

# if for while try

2019/10/30

木南 貴志

# 0. 事前知識

# 事前知識：インデント

インデント：空白を開けて字下げを行うこと

pythonはインデントによってブロック（かたまり）を認識

下記のようにインデントを揃えないと実行時にインデントエラーが発生

```
x = 3
y = x + 3
print(y)
```

意味のないインデント ⇒ エラー

後述するif文,while文,try文などでは、インデントによってひとつのブロックと認識

下にif文の例を示す（if文の詳細は後述）

```
if x > 0:
    y = x + 3
    print(y)
z = x * y
```

インデントした部分まで(2行目,3行目) がif文の対象範囲

←4行目はインデントされていないので、if文に含まれない

# 事前知識：関係演算子

pythonで比較をする時に使用する関係演算子は以下

演算子	機能
<code>x &gt; y</code>	xはyより大きい
<code>x &lt; y</code>	xはyより小さい（未満）
<code>x &gt;= y</code>	xはy以上
<code>x &lt;= y</code>	xはy以下
<code>x == y</code>	xとyは等しい
<code>x != y</code>	xとyは等しくない
<code>x in y</code>	xという要素がyに存在する
<code>x not in y</code>	xという要素がyに存在しない

# 事前知識： $x \text{ in } y$ のイメージ

**$x \text{ in } y$**

$y =$

20 40 76 11

$x =$

20

$x(=20)$  という要素が  $y$  に存在する

**$x \text{ not in } y$**

$y =$

20 40 76 11

$x =$

80

$x(=80)$  という要素が  $y$  に存在しない

# 事前知識：論理演算子

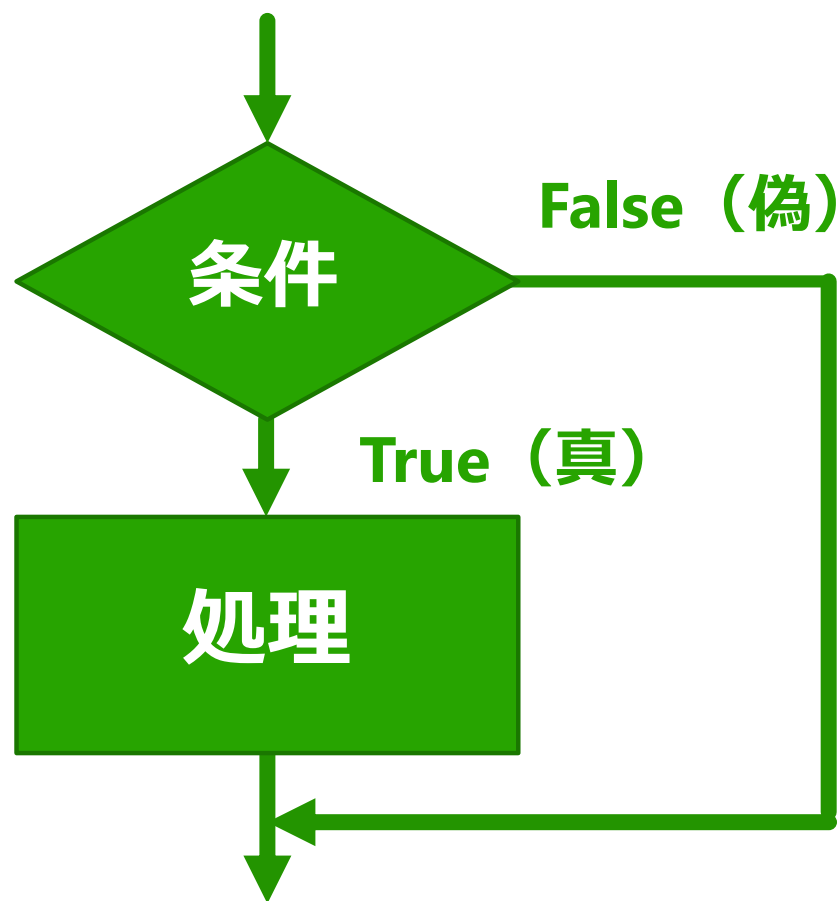
pythonでの論理演算子は以下

演算子	真の条件
x or y	x または yが 真
x and y	x と yの両方が真
x not	xが真であれば偽 xが偽であれば真

# 1. if文

# if文

if文:**条件分岐**を行うときに使用 (もしも〇〇という【条件】ならば△△という【処理】を実行)



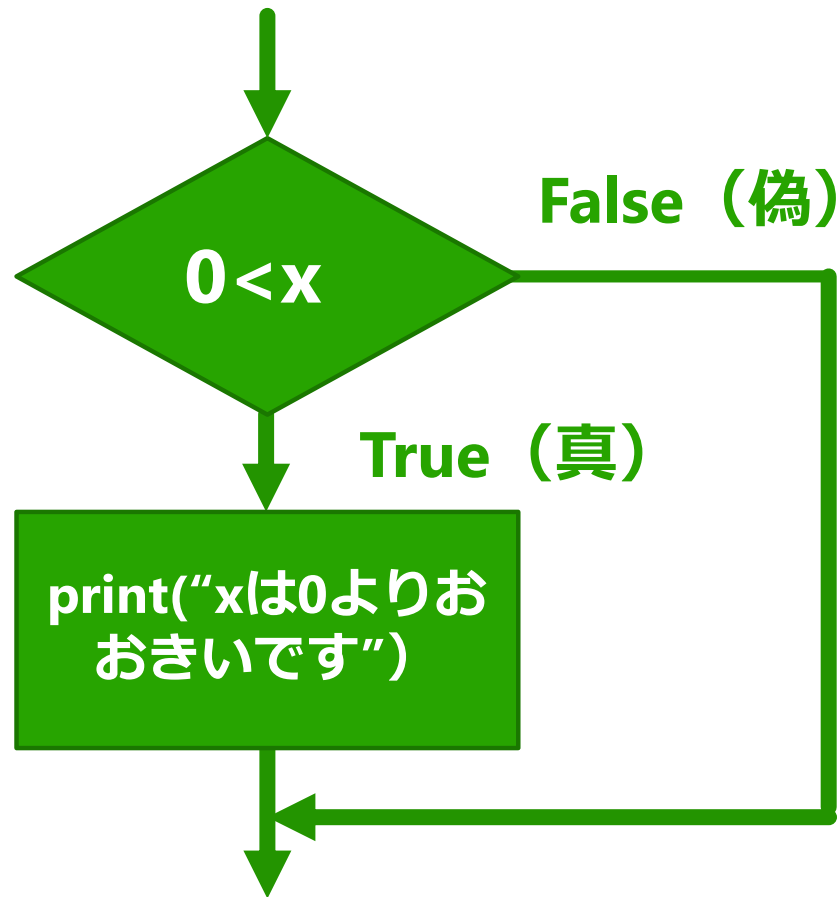
左図をプログラムで表すと下記

```
if 条件:  
    処理
```



# if文

例) 変数xが0より大きいならprint

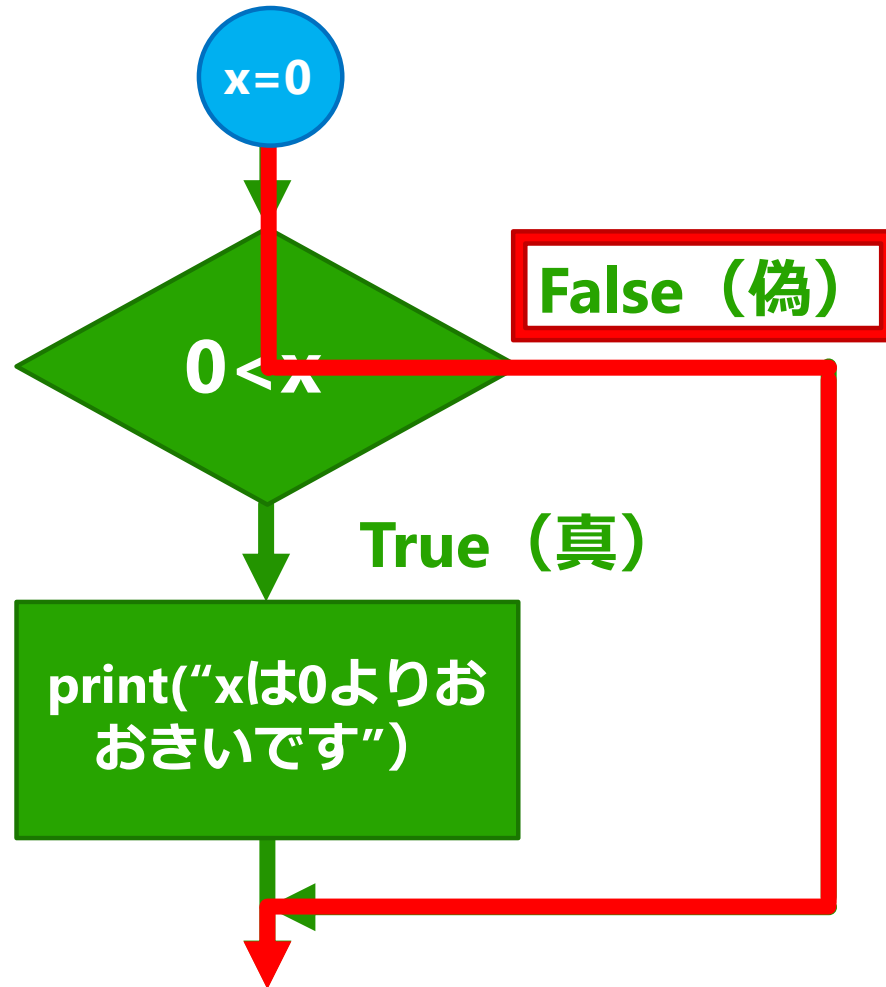


左図をプログラムで表すと下記

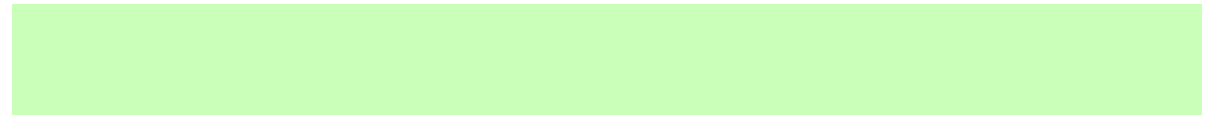
```
if x > 0:  
    print("xは0よりおおきいです")
```

# if文

1)  $x = 0$

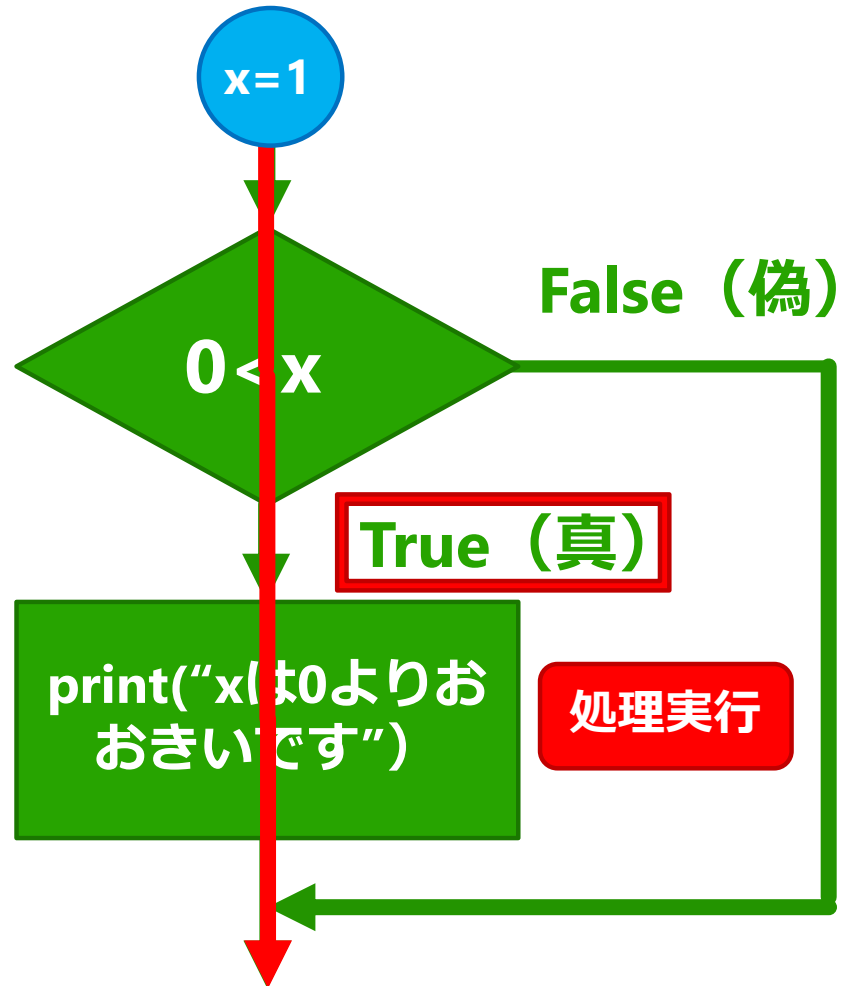


出力結果 ※出力されない



# if文

2)  $x = 1$

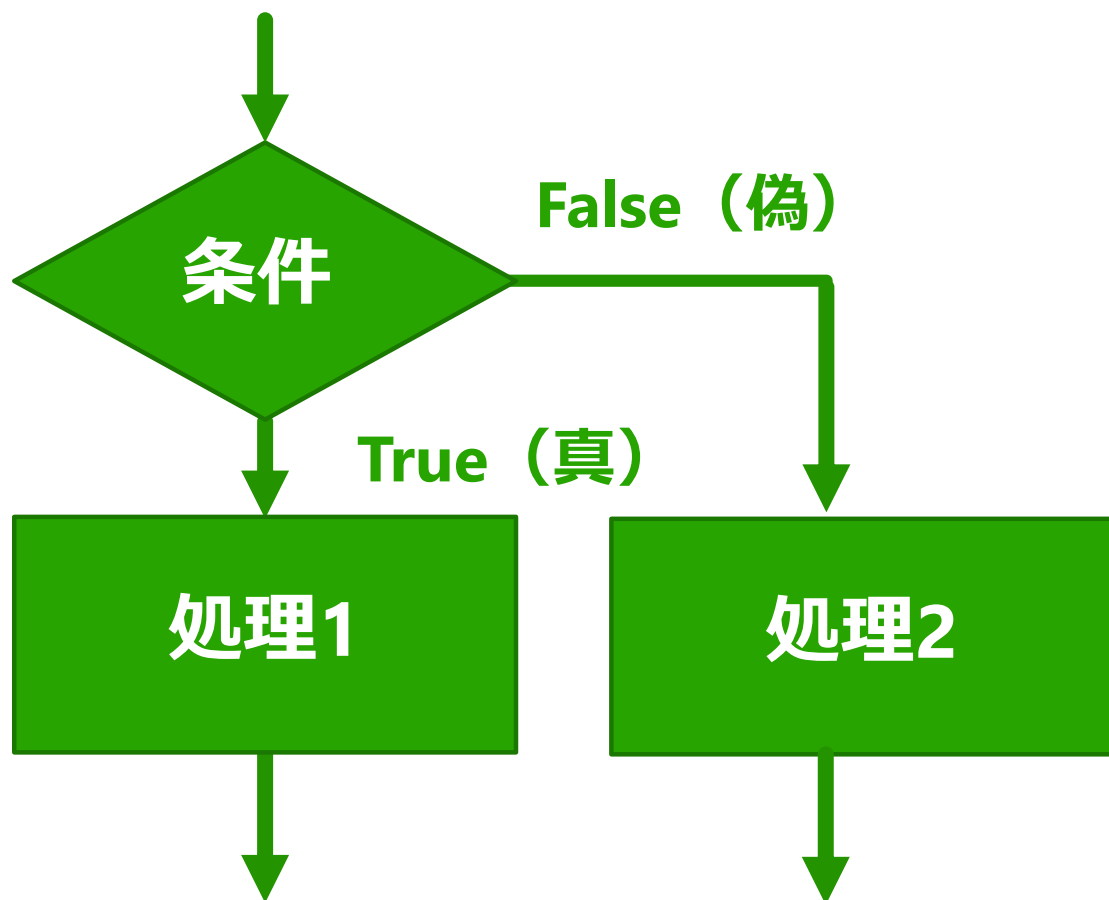


## 出力結果

xは0よりおおきいです

# else

else : if文の条件に当てはまらなかった場合に実行

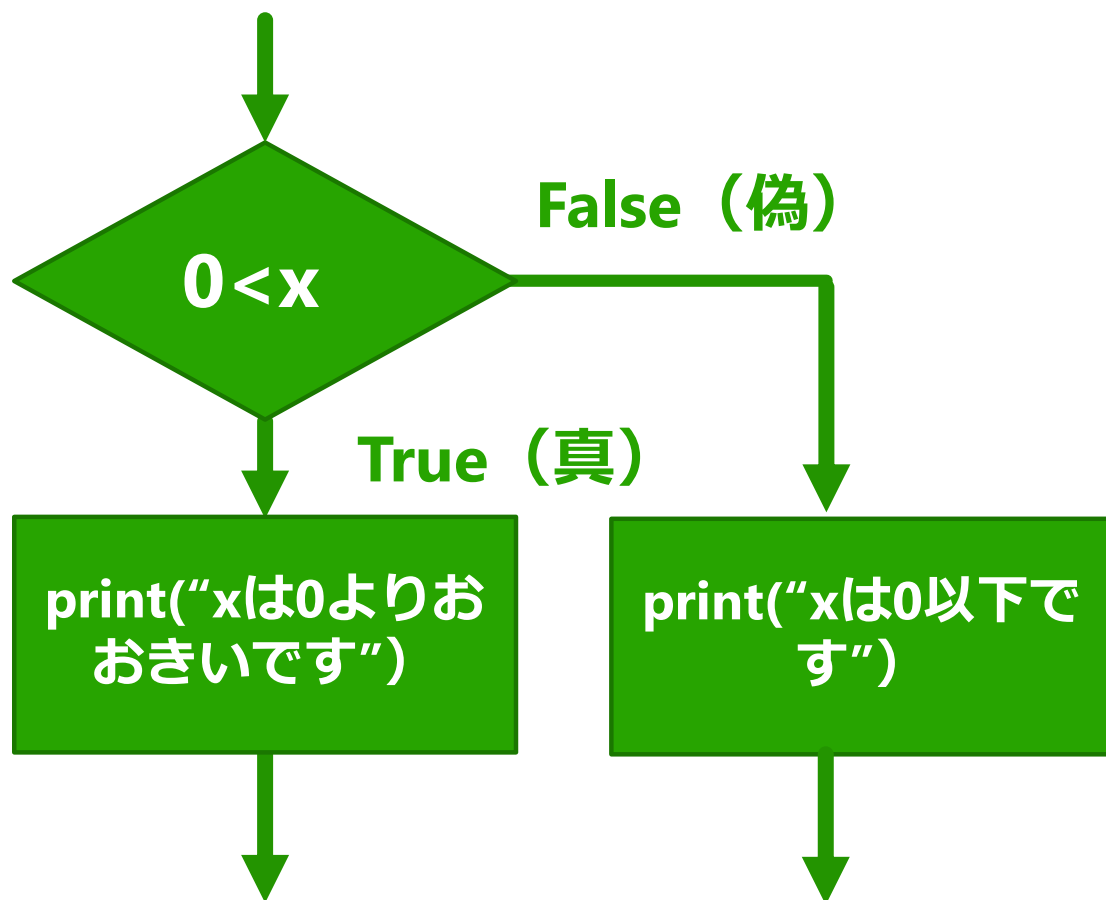


左図をプログラムで表すと下記

```
if 条件:  
    処理1  
else:  
    処理2
```

# else

例)  $0 < x$ ならば"xは0よりおおきいです"と出力、そうでなければ"xは0以下です"と出力

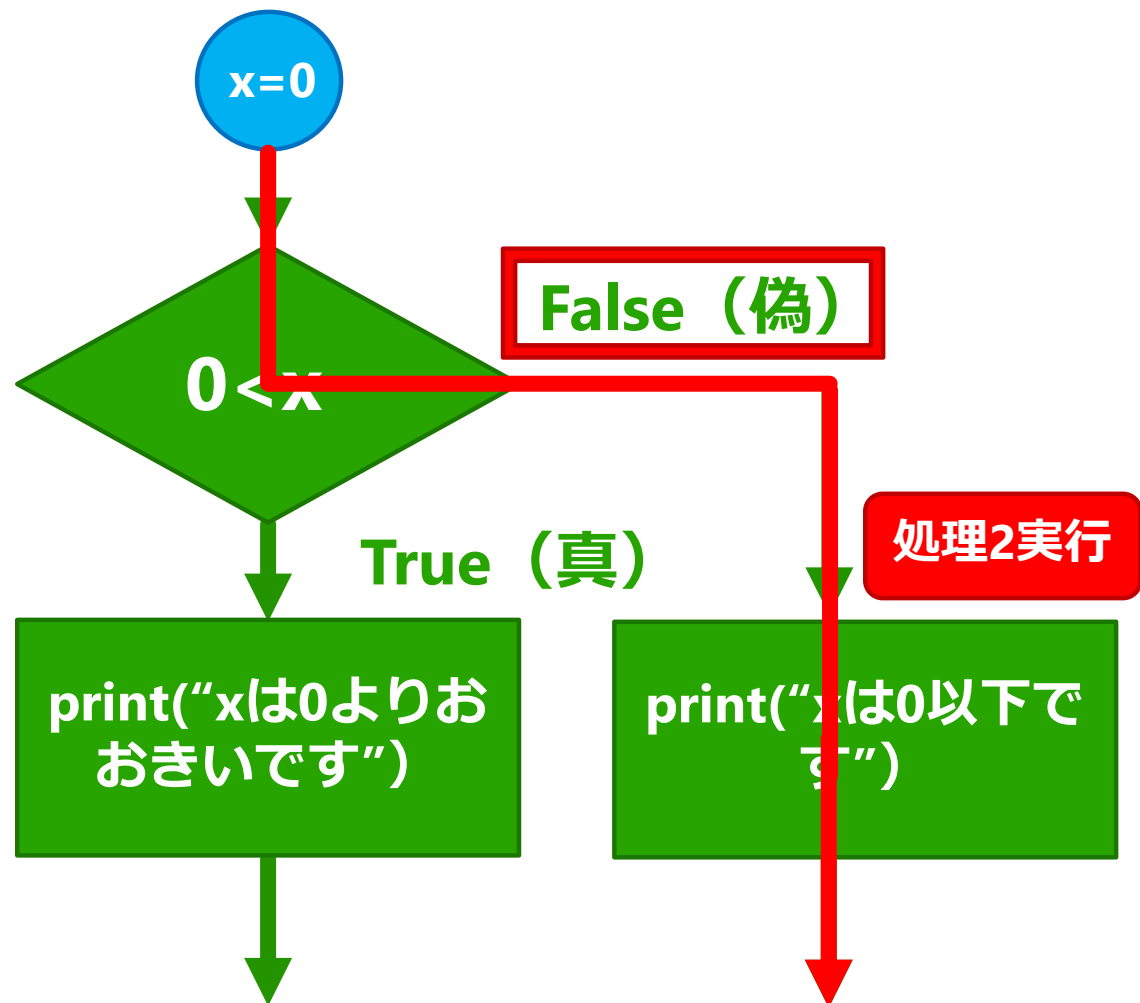


左図をプログラムで表すと下記

```
if 0 < x:  
    print("xは0よりおおきいです")  
else:  
    print("xは0以下です")
```

# else

1)  $x = 0$

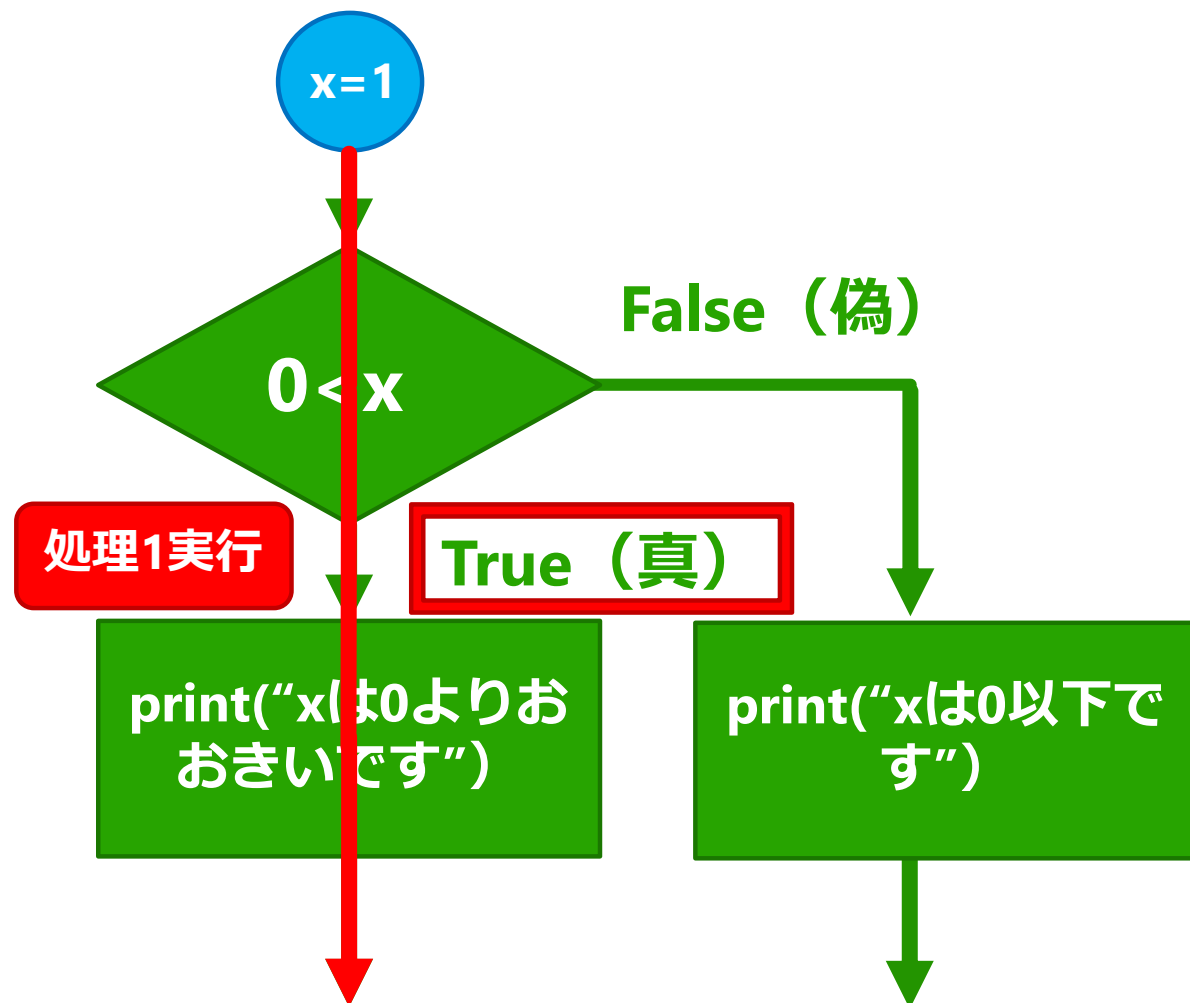


出力結果

xは0以下です

# else

2)  $x = 1$

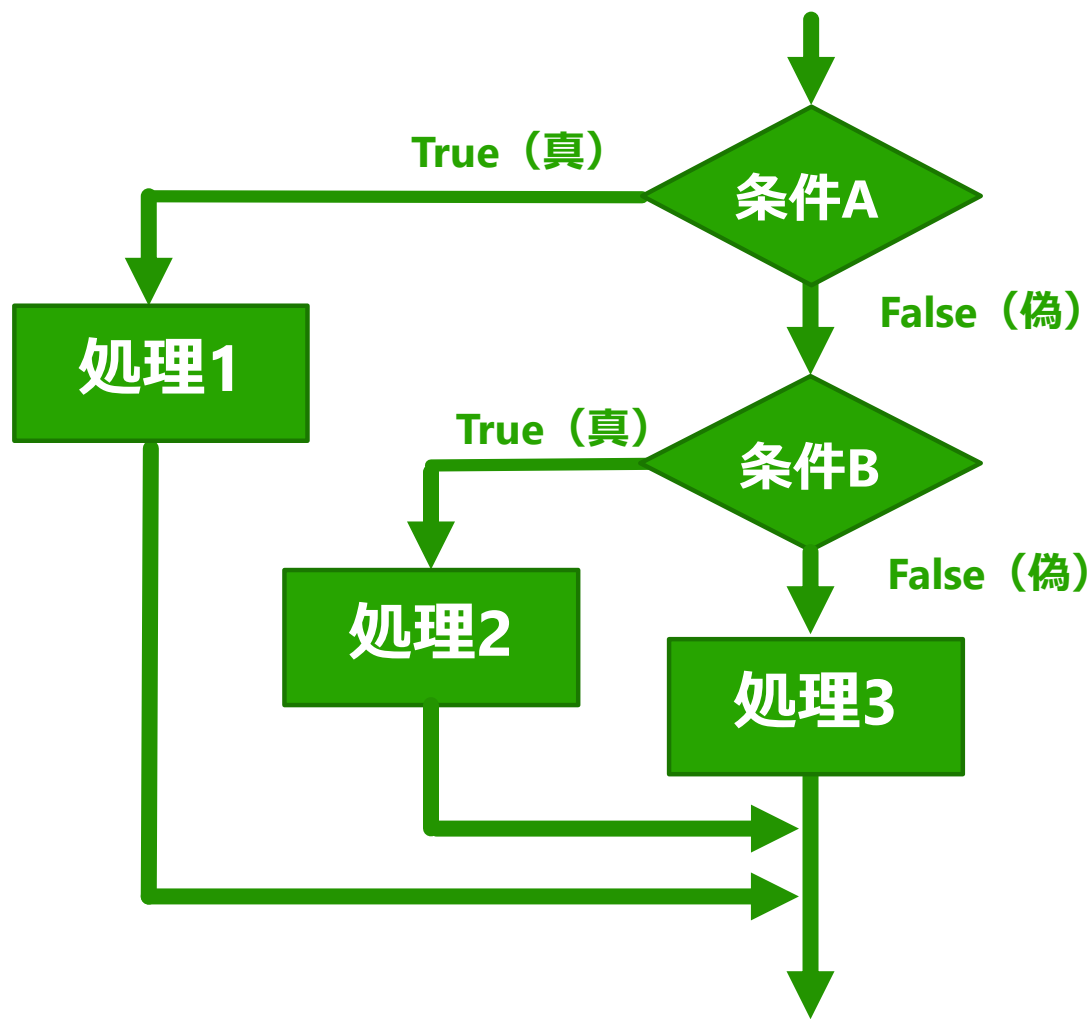


出力結果

xは0よりおおきいです

# elif

elif : 【条件Aにあてはまらなかったら条件B、条件Bに当てはまらなかったら条件C...】  
というような処理を実行したい時に使用（下図は条件が二つ）



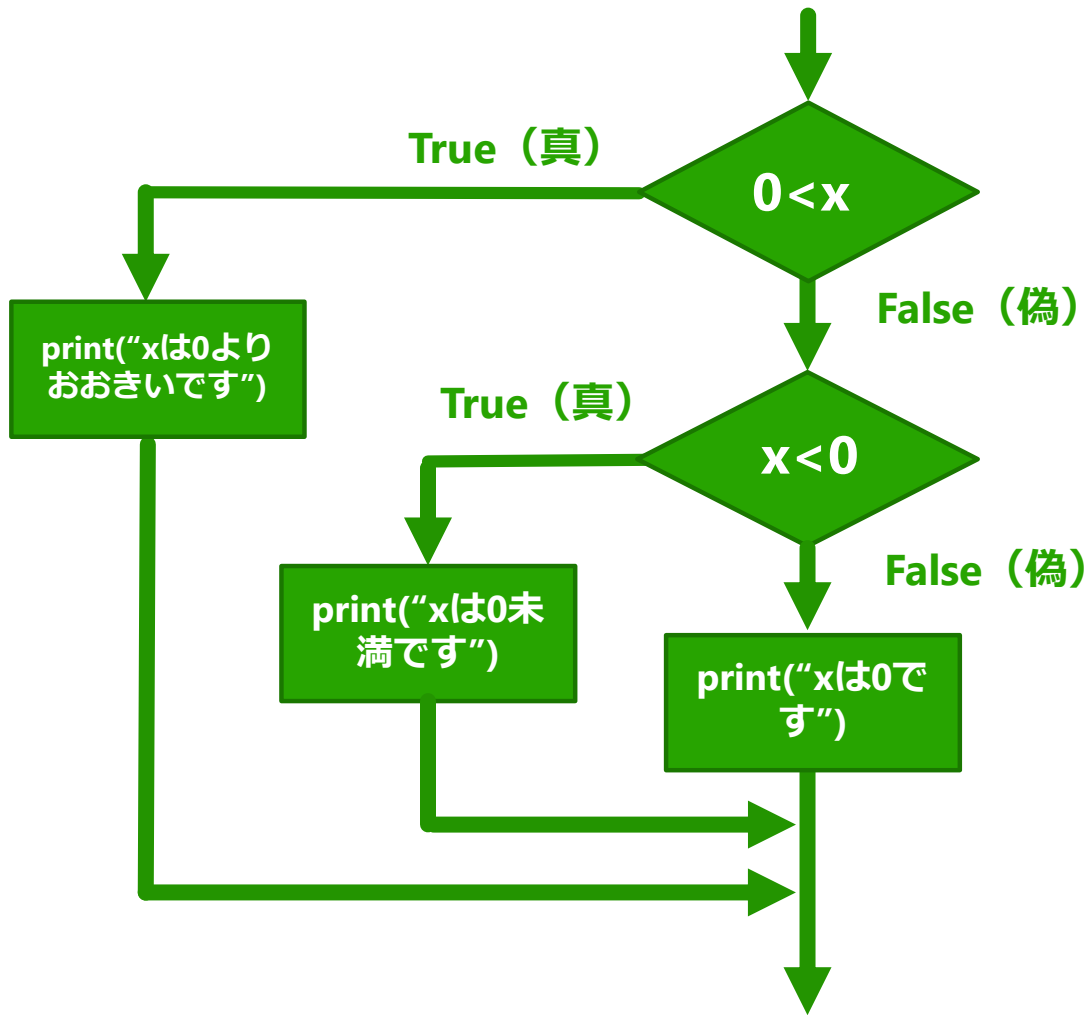
左図をプログラムで表すと下記

```
if 条件A:  
    処理1  
elif 条件B:  
    処理2  
else:  
    処理3
```



# elif

例)  $0 < x$ ならば、"xは0より大きいです"、 $x < 0$ ならば、"xは0未満です"  
それ以外ならば、"xは0です"と表示

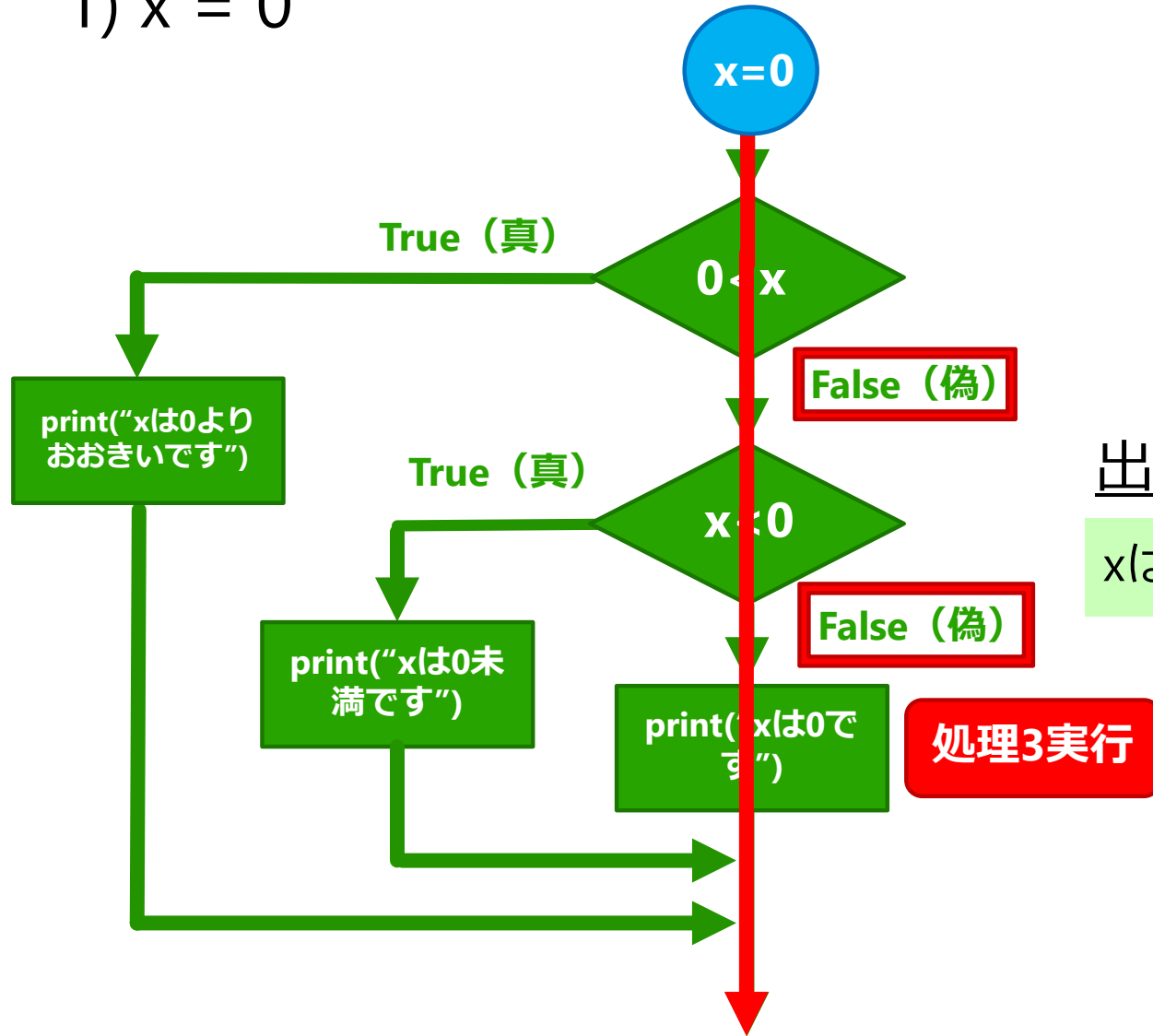


左図をプログラムで表すと下記

```
if 0 < x:
    print("xは0よりおおきいです")
elif x < 0:
    print("xは0未満です")
else:
    print("xは0です")
```

# elif

1)  $x = 0$

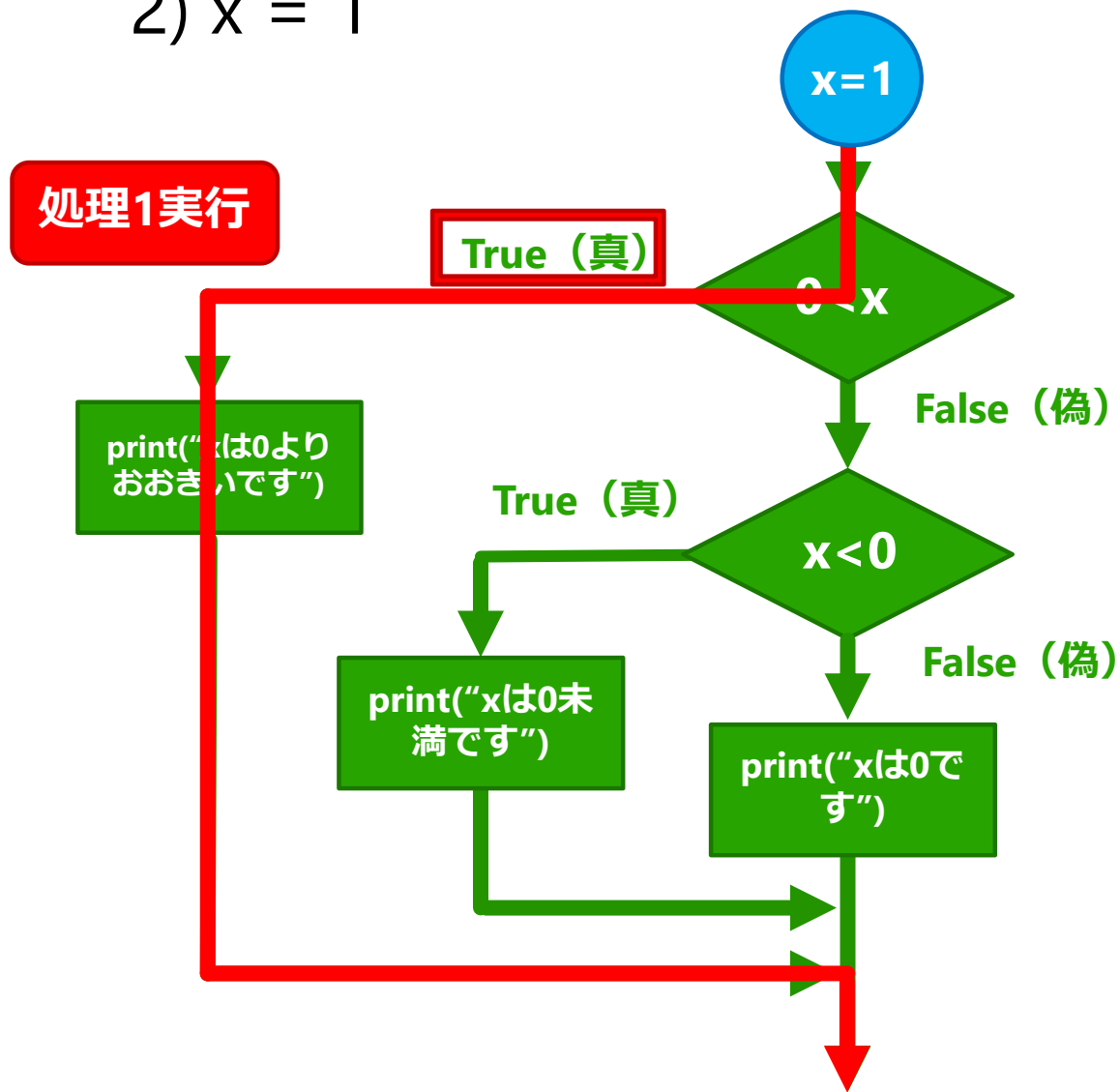


出力結果

xは0です

# *elif*

2)  $x = 1$

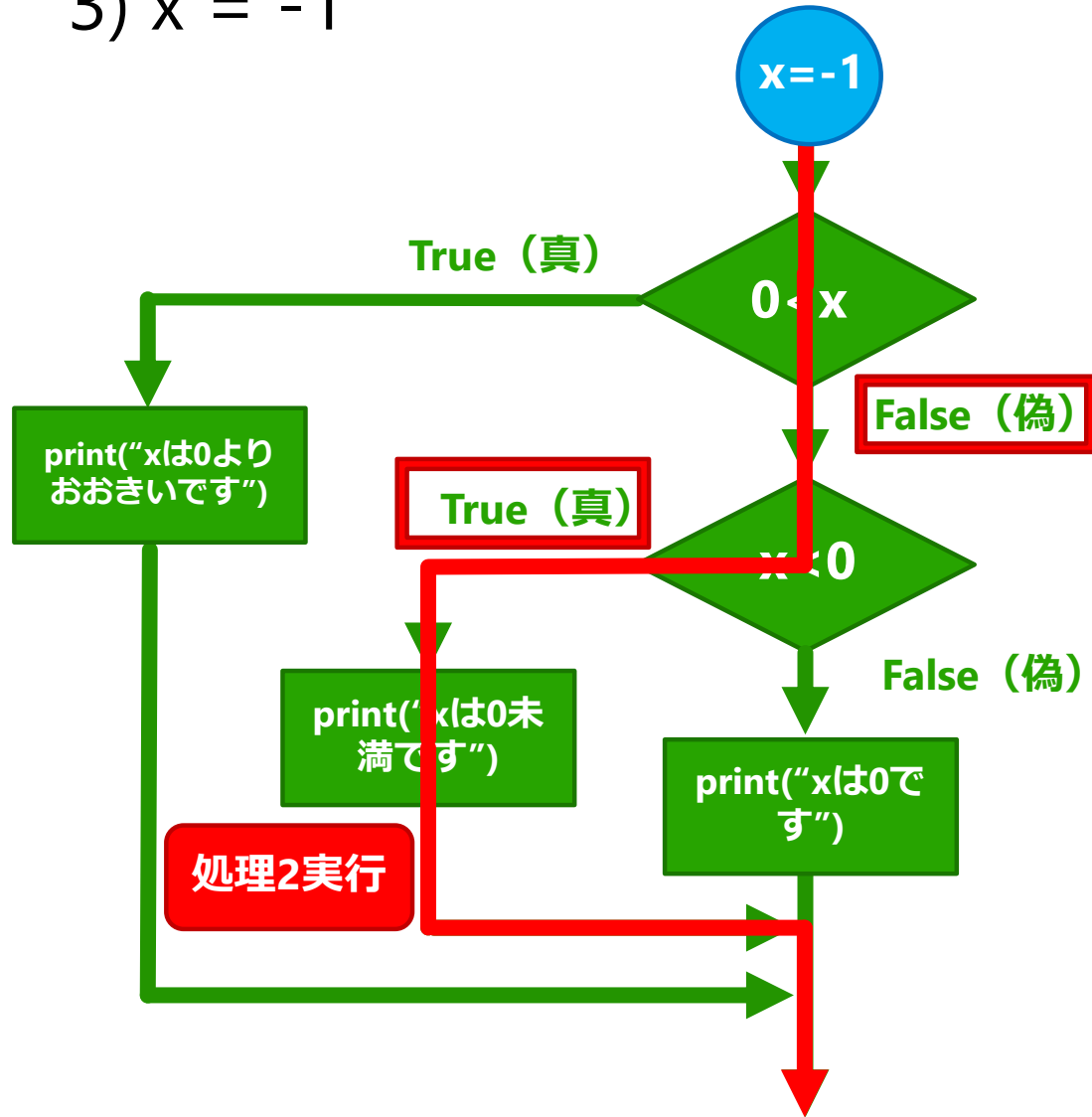


## 出力結果

xは0より大きいです

# elif

3)  $x = -1$

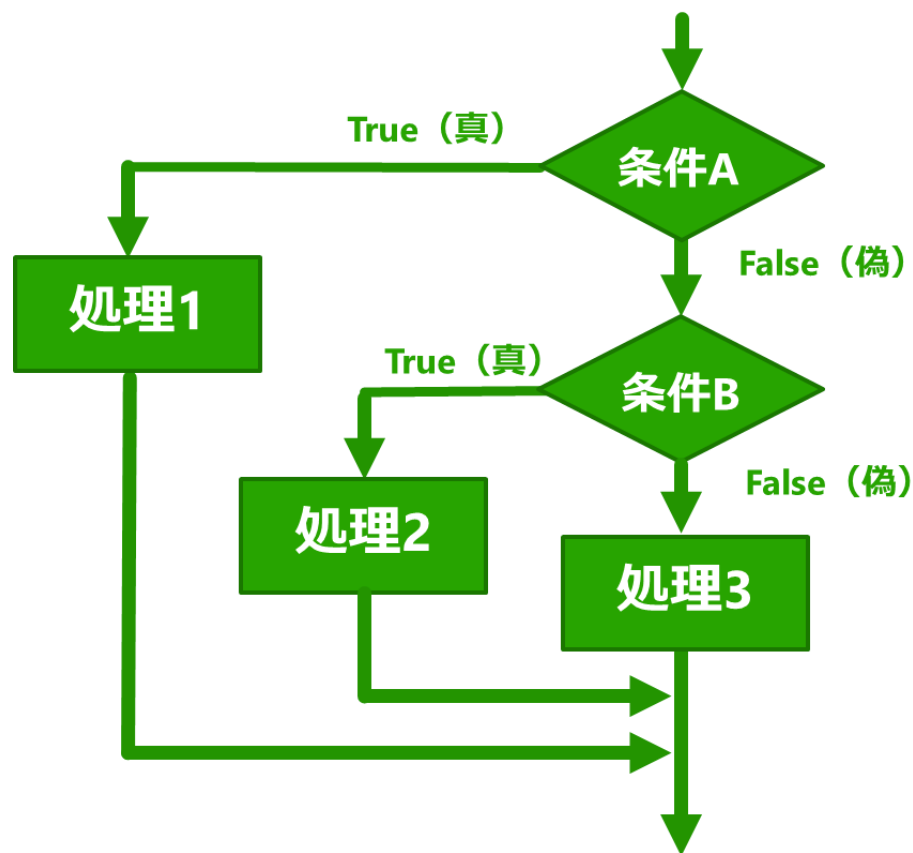


## 出力結果

xは0未満です

# if文 優先順位

if文の優先順位は、フローチャートをもっても分かるように上から順番



```
if 条件A:  
    処理1  
elif 条件B:  
    処理2  
else:  
    処理3
```

高  
優先順位  
低

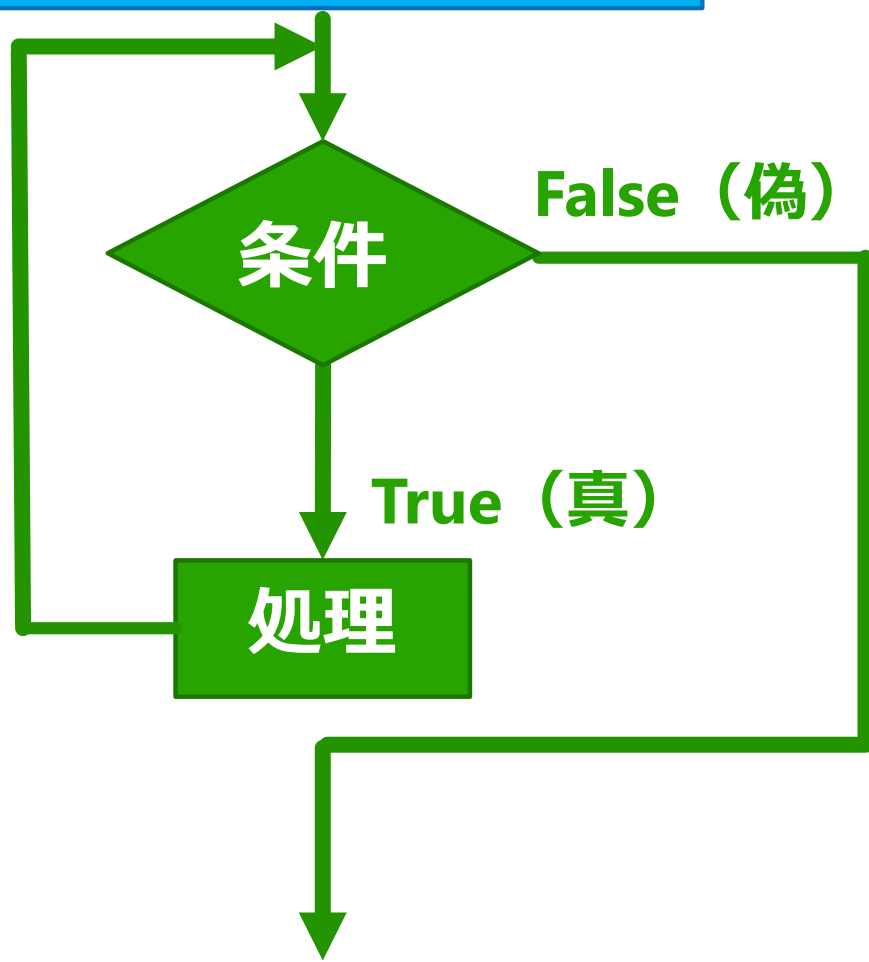
フローチャートを用いると行っている処理が視覚的にわかりやすくなる

2. for 文

# for文

for文：繰り返し処理をする時に用いる。

イテラブルオブジェクト



※他の言語におけるforeach文がpythonでの標準の書き方

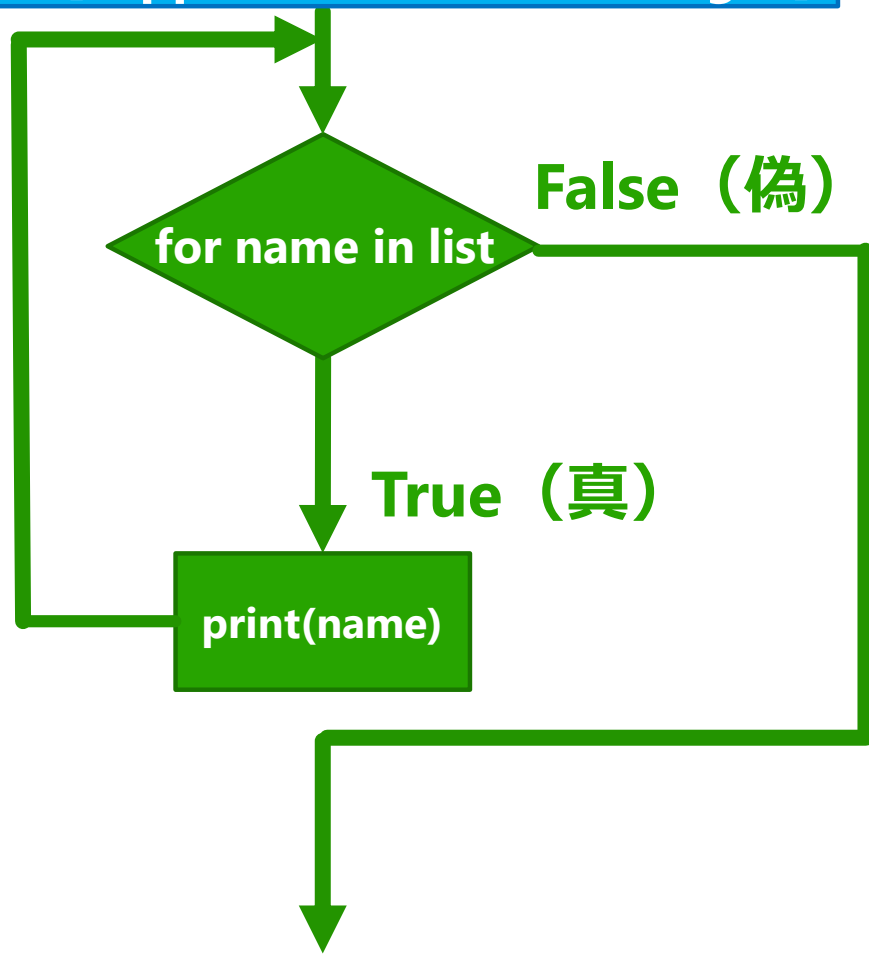
左図をプログラムで表すと下記

```
for 変数名 in イテラブルオブジェクト:  
    処理
```

# for文 例

例) list内の要素を順番に、nameに代入する

```
list = ["apple", "banana", "orange"]
```



```
list = ["apple", "banana", "orange"]
```

左図をプログラムで表すと下記

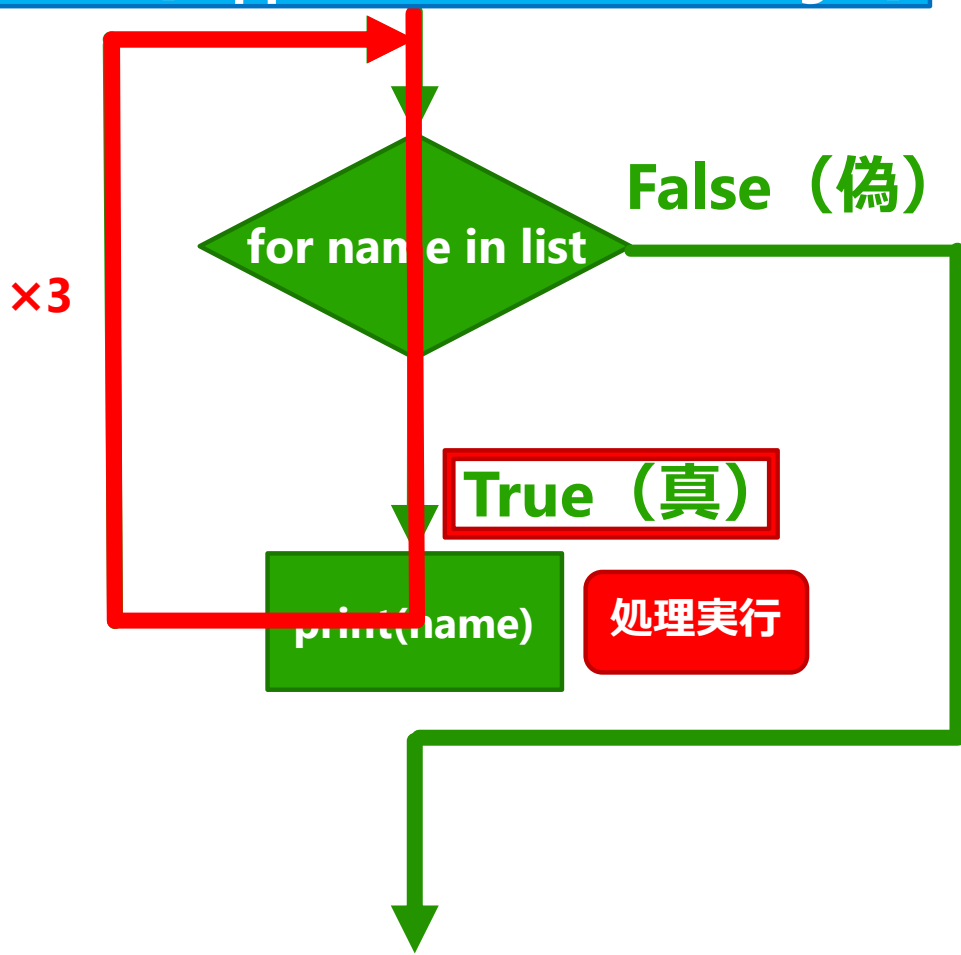
```
for name in list:  
    print(name)
```



# for文 例

①for文1～3週目 (list内の要素を順番に、nameに代入する)

```
list = ["apple", "banana", "orange"]
```



## 出力結果

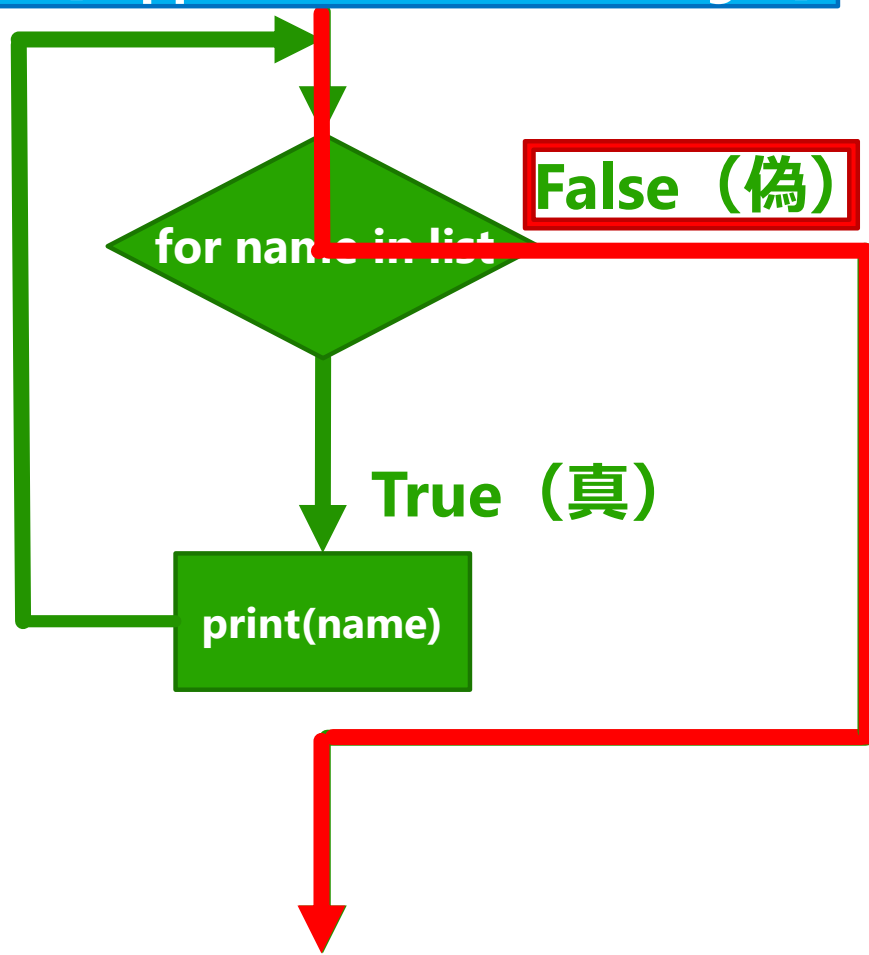
```
apple  
banana  
orange
```

# for文 例

## ②for文から抜ける

(list内の要素を順番に、nameに代入する)

```
list = ["apple", "banana", "orange"]
```



## 出力結果

```
apple  
banana  
orange
```

# for文 イメージ

listからnameに順番に要素を入れる

```
for name in list:  
    print(name)
```

name



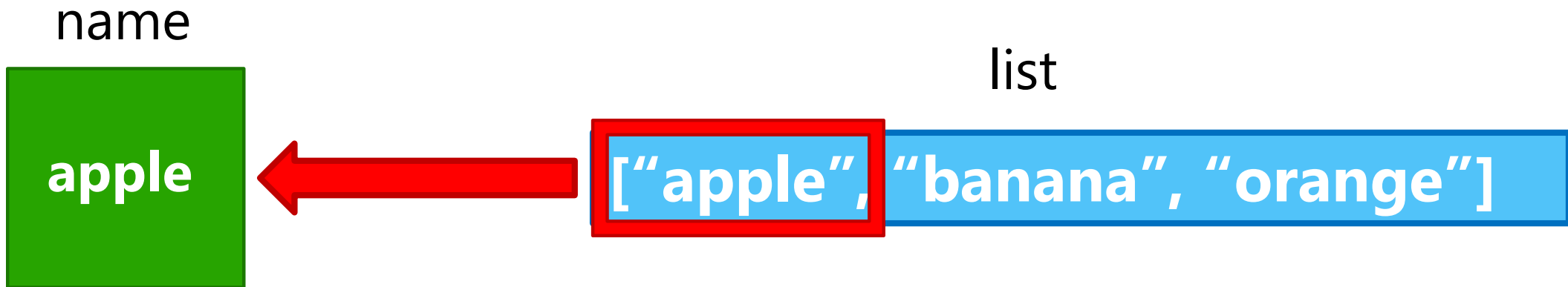
list

["apple", "banana", "orange"]

# for文 イメージ

## for文1週目

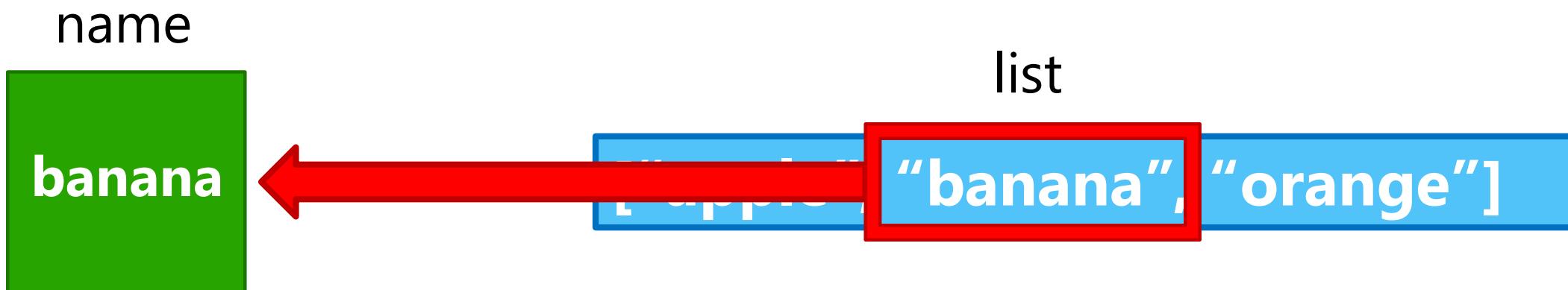
```
for name in list:  
    print(name)
```



# for文 イメージ

## for文2週目

```
for name in list:  
    print(name)
```

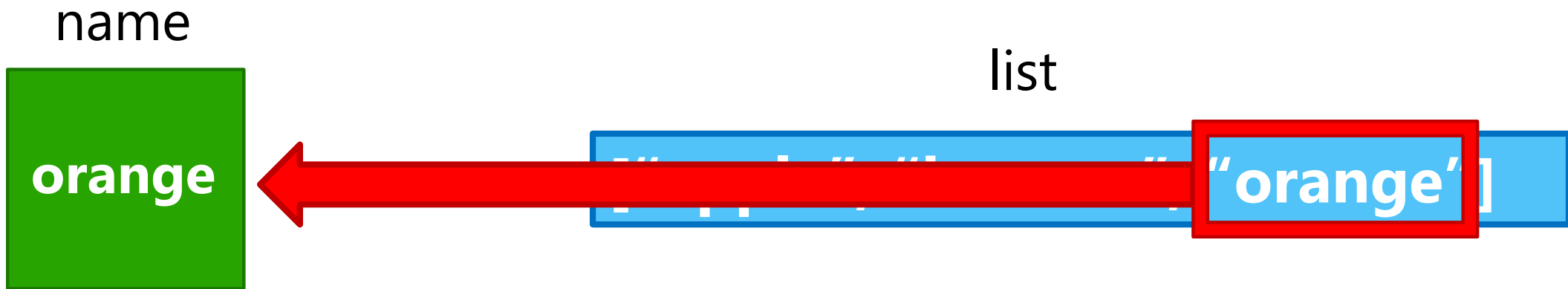


apple ➡ bananaに上書き

# for文 イメージ

## for文3週目

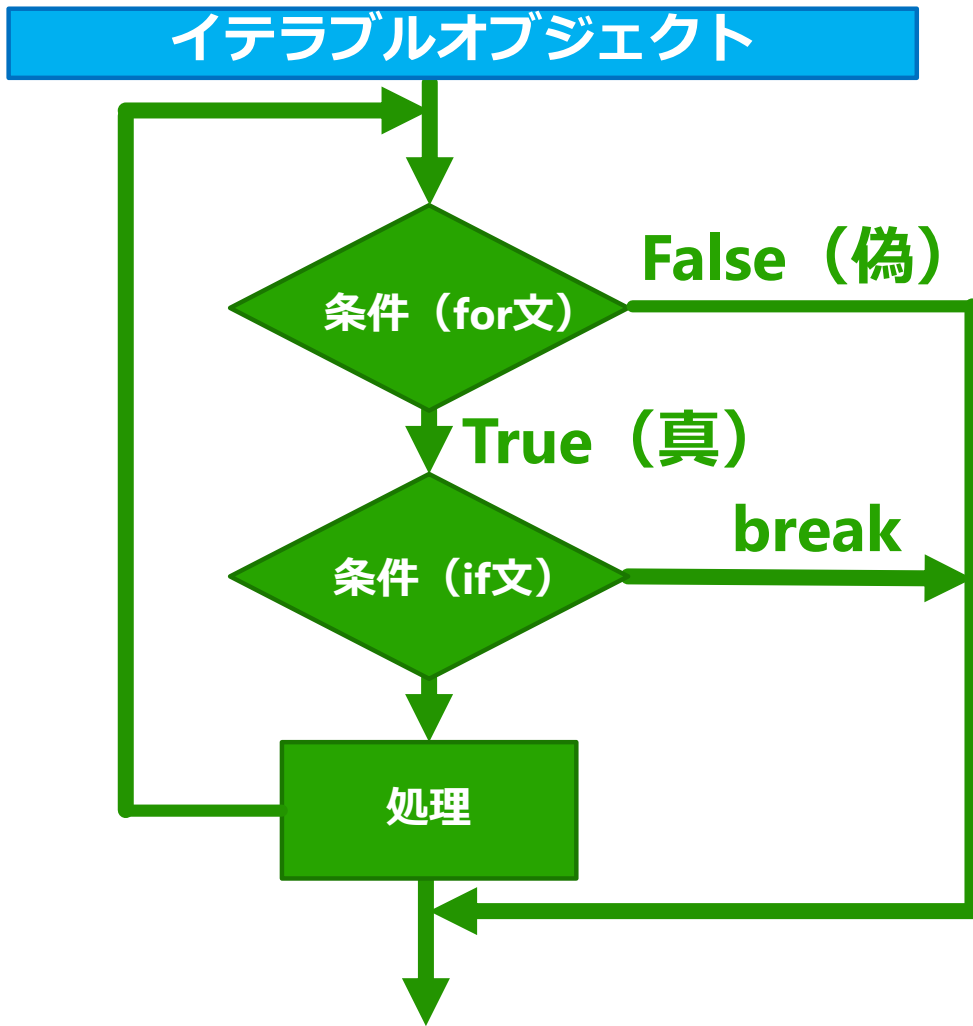
```
for name in list:  
    print(name)
```



banana ➡ orangeに上書き

# for文 : break

break : ある条件でfor文から抜ける



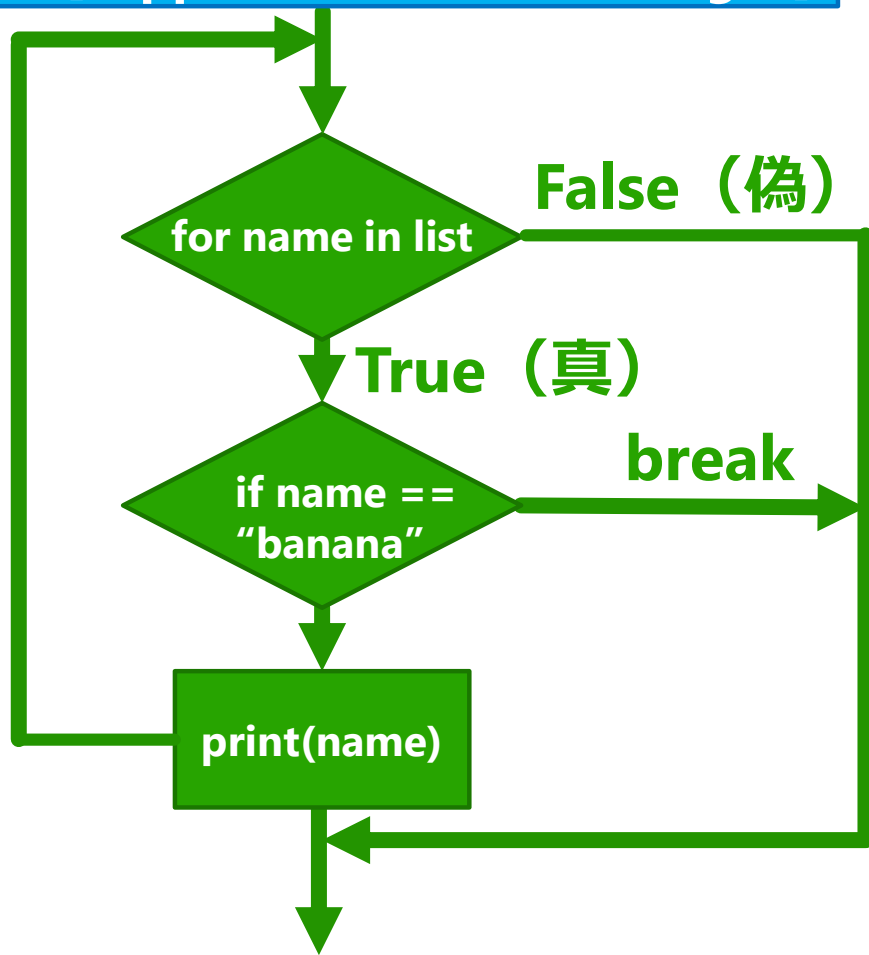
左図をプログラムで表すと下記

```
for 変数名 in イテラブルオブジェクト
    if 条件:
        break
    処理
```

# for文 : break 例

例) name=="banana"の時にbreak

```
list = ["apple", "banana", "orange"]
```



```
list = ['apple', 'banana', orange]
```

左図をプログラムで表すと下記

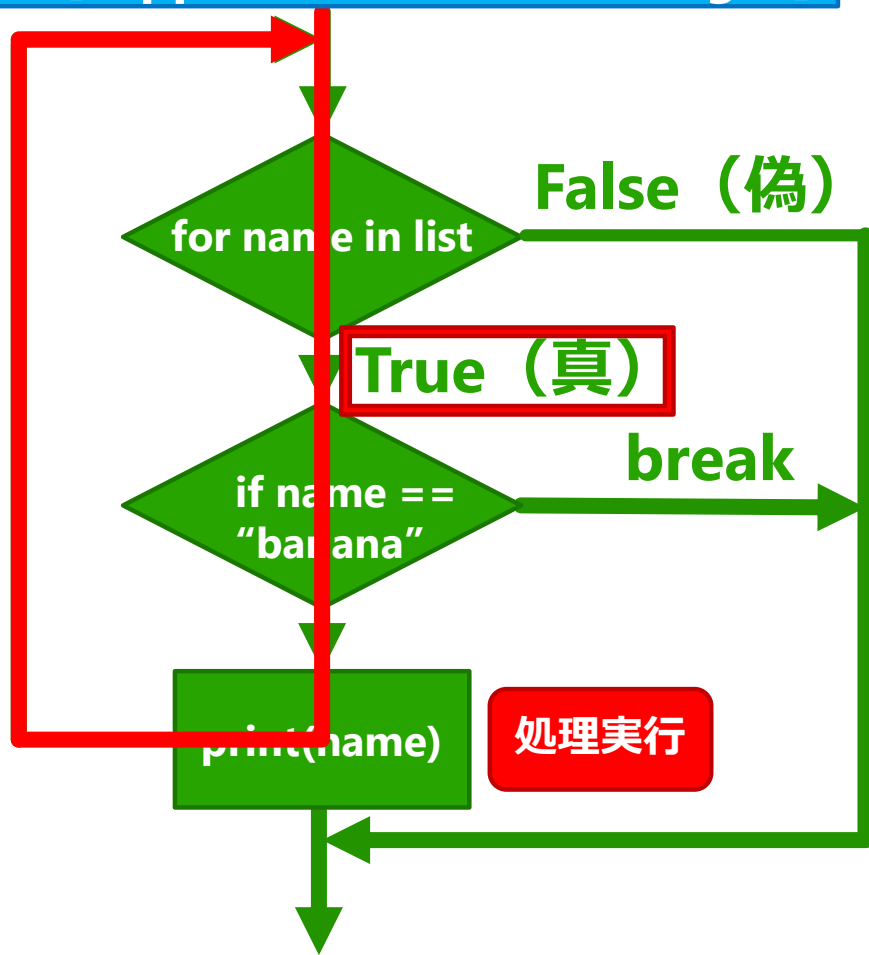
```
for name in list:  
    if name == "banana":  
        break  
    print(name)
```



# for文 : break 例

①for文1週目 (name=="banana"の時にbreak)

```
list = ["apple", "banana", "orange"]
```



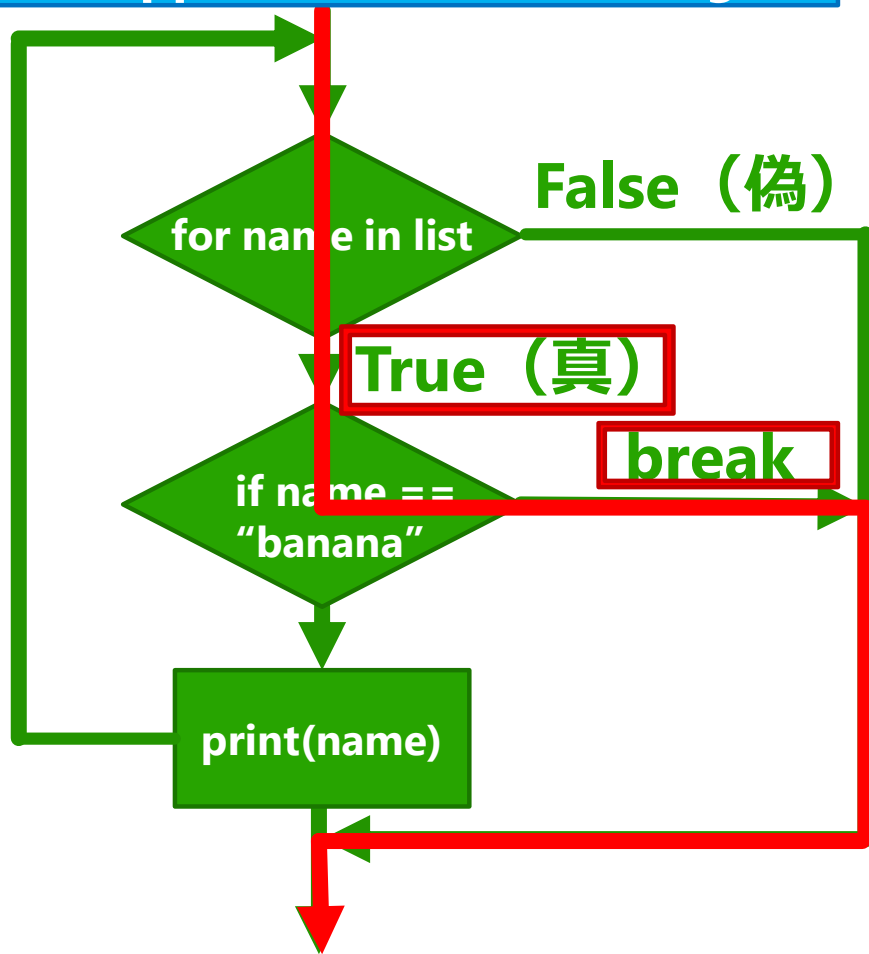
出力結果

apple

# for文 : break 例

② name == 'banana'なのでbreak ➡ for文終了 (name == "banana"の時にbreak)

```
list = ["apple", "banana", "orange"]
```



## 出力結果

apple

breakしたので"banana", "orange"は出力されない

# *for*文 : *continue*

`continue` : ある条件で一回スキップしてfor文を継続

イテラブルオブジェクト

条件 (for文)

False (偽)

True (真)

条件 (if文)

処理

`continue`

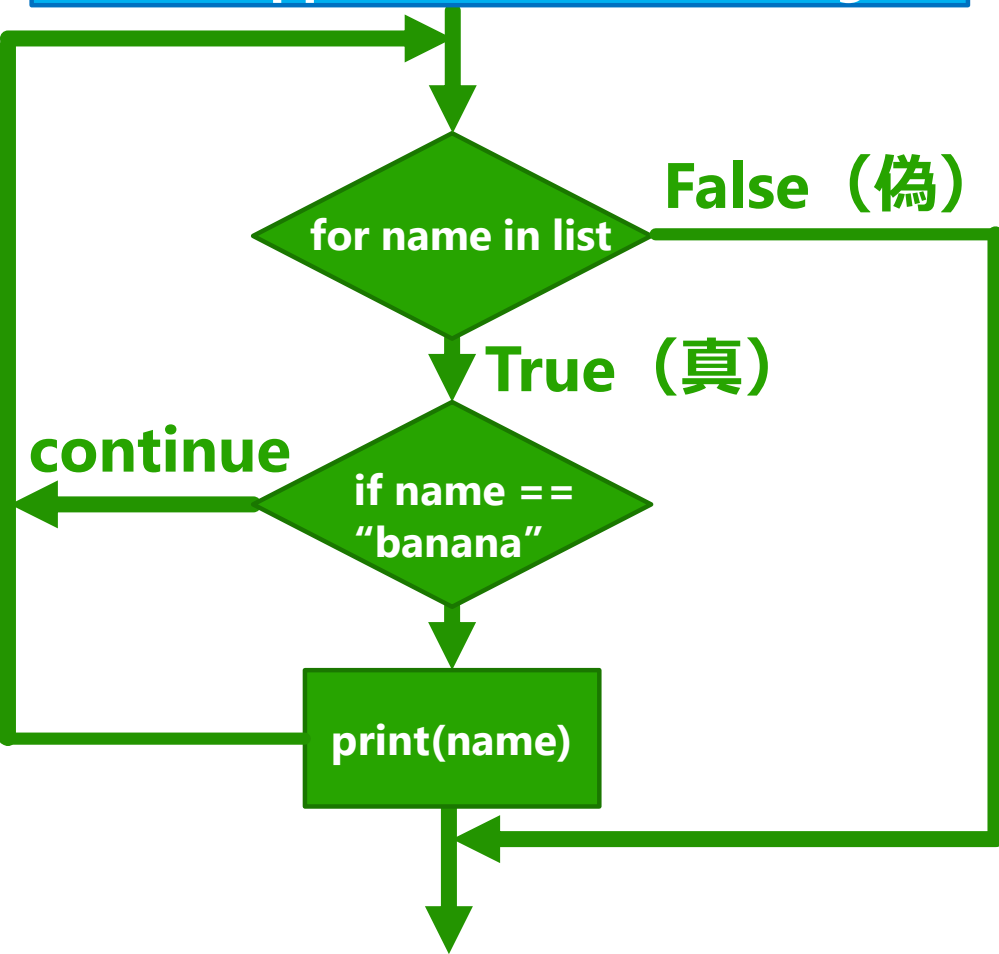
左図をプログラムで表すと下記

```
for 変数名 in イテラブルオブジェクト:  
    if 条件:  
        continue  
    処理
```

# for文 : continue 例

例) name == 'banana' のときcontinue

```
list = ["apple", "banana", "orange"]
```



```
list = ["apple", "banana", "orange"]
```

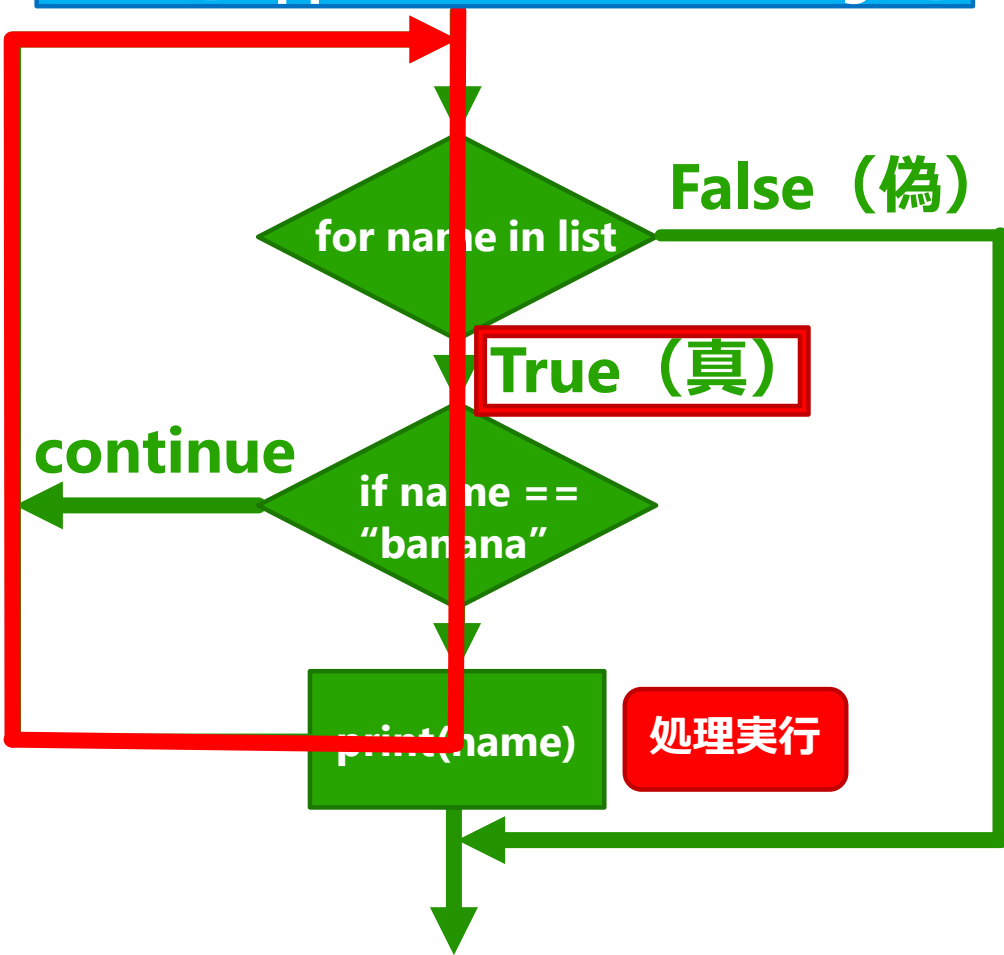
左図をプログラムで表すと下記

```
for name in list:  
    if name == "banana":  
        continue  
    print(name)
```

# for文 : continue 例

①for文一週目 (name == "banana" のときcontinue)

```
list = ["apple", "banana", "orange"]
```



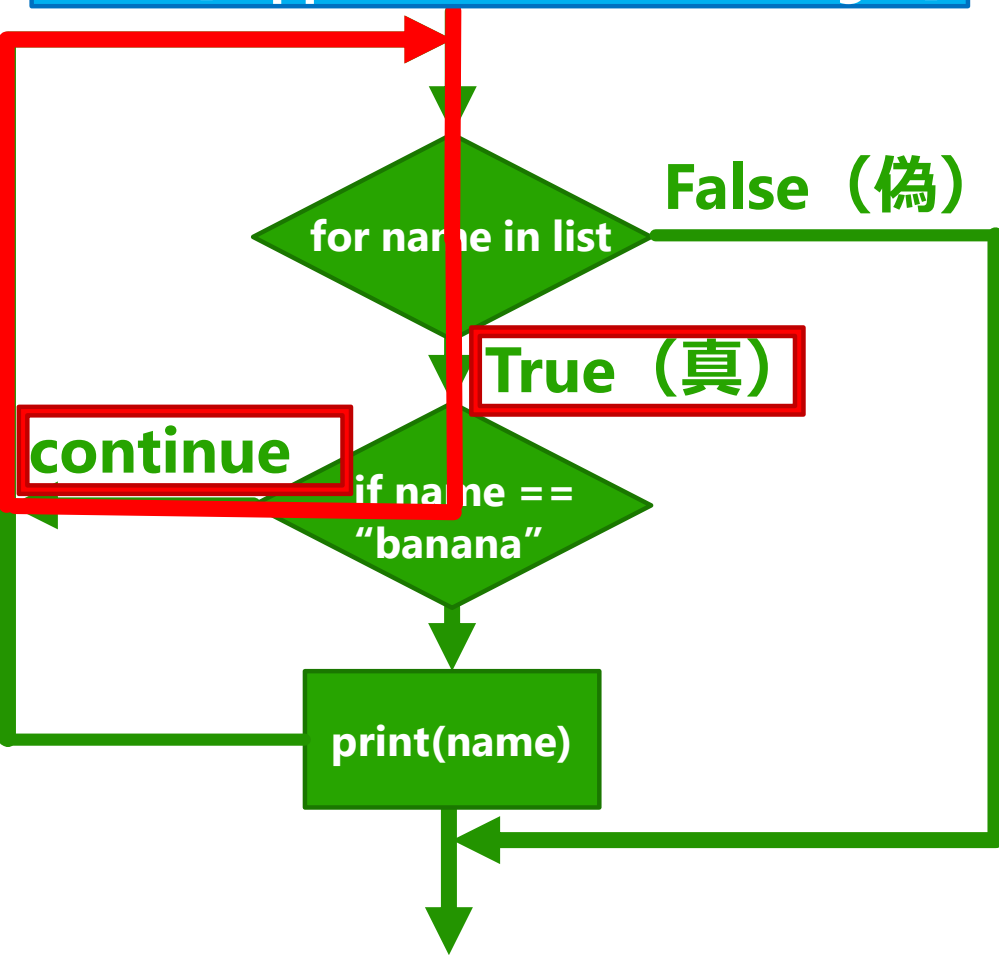
出力結果

apple

# for文 : continue 例

②for文2週目はname == "banana"なのでcontinueで一回スキップ  
(name == 'banana' のときcontinue)

```
list = ["apple", "banana", "orange"]
```



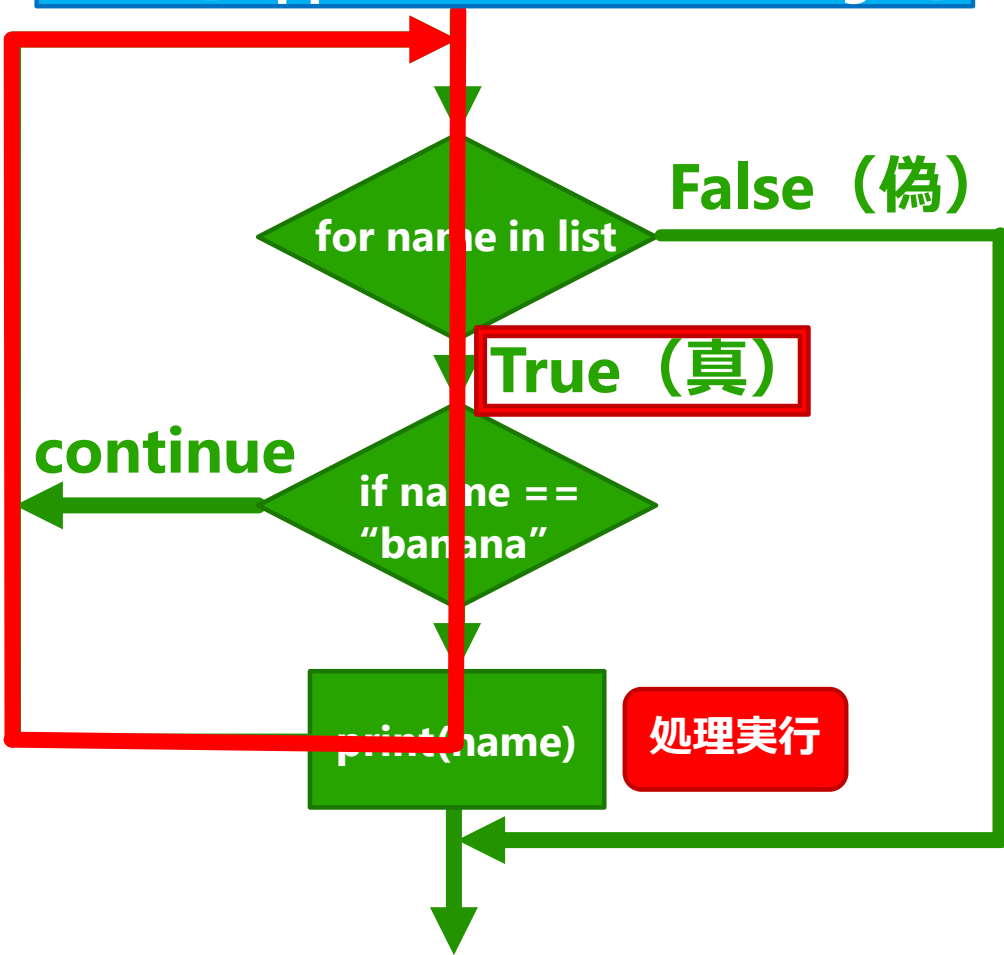
## 出力結果

apple

# for文 : continue 例

## ③for文3週目 (name == "banana" のときcontinue)

```
list = ["apple", "banana", "orange"]
```



## 出力結果

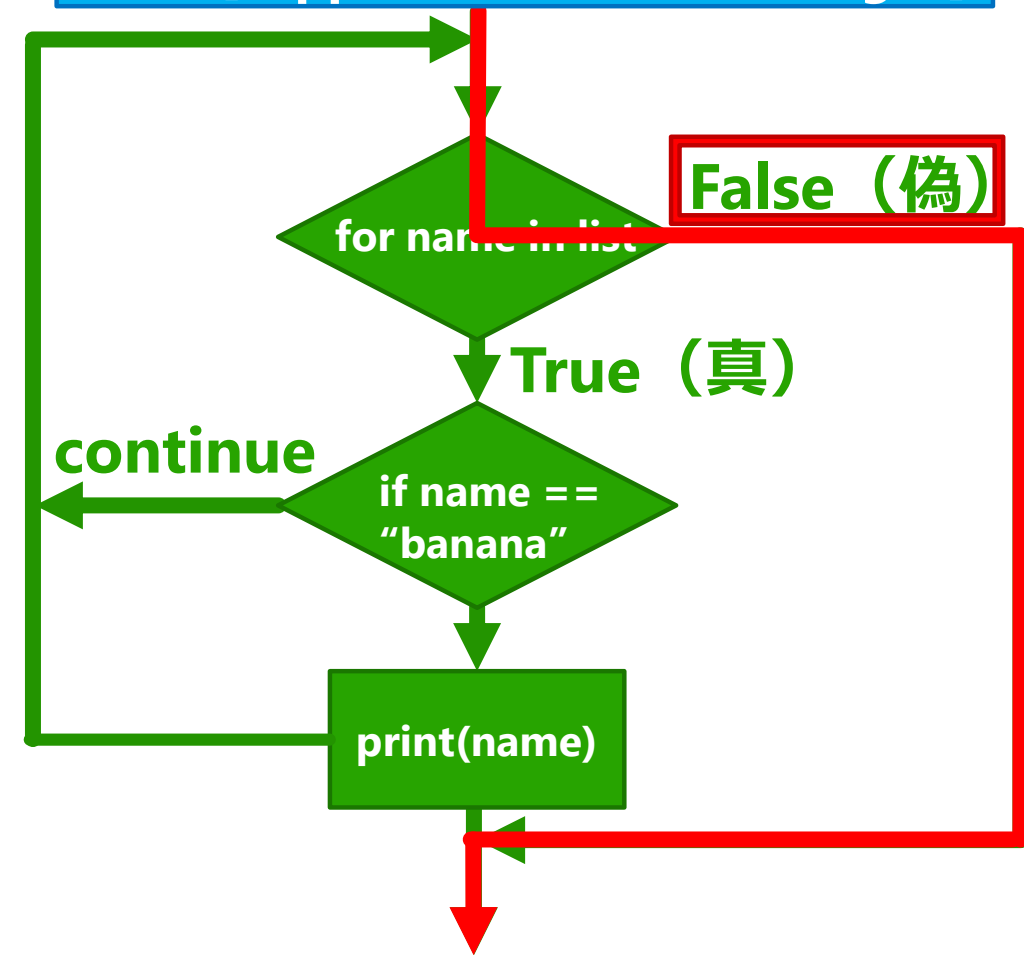
```
apple  
orange
```

# for文 : continue 例

## ④for文から抜ける

(name == "banana" のときcontinue)

```
list = ["apple", "banana", "orange"]
```



## 出力結果

apple  
orange



# *for*文で使うと便利な関数

---

- インデックス（カウンタ）：range()関数
- リストの要素とインデックス: enumerate()関数
- 複数リストの要素（複数変数）：zip()関数

# *range()*関数①

for文でインデックス（カウンタ）を取得

- `range(stop)`: 0からstop未満までの値を連番で取得

例) 

```
for i in range(3):  
    print(i)
```

## 出力結果

```
0  
1  
2
```

**※この時、指定した値（上の例の場合3）は含まれないので注意**

## *range()*関数②

for文でインデックス（カウンタ）を取得

- `range(start, stop)`: `start`～`stop`未満までの連番を取得

例) 

```
for i in range(2,5):  
    print(i)
```

### 出力結果

```
2  
3  
4
```

## *range()*関数③

for文でインデックス（カウンタ）を取得

- `range(start, step, stop)`: start～stop未満までstepずつ増加する値を取得

例) 

```
for i in range(3, 9, 3):  
    print(i)
```

3から9まで3飛ばしで取得

### 出力結果

```
3  
6
```

# *enumerate()*関数

リストなどのイテラブルオブジェクトの要素とインデックス（カウンタ）を同時に取得

例)

```
list = ['apple', 'banana', orange]

for name in enumerate(list):
    print(i, name)
```

## 出力結果

```
0 apple
1 banana
2 orange
```

# enumerate()関数

リストなどのイテラブルオブジェクトの要素とインデックス（カウンタ）を同時に取得

例)

```
list = ["apple", "banana", "orange"]
```

```
for name in enumerate(list):  
    print(i, name)
```

listの要素 → 順番にname に格納

## 出力結果

インデックス(i)

0	apple
1	banana
2	orange

要素

(name)

# zip()関数

複数のイテラブルオブジェクトの要素を複数の変数としてまとめて取得

例)

```
names = ["apple", "banana", "orange"]  
prices = [80, 100, 150]  
  
for name, price in zip(names, prices):  
    print(name, price)
```

## 出力結果

```
apple 80  
banana 100  
orange 150
```

# zip()関数

複数のイテラブルオブジェクトの要素を複数の変数としてまとめて取得

例)

```
names = ["apple", "banana", "orange"]
```

```
prices = [80, 100, 150]
```

```
for name, price in zip(names, prices):  
    print(name, price)
```



namesの要素 → 順番にname に格納

pricesの要素 → 順番にpriceに格納

## 出力結果

```
apple 80
```

```
banana 100
```

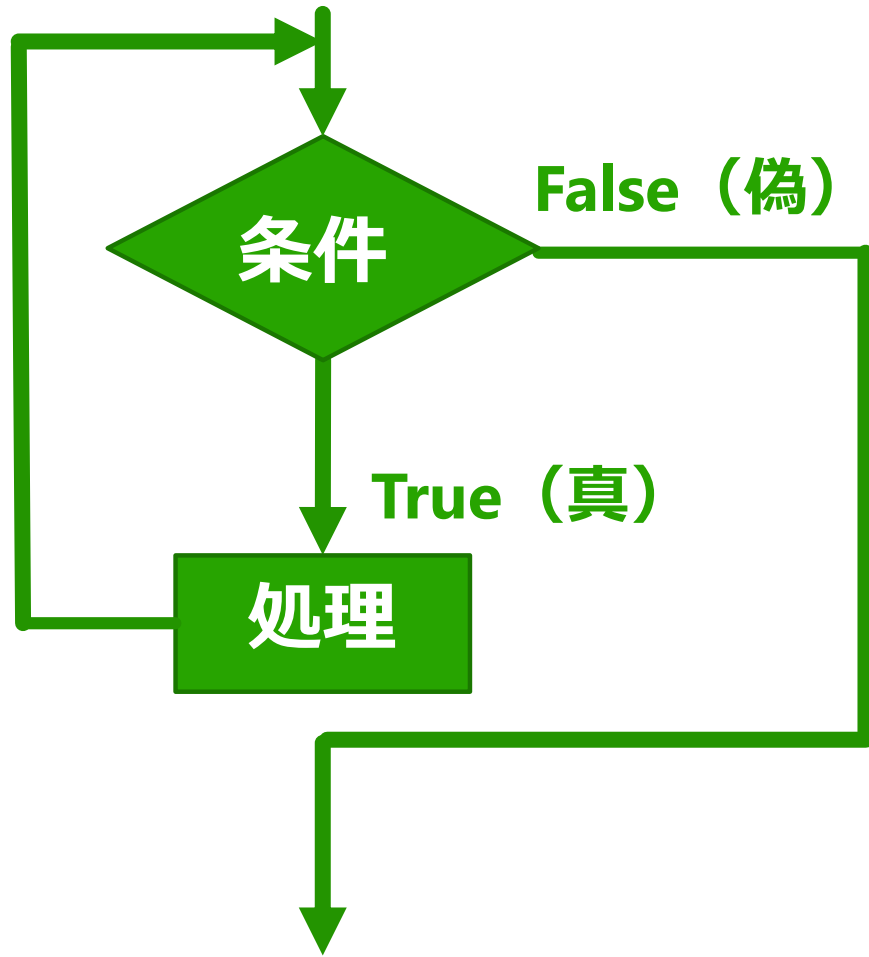
```
orange 150
```



# 3. while文

# while文

while文：繰り返し処理を行うときに使用

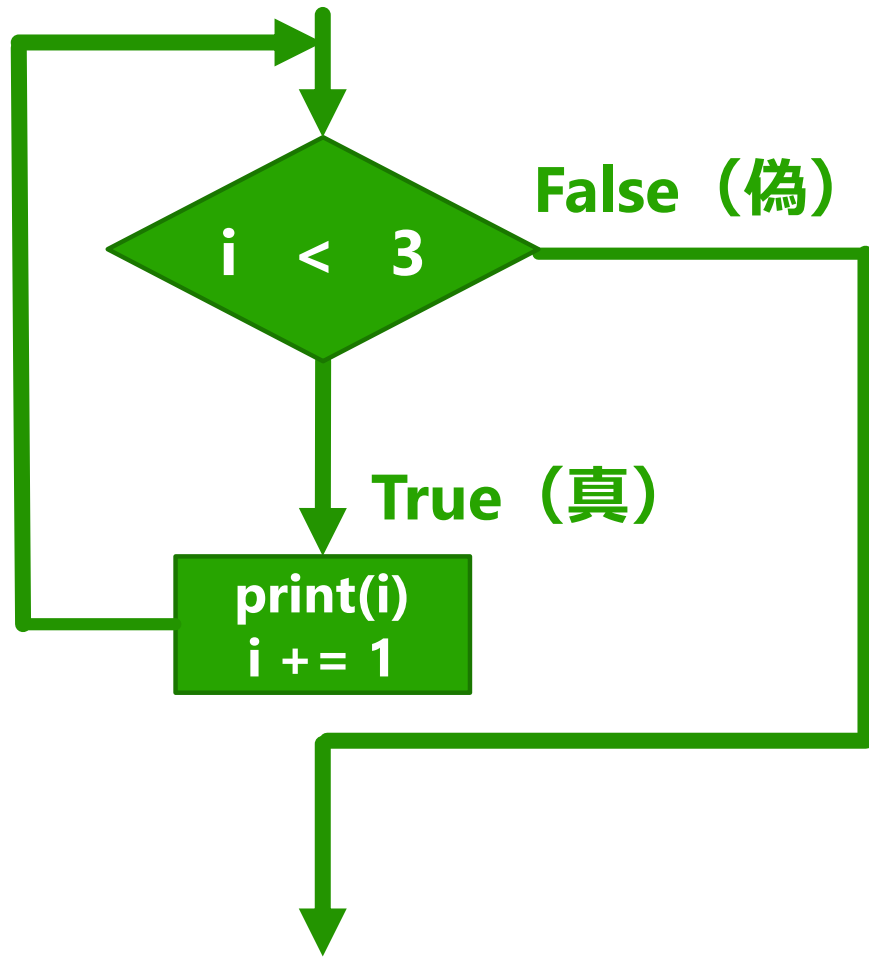


左図をプログラムで表すと下記

```
while 条件:  
    処理
```

# while文 例

例)  $i < 3$ の間、 $i$ をprintする ※  $i$ の初期値は $i=0$ とする



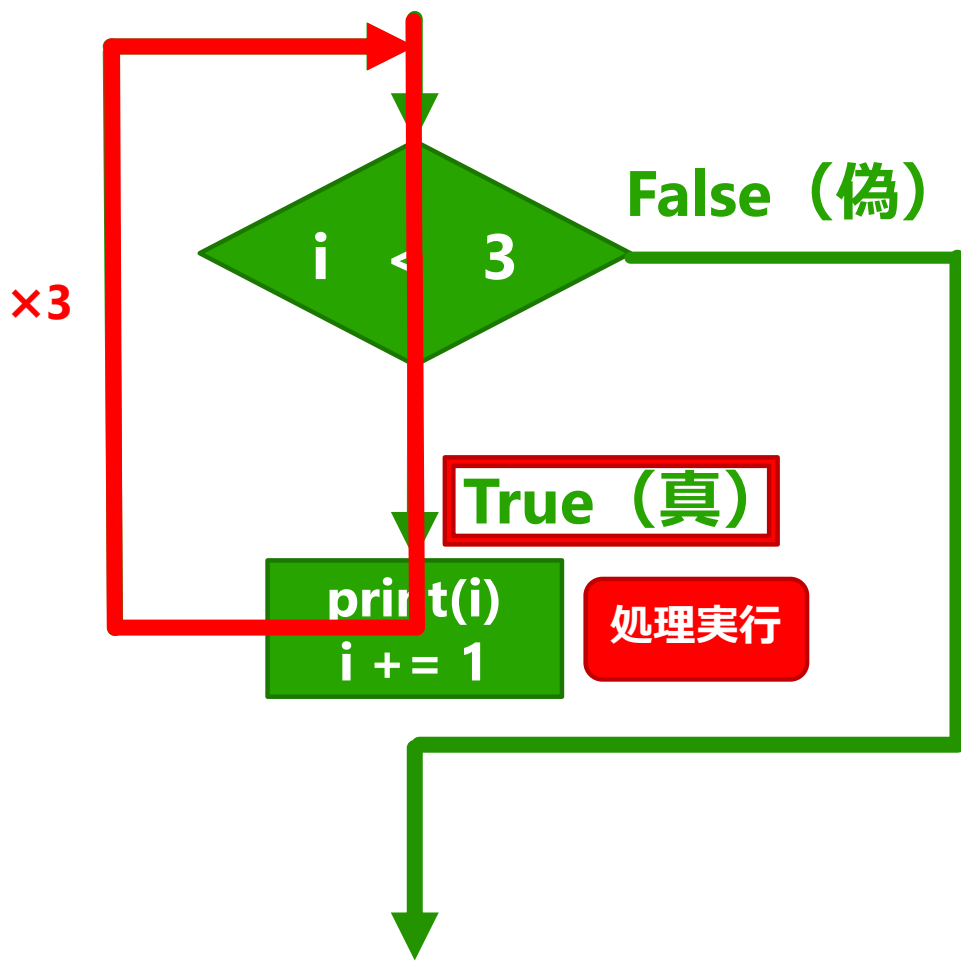
$i = 0$

左図をプログラムで表すと下記

```
while i < 3:  
    print(i)  
    i += 1
```

# while文 例

①while文1～3週目 (i < 3の間、iをprintする)

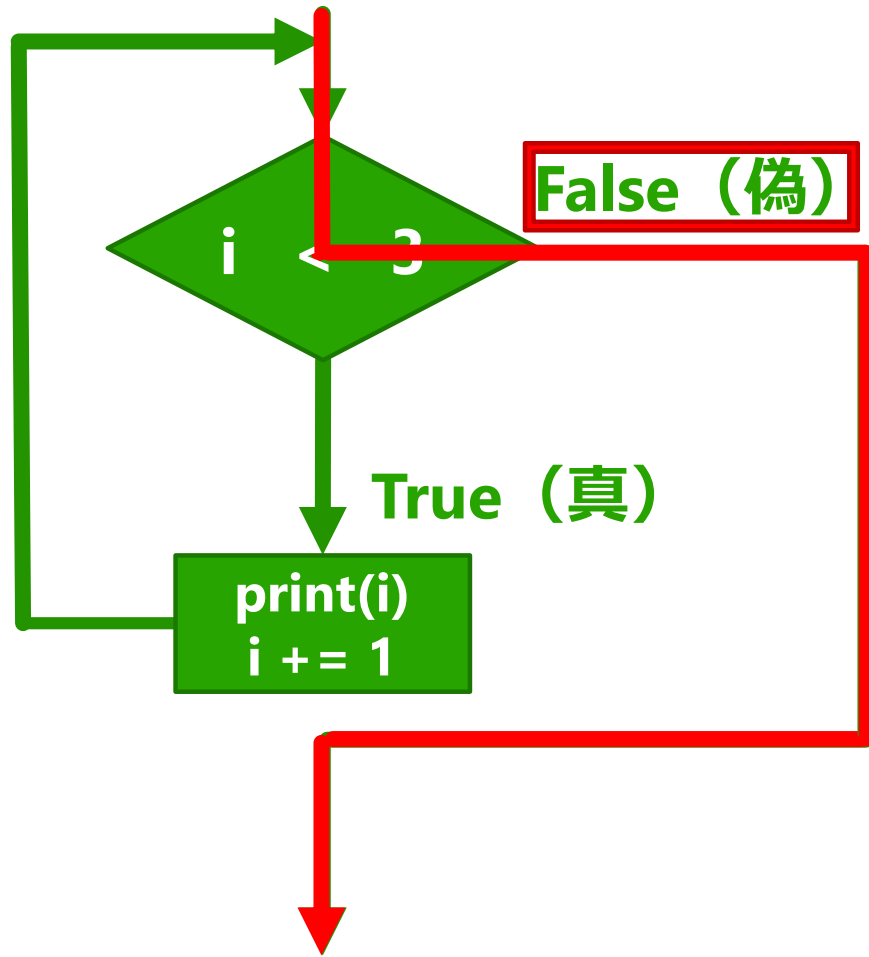


## 出力結果

0  
1  
2

# while文 例

②while文から抜ける (i < 3の間、iをprintする)



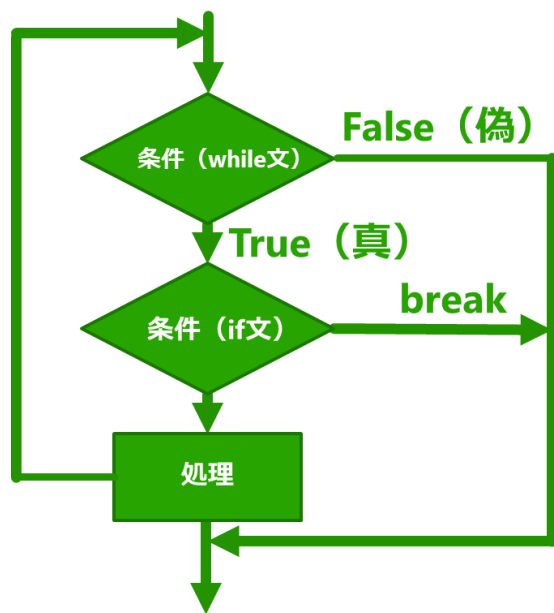
## 出力結果

0  
1  
2

# while文 : break, continue

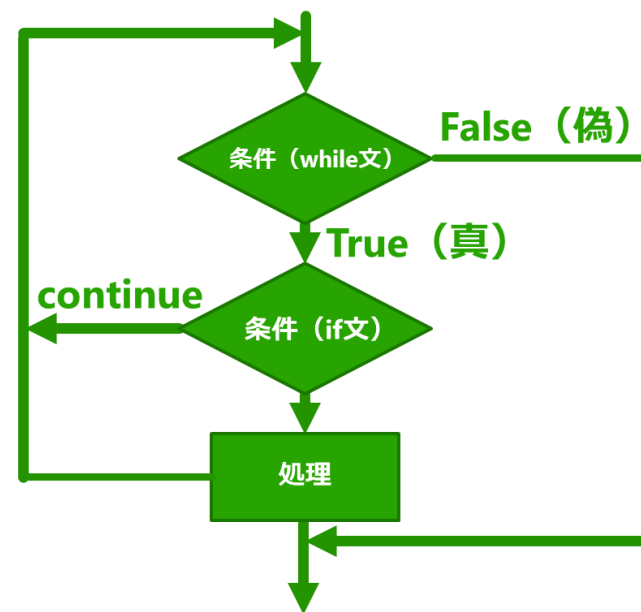
for文と考え方は同じため、例は省略する

## break



```
while 条件:  
    if 条件:  
        break  
    処理
```

## continue



```
while 条件:  
    if 条件:  
        continue  
    処理
```

# while文

## while文・for文の違い

for文：指定した回数のループが完了すればループを抜ける

while文：条件を指定して、その条件がFalseになればループを抜ける

while文↔for文は基本的には変換ができる

※ただし、やりたいことによって使い分けたほうがスマートなコードが書ける

4. try文



# 構文エラーと例外

構文エラー：文法的な違いによって起こるエラー（Syntax Errorともいう）

例）下記のような間違った文法を記述

```
print "hello"      #正しくはprint("hello")
```

実行すると下記のようなエラーが出る

```
File "<stdin>", line 1
  print "hello world"
    ^
SyntaxError: Missing parentheses in call to 'print'
```

構文エラーは、基本的なルールに反していることによって起こるエラーのため、正しい文法に書き直す以外の解決方法はない

# 構文エラーと例外

例外：正しい文法で書いているにもかかわらず発生するエラー（Exceptionともいう）

例）下記のようなコードを記述

```
x = 1  
y = 0  
z = x / y
```

実行すると下記のようなエラーが出る

```
Traceback (most recent call last):  
  File "Main.py", line 3, in <module>  
    z = x/y  
ZeroDivisionError: division by zero
```

上記は、pythonの文法的には正しいが【0で割ってはいけない（ゼロ除算）】という数学的なルールに反している。

# 構文エラーと例外

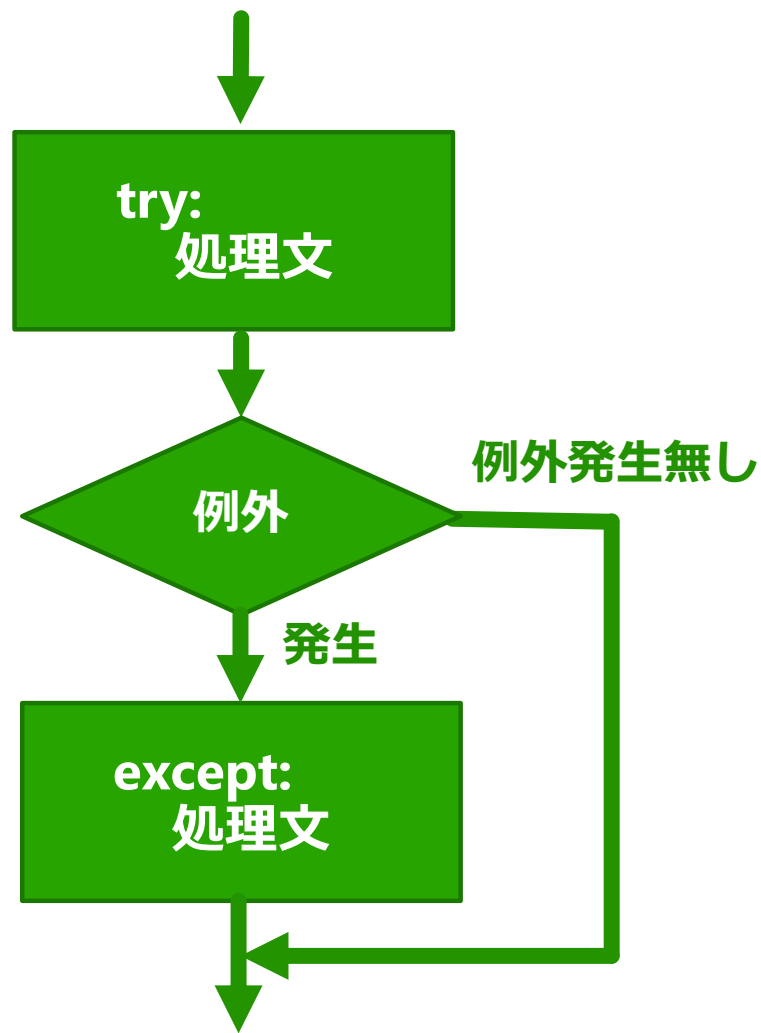
前ページで示したZeroDivisionError（ゼロ除算による）エラーの他にも代表的なエラーには下記がある

- NameError：定義していない変数を使ったときに起きるエラー
- AttributeError：存在しない属性にアクセスしようとしたときに起きるエラー

⇒これらのような特定のエラーが起こる可能性があるときに使用するのがtry文

# try文

**特定の例外**が発生した時に指定の処理を行う（例外が発生する可能性があるが実行したい時）

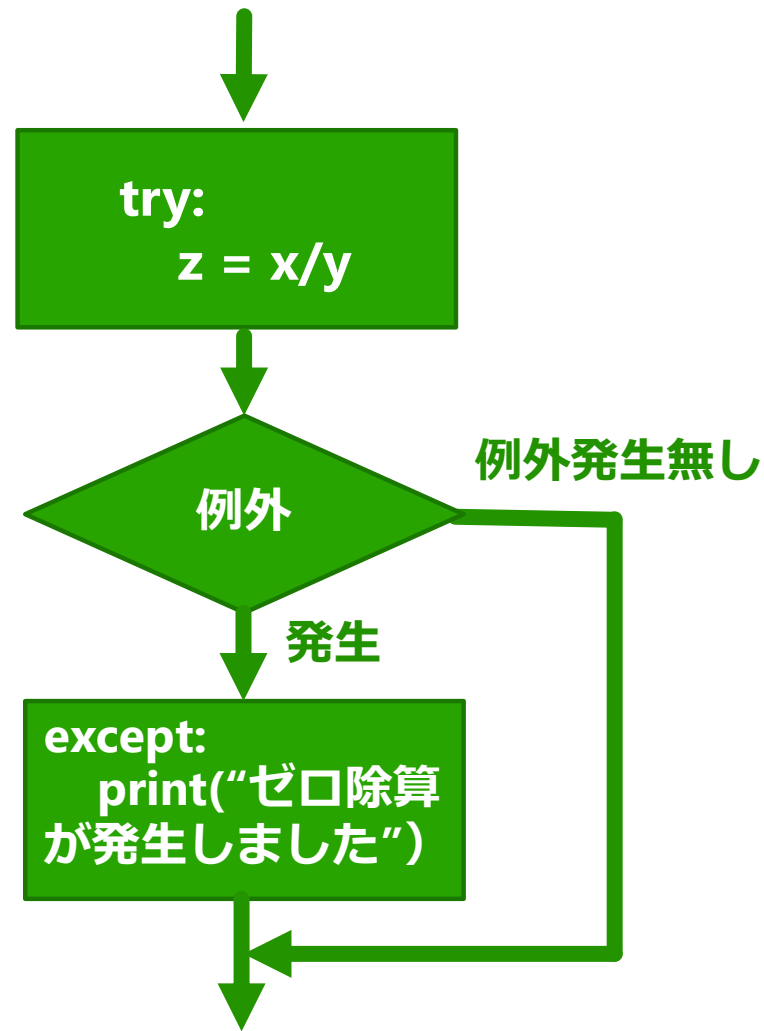


左図をプログラムで表すと下記

```
try:
    処理文
except:
    処理文（例外が起こった時に行う処理）
```

# try文 例

例) ZeroDivisionError (ゼロ除算エラー) が起こる可能性がある式をtry文に記述

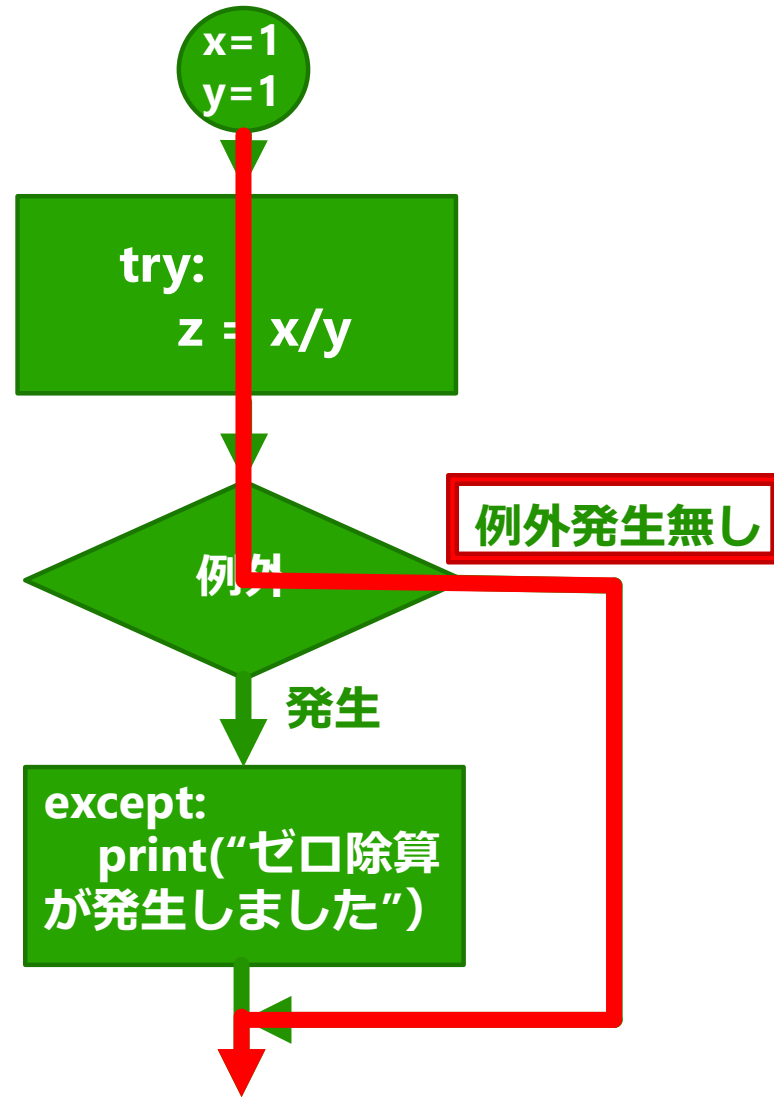


左図をプログラムで表すと下記

```
try:
    z = x/y
except:
    print("ゼロ除算が発生しました")
```

# *try*文 例

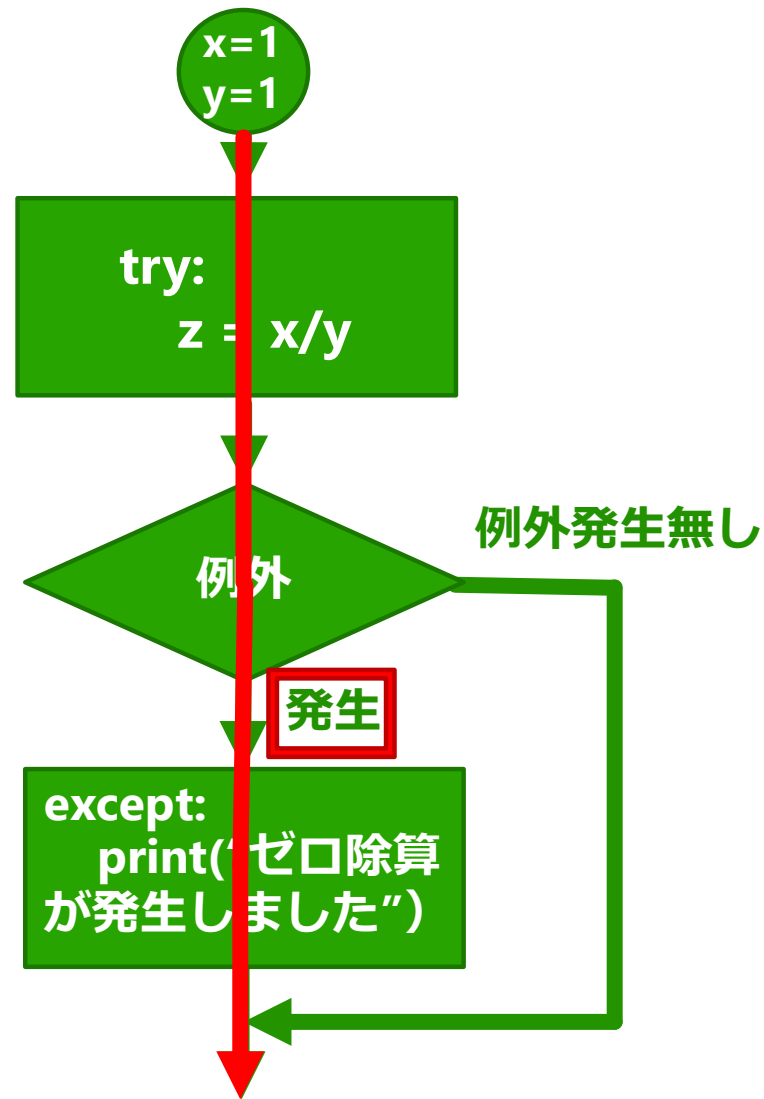
1)  $x = 1, y = 1$



出力結果 ※出力されない

# try文 例

1)  $x = 1, y = 0$

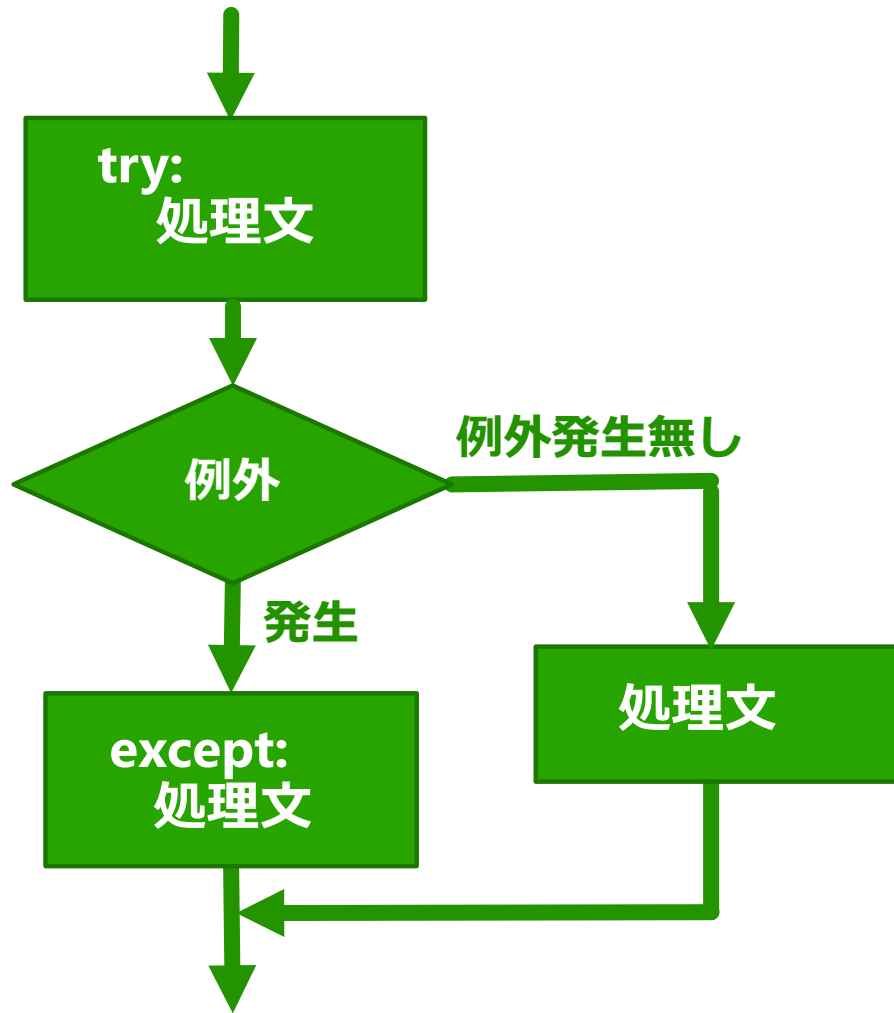


## 出力結果

ゼロ除算が発生しました

# try文 else

elseを使用すると、例外が発生しなかったときの処理も記述できる



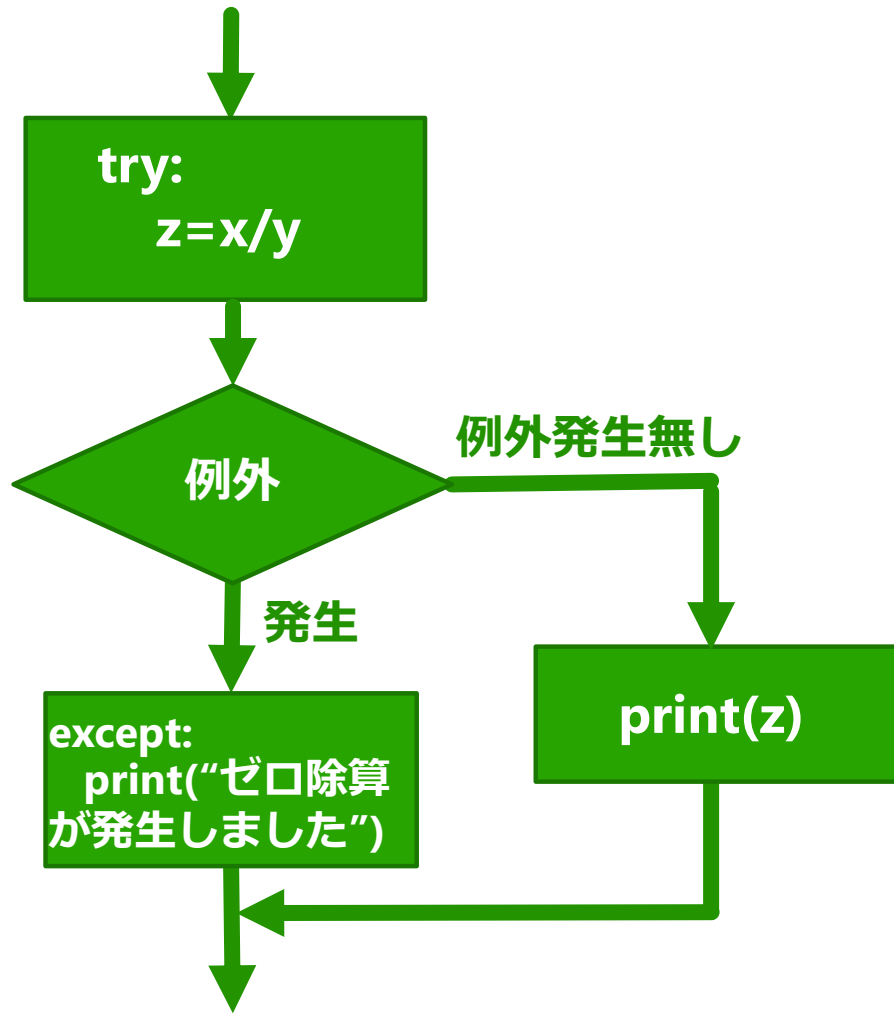
左図をプログラムで表すと下記

```
try:
    処理文
except:
    処理文 (例外が起こった時に行う処理)
else:
    処理文 (例外が起こらなかった時に行う処理)
```



# try文 else 例

例) ZeroDivisionError (ゼロ除算エラー) が起こる可能性がある式をtry文に記述

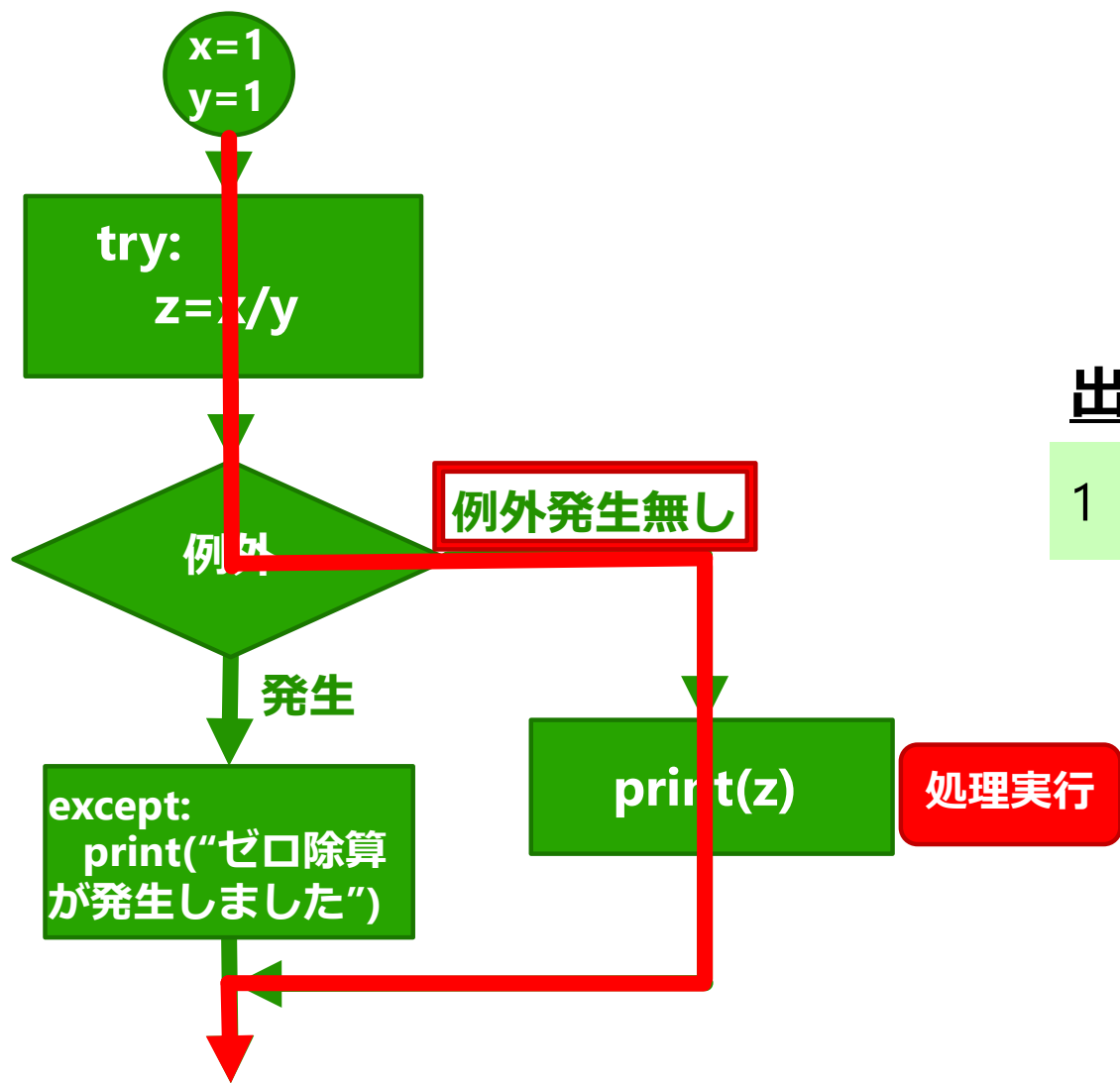


左図をプログラムで表すと下記

```
try:  
    z = x/y  
except:  
    print("ゼロ除算が発生しました")  
else:  
    print(z)
```

# *try*文 *else* 例

x=1, y=1

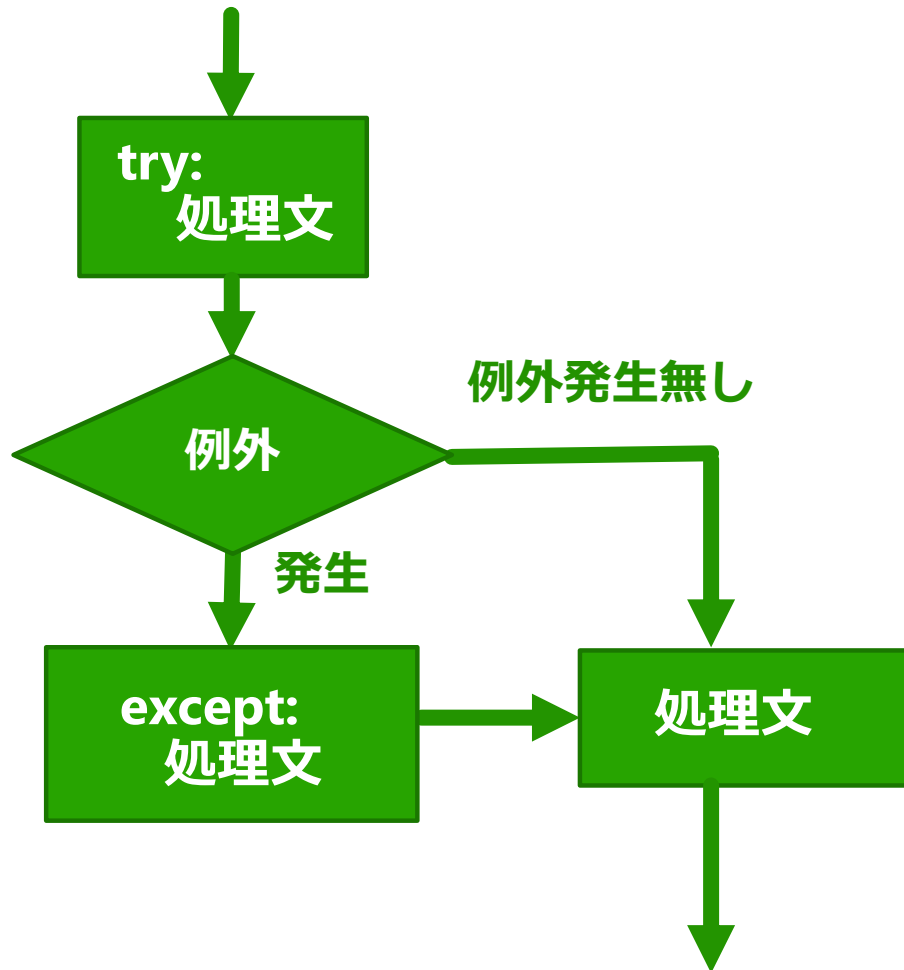


## 出力結果

1

# *try*文 *finally*

finally : 例外が起こる、起こらないにかかわらず実行される

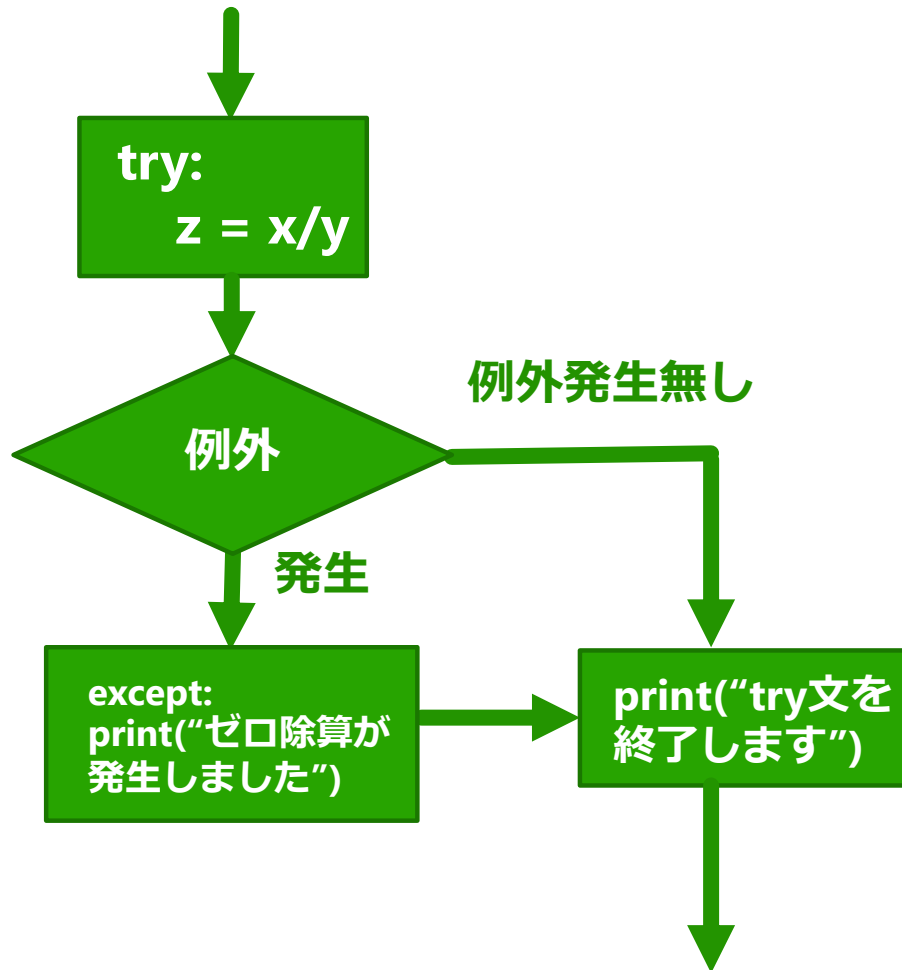


左図をプログラムで表すと下記

```
try:
    処理文
except:
    処理文 (例外が起こった時に行う処理)
finally:
    処理文
```

# *try*文 *finally* 例

例) ZeroDivisionError (ゼロ除算エラー) が起こる可能性がある式をtry文に記述

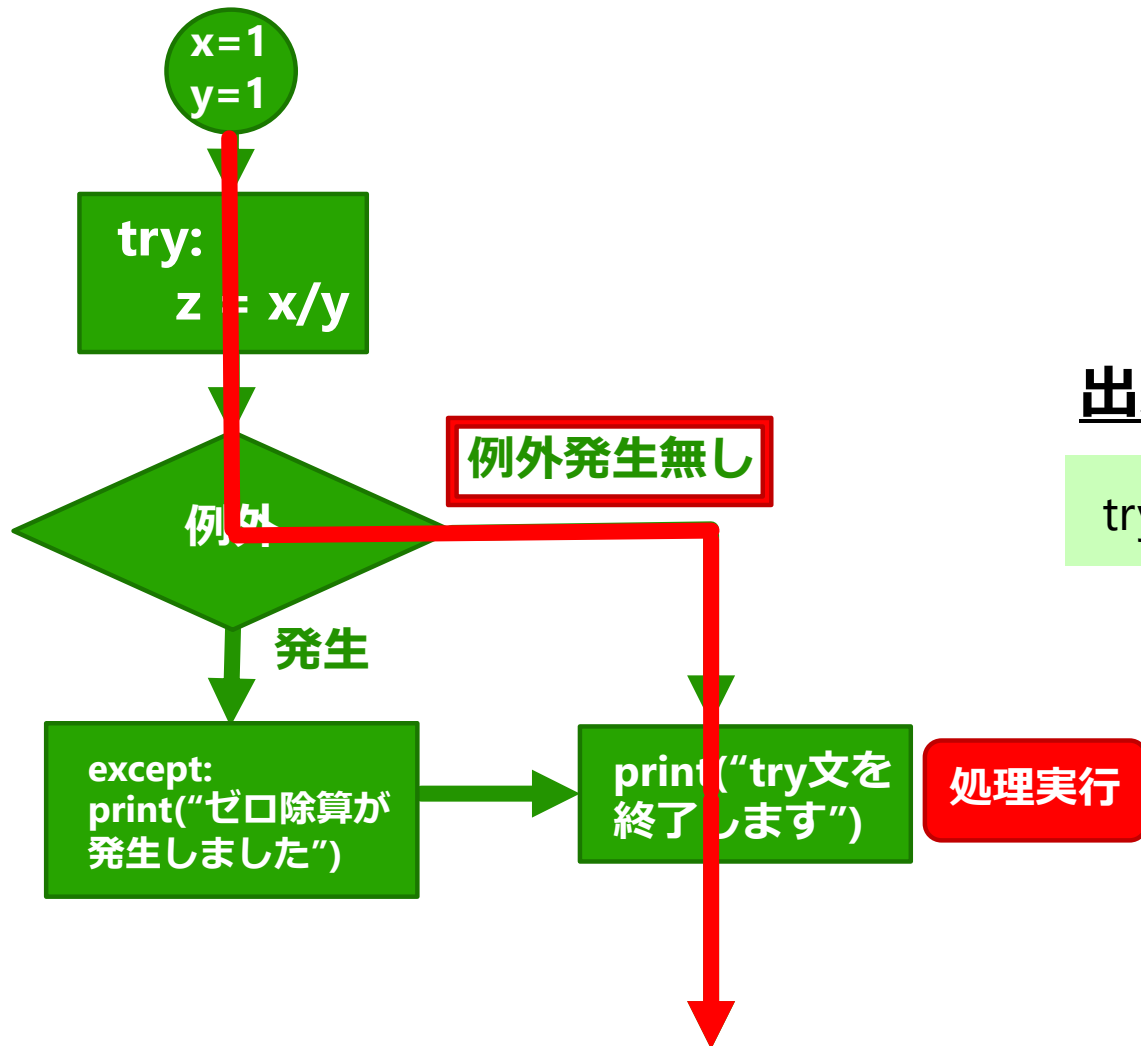


左図をプログラムで表すと下記

```
try:  
    z = x/y  
except:  
    print("ゼロ除算が発生しました")  
finally:  
    ("try文を終了します")
```

# *try*文 *finally*

1)  $x=1, y=1$

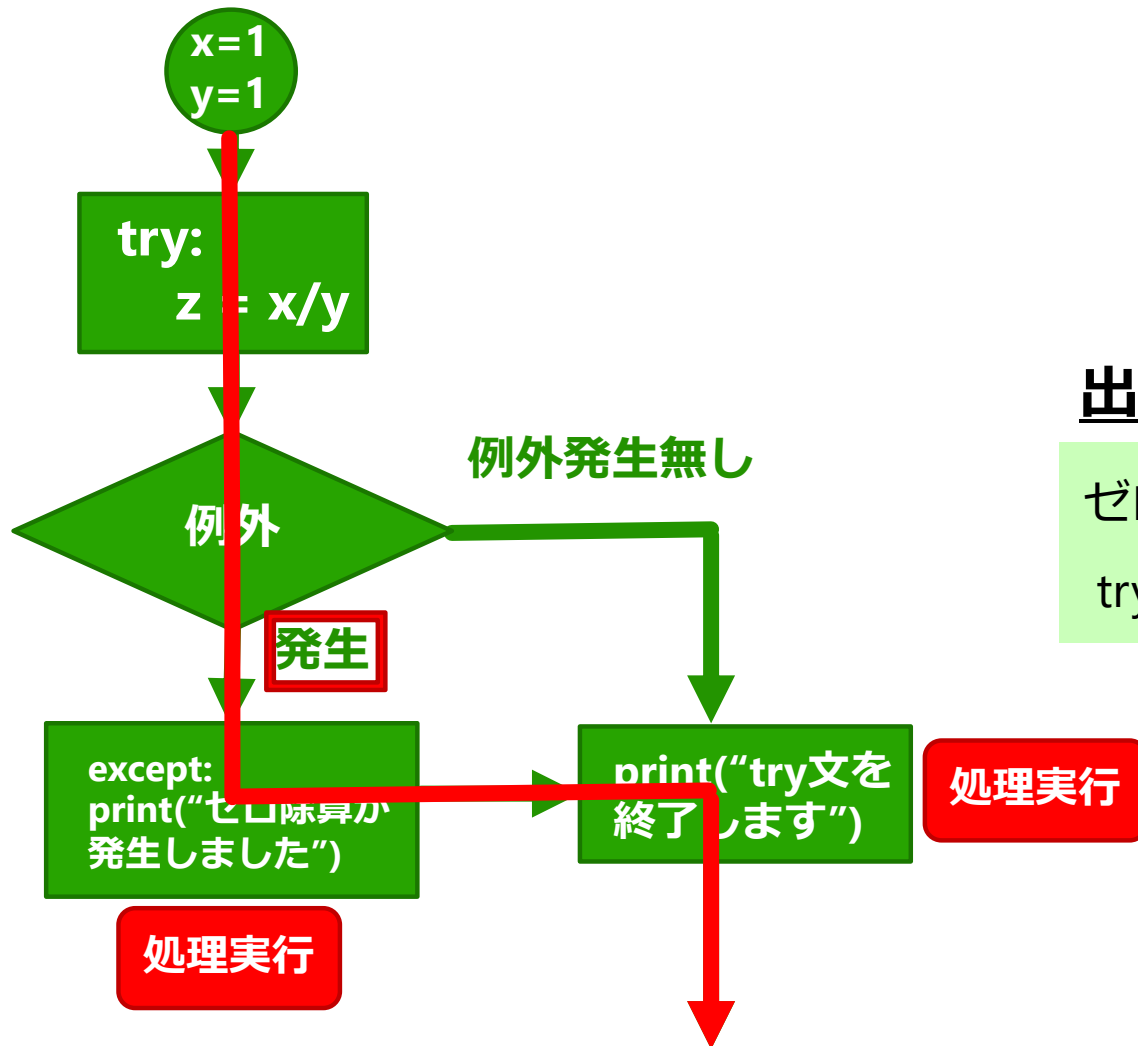


## 出力結果

try文を終了します

# try文 finally

1) x=1, y=1

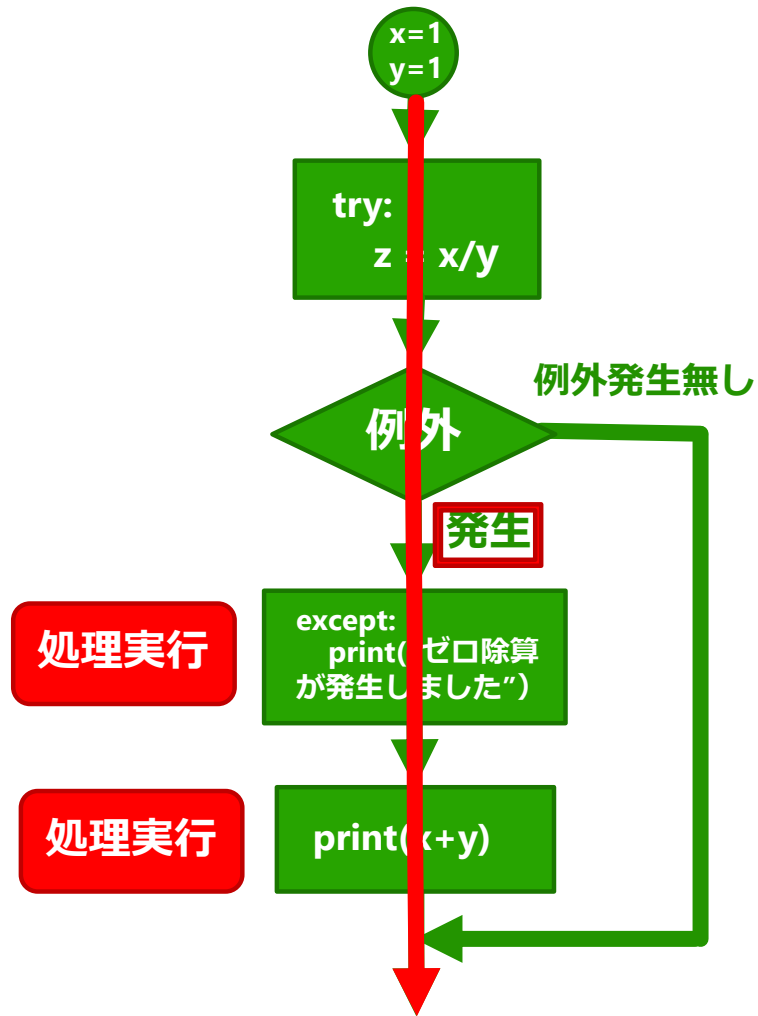


## 出力結果

ゼロ除算が発生しました  
try文を終了します

# try文の利点

- 通常、エラーが発生するとプログラムが強制終了されるが、try文を使用すると、プログラムが継続して実行される。



## 出力結果

ゼロ除算が発生しました  
2

**例外が起こった後も処理が続行**

# 課題

---

learn\_python内のディレクトリkadai内のディレクトリday2に提出  
ファイル名は 2019\_10\_30\_[自分の名前].py とする