目次へ

yieldのちょっとした理解

pythonのyieldは今までの常識を少なからず揺るがす（かもしれない）ものなので、いつ紹介しようかと悩んでいます。ないならないで、多分どんなこともyield抜きで書けるんでしょうが、筆者が個人的にyield大好きなわけですよ。サンプル書くにもyieldで楽したいわけですよ。そういうことなので、軽ーく説明しますので、わかんなくてもこの際いいや、くらいの構えでどうぞ。今回は練習問題も特にしません。

yieldはジェネレータなり、とか、コルーチンなり、とか、マイクロスレッドなり、とか、はたまた「継続」なり、とか色々な言い方をする人がいますけど、別に用語に構わなくてもいいです。これから挙げるような書き方ができる面白い機能、程度に理解してくれてもいいです。

実用的なスクリプトをつくりたいんだという観点のみからいえば、yieldは「ループに値を提供する」機能と限ってとらえてもいいと思います。下、コードの例。

```
def tissue_box():
  # ループに値を提供しているのはこちら
  yield 'sheet'
  yield 'sheet'
  yield 'last sheet'

# 提供された値を"消費"しているのはこちら
for i in tissue_box():
  print i
```

```
実行結果：
```

```
sheet
sheet
last sheet
```

まずはこのサンプルスクリプトと実行結果を眺めて、これが何を意味するのかを考えることにします。

tissue_box (ティッシュ箱?)という関数の中で、ここでは文字列を三回yieldしているようだ、と読みとれますね。（yieldって、英語としては「産出する」「（道などを）譲る」という意味がありますが、前者がここでは正しいイメージかな。実は後者みたいな側面も持ってるんですけど）

関数は通常、値を呼び出しもとに返してあげるためには return を使います。すると、今実行している行で関数は動作を終了し、「関数を呼び出した側」の実行がまた再開されます。改めて意識するまでもなく、あたりまえのことでしたね。yieldは、値を呼び出し元に返すという点で、まずはこれに似ています。

でも、yield を使って値を返す、というときは、呼び出し元に一回ならず「何回も」値を返すことになります。この例では、tissue_box関数は、'sheet' 'sheet' 'last sheet' と合計三回、値を返しました。returnを使ってこんなノリで三回値を返すことはできません。もしやってみたとしても、最初のreturnで関数の実行は終わりますから、結局一回（最初の'sheet'）しか値を返せません。'return'というだけあって、呼び出し元に「帰っちゃう」わけです。

いや、returnで値を三個返すことはできるぞ。yieldなんか持ち出さずに、リストをreturnすればいいじゃないか、といわれそうですね。例えば下のように。

```
def another_tissue_box():
  return ['sheet', 'sheet', 'last sheet']

for i in another_tissue_box():
  print i
```

これもさっきと同じ実行結果を返します。でもこれは、やっぱり、リストを「一回だけ」返しているに過ぎません。同じことができるんなら構わないじゃないか、ともいえますが、返す値が例えば1000000個だったら？　1000000の値を持ったリストを一旦作って、そいつを次の仕事に引き継ぐというスクリプトになってしまいます。これでは無駄にメモリに厳しいです。

例え話にしてみようと思います。ティッシュ箱にティッシュが詰まっていて、取り出し口には一枚分の切れ端が飛び出ています。その一枚を引っ張り出すと次の紙が改めて飛び出します。これがティッシュ箱の便利なところですね。コレに対して、切れ端を引っ張り出したら、ティッシュ箱の中にある百何十枚の紙が束になってごっそり出てきてしまったらどうでしょう。結局全部使うのなら用事としては足りるとはいうものの、ちょっと困りますね。リストがいきなりreturnされるというのは、この「ティッシュ束」のイメージです。yieldをこのティッシュ箱のイメージの助けで理解してみましょう。つまり、「forループが値を使うために、「ティッシュ一枚ずつ」提供するのがyield」です。最初の一枚（さっきの例では'sheet'）を使うときは、まだ残りの二回分のyieldは実行さえされていない状態です。次の一枚を使うときに、はじめて二回目のyieldが実行されます。返す値が全部で1000000個だったとしても、1000000個のリストを作ったりはしません。最初の1個を返すときは、まだ残りの999999個の値はどこにも発生していませんから、少なくともメモリがいっぱいになってしまうということが避けられそうです。

yieldするほうの関数は、とにかく複数回のyieldをするということさえ達成できれば、どんな書き方をしてあってもよいです。一方、そいつを「消費」するforループのほうは、ティッシュ箱の中でどんなメカニズムが動いていようとちっとも構いません。ティッシュ一枚引っ張り出したら次のものが出てくればよい。ティッシュが出てこなくなったら（つまり関数の実行が終わったら）それでループを終わる目印にするから、それが分かればよい。ループ側（つまり消費側）にとって、yieldはその程度に仕事を任されているんです。よしなに頼む、といったところです。

だから下のようなコードも正当です。

```
def strange_box():
  for i in xrange(10):
    yield i
  yield 99
  for i in xrange(10):
    yield i+10

for i in strange_box():
  print i,
```

```
実行結果：
0 1 2 3 4 5 6 7 8 9 99 10 11 12 13 14 15 16 17 18 19
```

yieldの値を使うループのほうは、strange_box関数を実行した結果帰ってくる「ティッシュ箱」から何も考えずにズバズバ引き抜いているだけですが、yieldする側のほう（strange_box）は内部でループを二回行ったり、その途中で脈絡なく99なんて値を混ぜ込んでみたり、実に奇妙なやりくりをしています。これでも一向に問題ないのです。これを使うループのほうは、とにかく順番に何かが出てくれば満足です。

一応ここまで説明してみましたが、なんとなく使い方の見当がつきましたでしょうか。別に「常識を揺るがす」というものでもなかったなあ、とお考えになるでしょうか。それならそれで筆者は安心です。

でも、せっかくなので常識を揺るがされるという気持ちになってみたかったら、下のコードの実行例を見てみてください。

```
def tissue_box():
  for i in xrange(10):
    print "yield %d !" % i
    yield i

for i in tissue_box():
  print "consume %d !" % i
```

```
実行結果：
yield 0 !
consume 0 !
yield 1 !
consume 1 !
yield 2 !
consume 2 !
yield 3 !
consume 3 !
yield 4 !
consume 4 !
yield 5 !
consume 5 !
yield 6 !
consume 6 !
yield 7 !
consume 7 !
yield 8 !
consume 8 !
```

```
yield 9 !
consume 9 !
```

それぞれのコードがいつ実行されているのかを確かめるために、要所にprintを置いてみました。これを見ると、yieldするほうのループを、それを消費しようとするループが交互に実行経過を報告しています。まるで「同時に動いている」かのようですね。ここらへんが、不思議なところです。興味があったら、どの行がどういう順番で実行されたのかを目で追いかけてみるといいかもしれません。とはいえ、今のところは、「便利なんだからこの点は気にしなくても可」とあえて言っておきましょう。

ふたりの作業者が、値を「バケツリレー」しているようだと思ってもらってもよさそうですね。上流の作業者がyieldするそばから、次の作業者によってそれが利用されていると。

- 興味のある方向けに、yieldの動作の秘密をもうすこし詳しく説明してみます。中でyieldが実行された関数は、「ふたつめのプログラム」のようにふるまい始めるという所がミソです。たとえば関数の中で合計三回yieldするものがあったとき、一回目のyieldが実行されると、呼び出し元（メインルーチンと呼びましょう）に値をひとつ渡すとともに、自分自身の動作はいったん休止します。休止はしても、関数の実行を終わるわけではなく、実行が再開できるチャンスまでじっと待っているという状態です。で、メインルーチン側は「次の値がほしいな」という合図をよこしてくれますから、そのときにまた関数の実行が「再開」されて、次のyieldにたどり着くまで実行され、メインルーチンに次の値を渡します。メインルーチンと関数が、たがいに実行順序を「譲り合いながら」じゅんばんこに実行されるってイメージですね。で、関数の実行が終わったら、そこで「ふたつめのプログラム」は消滅して、その後はメインルーチンだけが実行を続けるのです。

使用例

このyieldが一体どんなときに便利なんだい、と言われそうです。結構な思考のジャンプが必要な話ですし、これが本当に役に立つんだと実感できないと、使ってみたいとは思えないでしょうね。

筆者だったらこんなときにyieldを使うだろうな、と想像する例を挙げてみます。

たとえばこんなテキストデータを処理する必要があるとして...

receipt.txt

```
*yamamoto
100
200
380
98
*yamada
2800
68
32000
*hara
210
105
...略...
```

「*」で始まるのが人の名前、その後につづくのがその人が使ったお金、くらいにとらえましょう。ここでは yamamoto さんが 100 + 200 + 380 + 98 = 778円使ったんだ、ということを意味します。これで、個人ごとの使用金額一覧を利用して何かしたいんだとしましょう。

そしたら、まず筆者はこんなスクリプトから書き始めるでしょう。

```
for line in open("receipt.txt"):
  data = line[:-1]
  if data[0] == '*':
    # 何かする…
  else:
    # 何かする…
```

データを一行ずつ読み込んで、行頭に「*」があるときとないときで別々の処理をするだろうな、ということをまずは表現しました。

で、現在の名前と小計をとっておく変数を準備して、それを操作する処理も書き足します。

```
current_name = ''
temp_sum = 0
for line in open("receipt.txt"):
  data = line[:-1]
  if data[0] == '*':
    if current_name != '':
      print current_name, temp_sum  #<---- とりあえず表示してみるか
    temp_sum = 0   # で、小計をリセット
    current_name = data[1:]  # 名前も覚えなおし。先頭の「*」は取った形で
  else:
    temp_sum += int(data)
print current_name, temp_sum  #<---- 残ったデータも表示して終了
```

たぶんこのスクリプトの実行結果はこんな風になるでしょうね。

```
yamamoto 778
yamada 34868
hara ...略...
...
```

で、ここまで書いたあとで、「じゃあこれをHTMLにしようかなー」と思い立ったとします。

今書いたコードに色々と書き足せばそれも実現できますが、yieldを使うと、ちょっとスッキリした書き方でそれが実現できます。まず、今まで書いたものを全部まとめて関数に仕立てて…

```
def sum_tissue_box():
  current_name = ''
  temp_sum = 0
  for line in open("receipt.txt"):
    data = line[:-1]
    if data[0] == '*':
      #print current_name, temp_sum
      yield [current_name, temp_sum]  #<---- printじゃなくてyieldに。誰が使うかはまだ知らぬ
      temp_sum = 0
      current_name = data[1:]
    else:
      temp_sum += int(data)
  yield [current_name, temp_sum]  #<---- ここもyieldに
```

で、printだった部分をyieldに書き変えました。

これだけでは動作しません。関数を定義しただけですからね。これを「消費」するコードも一緒に書けば、こうなります。

```
def sum_tissue_box():
  current_name = ''
  temp_sum = 0
  for line in open("receipt.txt"):
    data = line[:-1]
    if data[0] == '*':
      #print current_name, temp_sum
      yield [current_name, temp_sum]  #<---- printじゃなくてyieldに。誰が使うかはまだ知らぬ
      temp_sum = 0
      current_name = data[1:]
    else:
      temp_sum += int(data)
  yield [current_name, temp_sum]  #<---- ここもyieldに

#
# yieldを消費するループ
#
print "<table>"
for a in sum_tissue_box():
  print "<tr>"
  print "<td>%s</td><td>%d</td>" % (a[0], a[1])
  print "</tr>"
print "</table>"
```

yieldを消費するほうは、sum_tissue_box関数がテキストファイルを読み込んで色々苦心しているのを尻目に、ループで a という値（ここではひとつづつがリスト）が入ってくるものと前提して、こいつを使ってHTMLを組み立てればよくなりました。ちょっとHTML自体は手抜き気味ですが、手軽にやっている様子は分かっていただけるのではないでしょうか。

sum_tissue_box関数は、「オレはここまでデータ処理をして成果をこしらえたんだから、これを使って何かするのは誰かがやってくれや」と言っているかのようです。それを使うほうは、「細かいことは知らんが、ある程度整ったデータがsum_tissue_boxから流れてくるようだから、こいつを使って何かするか」と言っているかのようです。「分業」が成り立っています。

こんな手順のスクリプト開発ができるようになるというのが、筆者の理解する「yieldが便利な利用シーン」です。今後もたまに使ってみせますのでよろしくどうぞ。

あ、このテの関数に tissue_box とかつけるのは、別に決まりではないですよ。たまたまティッシュ箱の例えを引っ張っているだけです。