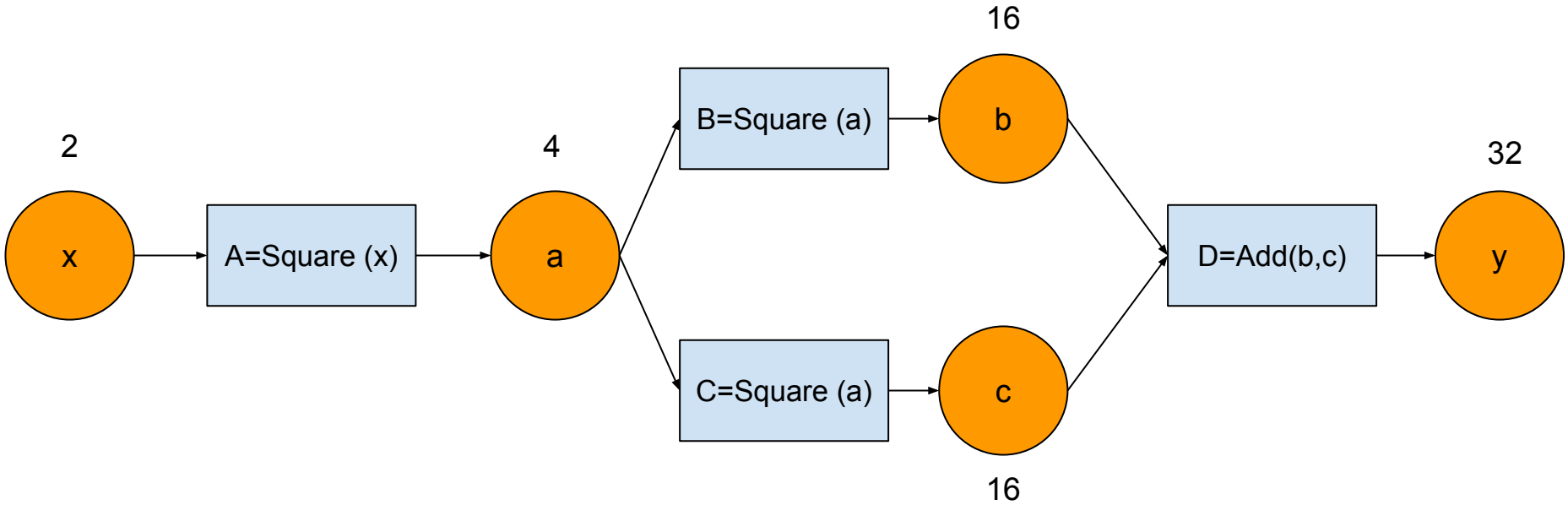
STEP15, 16

複雑な計算グラフ

現状の課題

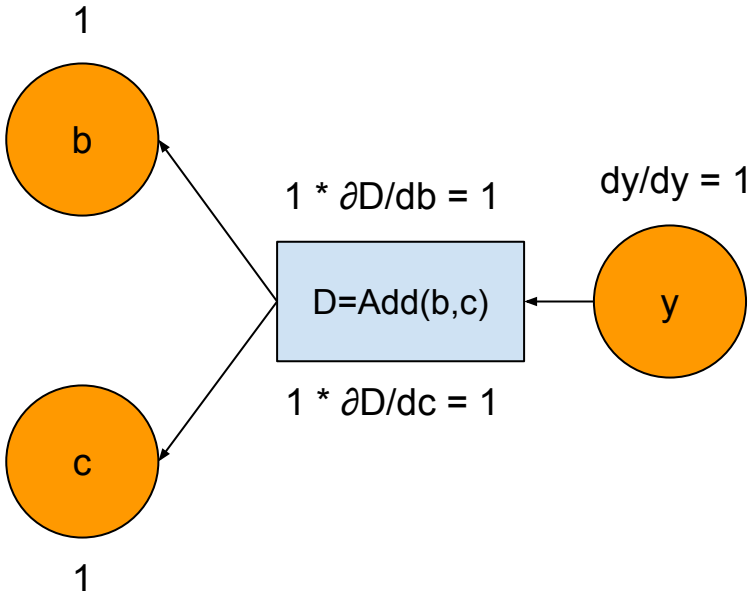
- 逆伝搬がただしく計算できない

まずは以下の順伝搬のケースを考える

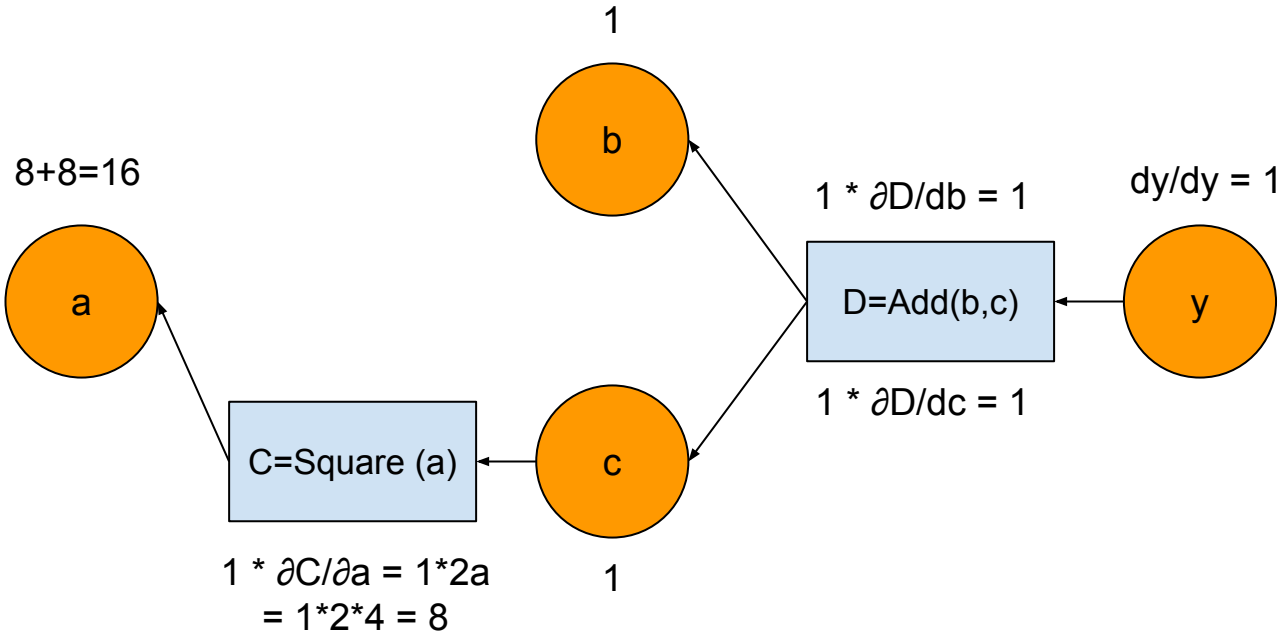


正しい逆伝搬

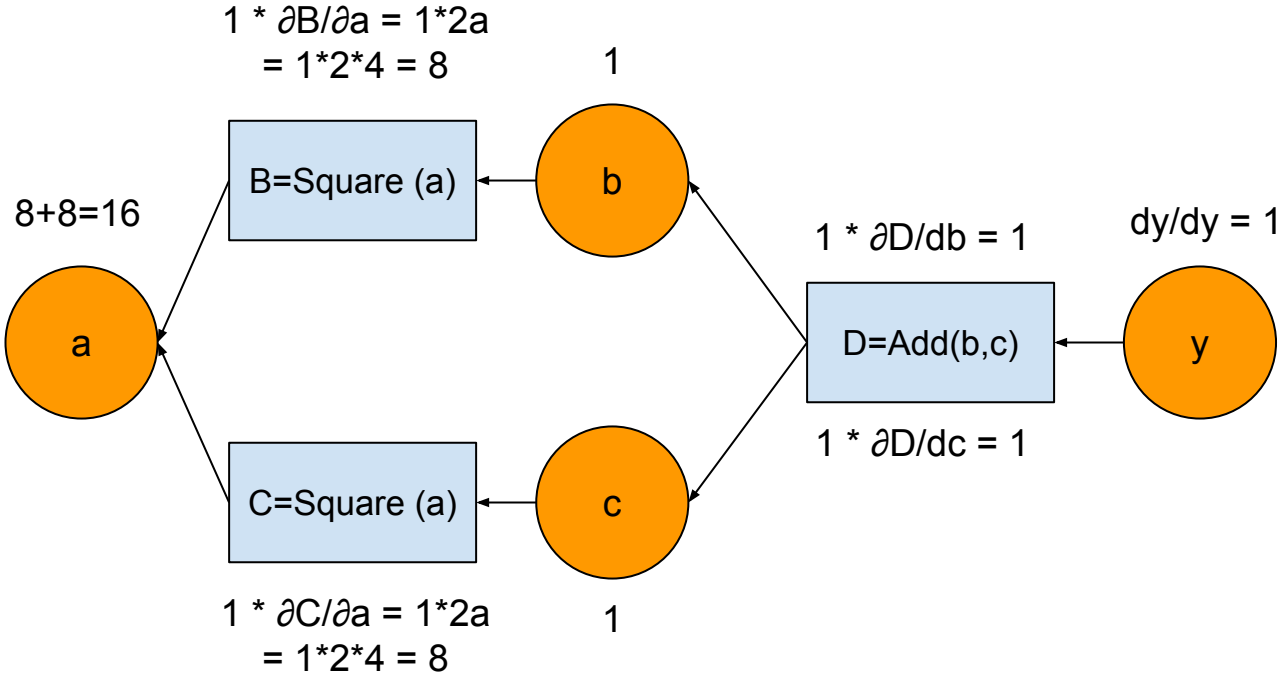
正しい逆伝搬



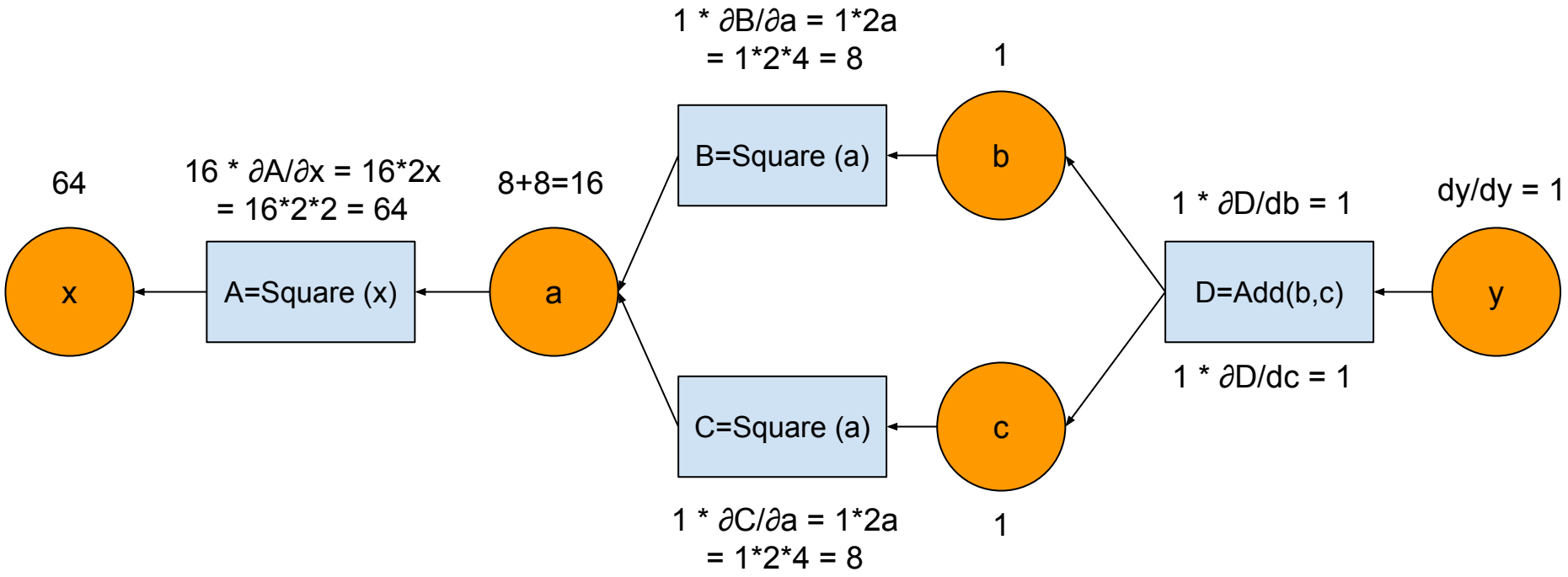
正しい逆伝搬



正しい逆伝搬

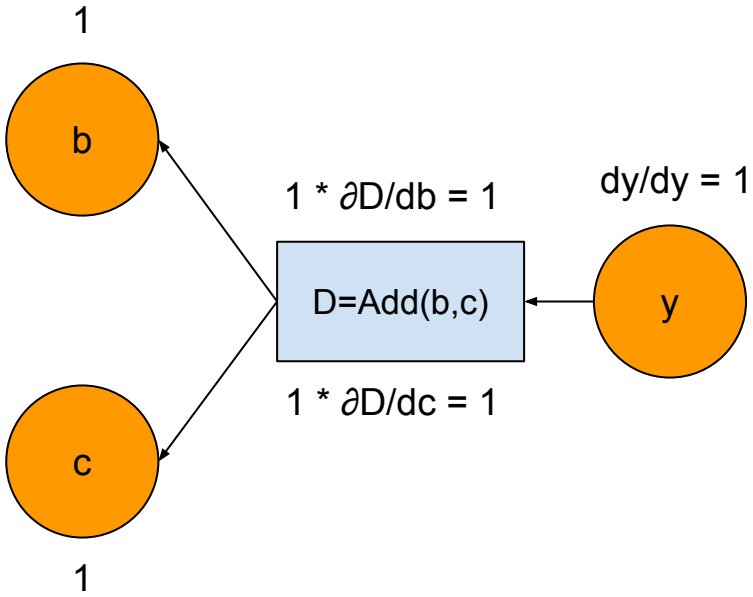

1 * ∂B/∂a = 1*2a
= 1*2*4 = 8
1
B=Square (a)
b
8+8=16
1 * ∂D/db = 1
dy/dy = 1
a
D=Add(b,c)
y
1 * ∂D/dc = 1
C=Square (a)
c
1 * ∂C/∂a = 1*2a
= 1*2*4 = 8
1

正しい逆伝搬

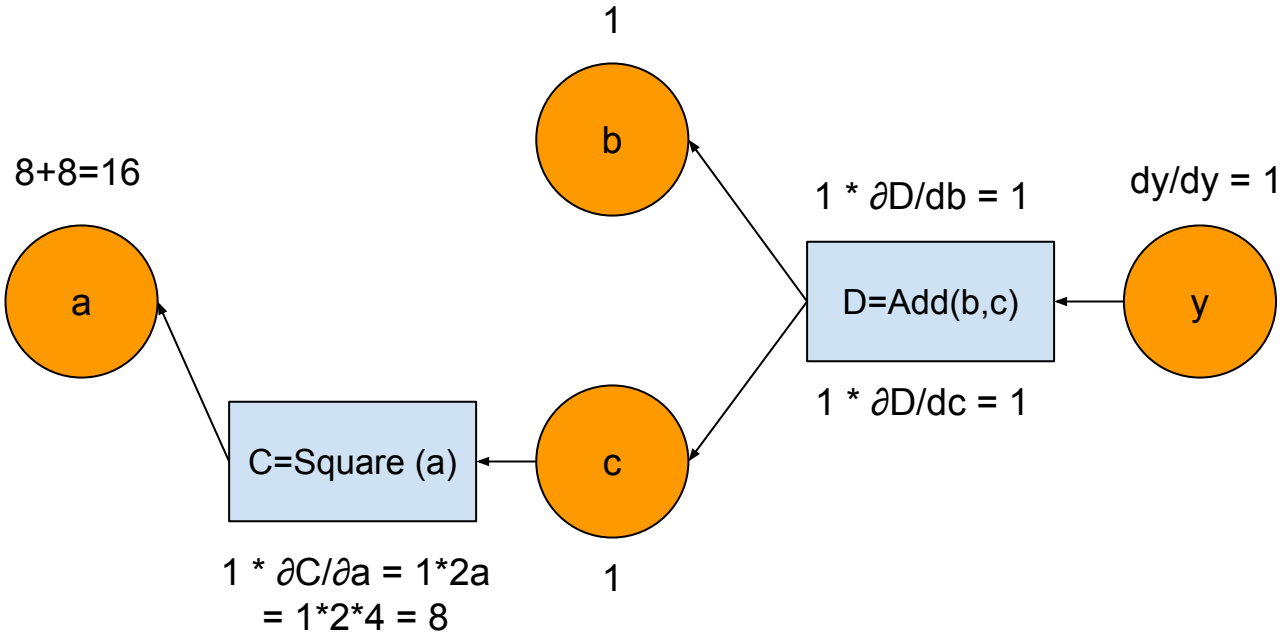

1 * ∂B/∂a = 1*2a
= 1*2*4 = 8
1
64
16 * ∂A/∂x = 16*2x
= 16*2*2 = 64
8+8=16
B=Square (a)
b
1 * ∂D/db = 1
dy/dy = 1
x
A=Square (x)
a
D=Add(b,c)
y
C=Square (a)
c
1 * ∂D/dc = 1
1 * ∂C/∂a = 1*2a
= 1*2*4 = 8
1

STEP14までの課題

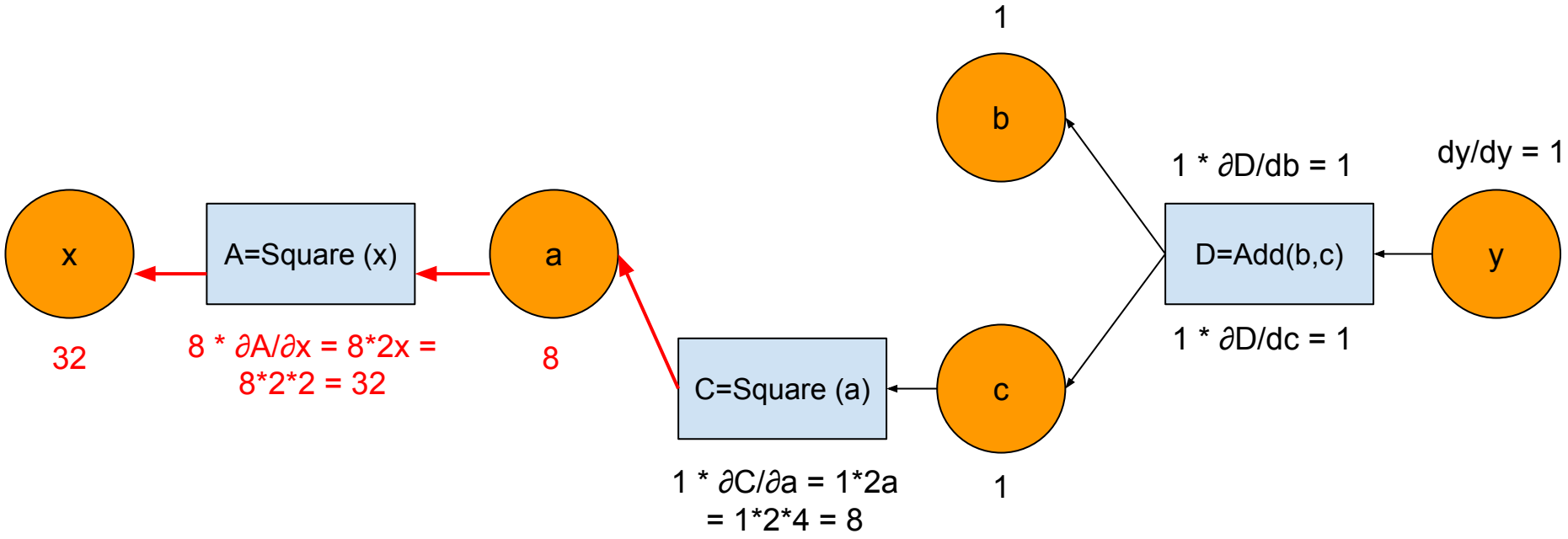
STEP14までの逆伝搬における不具合



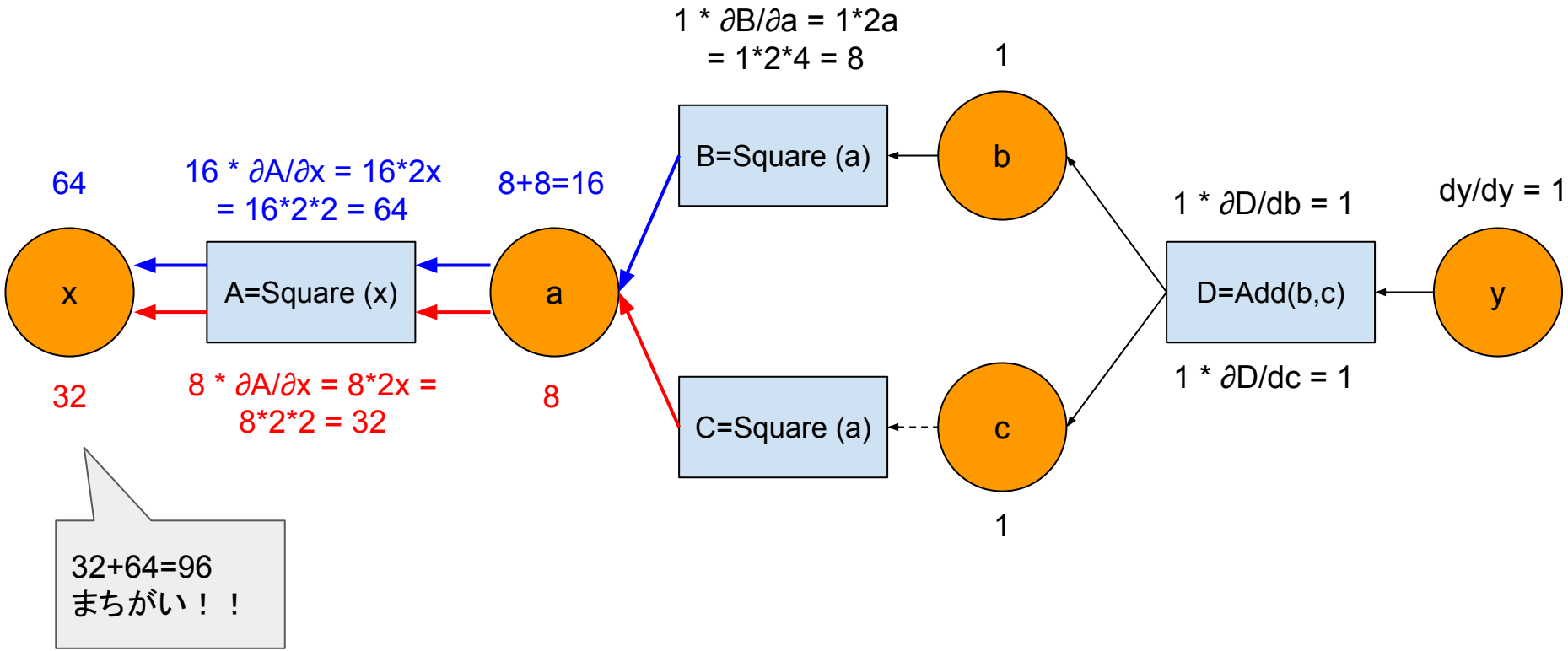
STEP14までの逆伝搬における不具合



STEP14までの逆伝搬における不具合

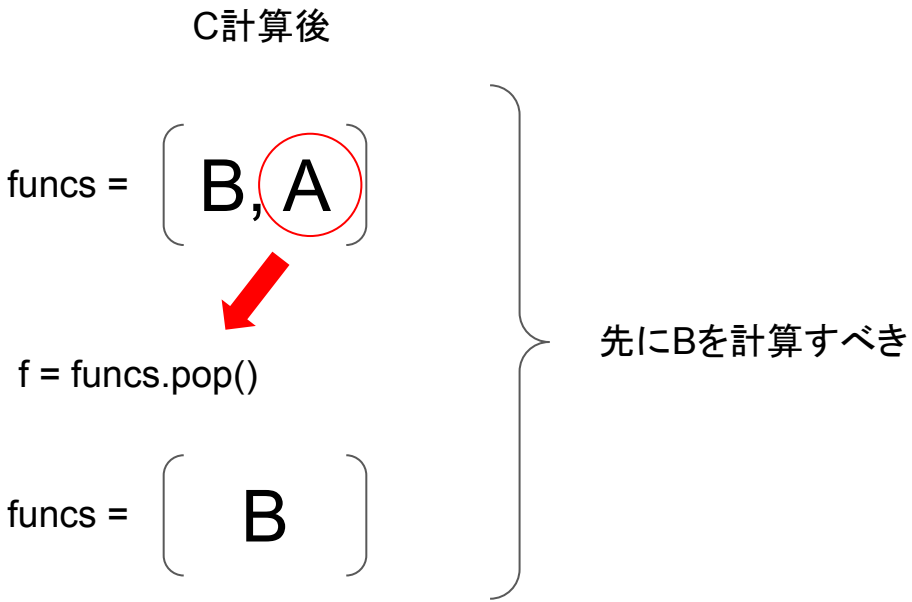


STEP14までの逆伝搬における不具合

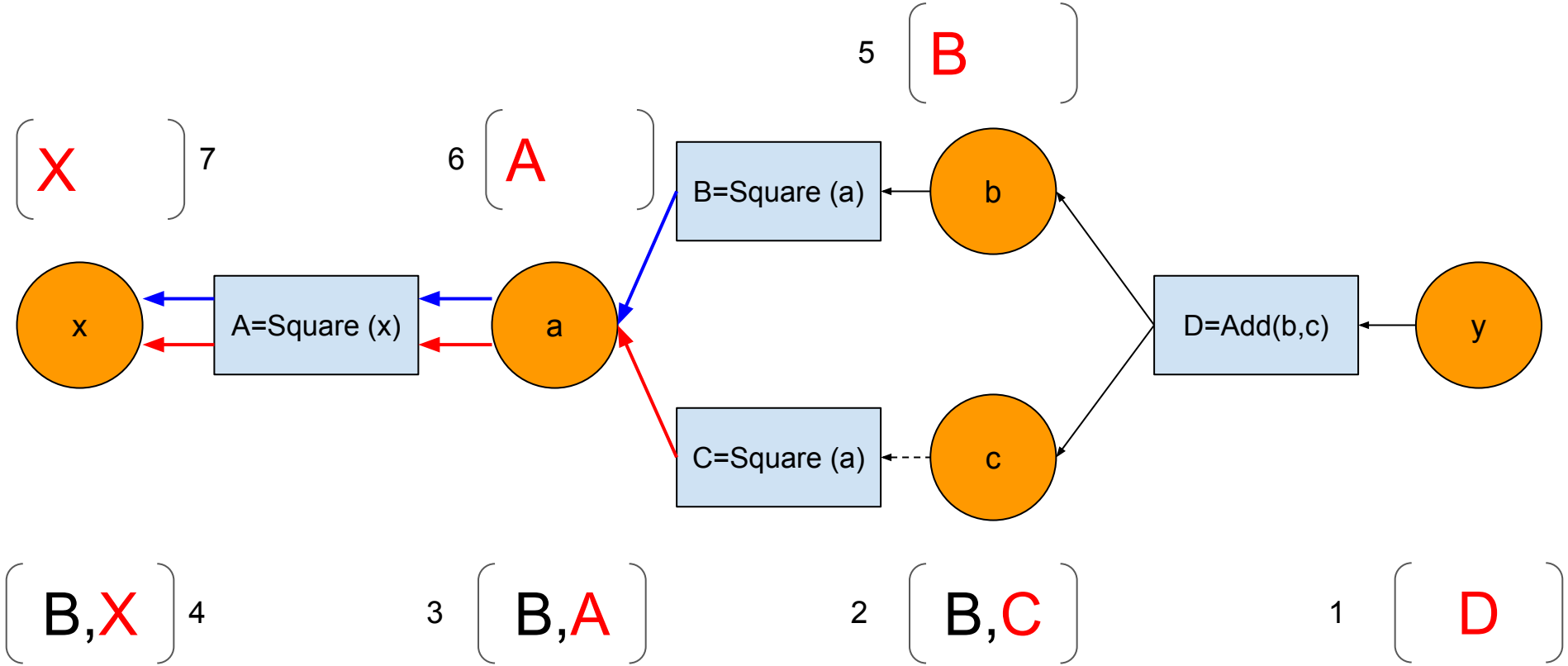


問題の原因は何？

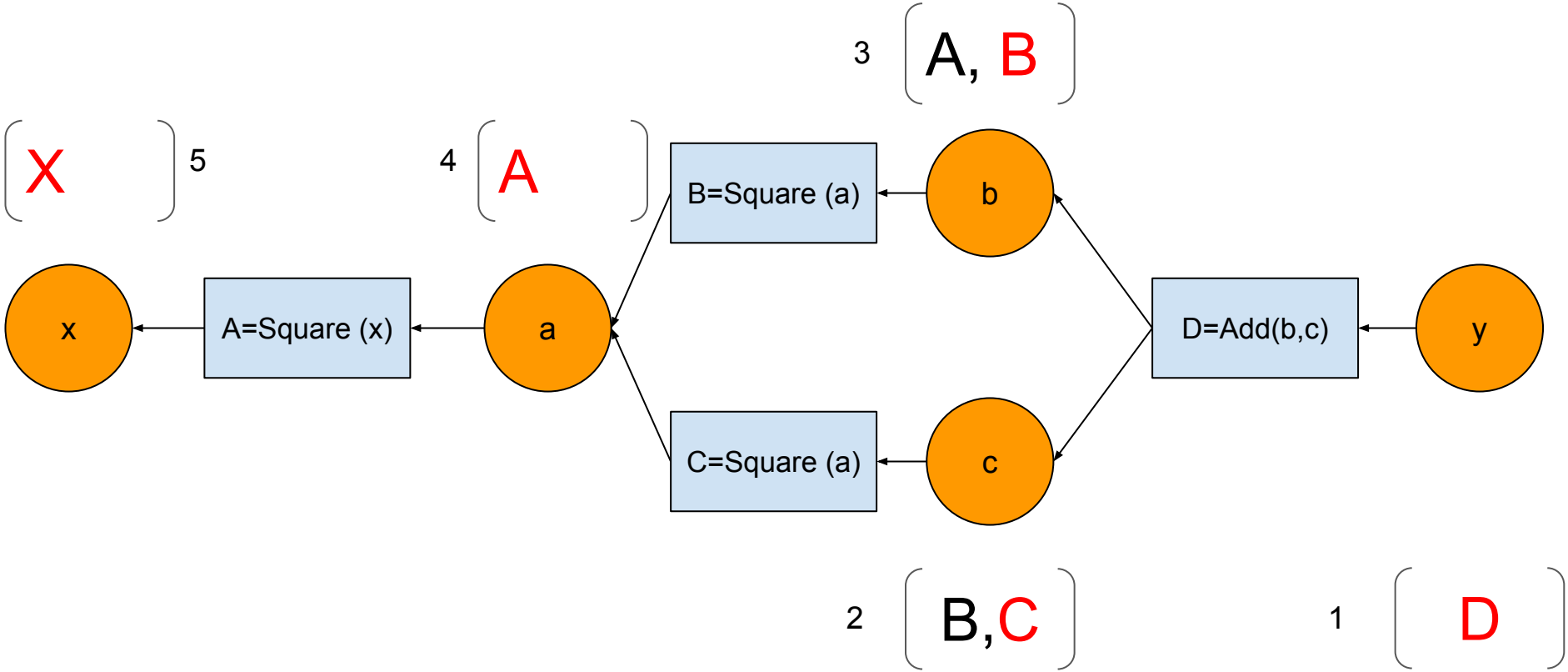
- funcsには次に処理すべき関数の候補が入っているが、funcs.pop()とし、リストの最後尾から単純に処理すべき関数を抜き出している



funcsとpop()の状態: 赤文字をpopが食べる (STE14)



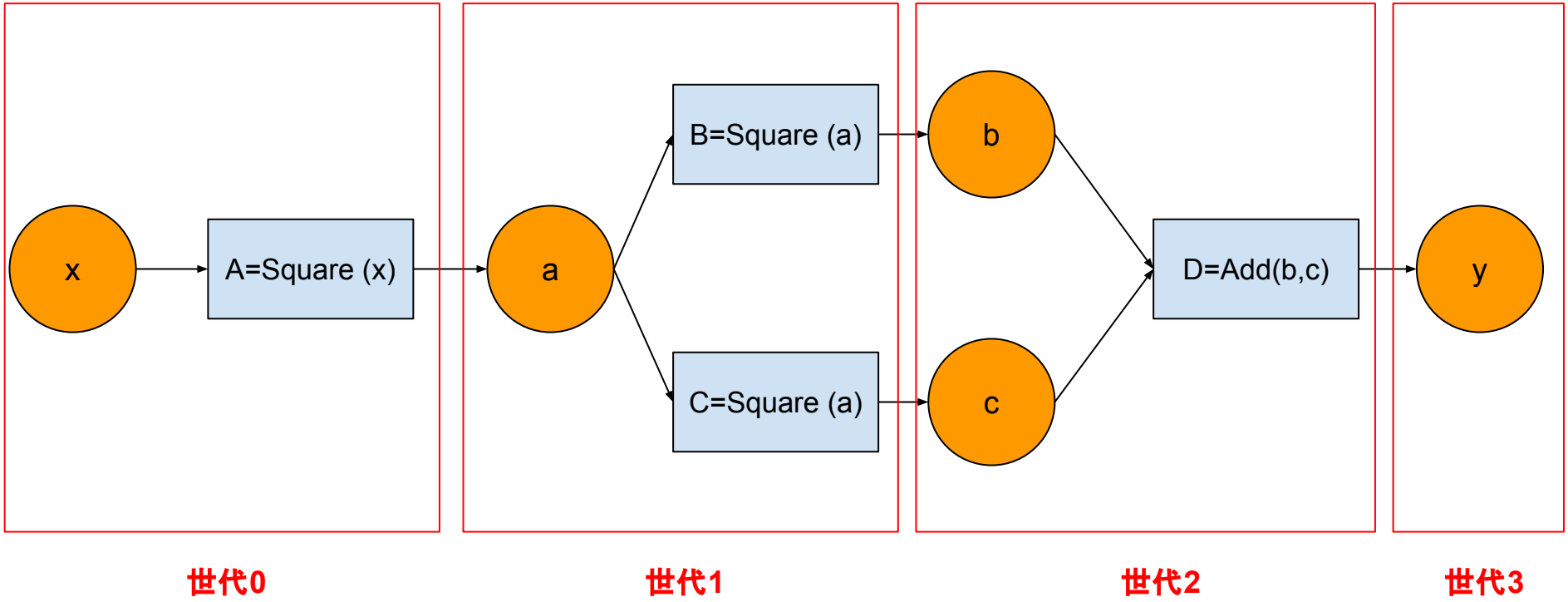
funcsとpop()の状態: 赤文字をpopが食べる (STE16, 改善策)



解決法

- 関数に優先度を与える
- 優先度の高い関数を先に計算するようにする
- 順伝搬時、関数が関数を生み出す過程を利用し、世代(generation)という優先度を定義する

世代 (Generation) の導入



世代の追加: Variableクラス

STEP14

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

        self.data = data
        self.grad = None
        self.creator = None


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

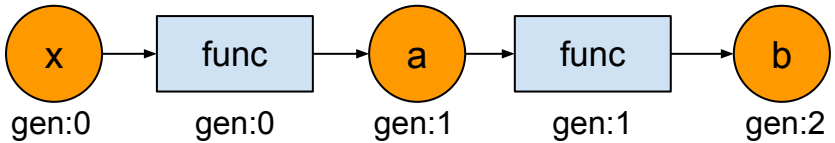
STEP15, 16

```
class Variable:
    def __init__(self, data):
        if data is not None:
            if not isinstance(data, n
                raise TypeError('{} i                mat(

        self.data = data
        self.grad = None
        self.creator = None
>       self.generation = 0

    def set_creator(self, func):
        self.creator = func
>       self.generation = func.generation + 1
```

順伝搬で生成される。つまり初期値は0世代

Class Functionでコールされるメソッド

関数のVariable型のoutputは、その世代が一つ増す



世代の追加: Functionクラス

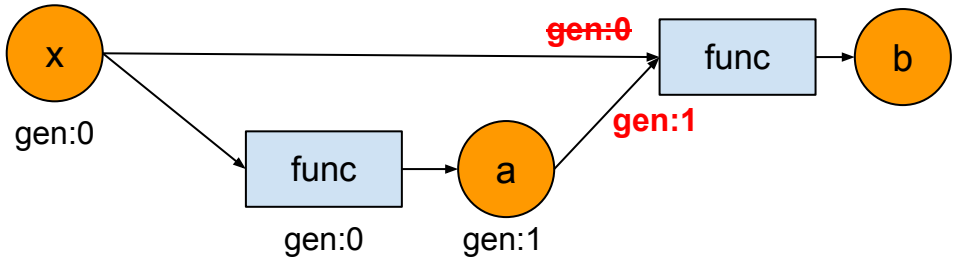
STEP14

```
class Function:
    def __call__(self, *inputs):
        xs = [x.data for x in inputs]
        ys = self.forward(*xs)
        if not isinstance(ys, tuple):
            ys = (ys,)
        outputs = [Variable(as_array(y)) for y in ys]


        for output in outputs:
            output.set_creator(self)
```

STEP15, 16

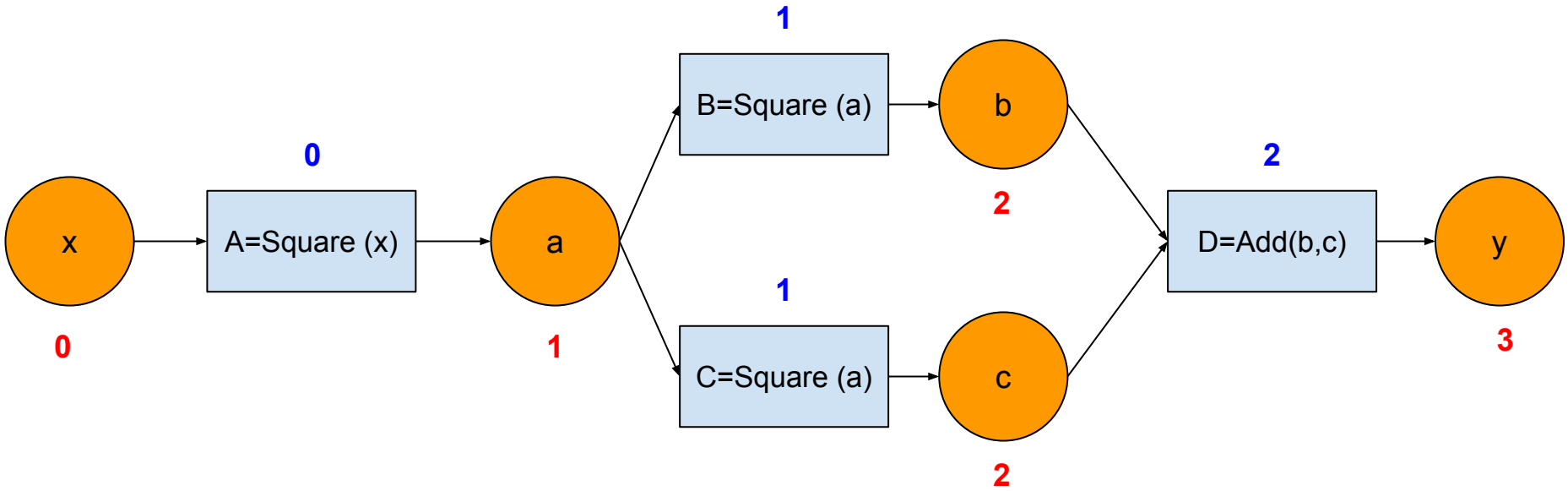
順伝搬で生成される。つまり初期値は0世代

```
class Function:
    def __call__(self, *inputs):
        xs = [x.data for x in inputs]
        ys = self.forward(*xs)
        if not isinstance(ys, tuple):
            ys = (ys,)
        outputs = [Variable(as_array(y)) for y in ys]

>       self.generation = max([x.generation for x in inputs])
        for output in outputs:
            output.set_creator(self)
```



gen = max([0, 1]) = 1

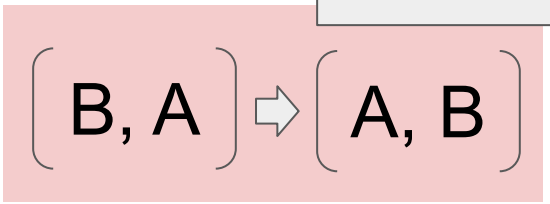
世代 (Generation) の例



逆伝搬に関連した変更: Variable クラス

STEP14

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

    funcs = [self.creator]

    while funcs:
        f = funcs.pop()
        gys = [output.grad for output in
        gxs = f.backward(*gys)
        if not isinstance(gxs, tuple):
            gxs = (gxs,)

        for x, gx in zip(f.inputs, gxs):
            if x.grad is None:
                x.grad = gx
            else:
                x.grad = x.grad + gx

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



世代順に並ぶ

funcsの一番うしろを取り出し、その要素を削除

STEP15, 16

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

    funcs = []
    seen_set = set()

    def add_func(f):
        if f not in seen_set:
            funcs.append(f)
            seen_set.add(f)
            funcs.sort(key=lambda x: x.generation)

    add_func(self.creator)

    while funcs:
        f = funcs.pop()
        gys = [output.grad for output in f.outputs]
        gxs = f.backward(*gys)
        if not isinstance(gxs, tuple):
            gxs = (gxs,)

        for x, gx in zip(f.inputs, gxs):
            if x.grad is None:
                x.grad = gx
            else:
                x.grad = x.grad + gx

            if x.creator is not None:
                add_func(x.creator)
```

出力層の処理
この例だとAdd関数

入力層と出力層を除く処理

補足

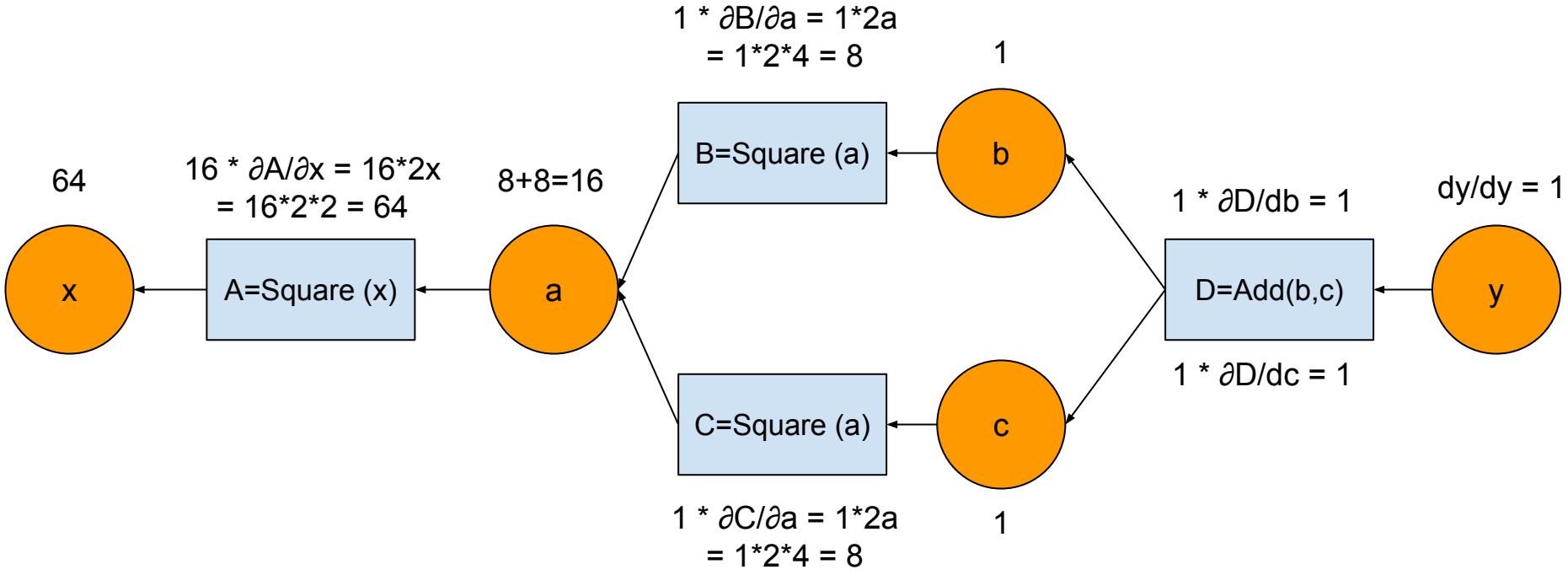
```
funcs.sort(key=lambda x: x.generation)
```

=

```
def compare_rank(x):
    rank = x.generation
    return rank

funcs.sort(key = compare_rank)
```

	各種Variable型の微分値					中間リスト変数		関数へのアクセス			
	Variables					Generation制御					
	y	c	b	a	x	funcs	seen_set	D	C	B	A
	y.grad	b.grad	c.grad	a.grad	x.grad			Add(b,c)	Squre(a)	Squre(a)	Squre(x)
	none	none	none	none	none						
self.grad = np.ones_like(self.data)	1										
add_func						[D]	[D]				
while						[D]	[D]				
funcs.pop()						[]	[D]				
gxs = f.backward(*gys)											
for ix, gx in zip(f.inputs, gxs):											
0 x.grad = gx			1								
add_func(x.creator)						[B]	[D, B]				
1 x.grad = gx		1									
add_func(x.creator)						[B, C]	[D, B, C]				
funcs.pop()						[B]	[D, B, C]				
gxs = f.backward(*gys)											
for ix, gx in zip(f.inputs, gxs):											
0 x.grad = gx		8									
add_func(x.creator)						[A, B]	[A, D, B, C]				
funcs.pop()						[A]	[A, D, B, C]				
gxs = f.backward(*gys)											
for ix, gx in zip(f.inputs, gxs):											
0 x.grad = x.grad + gx			8+8=16								
add_func(x.creator)						[A]	[A, D, B, C]				
funcs.pop()						[]	[A, D, B, C]				
gxs = f.backward(*gys)											
for ix, gx in zip(f.inputs, gxs):											
0 x.grad = gx			64								

正常動作確認


1 * ∂B/∂a = 1*2a
= 1*2*4 = 8
1
64
16 * ∂A/∂x = 16*2x
= 16*2*2 = 64
8+8=16
B=Square (a)
b
1 * ∂D/db = 1
dy/dy = 1
x
A=Square (x)
a
D=Add(b,c)
y
1 * ∂D/dc = 1
C=Square (a)
c
1 * ∂C/∂a = 1*2a
= 1*2*4 = 8
1

STEP17

メモリ管理と循環参照

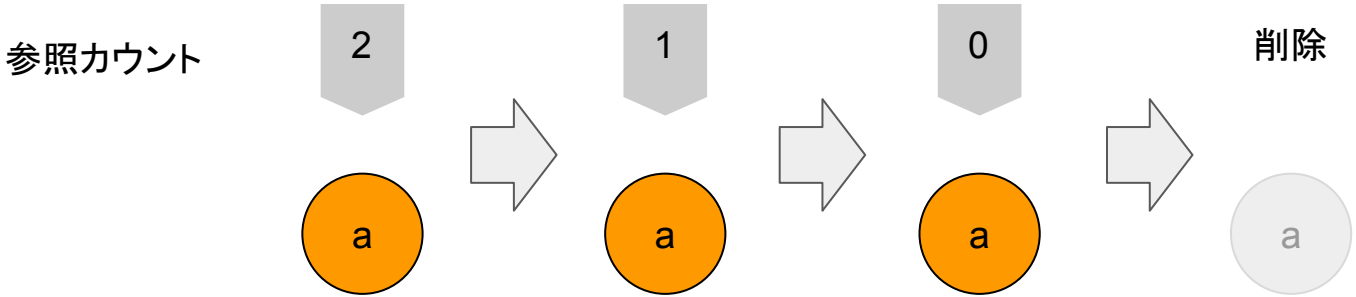
Pythonのメモリ管理

CPythonのメモリ管理

1. 参照カウント
2. GC (Garbage Collection)

参照カウント方式のメモリ管理

- 特徴
  - 単純かつ高速
  - カウントがゼロになるとメモリからデータを削除
- 参照カウントが増えるとき
  - 代入演算子を使ったとき : b = a (bはaを参照している)
  - 引数渡しをしたとき : f(a)
  - オブジェクトをコンテナ型オブジェクトに追加したとき : test_list.append(a)



getrefcountによる確認

参照カウントを表示できる

- getrefcount自身も+1とされるので注意
- Noneとすると、何か大きな値になる (0になるわけでない)
    - 内部的には変数自身の参照カウントが一つ減ってはいるように見える

```python
import sys
class obj:
    pass

def f(x):
    print("STEP3-1:", sys.getrefcount(x) - 1)


a = obj();          print("STEP1:",   sys.getrefcount(a) - 1)
b = a;              print("STEP2:",   sys.getrefcount(a) - 1)
f(a);               print("STEP3-2:", sys.getrefcount(a) - 1)
test_list = [a];    print("STEP4:",   sys.getrefcount(a) - 1)

# Noneにしても、0にならないことに注意
a = None;           print("STEP5:",   sys.getrefcount(a) - 1)
```

```
STEP1: 1
STEP2: 2
STEP3-1: 4
STEP3-2: 2
STEP4: 3
STEP5: 33706
```

変数aの自分自身の参照

代入演算子

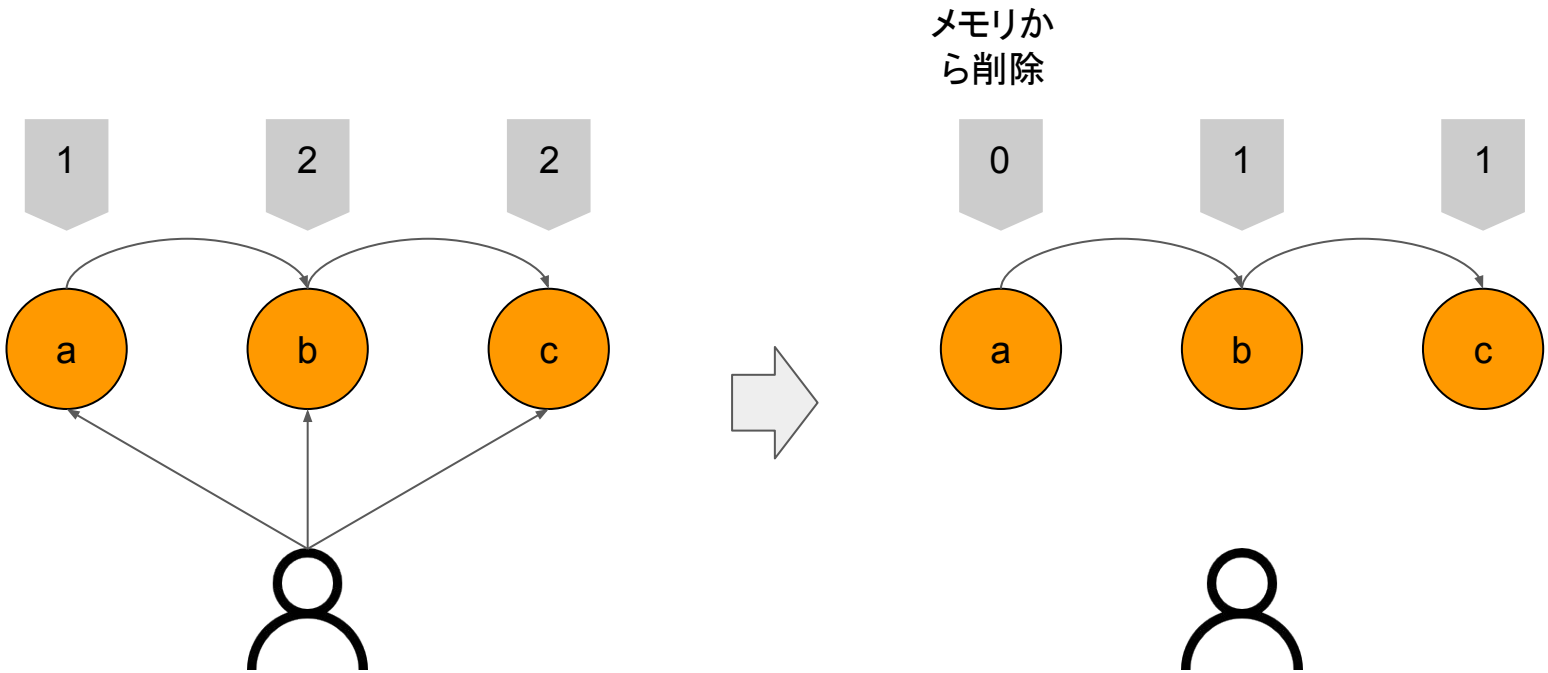
関数fの引数として

aをコンテナ型オブジェクトへ追加

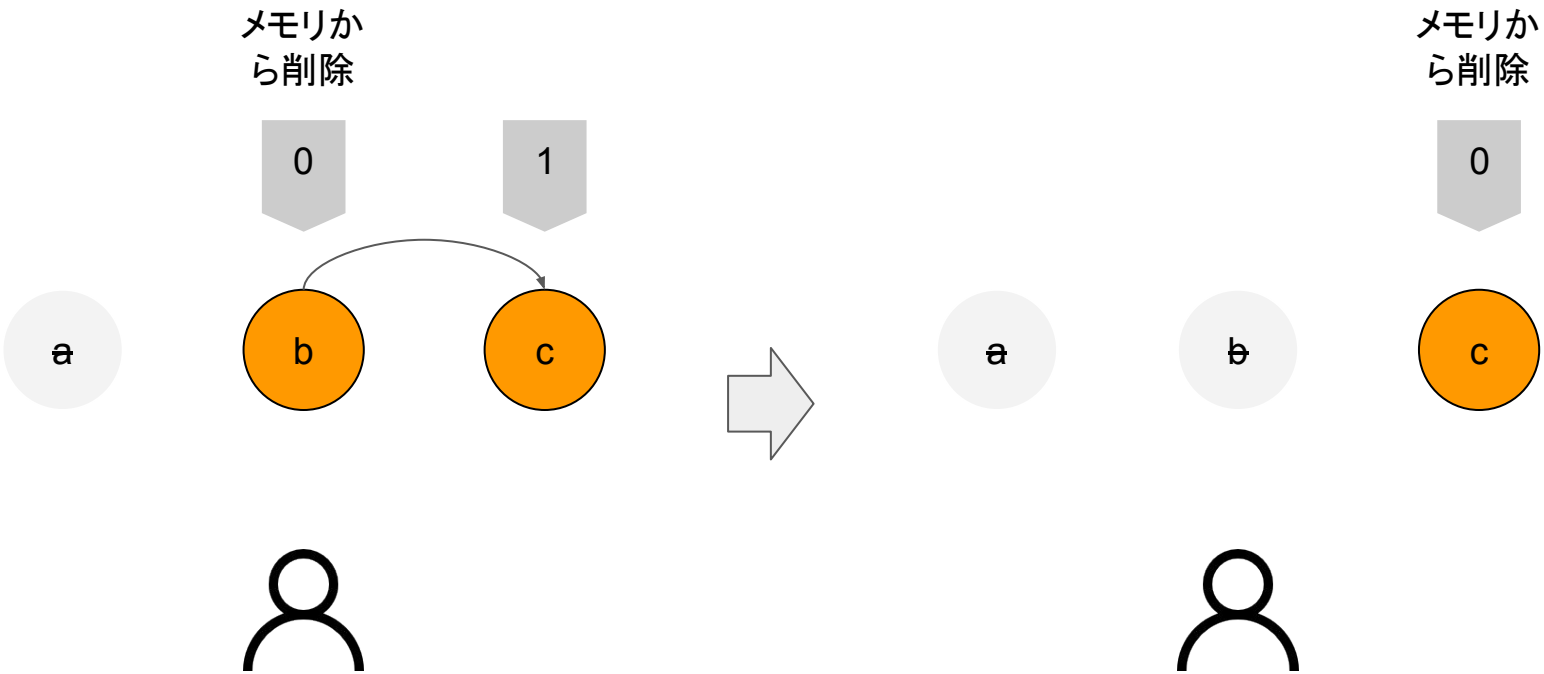
getrefcountを使うとNoneをしても、表示は0にならないことに注意

1. 変数a自身の参照
2. bがaを参照
3. aがf関数の引数
4. Pythonの関数スタックが参照

オブジェクト関係図 (参照の参照の振る舞い)



オブジェクト関係図



```python
import sys
class obj:
    pass

def f(x):
    print("STEP3-1:", sys.getrefcount(x) - 1)

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

print(f"STEP0 a: {sys.getrefcount(a) - 1}, b: {sys.getrefcount(b) - 1}, c: {sys.getrefcount(c) - 1}")

a.b = b;          print(f"STEP1 a: {sys.getrefcount(a) - 1}, b: {sys.getrefcount(b) - 1}, c: {sys.getrefcount(c) - 1}")
b.c = c;          print(f"STEP2 a: {sys.getrefcount(a) - 1}, b: {sys.getrefcount(b) - 1}, c: {sys.getrefcount(c) - 1}")
a = b = c = None; print(f"STEP3 a: {sys.getrefcount(a) - 1}, b: {sys.getrefcount(b) - 1}, c: {sys.getrefcount(c) - 1}")

print(sys.getrefcount(None))
```

```
STEP0 a: 1, b: 1, c: 1
STEP1 a: 1, b: 2, c: 1
STEP2 a: 1, b: 2, c: 2
STEP3 a: 33727, b: 33727, c: 33727
33730
```

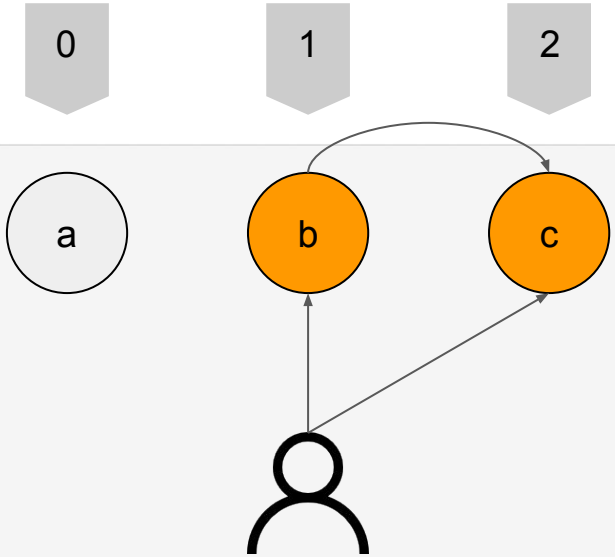
a, b, c 自分自身の参照

aはbを参照

bはcを参照

Noneにするとドミノ的に？
getrefcountだと上手く見られませんでした

参考: aだけをNoneにしたとき

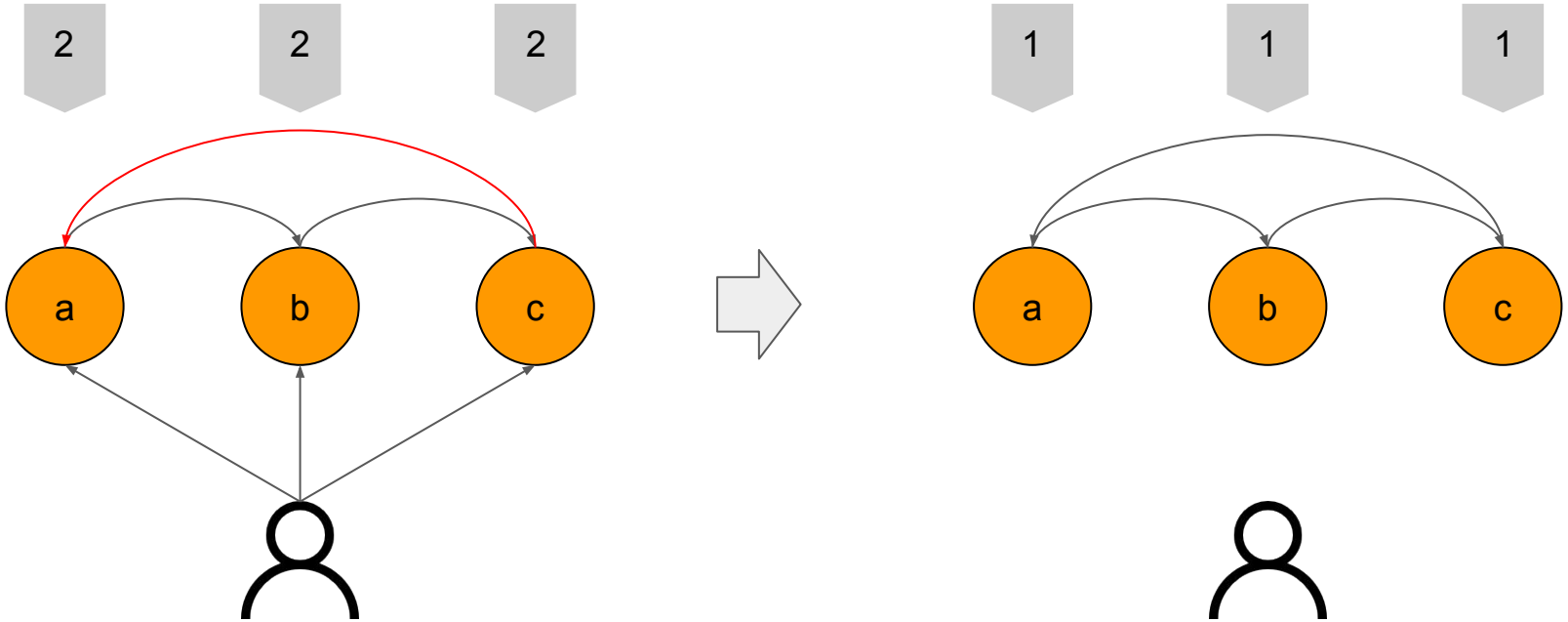



```python
import sys
class obj:
    pass

def f(x):
    print("STEP3-1:", sys.getrefcount(x) - 1)

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


a.b = b
b.c = c
a = None
print(f"a: {sys.getrefcount(a) - 1}, b: {sys.getrefcount(b) - 1}, c: {sys.getrefcount(c) - 1}")

print(sys.getrefcount(None))
```

試しにaのみ

```
a: 33716, b: 1, c: 2
33715
```

循環参照の場合のオブジェクトの関係図



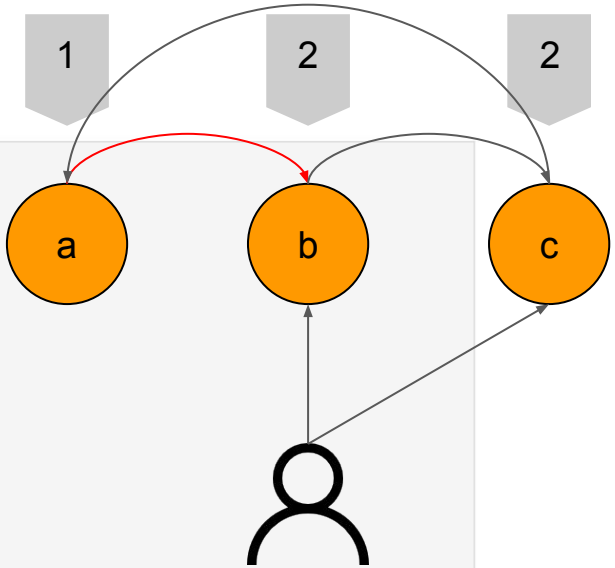
循環参照のテスト

```
import sys
class obj:
    pass

def f(x):
    print("STEP3-1:", sys.getrefcount(x) - 1)

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

print(f"STEP0 a: {sys.getrefcount(a) - 1}, b: {sys.getrefcount(b) - 1}, c: {sys.getrefcount(c) - 1}")

a.b = b;          print(f"STEP1 a: {sys.getrefcount(a) - 1}, b: {sys.getrefcount(b) - 1}, c: {sys.getrefcount(c) - 1}")
b.c = c;          print(f"STEP2 a: {sys.getrefcount(a) - 1}, b: {sys.getrefcount(b) - 1}, c: {sys.getrefcount(c) - 1}")
c.a = a;          print(f"STEP3 a: {sys.getrefcount(a) - 1}, b: {sys.getrefcount(b) - 1}, c: {sys.getrefcount(c) - 1}")
a = b = c = None; print(f"STEP4 a: {sys.getrefcount(a) - 1}, b: {sys.getrefcount(b) - 1}, c: {sys.getrefcount(c) - 1}")

print(sys.getrefcount(None))
```

```
STEP0 a: 1, b: 1, c: 1
STEP1 a: 1, b: 2, c: 1
STEP2 a: 1, b: 2, c: 2
STEP3 a: 2, b: 2, c: 2
STEP4 a: 33744, b: 33744, c: 33744
33747
```

Noneにすると、循環参照の様子が観測できない

参考: aだけをNoneにしたとき



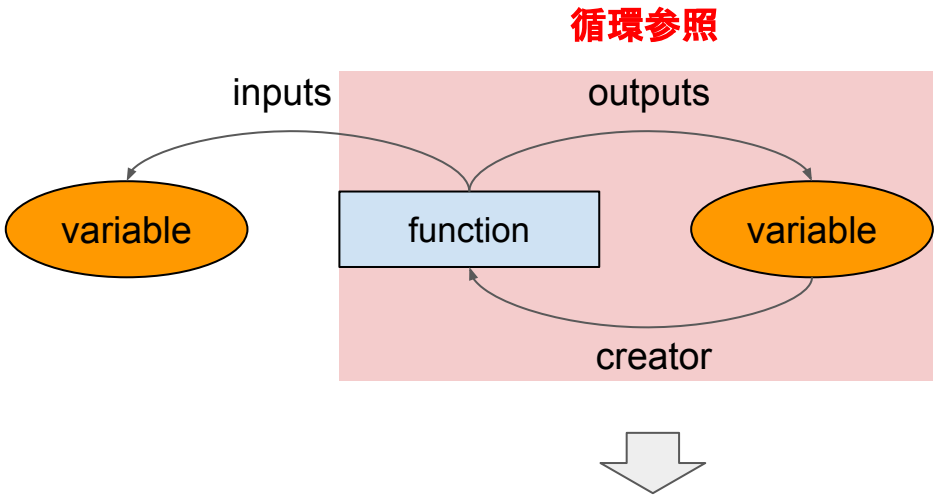

```python
import sys
class obj:
    pass

def f(x):
    print("STEP3-1:", sys.getrefcount(x) - 1)

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


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

print(f"a: {sys.getrefcount(a) - 1}, b: {sys.getrefcount(b) - 1}, c: {sys.getrefcount(c) - 1}")

a = None

print(f"a: {sys.getrefcount(a) - 1}, b: {sys.getrefcount(b) - 1}, c: {sys.getrefcount(c) - 1}")

print(sys.getrefcount(None))
```

試しにaのみ

```
a: 2, b: 2, c: 2
a: 33713, b: 2, c: 2
33716
```

bは2のまま。つまりaの参照は残っている

VariableとFunctionの循環参照



weakrefモジュールで解決できる

weakrefモジュール

weekrefなし

```
import sys
import weakref
import numpy as np
a = np.array([1, 2, 3])
b = a
print("b:", b)
#print("b():", b())
a = None
print("b(after a = none):", b)
```

```
b: [1 2 3]
b(after a = none): [1 2 3]
```

値は存在

weekrefあり

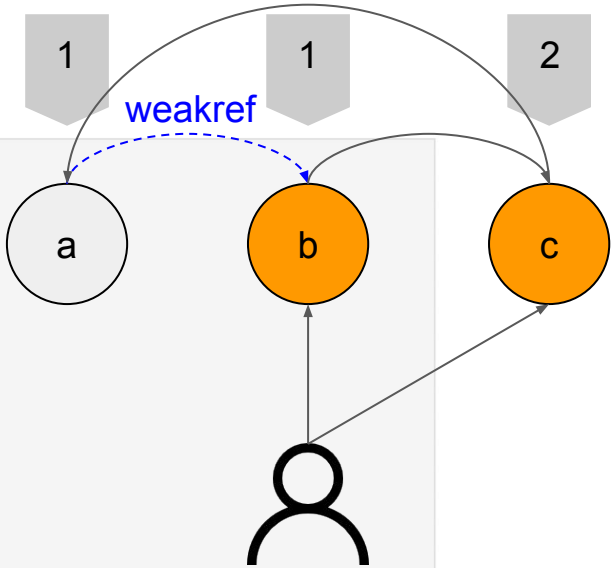
```
import sys
import weakref
import numpy as np
a = np.array([1, 2, 3])
b = weakref.ref(a)
print("b:", b)
print("b():", b())
a = None
print("b(after a = none):", b)
print("b()(after a = none):", b())
```

```
b: <weakref at 0x104c812c0; to 'numpy.ndarray' at 0x103ef6b10>
b(): [1 2 3]
b(after a = none): <weakref at 0x104c812c0; dead>
b()(after a = none): None
```

aにつられてdead

アドレス渡しに近い？
(cでいうポインタ渡し)

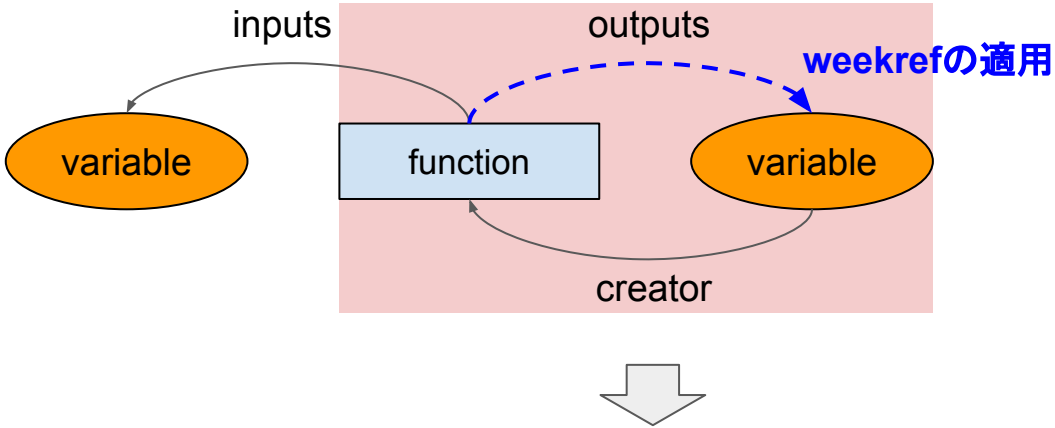
実験2




```
import sys
import weakref
class obj:
    pass

def f(x):
    print("STEP3-1:", sys.getrefcount(x) - 1)

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

#a.b = b
a.b = weakref.ref(b)
b.c = c
c.a = a

print(f"a: {sys.getrefcount(a) - 1}, b: {sys.getrefcount(b) - 1}, c: {sys.getrefcount(c) - 1}")

a = None

print(f"a: {sys.getrefcount(a) - 1}, b: {sys.getrefcount(b) - 1}, c: {sys.getrefcount(c) - 1}")

print(sys.getrefcount(None))
```

```
a: 2, b: 1, c: 2
a: 36918, b: 1, c: 2
36921
```

コードの変更点



weakrefモジュールで解決できる

weakrefの導入: Functionクラス

STEP16

```
import numpy as np

class Function:
    def __call__(self, *inputs):
        xs = [x.data for x in inputs]
        ys = self.forward(*xs)
        if not isinstance(ys, tuple):
            ys = (ys,)
        outputs = [Variable(as_array(y)) for y in ys]

        self.generation = max([x.generation for x in inputs])
        for output in outputs:
            output.set_creator(self)
        self.inputs = inputs
        self.outputs = outputs
        return outputs if len(outputs) > 1 else outputs[0]

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

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

STEP17

```
> import weakref
  import numpy as np

  class Function:
      def __call__(self, *inputs):
          xs = [x.data for x in inputs]
          ys = self.forward(*xs)
          if not isinstance(ys, tuple):
              ys = (ys,)
          outputs = [Variable(as_array(y)) for y in ys]

          self.generation = max([x.generation for x in inputs])
          for output in outputs:
              output.set_creator(self)
          self.inputs = inputs
          self.outputs = [weakref.ref(output) for output in out
          return outputs if len(outputs) > 1 else outputs[0]

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

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

weakrefの導入:Variableクラス

STEP16

```
while funcs:
    f = funcs.pop()
    gys = [output.grad for output in f.outputs]
    gxs = f.backward(*gys)
    if not isinstance(gxs, tuple):
        gxs = (gxs,)

    for x, gx in zip(f.inputs, gxs):
        if x.grad is None:
            x.grad = gx
        else:
            x.grad = x.grad + gx

        if x.creator is not None:
            add_func(x.creator)
```

STEP17

```
while funcs:
    f = funcs.pop()
    gys = [output().grad for output in f.outputs]  #
    gxs = f.backward(*gys)
    if not isinstance(gxs, tuple):
        gxs = (gxs,)

    for x, gx in zip(f.inputs, gxs):
        if x.grad is None:
            x.grad = gx
        else:
            x.grad = x.grad + gx

        if x.creator is not None:
            add_func(x.creator)
```

どのくらいメモリが削減できるか？

```
~/Documents/kinocode/07_study_group/chanet-san
miniforge3-4.10.1-5 ❯ mprof run step17_before.py
mprof: Sampling memory every 0.1s
running new process
running as a Python program...

~/Documents/kinocode/07_study_group/chanet-san 18s
miniforge3-4.10.1-5 ❯ mprof peak
Using last profile data.
mprofile_20220729193654.dat    647.406 MiB

~/Documents/kinocode/07_study_group/chanet-san
miniforge3-4.10.1-5 ❯ mprof run step17.py
mprof: Sampling memory every 0.1s
running new process
running as a Python program...

~/Documents/kinocode/07_study_group/chanet-san 17s
miniforge3-4.10.1-5 ❯ mprof peak
Using last profile data.
mprofile_20220729193719.dat    78.156 MiB
```

導入前

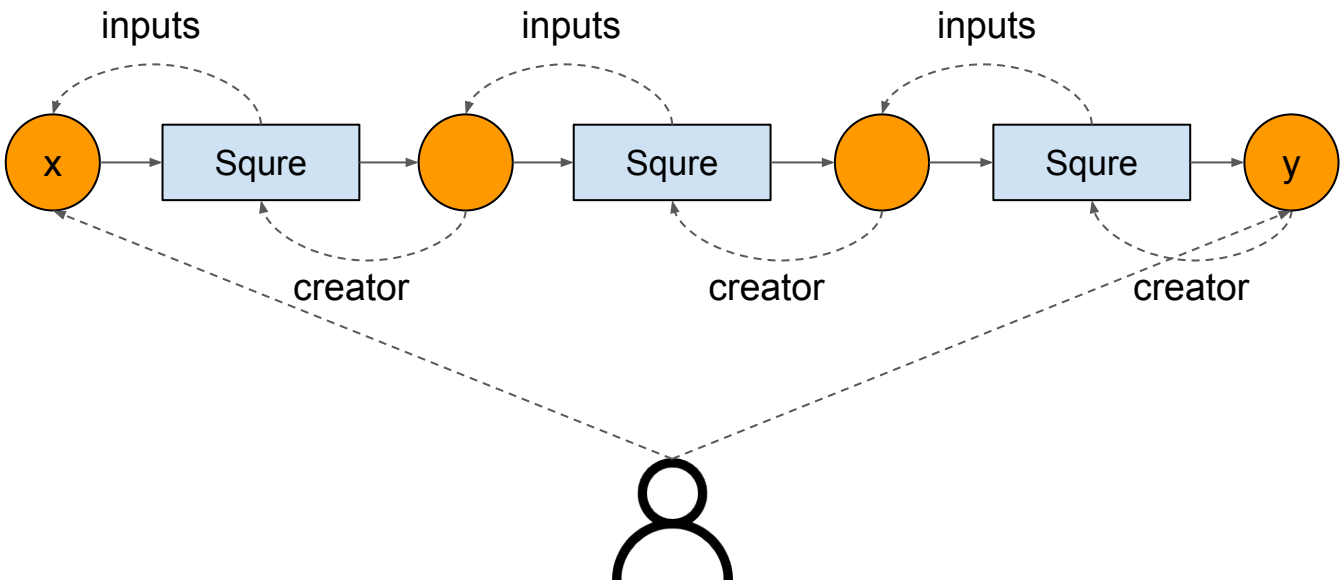
導入後

計測しやすいように値を大きくしてテスト

```python
for i in range(10000):
    x = Variable(np.random.randn(100000))  # big data
    y = square(square(square(x)))
```

[Note]
mprofは以下でインストールできます
$ pip install memory-profiler

参照カウント関係 (weakrefの関係は無いものとみなせる)



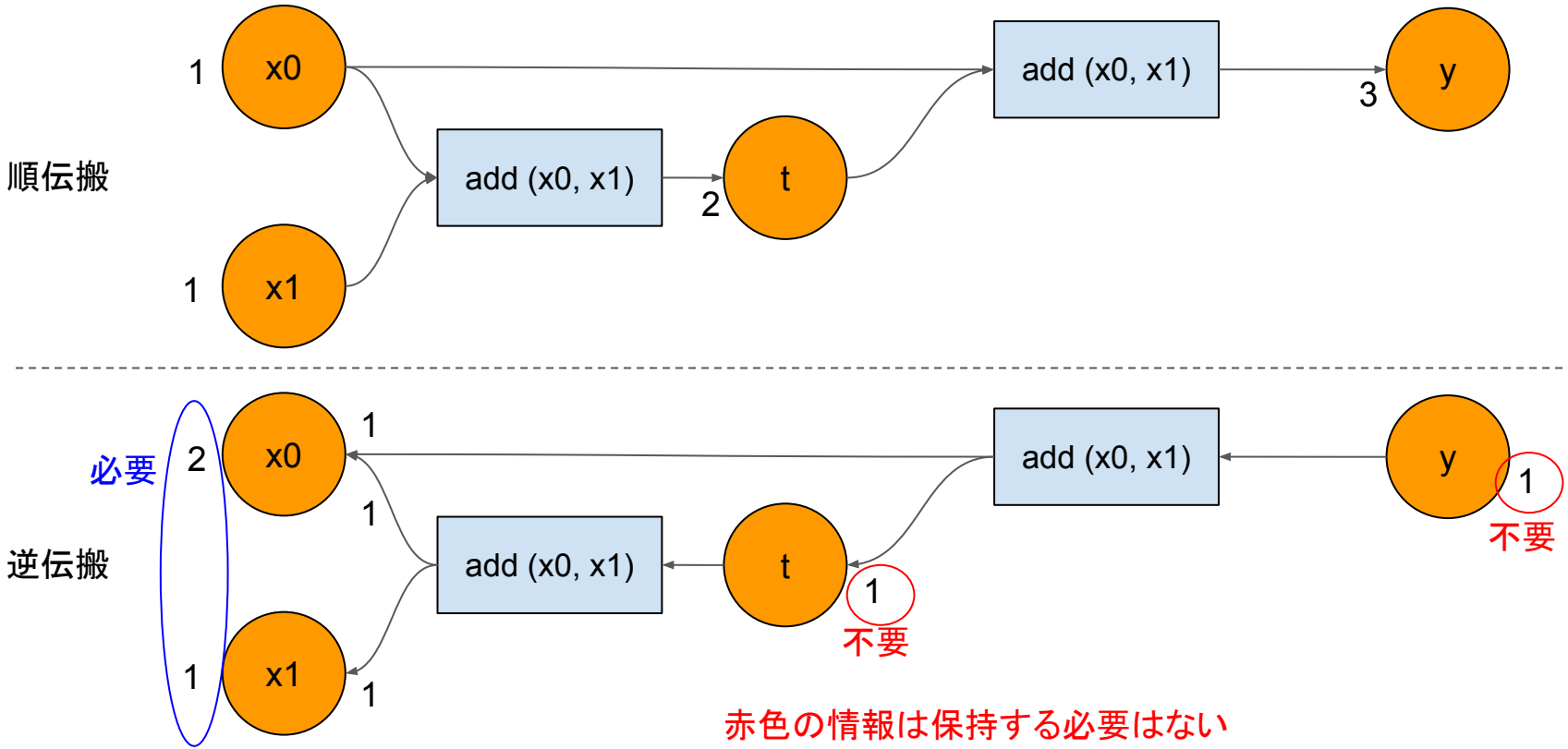
STEP18

メモリ使用量をへらすモード

メモリ使用量の改善策

- 逆伝搬で消費するメモリ使用量を削減する(不要な微分は削除できる)
- 逆伝搬が必要ない場合のモードを用意する

不要な微分は保持しない



不要な微分は保持しない: Variableクラスの改善

STEP16

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

            (省略)

    while funcs:
        f = funcs.pop()
        gys = [output().grad for output in f.outputs]  #
        gxs = f.backward(*gys)
        if not isinstance(gxs, tuple):
            gxs = (gxs,)

        for x, gx in zip(f.inputs, gxs):
            if x.grad is None:
                x.grad = gx
            else:
                x.grad = x.grad + gx
                        outputs
            if x.creator is not None:
                add_func(x.creator)
```

STEP17

```
def backward(self, retain_grad=False):
    if self.grad is None:
        self.grad = np.ones_like(self.dat

            (省略)

    while funcs:
        f = funcs.pop()
        gys = [output().grad for output in f.outputs]  #
        gxs = f.backward(*gys)
        if not isinstance(gxs, tuple):
            gxs = (gxs,)

        for x, gx in zip(f.inputs, gxs):
            if x.grad is None:
                x.grad = gx
            else:
                x.grad = x.grad + gx

            if x.creator is not None:
                add_func(x.creator)

        if not retain_grad:
            for y in f.outputs:
                y().grad = None  # y is weakref
```

Trueの場合、これまでどおり、すべての微分を保存

y().gradはNoneになり、入力層以外のgradは値を保持しない。メモリから削除
(y に値があれば,True)

動作確認

```
x0 = Variable(np.array(1.0))
x1 = Variable(np.array(1.0))
t = add(x0, x1)
y = add(x0, t)
y.backward()
```

```
# 途中の層
print(y.grad, t.grad)    # None None
# 入力層
print(x0.grad, x1.grad)  # 2.0 1.0
```

```
None None
2.0 1.0
```

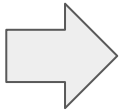
逆伝搬が必要ない場合のモードを用意する

- 順伝搬のみが必要なケースがある
  - 例：推論，モデルの評価
- Functionクラスのself.inputs, self.outputsは逆伝搬の計算のみ必要
  - Variableクラスのself.set_creator, self.generation, self.set_creatorも同様
- 順伝搬のみ必要なケースでは、これらの変数をそもそも保持する必要はない
- そのため逆伝搬の有効モードと逆伝搬の無効モードを作成する

最終的にやりたいこと, with+オレオレ前処理/後処理関数

with no_grad():

処理A
処理B
処理C



前処理=> 逆伝搬をOFF

処理A
処理B
処理C

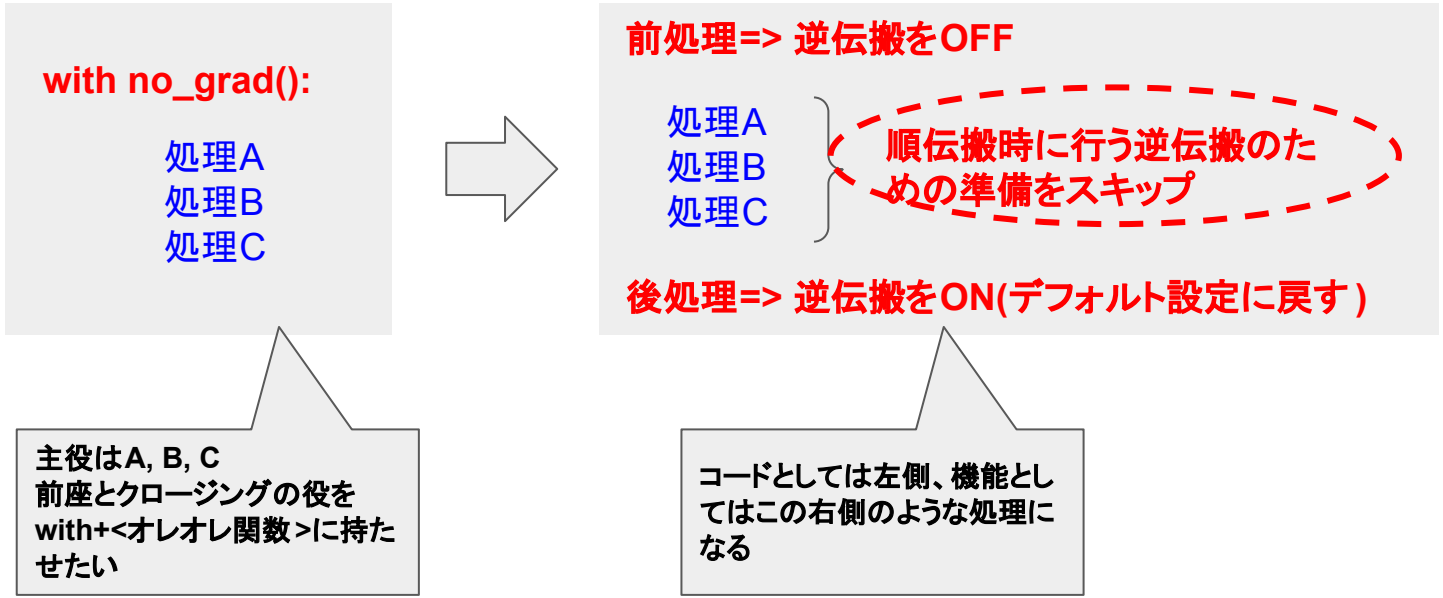
順伝搬時に行う逆伝搬のための準備をスキップ

後処理=> 逆伝搬をON(デフォルト設定に戻す)

主役はA, B, C
前座とクロージングの役を with+<オレオレ関数>に持たせたい

コードとしては左側、機能としてはこの右側のような処理になる

最終的にやりたいこと, with+オレオレ前処理/後処理関数



Configクラスによる切り替え

STEP16

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

STEP17

```
  import weakref
  import numpy as np
> import contextlib
>
>
> class Config:
>     enable_backprop = True
>
>
> @contextlib.contextmanager
> def using_config(name, value):
>     old_value = getattr(Config, name)
>     setattr(Config, name, value)
>     try:
>         yield
>     finally:
>         setattr(Config, name, old_value)
>
>
> def no_grad():
>     return using_config('enable_backprop', False)
```

Trueの場合、逆伝搬可能
これでスイッチング

後で説明

Functionクラス: 逆伝搬に関する値の保持を行わない

STEP16

```
class Function:
    def __call__(self, *inputs):
        xs = [x.data for x in inputs]
        ys = self.forward(*xs)
        if not isinstance(ys, tuple):
            ys = (ys,)
        outputs = [Variable(as_array(y)) for y in ys]

        self.generation = max([x.generation for x in inputs])
        for output in outputs:
            output.set_creator(self)
        self.inputs = inputs
        self.outputs = [weakref.ref(output) for output in out

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

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

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

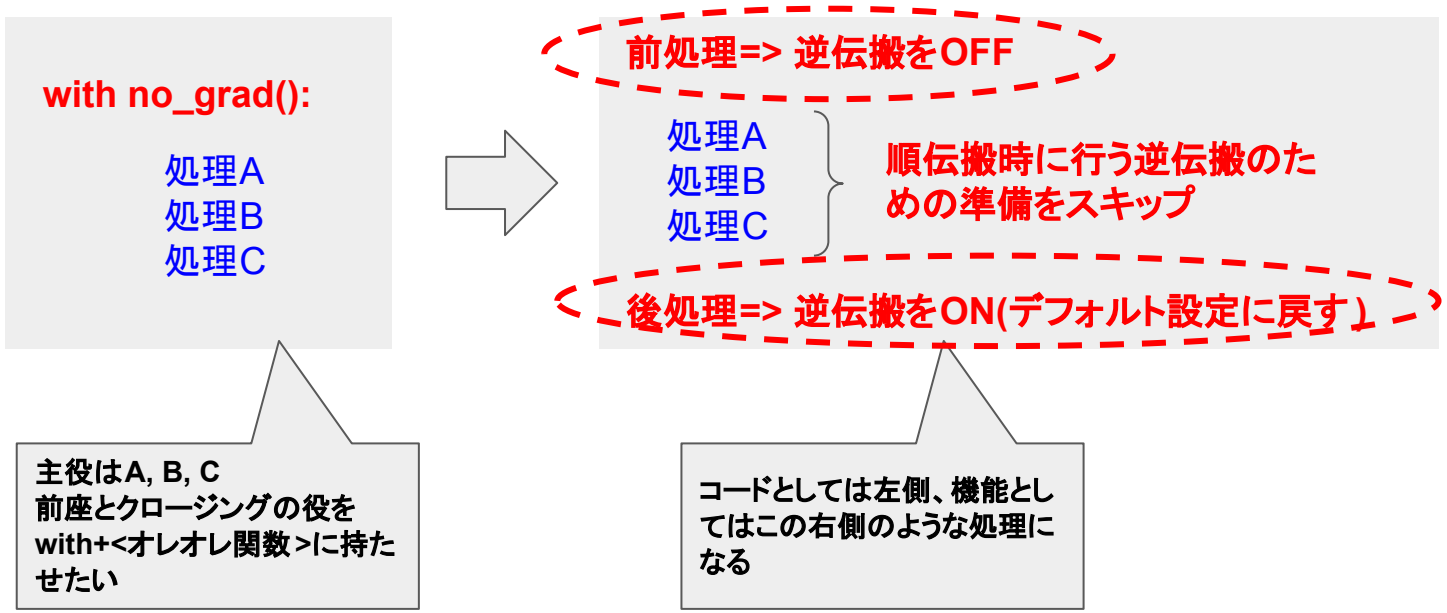
STEP17

```
class Function:
    def __call__(self, *inputs):
        xs = [x.data for x in inputs]
        ys = self.forward(*xs)
        if not isinstance(ys, tuple):
            ys = (ys,)
        outputs = [Variable(as_array(y)) for y in ys]

        if Config.enable_backprop:
            self.generation = max([x.generation for x in inpu
            for output in outputs:
                output.set_creator(self)
            self.inputs = inputs
            self.outputs = [weakref.ref(output) for output in

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

    def forward(self, xs):
        raise NotImplementedEr

    def backward(self, gys):
        raise NotImplementedEr
```

このif 文以外は、左と一緒

この領域全部、逆伝搬を計算しなければ不要

最終的にやりたいこと, with+オレオレ前処理/後処理関数



with文による切り替え(withのご利益)

自分で `f.close` をちゃんと書く必要あり。めんどくさい

```
f = open('sample.txt', 'w')
f.write('hello world')
f.close
```

```
<function TextIOWrapper.close()>
```

`with` 文を使うとこの手間を省ける

```
with open('sample.txt', 'w') as f:
    f.write('hello world')
```

closeの書き忘れを防げる以外に、例外発生時も、リソースを確実に解放できるメリットがある

この `with` 文の仕組みを使って「逆伝搬無効モード」へと切り替えることを考える

with+openの動作を with+自作関数でも機能させたい

```
import contextlib
@contextlib.contextmanager
def config_test():
    print('start')     # 前処理
    try:
        yield
    finally:
        print('done') # 後処理

with config_test():
    print('process...')
```

前処理をyieldの前

後処理をyieldの後

一番やりたいこと

```
start
process...
done
```

参考: 右のコードでもOK？ 同じ挙動らしい・・

```python
import contextlib
@contextlib.contextmanager
def config_test():
    print('start')    # 前処理
    try:
        yield
    finally:
        print('done') # 後処理

with config_test():
    print('process...')
```

```
start
process...
done
```

https://qiita.com/QUANON/items/c5868b6c65f8062f5876

```python
class config_test():

    def __enter__(self):
        print('start')    # 前処理

    def __exit__(self, type, value, traceback):
        print('done') # 後処理

with config_test():
    print('process...')
```

```
start
process...
done
```

With 文による切り替え: Configクラス

STEP16

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

STEP17

```
  import weakref
  import numpy as np
> import contextlib
>
>
> class Config:
>     enable_backprop = True
>
>
> @contextlib.contextmanager
> def using_config(name, value):
      old_value = getattr(Config, name)
>     setattr(Config, name, value)
>     try:
>         yield
>     finally:
>         setattr(Config, name, old_value)
>
>
> def no_grad():
>     return using_config('enable_backprop', False)
```

Trueの場合、逆伝搬可能

<前処理>
この例だと以下のようになる
old_value = True
Config.enable_backprop = False

<後処理>
元に戻してやる。つまり
Config.enable_backprop = True

with文による切り替え動作確認

```
with using_config('enable_backprop', False):
    x = Variable(np.array(2.0))
    y = square(x)
```

```
#y.backward()
print(y.generation)
```

確かに逆伝搬に必要な変数は変更がない。初期値のまま

```
0
```

上記のように記述するのが面倒なときは下記でもよい。同じ意味

```
with no_grad():
    x = Variable(np.array(2.0))
    y = square(x)
```