파이썬 필수문법

Table of Contents

1. Python Acvanced(1)

Variable Scope

> py_ad_1_1.py
>
> - [x] Scope
> - [x] Global
> - [x] Nonlocal
> - [x] Locals
> - [x] Globals

1. Vairable Scope
   - **Global** : Global Variable
   - **Local** : Local Variable

```
a = 10  # Global Variable
b = 20  # Global Variable
def foo():
    a = 20  # Local Variable
    b = b + 10  # UnboundLocalError: local variable 'b' referenced
before assignment
    print(a)

def bar():
    global a
    a = a + 10
    print(a)
```

Output

```
20
30
```

2. Nonlocal

   - **Nonlocal** : Local Variable in Nested Function

```
def outer():
    a = 10
    def inner():
        nonlocal a  # Nonlocal Variable
        a += 20
        print(a)
    return inner

foo = outer()
foo()
```

Output

```
30
```

3. Locals, Globals

   - `locals()` : 로컬 변수들의 상태를 딕셔너리 형태로 반환
   - `globals()` : 글로벌 변수들의 상태를 딕셔너리 형태로 반환

```
test_variable = 100
print("Ex > ", globals())
globals()["test_variable"] = 100  # 변수 선언 원리
```

Output

```
Ex >  {...,  'test_variable': 100}
```

4. 지역 -> 전역 변수 생성

```
for i in range(1, 10):
    for k in range(1, 10):
        globals()['mul_{}{}'.format(i, k)] = i * k  # 동적으로 전역 변수 생
성
```

```
print(mul_99)
print(globals())
```

Output

```
81
{..., 'mul_99': 81, ...}
```

> 📖 **Summary**
>
> ✅ 전역변수는 주로 변하지 않는 고정 값에 사용
> ✅ 지역변수 사용 이유 : 지역변수는 함수 내에 로직 해결에 국한, 소멸주기 : 함수 실행 해제 시
> ✅ 전역변수를 지역 내에서 수정되는 것은 권장하지 않음
> ✅ 전역변수는 주로 변하지 않는 고정 값에 사용

Lambda, Reduce, Map, Filter Functions

> py_ad_1_2.py
>
> - ☑ Lambda
> - ☑ Reduce
> - ☑ Map
> - ☑ Filter

1. Lambda
    - `lambda` : 익명 함수, 한 줄로 함수를 표현
        - 사용 즉시 소멸
        - 파이썬 가비지 컬렉션에 의해 메모리 관리 용이
        - 일반 함수 : 재사용성을 위해 메모리 저장
        - 시퀀스형 전처리에 주로 사용

```
# 일반 함수
def mul_10(num: int) -> int:
    return num * 10

# Lambda 함수
lambda_mul_10 = lambda num: num * 10
```

2. map
    - `map(func, iterable)` : iterable의 모든 요소에 func 적용

```python
# 일반 함수
def mul_10(num: int) -> int:
    return num * 10

# Lambda 함수
lambda_mul_10 = lambda num: num * 10

# map
map_ex = map(lambda_mul_10, [1, 2, 3, 4, 5])
print(list(map_ex))

# map 모듈화
def mul_10(nums: list(int)) -> object:
    def mul(num: int) -> int:
        return num * 10
    return map(mul, nums)

print(list(mul_10([1, 2, 3, 4, 5])))
```

Output

```
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50]
```

3. filter

   - `filter(func, iterable)` : iterable의 모든 요소에 func 적용 후 True인 요소만 반환

```python
# filter
filter_ex = filter(lambda x: x % 2 == 0, [1, 2, 3, 4, 5])
print(list(filter_ex))

# filter 모듈화
def filter_even(nums: list(int)) -> object:
    def even(num: int) -> bool:
        return num % 2 == 0
    return filter(even, nums)

print(list(filter_even([1, 2, 3, 4, 5])))
```

Output

```
[2, 4]
[2, 4]
```

4. reduce

    - `reduce(func, iterable)` : iterable의 모든 요소에 func 적용 후 누적

```
from functools import reduce

# reduce
reduce_ex = reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
print(reduce_ex)

# reduce 모듈화
def reduce_sum(nums: list(int)) -> object:
    def add(x, y) -> int:
        return x + y
    return reduce(add, nums)

print(reduce_sum([1, 2, 3, 4, 5]))
```

Output

```
15
15
```

> 📖 **Summary**
>
> ```
> ✅ Lambda : 익명 함수, 한 줄로 함수를 표현
> ✅ map : iterable의 모든 요소에 func 적용
> ✅ filter : iterable의 모든 요소에 func 적용 후 True인 요소만 반환
> ✅ reduce : iterable의 모든 요소에 func 적용 후 누적
> ```

Shallow Copy & Deep Copy

> py_ad_1_3.py
>
> - ☑ Shallow Copy
> - ☑ Deep Copy

1. 가변 / 불변 타입

    - **mutable : list, dict, set**
    - immutable : int, float, str, tuple

2. Copy 종류

    - **Shallow Copy** : 가변형 객체 안의 객체는 같은 주소값을 참조

```
import copy

c_list = [1, 2, 3, [4, 5, 6], [7, 8, 9]]
d_list = copy.copy(c_list)

print(id(c_list))
print(id(d_list))
d_list[2] = 100
d_list[3][1] = 1000
d_list[4][1] = 1000

print(c_list)
print(d_list)
```

Output

```
4380083712
4380083840

# ⭐ 객체 안의 객체는 같은 주소값을 참조
[1, 2, 3, [4, 1000, 6], [7, 1000, 9]]
[1, 2, 100, [4, 1000, 6], [7, 1000, 9]]
```

- **Deep Copy** : 값 복사

```
e_list = [1, 2, 3, [4, 5, 6], [7, 8, 9]]
f_list = copy.deepcopy(e_list)

print(id(e_list))
print(id(f_list))

f_list[3].append(100)
f_list[4][1] = 100000

print(e_list)
print(f_list)
```

Output

```
4380083712
4380083840

# ⭐ 객체 안의 객체는 다른 주소값을 참조
[1, 2, 3, [4, 5, 6], [7, 8, 9]]
[1, 2, 3, [4, 5, 6, 100], [7, 100000, 9]]
```

> 📖 **Summary**
>
> ✅ Copy 의 종류에 대해 이해하지 못한다면 디버깅이 힘들어질 수 있음

Context Manager

> py_ad_1_4.py / py_ad_1_5.py
>
> - ☑ Contextlib
> - ☑ With 기능 직접 구현
>   - ☑ **enter**
>   - ☑ **exit**
> - 타이머 클래스 구현
>   - Contextlib 구현

1. Context Manager context: 맥락, 문맥
   - 원하는 타이밍에 정확하게 리소스를 할당, 제공 및 반환하는 역할
   - 가장 대표적인 with 구문 이해
   - 정확한 이해 후 사용이 프로그래밍 개발에 중요(문제 발생 요소 인식)
   - **파일 핸들링, DB 커넥션, 소켓 처리** 등에 활용

```
class MyWithClass(object):
    def __init__(self, ...):
        ...

    def __enter__(self): # 리소스를 할당하거나 리소스를 제공
        ...

    def __exit__(self, exc_type, exc_value, traceback): # 리소스를 반환
        # exc_type : 예외 타입
        # exc_value : 예외 값
        # traceback : 예외 발생 위치
        ...
```

2. 타이머 클래스 구현

> <참고>
>
> 1. 예외와 에러의 차이
>   - 예외는 예측 가능한 오류
>   - 에러는 예측 불가능한 오류
> 2. `time.monotonic` : python 3.9 이상에서 나노초 단위의 정밀도를 제공

Example

```
import time

class ExcuteTimer(object):
    def __init__(self, msg):
        self._msg = msg
        self._start = None

    def __enter__(self):
        self._start = time.monotonic()  # 나노초 단위의 정밀도를 제공
        return self._start

    def __exit__(self, exc_type, value, traceback):
        if exc_type:
            print(f"Logging exception {(exc_type, value, traceback)}")
        else:
            print(f"{self._msg} : {time.monotonic() - self._start}")
        return True  # 문제없이 실행됐음을 반환하기 위해


with ExcuteTimer("Something job") as v:
    print(f"Recieved start monotonic: {v}")  # __enter__ 함수의 반환값
    # Excute job
    for i in range(10_000_000):
        pass

    # raise Exception(
    #     "Raise Exception!"
    # )  # Logging exception (<class 'Exception'>, Exception('Raise
Exception!'), <traceback object at 0x104f5cbc0>)
```

1. Python Acvanced(2)

Context Manager Annotation

> py_ad_2_1.py
>
> - [x] Decorator 사용
> - [x] Contextlib.contextmanager
> - [x] With 비교

1. contextmanager 함수 형태로 구현

```
from contextlib import contextmanager

@contextmanager
def my_timer(msg):
    start = time.monotonic()
    try:
        yield start
    except Exception as e:
```

```
        print(f"Logging exception {e}")
    finally:
        print(f"{msg} : {time.monotonic() - start}")
```

Example

```
with my_timer("Something job") as v:
    print(f"Recieved start monotonic: {v}")  # __enter__ 함수의 반환값
    # Excute job
    for i in range(10_000_000):
        pass

    # raise Exception(
    #     "Raise Exception!"
    # )  # Logging exception Raise Exception!
```

Property(1) - Underscore

> py_ad_2_2.py
>
> - ☑ Python Underscore
> - ☑ 다양한 언더스코어 사용
> - ☑ 접근지정자 이해

- underscore

1. 인터프리터
2. 네이밍(국제화, 자릿수)
3. 값 무시

   Example

   ```
   # Unpacking
   x, _, y = (1, 2, 3)
   print(x, y)
   # >> 1 3

   a, *_, b = (1, 2, 3, 4, 5)
   print(a, b)
   # >> 1 5

   for _ in range(10):
       pass
   ```

4. 접근 지정자 feat. Naming Mangling

1. `var` : Public, 읽기 쓰기 허용
2. `_var` : Protected, 읽기 쓰기 제한
3. `__var` : Private, 읽기 쓰기 제한, Naming Mangling

Example

```
class SampleA:
    def __init__(self):
        self.x = 0  # Public
        self.__y = 0  # Private
        self._z = 0  # Protected


a = SampleA()
a.x = 1

print(f"{a.x}")
# >> 1
# print(a.__y)  # AttributeError: 'SampleA' object has no
attribute '__y'
print(dir(a))
# >> ['_SampleA__y', ..., '_z', 'x']
```

- 언더스코어를 2개 사용하면 python 내부적으로 `_SampleA__y`로 변환하여 접근 제한
- `SampleA._SampleA__y`로 접근하면 접근 가능하지만 **권장 X**

Property(2) - Getter, Setter

> py_ad_2_3.py
>
> - ☑ Pythonic Code
> - ☑ @Property
> - ☑ Getter, Setter

- 프로퍼티(Property) 사용 장점

1. 파이써닉한 코드
2. 변수 제약 설정
3. Getter Setter 효과 동등(코드 일관성)
    - 캡슐화-유효성 **검사 기능 추가 용이**
    - 대체 표현(속성 노출, 내부의 표현 숨기기 가능)
    - 속성의 수명 및 메모리 관리 용이
    - 디버깅 용이
    - Getter, Setter 작동에 대해 설계된 여러 라이브러리(오픈소스) 상호 운용성 증가

Example

```python
class SampleA:
    def __init__(self):
        self.x = 0
        self.__y = 0  # Private

    @property  # Getter
    def y(self):
        print("Called get method.")
        return self.__y

    @y.setter
    def y(self, value):
        print("Called set method.")
        if value < 0:  # Setter 제약 조건 추가
            raise ValueError("0보다 큰 값을 입력하세요.")
        self.__y = value

    @y.deleter
    def y(self):
        print("Called delete method.")
        del self.__y


a = SampleA()
a.x = 1
a.y = 2

print(f"x : {a.x}")
print(f"y : {a.y}")

# deleter
del a.y
print(dir(a))
```

Output

```
Called set method.
x : 1
Called get method.
y : 2
Called delete method.
[..., 'x', 'y']  # _SampleA__y 속성이 삭제됨
```