하기 위해 사용합니다. 맨 앞에 -을 표현하여 반전을 대체합니다.

함수 사용

```
L1 = [True, True, False]

print(L1)
print(all(L1))
print(any(L1))

L2 = [1, 1, 0]

print(L2)
print(all(L2))
print(any(L2))

L3 = [3 > 0, 3 > 2, 2 == 2]

print(L3)
print(all(L3))
print(any(L3))
```

결과

```
[True, True, False]
False
True
[1, 1, 0]
False
True
[True, True, True]
True
True
```

all(목록)을 사용하여 목록 내의 모든 원소가 참일 경우 True를 반환합니다.
any(목록)을 사용하여 목록 내의 원소 중 하나라도 참일 경우 True를 반환합니다.

Tip : 논리 형식이나 상수형, 조건식 등 True나 False로 반환되는 값은 사용이 가능합니다.

```
numb = 33.2
```

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

print(isinstance(numb, int))
print(isinstance(numb, float))
print(isinstance(L, list))
print(isinstance(L[0], list))
```

결과

```
False
True
True
False
```

isinstance(객체, 클래스)를 사용하여 객체가 클래스의 객체인지를 확인합니다.

수학 모듈

Python에서는 수학 모듈을 이용하여 C/C++ 에서 while, for 등을 이용하여 구현해야 했던 함수들이 기본적으로 구현되어 있습니다.

수학 모듈을 사용한다면, 별도의 함수를 구성하지 않아도 수학적인 계산을 쉽게 해결할 수 있습니다.

```
import math
```

상단에 import math를 사용하여 수학 모듈을 포함시킵니다. 수학 함수의 사용방법은 math.*을 이용하여 사용이 가능합니다.

```
from math import *
```

위와 같이 import시킬 시 수학 함수를 사용할 때 math.를 입력하지 않아도 사용이 가능합니다.

* 대신 함수를 직접 적는다면 해당 함수만 사용이 가능합니다.

```
from math import pow

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

pow()는 정상 출력이 되지만 sqrt()는 포함시키지 않아 에러가 발생합니다.

표현 함수

연산	의미
ceil(x)	올림
floor(x)	내림
trunc(x)	절사

삼각 함수

연산	의미
cos(x)	코사인
sin(x)	사인
tan(x)	탄젠트
acos(x)	아크코사인
asin(x)	아크사인
atan(x)	아크탄젠트
atan2(x, y)	x/y 아크탄젠트

Tip : 라디안값으로 반환합니다.

하이퍼볼릭 함수

연산	의미
cosh(x)	하이퍼볼릭 코사인
sinh(x)	하이퍼볼릭 사인
tanh(x)	하이퍼볼릭 탄젠트
acosh(x)	하이퍼볼릭 아크코사인
asinh(x)	하이퍼볼릭 아크사인
atanh(x)	하이퍼볼릭 아크탄젠트

Tip : 라디안값으로 반환합니다.

각도 변환

연산	의미
degrees(x)	60분법으로 변환
radians(x)	호도법으로 변환

논리 함수

연산	의미
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

Tip : isclose(x, y, rel_tol=z) 에서 rel_tol=z를 미입력시 기본값은 1e-09로 계산합니다. 두 값의 차이가 5% 이내라면 z=0.05를 사용합니다.

로그 함수

연산	의미
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

Tip : log(x, y)에서 y를 미입력 시 밑을 e로 사용하여 자연로그로 이용합니다.

연산 함수

연산	의미
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)	감마함수
lgamma(x)	감마함수의 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 최대공약수
hypot(x, y)	유클리드 놈을 반환
modf(x)	x를 (소수부, 정수부)로 반환
copysign(x, y)	y의 부호를 사용하는 x를 반환

상수

연산	의미
e	e
pi	π
tau	τ
inf	∞
nan	Not a Number

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

$$frexp(x) = 가수부 \times 2^{지수부}$$

gamma(x) : 감마함수

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

hypot(x, y) : 유클리드 노름

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

리스트(List)

Python에서는 List를 이용하여 다양한 연산이 가능합니다.

데이터 형식과는 무관하게 저장할 수 있으며 List안에 또 다른 List를 포함시킬 수 있습니다.

List는 대괄호([])를 사용하며 순서를 중요시합니다. 또한 연산시 원소에 대한 값이 아닌 List 자체에 대한 연산을 실시합니다.

즉, 목록 자체에 대한 연산이므로 내부 요소가 아닌 목록에 대해 영향을 미칩니다.

생성

```
a = [1, 2, 3]
b = [4, 5, 6]

print(a)
print(b)
```

결과

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

리스트는 대괄호([])와 콤마(,)를 이용하여 생성이 가능합니다.

이어 붙이기

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
d = a + [9]

print(c)
print(d)
```

결과

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 9]
```

리스트는 +를 이용하여 리스트끼리 합치거나 새로운 값을 추가할 수 있습니다.

반복

```
a = [1, 2, 3]
b = [4, 5, 6]
```

```
print(a * 2)
print(b * 3)
```

결과

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

리스트는 *를 이용하여 리스트를 반복시켜 리스트 자체의 크기가 커집니다.

참조

```
a = [1, 2, 3]
b = [4, 5, 6]
c = [1, 2, 3, 4, 5, 6, 7, 8, 9]

print(a[0])
print(a[1])
print(a[2])
print(a[-1])
print(b[0:1])
print(b[0:-1])
print(c[0:-1:2])
```

결과

```
1
2
3
3
[4]
[4, 5]
[1, 3, 5, 7]
```

리스트는 :을 이용하여 리스트를 참조할 수 있습니다.

리스트[index]를 사용하면 index의 값을 출력합니다.

리스트[start:end]를 사용하면 start부터 end값 까지 출력합니다.

리스트[start:end:interval]를 사용하면 start부터 end값 까지 interval 간격만큼 출력합니다.

좌측의 첫번째 값은 0이며, 우측의 첫번째 값은 -1을 사용하여 출력이 가능합니다.

포함

```
a = [1, 2, 3]
b = [4, 5, 6]
c = [a, b]

print(c)
```

결과

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

리스트안에 리스트를 포함시켜 리스트를 포함한 리스트 또한 생성이 가능합니다.

리스트(List)

Python에서는 List를 이용하여 다양한 연산이 가능합니다.

데이터 형식과는 무관하게 저장할 수 있으며 List안에 또 다른 List를 포함시킬 수 있습니다.

List는 대괄호([])를 사용하며 순서를 중요시합니다. 또한 연산시 원소에 대한 값이 아닌 List 자체에 대한 연산을 실시합니다.

즉, 목록 자체에 대한 연산이므로 내부 요소가 아닌 목록에 대해 영향을 미칩니다.

조사

```
a = [1, 1, 2, 3, 5, 8, 13]
b = ["a", "a", "b", "c", "xyz", [1, 2, 3]]

print(len(a))
print(min(a))
print(max(a))
print(b.index([1, 2, 3]))
print(b.count("a"))
print("a" in b)
```

결과

```
7
```

```
1
13
5
2
True
```

len()을 이용하여 리스트의 길이를 확인할 수 있습니다.

min()을 이용하여 리스트에서 최솟값을 가지는 원소의 값을 확인할 수 있습니다.

max()를 이용하여 리스트에서 최댓값을 가지는 원소의 값을 확인할 수 있습니다.

리스트.index()를 이용하여 해당 값이 가지는 위치를 확인 할 수 있습니다. (시작값=0)

리스트.count()를 이용하여 해당 값이 가지는 개수를 확인할 수 있습니다.

값 in 리스트를 이용하여 해당 값이 존재하는지 확인할 수 있습니다.

Tip : 문자열이나 리스트가 포함된 리스트에는 max(), min()을 이용할 수 없습니다.

대입

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
a[1] = 2

print(a)
```

결과

```
[0, 2, 2, 3, 4, 5, 6, 7, 8, 9]
```

리스트의 값을 직접적으로 변경시켜 값을 변경할 수 있습니다.

삽입

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
a.insert(0, -1)
a.insert(-1, 10)

print(a)
```

결과

```
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 9]
```

리스트.insert(index, value)를 이용하여 index의 바로 앞자리에 value를 삽입합니다.

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
a.append(10)
print(a)
```
결과
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

리스트.append(value)를 이용하여 리스트의 마지막 자리에 value를 삽입합니다.

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
b = [10, 11]
a.extend(b)
print(a)
```
결과
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

리스트.extend(list)를 이용하여 리스트의 마지막 자리에 list의 원소들을 삽입합니다.

삭제

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
del a[1:-1]
print(a)
```
결과
```
[0, 9]
```

del 리스트[start, end]를 이용하여 start부터 end-1까지의 원소를 삭제합니다.

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
a.pop(1)
print(a)
```
결과
```
[0, 2, 3, 4, 5, 6, 7, 8, 9]
```

리스트.pop(index)를 이용하여 index에 위치한 원소를 삭제합니다.

Tip : index를 생략할 경우 index의 값을 -1로 간주합니다.

```
a = ["x", "y", "z", "y"]
a.remove("y")
print(a)
```
결과

['x', 'z', 'y']

리스트.remove(value)를 이용하여 value와 동일한 값을 지니는 원소 하나를 삭제합니다.
Tip : index가 0에 가까운 순서부터 삭제합니다.

자르기

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
a = a[:4]
print(a)
print(a[1:])
print(a)
```

결과

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

a[start:end]를 이용하여 리스트를 자를 수 있습니다. start와 end를 미입력시 각각 첫 번째 index와 마지막 index로 간주합니다.

Tip : 대입 연산을 하지 않으면 원본 리스트에는 영향을 주지 않습니다.

정렬

```
a = [1, 0, 2, 4, 3]
a.sort()
print(a)
```

결과

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

리스트.sort()를 이용하여 리스트를 오름차순으로 정렬합니다.

```
a = [1, 0, 2, 4, 3]
a.sort(reverse=True)
print(a)
```

결과

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

리스트.sort(reverse=True)를 이용하여 리스트를 내림차순으로 정렬합니다.
Tip : reverse=False로 사용할 경우 오름차순으로 정렬합니다.

```
a = [1, 0, 2, 4, 3]
a.reverse()
print(a)
```

결과

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

리스트.reverse()를 이용하여 리스트를 역순으로 정렬합니다.

```
a = [1, 0, 2, 4, 3]
a = sorted(a, reverse=True)
print(a)
```

결과

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

sorted(리스트, reverse=bool)를 이용하여 리스트를 내림차순으로 정렬합니다.

sort()와 sorted()는 key 매개변수에 lambda를 이용해 더 세부적인 정렬을 할 수 있습니다.

문자열(String)

Python에서는 따옴표를 이용하여 문자열을 생성할 수 있습니다.

문자열의 문자들에 대한 값을 수정하거나 추가, 삭제 할 수 있습니다.

일반적으로 문자열은 리스트(List)로 간주하므로, List에서 사용되는 함수도 사용할 수 있습니다.

생성

```
a = 'Python 3.6.4'
b = "It's you"
c = 'It\'s you'
d = "alpha\nbeta"

print(a)
print(b)
print(c)
print(d)
```

결과

```
Python 3.6.4
It’s you
It’s you
alpha
beta
```

작은따옴표('')를 사용하거나 큰따옴표("")를 사용하여 문자열을 생성할 수 있습니다.

문자열 안에 따옴표를 포함시키려면 \을 병기하여 사용가능합니다.

줄바꿈은 \n을 사용하여 줄을 바꿀 수 있습니다.

Tip : \을 사용하는 이스케이프 코드는 \n을 비롯하여 수평 탭 : \t, \ 문자 : \\, 백스페이스 : \b 등이 존재합니다.

연산

```
a = "al"
b = "pha"
c = "alphabet"

print(a + b)
print(a * 2)
print(a[-1])
print(b[0:1])
print(c[0:-1:2])
```

결과

```
alpha
alal
l
p
apae
```

+를 이용하여 문자열끼리 합칠 수 있습니다.

*를 이용하여 문자열을 반복할 수 있습니다.

문자열[index]을 이용하여 index에 위치한 문자를 출력합니다.

문자열[start:end]을 이용하여 start부터 end까지의 문자열을 출력합니다.

문자열[start:end:step]을 이용하여 start부터 end까지 step간격의 문자열을 출력합니다.

포맷

```
a = "%d %f %e" % (10, 10, 10)
b = "%o %x" % (8, 16)
c = "%c %s" % ('A', "AAA")

print(a)
print(b)
print(c)
```

결과

```
10 10.000000 1.000000e+01
10 10
```

A AAA

"%포맷형식" % (변수 또는 값)의 형태로 포맷을 구성할 수 있습니다.

연산	의미
%d	정수
%f	실수
%e	지수
%o	8진법
%x	16진법
%c	문자
%s	문자열

Tip : %c는 상수를 입력할 경우 아스키값으로 판단하여 출력합니다.

변환

```
a = "abcd"
b = "ABCD"

print(a.upper())
print(a.lower())
```

결과

```
ABCD
abcd
```

문자열.upper()은 소문자를 대문자로 변경합니다.

문자열.lower()은 대문자를 소문자로 변경합니다.

공백 제거

```
a = "   l   aa   l   "

print(a.strip())
```

```
    print(a.rstrip())
    print(a.lstrip())
```
결과
```
    l aa l
          l aa l
    l aa l
```

문자열.strip()은 양측 공백을 제거합니다.

문자열.rstrip()은 우측 공백을 제거합니다.

문자열.lstrip()은 좌측 공백을 제거합니다.

검출

```
    a = "Time is an illusion."

    print(a.find('x'))
    print(a.index('u'))
    print(a.count('i'))
```
결과
```
    -1
    14
    4
```

문자열.find(x)은 해당 문자가 가장 처음에 나온 위치를 반환합니다.

문자열.index(x)은 해당 문자가 가장 처음에 나온 위치를 반환합니다.

문자열.count(x)은 해당 문자의 횟수를 반환합니다.

Tip : find와 index의 차이점은 find의 경우 찾지 못했을 경우 -1을 반환하며, index는 Error가 발생합니다.

변환

```
a = "Time is an illusion."
b = "/"

print(a.split())
print(a.replace("Time", "Today"))
print(b.join(a))
print(b.startswith('/'))
```

결과

```
['Time', 'is', 'an', 'illusion.']
Today is an illusion.
T/i/m/e/ /i/s/ /a/n/ /i/l/l/u/s/i/o/n/.
True
```

문자열.split(x)은 띄어쓰기마다 분리하여 리스트로 변환합니다.

문자열.relpace(x, y)은 해당 문자를 다른 문자로 변경합니다.

문자열.join(x)은 x 문자열 사이사이에 문자열을 삽입합니다.

문자열.startswith(x)은 문자열이 x로 시작하는지 확인합니다.

튜플(Tuple)

Python에서는 소괄호(())를 이용하여 튜플을 생성할 수 있습니다.

튜플은 정의에 사용되며 변경이 불가하고 순서를 중요시합니다.

주로, 변하지 않는 값이나 위치, 크기 등 한 쌍을 이루는 객체들을 표현할 때 사용합니다.

생성

```
a = (1, 2, 3, 4, 5)
b = (6,)
c = tuple("123")

print(a)
print(b)
print(c)
```

결과

```
(1, 2, 3, 4, 5)
(6,)
('1', '2', '3')
```

소괄호(())를 이용하여 List와 동일한 방법으로 튜플을 생성할 수 있습니다.

요소를 하나만 갖는 튜플을 생성할 경우, 요소 내에 콤마를 추가해 튜플로 표현할 수 있습니다.

또한, tuple()을 통해서도 다른 형식의 데이터를 튜플로 변경할 수 있습니다.

참조

```
a = (1, 2, 3, 4, 5)

print(a[0])
print(a[1])
print(a[2])
print(a[-1])
```

```
print(a[0:2])
print(a[0:-1])
print(a[0:-1:2])
```

결과

```
1
2
3
5
(1, 2)
(1, 2, 3, 4)
(1, 3)
```

튜플은 :을 이용하여 튜플을 참조할 수 있습니다.

튜플[index]를 사용하면 index의 값을 출력합니다.

튜플[start:end]를 사용하면 start부터 end값 까지 출력합니다.

튜플[start:end:interval]를 사용하면 start부터 end값 까지 interval 간격만큼 출력합니다.

첫번째 값은 0이며, 우측의 첫번째 값은 -1을 사용하여 출력이 가능합니다.

조사

```
a = (1, 2, 3, 4, 5)

print(len(a))
print(max(a))
print(min(a))
print(a.index(3))
print(a.count(1))
print(6 in a)
```

결과

```
5
5
1
2
1
```

False

len()을 이용하여 튜플의 길이를 확인할 수 있습니다.

min()을 이용하여 튜플에서 최솟값을 가지는 원소의 값을 확인할 수 있습니다.

max()를 이용하여 튜플에서 최댓값을 가지는 원소의 값을 확인할 수 있습니다.

튜플.index()를 이용하여 해당 값이 가지는 위치를 확인 할 수 있습니다. (시작값=0)

튜플.count()를 이용하여 해당 값이 가지는 개수를 확인할 수 있습니다.

값 in 튜플을 이용하여 해당 값이 존재하는지 확인할 수 있습니다.

사전(Dictionary)

Python에서는 중괄호({})를 이용하여 사전을 생성할 수 있습니다. 사전은 매칭에 사용되며 key와 value로 구성 되어있습니다.

key를 호출하여 value를 불러옵니다. key는 중복이 불가능하며 value는 중복이 가능합니다. 순서는 무관합니다.

데이터베이스처럼 key와 value로 구성되어 있고, value의 값으로 dict나 list를 사용할 수도 있습니다.

생성

```
a = {"one": "하나", "two": "둘", "three": "셋"}
b = dict.fromkeys(["one", "two", "three"], "알 수 없음")

print(a)
print(b)
```

결과

```
{'one': '하나', 'two': '둘', 'three': '셋'}
{'one': '알 수 없음', 'two': '알 수 없음', 'three': '알 수 없음'}
```

사전은 { key1:value1, key2:value2, ... }의 형태로 생성할 수 있습니다.

또는 dict.fromkeys(key, default)를 통하여 각각의 key에 default의 값을 지니는 사전을 생성할 수 있습니다.

접근

```
a = {"one": "하나", "two": "둘", "three": "셋"}

print(a["one"])
print(a.get("two"))
print(a.get("four", "넷"))
```

결과

```
하나
둘
넷
```

사전의 접근 방식은 중괄호([])에 index대신에 key를 호출하여 value를 출력합니다.

또는 사전.get(key)를 통해서 value를 호출할 수 있습니다.

사전.get(key, value)로 value를 추가한다면, 사전에 key값 존재하지 않는다면 value값을 대신 출력합니다.

추가

```
a = {"one": "하나", "two": "둘", "three": "셋"}

a["four"] = "넷"
# a.setdefault("four", "넷")
print(a["four"])

a.update({"five": "다섯", "six": "여섯"})
print(a)
```

결과

```
넷
{‘one’: ‘하나’, ‘two’: ‘둘’, ‘three’: ‘셋’, ‘four’: ‘넷’, ‘five’: ‘다섯’, ‘six’: ‘여섯’}
```

사전[key]=value를 이용하여 요소를 추가할 수 있습니다.

사전.setdefault(key, value)를 이용하여 요소를 추가할 수 있습니다.

사전.update({'key':'value', 'key':'value'})를 이용하여 다수의 요소를 추가할 수 있습니다.

Tip : setdefault()는 초깃값을 설정하는 함수이므로, 이미 key가 존재하는 경우 value는 수정되지 않습니다.

수정

```
a = {"one": "하나", "two": "둘", "three": "셋"}
```

```
a["three"] = "둘"
a.update(two="셋")

print(a)
```
결과
```
{'one': '하나', 'two': '셋', 'three': '둘'}
```

사전[key]=value를 이용하여 value 값을 수정 할 수 있습니다.

사전.update(key=value)를 사용하여 기존에 존재하는 key에 대한 value 값을 수정할 수 있습니다.

병합

```
a = {"one": "하나", "two": "둘", "three": "셋"}
b = {"five": "다섯", "six": "여섯"}

a.update(b)

print(a)
print(b)
```
결과
```
{'one': '하나', 'two': '둘', 'three': '셋', 'five': '다섯', 'six': '여섯'}
{'five': '다섯', 'six': '여섯'}
```

사전.update(사전)을 이용하여 서로 다른 사전을 병합할 수 있습니다.

삭제

```
a = {"one": "하나", "two": "둘", "three": "셋"}
del a["three"]
# a.pop("three")
# a.popitem()
print(a)
```

```
a.clear()
print(a)
```
결과
```
{‘one’: ‘하나’, ‘two’: ‘둘’}
{}
```

del 사전['key']을 이용하여 사전의 특정 요소를 삭제할 수 있습니다.

사전.pop('key')을 이용하여 del처럼 사전의 특정 요소를 삭제할 수 있습니다.

사전.popitem()을 이용하여 사전의 마지막 요소를 삭제할 수 있습니다.

사전.clear()을 이용하여 사전의 모든 값을 삭제합니다.

조사

```
a = {"one": "하나", "two": "둘", "three": "셋"}

print(a.items())
print(a.keys())
print(a.values())
print("one" in a)
print("four" not in a)
```
결과
```
dict_items([(‘one’, ‘하나’), (‘two’, ‘둘’), (‘three’, ‘셋’)])
dict_keys([‘one’, ‘two’, ‘three’])
dict_values([‘하나’, ‘둘’, ‘셋’])
True
True
```
사전.items()를 이용하여 key와 value를 List(목록)으로 감싸는 key, vaule의 튜플(Tuple)로 반환합니다.

사전.keys()을 이용하여 key의 모든 목록을 반환합니다.
사전.values()을 이용하여 value의 모든 목록을 반환합니다.

key in 사전을 이용하여 사전에 key 값이 있는지 확인합니다.

key not in 사전을 이용하여 사전에 key 값이 없는지 확인합니다.

집합(Set)

Python에서는 중괄호({})를 이용하여 집합을 생성할 수 있습니다.

집합은 연산에 사용되며 중복이 불가능하며, 순서는 무관합니다.

데이터를 집합으로 변경시킨 뒤, 집합 연산으로 간단하게 데이터를 정제할 수 있습니다.

생성

```
a = {1, 2, 3}
b = {1, 3, 5, 7, 9}

print(a)
print(b)
```

결과

```
{1, 2, 3}
{1, 3, 5, 7, 9}
```

중괄호({})를 사용하여 집합을 생성할 수 있습니다.

추가

```
a = {1, 2, 3}
b = {1, 3, 5, 7, 9}

a.add(4)
b.add(11)

print(a)
print(b)
```

결과

```
{1, 2, 3, 4}
{1, 3, 5, 7, 9, 11}
```

집합.add(x)를 이용하여 집합에 x값을 가지는 원소를 추가할 수 있습니다.

삭제

```
a = {1, 2, 3}
b = {1, 3, 5, 7, 9}

a.discard(3)
b.discard(7)

print(a)
print(b)
```

결과

```
{1, 2}
{1, 3, 5, 9}
```

집합.discard(x)를 이용하여 집합에서 x값을 가지는 원소를 삭제할 수 있습니다.

변환

```
L = [1, 2, 7, 2, 3]

a = set(L)

print(a)
```

결과

```
{1, 2, 3, 7}
```

set(x)을 이용하여 리스트 x를 집합으로 변환할 수 있습니다.

중복되는 값은 사라집니다.

합집합

```
a = {1, 2, 3}
```

```
b = {1, 3, 5, 7, 9}

c = a | b
d = a.union(b)

print(c)
print(d)
```

결과

```
{1, 2, 3, 5, 7, 9}
{1, 2, 3, 5, 7, 9}
```

| 기호를 사용하거나, 집합.union(집합)을 이용하여 합집합 연산을 할 수 있습니다.

교집합

```
a = {1, 2, 3}
b = {1, 3, 5, 7, 9}

c = a & b
d = a.intersection(b)

print(c)
print(d)
```

결과

```
{1, 3}
{1, 3}
```

& 기호를 사용하거나, 집합.intersection(집합)을 이용하여 교집합 연산을 할 수 있습니다.

차집합

```
a = {1, 2, 3}
b = {1, 3, 5, 7, 9}
```

```
c = a - b
d = b - a

print(c)
print(d)
```

결과

```
{2}
{9, 5, 7}
```

- 기호를 사용하여 차집합 연산을 할 수 있습니다.

조건문(if)

조건문(if)은 제어문 중 하나로, 알고리즘의 논리적 제어를 표현하는 수단입니다.

조건문을 통해 어떤 목적이나 상태가 만족할 때 실행되게 하거나, 특정 상태가 하나라도 만족할 때 등 알고리즘의 실행 흐름을 제어할 수 있습니다.

조건문은 조건문에 작성된 조건 판단식을 통해 실행 여부를 결정하게 됩니다.

조건 판단식이 참(True) 값일 때 조건문 내부의 알고리즘을 실행하게 됩니다.

Python에서는 if, elif, else 또는 Tuple, Dictionary, 삼항연산자을 이용하여 조건문을 구성할 수 있습니다.

조건 판단식

조건 판단식은 주로 관계 연산자(Relational Operator)와 논리 연산자(Logical Operator) 등을 통해 식을 구성합니다.

관계 연산자는 등식이나 부등식을 사용해 표현하며, 종류는 다음과 같습니다.

관계 연산자(Relational Operator)

연산	의미
x >y	x가 y보다 큼
x >= y	x가 y보다 크거나 같음
x <y	x가 y보다 작음
x <= y	x가 y보다 작거나 같음
x == y	x와 y가 같음
x != y	x와 y가 같지 않음

논리 연산자는 앞선 6강의 비트 연산과 동일하거나 비슷한 의미를 갖습니다.

논리곱(and), 논리합(or), 논리부정(not) 등을 통해 표현할 수 있습니다.

주로, 복합적인 조건 판단식을 구성할 때 사용합니다.

논리 연산자(Logical Operator)

연산	의미
x and y	x와 y의 연산 결과가 모두 참 값
x or y	x와 y의 연산 결과 중 하나라도 참 값
not x	x의 연산 결과를 반전

```
if
a = 100
if a > 100:
    print("크다")
elif a == 100:
    print("같다")
else:
    print("작다")
```

결과

```
같다
```

콜론(:)과 탭(Tab)을 이용하여 해당 분기문에 대한 내용을 구성할 수 있습니다.

분기문은 첫 분기(if)에 부합하지 않으면 다음 분기(elif)로 넘어가며, 어떤 분기에도 포함되지 않는 경우 최종 분기(else)로 넘어갑니다.

만약, 최종 분기(else)를 작성하지 않으면, 어떠한 분기도 포함되지 않아, 분기를 무시하고 넘어갑니다.

```
a = 100
b = 40

if (100 <= a < 130) and not (50 < a < 100):
    print("if-1 : 모두 조건에 만족")

if a < 50 or (b - 40) == 0:
    print("if-2 : 하나라도 만족")

if b > 0 or b > 0 and a < 50:
    print("if-3 : 조건 우선식")
```

결과

if-1 : 모두 조건에 만족
if-2 : 하나라도 만족
if-3 : 조건 우선식

조건문을 사용할 때 주의점은 조건식 내부에도 우선 순위가 있다는 점입니다.

if-1과 if-2 조건에는 명시적으로 소괄호(())를 사용해 연산이 우성되어야할 정보를 입력했습니다.

하지만, if-3을 확인해본다면, True or True and False가 되어, 어떤 연산이 우선될지 알 수 없습니다.

(True or True) and False로 소괄호가 앞에 연결되었다면, True and False가 되어 최종적으로 False가 됩니다.

True or (True and False)로 소괄호가 뒤에 연결되었다면, True or False가 되어 최종적으로 True가 됩니다.

즉, 조건식 내부에도 우선 순위가 있음을 확인할 수 있습니다.

수식의 우선 순위는 아래와 같습니다.

연산자 우선 순위(Operators Precedence)

연산	의미
**	거듭 제곱
~x, +x, -x	단항 연산
*, @, /, //, %	곱셈 및 나누기 연산
«, »	Bitwise Shift
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
>, >=, <, <=	비교 연산자
==, !=	비교 연산자(평등)
in, not in, is, is not	식별 연산자
not	논리 연산자
and	논리 연산자
or	논리 연산자

연산자 우선 순위에서 확인할 수 있듯이 and는 or보다 먼저 연산됩니다.

True or True and False 연산은 and가 먼저 연산되므로 True or (True and False)가 됩니다.

즉, 최종 결과는 True가 됩니다.

연산자 우선 순위를 모두 외우고 사용하는 것은 불가능하며, 코드 구성에 있어서 효율적이지도 않습니다.

그러므로, 소괄호를 사용해 먼저 연산될 수식을 설정합니다.

Tuple

```
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)
```

결과

```
1
1
```

튜플을 이용하여 if문처럼 사용할 수 있습니다.

튜플을 생성하여 (거짓, 참) [조건]으로 사용할 수 있습니다.

Dictionary

```
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)
```

결과

```
알 수 없음
알 수 없음
```

사전을 이용하여 if문처럼 사용할 수 있습니다.

사전을 생성하여 key와 value 값을 할당합니다.

사전.get(key, 예외)을 이용하여 key를 호출하여 value를 불러옵니다. 값이 없는 경우 예외 구문을 출력합니다.

삼항연산자

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



a = 5
b = a * 2 if a > 5 else a - 4

print(b)
```

결과

```
1
1
```

삼항연산자를 사용하여 코드를 간략화 할 수 있습니다.

참값 if 조건 else 거짓값으로 구성할 수 있습니다.

조건에 부합할 경우 참값을 실행하게 되며 부합하지 않을 경우 거짓값을 실행합니다.

반복문(for, while)

반복문이란 특정한 부분의 코드가 반복적으로 수행되는 구문입니다.

특정 조건을 만족할 때까지 반복해서 수행하게 되며, 루프(Loop)라고도 합니다.

Python에서는 for와 while을 이용하여 반복문을 구성할 수 있습니다.

반복문의 조건으로 List, Tuple, Str, Range 값도 사용할 수 있습니다.

for 문

```
L = ["일","이","삼"]
T = (1, 2, 3)
S = "윤대희"
total = 0

for i in L:
    print(i)

for i in T:
    print(i)

for i in S:
    print(i)

for i in range(1,10):
    total += i
print (total)
```

결과

```
일
이
삼
1
2
3
윤
대
희
45
```

for 원소 in 반복값:을 이용하여 반복문을 실행시킬 수 있습니다.

반복값 내부에 있는 요소를 순차적으로 꺼내 사용하게 됩니다.

반복값에는 리스트, 튜플, 문자열, 범위 값 등을 이용할 수 있습니다.

for 문 - else

```
for i in range(3):
    print("i =", i)
else:
    print("END")


for j in range(3):
    if j == 2:
        break
    print("j =", j)
else:
    print("END")
```

결과

```
i = 0
i = 1
i = 2
END
j = 0
j = 1
```

for else문은 for 문이 끝까지 실행되어 종료됐을 때, else 구문으로 넘어갑니다.

즉, 중간에 중단(break)되지 않고 끝까지 실행된다면 else 구문을 실행합니다.

else는 for의 탭 간격과 동일하게 사용되며, 도중에 중단된다면 else 구문을 실행하지 않습니다.

while 문

```
L = ["일","이","삼"]
T = (1, 2, 3)
S = "윤대희"
total = 0

i = 0
while i < len(L):
    print(L[i])
    i += 1

i = 0
while i < len(T):
    print(T[i])
    i += 1

i = 0
while i < len(S):
    print(S[i])
    i += 1

i = 0
while i < 10:
    total += i
    i += 1
print(total)
```

결과

```
일
이
삼
1
2
3
윤
대
희
45
```

while 조건:을 이용하여 반복문을 실행시킬 수 있습니다.

조건에는 참과 거짓의 값으로 반환되어야 합니다.

참 값일 때 반복하며 거짓 값일 때 종료합니다.

만약, 참 값이 되지 않는다면 무한 반복에 빠지게 되며, 강제로 종료하기 전까지 종료되지 않습니다.

for문 - continue & break

```
for i in range(5):
    if i == 3: break
    print(i)
print("break")

for i in range(5):
    if i == 3: continue
    print(i)
print("continue")
```

결과

```
0
1
2
break
0
1
2
4
continue
```

반복문은 반복 횟수 만큼 반복이 진행되거나, 특정 조건을 만족해야 반복문이 종료됩니다.

하지만, 조건문(if) 등을 통해 반복 도중 특정 조건이 만족한다면 모든 반복을 완료하지 않아도 반복을 종료할 수 있습니다.

조건문과 break, continue 등을 이용해 특정 조건을 만족할 때 종료하거나 건너뛰게 할 수 있습니다.

while문 - continue & break

```
i = 0
while i < 5:
    i += 1
    if i == 3: break
    print(i)
print("break")

i = 0
while i < 5:
    i += 1
    if i == 3: continue
    print(i)
print("continue")
```

결과

```
0
1
2
break
0
1
2
4
5
continue
```

while 문도 for 문과 마찬가지로 조건문 등을 이용해 내부 반복을 수정할 수 있습니다.

조건문과 break, continue를 이용해 특정 조건을 만족할 때 종료하거나 건너뛰게 할 수 있습니다.

Tip : break 또는 continue 구문을 만났을 때 종료하거나 건너뜁니다.

Tip : while문에서 continue구문을 잘못 사용할 경우 무한 루프에 빠질 수 있습니다.

데이터 입력(input)

입력(input) 함수를 이용해 사용자에게서 데이터를 입력받을 수 있습니다.

입력 받은 초기 데이터 형식은 숫자(Number)를 입력하더라도 문자열(str)로 간주합니다.

단일 데이터 입력

```
datum = input("입력 : ")

answer = int(datum) + 5
print(answer)
```

결과

```
입력 : 3
8
```

int(데이터)를 통하여 문자열을 정수형으로 변환하여 계산합니다.

다중 데이터 입력

```
data = input("입력 (x,y,z) : ")

L = data.split(',')
x, y, z, = L[0], L[1], L[2]

print(x)
print(y)
print(z)
```

결과

```
입력 (x,y,z) : 1,2,3
1
2
3
```

a.split(b)를 이용하여 data에서 콤마(,)마다 분리하여 리스트로 저장합니다.

이 후, 각각의 변수에 할당합니다.

다중 데이터 입력 간소화

```
x, y, z = input("데이터 입력 (a,b,c) : ").split(',')

print(x)
print(y)
print(z)
```

결과

```
데이터 입력 (a,b,c) : 4,qqq,07
4
qqq
07
```

input()에서 데이터 할당과 동시에 a.split(b) 구문을 이용하여 콤마(,)마다 분리하여 저장합니다.

Tip : 반복문을 이용하여 콤마(,) 이외에도 글자 사이사이에 있는 공백 ( )을 제거하여 할당할 수 있습니다.
Tip : input() 함수의 할당 된 데이터는 str 형식으로 취급합니다.

데이터 출력(print)

출력(print) 함수를 이용해 확인하고자 하는 데이터를 출력할 수 있습니다.

출력 함수를 사용할 때, 데이터를 정렬하거나 간격 등을 설정한다면 효율적으로 값을 확인할 수 있습니다.

퍼센트 연산자

```
a = 3.0

print ("정수형 출력 : %d" % a)
print ("실수형 출력 : %f" % a)
```

결과

```
정수형 출력 : 3
실수형 출력 : 3.000000
```

% 연산자를 활용하여 형변환을 할 수 있습니다.

형식 알아보기 : 10강 바로가기

폭 설정

```
a = 3.0

print ("폭 설정 : %3d" % a)
print ("폭 설정 : %5d" % a)
```

결과

```
폭 설정 :   3
폭 설정 :     3
```

%상수를 입력하여 상수 크기 만큼의 폭이 설정됩니다.

정밀도 설정

```
a = 0.123456789

print ("정밀도 설정 : %.3f" % a)
print ("정밀도 설정 : %.5f" % a)
```

결과

```
정밀도 설정 : 0.123
정밀도 설정 : 0.12346
```

%.상수를 입력하여 소수점(.) 기호 뒤의 상수 크기 만큼의 정밀도가 설정됩니다.

Tip : 정밀도는 반올림하여 처리합니다.

폭과 정밀도 설정

```
a = 1234.56789

print ("폭&정밀도 설정 : %6.3f" % a)
print ("폭&정밀도 설정 : %13.5f" % a)
```

결과

```
폭&정밀도 설정 : 1234.568
폭&정밀도 설정 :    1234.56789
```

%n.m를 입력하여 n 크기 만큼의 폭과 m 크기 만큼의 정밀도로 설정합니다.

format 설정 (1)

```
a = 1
b = 2.0
c = "Python"

print ("정수:{0}\n실수:{1}\n문자열:{2}".format(a,b,c))
```

결과

```
정수:1
```

실수:2.0
문자열:Python

"{0}{1}{2}...{n}".format(index0, index1, index2, ..., indexn)을 사용하여 포맷 형식을 설정할 수 있습니다.

format의 인덱스의 순서대로 삽입됩니다.

format 설정 (2)

```
L = ["1번", "2번", "3번"]

print ("L0 = {0[0]}\nL1 = {0[1]}".format(L))
```

결과

```
L0 = 1번
L1 = 2번
```

"{0[0]}{0[1]}....{n[m]}".format(A,B...)"을 사용하여 포맷 형식을 설정할 수 있습니다.

n은 format에 포함된 순서를 의미하며 m은 인덱스 안의 요소를 의미합니다.

예외 형식(try-except-finally)

예외 형식(try-except-finally) 함수를 이용해 오류가 발생하는 부분을 처리할 수 있습니다.

특정 오류가 발생할 때, 이를 처리해 알고리즘이 원할하게 구동할 수 있도록 구현할 수 있습니다.

또한, 특정 오류가 발생했음을 사용자에게 알려주어 입력 데이터를 조건에 맞게 입력할 수 있게 합니다.

try - except - finally

```
while(True):

    a = input("숫자입력 : ")

    try:
        print(int(a))

    except:
        print ("숫자가 아닙니다.")

    finally:
        print("최종 :", a)

    print("----------------------")
```

결과

```
숫자입력 : 123
123
최종 : 123
--------
숫자입력 : alpha
숫자가 아닙니다.
최종 : alpha
--------
```

while(True):를 이용하여 일정 구문을 반복합니다.

try: 구문은 오류가 발생하지 않았을 때 실행되는 구문입니다.

except: 구문은 오류가 발생했을 때 실행되는 구문입니다.

finally: 구문은 오류와 무관하게 실행되는 구문입니다.

try-except-finally 구문을 활용하여 ValueError 등을 방지할 수 있습니다.

Tip : finally 구문은 필수요소가 아니며, 불필요한 경우 사용하지 않아도 됩니다.

try - except - finally

```
while(True):

    a = input("숫자입력 : ")

    try:
        print(int(a))

    except ValueError:
        try:
            print(int(float(a)))

        except:
            print("오류 발생")

    print("----------------------")
```

결과

```
숫자입력 : 123.123
123
--------
숫자입력 : 윤대희
오류 발생
--------
```

특정 오류에 대해서 except 오류사항:을 통해 예외처리할 수 있습니다.
ValueError에 대해서만 예외처리할 경우, 다른 오류가 발생한다면 프로그램이 중지됩니다.
예외 처리를 해도, except 구문 안에서도 오류가 발생할 수 있습니다.
예제와 같이 구문이 복잡해지지 않도록 하나의 예외 처리사항 안에서 문제를 해결하도록 구성합니다.

간소화(Comprehension)

간소화(Comprehension)란 반복 가능한 객체(iterable)들을 축약한 형태으로 생성하는 방법입니다.

대표적으로 리스트(List), 튜플(Tuple), 집합(Set) 사전(Dict), 생성자(Generator) 등이 있습니다.

즉, List나 Tuple 등을 한 줄로 생성하여 간소화할 수 있습니다.

반복 가능한 객체들을 간소화하게 되면 코드 구성이 깔끔해져 더 읽기 쉬운 코드를 구성할 수 있습니다.

또한, 조건문 등을 추가하여 코드를 간략화 할 수 있습니다.

List Comprehension

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

print(L)
```

결과

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

[리스트 값 for 변수 in 범위]를 이용하여 범위만큼 반복하여 변수를 할당해 리스트에 리스트 값을 채웁니다.

range(10)은 0~9의 값을 지니고 있으며 i에서 각각의 값들을 호출합니다.

for문을 통해 반복하게 되어 리스트 값을 조건에 맞게 채우게됩니다.

Comprehension + if

```
L1 = [i ** 2 for i in range(10) if (i > 5)]

print(L1)
```

```
L2 = [i ** 2 for i in range(10) if (i > 5) and (i % 2 == 0)]

print(L2)

L3 = [i ** 2 if i < 5 else i for i in range(10)]

print(L3)
```

결과

```
[36, 49, 64, 81]
[36, 64]
[0, 1, 4, 9, 16, 5, 6, 7, 8, 9]
```

[리스트 값 for 변수 in 범위 if (조건)]을 이용하여 if (조건)을 추가해 조건식에 맞는 경우의 리스트를 생성할 수 있습니다.

if (i > 5)는 6 ~ 9의 값을 지니고 있으며 i에서 각각의 값을 호출할 때 5부터 값을 호출합니다.

조건에는 and나 or 등을 포함하여 조건식을 추가할 수 있습니다.

또한, 간소화 구문에는 삼항 연산자(Ternary operators)를 적용할 수 있습니다.

참 값 if 조건 else 거짓 값의 형태로 값이 반환되므로, 리스트 값 영역에 참 값 또는 거짓 값을 할당할 수 있습니다.

Multiple Comprehension

```
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)
```

결과

```
['a', 'b', 'c', 'd', 'e', 'f']
```

[[‘aa’, ‘bb’, ‘cc’], [‘dd’, ‘ee’, ‘ff’]]

간소화 구문은 여러 번 사용하여 반복할 수 있습니다.

L은 2차원 리스트로 구성되어 있어 1차원 리스트로 변환한다면 두 번의 반복문의 구성을 필요로 합니다.

여기서, 간소화 구문을 두 번 반복하여 1차원 리스트로 간단하게 변경할 수 있습니다.

기본적인 간소화 구문은 [i for i in L]의 형태를 갖습니다.

하지만, 이 구문은 [‘a’, ‘b’, ‘c’]와 [‘d’, ‘e’, ‘f’]를 반복하므로 한 번 더 반복해야 합니다.

그러므로 i에 대한 반복문을 추가로 구성합니다.

[j for j in i]의 구성이 필요합니다. 여기서, 앞 부분이 아닌 뒷 부분에 연결합니다.

그러므로, [i for i in L]에서 반환 해줄 리스트 값은 j가 되므로 [j for i in L]로 먼저 변경합니다.

이후에, L 뒷 부분에 for j in i를 이어 붙여 [j for i in L for j in i] 형태로 구성합니다.

만약, 3차원 리스트를 평평하게 만든다면 위와 같은 방식으로 맨 앞의 j 대신에 k로 변경한 다음, for k in j를 그대로 이어 붙이면 됩니다.

즉, [k for i in L for j in i for k in j]과 같은 구성이 됩니다.

다음으로는, 뒷 부분에 이어붙이는 방법이 아닌 앞 부분에 이어 붙이는 방법입니다,

이 방법은 차원을 그대로 유지한 상태로 값을 변경할 수 있습니다.

리스트에 채워질 i에 대해서 다시 간소화 구문을 반복하는 방법입니다.

Tuple Comprehension

data = [0, 7, 6, 9, 2, 3]

```
T = tuple((i for i in data))

print(T)
```

결과

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

Tuple은 다른 간소화와 다르게, 명시적으로 tuple()로 변경해야 합니다.

소괄호를 사용한 간소화는 생성자 표현(Generator Expression)입니다.

생성자는 반복 가능한(iterable) 형식으로 반환할 수 있습니다.

그러므로, 생성자(Generator)를 tuple()로 변경해 사용합니다.

Set Comprehension

```
text = "YUNDAEHEE"

S1 = set([i for i in text])

print(S1)

S2 = {i for i in text}

print(S2)
```

결과

```
{'E', 'Y', 'N', 'D', 'A', 'H', 'U'}
{'E', 'Y', 'N', 'D', 'A', 'H', 'U'}
```

간소화을 이용할 때, 범위를 문자열로 사용하여 문자 값을 직접 할당할 수 있습니다.

또한 Set의 연산이 필요할 때, Set Comprehension를 이용한다면 S1에서 사용된 List → Set 구간을 줄여 전체적인 연산량을 감소시킬 수 있습니다.

Dictionary Comprehension

```
text = "cheese"
```

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

print(D)
```

결과

```
{'c': 1, 'h': 1, 'e': 3, 's': 1}
```

List 이외에도 Dictionary Comprehension를 이용하여 사전을 간소화할 수 있습니다.

{key : value for 변수 in 범위}를 이용하여 범위만큼 반복하여 변수를 할당해 key값과 value값을 채웁니다.

key에는 문자가 담겨있으며 value에는 문자의 개수가 담겨있습니다.

사용자 정의 함수(User-defined Functions)

사용자 정의 함수(User-defined Functions)는 코드를 구조적으로 설계할 수 있게 하는 역할을 합니다.

이를 통해 코드의 구조화(structuring), 모듈화(modularization), 재사용성(reusable), 가독성(Readability) 등을 높일 수 있습니다.

Python에서는 def와 return을 사용하여 사용자 정의 함수를 생성할 수 있습니다.

동일한 작업이 반복되는 경우에도 사용자 정의 함수를 호출하여 사용할 수 있습니다.

> Tip : 동일한 작업이 반복되지 않더라도, 코드의 가독성이나 모듈화 등을 위해 사용자 정의 함수를 사용합니다.

함수 생성

```
def add(a, b):
    return a + b


print(add(3, 4))
print(add([1, 2, 3], [4, 5, 6]))
print(add("YUN", "DAEHEE"))
```

결과

```
7
[1, 2, 3, 4, 5, 6]
YUNDAEHEE
```

def 함수명(매개변수1, 매개변수2, ...):를 사용하여 사용자 정의 함수를 선언할 수 있습니다.

함수명으로 사용자 정의 함수를 호출합니다.

해당 함수에서 사용될 인수들을 매개변수들에 선언합니다.

return 반환값을 사용하여 반환값이 결과로 반환됩니다.

> Tip : 정수, List, 문자열 등을 사용 할 수 있습니다.

기본값 설정

```
def add(a, b=10):
    return a + b


print(add(1))
print(add(1, 2))
```

결과

```
11
3
```

매개변수에 기본값을 할당할 수 있습니다.

기본값이 할당되면 함수를 호출할 때 매개변수를 채우지 않으면 기본값으로 사용합니다.

다중 입력

```
def add(*num):
    result = 0

    for i in num:
        result += i

    return result


print(add(1, 2, 3))
```

결과

```
6
```

매개변수에 *를 입력할 경우 개수를 지정하지 않고 매개변수를 지정할 수 있습니다.

이를 인자(Argument)라 하며, 매개변수에 전달되는 실질적인 값(value)를 의미합니다.

다중 반환

```
def calc(a, b):
    return a + b, a - b, a * b, a / b


print(calc(3, 1))
```

결과

```
(4, 2, 3, 3.0)
```

반환값을 여러개로 지정할 경우 Tuple 형식으로 결과들이 반환됩니다.

결괏값을 하나만 사용하는 res = calc(3, 1) 구조일 경우 Tuple 값으로 하나만 반환합니다.

만약, 결괏값을 함수 반환값 개수 만큼 사용하는 res1, res2, res3, res4 = calc(3, 1) 구조일 경우 각각의 결괏값이 변수에 할당됩니다.

일급 함수

```
def add(a, b):
    return a + b


def func(x, y, z):
    return x(y, z)


plus = add

print(plus(3, 4))
print(func(plus, 3, 4))
```

결과

```
7
7
```

사용자 정의 함수를 다른 함수의 인수로 사용할 수 있으며, 변수나 자료구조에도 저장이 가능합니다.

사용자 정의 함수로 구성된 add 함수에 plus=add를 추가하여 add 함수를 plus로 사용이 가능합니다.

또한, func함수에서 x는 함수, y, z를 인수로 사용이 가능합니다.

func(함수, 인수1, 인수2)로 사용하여 사용자 정의 함수안에서 사용자 정의 함수를 호출 할 수 있습니다.

람다(lambda) 함수

람다(lambda) 함수는 사용자 정의 함수와 비슷한 역할을 합니다.

함수명을 정해 사용하는 사용자 정의 함수와 다르게 이름없이 사용하는 익명 함수입니다.

간단한 기능을 정의해 한 번만 사용하고 사라지게 됩니다.

람다 함수는 함수명 = lambda 변수명 : 수식의 형태로 사용합니다.

변수명은 x, y, z 등의 값을 사용하며, 수식은 변수명에서 사용된 값을 활용합니다.

lambda를 사용하여 한 줄로 되어있는 수학 함수를 생성하거나 특정 형태나 형식 등으로 변환할 수 있습니다.

간단한 수학함수가 필요한 경우 람다 함수를 호출하여 사용합니다.

함수 생성

```
f = lambda x, y: x + y

print(f(1, 2))
print(f([1, 2], [3, 4]))
print(f("YUN", "DAEHEE"))
```

결과

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

함수명=lambda 매개변수1, 매개변수2, ... : 반환식을 사용하여 람다 함수를 선언할 수 있습니다.

함수명(인수1, 인수2 ...)를 사용하여 함수를 호출합니다.

Tip : 정수, List, 문자열 등을 사용 할 수 있습니다.

기본값 설정

```
f = lambda x, y=3: x + y

print(f(1))
print(f(3, 3))
```

결과

```
4
6
```

매개변수에 기본값을 할당할 수 있습니다.

기본값이 할당되면 함수를 호출할 때 매개변수를 채우지 않으면 기본값으로 사용합니다.

다중 입력

```
f = lambda *x: max(x) * 2

print(f(1, 3, 7))
```

결과

```
14
```

매개변수에 *를 입력할 경우 개수를 지정하지 않고 매개변수를 지정할 수 있습니다.

여러개의 매개변수들이 포함될 수 있습니다.

단, 반환값은 하나의 값만 반환할 수 있습니다.

다중 반환

```
f = [lambda x: x + 1, lambda x: x + 2, lambda x: x + 3]

print(f[0](1))
print(f[1](1))
print(f[2](1))
```

결과

```
2
3
4
```

List로 사용할 경우 대괄호([]) 를 사용하여 선택된 수식으로 반환됩니다.

List가 아닌, Dict로 사용할 경우, 원하는 Key 값을 통해 특정 람다 함수를 사용할 수도 있습니다.

최대/최소

```
L = [[9, 1], [8, 2], [7, 3], [6, 4]]

max1 = max(L, key=lambda x: x[0])
max2 = max(L, key=lambda x: x[1])
min1 = min(L, key=lambda x: x[0])
min2 = min(L, key=lambda x: x[1])

print(max1)
print(max2)
print(min1)
print(min2)
```

결과

```
[9, 1]
[6, 4]
[6, 4]
[9, 1]
```

최댓값 함수와 최솟값 함수에도 람다식을 적용하여 특정 최대/최소를 검출할 수 있습니다.

1차원 이상의 객체에서 특정 요소에 있는 값을 기준으로 찾으려 할 때, 어떤 요소의 값을 기준으로 찾을지 설정할 수 있습니다.

lambda x: x[0]라면 첫 번째 요소의 값으로 검색을 시작하며, lambda x: x[1]라면 두 번째 요소의 값으로 검색을 시작합니다.

정렬

```
L = ["가", "각", "감", "갂", "갔", "+", "/", "!", "간", "]", "[", "\\"]

L.sort(key=lambda x: x)

print(L)

print([ord(i) for i in L])
```

결과

```
['!', '+', '/', '[', '\', ']', '가', '각', '갂', '갔', '간', '감']
[33, 43, 47, 91, 92, 93, 44032, 44033, 44034, 44035, 44036, 44048]
```

Python에서 사용되는 모든 문자열은 유니코드(Unicode)로 간주합니다.

그러므로, 문자열을 정렬할 때 각 문자들을 유니코드로 변경하고 인스턴스 끼리의 비교(<, >)를 진행하게 됩니다.

정렬에 사용되는 람다식의 x는 “가”, “각”, “감”, … “\“이 되며, 이 값들을 유니코드로 변경하여 대소 비교를 통해 정렬합니다.

만약, 정렬하려는 반복 가능한 객체(iterable)에 서로 다른 데이터 타입이 존재한다면, '<' not supported between instances of 'int' and 'str'등 의 오류가 발생합니다.

이럴 경우에는 명시적으로 요솟값을 str이나 int로 변경하여 정렬합니다.

복합 정렬

```
L = [[3, "d"], [5, "c"], [1, "a"], [0, "b"], [0, "a"]]
L1 = sorted(L, key=lambda x : -x[0])
L2 = sorted(L, key=lambda x : x[1])
L3 = sorted(L, key=lambda x : [x[0], x[1]])

print(L)
print(L1)
print(L2)
print(L3)
```

결과

```
[[3, 'd'], [5, 'c'], [1, 'a'], [0, 'b'], [0, 'a']]
[[5, 'c'], [3, 'd'], [1, 'a'], [0, 'b'], [0, 'a']]
```

[[1, ‘a’], [0, ‘a’], [0, ‘b’], [5, ‘c’], [3, ‘d’]]
[[0, ‘a’], [0, ‘b’], [1, ‘a’], [3, ‘d’], [5, ‘c’]]

정렬하려는 객체가 2차원 이상이라면, 비교하는 x에는 [3, “d”], [5, “c”], … [0, “a”]가 됩니다.

만약, 특정 원소를 지정하지 않고 key=lambda x : x처럼 사용한다면, 첫 번째 원소를 기준으로 정렬하게 됩니다.

특정한 원소를 기준으로 정렬한다면, 어떤 값을 기준으로 정렬할지 표기합니다.

여기서, 정렬 함수는 기본적으로 오름차순 형태로 정렬합니다.

그러므로, 비교하는 원소를 음수로 변경한다면 내림차순으로 변경할 수 있습니다.

여러가지 기준으로 정렬하려고 한다면, 객체로 감싸 정렬 우선순위를 할당할 수 있습니다.

예제의 L3의 경우 첫 번째 원소를 기준으로 정렬하면서, 두 번째 원소를 차선으로 정렬하게 됩니다.

즉, [0, "b"]와 [0, "a"]를 비교할 때, 0이 동일하므로, 그 다음 조건으로 a와 b를 대상으로 다시 정렬하게 됩니다.

필터(filter) 함수

필터(filter) 함수는 반복 가능한 데이터(목록, 사전 등)의 요소에서 특정 조건을 만족하는 값들만 추출합니다.

필터 함수는 결괏값 = filter(조건 함수, 범위)의 형태로 사용합니다.

조건 함수는 논리값(True or False)을 반환하는 함수를 사용하며, 참(True)이 되는 값만 반환합니다.

lambda 함수와 list 함수 등을 같이 사용하여 결과를 반환합니다.

함수 생성

```
f = lambda x: x > 0

print(list(filter(f, range(-5, 5))))
```

결과

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

filter(함수, 범위)를 사용하여 참(True)의 결과를 반환합니다.

범위의 값을 함수에 대입해 참이 되는 값만을 묶어 반환합니다.

list() 또는 set() 등의 구문을 추가하지 않으면 <filter object at 0x05B3A830> 등의 iterator 형태로 출력됩니다.

사용자 정의 함수 사용

```
def func(x):
    if x > 0:
        return x
    else:
        return x - 100


print(list(filter(func, range(-5, 5))))
```

결과

[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

필터 함수에서 사용되는 함수는 논리값을 반환하는 함수로 인지합니다.

그러므로, 반환값이 0이 아닌 모든 값은 참(True) 값으로 간주하게 됩니다.

결국, 어떤 반환 형태도 0을 반환하지 않으므로 모두다 참 값이 되어, 범위를 그대로 반환하게 됩니다.

```
def func(x):
    if (x / 2) - 1 > 0:
        return True
    elif x % 2 == 0:
        return True
    else:
        return False


print(list(filter(func, range(-5, 5))))
```

결과

[-4, -2, 0, 2, 3, 4]

사용자 정의 함수를 이용한다면, 복잡한 형태의 필터 함수를 쉽게 구현할 수 있습니다.

반환값이 참인 경우에만 해당 값을 반환하므로, 특정한 형태의 데이터만 추출할 수 있습니다.

Tip : 반환값이 없는 경우, 거짓(False) 값으로 간주해 필터 함수에서 값이 반환되지 않습니다.

맵(map) 함수

맵(map) 함수는 반복 가능한 데이터(목록, 사전 등)의 요소를 함수에 적용해 결과를 반환하는 함수입니다.

주로 반복문을 간소화하거나, 람다(lambda) 함수에 모든 값을 적용할 때 사용합니다.

맵 함수는 결괏값 = map(적용 함수, 범위)의 형태로 사용합니다.

map를 사용하여 범위의 대한 모든 값을 적용 함수에 대입하며, 모든 결과를 반환합니다.

lambda 함수와 list 함수를 같이 사용하여 결과를 반환합니다.

함수 생성

```
f = lambda x: x > 0
g = lambda x: x ** 2

print(list(map(f, range(-5, 5))))
print(list(map(g, range(5))))
```

결과

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

map(적용 함수, 범위)를 사용하여 모든 결과를 반환합니다.

list() 또는 set() 등의 구문을 추가하지 않으면 <map object at 0x05A2A890> 등의 iterator 형태로 출력됩니다.

형 변환

```
data = [-1.3, 5.5, 5.4]

f = map(int, data)

print(list(f))
```

결과

[-1, 5, 5]

간단한 형변환의 경우에는 map 함수를 사용한다면, 반복문 구성없이 간단하게 구현할 수 있습니다.

특정 연산 적용

```
def calc(x):
    return (x / 2) + 1


data = [-1.3, 5.5, 5.4]

f = map(calc, data)

print(list(f))
```

결과

[0.35, 3.75, 3.7]

맵(map) 함수는 논리 값과 관계 없이 모든 원소값을 함수에 적용하므로, 단순 반복 형태의 코드를 간단하게 구현할 수 있습니다.

필터(filter) 함수의 경우, 논리 값에 허용되는 조건만 적용하므로, 특정 조건에 관계 없이 일괄 적용할 수 있습니다.

Tip : 맵 함수와 필터 함수를 동시에 적용할 경우, 복잡한 형태의 구문도 간단하게 구현할 수 있습니다.

집(zip) 함수

집(zip) 함수는 길이가 같은 반복 가능한 데이터(목록, 사전 등)를 하나로 묶어 반환하는 함수입니다.

주로, 형태가 다른 값들을 하나의 구조나 변수로 활용하기 위해 사용합니다.

집 함수는 결괏값 = map(범위 1, 범위 2, ...)의 형태로 사용합니다.

목록(list), 집합(set), 사전(dict) 등을 묶어 결과를 반환합니다.

함수 생성

```
a = "YUN"
b = [1, 2, 3]
c = ("하나", "둘", "셋")

print(list(zip(a, b, c)))
print(set(zip(a, b, c)))
print(dict(zip(a, b)))
```

결과

```
[('Y', 1, '하나'), ('U', 2, '둘'), ('N', 3, '셋')]
{('Y', 1, '하나'), ('N', 3, '셋'), ('U', 2, '둘')}
{'Y': 1, 'U': 2, 'N': 3}
```

zip(자료형1, 자료형2, ...)을 사용하여 묶어 결과를 반환합니다.

집(zip) 함수는 list() 또는 set() 등의 구문을 추가하지 않으면 <zip object at 0x0506C940> 등의 iterator 형태로 출력됩니다.

dict()의 경우 key와 value로 구성되어 있으므로 3개 이상은 묶을 수 없습니다.

반복문 적용

```
L1 = ["A", "B", "C", "D"]
L2 = ["가", "나", "다", "라"]
```

```
    for i, j in zip(L1, L2):
        print(i, j)
```

결과

```
    A 가
    B 나
    C 다
    D 라
```

집(zip) 함수는 서로 다른 자료형을 하나로 묶을 수 있기 때문에, 두 종류 이상의 색인(index)을 반복할 수 있습니다.

이 특성으로 인해, 다중 반복문(중첩 반복문)을 사용하지 않고 하나의 반복문으로 간소화 할 수 있습니다.

언패킹(Unpacking) / 패킹(Packing)

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

    print(*numbers)
    print(list(zip(*numbers)))
    print(list(zip([1, 2, 3], [4, 5, 6])))
```

결과

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

반복 가능한 객체에 별표(Asterisk, *)를 함께 사용한다면, 언패킹되어 묶여있던 객체들이 나눠지게 됩니다.

즉, 2차원 리스트는 1차원 리스트 만큼 나눠서 반환하게 됩니다.

이 때, 집 함수를 사용하여 다시 패킹한다면 각 원소마다 묶기 때문에 여러 반복문을 구성하지 않아도 다시 패킹 할 수 있습니다.

zip(*n차원 객체)는 zip(n-1차원 객체1, n-1차원 객체2, ...)로 볼 수 있습니다.

그러므로, 반복 가능한 객체의 원소들을 대상으로 새로운 데이터를 구현할 수 있습니다.

파일 읽기 & 쓰기(File Read & Write)

파일 열기 함수(open)는 대부분의 파일을 읽고 쓰기가 가능한 함수입니다.

파일 열기 함수는 주로 텍스트 파일 포맷의 형태를 대상으로 사용합니다.

open과 close를 이용하여 텍스트 파일을 다룰 수 있습니다.

텍스트 파일 열기 & 닫기

```
file = open("d:/textfile.txt", mode="r")
file.close()
```

텍스트 파일을 작성하기 위해선 저장될 경로를 미리 할당해야 합니다.

파일 열기 함수(open)을 사용해 파일을 작성할 수 있습니다.

변수 = open(경로, 모드)을 의미합니다.

경로는 파일을 작성하거나 읽을 경로를 의미합니다.

모드는 파일을 어떻게 사용할지를 설정합니다.

파일 열기 함수 모드 플래그

플래그	의미
r	읽기 전용(기본값)
w	쓰기 전용
a	내용 추가
t	텍스트 모드(기본값)
b	이진 모드

모드는 크게 r, w, a의 세 가지와 t, b의 두 가지로 성격을 나눌 수 있습니다.

파일을 어떻게 사용할지 선택(r, w, a)하고, 어떤 방식으로 다룰지 선택(t, b)하게 됩니다.

어떻게 사용할지는 필수 값이며, 어떤 방식으로 다룰지는 옵션 값입니다.

예를 들어, mode="r"로 사용한다면, 텍스트 파일로 읽는다는 의미가 되며, mode="rb"는 이진 파일로 읽는다는 의미가 됩니다.

예제는 open("d:/textfile.txt", mode="r")로 사용하므로, D 드라이브의 textfile.txt을 읽습니다.

변수명.close()를 사용하여 텍스트 파일의 작성을 종료합니다.

텍스트 파일의 작성을 종료하지 않으면, 파일이 열어진 상태가 유지됩니다.

프로그램이 종료될 때 텍스트 파일이 자동으로 닫아지지만, 닫지 않고 종료할 경우 오류가 발생할 수 있습니다.

파일 자동 닫기

```
with open("d:/textfile.txt") as file:
    ...
```

파일을 읽거나 쓸 때, close 메서드를 사용해 명시적으로 닫아야합니다.

하지만, with 키워드를 사용한다면 탭 구문을 벗어낫을 때 자동으로 파일이 닫힙니다.

as 키워드는 열린 파일의 변수명을 설정합니다.

탭 간격으로 파일이 언제 열리고 닫히는지 확인할 수 있으므로, 코드를 더 효율적으로 관리할 수 있습니다.

텍스트 파일 쓰기

```
with open("d:/textfile.txt", mode="w") as file:
    words = ["Python\n", "YUNDAEHEE\n", "076923\n"]

    file.write("START\n")
    file.writelines(words)
    file.write("END")
```

결과

```
START
Python
```

YUNDAEHEE
076923
END

예제는 open("d:/textfile.txt", mode="w")로 사용하므로, textfile.txt 형태로 D 드라이브에 저장됩니다.

변수명.write("내용")을 사용하면 텍스트 파일에 단일 문자열을 작성할 수 있습니다.

변수명.writelines(반복 가능한 객체)을 사용하면 텍스트 파일에 문자열 목록을 순차적으로 작성할 수 있습니다.

텍스트 파일 한 줄씩 읽기

```
with open("d:/textfile.txt", mode="r") as file:
    content = list()

    while True:
        sentence = file.readline()

        if sentence:
            content.append(sentence)
        else:
            break

    print(content)


with open("d:/textfile.txt", mode="r") as file:
    content = list()

    for f in file:
        content.append(f)

    print(content)

print(L)
```

결과

['START\n', 'Python\n', 'YUNDAEHEE\n', '076923\n', 'END']

['START\n', 'Python\n', 'YUNDAEHEE\n', '076923\n', 'END']

with 키워드를 활용해 D 드라이브에 저장되어있는 textfile.txt를 불러옵니다.

텍스트 파일을 한 줄씩 읽는 방법은 크게 두 가지가 있습니다.

첫 번째 방식은 While 문을 활용한 방법입니다.

readline() 메서드는 파일의 텍스트를 한 줄씩 불러옵니다.

readline() 메서드는 다음 번 호출 때 자동적으로 다음 줄의 텍스트를 불러옵니다.

무한히 반복될 수 있으므로, 불러온 문장이 아무 것도 없다면 종료하게됩니다.

두 번째 방식은 for 문을 활용한 방법입니다.

file 변수에 텍스트들이 저장되어 있으므로, 반복해 문자열을 불러옵니다.

만약, 더 이상 불러올 문자열이 없다면, 반복이 종료됩니다.

Tip : \n은 개행 문자로, 줄 바꿈을 의미합니다.

텍스트 파일 모두 읽기

```
with open("d:/textfile.txt", mode="r") as file:
    lines = file.readlines()
    print(lines)

with open("d:/textfile.txt", mode="r") as file:
    lines = file.read()
    print(lines)
```

결과

['START\n', 'Python\n', 'YUNDAEHEE\n', '076923\n', 'END']

START

Python
YUNDAEHEE
076923
END

텍스트 파일을 모두 읽는 방법은 한 줄씩 읽는 방법과 동일하게 두 가지가 있습니다.

첫 번째 방식은 변수명.readlines()을 이용하여 목록 형식으로 모두 불러오는 방법입니다.

readline에서 s가 추가되어 readlines로 사용합니다.

두 번째 방식은 변수명.read()를 이용해 문자열 형식으로 모두 불러오는 방법입니다.

문자열 자체로 저장하기 때문에, 개행 문자인 \n을 자동으로 줄 바꿈 처리합니다.

위 방식을 이용하면 파일 내의 모든 문자열을 한 번에 불러와 변수에 저장할 수 있습니다.

난수 모듈

Python에서는 난수 모듈을 이용하여 특정한 순서나 규칙을 가지지 않은 무작위의 숫자를 발생시키는 함수입니다.

임의의 숫자나 확률이 필요한 알고리즘이나 코드에 사용합니다.

import random
상단에 import random를 사용하여 난수 모듈을 포함시킵니다.

난수 함수의 사용방법은 random.*을 이용하여 사용이 가능합니다.

```
import random

print(random.random())
print(random.uniform(3.5, 3.6))
print(random.randrange(10))
print(random.randrange(3, 7))
print(random.randint(5, 9))
```

결과

```
0.4383375274996887
3.528242770358927
0
6
8
```

random.*을 이용하여 무작위의 숫자를 발생시킵니다.

특정 범위를 갖는 무작위 값을 반환하거나, 특정 분포의 형태를 갖는 값을 생성할 수도 있습니다.

함수	의미	반환 형식
random()	0.0 <= x <1.0	실수형
unform(a, b)	a <= x <b	실수형
randrange(a)	0 <= x <a	정수형
randrange(a, b)	a <= x <b	정수형
randint(a, b)	a <= x <= b	정수형

목록을 사용하는 난수 함수

```
import random

L = [1, 10, 100, 1000]

print(random.choice(L))
print(random.sample(L, 2))
random.shuffle(L)
print(L)
```

결과

```
100
[1000, 100]
[100, 1000, 10, 1]
```

random.*을 이용하여 목록(List)의 값을 추출하거나 변경할 수 있습니다.

함수	의미	반환 형식
choice(L)	임의의 원소값 하나를 반환	원소의 데이터 형식
sample(L, n)	임의의 원소값 n개를 반환	목록
shuffle(L)	목록 무작위 재배열	목록

난수 상태 설정

```
import random

random.seed(0)

state = random.getstate()
print(random.sample(range(10), k=5))
print(random.sample(range(10), k=5))
random.setstate(state)
print(random.sample(range(10), k=5))
```

결과

```
[6, 9, 0, 2, 4]
[7, 6, 4, 3, 2]
[6, 9, 0, 2, 4]
```

난수 함수가 생성하는 데이터는 현재 시스템 시간을 기준으로 무작위의 숫자값을 반환합니다.

그러므로, 실행할 때마다 결괏값이 달라지게 됩니다.

하지만, 발생 기준값을 동일하게 맞춘다면 항상 같은 난수를 생성하게 됩니다.

시드값 설정 함수(random.seed)로 기준값을 설정할 수 있습니다.

random.seed(시드값)을 통해 동일한 난수를 발생시킬 수 있습니다.

만약, 시드값에 None을 입력하거나 작성하지 않을 경우 현재 시간으로 설정됩니다.

시드값은 프로그램 단위로 적용되며, 코드의 줄 단위로는 일련의 규칙을 따라가게 됩니다.

즉, 첫 번째, 두 번째, … N 번째의 전체 패턴은 같아지지만, 첫 번째와 두 번째의 값은 서로 다르게 됩니다.

첫 번째와 두 번째 또는 임의의 난수 패턴을 동일하게 맞추려면, 난수 생성 상태를 저장하고 불러오는 방법이 있습니다.

난수 상태 가져오기 함수(random.getstate)를 통해 난수가 발생 될 때의 생성 조건을 가져옵니다.

난수 상태 적용하기 함수(random.setstate)로 저장한 난수 상태를 적용할 수 있습니다.

난수 상태를 가져온 뒤, 특정 구문에 적용한다면 언제 실행하더라도 항상 동일한 결과를 얻을 수 있습니다.

클래스(class)

클래스(class)는 객체 지향 프로그래밍(OOP)에서 특정한 객체를 생성하기 위해서 변수, 함수, 메소드 및 이벤트 등을 정의하는 틀입니다.

프로그램을 구성할 때, 동일한 코드나 알고리즘이 반복되거나 함수들을 하나로 묶어주는 공간이 필요하게 됩니다.

클래스를 사용하게 되면 재사용성, 가독성, 간결화된 코드를 구현할 수 있습니다.

예를 들어, 동일한 함수를 사용하더라도 변수를 추가로 더 많이 만들지 않아도 되며 어떤 함수를 실행하기 위해서 선행되어야 하는 함수 등도 중복해서 구문에 포함하지 않아도 됩니다.

```
class Human:
    def __init__(self):
        self.name = "알 수 없음"
        self.age = 99

    def man(self, name, age=10):
        self.name = name
        self.age = age

    def woman(self, name, age):
        self.name = name
        self.age = age

    def prt(self):
        print("이름은 " + self.name + "이고, 나이는 " + str(self.age))


a = Human()
a.man("박XX")

b = Human()
b.woman("김XX", 30)

c = Human()

a.prt()
```

```
    b.prt()
    print(vars(c))
```
결과
```
    이름은 박XX이고, 나이는 10
    이름은 김XX이고, 나이는 30
    {'name': '알 수 없음', 'age': 99}
```

```
    class Human:
```

class 클래스 이름:을 사용하여 클래스의 이름을 설정할 수 있습니다.

Python의 클래스 이름 표기 방식은 카멜 표기법(CamelCase)을 따르며, 첫 글자를 대문자로 표기합니다.

```
    def __init__(self):
        self.name = "알 수 없음"
        self.age = 99
```
def __init__(self)을 통해 클래스가 생성되었을 때의 변수나 조건 등을 초기화 및 실행할 수 있습니다.

self 인자는 자기 자신을 의미합니다. 여기서의 자기 자신은 클래스를 통해 생성된 인스턴스(instance)를 의미합니다.

클래스도 함수처럼 할당해서 사용하게 됩니다. 이때 할당된 변수를 인스턴스라 볼 수 있습니다.

만약 Human 클래스를 두 번 사용해 각자 다른 인스턴스를 생성한다면, 첫 번째로 생성한 인스턴스와 두 번째로 생성한 인스턴스는 서로 영향을 미치지 않습니다.

즉, self는 인스턴스의 자기 자신을 의미합니다.

self.변수 이름을 통하여 인스턴스에서 사용될 변수들의 초기값을 할당할 수 있습니다.

name 변수와 age 변수에 아무런 값도 할당되지 않는다면, 알 수 없음과 99의 값이 각각 할당됩니다.

```
    def man(self, name, age=10):
        self.name = name
        self.age = age
```
man 함수에는 self, name과 age를 매개변수로 사용합니다.

self는 필수적으로 입력되어야 하는 변수이며, 인스턴스를 의미합니다.

이 self의 유/무로도 클래스의 메서드인지, 일반 사용자 정의 함수인지도 구별할 수 있습니다.

또한, self.name과 name은 서로 다른 변수입니다.

self.name은 인스턴스 전반에 걸쳐 영향을 미치며, name은 함수 내부에서만 영향을 미칩니다.

age의 값은 10으로 초기설정이 되어있으므로, age 값을 입력하지 않는다면 10으로 자동 할당됩니다.

```
    def woman(self, name, age):
        self.name = name
        self.age = age
```
woman 함수에는 name과 age를 매개변수로 사용하며 두 개의 변수를 모두 할당해야 오류가 발생하지 않습니다.

만약, 특정 함수에서 self.변수 이름으로 선언한다면, 인스턴스 내부의 함수에서도 불러와 사용할 수 있습니다.

__init__에서 초기화하지 않더라도 일반 함수에서도 초기화, 생성, 삭제, 변경 등이 가능합니다.

```
    def prt(self):
        print("이름은 " + self.name + "이고, 나이는 " + str(self.age))
```
prt 함수는 출력용 함수이며, 값을 입력하지 않고 호출만으로도 함수가 실행됩니다.

함수에서 어떤 값도 입력받지 않고 싶더라도, self는 선언해야 사용할 수 있습니다.

```
    a = Human()
```

```
a.man("박XX")

b = Human()
b.woman("김XX", 30)

c = Human()

a.prt()
b.prt()
print(vars(c))
```

클래스 이름을 변수에 할당하여 클래스의 내부 함수를 실행시킬 수 있습니다.

변수명.클래스 함수명(매개변수1, 매개변수2)를 이용하여 a와 b 변수에 값을 할당할 수 있습니다.

```
a.prt()
b.prt()
print(vars(c))
```

*.prt() 함수는 아무런 인자를 필요로 하지 않으므로, 바로 사용이 가능합니다.

vars(변수이름)을 사용하여 사전 형식으로 할당된 값을 확인할 수 있습니다.

c 변수에는 __init__(self)를 통하여 초기화한 값인 알 수 없음과 99의 값이 할당되어 있습니다.

상속(inheritance)

상속(inheritance)은 클래스가 다른 클래스로 부터 생성된 모든 속성을 가져올 수 있는 기능입니다.

부모 클래스(Parent Class, Super Class)에서 생성된 속성을 자식 클래스(Child Class, Sub Class)에서 사용할 수 있습니다.

상속을 하더라도, 부모 클래스의 기능을 그대로 사용할 수 있습니다.

상속을 통해 코드를 구성한다면 중복되는 코드가 사라져 더 간결한 코드와 가독성 높은 프로그램을 구성할 수 있습니다.

Tip : 부모 클래스를 슈퍼 클래스라고도 부르며, 마찬가지로 자식 클래스도 서브 클래스로도 부를 수 있습니다.

```
class Human:
    def __init__(self):
        self.name = "알 수 없음"
        self.age = 99

    def set_name(self, name):
        self.name = name
        print("Human Class")

    def set_age(self, age):
        self.age = age


class Man(Human):
    def set_name(self, name):
        if "XX" in name:
            self.name = name.replace("XX", " 모 씨")
            print("Man Class")

    def prt(self):
        print("이름은 " + self.name + "이고, 나이는 " + str(self.age))


a = Man()
a.set_name("김XX")
a.prt()
```

결과

```
Man Class
이름은 김 모 씨이고, 나이는 99
```

```
class Man(Human):
```

...

class 클래스 이름(상속 받을 클래스 이름):을 사용하여 부모 클래스로부터 속성을 상속받을 수 있습니다.

만약 상속 받는 클래스가 여러 개라면, class 클래스 이름(클래스1, 클래스2, ...):의 형태로 다중으로 상속받을 수 있습니다.

다중 상속시 상속 받을 클래스의 이름이 앞 쪽에 있는 클래스가 우선권을 얻습니다.

즉, 클래스1과 클래스2, 클래스3에 동일한 이름의 메서드가 있다면 클래스1의 메서드를 상속해옵니다.

부모 클래스에 정의된 내용을 그대로 사용할 수 있으며, 추가적인 기능(prt 메서드)을 포함할 수도 있습니다.

```
class Man(Human):
    def set_name(self, name):
        if "XX" in name:
            self.name = name.replace("XX", " 모 씨")
            print("Man Class")
```

만약 부모 클래스에 정의된 메서드의 이름과 동일한 이름으로 자식 클래스에 생성한다면, 부모 클래스의 메서드는 무시되고 자식 클래스의 메서드가 실행됩니다.

이를 메서드 오버라이딩(Method Overriding)이라 부릅니다.

즉, 자식 클래스에서 메서드를 재정의해 사용할 수 있습니다.

메서드를 재정의하지 않는다면 부모 클래스의 메서드를 호출합니다.

```
class Man(Human):
    def set_name(self, name):
        if "XX" in name:
            self.name = name.replace("XX", " 모 씨")
            print("Man Class")
            super().set_name(self.name + "(가명)")
```

super().메서드 이름을 사용해 부모 클래스의 메서드를 호출할 수 있습니다.

또한, super().속성으로 부모 클래스의 name이나 age를 접근할 수도 있습니다.

메서드 오버라이딩을 하더라도, 부모 클래스에 접근해 재정의하기 전의 메서드나 속성을 사용할 수 있습니다.

PIP(Package Manager)

PIP(Package Manager)는 Python에서 작성된 패키지 소프트웨어를 설치하는데 사용합니다.

명령 줄 인터페이스만으로도 손쉽게 소프트웨어 패키지를 설치할 수 있습니다.

Python, Conda로 PIP 설치하기
PIP를 찾을 수 없는 경우 해결 방법
환경변수 편집 없이 PIP 설치하기
첫 번째 방법으로 문제없이 설치가 됐다면 두 번째, 세 번째 방법은 진행하지 않으셔도 됩니다.

리눅스나 맥 OS의 운영체제는 쉘(Shell)에서 설치합니다.

PIP 실행 및 업그레이드 (1)

pip install --upgrade pip
pip install numpy
명령 프롬프트 창(cmd)를 실행시킨 다음, pip install --upgrade pip로 패키지 매니저를 최신 버전으로 업그레이드합니다.

업그레이드가 모두 완료됐다면, pip install numpy로 최신 버전의 Numpy 패키지를 설치할 수 있습니다.

최신 버전이 아닌, 특정 버전을 받아야한다면 pip install numpy==Version으로 특정 버전을 내려받을 수 있습니다.

이미 Numpy 패키지가 설치되어있지만 이전 버전이라면, pip install --upgrade numpy를 통해 최신 버전으로 업그레이드 할 수 있습니다.

위 방법으로 문제없이 설치된 경우, 아래의 방법을 진행하지 않으셔도 됩니다.

환경 변수 등록 (2)

Python이 정상적으로 설치되었지만, 명령어가 작동하지 않는다면 환경 변수가 등록되어 있지

않거나, 환경 변수가 올바른 경로가 아닐 때 인식하지 못합니다.

이 경우, Python을 설치시 환경 변수 등록 체크 박스를 체크한 다음 설치하거나, 환경 변수를 직접 추가하는 방법이 있습니다.

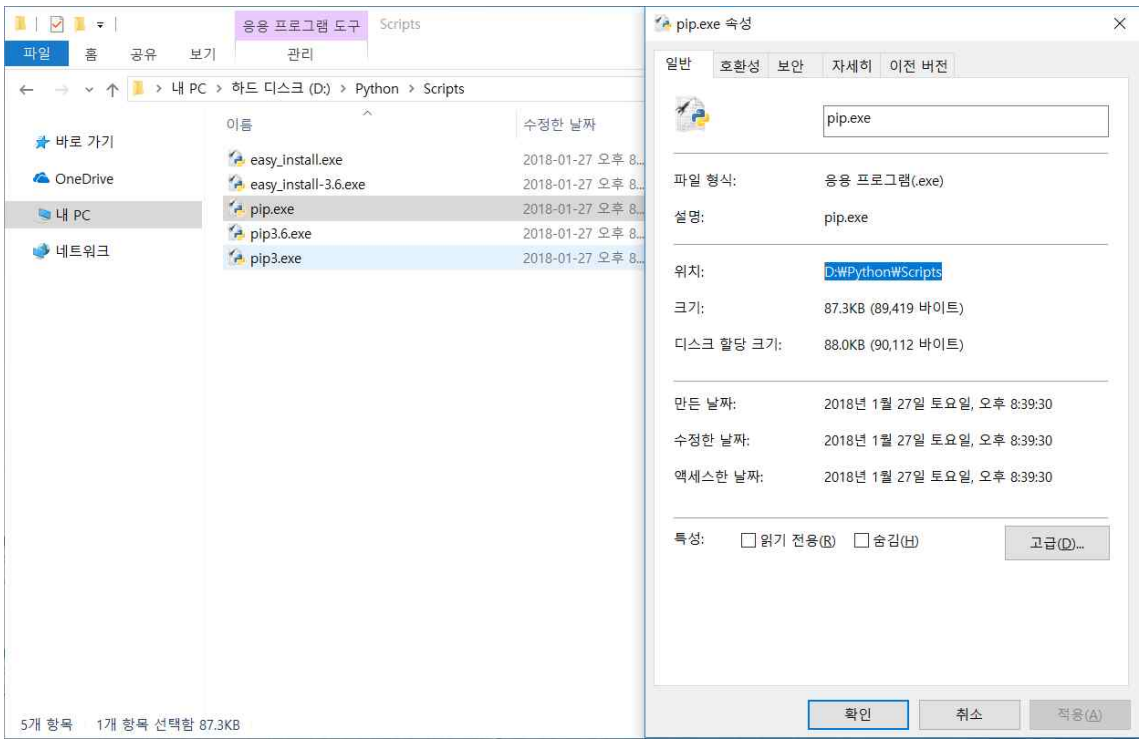
환경 변수를 추가하는 방법은 내 컴퓨터(내 PC) → 우 클릭 → 속성 → 좌측 탭 → 고급 시스템 설정 → 고급 탭 → 환경 변수 버튼 → 사용자 변수 탭 → Path 편집 → 새로 만들기 → PIP 경로 붙여넣기의 과정으로 등록할 수 있습니다.

Python이 저장된 경로\Python\Scripts를 붙여넣으시면 됩니다.

이 후, Python, Conda로 PIP 설치하기을 진행합니다.

위 방법으로 문제없이 설치된 경우, 아래의 방법을 진행하지 않으셔도 됩니다.

PIP 실행 및 업그레이드 (3)

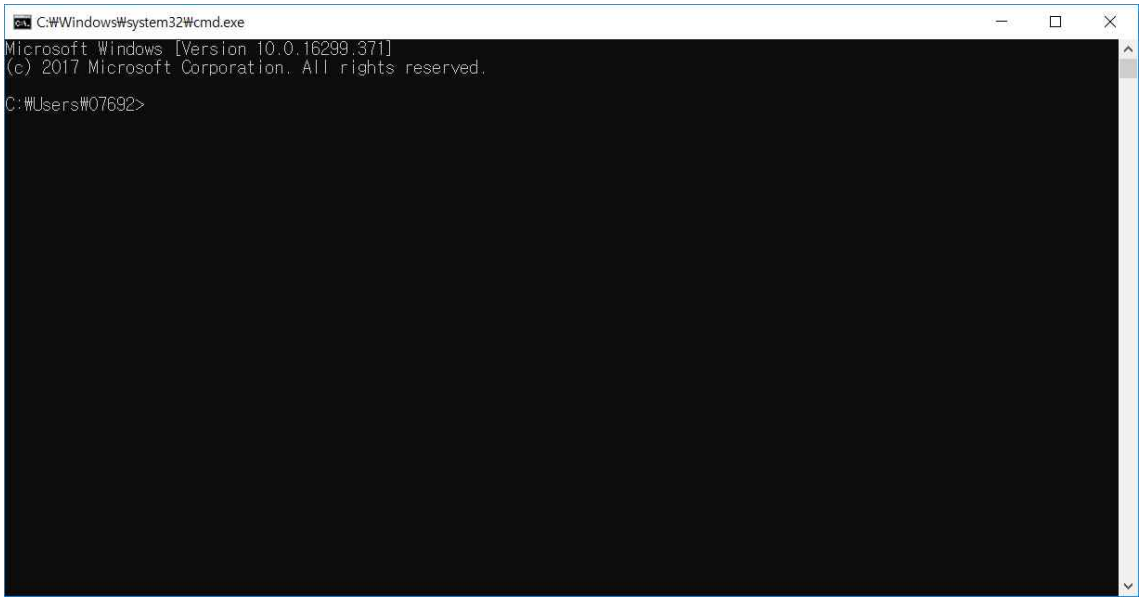


Python이 저장된 경로\Python\Scripts에 pip.exe, pip3.exe 등이 저장되어 있습니다.

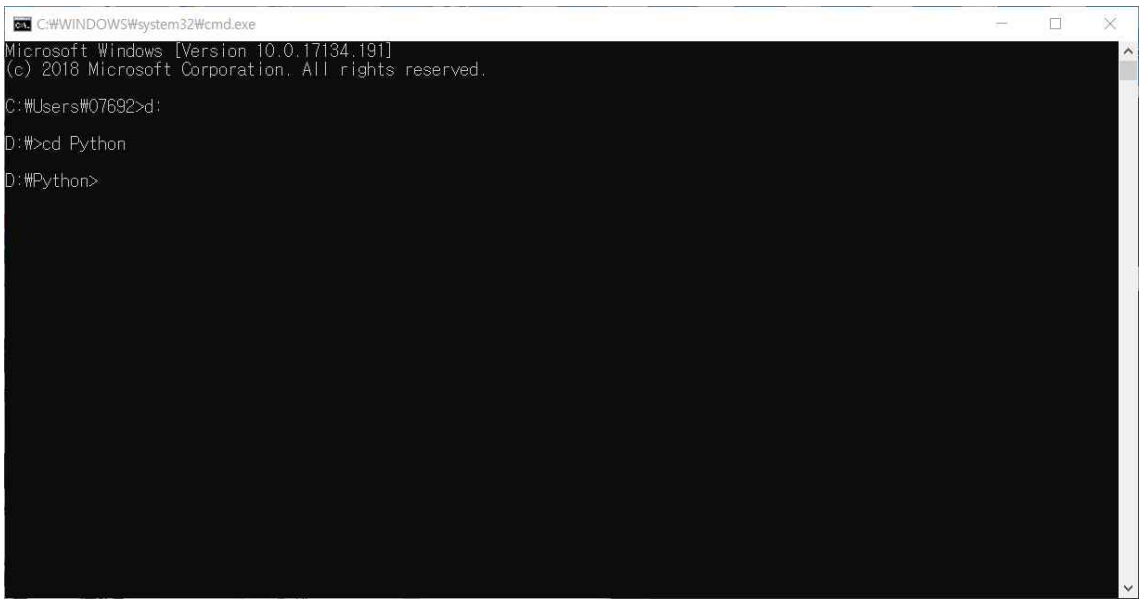
만약, Anaconda로 설치하셨다면 설치 경로\Anaconda3\lib\site-packages에 pip가 설치되

어 있습니다.

이 pip를 불러와 패키지 소프트웨어를 설치할 수 있습니다.

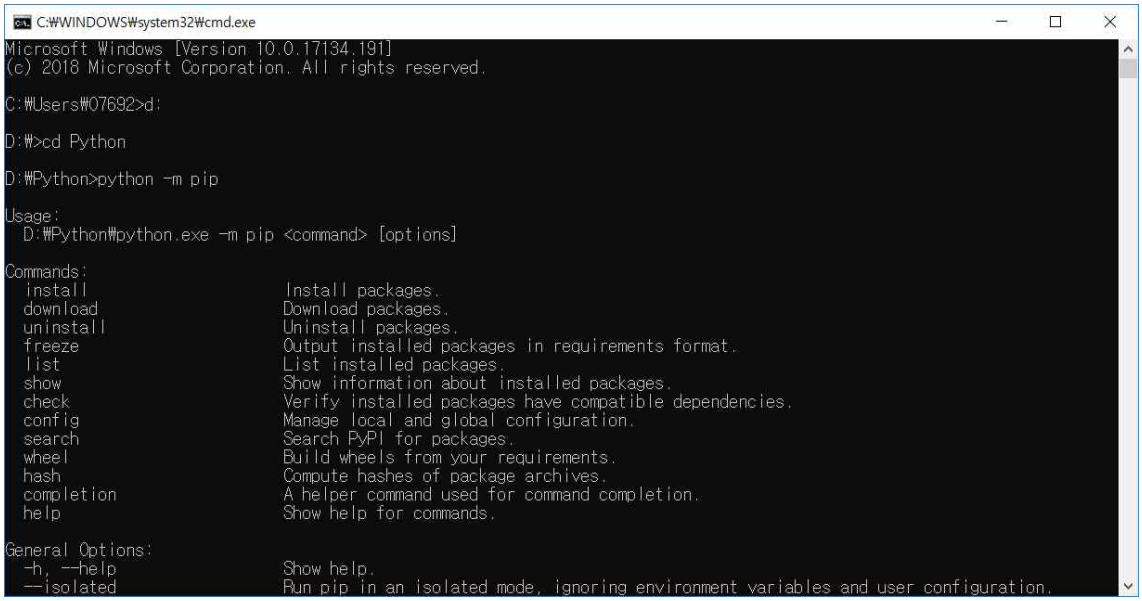


명령 프롬프트 (cmd.exe)를 실행시킵니다.



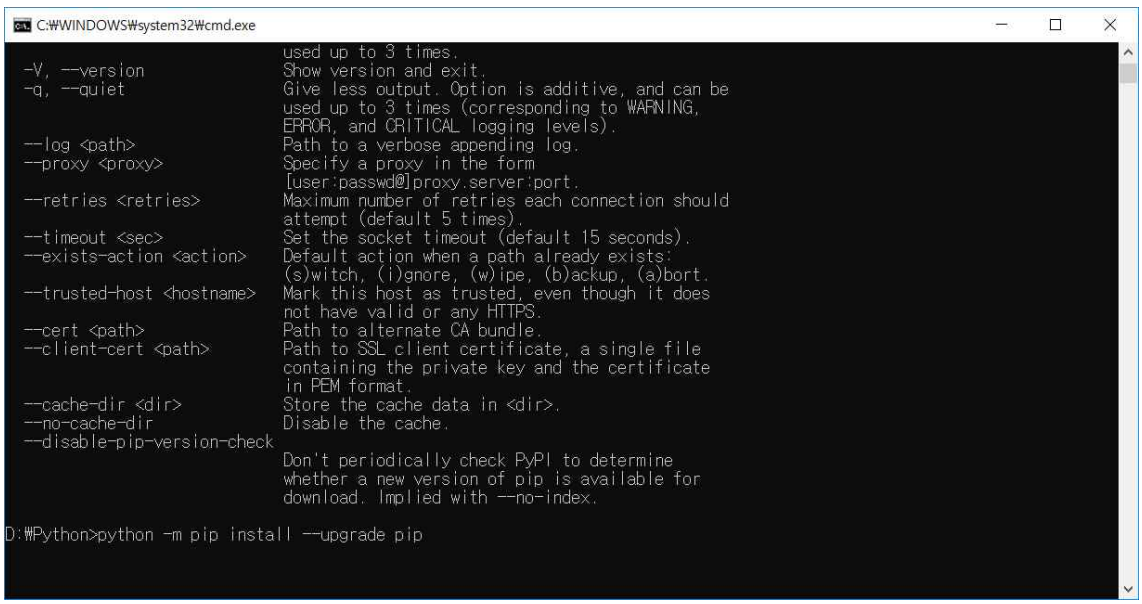
Python이 저장된 경로가 D 드라이브(d:)이므로 이동합니다.

그 후, cd python을 통하여 Python 폴더로 경로를 변경합니다.



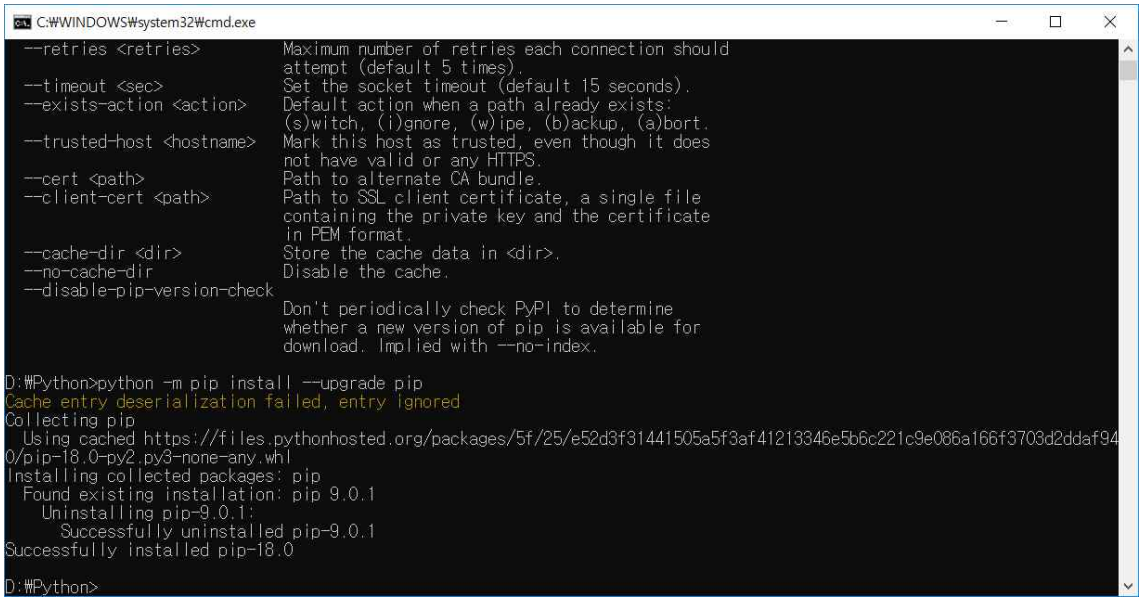
python -m pip를 통하여 pip가 정상적으로 작동하는지 확인합니다.

Commands의 목록을 통하여 pip를 제어할 수 있습니다.



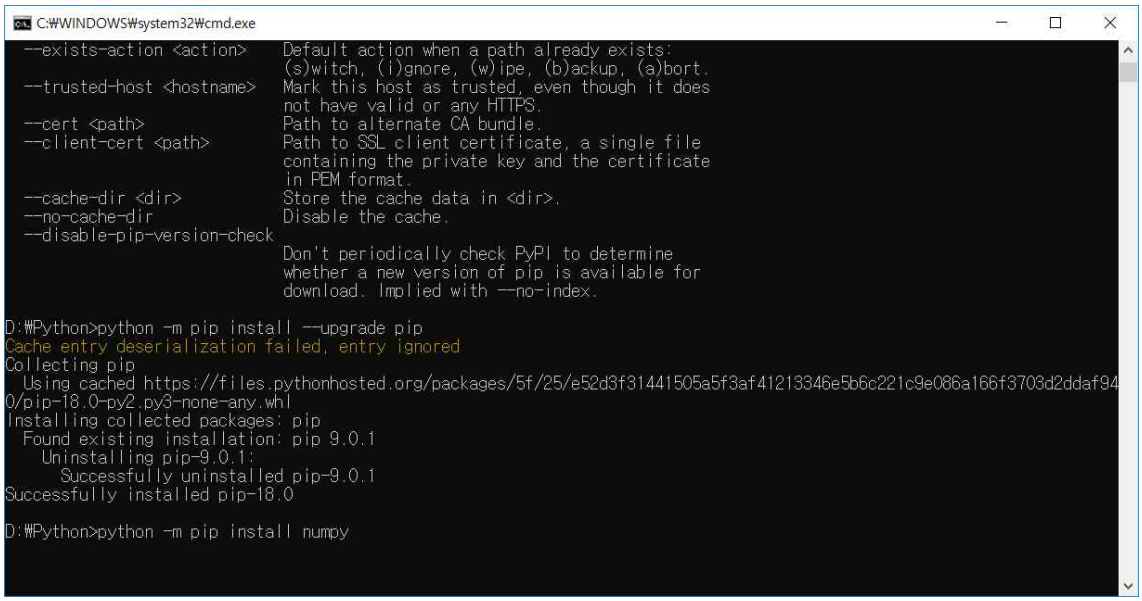
먼저 pip의 버전을 업그레이드합니다.

python -m pip install --upgrade pip를 통하여 버전을 업그레이드합니다.



Sucessfully installed pip-버전이 출력된다면 정상적으로 버전이 업그레이드 되었습니다.

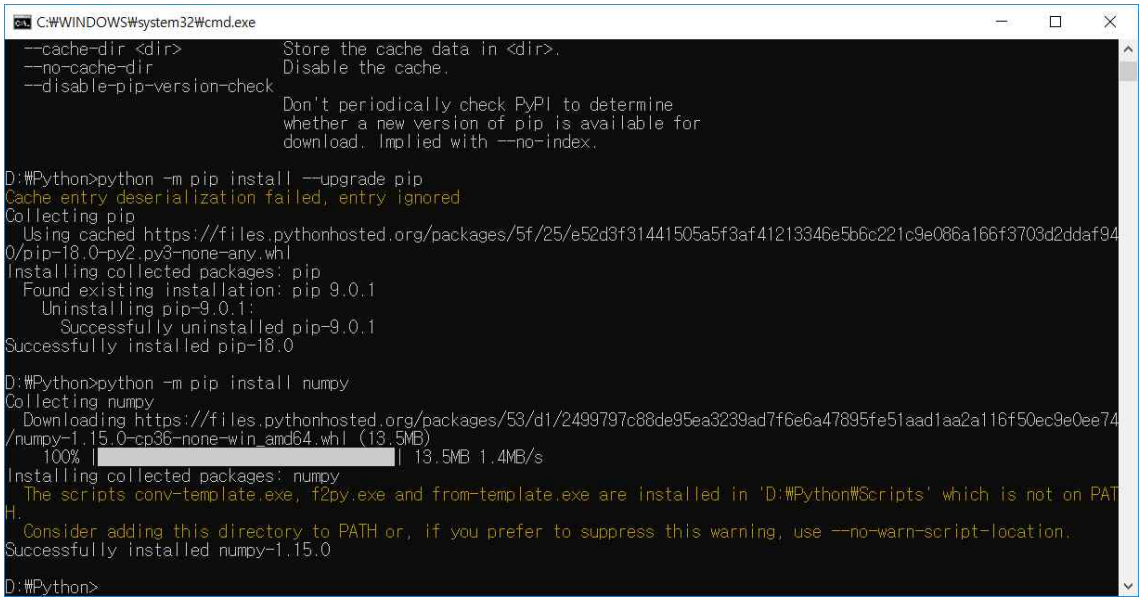
PIP 패키지 소프트웨어 설치



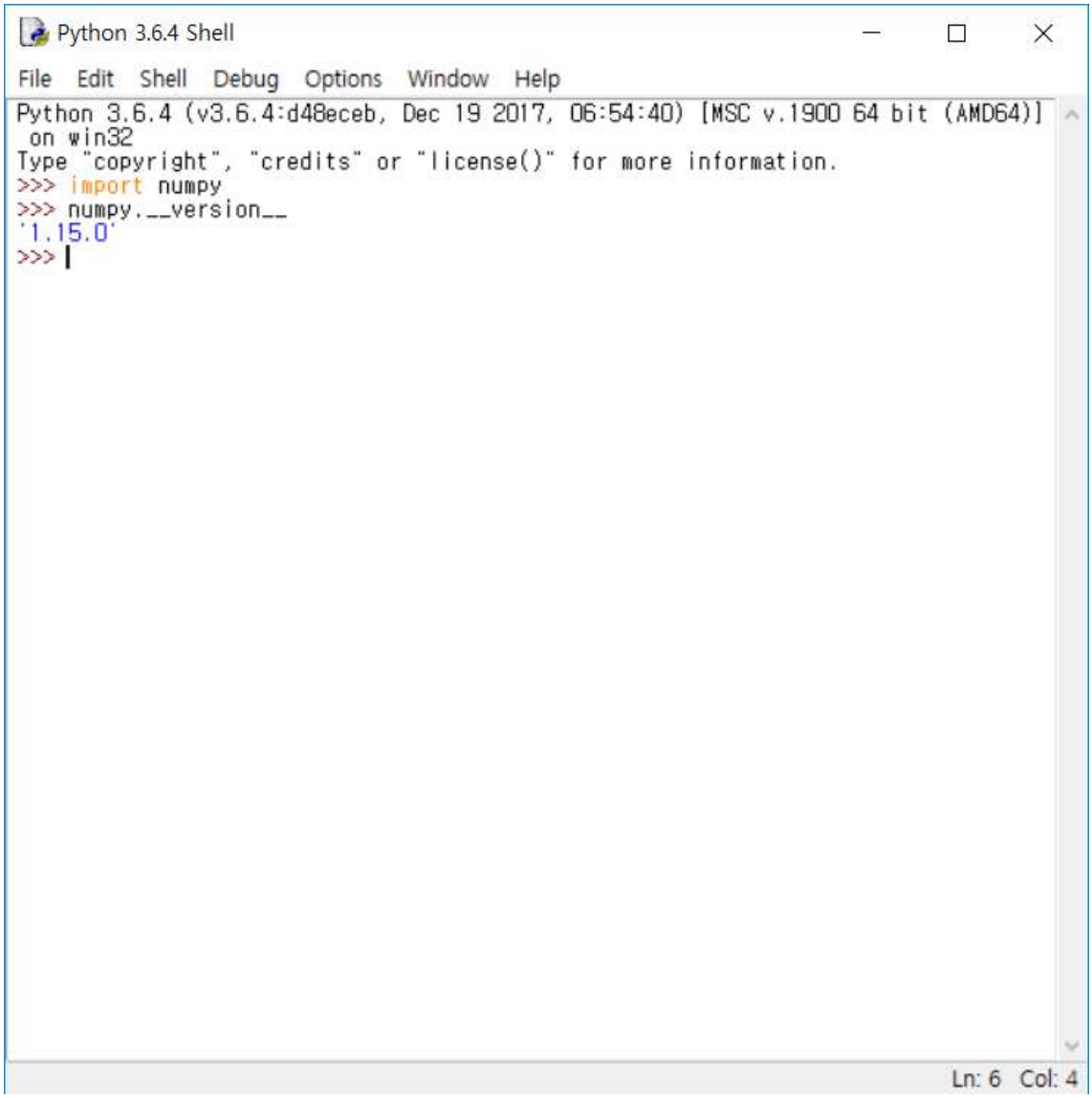
numpy 라이브러리를 설치해보도록 하겠습니다.

python -m pip install numpy를 통하여 numpy 라이브러리를 설치할 수 있습니다.

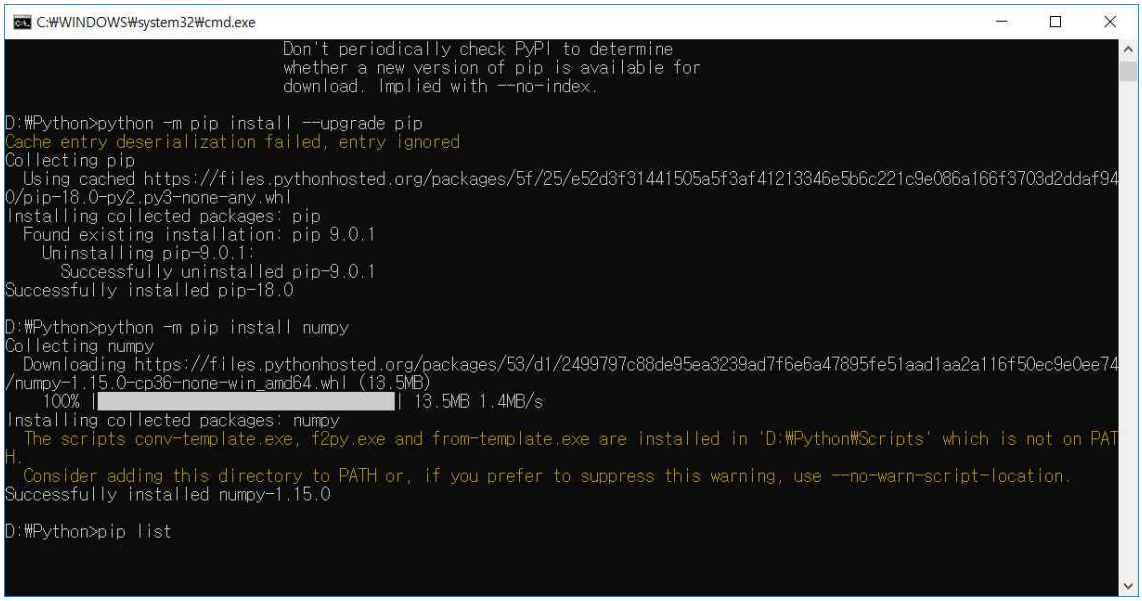
Tip : OpenCV 라이브러리는 opencv-python으로 설치할 수 있습니다.



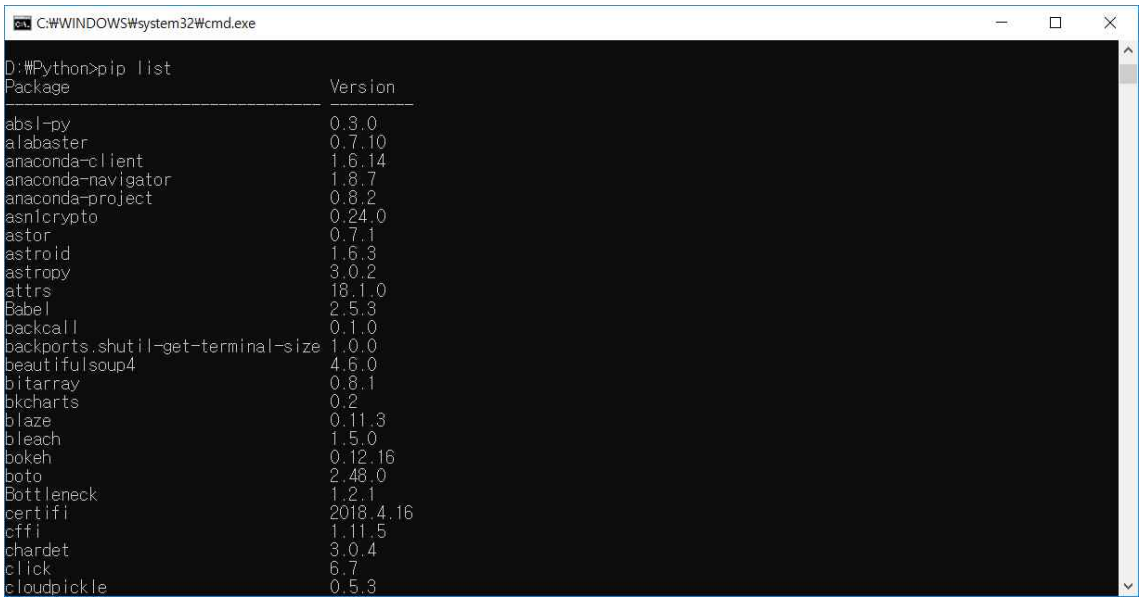
Sucessfully installed numpy-버전이 출력된다면 정상적으로 설치되었습니다.



Python Shell을 실행 후, import numpy와 numpy.__version__을 통하여 numpy의 정상 설치 유/무와 버전을 확인할 수 있습니다.



또는 pip list를 통하여 설치된 패키지와 버전을 확인할 수 있습니다.



다음과 같이 설치된 모든 라이브러리를 확인하며 버전까지 확인할 수 있습니다.

열거형(enumerate)

열거형(enumerate)은 반복자(iterator)를 지원하는 객체를 색인 값과 요소 값을 동시에 반환하는 객체입니다.

색인 값을 활용하기 위해 range(len(n))의 형태로 사용하지 않고 enumerate()로 활용이 가능합니다.

반복문 적용

```
data = ["Python", "076923", "YUNDAEHEE"]

for i, datum in enumerate(data):
    print(i, datum)
```

결과

```
0 Python
1 076923
2 YUNDAEHEE
```

for 색인 값, 요소 값 in enumerate(반복자 지원 객체):의 형태로 구성됩니다.

반복문을 for i in range(len(data))로 구성하지 않고 열거형으로 반복문을 구성합니다.

색인 값이 앞에 오며, 요소 값이 뒤에 오는 구조가 됩니다.

다중 색인 반환

```
data1 = ["Python", "076923", "YUNDAEHEE", "X"]
data2 = ["파이썬", "076923", "윤대희"]

for i, (datum1, datum2) in enumerate(zip(data1, data2)):
    print(i, datum1, datum2)
```

결과

```
0 Python 파이썬
1 076923 076923
2 YUNDAEHEE 윤대희
```

for 색인 값, (요소 값1, 요소 값2) in enumerate((반복자 지원 객체1, 반복자 지원 객체2)):
의 형태로 구성됩니다.

집 함수로 두 개 이상의 반복자 지원 객체를 병합해 활용할 수도 있습니다.

집 함수의 특성상 요소의 길이가 다르다면, 더 작은 길이를 갖는 반복횟수로 설정됩니다.

4개의 요소를 갖는 객체와 3개의 요소를 갖는 객체를 열거자로 묶어 반복한다면 소괄호(())로 묶습니다.

요소 값도 소괄호(())로 묶어 반환해야 합니다.

두 개 이상의 객체가 색인 값과 함께 반복되어 요소 값이 반환됩니다.

오류 발생(raise)

오류 발생(raise)은 프로그램이 허용된 범위 안에서 작동하지 않을 때 강제로 오류를 발생시키는 역할을 합니다.

코드상에서 문제는 없지만 알고리즘상에서 문제가 발생하거나 허용하지 않는 경우 강제로 오류를 발생시킵니다.

강제 오류 발생

```
fruit = ["apple", "banana", "grape", "watermelon"]
print(fruit)

while True:
    index = input("과일 색인 값 선택: ")

    try:
        index = int(index)

        if index == 1:
            raise NameError

        else:
            print(fruit[index])

    except NameError:
        print("바나나는 불가능합니다.")

    except IndexError:
        print("해당 색인은 호출할 수 없습니다.")

    except ValueError:
        print("소수점은 사용할 수 없습니다.")

    print("------------")
```

결과

```
['apple', 'banana', 'grape', 'watermelon']
과일 색인 값 선택: 0
apple
————
```

```
과일 색인 값 선택: 1
바나나는 불가능합니다.
----
과일 색인 값 선택: 5
해당 색인은 호출할 수 없습니다.
----
과일 색인 값 선택: 9.5
소수점은 사용할 수 없습니다.
----
```

raise 오류를 통해 강제로 오류를 발생시킬 수 있습니다.

index가 1일 때 정상적으로 출력되는 것이 맞지만 허용하지 않는다면 raise를 통해 강제로 오류를 발생시킵니다.

1을 입력했을 때 NameError가 발생하며, except 구문으로 넘어갑니다.

사용자 정의 오류 발생

```
class AdminError(Exception):
    pass


while True:

    admin = input("관리자 계정 입력: ")

    try:
        if admin != "yundaehee":
            raise AdminError()
        else:
            print("관리자 계정입니다.")

    except AdminError:
        print("관리자 계정이 아닙니다.")

    print("------------")
```

결과

```
관리자 계정 입력: admin
관리자 계정이 아닙니다.
----
관리자 계정 입력: yundaehee
관리자 계정입니다.
----
```

클래스가 Exception을 상속 받는다면 사용자 정의 오류를 발생시킬 수 있습니다.

입력된 값이 yundaehee가 아니라면, raise 구문으로 넘어가게 되고, AdminError 클래스로 오류가 발생합니다.

except 구문에서 사용자 정의 오류를 예외 처리할 수 있습니다.

쓰레드(Thread)

쓰레드(Thread)는 프로세스에서 실행되는 흐름의 단위를 의미합니다.

일반적으로 프로그램은 하나의 쓰레드를 갖고서 알고리즘이 진행됩니다.

만약, 한 번에 두 가지 이상의 문제를 해결하기 위해선 두 개 이상의 쓰레드를 구동해야합니다.

이러한 실행 방식을 멀티쓰레드(Multi-Thread)라 부릅니다.

서브 쓰레드(Sub-Thread)

```
import threading


def first_task(data):
    for i in data:
        print("first_task :", i)


def second_task(data1, data2):
    for i, j in zip(data1, data2):
        print("second_task :", i, j)


task1 = threading.Thread(target=first_task, args=(range(5),))
task2 = threading.Thread(target=second_task, args=(range(5), range(5)))

print("START")
task1.start()
task2.start()
print("END")
```

결과

```
START
first_task : 0
first_task : 1
second_task : 0 0
first_task :ENDsecond_task : 2
first_task : 3
```

```
first_task :
4
1 1
second_task : 2 2
second_task : 3 3
second_task : 4 4
```

멀티 쓰레드를 구성하기 위해선 서브 쓰레드를 구성해야 합니다.

서브 쓰레드를 사용하면 기존의 직렬 구조에서 병렬 구조로 연산이 가능합니다.

쓰레드를 사용하기 위해선 threading 모듈을 포함시킵니다.

쓰레드는 threading 모듈의 Thread 클래스로 실행시킬 수 있습니다.

쓰레드 클래스는 threading.Thread(target=함수, args=(함수의 매개변수))입니다.

args는 튜플만 지원하며, 하나의 매개변수를 전달하는 경우, 콤마(,)를 추가해 튜플로 설정합니다.

출력 결과에서 확인할 수 있듯이 print("END") 구문이 먼저 출력됩니다.

또한 first_task 함수와 second_task가 별도의 쓰레드에서 연산되는 것을 알 수 있습니다.

데몬 쓰레드

```
import threading


def first_task(data):
    for i in data:
        print("first_task :", i)


def second_task(data1, data2):
    for i, j in zip(data1, data2):
        print("second_task :", i, j)
```

```
    task1 = threading.Thread(target=first_task, args=(range(5000),))
    task2 = threading.Thread(target=second_task, args=(range(5), range(5)))

    task1.daemon = True
    task2.daemon = True

    print("START")
    task1.start()
    task2.start()
    print("END")
```

결과

```
    START
    first_task : 0
    first_task : 1
    first_task : 2
    first_task : 3END
    second_task : 0 0
    first_task :second_task : 1 14
    second_task : 2 2
    second_task : 3 first_task :3 5
    first_task :second_task : 6 4 4
    first_task : 7
    first_task : 8
    first_task : 9
    first_task : 10
    first_task : 11
    first_task : 12
    first_task : 13
    ...
    first_task : 752
    first_task : 753
    first_task : 754
    first_task : 755
    first_task :
```

데몬 쓰레드(Daemon-Thread)란 메인 쓰레드가 종료되면 같이 종료되는 쓰레드를 의미합니다.

서브 쓰레드는 메인 쓰레드가 종료되도 연산을 끝까지 진행합니다.

하지만, 데몬 쓰레드는 메인 쓰레드가 종료되면 연산을 중단합니다.

쓰레드의 daemon 멤버의 값을 True로 설정하면 데몬 쓰레드가 됩니다.

출력 결과에서 확인할 수 있듯이 first_task : 756를 출력하다 종료합니다.

print("END")구문을 출력됐지만, 조금 더 결과가 출력되는 이유는 print("END") 이후에 메인 쓰레드가 종료되는 시간까지 포함됩니다.

그러므로, END 출력 이후, 메인 쓰레드가 종료될 때 까지 연산을 진행합니다.

Tip : daemon의 기본값은 False입니다.

인자(Argument)

인자(Argument)는 매개변수(Parameter)에 전달되는 실질적인 값(value)를 의미합니다.

함수에 정의된 값을 전달하는 것이 인자가 됩니다.

def func(a, b)에서 a, b는 매개변수가 되며, a와 b에 전달하는 값이 인자입니다.

Python에서는 사전에 정의되지 않은 여러 개의 인자를 전달할 수 있습니다.

```
*args(Tuple)
def func(*args):
    print("Type:", type(args))
    print("Lenght:", len(args))
    print(args)


func([1, 2], 3, 4, (5))
```

결과

```
Type: <class 'tuple'>
Lenght: 4
([1, 2], 3, 4, 5)
```

매개변수를 할당할 때 *args를 사용한다면, 여러 개의 인자를 받아 처리할 수 있습니다.

args는 argument의 약어로, 매개변수의 이름을 *args가 아닌, 다른 이름으로도 할당이 가능합니다.

매개변수의 이름에 와일드카드(*)가 작성되면, 입력되는 인자들을 튜플로 처리합니다.

입력된 인자들의 순서에 따라 args 변수에 값이 할당됩니다.

```
**kwargs(Dictionary)
def func(**kwargs):
    print("Type:", type(kwargs))
    print("Lenght:", len(kwargs))
    print(kwargs)


func(a=1, b=2, c=3)
```

결과

```
Type: <class 'dict'>
Lenght: 3
{'a': 1, 'b': 2, 'c': 3}
```

매개변수를 할당할 때 **kwargs를 사용한다면, 여러 개의 인자를 받아 처리할 수 있습니다.

kwargs는 keyword argument의 약어로, 매개변수의 이름을 **kwargs가 아닌, 다른 이름으로도 할당이 가능합니다.

매개변수의 이름에 와일드카드(*)가 두 번 작성되면, 입력되는 인자들을 사전으로 처리합니다.

입력된 인자들의 키 값에 따라 kwargs 변수에 값이 할당됩니다.

```
*args(Tuple), **kwargs(Dictionary) 혼용
def func(*data, **method):
    num = sum(data) * method["scale"]
    print(num, method["unit"] + "입니다.")


func(3, 4, 5, scale=10, unit="개")
```

결과

```
120 개입니다.
```

*args와 **kwargs를 동시에 사용한다면, *args, **kwargs 순서로 사용합니다.

키(Key) 값이 할당되지 않은 인자는 *args로 처리하며, 할당된 인자는 **kwargs로 처리합니다.

기본 인자가 포함되는 경우 다음과 같이 사용할 수 있습니다.

```
def func(*data, message, **method):
    print(message)

    num = sum(data) * method["scale"]
    print(num, method["unit"] + "입니다.")
```

```
    func(3, 4, 5, message="계산된 값입니다.", scale=10, unit="개")
```
결과
```
    계산된 값입니다.
    120 개입니다.
```

위치 인자(Positional Argument)는 항상 **kwargs 보다 앞에 있어야합니다. 즉, **kwargs 가 가장 마지막에 있어야합니다.

위치 인자는 항상 키(Key)를 갖고 있기 때문에, *args보다 뒤에 올 수 있습니다.

단, 위치 인자가 *args보다 뒤에 있을 경우, 키(Key)를 명시해야합니다.

위치 인자가 *args보다 앞에 있을 경우, 키(Key)를 명시하지 않아도됩니다.

위치 인자가 앞에 포함되는 경우 다음과 같이 사용할 수 있습니다.

```
    def func(message1, message2, *data, **method):
        print(message1)
        print(message2)

        num = sum(data) * method["scale"]
        print(num, method["unit"] + "입니다.")

    func("계산된 값입니다.", "값이 10배 커집니다.", 3, 4, 5, scale=10, unit="개")
```
결과
```
    계산된 값입니다.
    값이 10배 커집니다.
    120 개입니다.
```

위치 인자가 앞에 포함되면, 매개변수의 순서대로 값이 할당됩니다.

*args(Tuple)와 **kwargs(Dictionary)를 혼용해 사용하는 경우 매개변수의 할당 순서가 중요합니다.

매개변수의 개수와 일치하지 않는 경우, message2에 3이 할당되어 함수가 잘못된 결과를 반환할 수도 있습니다.

키워드 인자화(Transform Keyword Argument)

키워드 인자화(Transform Keyword Argument)는 위치 인자(Positional Argument)를 키워드 인자처럼 사용하도록 강제하는 방법입니다.

**kwargs처럼 매개변수의 이름을 명확히 할당해야 함수를 사용할 수 있습니다.

Keyword Argument (1)

```
def func(name, *, value1, value2):
    total = value1 + value2
    print(name + "는", total, "입니다.")


func("Plus", value1=2, value2=3)
```

결과

```
Plus는 5 입니다.
```

매개변수를 선언할 때 와일드카드(*)가 도중에 할당된다면, 이후의 인자들은 키워드 인자가 됩니다.

함수의 value1과 value2는 함수를 사용할 때, 명시적으로 할당되어야 합니다.

인자의 사용 유/무를 강제하는 것이 아닌, 인자를 전달할 때 키워드로 전달하도록 강제합니다.

만약, 기본값이 할당되어 있다면 필수로 할당하지 않아도 됩니다.

또한, 인자를 전달할 때 위치 인자(Positional Argument)를 사용하듯이 순서를 지키지 않아도 됩니다.

위의 사항을 다음과 같이 사용할 수 있습니다.

Keyword Argument (2)

```
def func(name, *, value1, value2=3, value3):
    total = value1 + value2 + value3
    print(name + "는", total, "입니다.")
```

```
    func("Plus", value3=2, value1=3)
```
결과
```
    Plus는 8 입니다.
```

위와 같이 인자의 순서를 무시해도 됩니다.

기본값이 할당되어 있다면 필수적으로 값을 입력하지 않아도됩니다.

함수 주석(Function Annotations)

함수 주석(Function Annotations)은 함수의 매개변수와 반환값에 주석(Annotations)을 작성합니다.

함수에 명시적으로 주석을 작성하는 것이므로 실제 코드에 포함됩니다.

단, 주석이므로 강제성은 없어 무시하고 사용할 수 있습니다.

함수 주석 작성

```
def func(a: str, b: float = 3.5) -> int:
    return a + b


value = func(3)
print(value)
```

결과

```
6.5
```

함수에 직접 주석을 작성할 수 있습니다.

콜론(:)을 사용해 매개변수에 할당되어야 하는 형식을 할당합니다.

->(화살표)를 사용해 반환값에 할당되어야 하는 형식을 할당합니다.

매개변수에 기본값을 할당하는 경우, 변수: 주석 = 기본값의 형태로 할당합니다.

매개변수에 할당된 형식을 맞추지 않아도 되며, 반환되는 값의 형식이 변환되어 반환되지 않습니다.

매직 메서드(Magic Method)

매직 메서드(Magic Method)는 미리 정의되어 있는 메서드들을 재정의하여 클래스를 활용할 수 있도록 변경합니다.

내장 함수들이 처리하는 연산을 변경해 사용자 정의 클래스나 함수 등을 효율적으로 사용할 수 있습니다.

Under Score(_)를 두 번 사용해 매직 메서드를 정의할 수 있습니다.

인스턴스(Instance) 정의

```
class Daeheeyun:
    def __new__(cls, *args, **kwargs):
        print("인스턴스 할당")
        return super(Daeheeyun, cls).__new__(cls)

    def __init__(self, site="076923"):
        print("인스턴스 초기화")
        self.site = site
        self.link = site + ".github.io"

    def __call__(self, protocol=True):
        print("인스턴스 호출")
        if protocol == True:
            return "https://" + self.link
        else:
            return "http://" + self.link

    def __del__(self):
        print("인스턴스 소멸")


instance = Daeheeyun()
print(instance.link)
print(instance(False))
del instance
```

결과

```
인스턴스 할당
인스턴스 초기화
```

076923.github.io
인스턴스 호출
http://076923.github.io
인스턴스 소멸

__new__ : 할당 메서드

새로운 인스턴스를 만들기 위해 가장 먼저 호출되는 메서드입니다.

__new__에서 인스턴스를 반환하지 않는다면 __init__은 실행되지 않습니다.

즉, __new__ 메서드가 __init__ 메서드를 호출합니다.

super().__new__(cls[, ...])의 형태로 슈퍼 클래스 호출해서 반환합니다.

__new__는 일반적으로 사용되지 않으며, 불변형(int, str, tuple) 등의 서브 클래스에서 인스턴스 생성을 커스터마이징할 수 있도록 하는 데 사용됩니다.

__init__ : 초기화 메서드

새로운 인스턴스를 만들 때 사용될 인자들을 선언하는 메서드입니다.

할당된 인자들을 선언해서 사용하며, 일반적으로 매개변수를 할당받을 때 동일한 이름에 self를 추가해 동일한 명칭으로 사용합니다.

__call__ : 호출 메서드

인스턴스가 함수로 호출될 때 실행되는 메서드입니다.

__call__ 메서드가 존재하면, 호출 가능한 형식(callable type)이 되며, 함수처럼 사용할 수 있습니다.

__call__은 인스턴스를 생성한 다음 값을 할당할 수 있습니다.

반대로 __init__은 인스턴스를 생성할 때 값을 할당할 수 있습니다.

__del__ : 소멸 메서드

인스턴스가 제거될 때 호출되는 메서드입니다.

소멸자라 부르며, del 등을 통해 간접적으로 호출될 수 있습니다.

인스턴스가 삭제되기 전에 호출됩니다. 정확하게는 참조 횟수(reference count)가 0이 될 때 호출됩니다.

객체가 종료될 때 사용되는 메서드입니다.

인스턴스(Instance) 표현

```
class Daeheeyun:
    def __init__(self, site="076923"):
        self.site = site
        self.link = site + ".github.io"

    def __repr__(self):
        return str({"site": self.site, "link": self.link})

    def __str__(self):
        return "Daeheeyun(site = " + self.site + ", link = " + self.link + ")"

    def __format__(self, format_spec):
        return format(self.link, format_spec)

    def __bytes__(self):
        return str.encode(self.link)


instance = Daeheeyun()
print(repr(instance))
print(str(instance))
print(format(instance, ">30"))
print(bytes(instance))
```

결과

```
{'site': '076923', 'link': '076923.github.io'}
Daeheeyun(site = 076923, link = 076923.github.io)
                  076923.github.io
b'076923.github.io'
```

__repr__ : 형식적인 문자열 메서드

repr()을 호출해 형식적인(official) 문자열 표현에 사용됩니다.

가능한 올바른 표현식을 사용하며, 디버깅을 위해 주로 사용합니다.

명확한 정보를 담아 활용하며, 문자열 형식으로 반환해야합니다.

__str__ : 비형식적인 문자열 메서드

str()을 호출해 비형식적인(informal) 문자열 표현에 사용됩니다.

repr()과는 다르게 사용자가 이해하기 쉬운 형태로 표현할 때 사용합니다.

repr과 str의 차이점을 numpy 배열로 예를 든다면 다음과 같습니다.

```
arr = np.array([1, 2, 3])
print(repr(arr))
print(str(arr))
```

결과

```
array([1, 2, 3])
[1 2 3]
```

repr()을 통해 반환된 문자열은 np.array(repr(arr))로도 바로 변환해 사용이 가능합니다.

하지만, str()을 통해 반환된 문자열은 활용할 수 없습니다.

__format__ : 포맷 형식

format()을 호출해 포맷팅(formatting)할 때 사용합니다.

포맷팅의 format_spec은 format()에서 지원하지 않는 사용자 임의의 형식도 생성할 수 있습니다.

예제의 >30은 우측 정렬, 30 너비 지정입니다.

__bytes__ : 바이트 형식

bytes()를 호출해 바이트 문자열로 변환할 때 사용합니다.

접두사 b가 추가되며, 바이트 문자열로 처리됩니다.

매직 메서드(Magic Method)

매직 메서드(Magic Method)는 미리 정의되어 있는 메서드들을 재정의하여 클래스를 활용할 수 있도록 변경합니다.

내장 함수들이 처리하는 연산을 변경해 사용자 정의 클래스나 함수 등을 효율적으로 사용할 수 있습니다.

Underscore(_)를 두 번 사용해 매직 메서드를 정의할 수 있습니다.

Tip : 매직 메서드는 이중 밑줄(Double Underscore)를 사용해 정의합니다.

산술 연산 정의

```
class Daeheeyun:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return self.value + int(other)

    def __radd__(self, other):
        return self.value + other

    def __iadd__(self, other):
        return self.value + abs(other)


instance = Daeheeyun(5)
add = instance + 3.076923
radd = 3.076923 + instance
instance += -3.076923

print(add)
print(radd)
print(instance)
```

결과

```
8
8.076923
8.076923
```

정방향 연산자

정방향 연산자는 x + y에 대한 연산을 정의합니다.
인스턴스의 값이 x가 되며, 외부 값이 y가 됩니다.
예제는 y를 int로 변환 후, 연산합니다.

역방향 연산자

정방향 연산자는 y + x에 대한 연산을 정의합니다.

인스턴스의 값이 x가 되며, 외부 값이 y가 됩니다.

예제는 y를 변환 없이 연산합니다.

복합 대입 연산자

복합 대입 연산자는 x += y에 대한 연산을 정의합니다.

인스턴스의 값이 x가 되며, 외부 값이 y가 됩니다.

예제는 y를 절댓값으로 변환하고 연산합니다.

산술 연산자	의미	역방향	복합 대입
object.__add__(self, other)	object + other 연산 정의	__radd__	__iadd__
object.__sub__(self, other)	object - other 연산 정의	__rsub__	__isub__
object.__mul__(self, other)	object * other 연산 정의	__rmul__	__imul__
object.__matmul__(self, other)	object @ other 연산 정의	__rmatmul__	__imatmul__
object.__truediv__(self, other)	object / other 연산 정의	__rtruediv__	__itruediv__
object.__floordiv__(self, other)	object // other 연산 정의	__rfloordiv__	__ifloordiv__
object.__mod__(self, other)	object % other 연산 정의	__rmod__	__imod__
object.__pow__(self, other[, modulo])	pow() 연산 정의	__rpow__	__ipow__
object.__divmod__(self, other)	divmod() 연산 정의	__rdivmod__	__idivmod__
object.__round__(self)	round() 연산 정의	없음	없음
object.__trunc__(self)	math.trunc() 연산 정의	없음	없음
object.__floor__(self)	math.floor() 연산 정의	없음	없음
object.__ceil__(self)	math.ceil() 연산 정의	없음	없음

단항 연산 정의

```
class Daeheeyun:
    def __init__(self, value):
        self.value = value
```

```
        def __pos__(self):
            return self.value + 2

        def __neg__(self):
            return self.value - 2

        def __abs__(self):
            return -abs(self.value)

        def __invert__(self):
            return ~self.value


    instance = Daeheeyun(5)
    print(+instance)
    print(-instance)
    print(abs(instance))
    print(~instance)
```

결과

```
    7
    3
    -5
    -6
```

__pos__ : Positive

+object 연산자는 +x에 대한 연산을 정의합니다.

예제는 x를 양수를 곱하지 않고, +2 연산으로 변경합니다.

__neg__ : Negative

-object 연산자는 -x에 대한 연산을 정의합니다.
예제는 x를 음수를 곱하지 않고, -2 연산으로 변경합니다.

__abs__ : Absolute

abs() 연산자는 abs(x)에 대한 연산을 정의합니다.
예제는 x를 절대값으로 취하지 않고, 음수로 변경합니다.

__invert__ : Invert

~object 연산자는 ~x에 대한 연산을 정의합니다.
예제는 ~x를 변환 없이 연산합니다.

단항 연산자	의미
object.__pos__(self)	+object 연산 정의
object.__neg__(self)	-object 연산 정의
object.__abs__(self)	abs() 연산 정의
object.__invert__(self)	~object 연산 정의

비트 연산 정의

비트 연산자 중 not 연산자는 재정의할 수 없으며, __bool__ 연산자를 통해 변경할 수 있습니다.

또는 ~object 연산자를 재정의해 not 연산자로 활용할 수 있습니다.

비트 연산자	의미	역방향	복합 대입
object.__lshift__(self, other)	object « other 연산 정의	__rlshift__	__ilshift__
object.__rshift__(self, other)	object » other 연산 정의	__rrshift__	__irshift__
object.__and__(self, other)	object &other 연산 정의	__rand__	__iand__
object.__or__(self, other)	object | other 연산 정의	__ror__	__ior__
object.__xor__(self, other)	object ^ other 연산 정의	__rxor__	__ixor__

비교 연산 정의

비교 연산자는 역방향 연산자와 복합 대입 연산자는 존재하지 않습니다.

정방향 연산자만 정의할 수 있으며, 역방향으로 연산할 경우, 정방향으로 간주됩니다.

비교 연산자	의미
object.__lt__(self, other)	object <other 연산 정의
object.__le__(self, other)	object <= other 연산 정의
object.__eq__(self, other)	object == other 연산 정의
object.__ne__(self, other)	object != other 연산 정의
object.__gt__(self, other)	object >other 연산 정의
object.__ge__(self, other)	object >= other 연산 정의

형식 변환 정의

```
class Daeheeyun:
    def __init__(self, value):
        self.value = value

    def __index__(self):
```

```
        return 2


L = ["A", "B", "C", "D", "E"]
instance = Daeheeyun(100)
print(L[instance])
```

결과

```
C
```

형식 변환은 인스턴스의 형식을 변환할 때 사용됩니다.
그 중, __index__는 slice 연산을 진행할 때 할당되는 index를 정의합니다.
예제의 index의 값을 2로 정의하여, 리스트에서 C가 출력됩니다.

형식 변환	의미
object.__int__(self)	int() 연산 정의
object.__float__(self)	float() 연산 정의
object.__complex__(self)	complex() 연산 정의
object.__bool__(self)	bool() 연산 정의
object.__hash__(self)	hash() 연산 정의
object.__index__(self)	slice 연산의 index 정의

속성(Attribute)

속성(Attribute)은 클래스 내부에 포함돼 있는 메서드나 변수를 의미합니다.

Python에서 속성(Attribute)은 크게 클래스 속성과 인스턴스 속성으로 나뉩니다.

클래스 속성은 클래스 내부의 메서드 단계와 동일한 영역에 위치한 변수를 의미합니다.

클래스 속성에 접근할 경우, 모든 클래스에 동일하게 영향을 미칩니다.

인스턴스 속성은 self를 통해 할당된 인스턴스만의 변수를 의미합니다.

주로, __init__이나 메서드 내부에서 할당된 변수를 의미합니다.

Tip : self는 자기자신을 의미합니다. 즉, 인스턴스를 지칭합니다.

클래스 속성과 인스턴스 속성 정의

```
class 클래스명:
    클래스 속성 = 값

    def __init__(self, *args, *kwargs):
        self.인스턴스 속성 = 값

    …
```

클래스 속성

클래스 속성은 메서드와 동일 단계에 작성하게 됩니다.

self를 사용하지 않고 정의합니다. 그러므로 모든 클래스에 동일하게 영향을 미칩니다.

인스턴스가 생성될 때 초기화되지 않으므로, 클래스명.클래스 속성으로 참조가 가능합니다.

인스턴스 속성

인스턴스 속성은 인스턴스를 초기화할 때 생성됩니다.

self를 사용해 정의합니다. 그러므로 인스턴스 내에서만 영향을 미칩니다.

인스턴스가 생성될 때나 생성된 후 할당되므로, 인스턴스명.인스턴스 속성으로 참조가 가능합니다.

```
class Daeheeyun:

    class_value = 0

    def __init__(self):
        self.instance_value = 0

    def set_class_value(self):
        Daeheeyun.class_value = 10

    def set_instance_value(self):
        self.class_value = 20


instance1 = Daeheeyun()
instance2 = Daeheeyun()

print("--클래스 속성 변경--")
instance1.set_class_value()
print(instance1.class_value, instance2.class_value)

print("--인스턴스 속성 변경--")
instance1.set_instance_value()
print(instance1.class_value, instance2.class_value)

print("--속성(Attribute) 출력--")
print(instance1.__dict__)
print(instance2.__dict__)
```

결과

```
-클래스 속성 변경-
10 10
-인스턴스 속성 변경-
```

```
20 10
-속성(Attribute) 출력-
{'instance_value': 0, 'class_value': 20}
{'instance_value': 0}
```

Daeheeyun 클래스

instance1과 instance2를 생성해 instance1에 대해서만 작업을 진행합니다.

set_class_value 메서드는 클래스 속성을 변경하는 메서드입니다.

set_instance_value 메서드는 인스턴스 속성을 변경하는 메서드입니다.

set_class_value 메서드

instance1에 클래스 값을 변경하면, Daeheeyun에 대한 동일한 속성이 일괄 변경됩니다.

즉, instance2을 변경하지 않아도 클래스 자체의 값이 변경되어 instance2의 값도 변경됩니다.

set_instance_value 메서드

instance1에 인스턴스 값을 변경하면, instance1에 대한 속성만 변경됩니다.

메서드 내부에서 class_value를 인스턴스화합니다. 즉, instance2는 영향을 미치지 않습니다.

__dict__ 속성

__dict__는 현재 인스턴스에 할당된 인스턴스 속성만 출력합니다.

instance1의 인스턴스 속성은 instance_value와 인스턴스화 된 class_value입니다.

instance2의 인스턴스 속성은 instance_value입니다.

instance2에서 set_instance_value 메서드를 통해 class_value를 인스턴스화 하지 않아, 클래스 속성이 유지되기 때문입니다.

매직 메서드(Magic Method)

클래스의 속성을 사용할 때, 속성을 관리하는 메서드를 통해 속성이 정의되거나 할당됩니다.

이 메서드들을 통해 속성을 관리할 수 있습니다. 속성을 관리하는 매직 메서드를 재정의해 속성을 관리합니다.

__getattr__ : 존재하지 않는 속성 호출

```
class Daeheeyun:
    def __init__(self):
        self.value = 0

    def __getattr__(self, name):
        return name + "은 존재하지 않습니다."


instance = Daeheeyun()
print(instance.value)
print(instance.nothing)
```

결과

```
0
nothing은 존재하지 않습니다.
```

존재하지 않는 속성을 참조한다면, 일반적으로 AttributeError가 발생합니다.

하지만, __getattr__ 메서드를 정의한다면 존재하지 않는 속성을 호출할 때, __getattr__ 메서드가 실행됩니다.

이를 통해 오류를 발생시키지 않고, 속성을 새로 정의하거나 무시할 수 있습니다.

__getattr__ 메서드는 인스턴스의 다른 속성에 접근 할 수 없습니다.

__getattribute__ : 속성 호출

```
class Daeheeyun:
    def __init__(self):
        self.value = 0

    def __getattribute__(self, name):
        try:
            return super().__getattribute__(name)

        except AttributeError:
            value = "Empty"
            setattr(self, name, value)
            return value


instance = Daeheeyun()
print(instance.value)
print(instance.nothing)
```

결과

```
0
Empty
```

속성을 호출한다면, __getattribute__ 메서드가 실행됩니다.

어떠한 속성이나 메서드를 호출한다면 __getattribute__ 메서드를 거치게 됩니다.

__getattr__ 메서드는 존재하지 않는 속성만 호출되지만, __getattribute__는 모든 속성이 호출됩니다.

__getattribute__ 메서드를 재정의했다면, __getattr__는 호출되지 않습니다.

존재하지 않는 속성을 호출했다면, __getattribute__ 내부에서 AttributeError가 발생합니다.

Tip : getattr() 함수를 통해 __getattribute__를 호출할 수 있습니다. getattr(instance, "value")와 같이 사용합니다.

Tip : setattr() 함수는 속성을 정의하는 함수입니다. setattr(instance, name, value)와 같이 사용합니다.

__setattr__ : 속성 할당

```
class Daeheeyun:
    def __init__(self):
        self.value = 0

    def __setattr__(self, name, value):
        return super().__setattr__(name, value * 2)


instance = Daeheeyun()
instance.value = 10
instance.nothing = 30
print(instance.value)
print(instance.nothing)
```

결과

```
20
60
```

속성을 변경하거나 할당한다면 __setattr__을 통해 값이 할당됩니다.

주의사항으로는 __setattr__ 내부에서 속성을 변경하거나 할당한다면, 다시 __setattr__가 실행됩니다.

그러므로, 재귀 되어 무한루프에 빠지게 됩니다.

__setattr__ 메서드 내부에서는 속성을 변경하거나 할당하지 않습니다.

__delattr__ : 속성 제거

```
class Daeheeyun:
    def __init__(self):
        self.value = 0

    def __delattr__(self, name):
        print("제거 :", name)
        return super().__delattr__(name)
```

```
instance = Daeheeyun()
del instance.value
```

결과

제거 : value

속성을 제거하면 __delattr__가 호출됩니다.

del이나, delattr() 함수를 통해 속성을 제거할 때 실행됩니다.

return super().__delattr__(name) 방식 이외에도, return 없이 del self.__dict__[name]처럼 속성을 제거할 수 있습니다.

Tip : delattr() 함수는 속성을 제거하는 함수입니다. delattr(instance, "value")와 같이 사용합니다.

__slots__ : 속성 제한, __dir__ : 속성 보기

```
class Daeheeyun:

    __slots__ = ["value"]

    def __init__(self):
        self.value = 0

    def __dir__(self):
        return sorted(super().__dir__(), key=str.upper)


instance = Daeheeyun()
print(instance.__slots__)
print(instance.__dir__())
```

결과

['value']

['value', '__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__getattribute__', '__ge__', '__gt__', '__hash__', '__init_subclass__', '__init__', '__le__', '__lt__', '__module__', '__new__', '__ne__', '__reduce_ex__', '__reduce__', '__repr__', '__setattr__', '__sizeof__', '__slots__', '__str__',

'__subclasshook__']

__slots__은 클래스 내부에서 사용가능한 속성의 이름을 제한합니다.
즉, 사용할 속성(변수)의 이름을 미리 정의합니다. 다른 이름을 가진 속성명은 할당할 수 없습니다.

__slots__으로 속성이 제한되면, __dict__를 생성하지 않습니다.

__dir__은 클래스 내부에서 사용 가능한 모든 속성을 출력합니다.

dir(instance)를 통해서도 호출이 가능합니다.

속성(Property)

속성(Property)은 클래스에서 멤버의 값을 읽거나 쓸 수 있도록 제공하는 방식입니다.

클래스의 멤버를 직접 호출해 읽거나 변경할 수 있지만, 속성(Property)를 사용하면, 안정성과 유연성을 향상시킬 수 있습니다.

멤버를 은닉시켜 속성의 직접 접근을 막는다면, 안정성을 향상시킵니다.

멤버를 직접변경하는 방식이 아닌, 속성(Property)를 통해 변경하면 추가적인 작업을 진행할 수 있어 유연성을 향상시킵니다.

Private & Property

```
class Daeheeyun:
    def __init__(self):
        self.__value = 0.76923

    @property
    def value(self):
        return self.__value


instance = Daeheeyun()
print(instance.value)
```

결과

```
0.76923
```

Private

Python에도 한정자를 구현할 수 있습니다.

변수명 앞에 이중 밑줄(Double Underscore)로 구현할 수 있습니다.

한정자란 접근 한정자라고도 부르며, 접근 수준을 제한하는 역할을 합니다.

대표적으로 public 형식과, private 형식이 있습니다.

public은 부모 클래스, 자식 클래스에서 모두 접근 가능한 방식입니다.

private은 부모 클래스에서만 접근이 가능하며, 자식 클래스에서는 접근이 불가능합니다.

간단히 말해, public은 어디에서나 접근이 가능하며, private는 내부 클래스에서만 접근이 가능합니다.

만약, print(instance.__value)으로, 외부에서 __value에 접근할 경우, 오류가 발생합니다.

이를 통해, 클래스의 속성값이 허용 가능한 범위에서만 변경이 가능하도록 제어할 수 있습니다.

> Tip : 단일 밑줄(Single Undersocre)로 정의된 클래스나, 변수는 from <module> import * 형태로 import할 때 불러와지지 않습니다.

Property

속성(Property)은 메서드명 위에 @property를 통해 속성으로 할당할 수 있습니다.

속성으로 할당된 메서드는 하나의 인수(self)만 사용할 수 있습니다.

메서드처럼 구현되었지만, 속성을 호출하는 방식처럼 instance.value로 사용합니다.

@property, value는 기본적으로 속성 값을 반환하는 getter가 됩니다.

> Tip : @는 데코레이터(Decorator)를 의미합니다.
> Tip : instance.value()로 사용할 경우, 오류를 발생시킵니다.

Getter & Setter & Deleter

```
class Daeheeyun:
    def __init__(self):
        self.__value = 0.76923

    @property
    def value(self):
        pass
```

```
        @value.getter
        def value(self):
            print("GET")
            return self.__value

        @value.setter
        def value(self, value):
            print("SET")
            self.__value = value

        @value.deleter
        def value(self):
            print("DEL")
            del self.__value


    instance = Daeheeyun()
    instance.value = 100
    print(instance.value)
    del instance.value
```

결과

```
    SET
    GET
    100
    DEL
```

Getter

Getter는 속성의 값을 호출할 때 발생합니다.

value.getter를 사용하지 않는다면, @property가 Getter가 됩니다.

속성을 호출햇을 때에도 추가적인 프로세스를 가질 수 있습니다.

Setter

Setter는 속성의 값을 할당할 때 발생합니다.

value.setter를 사용하지 않는다면, 값을 변경할 수 없습니다.

Setter로 할당된 메서드는 두 개의 인수(self, value)만 사용할 수 있습니다.

Deleter

Deleter는 속성의 값을 제거할 때 발생합니다.

value.deleter를 사용하지 않는다면, 값을 제거할 수 없습니다.

특수 속성(Special attributes)

특수 속성(Special attributes)은 실제 코드의 프로세스에는 거의 포함되지 않지만,

개발 과정이나 디버깅 과정 등에서 유용하게 사용할 수 있는 속성입니다.

정의된 클래스의 상황이나 상태 등을 점검할 수 있습니다.

Special attributes

```
class Daeheeyun:
    """ Daeheeyun CLASS

    Callable types example.
    blah blah..

    To use:
    >>> instance = Daeheeyun(value)

    Args:
        value    : int

    Returns:
        null
    """

    def __init__(self, value: int):
        self.value = value

    def func(self) -> int:
        """func : Execute value * 2"""
        return self.value * 2


instance = Daeheeyun(5)

print(instance.__doc__)
print("---------")
print(instance.func.__doc__)
print("---------")
```

```
print(instance.func.__name__)
print(instance.func.__qualname__)
print(instance.func.__module__)
print(instance.func.__annotations__)
print(instance.__dict__)
```

결과

```
Daeheeyun CLASS

    Callable types example.
    blah blah..

    To use:
    » instance = Daeheeyun(value)

    Args:
        value : int

    Returns:
        null

---
func : Execute value * 2
---
func
Daeheeyun.func
__main__
{'return': <class 'int'>}
{'value': 5}
```

__doc__ : 설명 주석 호출

__doc__는 함수를 설명하는 데 사용된 주석을 반환합니다.

디버깅 과정에서 직접 모듈을 열어 확인하지 않고도 설명 주석을 확인할 수 있습니다.

__name__ : 함수명 호출

__name__은 함수의 이름 자체를 반환합니다.

모듈을 직접 실행됐을 때, 실행되는 코드인 if __name__ == '__main__':에서 __name__과 의미가 같습니다.

현재 스크립트에서 실행되는 함수나 모듈명을 출력합니다.

__qualname__ : 함수명과 경로 호출

__qualname__은 함수의 경로를 포함한 이름을 반환합니다.

이를 통해 실행되는 함수나 클래스가 어디서부터 실행되는지 확인할 수 있습니다.

__module__ : 정의된 모듈명 호출

__module__은 함수가 정의된 모듈의 이름을 반환합니다.

현재 코드에서 실행시키면, __main__을 반환하며, 다른 곳에서 import하는 경우 모듈의 이름을 반환합니다.

모듈을 직접 실행됐을 때, 실행되는 코드인 if __name__ == '__main__':에서 __main__과 의미가 같습니다.

__annotations__ : 함수의 주석 호출

__annotations__은 함수의 매개변수와 반환값에 정의된 주석(Annotations)을 반환합니다.

함수가 어떤 매개변수와 반환값을 요구하는지 확인할 수 있습니다.

__dict__ : 함수의 주석 호출

__dict__는 현재 인스턴스에 할당된 인스턴스 속성을 출력합니다.

인스턴스의 key 값과 value 값을 확인할 수 있습니다.

클로저(Closure)

클로저(Closure)란 함수가 내부 함수를 포함하고 있는 형태를 의미하며, 내부 함수 밖에 있는 외부 함수의 지역 변수를 참조하는 형태입니다.

외부 함수가 종료되어도 내부 함수의 지역 변수는 사라지지 않고 내부 함수에서 사용할 수 있습니다.

클로저의 정의는 다음과 같은 조건을 만족해야 합니다.

중첩 함수(Nested Function) 형태

내부 함수는 외부 함수의 지역 변수를 참조하는 형태
외부 함수는 내부 함수를 반환하는 형태

클로저를 사용하면, 전역 변수의 사용을 최소화할 수 있으며, 데이터를 은닉하여 지속성을 보장할 수 있습니다.

또한, 접근할 수 없는 범위의 데이터를 접근해 사용할 수 있습니다.

```
클로저(Closure) 사용하기
def outer_func(x):

    total = x

    def inner_func(y):
        return total + y

    return inner_func


main = outer_func(1)
sub1 = main(2)
sub2 = main(3)
sub3 = main(4)
print(sub1, sub2, sub3)
```

결과

3 4 5

main 변수에 outer_func의 x(total) 값을 1로 정의하고, inner_func를 반환합니다.

main 변수에 inner_func가 할당되며, inner_func의 y 값을 각각 2, 3, 4로 할당합니다.

inner_func 함수 기준에서 total 변수는 함수의 바깥 범위에 있지만, 결과에서 확인할 수 있듯이 참조가 가능한 형태입니다.

total 변수는 inner_func에서 사용되지만, 전역 변수도 아니며 inner_func 내부에서 정의하지 않은 변수입니다.

이 변수를 자유 변수(free variable)라 합니다.

중첩 함수(Nested Function)가 아닌 형태에서는 전역 변수를 사용하게 되어, 코드가 복잡해질 경우 어디서 변경되었는지 확인하기 어려워집니다.

이를 방지하고자 클로저(Closure)를 사용합니다. 즉, 전역 변수 대신에 자유 변수를 사용합니다.

지역 변수(local variable) 참조

```
def outer_func():
    value = 0

    def inner_func():
        value = 100

    inner_func()
    return value


func = outer_func()
print(func)
```

결과

```
0
```

outer_func 함수에서 정의된 value 변수를 inner_func에서 변경을 시도한다면, 값이 정상적으로 변경되지 않습니다.

위의 코드를 작성할 경우, Unused variable 'value' 형태의 문제가 발생합니다.

이는, inner_func 함수에서 value 변수가 재정의됐다는 의미가 됩니다.

위 코드에서 value 변수는 outer_func 함수의 value와 inner_func 함수의 value로 두 가지가 정의된 형태입니다.

만약, 외부 함수의 지역 변수를 수정하려면, 다음과 같이 사용합니다.

```
    def outer_func():
        value = 0

        def inner_func():
            nonlocal value
            value = 100

        inner_func()
        return value


    func = outer_func()
    print(func)
```

결과

```
    100
```

nonlocal value의 형태로 사용하면, 내부 함수의 value는 지역 변수로 새로이 생성되지 않고, 외부 함수의 value로 인식합니다.

nonlocal 키워드는 가장 빨리 만나는 상위 변수인 0을 참조하게 됩니다.

중첩이 더 깊어질 경우, 한 계단씩 올라가면서 변수를 찾게됩니다.

지역 변수(local variable) 변경

```
def outer_func():
    var = list()

    def inner_func(value):
        var.append(value)
        return var

    return inner_func


main = outer_func()
result = main({"A": 65})
print(result)
result = main({"B": 66})
print(result)
```

결과

```
[{'A': 65}]
[{'A': 65}, {'B': 66}]
```

외부 함수의 지역 변수를 변경하게 되면, 값이 지속적으로 유지됩니다.

클로저를 통해서 전역 변수를 사용하지 않고, 외부 함수의 자유 변수를 통해서도 구현이 가능합니다.

자유 변수(free variable)와 셀(cell)

```
def outer_func():
    var = list()

    def inner_func(value):
        var.append(value)
        return var

    return inner_func


main = outer_func()
result = main({"A": 65})
```

```
result = main({"B": 66})
print(main.__closure__)
print(type(main.__closure__), len(main.__closure__))
print(main.__closure__[0].cell_contents)
```

결과

```
(<cell at 0x00000157B5967708: list object at 0x00000157B599D508>,)
<class 'tuple'> 1
[{'A': 65}, {'B': 66}]
```

속성(attributes) 중 하나인 __closure__를 통해 자유 변수를 확인할 수 있습니다.

클로저 속성은 셀(cell) 형태로 스코프(Scope)에서 참조하는 변수를 보여줍니다.

스코프(Scope)란 정의된 변수가 보이는 유효 범위를 의미합니다.

즉, main 함수의 셀은 outer_func 함수 내에서 정의된 변수를 보여줍니다.

변수는 중복될 수 없으므로, 튜플의 형태를 가지며, 자유 변수의 개수만큼 길이를 가집니다.

외부 함수의 자유 변수의 개수는 1개이므로, 셀의 0 번째 값에 var 변수가 담겨있습니다.

main.__closure__[0]는 0 번째 셀을 출력합니다.

실제 값을 확인하기 위해서는 cell_contents를 사용해 자유 변수의 값을 확인할 수 있습니다.

반복자(Iterator)

반복자(Iterator)란 반복 가능한(iterable) 형식의 멤버를 순차적으로 반환할 수 있는 객체를 의미합니다.

배열 등의 요솟값을 순차적으로 접근할 때 사용합니다.

배열에서 색인(index)으로 값을 접근할 때는 데이터 메모리 레이아웃에서 배열의 시작 주소값과 오프셋(offset) 값으로 배열의 요소에 접근합니다.

즉, 배열에 접근할 때에는 배열 요소의 크기와 개별 요소에 접근하는 방식 등을 모두 고려해 접근합니다.

하지만, 반복자를 사용하면 값이 필요할 때 값을 생성하거나 계산해 반환합니다.

이를 느긋한 계산법(Lazy evaluation)이라 하며, 실제 값이 필요할 때까지 계산을 미루게 되므로 실행 속도 향상과 메모리 사용량을 줄일 수 있습니다.

또한, 데이터 내부 구조나 접근 방식 등에 상관 없이 요솟값에 접근할 수 있습니다.

반복자(Iterator) 사용하기

```
import numpy as np

_str = iter("1234")
_tuple = iter((1, 2, 3, 4))
_list = iter([1, 2, 3, 4])
_dict = iter({"a": 1, "b": 2, "c": 3, "d": 4})
_set = iter({1, 2, 3, 4})
_array = iter(np.array([[1, 2], [3, 4]]))

print(next(_str))
print(next(_str))
print(next(_str))
```

결과

```
1
2
3
```

반복자(Iterator)는 iter() 함수를 통해 반복 가능한 형식의 객체를 생성할 수 있습니다.

iterator 형식의 객체만 가능하며, next() 함수로 다음 번째 요솟값을 참조할 수 있습니다.

next()마다 다음 요솟값을 참조하게 되며, 마지막 요솟값 참조 이후에도 다음 값을 호출하는 경우엔 StopIteration 예외를 발생시킵니다.

반복문과 동시 사용

```
_dict = iter({"a": 1, "b": 2, "c": 3, "d": 4})

print(next(_dict))

for i in _dict:
    print(i)
```

결과

```
a
b
c
d
```

next() 함수와 반복문 등을 동시에 사용하게 되면, next() 함수를 호출한 것과 동일한 기능을 합니다.

next() 함수로 요솟값에 접근한 다음, 반복자에 반복문을 사용한다면 이미 앞서서 다음 번째 값을 계산하였기 때문에 순차적으로 출력됩니다.

반복문 이후에 next() 함수로 다음 값을 호출한다면, 더 이상 참조할 값이 없어 StopIteration 예외를 발생시킵니다.

itertools 모듈

```
from itertools import count

infinite = count()

print(next(infinite))
```

```
for i in infinite:
    print(i)
```

결과

```
0
1
2
…
8893
…
```

itertools 모듈의 count() 함수를 사용한다면, 크기가 무한한 형태의 반복자를 구현할 수 있습니다.

반복문을 종료하거나 프로세스를 직접 종료하지 않는다면 count가 무한히 증가합니다.

이를 통해, 무한히 반복하거나 매우 큰 횟수의 반복을 구현할 수 있습니다.

> Tip : infinite = range(10 ** 10000000000000000)의 형태로도 구현이 가능하지만, 앞서 설명한 것처럼 배열을 계산해야하므로 실행까지 큰 시간이 필요하거나 메모리가 부족할 수 있습니다.

생성자(Generator)

생성자(Generator)란 반복자(Iterator)를 생성해주는 함수입니다.

yield 키워드를 활용해 반복자 형태를 구현할 수 있습니다.

함수 내부에 yield가 존재하면, 함수 전체를 생성자(Generator)로 간주합니다.

일반적인 함수는 스코프(Scope)내의 코드를 모두 실행시킨 후, 소멸합니다.

하지만, 생성자 함수는 스코프 내에서 일시 중지해가며 실행시킬 수 있습니다.

즉, 함수가 실행되고 있는 도중에 중지하고 값을 수정할 수도 있습니다.

생성자(Generator) 사용하기

```
def generator():
    data = [0]

    print("First")
    data.append(1)
    yield data

    print("Second")
    data.append(2)
    yield data

    print("Third")


gen = generator()
print(next(gen))
print("---")
print(next(gen))
print("---")
print(next(gen, "END"))
print("---")
print(next(gen, "END"))
print("---")
```

결과

```
First
[0, 1]
---
Second
[0, 1, 2]
---
Third
END
---
END
---
```

생성자(Generator)는 yield 키워드를 통해 구현할 수 있습니다.

반복자(Iterator)를 사용해 생성자 객체를 반환합니다.

next() 함수를 실행시킬 때, yield 키워드 구문에서 일시 중지하게 됩니다.

함수를 초기화하는 것이 아닌 yield 구문에서 중지합니다.

함수 내부의 변수나 멤버는 유지되며, 변경이 가능합니다.

yield 키워드가 더 이상 남아있지 않을 때에는 남은 구문을 모두 출력하며, StopIteration 예외를 발생시킵니다.

하지만, next() 함수에 StopIteration 예외 발생시, 반환할 값을 설정할 수 있습니다.

StopIteration 예외 발생 이후에도 반환한다면, 더 이상 남아있는 구문이 없어 END 값만 반환합니다.

반복 가능한(iterable) 형식 반환하기

```
def generator():
    value = [1, 2, 3]
    yield value


gen = generator()
print(list(gen))
```

결과

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

yield 키워드를 통해 반복 가능한(iterable) 객체를 반환할 경우, 값이 묶여 반환됩니다.

이를 해결하기 위해 from 키워드를 추가해 한 번에 반환할 수 있습니다.

```
def generator():
    value = [1, 2, 3]
    yield from value


gen = generator()
print(list(gen))
```

결과

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

yield from iterable 형태로 값을 반환하면, 객체를 두 번 묶지 않고 반환할 수 있습니다.

이를 통해 불필요한 반복문이나 객체를 한 번 더 푸는 작업을 진행하지 않아도 됩니다.

코루틴(Coroutine)

```
from itertools import count


def coroutine():

    for c in count():
        value = yield c
        print("Value:{}, Count:{}".format(value, c))


cor = coroutine()
cor.send(None)
cor.send("A")
```

```
    cor.send(200)
    cor.send(50)
```
결과
```
    Value:A, Count:0
    Value:200, Count:1
    Value:50, Count:2
```

여태까지 사용하던 함수는 서브루틴(Subroutine) 함수로, 진입 지점이 하나이며 return 키워드를 통해 종료되었습니다.

하지만, 생성자(Generator)는 함수 실행 도중에 일시 정지를 할 수 있으므로, 일시 중지한 시점에서 값을 추가로 입력할 수 있습니다.

즉, 진입 지점이 여러 개로 늘어나게 되며 값을 주고 받을 수도 있게됩니다. 이를 코루틴(Coroutine)이라 합니다.

코루틴 함수에서 value = yield c의 형태로 사용한다면, 생성자 함수로 값을 보낼 수 있습니다.

여기서 값을 보내기 위해서는 함수를 초기화해야합니다.

coroutine() 함수가 생성된 이후, cor.send(None) 또는 next(cor)를 활용해 생성자 함수를 초기화합니다.

이후, cor.send() 메서드를 통해 값을 전달할 수 있습니다.

앞서, yield 구문을 만나면 해당 구문에서 일시 정지하였지만, 코루틴 함수는 해당 블록(for문 내부)을 모두 실행시킵니다.

그러므로, print() 함수가 실행됩니다. 즉, 값을 전달하고 블록을 모두 실행시킬 수 있습니다.

```
    from itertools import count


    def coroutine():

        for c in count():
            value = yield c
```

```
            print("next() - Value:{}, Count:{}".format(value, c))
            yield value
            print("send() - Value:{}, Count:{}".format(value, c))


    cor = coroutine()
    next(cor)
    cor.send("A")
    next(cor)
    cor.send(200)
    next(cor)
    cor.send(50)
    next(cor)
```

결과

```
    next() - Value:A, Count:0
    send() - Value:A, Count:0
    next() - Value:200, Count:1
    send() - Value:200, Count:1
    next() - Value:50, Count:2
    send() - Value:50, Count:2
```

value에 yield c를 할당한 이후에 yield 키워드를 추가해 반이중 방식(half-duplex) 형태와 흡사한 함수를 구현할 수 있습니다.

cor.send() 메서드를 통해 값을 전달하는 방식과 동일하지만, 함수 내부에 yield value가 추가되어 주고 받는 형식이 됩니다.

send() 메서드에서는 다음 번째 yield value가 만나기 전까지 모든 구문을 실행합니다.

yield value 구문은 next() 함수를 통해 넘어갈 수 있었습니다.

값을 다시 반환받기 위해서는 next(cor)를 활용합니다.

즉, 값을 보낼때는 cor.send()를 사용하며, 값을 받을때는 next()를 사용한다 볼 수 있습니다.

> Tip : cor.send()의 반환값은 value 값을 반환하며, next(cor)의 반환값은 c 값을 반환합니다.

오류 발생(throw)

```
from itertools import count


def coroutine():

    for c in count():
        try:
            value = yield c
            print("next() - Value:{}, Count:{}".format(value, c))
            yield value
            print("send() - Value:{}, Count:{}".format(value, c))
        except ValueError:
            print("Error", c)


cor = coroutine()
next(cor)
cor.send("A")
cor.throw(ValueError, "오류가 발생했습니다.")
cor.send(200)
cor.throw(ValueError, "오류가 발생했습니다.")
cor.throw(ValueError, "오류가 발생했습니다.")
```

결과

```
next() - Value:A, Count:0
Error 0
next() - Value:200, Count:1
Error 1
Error 2
```

코루틴(Coroutine) 도중 오류를 강제로 발생시켜, 예외 처리를 진행할 수 있습니다.

cor.throw()를 통해, 강제로 오류를 발생시킬 수 있습니다.

오류를 발생시켜도, 진행 중인 Count의 값은 증가하며, next() 함수와 동일한 기능을 합니다.

즉, 입력 → 출력, 입력 → 오류, 입력 → 출력 구조의 형태로도 구현이 가능합니다.

도중에 오류를 발생시키는 이유는, 함수 내부적으로 오류를 처리하는 것이 아닌 외부에서 처리하기 위함입니다.

이를 통해 특정 오류 상황을 알릴 수 있으며, 코드가 더 간결해질 수 있습니다.

생성자(Generator) 종료하기

```
from itertools import count


def coroutine():

    for c in count():
        try:
            value = yield c
            print("next() - Value:{}, Count:{}".format(value, c))
            yield value
            print("send() - Value:{}, Count:{}".format(value, c))
        except ValueError:
            print("Error", c)


cor = coroutine()
next(cor)
cor.send("A")
cor.close()
print(next(cor, "END"))
```

결과

```
next() - Value:A, Count:0
END
```

현재 생성자(Generator)는 무한히 반복되는 구조입니다.

코드 상에 종료 구문이 없으므로 끝이 존재하지 않습니다.

하지만, close() 함수를 통해 강제로 생성자를 종료할 수 있습니다.

cor.close()가 호출된 이후 부터 StopIteration 예외를 발생시킵니다.

즉, 더 이상 참조할 수 없는 형태로 변경합니다.

컨테이너 메서드(Container Method)

컨테이너(Container)란 자료형(Data type)의 저장 모델로 종류에 무관하게 데이터를 저장할 수 있음을 뜻합니다.

문자열, 튜플, 리스트, 사전, 집합 등은 종류에 무관(Container)한 형식이며, 정수, 실수, 복소수등은 단일 종류(Literal)한 형식입니다.

컨테이너 메서드(Container Method)는 위와 같이 종류에 무관하게 저장할 수 있는 자료형의 매직 메서드(Magic Method)를 뜻합니다.

메서드 정의

```
import operator


class Daeheeyun(dict):

    _dict = {"A": 1, "B": 2, "C": 3}

    def __len__(self):
        print("length")
        return len(self._dict)

    def __length_hint__(self):
        print("length_hint")
        return operator.length_hint(self._dict)

    def __getitem__(self, key):
        try:
            return self._dict[key]
        except:
            return self.__missing__(key)

    def __setitem__(self, key, value):
        self._dict[key] = value

    def __delitem__(self, key):
        del self._dict[key]
```

```
        def __missing__(self, key):
            return self._dict

        def __iter__(self):
            return iter(self._dict)

        def __reversed__(self):
            return dict(zip(self._dict.values(), self._dict.keys()))

        def __contains__(self, item):
            if item in self._dict:
                print("%s is contained in Key." % item)
                return True
            elif item in self._dict.values():
                print("%s is contained in Value." % item)
                return True
            else:
                print("%s is not contained in Key & Value." % item)
                return False


    daehee = Daeheeyun()
    print(operator.length_hint(daehee))
    daehee["E"] = 5
    print(daehee["D"])
    print(reversed(daehee))
    print(operator.contains(daehee, 1))
```

결과

```
    length
    3
    {'A': 1, 'B': 2, 'C': 3, 'E': 5}
    {1: 'A', 2: 'B', 3: 'C', 5: 'E'}
    1 is contained in Value.
    True
```

__len__ : 길이 반환 메서드

len()을 호출해 객체의 길이를 반환합니다.

__bool __() 메서드를 정의하지 않은 채 bool()을 호출하면 __len__ 반환 값에 의존합니다.

즉, len()의 반환값이 0이라면 False가 반환되며, 0이 아니라면 True를 반환합니다.

__length_hint__ : 예상 길이 반환 메서드

operator.length_hint()을 호출해 객체의 예상 길이를 반환합니다.

length_hint() 함수는 len() 함수와 기능과 역할이 흡사합니다.

len() 함수보다 비교적 정확성이 낮지만, length_hint()는 len() 함수가 반환하지 못하는 range_iterator 형태의 컨테이너도 길이 반환이 가능합니다.

__len__() 메서드와 같이 정의돼 있다면, 우선적으로 __len__() 메서드를 실행합니다.

__len__() 메서드에서 반환하지 못한다면, 그 다음으로 __length_hint__() 메서드를 통해 값을 반환합니다.

__getitem__ : 호출 메서드

self[key]로 key 값을 호출할 때 실행되는 메서드입니다.

__getitem__() 메서드를 재정의한다면, TypeError, IndexError, KeyError 오류 등에 대한 동작을 재정의해야 합니다.

__missing__() 메서드를 재정의했을 때, KeyError 등의 오류가 발생했을 때 별도로 연결시켜야합니다.

__setitem__ : 할당 메서드

self[key] = value로 key에 대한 value로 값을 할당할 때 실행되는 메서드입니다.

호출 메서드와 동일하게 __setitem__() 메서드를 재정의한다면, TypeError, IndexError, KeyError 오류 등에 대한 동작을 재정의해야 합니다.

__delitem__ : 제거 메서드

del self[key]로 key에 대한 값을 제거할 때 실행되는 메서드입니다.

호출 메서드와 동일하게 __delitem__() 메서드를 재정의한다면, TypeError, IndexError, KeyError 오류 등에 대한 동작을 재정의해야 합니다.

__missing__ : 누락 메서드

self[key]로 key 값을 호출할 때 사전(dictonary)에 key 값이 없을 때 실행되는 메서드입니다.

__getitem__() 메서드에 의해 호출되며, __getitem__()를 재정의 했다면, 예제와 같이 직접 호출해야합니다.

__missing__() 메서드는 사전 자료형에 대해서 동작하며, 예제와 같이 Daeheeyun(dict)로 dict 자료형을 상속받아야 합니다.

Tip : 사전(dict) 자료형 이외의 컨테이너 자료형은 key가 아닌 index로 value를 호출합니다. 그러므로, KeyError가 아닌 IndexError가 발생합니다.

__iter__ : 반복자 메서드

iter()를 호출해 반복자(iterator)를 반환합니다.

컨테이너 자료형은 iterator 형식의 자료형을 반환해야하며, 사전(dict)의 경우 Key로 구성된 iterator를 반환합니다.

__reversed__ : 반복자 메서드

reversed()를 호출해 컨테이너의 객체를 역순으로 정렬해 반환합니다.

예제는 사전(dict) 자료형을 사용해 정렬이 불가능하므로, Key와 Value를 반대로 정의하는 메서드로 구현하였습니다.

__contains__ : 포함 확인 메서드

operator.contains(self, item)을 호출해 자료형에 해당 item 값이 존재하는지 확인합니다.

객체에 값(item) 있으면 True를 반환하며, 존재하지 않다면 False를 반환합니다.

항상, True와 False로만 반환하며, 다른 값을 반환하더라도 0이 아닌 값은 True로 간주합니다.

예제는 Key와 Value 둘 다 검사하여 어느 곳에라도 item 값이 존재하면 True를 반환합니다.

정규표현식(Regular Expression)

정규 표현식(Regular Expression)은 프로그래밍에서 사용하는 형식 언어입니다.

특정한 규칙을 가진 문자열을 검색, 분리, 치환하는 데 주로 활용되며, 특정한 패턴과 일치하는 텍스트를 입력값에서 찾아 반환합니다.

또한, 정규 표현식의 패턴 표현은 어떤 프로그래밍 언어를 사용하던 동일한 의미를 갖습니다.

만약 정규 표현식을 사용하지 않고 문자열에서 특정 패턴을 찾는 경우 매우 복잡한 코드를 작성해야 합니다.

하지만 정규 표현식을 활용할 경우 코드가 매우 간결해지며 유사한 문자까지 일치시켜 검색할 수 있습니다.

패턴 정의

```
import re

string = """
정규 표현식(Regular Expression)은 프로그래밍에서 사용하는 형식 언어입니다.

특정한 규칙을 가진 문자열을 검색, 분리, 치환하는 데 주로 활용되며, 특정한 패턴과 일치하는 텍스트를 입력값에서 찾아 반환합니다.

정규 표현식을 사용하지 않고 문자열에서 특정 패턴을 찾는 경우 매우 복잡한 코드를 작성해야 합니다.

하지만 정규 표현식을 활용할 경우 코드가 매우 간결해지며 유사한 문자까지 일치시켜 검색할 수 있습니다.
"""

    pattern = re.compile(r"\((.*?)\)")
    find = re.findall(pattern, string)
    print(find)
```

결과

```
    ['Regular Expression']

    import re
```

Python에서 정규 표현식 모듈은 re로, 정규 표현식을 사용하기 위해 import합니다.

```
pattern = re.compile(r"\((.*?)\)")
```

정규 표현식을 사용하기 위해, 찾을 문자열의 패턴을 정의합니다.

정규 표현식의 패턴을 구성할 때 일반적으로 패턴의 문자열은 로 문자열 표기법(Raw string notation)으로 작성합니다.

로 문자열 표기법은 문자열에 'r'을 포함해 작성합니다.

위 예제에서 패턴은 ((.*?))를 의미합니다.

이 패턴을 정규 표현식 컴파일 함수(re.compile)로 Python 환경에 맞게 컴파일합니다.

re.compile(패턴, 플래그)을 의미합니다.

패턴은 검출할 문자열의 정규 표현식입니다.

플래그는 컴파일할 때 추가적인 설정을 의미합니다.

먼저, 패턴에 들어가는 표현식과 플래그는 다음의 표와 같습니다.

메타 문자

메타 문자	설명
.	줄바꿈 문자를 제외한 모든 문자를 포함
?	0개 또는 1개의 문자를 포함
*	0개 이상의 문자를 포함
+	1개 이상의 문자를 포함
^	문자열의 시작과 일치(MULTILINE 플래그 설정 시 각 행의 처음과 매치)
$	문자열의 끝과 일치(MULTILINE 플래그 설정 시 각 행의 마지막과 매치)
|	문자 OR 연산(둘 중 하나라도 매치)
[]	문자의 집합(집합 중 하나라도 매치)
[^]	문자 NOT 연산([^abc] = a, b, c 문자를 제외)
[-]	문자 범위 집합([0-9] = 0~9까지의 숫자만 매치)
{n}	n회 반복인 문자를 포함
{n,m}	n회 이상, m회 이하 반복인 문자를 포함
{n,}	n회 이상 반복인 문자를 포함
()	정규식 그룹화
(?:)	정규식 그룹화 제외

이스케이프 문자 목록

이스케이프 문자	설명
₩w	문자 또는 숫자와 매치 [a-zA-Z0-9_]
₩W	문자 또는 숫자를 제외한 매치 [^a-zA-Z0-9_]
₩d	숫자와 매치 [0-9]
₩D	숫자를 제외한 매치 [^0-9]
₩s	공백 문자와 매치 [₩t₩n₩r₩f₩v]
₩S	공백 문자를 제외한 매치 [^₩t₩n₩r₩f₩v]
₩b	단어 사이의 공백 매치
₩B	단어 사이의 공백을 제외한 매치
₩A	문자열의 처음과 매치
₩Z	문자열의 마지막과 매치
₩₩	역슬래시
₩”	큰따옴표
₩’	작은따옴표
₩t	탭
(	좌측 소괄호
)	우측 소괄호

최소 매칭 문자 목록

최소 매칭 문자	설명
??	?의 기능에서 반환되는 문자열을 최소 크기로 매치
*?	*의 기능에서 반환되는 문자열을 최소 크기로 매치
+?	+의 기능에서 반환되는 문자열을 최소 크기로 매치
{n,m}?	{n,m}의 기능에서 반환되는 문자열을 최소 크기로 매치

탐색 문자 목록

탐색 문자	설명
(?=)	긍정형 전방 탐색(앞의 문자가 포함되어야 함)
(?!)	부정형 전방 탐색(앞의 문자가 포함되지 않아야 함)
(?<=)	긍정형 후방 탐색(뒤의 문자가 포함되어야 함)
(?<!)	부정형 후방 탐색(뒤의 문자가 포함되지 않아야 함)

플래그 목록

플래그	설명
re.A re.ASCII	₩w, ₩W, ₩b, ₩B, ₩s, ₩S를 아스키코드로 매칭
re.U re.UNICODE	₩w, ₩W, ₩b, ₩B, ₩s, ₩S를 유니코드로 매칭
re.L re.LOCALE	₩w, ₩W, ₩b, ₩B, ₩s, ₩S를 현재 로케일 설정으로 매칭
re.I re.IGNORECASE	대소문자 구분 없이 매칭
re.M re.MULTILINE	문자열이 여러 줄인 경우 메타 문자 ^와 메타 문자 $는 각 행의 처음과 끝에 매칭
re.S re.DOTALL	메타 문자 .이 줄바꿈 문자도 포함해서 매칭
re.X re.VERBOSE	정규 표현식에 주석을 사용할 수 있도록 변경(#과 공백은 무시됨. 공백을 활용할 경우 메타 문자 ₩를 사용)

정규 표현식에서 사용되는 문자와 플래그는 위와 같습니다.

모두 외울 필요는 없으며, 필요한 기능을 찾아서 응용해 활용하시면 됩니다.

메타 문자, 이스케이프 문자, 최소 매칭 문자는 정규 표현식의 패턴을 정의할 때 사용됩니다.

예제에서 사용된 패턴 문자는 다음과 같습니다.

```
\((.*?)\)
```

\(와 \)는 이스케이프 문자로, 좌측 소괄호와 우측 소괄호를 의미합니다.

즉, 소괄호로 감싸진 (.*?)를 찾는다는 의미가 됩니다.

다음으로 나타나는 소괄호는 메타 문자로 정규식 그룹화를 의미합니다.

정규식 그룹화란 여러 패턴 문자 중, 실제로 찾고 싶은 문자를 하나의 그룹으로 만드는 역할을 합니다.

그러므로, 소괄호 사이에 포함된 문자(.*?)만 찾고 싶다는 의미가 됩니다.

.은 메타 문자이며, *?은 최소 매칭 문자입니다.

.은 줄바꿈 문자를 제외한 모든 문자를 포함입니다. 즉, 줄바꿈을 제외한 모든 문자를 찾습니다.

다음으로, *?은 *의 기능에서 반환되는 문자열을 최소 크기로 매치입니다.

먼저, *는 0개 이상의 문자를 포함한다는 뜻입니다.

이를 다시 정리해서 풀이한다면 다음과 같습니다.

줄 바꿈 문자를 제외한 모든 문자를 0개 이상 찾되, 최소 크기로 찾는다.

여기서 최소 크기는 다음과 같이 설명할 수 있습니다.

ABC라는 문자열에 (.*?)로 매칭한 다음 -라는 문자을 붙인다면, -A-B-C-의 형태가 됩니다.

? 문자를 제외하고 (.*)로 매칭한 다음 -라는 문자를 붙인다면, ABC-의 형태가 됩니다.

Tip : *은 0개 이상을 찾으므로, 문자가 없는 맨 앞줄도 찾아지게 됩니다.

다시, \((.*?)\)를 풀이한다면 다음과 같습니다.

소괄호로 둘러 쌓인 문자열을 찾아 하나의 그룹으로 형성한다.

이미 소괄호로 둘러 쌓인 문자열로 정의했기 때문에, 예제 기준으로는 ?를 포함 유/무는 큰 차이가 없습니다.

하지만, 소괄호 안에 또 다른 소괄호가 있다면 결과가 달라집니다.

(Regular (정규 표현식) Expression)

위와 같은 문자열에서 \((.*?)\)와 \((.*)\)를 적용한 결과는 다음과 같습니다.

\((.*?)\) : Regular (정규 표현식

\((.*)\) : Regular (정규 표현식) Expression

그러므로, 정규 표현식의 패턴을 정의할 때 전체 데이터의 구조가 어떻게 되어있는지 고려해서 패턴을 구성해야 합니다.

위와 같은 경우가 빈번하게 발생하며 원하지 않는 문자열 패턴이 검출되거나 검출하지 못할수도 있습니다.

find = re.findall(pattern, string)
패턴 정의가 모두 완료됐다면, 목록 반환 검출 함수(re.findall)로 문자열에서 패턴을 검출할 수 있습니다.

re.findall(패턴, 문자열, 플래그)를 의미하며, 목록(List)의 형태로 검출 결과를 반환합니다.

매치 객체

```
import re

string = """
정규 표현식(Regular Expression)은 프로그래밍에서 사용하는 형식 언어입니다.

특정한 규칙을 가진 문자열을 검색, 분리, 치환하는 데 주로 활용되며, 특정한 패턴과 일치하는 텍스트를 입력값에서 찾아 반환합니다.

정규 표현식을 사용하지 않고 문자열에서 특정 패턴을 찾는 경우 매우 복잡한 코드를 작성해야 합니다.

하지만 정규 표현식을 활용할 경우 코드가 매우 간결해지며 유사한 문자까지 일치시켜 검색할 수 있습니다.
"""

pattern = re.compile(r"\((.*?)\)")
match = re.search(pattern, string)

print(match)
print(match.group(), match.start(0))
```

결과

```
<re.Match object; span=(7, 27), match='(Regular Expression)'>
(Regular Expression) 7
```

앞선 목록 반환 검출 함수(re.findall)로 문자열을 검출한다면 단순하게 찾은 문자열만 반환합

니다.

만약, 더 세부적인 정보를 필요로 한다면 매치(Match) 객체를 반환해야합니다.

매치 객체는 검출된 문자열의 그룹을 상세하게 나누거나 검철된 문자열의 색인 값 등 자세한 정보를 담고 있습니다.

그룹은 패턴에 포함된 소괄호()를 의미합니다.

매치 반환 전체 문자열 검출 함수(re.search)로 문자열에서 패턴을 검출할 수 있습니다.

re.search(패턴, 문자열, 플래그)를 의미하며, 매치(Match)의 형태로 검출 결과를 반환합니다.

매치 객체의 메서드와 속성은 다음과 같습니다.

매치 메서드

메서드	설명
match.group()	매칭된 문자열의 그룹을 반환
match.group(n)	매칭된 문자열의 n번째 그룹을 반환
match.groups()	매칭된 문자열의 그룹을 튜플로 반환
match.groupdict()	매칭된 문자열의 그룹을 사전으로 반환
match.start(n/name)	매칭된 문자열 그룹의 시작 색인 값을 반환
match.end(n/name)	매칭된 문자열 그룹의 종료 색인 값을 반환
match.span()	매칭된 문자열 그룹의 (시작, 끝)에 해당되는 색인 값을 튜플을 반환

매치 속성

속성	설명
match.string	입력 문자열(string)을 반환
match.pos	입력 문자열(string)의 검색을 시작하는 위치 반환
match.endpos	입력 문자열(string)의 검색을 종료하는 위치 반환
match.lastindex	매칭된 문자열의 마지막 색인을 반환(없을 경우 None을 반환)
match.lastgroup	매칭된 문자열의 마지막 이름을 반환(없을 경우 None을 반환)

매치 반환 전체 문자열 검출 함수(re.search)는 하나의 대상만 검출합니다.

만약, 예제에서 콤마(,) 앞에 있는 문자열을 검출한다고 가정하면 검색만 검출하며 분리나 활용되며는 검출되지 않습니다.

여러 대상을 검출하려면 반복자 반환 검출 함수(re.finditer)를 사용해야 합니다.

Python 정규 표현식에서 사용할 수 있는 함수는 아래와 같습니다.

정규 표현식 함수

re.compile(pattern, flags=0)
패턴(pattern)과 플래그(flags)를 컴파일해서 정규 표현식 객체로 반환합니다.

re.findall(pattern, string, flags=0)
입력 문자열(string)에서 패턴(pattern)과 일치하는 문자를 목록으로 반환합니다.

re.finditer(pattern, string, flags=0)
입력 문자열(string)에서 패턴(pattern)과 일치하는 문자를 Match 객체 반복자로 반환합니다.

re.split(pattern, string, maxsplit=0, flags=0)
입력 문자열(string)에서 패턴(pattern)과 일치하는 문자를 최대 분할 크기(maxsplit) 개수만큼 분할해 목록으로 반환합니다.

(만약 최대 분할 크기(maxsplit)에 3을 입력할 경우, 반환되는 리스트의 길이는 4가 됩니다.)

re.sub(pattern, repl, string, count=0, flags=0)
입력 문자열(string)에서 패턴(pattern)과 일치하는 문자를 패턴 최대 반복 수(count)만큼 반복해 repl로 대체해서 문자열로 반환합니다.

(패턴 최대 반복 수(count)가 0일 경우, 횟수 제한 없이 모두 대체합니다.)

re.subn(pattern, repl, string, count=0, flags=0)
입력 문자열(string)에서 패턴(pattern)과 일치하는 문자를 패턴 최대 반복 수(count)만큼 반복해 repl로 대체해서 튜플로 반환합니다.

(패턴 최대 반복 수(count)가 0일 경우, 횟수 제한 없이 모두 대체합니다.)

re.match(pattern, string, flags=0)
입력 문자열(string)의 첫 부분에 대해 패턴(pattern)과 일치하는 문자를 Match 객체로 반환합니다.

(없을 경우 None을 반환합니다.)

re.search(pattern, string, flags=0)
입력 문자열(string)의 전체에 대해 패턴(pattern)과 일치하는 문자를 Match 객체로 반환합니다.

(없을 경우 None을 반환합니다.)

디스패치(Dispatch)

디스패치(Dispatch)는 오버로딩(Overloading) 구현을 위한 라이브러리입니다.

오버로딩이란 같은 이름의 메서드(method)에 매개변수의 개수나 형식을 다르게 지정해 2개 이상의 메서드를 정의하는 것을 의미합니다.

Python에서는 오버로딩을 지원하지 않지만, multipledispatch 라이브러리를 통해 오버로딩을 구현할 수 있습니다.

multipledispatch 라이브러리는 인스턴스 메서드, 내장 추상 클래스(int, float, str, …)를 지원하며, 캐시를 통해 빠르게 반복 조회할 수 있습니다.

디스패치(Dispatch) 설치

```
pip install multipledispatch
```

multipledispatch 라이브러리는 Python 2.6+, Pythono 3.2+를 지원합니다.

종속된 라이브러리로는 six 라이브러리만을 활용해 구현되어 있습니다.

> Tip : 아나콘다(Anaconda)를 통해 Python을 설치한 경우, 라이브러리가 설치되어 있을 수도 있습니다.

오버로딩(Overloading)

```
from multipledispatch import dispatch


@dispatch(int, int)
def add(x, y):
    return x + y


@dispatch(str, int)
def add(x, y):
    return f"{x} = {y}"


print(add(3, 6))
```

```
print(add("f(x)", 6))
```

결과

```
9
f(x) = 6
```

디스패치(Dispatch)를 사용하기 위해서 오버로딩하려는 메서드 위에 데코레이터(Decorator)를 적용합니다.

메서드는 동일한 이름으로 선언하며, @dispatch(*args)의 구조로 정의합니다.

적용하려는 오버로딩 메서드의 매개변수에 개수나 형식을 다르게 지정해 메서드의 기능을 나눕니다.

dispatch의 매개변수에 따라 서로 다른 메서드가 실행됩니다.

또한, 매개변수의 개수가 달라도 오버로딩을 구현할 수 있습니다.

Tip : @dispatch(str, int)와 @dispatch(int, str)는 서로 다른 메서드가 실행됩니다.
Tip : 동일한 매개변수를 갖는 메서드들이 선언되었다면 가장 아래쪽의 메서드로 적용됩니다.
Tip : 상위 클래스(Number 등)와 하위 클래스(int, float 등)로 구현한 메서드가 있다면, 하위 클래스의 메서드로 연결됩니다.

프로세스 기반 병렬 처리(Multi Processing)

프로세스 기반 병렬 처리(Multi Processing)란 쓰레딩(threading) 모듈과 비슷한 API를 활용하여 프로세스 스포닝(Process Spawning)을 지원하는 패키지입니다.

프로세스(Process)는 프로그램을 메모리 상에서 실행중인 작업을 의미합니다. 그러므로, 멀티프로세싱은 하나 이상의 프로세스들을 동시에 처리하는 것을 의미합니다.

대용량 데이터를 처리하는 과정이나 데이터를 분배하여 동시에 처리하고자 할 때 주로 활용합니다.

멀티 프로세싱은 다수의 프로세스로 처리하므로 안전성이 높지만, 각각 독립된 메모리 영역을 갖고 있어 작업량 많을 수록 오버헤드(Overhead)가 발생할 수 있습니다.

> Tip : 프로세스 스포닝(Process Spawning)이란 부모 프로세스(Parent Proecess)가 운영체제에 요청해 새로운 자식 프로세스(Child Process)를 만들어내는 과정입니다.

> Tip : 스레드(Thread)는 프로세스(Process) 안에서 실행되는 여러 흐름 단위를 의미합니다.

프로세스(Process)

```
import os
import time
from multiprocessing import Process, freeze_support


def task(idx, count):
    print(f"PID : {os.getpid()}")
    logic = sum([i ** 2 for i in range(count)])
    return idx, logic


if __name__ == "__main__":
    freeze_support()

    job = [("첫 번째", 10 ** 7), ("두 번째", 10 ** 7), ("세 번째", 10 ** 7), ("네 번째", 10 ** 7)]

    start = time.time()
```

```
    process = []
    for idx, count in job:
        p = Process(target=task, args=(idx, count))
        p.start()
        process.append(p)

    for p in process:
        p.join()

    print(f"End Time : {time.time() - start}s")

    start = time.time()

    for idx, count in job:
        task(idx, count)

    print(f"End Time : {time.time() - start}s")
```

결과

```
PID : 27792
PID : 28488
PID : 26248
PID : 26508
End Time : 7.55154824256897s
PID : 24800
PID : 24800
PID : 24800
PID : 24800
End Time : 13.973652124404907s
```

프로세스(Process) 클래스는 Process 객체를 생성한 후 start() 메서드를 호출해서 스폰합니다.

이후 각 프로세스는 join() 메서드를 통해 자식 프로세스가 종료될 때까지 대기합니다.

각 프로세스마다 ID가 존재하므로, process 목록을 통해 프로세스가 종료될 때 까지 대기하기 위해 join() 메서드를 호출합니다.

처리해야하는 연산량이 많은 경우, 프로세스 클래스를 통해 병렬 처리를 진행할 수 있습니다.

프로세스(Process)는 각 작업마다 새로운 프로세스가 할당되어 작업을 처리합니다.

Tip : Windows 환경에서는 freeze_support()를 통해 프로세스 개체에 대한 코드를 실행할 수 있게 설정합니다.
Tip : os.getpid()를 통해 프로세스마다 서로 다른 PID 값을 가진 프로세스가 실행되는 것을 확인할 수 있습니다.

풀(Pool)

```
import os
import time
from multiprocessing import Pool, freeze_support


def task(pairs):
    print(f"PID : {os.getpid()}")
    idx, count = pairs
    logic = sum([i ** 2 for i in range(count)])
    return idx, logic


if __name__ == "__main__":
    freeze_support()
    job = [("첫 번째", 10 ** 7), ("두 번째", 10 ** 7), ("세 번째", 10 ** 7), ("네 번째", 10 ** 7)]

    start = time.time()

    p = Pool(processes=2)
    result = p.map(task, job)

    print(result)
    print(f"End Time : {time.time() - start}s")

    start = time.time()

    result = [task(j) for j in job]

    print(result)
    print(f"End Time : {time.time() - start}s")
```

결과

PID : 23736
PID : 27140
PID : 23736
PID : 27140
[('첫 번째', 333333283333335000000), ('두 번째', 333333283333335000000), ('세 번째', 333333283333335000000), ('네 번째', 333333283333335000000)]
End Time : 8.848436832427979s
PID : 27316
PID : 27316
PID : 27316
PID : 27316
[('첫 번째', 333333283333335000000), ('두 번째', 333333283333335000000), ('세 번째', 333333283333335000000), ('네 번째', 333333283333335000000)]
End Time : 13.5242877006530 76s

풀(Pool) 객체는 여러 입력 값에 걸쳐 함수의 실행을 병렬 처리하고 입력 데이터를 프로세스에 분산시킵니다.

풀의 인스턴스를 생성하고 2개의 작업자를 생성합니다.

map() 메서드를 통해 실행하려는 함수와 반복 가능한 객체를 입력하여 각 프로세스에 매핑합니다.

풀(Pool)은 사전에 프로세스(processes)의 개수를 설정하여 반복합니다.

Tip : 프로세스의 개수가 2개라면, 첫 번째와 세 번째 작업은 같은 PID를 갖습니다.

병렬(Parallel)

```
import os
import time
from joblib import Parallel, delayed


def task(idx, count):
    print(f"PID : {os.getpid()}")
    logic = sum([i ** 2 for i in range(count)])
    return idx, logic
```

```
    job = [("첫 번째", 10 ** 7), ("두 번째", 10 ** 7), ("세 번째", 10 ** 7), ("네 번째", 10 ** 7)]

    start = time.time()

    result = Parallel(n_jobs=4)(delayed(task)(idx, count) for idx, count in job)

    print(result)
    print(f"End Time : {time.time() - start}s")

    start = time.time()

    result = [task(*j) for j in job]

    print(result)
    print(f"End Time : {time.time() - start}s")
```

결과

```
    PID : 21636
    PID : 28004
    PID : 4808
    PID : 1604
    [('첫 번째', 333333283333335000000), ('두 번째', 333333283333335000000), ('세 번째', 333333283333335000000), ('네 번째', 333333283333335000000)]
    End Time : 6.701249122619629s
    PID : 25380
    PID : 25380
    PID : 25380
    PID : 25380
    [('첫 번째', 333333283333335000000), ('두 번째', 333333283333335000000), ('세 번째', 333333283333335000000), ('네 번째', 333333283333335000000)]
    End Time : 13.458436727523804s
```

joblib 라이브러리는 multiprocessing 모듈과 동일한 기능을 포함하고 있습니다.

주요한 차이점으로는 매개 변수를 조금 더 쉽게 전달할 수 있으며, 대규모 Numpy 기반 데이터 구조에 대해 작업자 프로세스와 공유 메모리를 효율적으로 사용할 수 있습니다.

Parallel(n_jobs=프로세스 개수)(delayed(함수)(인수))의 구조로 사용할 수 있습니다.

Parallel 클래스는 병렬 매핑을 위한 클래스입니다. 백엔드(backend)를 설정하거나, 배치 크기(batch_size) 등을 추가로 설정할 수 있습니다.

delayed 메서드는 함수의 인수를 캡처하는 데 사용되는 데코레이터입니다.

대규모 작업에서는 joblib를 활용하는 것이 효율적인 방법입니다.

> Tip : 멀티 프로세싱은 여러 프로세스를 생성하는 데, 시간이 소요되므로 비교적 작업량이 적은 경우 단일 프로세스가 더 빠를 수 있습니다.