STEP 1. 상자로서의 변수

```
class Variable:
    def __init__(self, data):
        self.data = data
```

주어진 인수를 인스턴스 변수 data에 대입.

```
import numpy as np

data = np.array(1.0)
x = Variable(data)
print(x.data)
```

⇒ 1.0

x → Variable 인스턴스

실제 데이터는 x에 담겨있음.

STEP 2. 변수를 낳는 함수

- Variable 인스턴스를 변수로 다룰 수 있는 함수 구현

```
class Function:
    def *1 __call__(self, input):
        x = input.data           # 데이터를 꺼냄.
        y = x ** 2               # 실제 계산
        output = Variable(y)     # Variable 형태로 되돌림.
        return output

x = Variable(np.array(10))
f = Function()
y = f(x)

print(type(y))     # type() 함수: 객체의 클래스를 알려줌.
print(y.data)
```

↘ y의 클래스는 Variable 이며,
데이터는 y.data 에 저장되어 있음.

→ Function 클래스 수정

RULE

- Function 클래스는 기반클래스, 모든 함수에 공통되는 기능 구현
- 구체적인 함수는 Function 클래스를 상속한 클래스에서 구현

```
class Function:
    def __call__(self, input):
        x = input.data
        y = self.forward(x)    # 구체적인 계산은 forward 메서드에서
        output = Variable(y)   # Variable 형태로 되돌림
        return output

    def forward(self, x):
        raise NotImplementedError()   → forward 메서드의 구체적인 동작을 하위 클래스에서 구현
```

↓ Function 클래스를 상속하여 입력값을 제공하는 클래스 구현

```
class Square(Function):
    def forward(self, x):
        return x ** 2


x = Variable(np.array(10))
f = Square()
y = f(x)

print(type(y))
print(y.data)
```

STEP 3. 함수 연결

```
class Exp(Function):
    def forward(self, x):
        return np.exp(x)
```

Function 클래스의 __call__ 메서드는 입력과 출력이 모두 Variable 인스턴스이므로

DeZero 함수들을 연이어 사용 가능.

Example :

$y = (e^{x^2})^2$

```
A = Square()
B = Exp()
C = Square()

x = Variable(np.array(0.5))
a = A(x)
b = B(a)
y = C(b)

print(y.data)
```

STEP 4. 수치미분

f(x) 라는 함수에 대한 미분

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

극한

$y = f(x)$ 그래프의 두 점을 지나는 직선의 기울기

수치미분이란?

- 미세한 차이 (0.0001 = 1e-4)를 이용해 함수의 변화량을 구하는 방법.
- 작은값을 이용하여 '진정한 미분'을 근사
- 근사오차를 줄이는 방법? '중앙차분 (centered difference)'
  - 직선의 기울기 : $\frac{f(x+h) - f(x-h)}{2h}$

numerical_diff (f, x, eps = 1e-4)

- f : 미분의 대상이 되는 함수 (Function 의 인스턴스)
- x : 미분을 계산하는 변수 (Variable 인스턴스)
- eps (= epsilon) : 작은값

```
def numerical_diff (f, x, eps = 1e-4):
    x0 = Variable (x.data - eps)
    x1 = Variable (x.data + eps)
    y0 = f(x0)
    y1 = f(x1)
    return (y1.data - y0.data) / (2 * eps)
```

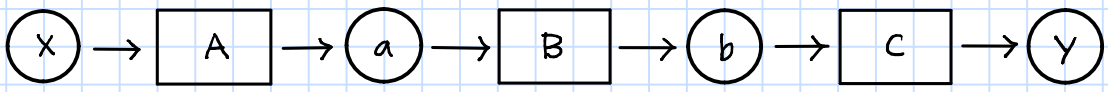
Square 클래스를 대상으로 미분

```
f = Square ()
x = Variable (np.array (2.0))
dy = numerical_diff (f,x)
print (dy)
```

STEP 5. 역전파 이론

- 역전파를 통해 미분을 효율적으로 계산 & 결과값 오차도 더 적음.

- 합성함수 $y = F(x)$

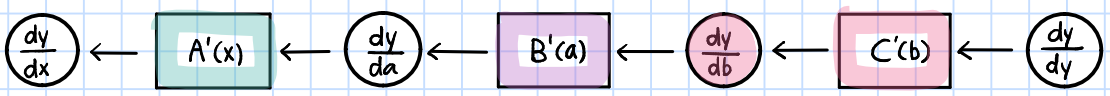


  - $a = A(x)$, $b = B(a)$, $y = C(b)$
  - x에 대한 y의 미분 $\frac{dy}{dx} = \frac{dy}{db}\frac{db}{da}\frac{da}{dx}$ → $\frac{dy}{dy}\frac{dy}{db}\frac{db}{da}\frac{da}{dx}$

역방향 (출력 → 입력)으로 계산

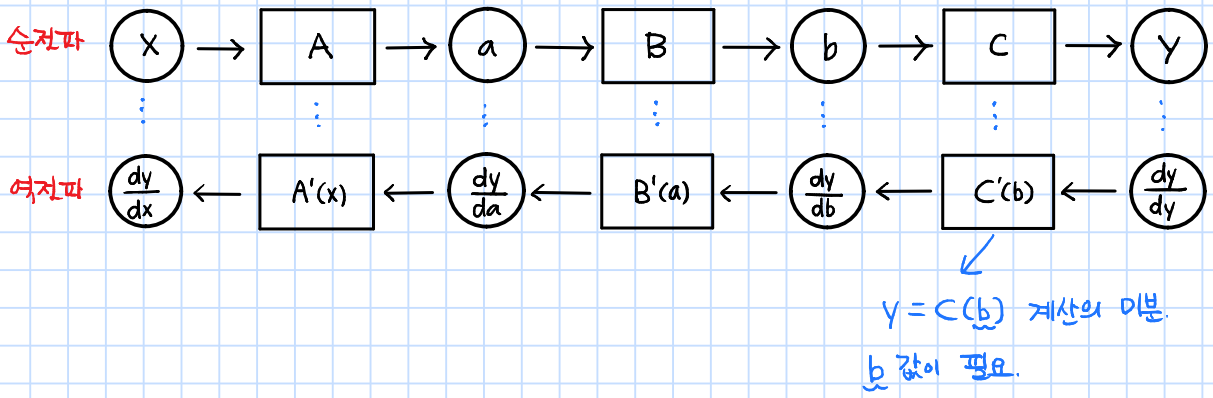
  - $\frac{dy}{dx} = \left(\left(\frac{dy}{dy}\frac{dy}{db}\right)\frac{db}{da}\right)\frac{da}{dx}$

$\frac{dy}{dy}\frac{dy}{db}$: $C'(b)$, $\frac{db}{da}$: $B'(a)$, $\frac{da}{dx}$: $A'(x)$



  - 변수 y, b, a, x 에 대한 미분값이 오른쪽 → 왼쪽 전파. (역전파)
  - 전파되는 데이터가 모두 y의 미분값!
  - 손실함수의 각 매개변수에 대한 미분을 계산

    : 출력 → 입력 방향으로 전파하여 한 번의 전파만으로 모든 매개변수에 대한 미분을 계산할 수 있음.



- 역전파에는 순전파 시 이용한 데이터가 필요.
- 역전파를 구현하려면 먼저 순전파를 하고, 각 함수가 입력변수 (ex. x, a, b)의 값을 기억해두어야 함

STEP 6. 수동 역전파

6.1. Variable 클래스 추가 구현

```
class Variable :
    def __init__(self, data):
        self.data = data
        self.grad = None    # 미분값 , None 초기화 ( 실제 역전파 시 미분값 계산하여 대입)
```

6.2 Function 클래스 추가 구현

- 미분을 계산하는 역전파 (backward 메서드)
- forward 호출 시 건네받은 Variable 인스턴스 유지

```
class Function :
    def __call__(self, input):
        x = input.data
        y = self.forward(x)
        output = Variable(y)
        self.input = input    # 입력변수를 기억(보관)
        return output

    def forward(self, x):
        raise NotImplementedError()

    def backward(self, gy):
        raise NotImplementedError()
```

6.3. Square와 Exp 클래스 추가 구현

```
class Square(Function):
    def forward(self, x):
        y = x ** 2
        return y

    def backward(self, gy):
        x = self.input.data
        gx = 2 * x * gy
        return gx
```

$y = x^2$ 의 미분

ndarray 인스턴스
(출력 쪽에서 전해지는 미분값을 전달하는 역할)

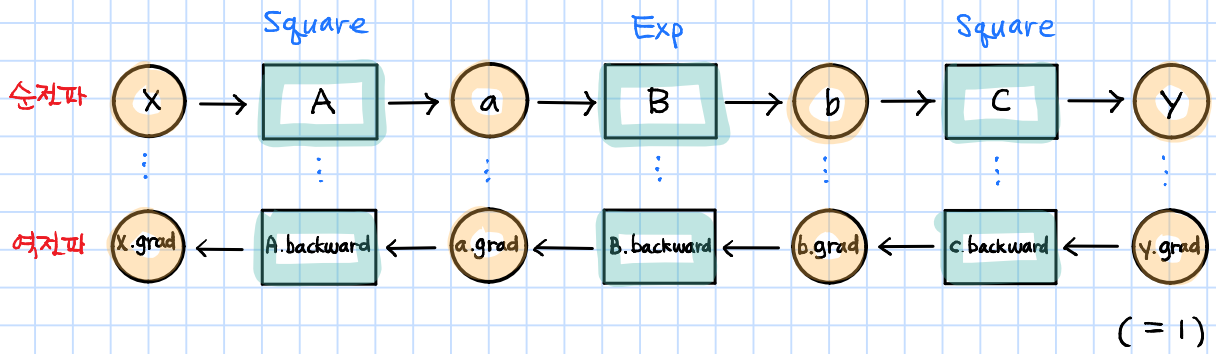
$y = x^2$ 의 미분 $\frac{dy}{dx} = 2x$

```
class Exp(Function):
    def forward(self, x):
        y = np.exp(x)
        return y

    def backward(self, gy):
        x = self.input.data
        gx = np.exp(x) * gy
        return gx
```

$y = e^x$ 의 미분 $\frac{dy}{dx} = e^x$

6.4 역전파 구현



순전파 코드

```
A = Square()
B = Exp()
C = Square()

x = Variable(np.array(0.5))
a = A(x)
b = B(a)
y = C(b)
```

역전파 구현

```
y.grad = np.array(1.0)
b.grad = C.backward(y.grad)
a.grad = B.backward(b.grad)
x.grad = A.backward(a.grad)

print(x.grad)
```

역전파는 $\frac{dy}{dy} = 1$ 에서 시작.

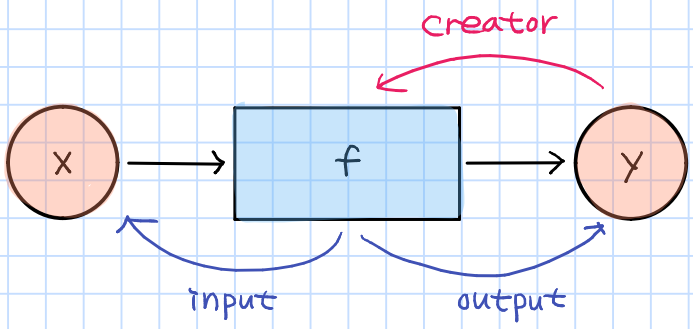
C → B → A 순으로 backward 메서드 호출.

STEP 7. 역전파 자동화

Define-by-Run !

7.1 역전파 자동화의 시작

- 함수와 변수의 관계



- 추가내용

```
class Variable:
    def __init__(self, data):
        self.creator = None
    def set_creator(self, func):
        self.creator = func

class Function:
    def __call__(self, input):
        output.set_creator(self)        → output에 창조자를 기억시켜.
                                          ('연결'을 동적으로 만들기 위함의 핵심)
        self.output = output
```

- '연결'된 Variable과 Function이 있다면 계산 그래프를 거꾸로 거슬러 올라갈 수 있음.

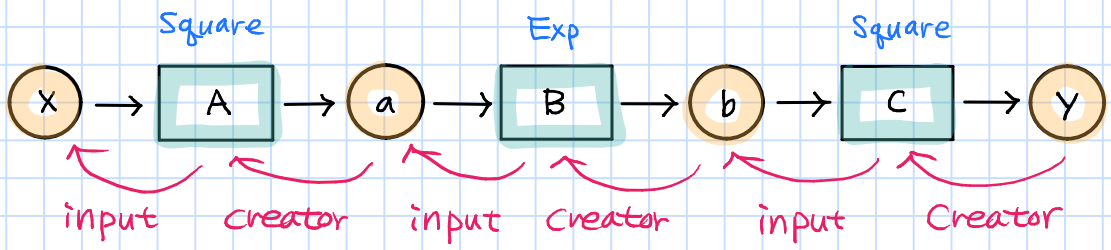
```
A = Square()
B = Exp()
C = Square()

x = Variable(np.array(0.5))
a = A(x)
b = B(a)
c = C(b)

# 계산그래프의 노드들을 거꾸로 거슬러 올라간다.
assert y.creator == C
assert y.creator.input == b
assert y.creator.input.creator == B
assert y.creator.input.creator.input == a
assert y.creator.input.creator.input.creator == A
assert y.creator.input.creator.input.creator.input == x
```

- 계산 그래프는 함수와 변수 사이의 연결로 구성됨.
- '연결'이 실제 계산을 수행하는 시점 (순전파로 데이터를 흘려보낼 때) 에 만들어짐.
  → "Define-by-Run"
- "Linked List"

7.2 역전파 도전




```
y.grad = np.array(1.0)

C = y.creator      # 1. 함수를 가져온다.
b = C.input        # 2. 함수의 입력을 가져온다.
b.grad = C.backward(y.grad)    # 3. 함수의 backward 메서드를 호출

B = b.creator
a = B.input
a.grad = B.backward(b.grad)

A = a.creator
x = A.input
x.grad = A.backward(x.grad)

print(x.grad)
```

7.3 backward 메서드 추가.

```
def backward(self):
    f = self.creator          # 1. 함수를 가져온다.
    if f is not None:
        x = f.input           # 2. 함수의 입력을 가져온다.
        x.grad = f.backward(self.grad)    # 3. 함수의 backward 메서드를 호출한다.
        x.backward()          # 3. 하나 앞 변수의 backward 메서드를 호출한다. (재귀)
```

- 역전파 자동 실행.

```
A = Square()
B = Exp()
C = Square()

x = Variable(np.array(0.5))
a = A(x)
b = B(a)
c = C(b)

# 역전파
y.grad = np.array(1.0)
y.backward()
print(x.grad)
```

STEP 8. 재귀에서 반복문으로.

8.2. 반복문을 이용한 구현

```
def backward(self):
    funcs = [self.creator]
    while funcs:
        f = funcs.pop()          # 맨 마지막 함수
        x, y = f.input, f.output    # 함수의 입력과 출력을 가져온다.
        x.grad = f.backward(y.grad)    # backward 메서드를 호출

        if x.creator is not None:
            funcs.append(x.creator)
```

STEP 9. 함수를 더 편리하게

9.1 파이썬 함수로 이용하기

```
def square(x):
    return Square()(x)

def exp(x):
    return Exp()(x)
```

- 함수 적용

```
x = Variable(np.array(0.5))
a = square(x)
b = exp(a)
y = square(b)

y.grad = np.array(1.0)
y.backward()
print(x.grad)
```

↓ 함수 연속적용.

```
x = Variable(np.array(0.5))
y = square(exp(square(x)))    # 연속하여 적용
y.grad = np.array(1.0)
y.backward()
print(x.grad)
```

9.2. backward 메서드 간소화

- y.grad = np.array(1.0) 코드 생략

```
x = Variable(np.array(0.5))
y = square(exp(square(x)))
y.backward()
print(x.grad)
```

9.3. ndarray 만 취급하기

```
def as_array(x):
    if np.isscalar(x):
        return np.array(x)
    return x
```

STEP 10. 테스트

10.1 파이썬 단위 테스트

```
import unitest

class SquareTest (unitest.TestCase):
    def test_forward (self):
        x = Variable (np.array (2.0))
        y = square (x)
        expected = np.array (4.0)
        self. assertEqual (y.data, expected)
```

주어진 두 개체가 동일한지 판단

10.2. square 함수의 역전파 테스트

```
    def test_backward (self):
        x = Variable (np.array (3.0))
        y = square (x)
        y. backward ()
        expected = np. array (6.0)
        self. assertEqual (x.grad, expected)
```

10.3. 기울기 확인을 이용한 자동 테스트

- 기울기 확인 (gradient checking)
  - 수치미분으로 구현한 결과와 역전파로 구현한 결과를 비교하여
    그 차이가 크면 역전파 구현에 문제가 있다고 판단하는 검증기법.
  - 기댓값을 몰라도 입력값만 준비하면 되므로 테스트 효율을 높여줌

```
def numerical_diff (f, x, eps=1e-4):
    x0 = Variable (x.data - eps)
    x1 = Variable (x.data + eps)
    y0 = f(x0)
    y1 = f(x1)
    return (y1.data - y0.data) / (2 * eps)
```

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

        self.data = data
        self.grad = None    # 미분값 , None 초기화 ( 실제 역전파 시 미분값 계산하여 대입)
        self.creator =None

    def set_creator (self, func):
        self.creator = func

    def backward (self):
        if self.grad is None :
            self.grad = np.ones_like (self.data)

        funcs = [self.creator]
        while funcs :
            f = funcs.pop()      # 맨 마지막 함수
            x, y = f.input, f.output    # 함수의 입력과 출력을 가져온다.
            x.grad = f.backward (y.grad)   # backward 메서드를 호출

            if x.creator is not None :
                funcs . append (x.creator)
```

```
class Function:
    def __call__(self, input):
        x = input.data
        y = self.forward(x)
        output = Variable(as_array(y))      # output이 항상 ndarray 인스턴스가 되도록 보장
        output.set_creator(self)    # output에 creator를 설정
        self.input = input    # 입력변수를 기억(보관)
        self.output = output    # 출력 기억(보관)
        return output

    def forward(self, x):
        raise NotImplementedError()

    def backward(self, gy):
        raise NotImplementedError()
```

```
class Square(Function):
    def forward(self, x):
        y = x ** 2
        return y

    def backward(self, gy):
        x = self.input.data
        gx = 2 * x * gy
        return gx
```

2 * x : $y = x^2$의 미분

gy : ndarray 인스턴스 (출력 쪽에서 전해지는 미분값을 전달하는 역할)

→ $y = x^2$ 의 미분 $\frac{dy}{dx} = 2x$

```
class Exp(Function):
    def forward(self, x):
        y = np.exp(x)
        return y

    def backward(self, gy):
        x = self.input.data
        gx = np.exp(x) * gy
        return gx
```

→ $y = e^x$ 의 미분 $\frac{dy}{dx} = e^x$

```
class SquareTest(unittest.TestCase):
    def test_forward(self):
        x = Variable(np.array(2.0))
        y = square(x)
        expected = np.array(4.0)
        self.assertEqual(y.data, expected)
```

주어진 두 개체가 동일한지 판단

```
    def test_gradient_check(self):
        x = Variable(np.random.rand(1))      # 무작위 입력값 생성
        y = square(x)
        y.backward()
        num_grad = numerical_diff(square, x)    # 수치미분으로 계산
        flg = np.allclose(x.grad, num_grad)     # 구한 값이 거의 일치하는지 확인
        self.assertTrue(flg)
```

*1 : __call__

- 파이썬 특수 메소드
- f = Function() 처럼 함수의 인스턴스를 변수 f에 대입해두고,
  나중에 f(...) 처럼 __call__ 메서드 호출 가능