STEP17. 메모리 관리와 순환 참조

파이썬 인터프리터 == 파이썬 코드를 실행하는 프로그램
파이썬 메모리 관리는 표준으로 사용되는 파이썬 인터프리터인 CPython 기준으로 함

메모리 관리

CPython의 메모리 관리

- 파이썬은 필요 없어진 객체를 메모리에서 자동 삭제함
- 코드 작성에 따라 메모리 누수 (memory leak) | 메모리 부족 (out of memory) 문제 발생

신경망의 메모리 관리

- 큰 데이터를 다룸
- 메모리 관리가 적절하지 못하면 실행 시간이 오래 걸리는 일이 자주 발생

파이썬의 메모리 관리 두 가지 방식

- 참조 카운트 == 참조 (reference) 수를 세는 방식
- GC (Garbage Collection) == 세대를 기준으로 쓸모 없어진 객체를 회수하는 방식

참조 카운트 방식의 메모리

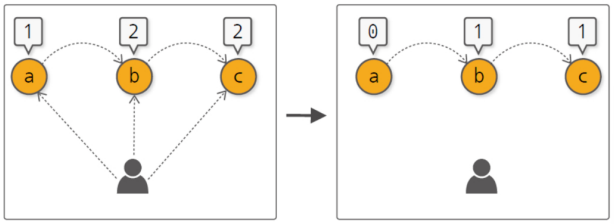
참조 카운트 증가
- 대입 연산자를 사용할 때
- 함수에 인수로 전달할 때
- 컨테이너 타입 객체 (리스트, 튜플, 클래스 등) 에 추가할 때

```
class obj:
  pass

def f(x):
  print(x)

a = obj() # 변수에 대입 : 참조 카운트 1
f(a) # 함수에 전달 : 함수 안에서는 참조 카운트 2
# 함수 완료 : 빠져나오면 참조 카운트 1
a = None # 대입 해제 : 참조 카운트 0
```

- 모든 객체는 참조 카운트 0인 상태로 생성되고, 다른 객체가 참조할 때마다 1씩 증가
- 객체에 대한 참조가 끊길 때마다 1만큼 감소하다가 0이 되면 회수

```
a = obj()
b = obj()
c = obj()

a.b = b
b.c = c

a = b = c = None
```

그림 17-1 객체 관계도(참조 관계는 점선, 숫자는 참조 카운트)

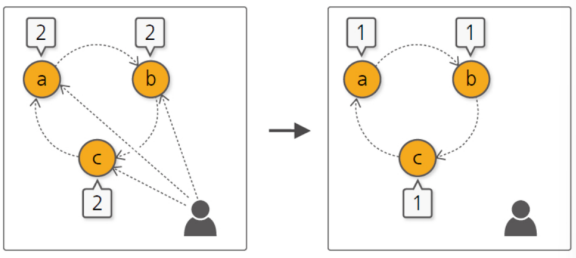


순환 참조

```
# 참조 카운트로 해결할 수 없는 문제에 사용

a = obj()
b = obj()
```

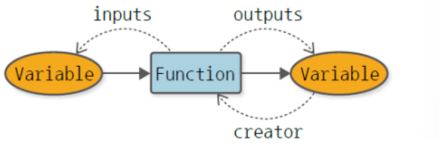
```
c = obj()

a.b = b
b.c = c
c.a = a

a = b = c = None
```

- `a = b = c = None` 해도 메모리에서 삭제 되지 않음



세대별 가비지 컬렉션 (generational garbage collection)

- GC : 순환 참조를 올바르게 처리함
- 메모리가 부족해지는 시점에 파이썬 인터프리터에 의해 자동 호출 (gc.collect()로 명시적 호출 가능)
- 메모리 해제를 GC에 미루다 보면 메모리 사용량이 커지는 원인이 됨
- DeZero 개발할 때는 순환참조를 만들지 않는 것이 좋음



weakref 모듈

약한 참조 (weak reference)

- 다른 객체를 참조하되 참조 카운터는 증가시키지 않는 기능
- 파이썬에서는 weakref.ref 함수를 사용하여 약한 참조를 만듦

```
>>> import weakref
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> b = weakref.ref(a)
>>> b
<weakref at 0x000001DBC37C7090; to 'numpy.ndarray' at 0x000001DBC3762DB0>
>>> b()
array([1, 2, 3])
```

- b는 약한 참조이고, 약한 참조된 데이터에 접근하려면 b( )라고 쓰면 됨

a = None을 실행할 때 b는 ?

```
>>> a = None
>>> b
<weakref at 0x000001DBC37C7090; dead>
>>>
```

- b를 출력하면 dead라는 문자 나옴 -> 인스턴스가 삭제 됐음을 알수 있음

1. Weakref 구조를 DeZero에 도입

```
import weakref

# Variable 인스턴스를 변수로 다룰 수 있는 함수를 Function클래스로 구현
class Function:
    # *ㅁㅁㅁ : 임의 개수의 인수 ( 가변길이 ) 를 건내 함수를 호출할 수 있음
    def __call__(self, *inputs):
        # 리스트 xs를 생성할 때, 리스트 내포 사용
        # 리스트의 각 원소 x에 대해 각각 데이터 ( x.data ) 를 꺼냄
        xs = [x.data for x in inputs]

        # forward 메서드에서 구체적인 계산을 함
        ys = self.forward(*xs) # 리스트 언팩 ( 원소를 낱개로 풀어서 전달 )

        if not isinstance(ys, tuple): # 튜플이 아닌 경우 추가 지원
            ys = (ys, )

        # ys의 각 원소에 대해 Variable 인스턴스 생성, outputs 리스트에 저장
        outputs = [Variable(as_array(y)) for y in ys]

        self.generation = max([x.generation for x in inputs])

        # 각 output Variable 인스턴스의 creator를 현재 Function 객체로 설정
        for output in outputs:
            output.set_creator(self)

        self.inputs = inputs # 입력 저장
        self.outputs = [weakref.ref(output) for output in outputs]

        # 리스트의 원소가 하나라면 첫 번째 원소를 반환함
```

```
        return outputs if len(outputs) > 1 else outputs[0]

        ```
```

- self.outputs가 대상을 약한 참조로 가리키게 변경함

```
class Variable:

  ```

  def backward(self):

    ```

    while funcs:
            f = funcs.pop() # 함수를 가져온다.
            # 수정전 : gys = [output.grad for output in f.outputs]
            gys = [output().grad for output in f.outputs]

            ```
```

동작 확인

DeZero 순환 참조 문제 해소 방법

```
for i in range(10):
    x = Variable(np.random.randn(10000)) # 거대한 데이터
    y = square(square(square(x))) # 복잡한 계산을 수행함
```

- for 문이 두 번째 반복될 때 x와 y가 덮어 써짐
- 사용자는 이전의 계산 그래프를 더 이상 참조하지 않게 됨
- 참조 카운트가 0이 되므로 이 시점에 계산 그래프에 사용된 메모리가 삭제됨

💡 - 파이썬으로 메모리 사용량을 측정하려면 외부 라이브러리인 memory profiler 등을 사용하면 편리함
- 실제로 위의 코드를 측정해보면 메모리 사용량이 전혀 증가하지 않음을 확인할 수 있음