

O'REILLY®

파이썬으로 직접 구현하며 배우는 딥러닝 프레임워크

Deep Learning from Scratch ③

밑바닥부터 시작하는 딥러닝 3



한빛미디어
HANBIT MEDIA, INC.

사이토 고키 지음
개원엽시 옮김

Book Review & Project

2021.3.2 ~ 3.18

한국IT교육원 303호



CONTENTS

제 1고지

미분 자동 계산

제 2고지

자연스러운 코드로

제 3고지

고차 미분 계산

제 4고지

신경망 만들기

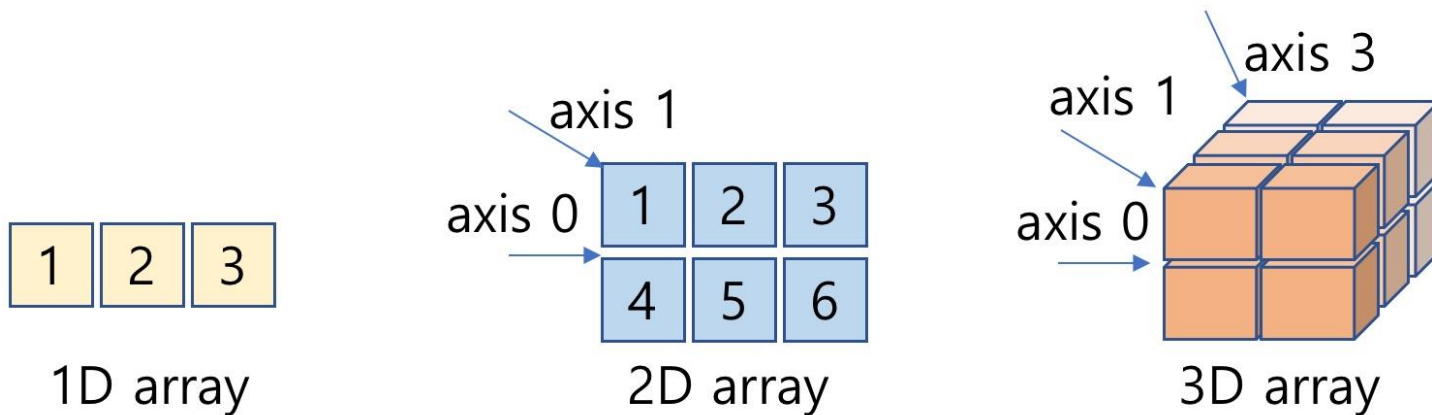
제 5고지

DeZero의 도전

제 1고지 미분 자동 계산

1단계: 상자로서의 변수 & steps/steps01.py

- Variable 클래스 구현 (*PEP8 규칙 참조)
- ML 시스템에서 기본 데이터 구조는 “다차원 배열”



<출처: http://taewan.kim/post/numpy_cheat_sheet/>

2단계: 변수를 낳는 함수 & steps/steps02.py

○ Function 클래스 구현

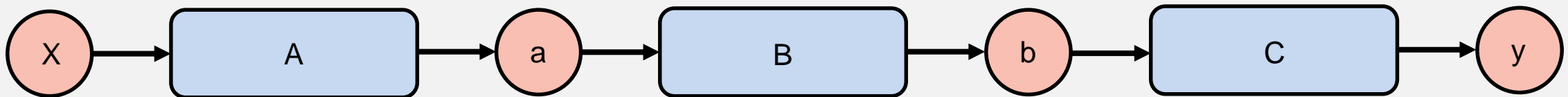
- `__call__(self, input)` 메서드의 인수 `input`은 `Variable` 인스턴스라 가정*
- `__call__` 메서드를 정의하면 `f = Function()` 형태로 사용 가능
: 예) `f(...)` 형태로 `__call__` 메서드 호출

* <https://tech.ssut.me/python-functions-are-first-class/>

3단계: 함수 연결 & steps/steps03.py

○ Exp 함수 구현

- $y = e^x$ 계산하는 함수 구현
: e 는 자연로그의 밑, 2.718...
- Square() & Exp() 모두, Function 클래스 상속*
- 합성 함수(Composite Function)



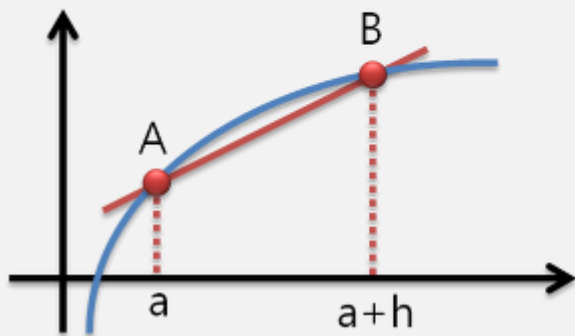
* <https://wikidocs.net/16073>

4단계: 수치 미분 & steps/steps04.py

○ 미분 (Derivative)

- 변화율 (예, 물체의 시간에 따른 위치 변화율)*

$$\text{평균변화율: } \frac{f(a+h) - f(a)}{(a+h) - a} = \frac{f(a+h) - f(a)}{h}$$



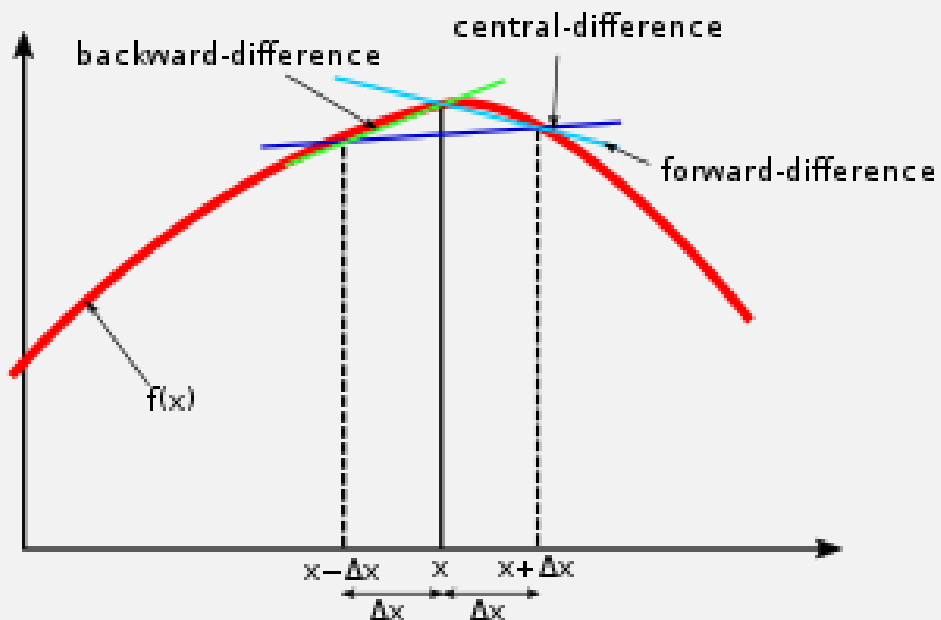
두 점 A와B의 기울기를 의미함
=평균변화율

* <https://j1w2k3.tistory.com/295>

4단계: 수치 미분 & steps/steps04.py

○ 미분 (Derivative)

- 변화율 (예, 물체의 시간에 따른 위치 변화율)*
- 중앙차분 (Centered Difference)



* <https://j1w2k3.tistory.com/295>

4단계: 수치 미분 & steps/steps04.py

○ 미분 (Derivative)

- 변화율 (예, 물체의 시간에 따른 위치 변화율)
- 중앙차분 (Centered Difference)
- $\text{eps} = \text{가장 작은 값} = 1\text{e-}4^*$
- 결괏값 3.297의 의미: x 를 0.5만큼 변화하면 3.297 만큼 변한다는 의미

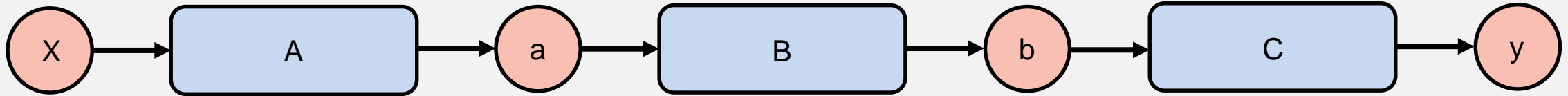
○ 수치 미분의 문제점

- 오차가 존재 (자릿수 누락, 예: 1.233와 1.23333은 다른 숫자 이지만, 유효숫자 처리에 따라 같아 질 수 있음)
- 계산량이 많아짐. 특히, 신경망에서는 모두 미분으로 구할 수 없음

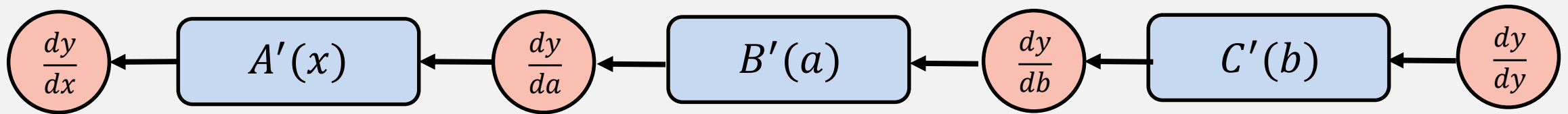
* 엡실론은 수학에서 가장 작은 양의 값을 나타내는 데 사용, 컴퓨터에서 아주 작은 양의 부동소수점 값을 담는 변수의 이름으로 사용 중

5단계: 역전파

○ 순전파



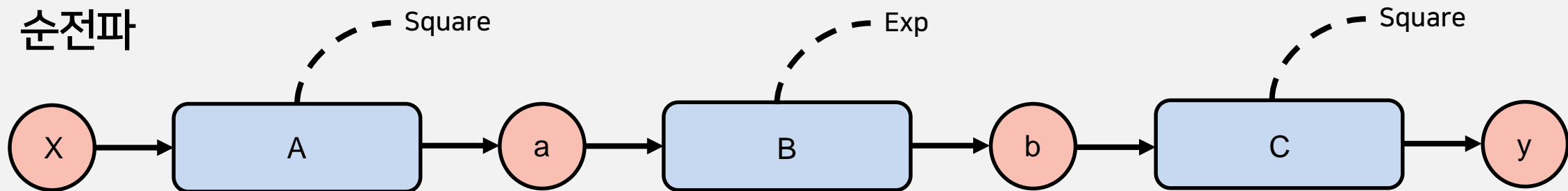
○ 역전파



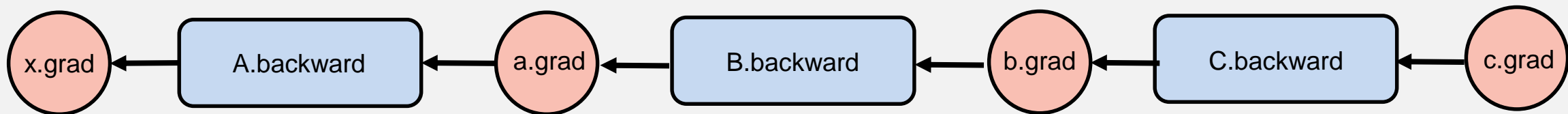
5단계: 역전파

6단계: 역전파 & steps/steps06.py

○ 순전파

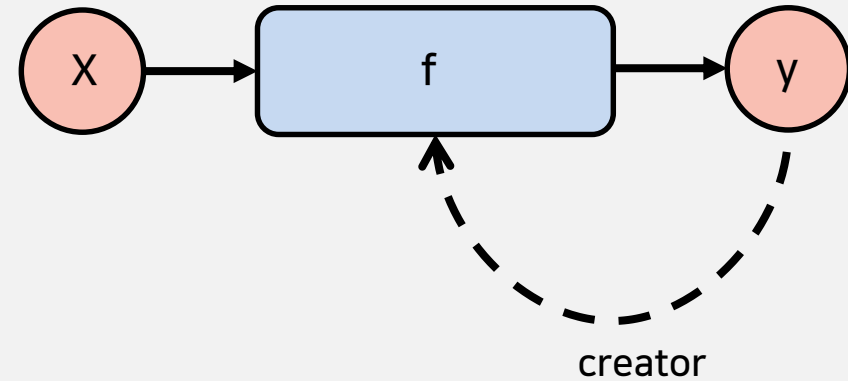
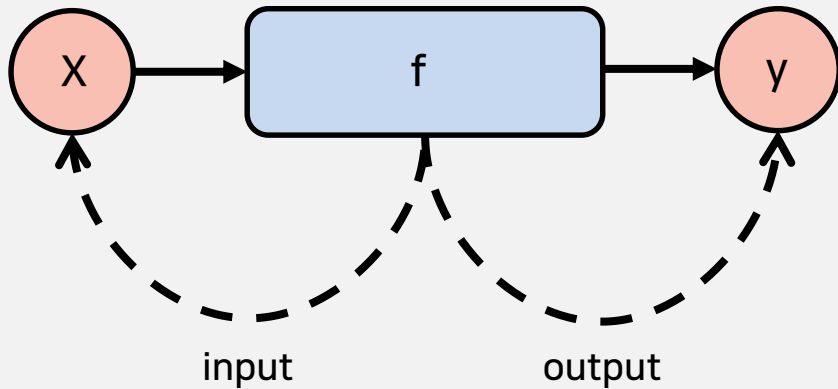


○ 역전파



7단계: 역전파 자동화 & steps/steps07.py

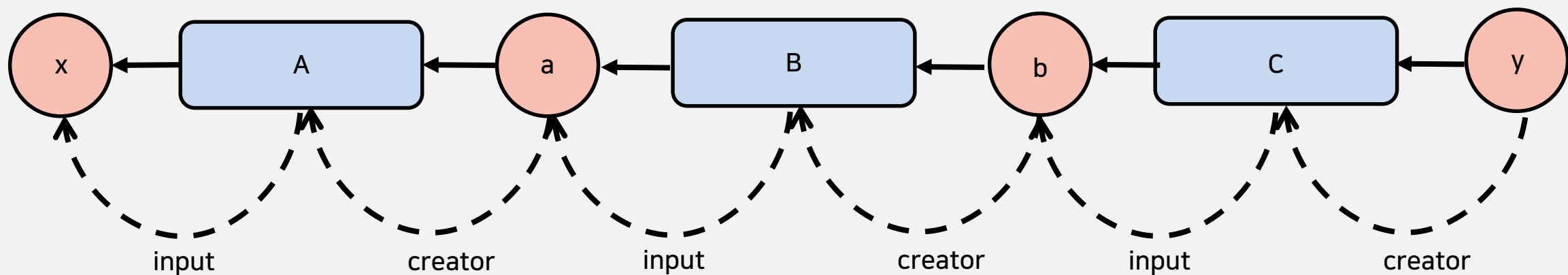
○ 변수와 함수와의 관계



○ assert 문: 평가 결과가 True가 아니면 예외 발생

7단계: 역전파 자동화 & steps/steps07.py

○ 계산 그래프 역 추적



8단계: 재귀에서 반복문으로 & steps/steps08.py, helper/recursive_call.py

○ 반복문의 장점

- 재귀는 함수를 재귀적으로 호출할 때마다 중간 결과를 메모리에 유지스택에 쌓으면서 처리
- 처리 효율도 우수
- 이 때, 반복문에서는 For-Loop 대신, while 반복문 사용

9단계: 함수를 더 편리하게 & steps/steps09.py

- 파이썬 함수로 이용하기
 - square & exp 함수 구현 완료
 - 함수를 연속으로 적용 가능

```
...  
y = square(exp(square(x)))  
...
```


9단계: 함수를 더 편리하게 & steps/steps09.py

○ Backward 메서드 간소화

```
...  
def backward(self):  
    if self.grad is None:  
        self.grad = np.ones_like(self.data)  
    funcs = [self.creator]  
    ...
```

- grad가 None이면 자동으로 미분값 생성, self.data가 스칼라이면, self.grad가 스칼라가 됨
- np.ones_like(): Variable data와 grad의 데이터 타입을 같게 만들도록 하기 위한 것
: 예) self.data 32비트 부동소수점이면 grad도 32비트 부동 소수점

9단계: 함수를 더 편리하게 & steps/steps09.py

○ ndarray만 취급하기

```
...  
def __init__(self, data):  
    if data is not None:  
        if not isinstance(data, np.ndarray):  
            raise TypeError('{ }은(는) 지원하지 않습니다.'.format(type(data)))  
...  

```

- as_array() 함수: Variable은 항상 ndarray() 가정하고 있어서, ndarray 인스턴스로 변환

10단계: 테스트 & steps/step10.py

- unittest의 사용
- Square 함수의 역전파 테스트
- 기울기 확인(Gradient checking)을 이용한 자동 테스트
 - $|a - b| \leq (atoll + rtol \times |b|) \Rightarrow$ 결괏값이 True 반환해야 함

```
w@HKIT203-LECT MINGW64 ~/Desktop/dl_framework (main)
$ python -m unittest steps/step10.py
...
-----
Ran 3 tests in 0.008s

OK
```

제 2고지 자연스러운 코드로

11, 12단계 - 가변 길이 인수(순전파) & steps/step11~12.py

○ Function 클래스 수정

- 인수와 반환값의 타입을 리스트로 변환
- 리스트 내포(list comprehension) 사용됨
- 인수 앞에 별표(*) 붙임 -> 임의 개수의 인수(가변 길이 인수)를 건네 함수를 호출
- 함수를 호출할 때 별표를 붙이면 리스트 언팩(list unpack)이 일어남*
- : 예) `xs = [x0, x1]`일 때 `self.forward(*xs)`를 하면 `self.forward(x0, x1)`로 호출하는 것과 동일하게 동작

* <https://dojang.io/mod/page/view.php?id=2345>

13단계 - 가변 길이 인수(역전파) & steps/step13.py

○ Variable 클래스 수정

- 여러 개의 변수에 대응할 수 있도록 수정됨

- ① 출력 변수인 outputs에 담겨 있는 미분값들을 리스트에 담음
- ② 함수 f의 역전파를 호출 - f.backward(*gys) 인수에 별표를 붙여 호출
- ③ gxs가 튜플이 아니라면 튜플로 변환
- ④ 역전파로 전파되는 미분값을 Variable의 인스턴스 변수 grad에 저장함

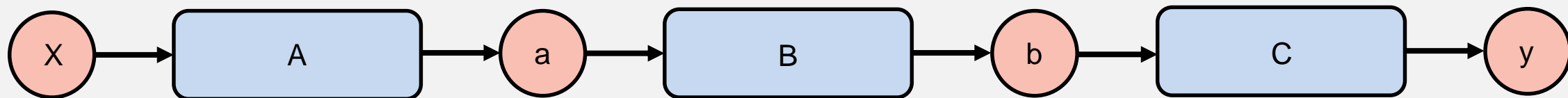
14단계: 같은 변수를 반복 사용 & steps/step13~14.py

- 문제 (1) : 역전파에서 미분값을 더해주는 과정에서 미분값을 그대로 대입
=> 미분의 합
 - 처음일 경우 : 지금과 같이 미분값 대입
 - 두번째 이후 : 이전 미분값과 전달받은 미분값을 더함
- 문제 (2) : 같은 변수로 다른 계산을 할 때
=> 미분값 초기화하는 `cleargrad`메서드 추가

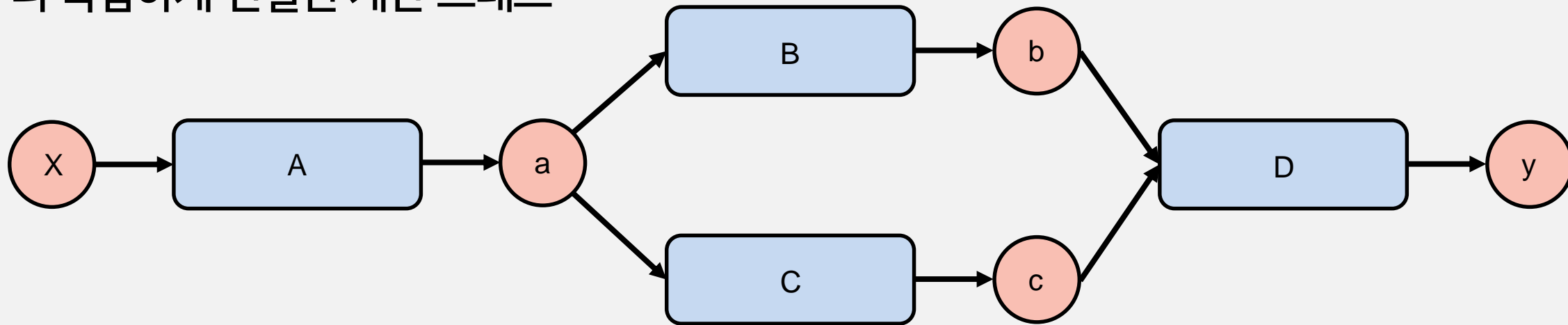


15~16단계: 복잡한 계산 그래프

- 일직선 계산 그래프 (앞에서 해왔던 것)



- 더 복잡하게 연결된 계산 그래프



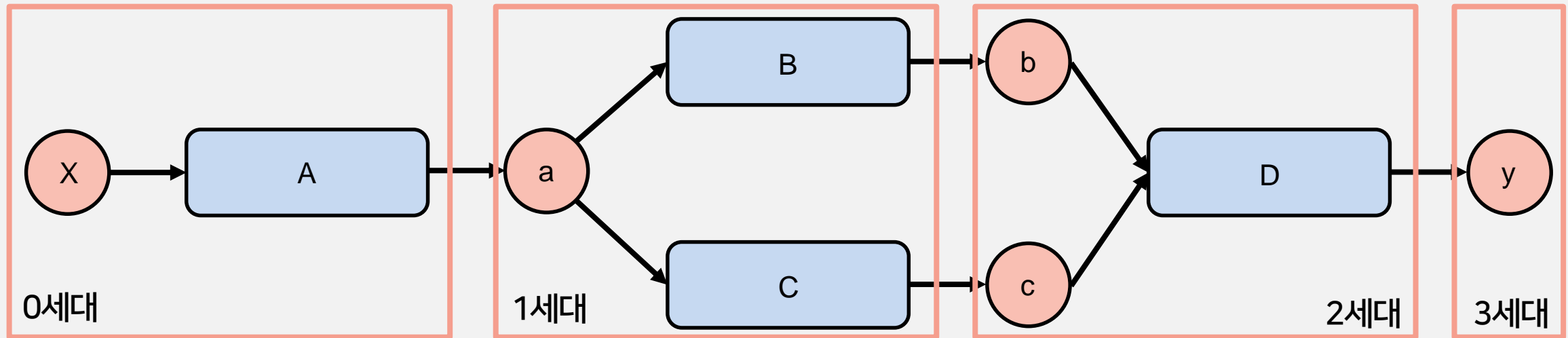
15~16단계: 복잡한 계산 그래프 & steps/step15~16.py

○ 역전파의 올바른 순서

- 같은 변수를 사용할 때 역전파는 출력쪽에서 전파되는 미분값을 더해야함
- 함수B와 C의 역전파를 모두 끝내고 함수A를 역전파 해야함. (B와 C의 순서는 상관없음)

구현방법

- 세대(generation) 추가 : 순전파에서 사용하는 함수의 순서를 기억
- 세대순으로 꺼내기 : 최근 세대의 함수부터 꺼내도록 함
- 중복을 피하기 위해 set()



17단계: 메모리 관리 및 순환 참조.

- 메모리 관리 방식 2가지
 1. 참조(Reference) 수 Count
 2. 세대(Generation)를 기준으로 쓸모없어진 객체(garbage)를 회수(collection)

17단계: 메모리 관리 및 순환 참조.

○ 참조(Reference) 수 Count

1. 모든 클래스 객체는 생성 시 참조 카운트가 0인 상태로 생성
2. 생성된 객체를 참조하는 프로그래밍 문법이 발생할 경우 카운트 1만큼 증가 (Edge 생성)
 - ※ 참조 카운트가 증가되는 경우(일부)
 - 대입 연산자 사용
 - 함수에 인수로 전달
 - 컨테이너 (리스트, 튜플, 클래스 등)에 추가할 경우
3. 이와 대조적으로 객체에 대한 참조가 끊어질 경우 (Edge 소멸) 참조 카운트 1만큼 감소

17단계: 메모리 관리 및 순환 참조.

○ 참조(Reference) 수 Count

```
# 객체 생성 (참조 카운트 0)
```

```
a = obj()
```

```
b = obj()
```

```
c = obj()
```

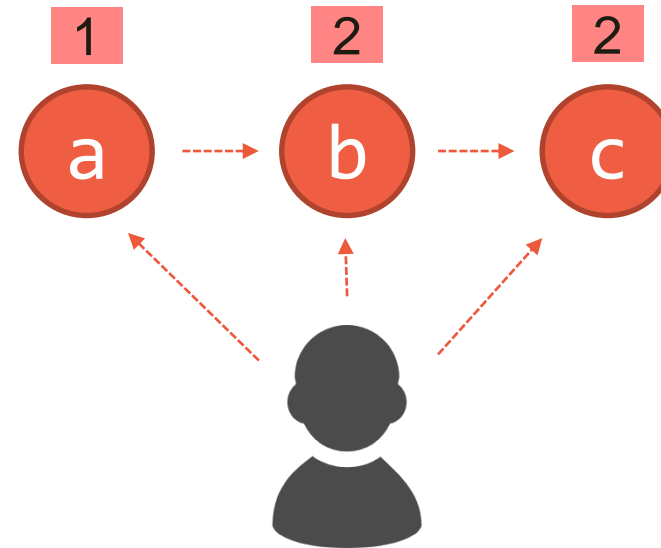
```
# 대입 연산자 사용
```

```
a.b = b
```

```
b.c = c
```

```
# 대입 연산자를 활용하여 객체 소멸
```

```
a = b = c = None
```



어떤 경우에 의해 참조된 경우 참조 수 Count. b와 c는 각각 대입연산자와 class 멤버변수로 호출되어 2번 참조됨

17단계: 메모리 관리 및 순환 참조.

○ 참조(Reference) 수 Count

```
# 객체 생성 (참조 카운트 0)
```

```
a = obj()
```

```
b = obj()
```

```
c = obj()
```

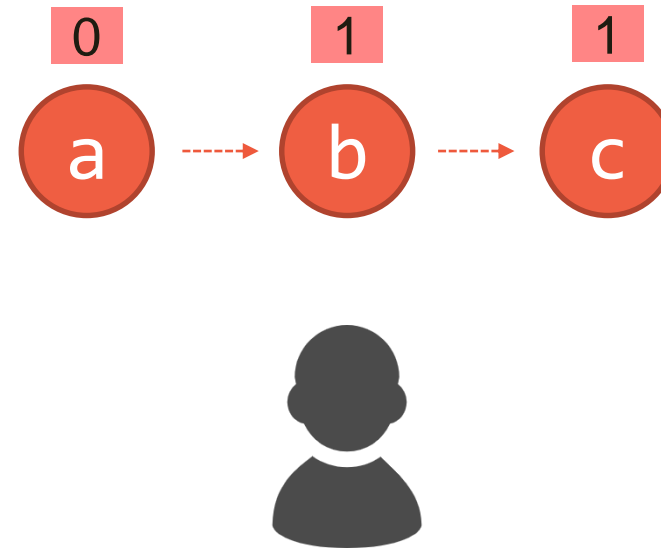
```
# 대입 연산자 사용
```

```
a.b = b
```

```
b.c = c
```

```
# 대입 연산자를 활용하여 객체 소멸
```

```
a = b = c = None
```



참조되어 인스턴스화되었던 a, b, c 클래스에 대해 메모리 해제 작업 시 참조 수 Count 확인

a, b, c가 각각 0, 1, 1의 값을 처음에 나타내고 b, c는 순차적으로 앞 노드(클래스 객체)에 의해 메모리 할당이 해제됨

17단계: 메모리 관리 및 순환 참조.

○ 순환 참조*

```
# 객체 생성 (참조 카운트 0)
```

```
a = obj()
```

```
b = obj()
```

```
c = obj()
```

```
# 대입 연산자 사용
```

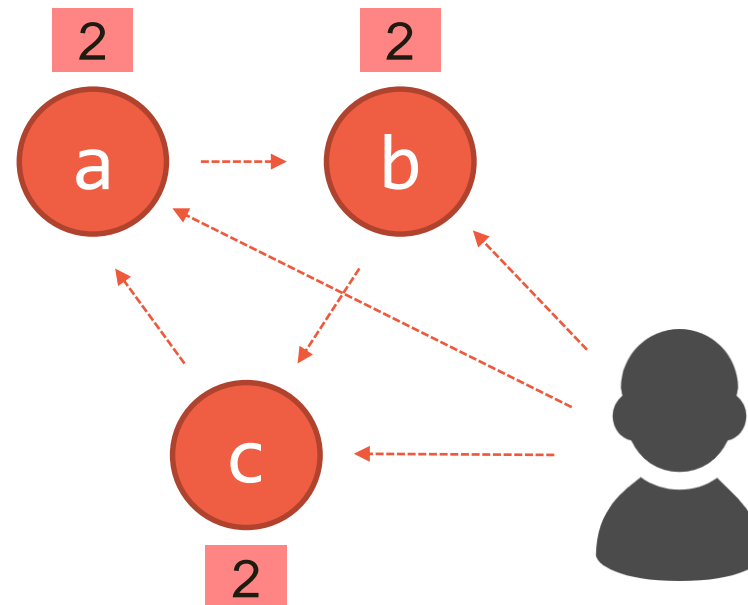
```
a.b = b
```

```
b.c = c
```

```
c.a = a
```

```
# 대입 연산자를 활용하여 객체 소멸
```

```
a = b = c = None
```



a, b, c 객체가 서로 순환하는 구조를 가진 경우 참조 count.

17단계: 메모리 관리 및 순환 참조.

○ 순환 참조

```
# 객체 생성 (참조 카운트 0)
```

```
a = obj()
```

```
b = obj()
```

```
c = obj()
```

```
# 대입 연산자 사용
```

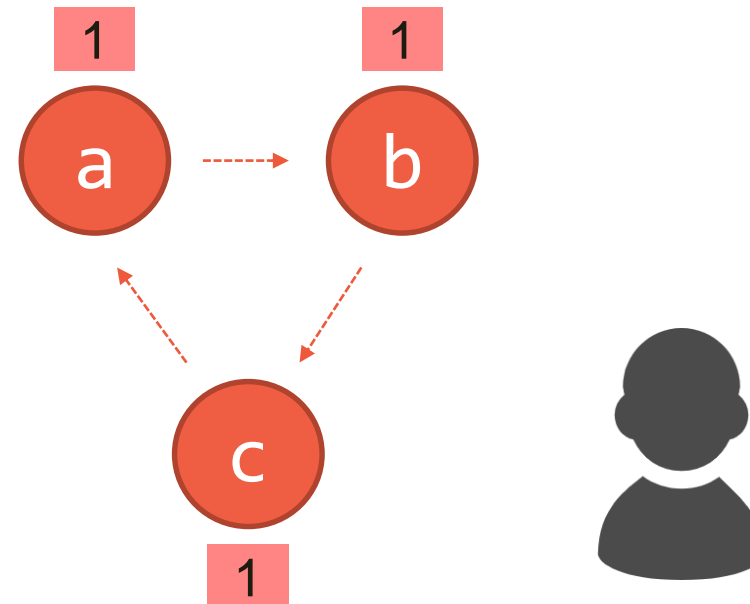
```
a.b = b
```

```
b.c = c
```

```
c.a = a
```

```
# 대입 연산자를 활용하여 객체 소멸
```

```
a = b = c = None
```



세 객체에 대해 메모리 해제를 시도했으나, 순환 참조에 의해 메모리 해제가 불가. (메모리 누수)

17단계: 메모리 관리 및 순환 참조.

○ weakref

```
>>> import weakref
>>> import numpy as np

>>> a = np.array([1, 2, 3])
>>> b = weakref.ref(a) # weakref 객체 생성

>>> b # b 메모리 주소 확인
<weakref at 0x103b7f048; to 'numpy.ndarray' at 0x103b67e90>
>>> b() # b 객체 내장데이터 확인
[1 2 3]
```

weakref 모듈을 사용하여 약한 연결고리를 가지는 순환 참조 객체를 생성

17단계: 메모리 관리 및 순환 참조.

○ weakref

```
>>> import weakref
>>> import numpy as np

>>> a = np.array([1, 2, 3])
>>> b = weakref.ref(a) # weakref 객체 생성

>>> b # b 메모리 주소 확인
<weakref at 0x103b7f048; to 'numpy.ndarray' at 0x103b67e90>
>>> b() # b 객체 내장데이터 확인
[1 2 3]

>>> a = None
>>> b # a 컨테이너에 대한 참조가 사라지면서 weakref status가 dead로 변경
<weakref at 0x103b7f048; dead>
```

17단계: 메모리 관리 및 순환 참조.

○ weakref & step/step17.py

```
import weakref

class Function:
    def __call__(self, *inputs):
        ...
        self.outputs = [weakref.ref(output) for output in outputs]
        ...

class Variable:
    ...
    def backward(self):
        ...
        gys = [output().grad for output in f.outputs]
        ...
```

17단계: 메모리 관리 및 순환 참조.

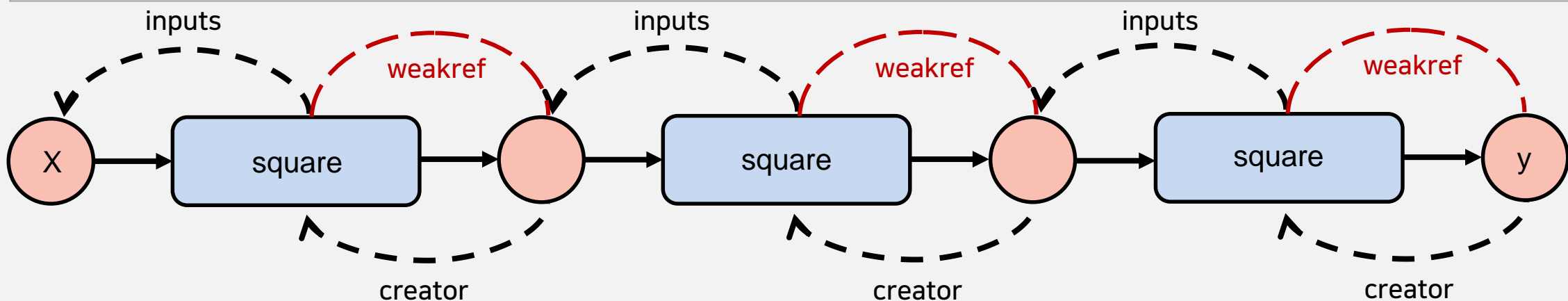
○ weakref & step/step17.py

```
import numpy as np
```

```
for i in range(10):
```

```
    x = Variable(np.random.randn(10000))
```

```
    y = square(square(square(x)))
```



17단계: 메모리 관리 및 순환 참조.

○ Garbage Collection *

- 세대(Generation)를 기준으로 쓸모없어진 객체(garbage)를 회수(collection)
- 장점: (3가지 케이스에 대해서) 메모리 수집 편리
 1. 유효하지 않은 포인터에 대한 접근
 2. 이중 해제
 3. 메모리 누수
- 단점
 1. GC에 의존하므로 메모리 해제 시점을 알아도 GC를 사용하므로 오버헤드 발생
 2. 메모리의 해제 시점을 정확하게 판단할 수 없다.

18단계: 메모리 절약 모드

○ 할당된 메모리 해제 작업

```
class Variable:
    ...
    def backward(self, retain_grad = False):
        ...
        if not retain_grad:
            for y in f.outputs:
                y().grad = None # 불필요한 메모리 해제, y의 기울기는 Framework 계산 시를 제외하고 활용되지 않음
```

Variable 클래스의 backward 메서드에 retain_grad 인자를 추가하는 것만으로도 불필요한 메모리 사용을 억제하는 효과

18단계: 메모리 절약 모드

○ Config 클래스 활용 - Flag 설정

```
class Config:
    enable_backprop = True

class Function:
    def __call__(self, *inputs):
        ...
        if Config.enable_backprop:
            ...
        ...
```

설정 데이터는 단 한 군데 지정하는게 전체적인 측면에서 유리하므로 Config 클래스를 인스턴스화하지 않고 '클래스' 그대로 활용

18단계: 메모리 절약 모드

- with 문을 활용한 모드 전환
 - Python의 들여쓰기를 활용하여 후처리를 자동으로 진행해주는 구문 생성
 - 대표적인 예: with < open, close >
 - 데코레이션(@)을 활용하여 [contextlib.contextmanager](#)에 접근

18단계: 메모리 절약 모드

○ with 문을 활용한 모드 전환

```
import contextlib

@contextlib.contextmanager
def using_config(name, value):
    old_value = getattr(Config, name) # 기존 설정내용 저장
    setattr(Config, name, value) # 새로운 설정내용으로 작업 수행
    try:
        yield # with문으로 호출 시 body 부분 작업을 수행하는 영역
    finally:
        setattr(Config, name, old_value) # 기존 설정내용으로 복구
```

contextlib 모듈 내에서 callable 함수 객체를 받아서 decorator에 의해 처리
위 코드의 경우 기존 설정을 백업해둔 상태로 새로운 작업을 수행하고 복구하는 함수.

18단계: 메모리 절약 모드

○ with 문을 활용한 모드 전환

```
with using_config('enable_backprop', False): # Config class's 멤버변수(str), name(bool)
    x = Variable(np.array(2.))
    y = square(x)
```



설정 간소화

```
Def no_grad():
    return using_config('enable_backprop', False)
```

```
with no_grad():
    x = Variable(np.array(2.))
    y = square(x)
```

with문을 활용하여 후처리 진행 + 반복 수행 시 번거로움을 방지하고자 no_grad 함수 생성

19단계: 변수 사용성 개선 & steps/step19.py

- 변수 이름 지정
 - 변수의 구분을 위해 변수 이름을 저장
 - 계산 그래프를 시각화 할 때 변수이름을 그래프에 표시할 수 있음
 - 지정하지 않을 경우 None로 할당
- ndarray 인스턴스 변수
 - shape : 다차원 배열의 형상
 - ndim : 차원 수
 - size : 원소 수
 - dtype : 데이터 타입
 - @property라는 데코레이터를 사용해 메서드를 인스턴스 변수처럼 사용
ex) x.shape

19단계: 변수 사용성 개선 & steps/step19.py

- len 함수와 print 함수
 - len : 리스트 등의 안에 포함된 원소 수를 반환
 - print : Variable안에 데이터 내용을 출력하는 기능
 - __len__와 같이 특수 메서드로 정의하여 함수처럼 사용
ex) len(x)

20단계: 연산자 오버로드(1) & steps/step20.py

○ 연산자 오버로드(operator overlaod)

- 연산자(+, *등) 대응
- 특수 메서드를 정의함으로써 사용자 지정 함수가 호출되도록 함

ex) `__mul__ => *`

`__add__ => +`

- 파이썬에서는 함수도 객체이므로 함수 자체를 할당할 수 있음

ex) Variable 의 `__mul__`를 호출할 때 파이썬의 `mul`함수 부름

`Variable.__add__ = add`

`Variable.__mul__ = mul`

21단계: 연산자 오버로드(2) & steps/step21.py

- 연산자 오버로드(operator overlaod)
 - 앞에서 $a * b$ or $a + b$ 같은 연산자를 사용한 코드 작성할 수 있게 됨
 - 1) $a * \text{np.array}(2.0)$ 같은 ndarray 인스턴스와 사용할 수 없음
 - 2) $a + 3$ 같은 수치 데이터와 사용할 수 없음



ndarray, int, float도 사용할 수 있게 하자

21단계: 연산자 오버로드(2) & steps/step21.py

○ ndarray, float, int와 함께 사용하기

- ndarray 인스턴스를 Variable인스턴스로 변환 -> Variable인스턴스로 통일
- float, int를 ndarray 인스턴스로 변환 -> 이후 Function 클래스에서 Variable 인스턴스로 다시
변환되기 때문에 결과적으로 전부 Variable인스턴스로 통일됨
- 하지만 지금의 이항 연산자 * 는 좌항의 인스턴스에 속해있는 메서드를 통해서만 호출됨

21단계: 연산자 오버로드(2) & steps/steps22.py

○ 좌항에 int, float가 있을 경우

- 구현되어 있지 않기 때문에 특수 메서드를 호출하지 못함

해결 : 피연산자의 위치에 따라 호출되는 특수 메서드를 지정하자

ex) 곱셈의 경우

- 피연산자가 좌항 -> `'__mul__'`
- 피연산자가 우항 -> `'__rmul__'` (r은 right의 r)

21단계: 연산자 오버로드(2) & steps/steps22.py

○ 좌항에 ndarray 가 있을 경우

- 좌항의 ndarray의 인스턴스를 통해 연산자가 호출됨

하지만 Variable 인스턴스로 통일 해주어야 해서 Variable 인스턴스의 특수메서드를 호출하길 원함

해결 : 연산자의 우선순위를 두자

Variable 인스턴스의 연산자 우선순위를 ndarray 인스턴스의 연산자 우선순위보다 높일 수 있음

ex) 덧셈의 경우

```
...  
class Variable:  
    __array_priority__ = 200  
...
```


22단계: 연산자 오버로드(3) & steps/steps23.py

○ 특수 메서드

특수 메서드	설명	예
<code>_neg_(self)</code>	부호 변환 연산자	<code>-self</code>
<code>_sub_(self, other)</code>	뺄셈 연산자	<code>self - other</code>
<code>_rsub_(self, other)</code>	역순 뺄셈 연산자	<code>other - self</code>
<code>_truediv_(self, other)</code>	나눗셈 연산자	<code>self / other</code>
<code>_rthredic_(self, other)</code>	역순 나눗셈 연산자	<code>other / self</code>
<code>_pow_(self, other)</code>	거듭제곱 연산자	<code>self ** other</code>

23단계 - 패키지로 정리 & steps/step13.py

○ 모듈 *

- 파이썬 파일, 특히 임포트(import)하여 사용하는 것을 가정하고 만들어진 파이썬 파일

○ 패키지

- 여러 모듈을 묶은 것

○ 라이브러리

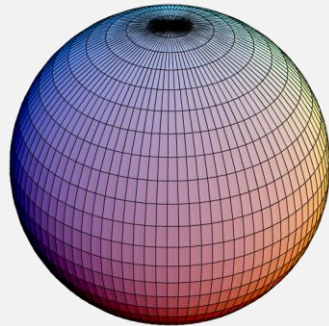
- 여러 패키지를 묶은 것, 때로는 패키지를 가리켜 '라이브러리'라고 부르기도 함

* <https://offbyone.tistory.com/106>

24단계: 복잡한 함수의 미분 (최적화 문제 - Benchmark 함수)

○ Sphere 함수

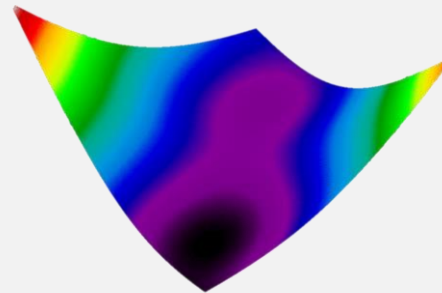
$$z = x^2 + y^2$$



Function 클래스에 정의된 모든 사칙연산에 대해 **z.Backward()** 하나로 역전파 해결

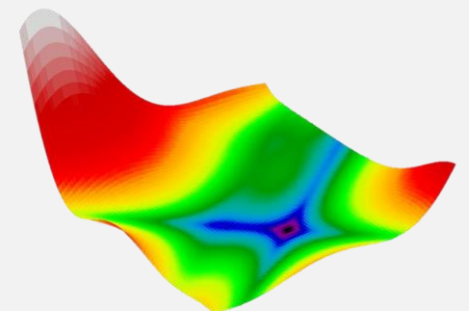
○ Matyas 함수

$$z = 0.26 * (x^2 + y^2) - 0.48 * x * y$$



○ Goldstein-Price 함수

$$z = (1 + (x + y + 1)^2 * (19 - 14x + 3x^2 - 14y + 6xy + 3y^2)) * (30 + (2x - 3y)^2 * (18 - 32x + 12x^2 + 48y - 36xy + 27y^2))$$



참고 문헌

- 코딩도장, 위치 인수와 리스트 언패킹 사용
<https://dojang.io/mod/page/view.php?id=2345>
- 모두의 코드. Weak pointer (weak reference)
<https://modoocode.com/252>
- 위키피디아. Garbage Collection
Wikipedia.org → Garbage Collection
- offbyone, 모듈과 패키지 사용하기
<https://offbyone.tistory.com/106>