プログラミング初級（Python）

反復処理

早稲田大学グローバルエデュケーションセンター

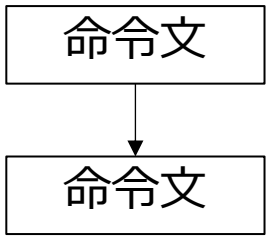
while文とfor文によるループ

賢明にそしてゆっくりと。速く走る者こそつまずく。

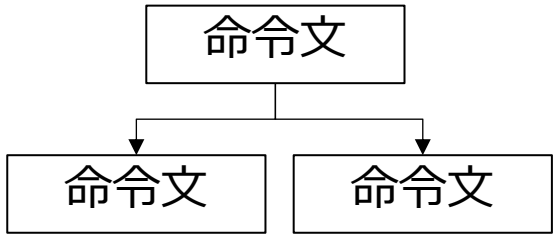
— ウィリアム・シェイクスピア
（「ロミオとジュリエット」より）

プログラムの処理形態

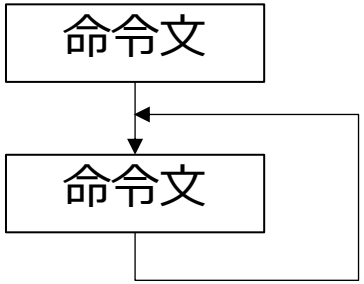
- 順次処理や分岐処理のみのプログラムでは命令の実行は基本的に１回。
- 順次処理や分岐処理のプログラムでは、命令が上から下に実行される。
- プログラムでは**同様の命令を2回以上連続的に実行**させることができる。
- これを**反復処理**と呼ぶ。**繰り返し**処理、**ループ**処理などとも呼ばれる。
- 反復処理ではプログラムの命令が上から下あるいは下から上に移動して実行されているように見える。ループと呼ばれるのはこのことから。
- これにて**３役揃い踏み**：順次処理、分岐処理、反復処理の３つを組み合わせることで、**様々なプログラムを作成できる**。



順次処理



分岐処理

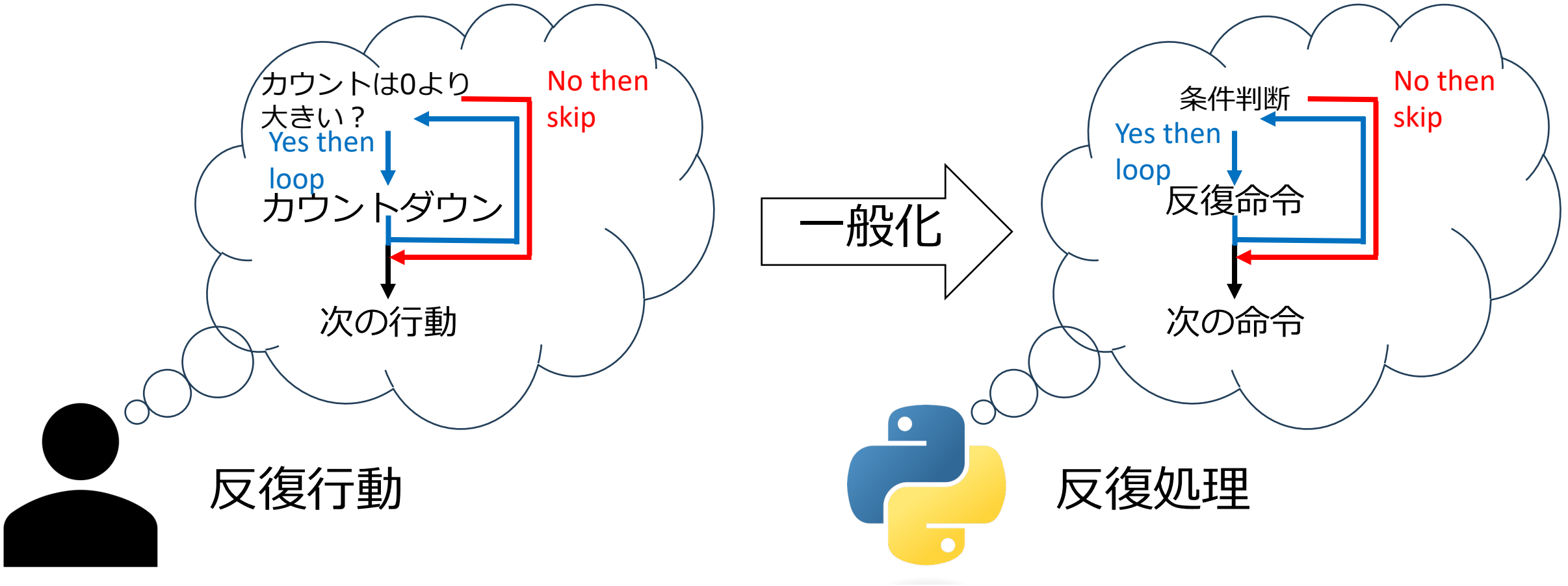


反復処理

反復処理の概念イメージ

例：カウントダウン

条件判断と繰り返しの実行



対話型インタプリタでプログラムを書く場合

```
1  >>> # Below is a loop processing program using the while statement.
2  >>> count = 5
3  >>> while count > 0:
   ...     print(count)
   ...     count -= 1
   ...
   5
   4
   3
   2
   1
4  >>> print("Hello, world!")
   Hello, world!
5  >>> # Below is a loop processing program using the for statement.
6  >>> for countdown in 5, 4, 3, 2, 1:
   ...     print(countdown)
   ...
   5
   4
   3
   2
   1
7  >>> print("Hello, world!")
   Hello, world!
```

テキストエディタ等でプログラムを書く場合

テキストエディタ等でPythonプログラム(countdownWhile.py)を作成

```
1 # Below is a loop processing program using the while statement.
2 count = 5
3 while count > 0:
4     print(count)
5     count -= 1
6 print("Hello, world!")
```

```
5
4
3
2
1
Hello, world!
```

テキストエディタ等でPythonプログラム(countdownFor.py)を作成

```
1 # Below is a loop processing program using the for statement.
2 for countdown in 5, 4, 3, 2, 1:
3     print(countdown)
4 print("Hello, world!")
```

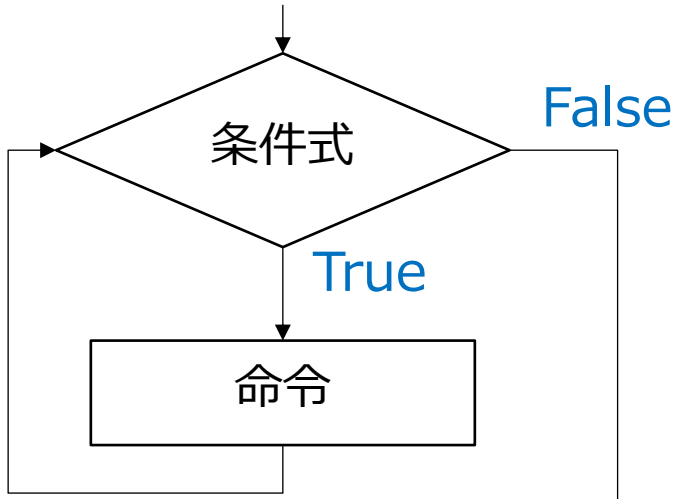
```
5
4
3
2
1
Hello, world!
```

while文

- **while文**とは条件が真となる場合にループ内の命令を反復実行し、偽となる場合にはループから抜けて命令の反復処理を止める**構文**。

構文

```
while 条件:
    命令
```



while文のフローチャート

- 条件がTrueの場合、反復処理を開始する。
- 反復処理としてループ内の命令を実行したら条件式の判定に戻る。
- 条件がFalseの場合、while文のループから抜ける。
- 条件がTrueである限り、反復処理を行う。
- 条件式の例：i ＞ 0
  iが0より大きい場合は条件がTrueとなり反復処理を行う。
  iが0以下の場合は条件がFalseとなり反復処理を停止する。

while文を用いたカウントダウンプログラム

```
>>> # Below is a loop processing program using the while statement.
>>> count = 5
>>> while count > 0:
...     print(count)
...     count -= 1
...
5
4
3
2
1
```

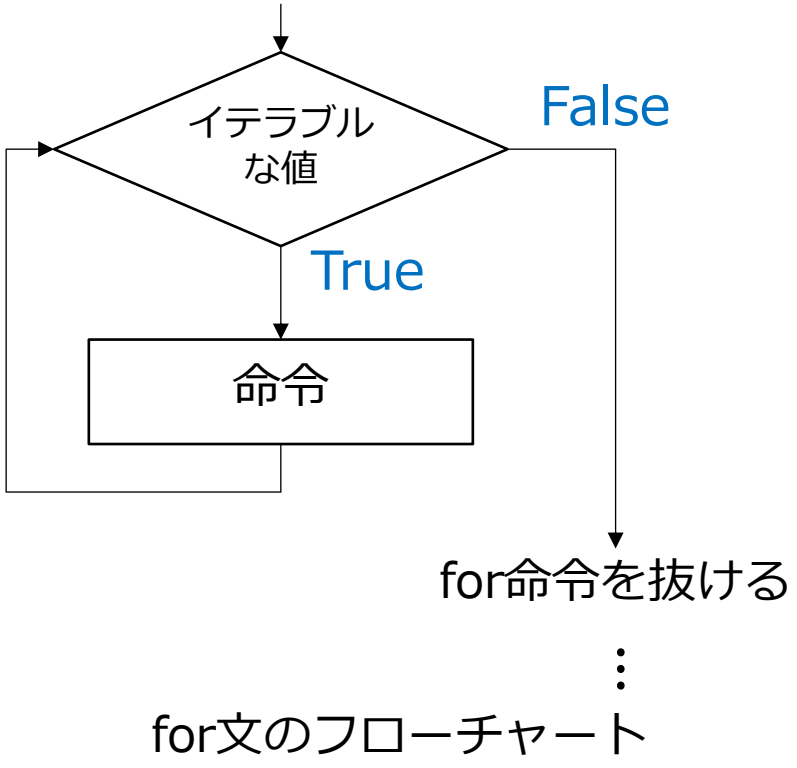
- 変数countに数値5を代入。変数名は任意。while文の前に初期化しておく。
- while文で比較演算を実行し、変数countの値が0より大きい場合反復実行。
- while文のブロック内では、変数countの値をprint関数で表示。
- count -= 1の式で変数countの値を1ずつデクリメントする。
- while文の条件式に戻って再び比較演算を実行する。countの値は4になっているが、0より大きいので再びブロック内の命令を実行する。
- 変数countの値が最後の文で1から0にデクリメントされるまで、反復処理が続く。
- 変数countの値が0になってからwhile文の条件式に戻ると、比較結果がFalseとなるので反復処理は終了する。

for文

- **for文**とは**反復可能なオブジェクト**の値がある場合にループ内の命令を反復実行し、値がない場合はループから抜けて反復処理を止める**構文**。

構文

```
for 要素 in イテラブル:
    ループ命令
```



for文のフローチャート

- 反復可能なオブジェクトを**イテラブル**と呼ぶ。
- イテラブルな値がある場合条件をTrueとして反復処理を開始する。
- 反復処理において命令の実行を終えたらイテラブルな値の判定に戻る。
- イテラブルな値がなくなれば条件をFalseとし、for文のループから抜けて反復処理を止める。
- イテラブルな値がある限りは処理を繰り返す。
- イテラブルな値の例：**数値の羅列**、**文字列**など。シーケンスであればその値はイテラブル。**単一の数値はイテラブルでない**。 例 314159

for文を用いたカウントダウンプログラム

```
>>> # Below is a loop processing program using the for statement.
>>> for countdown in 5, 4, 3, 2, 1:
...     print(countdown)
...
5
4
3
2
1
```

- for文で変数countdonwを宣言
- 変数には反復の都度、自動的にイテラブルな値が代入する。
- 最初にイテラブルな値がある場合、その値を変数に代入して命令を実行。
- for文のブロック内では、変数countdonwの値をprint関数で表示。
- for文の先頭に戻って、次のイテラブルな値が再び変数に代入する。
- countdonwの値は4になっているが、これはシーケンスとして最後の値ではないので再びブロック内の命令を実行する。
- 最後のイテラブルな値を使用するまで、反復処理が続く。
- イテラブルな値がなくなってからfor文の先頭に戻ると、反復処理は終了する。

無限ループ😿

```
>>> # Below is a loop processing program using the while statement.
>>> count = 5
>>> while count > 0:
...     print(count)
...     # count -= 1
...
5
5
5
5
5
⋮
```

実験としてコメントアウト

- while文を抜け出せたのは変数countの値が減少し、いずれ値0に更新されたから。
- while文を抜け出せたのは条件式の値（評価結果）がTrueからFalseになったから。
- 条件式の値がTrueからFalseになったのは、count -= 1の式を反復実行したから。
- ではcount -= 1を書かない、または書き忘れたら？変数countの値は5のまま…？
- 条件式の値はTrueのままなのでwhile文から抜け出せずに反復処理が延々と続く。
- これを**無限ループ**と呼ぶ。
- 反復処理を書く際は、プログラムが無限ループとなっていないか意識すると良い。
- もし無限ループを起こした場合は **[Ctl]キー ＋ 「C」キー**の同時押しで**強制終了**。

while文の条件式の値を直接Falseで更新する例

```
>>> count = 0
>>> is_continue = True
>>> while is_continue == True:
...     count += 1
...     print(f'{count} iteration')
...     value_continue = input('Would you like to end the input? (Y/N): ')
...     if value_continue == 'Y':
...         is_continue = False
...
1 iteration
Would you like to end the input? (Y/N): N
2 iteration
Would you like to end the input? (Y/N): N
3 iteration
Would you like to end the input? (Y/N): Y
>>>
```

最初にTrueで初期化。

このままだと常に条件式の評価はTrue。無限ループの可能性。

キーボードから文字入力。

値をチェックして中身が'Y'なら変数is_continueの値をFalseに更新。

- input関数で`**Y**`の文字が入力された場合は次の条件式の評価がFalseとなるので反復処理を**終了する**。
- input関数で`**Y**`**以外**の文字が入力された場合は次の条件式の評価がTrueとなるので反復処理は**終了しない**。
- このように**反復処理がいつ終了するか不明の場合**に**while文はしばしば用いられる**。

while文でbreak文を用いた例：整数を足し算するプログラム

```
1  >>> total = 0
2  >>> while True:
...      num = int(input('Integer to add [type -999 to quit]: '))
...      if num == -999:
...          break
...      total += num
...      print(total)
...
Integer to add [type q to quit]: 30
30
Integer to add [type q to quit]: -40
-10
Integer to add [type q to quit]: 50
40
Integer to add [type q to quit]: -999
3  >>>
```

条件式にはブーリアンリテラルのTrueを記述。このままでは無限ループとなる。

文字列型の値-999が入力されたらbreak文で反復処理を中断する。

反復処理は中断され、**break文以降の命令**である足し算や結果の表示は**実行されな**い。

break文：反復処理を中断し、ループから抜け出す命令。

while文でbreak文とcontinue文を用いた例：整数を足し算するプログラム（負の整数を除く）

```
1  >>> total = 0
2  >>> while True:
...        num = int(input('Integer to add [type -999 to quit]: '))
...        if num == -999:
...            break
...        if num <= 0:
...            continue
...        total += num
...        print(total)
...
Integer to add [type q to quit]: 30
30
Integer to add [type q to quit]: -40
Integer to add [type q to quit]: 50
80
Integer to add [type q to quit]: -999
3  >>>
```

条件式にはブーリアンリテラルのTrueを記述。このままでは無限ループとなる。

文字列型の値-999が入力されたらbreak文で反復処理を中断する。

負の整数となる文字を入力した場合、**反復処理は継続**しているが、**continue文以降の命令**である 足し算 や 和の表示 は**実行されない**。

30と50の合計80が表示された．-40は計算に含まれていない．

break文によって中断された結果。

continue文：反復処理を中断し、ループの先頭に戻る命令。

while-else文

```
1  >>> word = 'Pneumonoultramicroscopicsilicovolcanoconiosis'
2  >>> offset = 0
3  >>> while offset < len(word):
...      if word[offset] == '/':
...          break
...      print(word[offset])
...      offset += 1
... else:
...      print('Iteration completed successfully')
...
P
n
(略)
s
Iteration completed successfully
4  >>>
```

構文

```
while 条件式:
    ループ命令
else:
    通常条件で反復処理を終えた場合に実行される命令
```

while-else文とは、反復処理の中断がある場合とない場合とで命令パターンを変える**構文**。

- 利用ケース１：反復処理が本当に全て正常に実行されるかどうかを確かめるケース。（break文が実行されないことを前提とした作業。）例 上記のようなプログラム。
- 利用ケース２：反復処理が中断かつ終了した時にのみ**実行しない命令**があるケース。（break文が実行されることを前提とした作業。）例 次ページのようなプログラム。

while-else文でbreak文の実行が前提の例：クイズプログラム

```
>>> correct_answer = 'Python' # Correct answer
>>> max_attempts = 3 # Maximum number of attempts
>>> attempts = 0 # Current attempt count
>>> while attempts < max_attempts:
...     user_answer = input('What is my name?: ')
...     if user_answer == correct_answer:
...         print('Correct!')
...         break
...     else:
...         attempts += 1
...         print('Incorrect. Please try again.')
... else:
...     print(f'The correct answer was "{correct_answer}". Game over.')
...
What is my name?: Perl
Incorrect. Please try again.
What is my name?: Java
Incorrect. Please try again.
What is my name?: Python
Correct!
>>>
```

- break文が実行された場合も正常とみなしてプログラムを終了している例。
- ユーザの回答が正解だった場合は正答を提示せずに終了。提示の必要がないため。

while-else文でbreak文の実行が前提の例：クイズプログラム

```
>>> correct_answer = 'Python' # Correct answer
>>> max_attempts = 3 # Maximum number of attempts
>>> attempts = 0 # Current attempt count
>>> while attempts < max_attempts:
...     user_answer = input('What is my name?: ')
...     if user_answer == correct_answer:
...         print('Correct!')
...         break
...     else:
...         attempts += 1
...         print('Incorrect. Please try again.')
... else:
...     print(f'The correct answer was "{correct_answer}". Game over.')
...
What is my name?: Perl
Incorrect. Please try again.
What is my name?: Java
Incorrect. Please try again.
What is my name?: Ruby
Incorrect. Please try again.
The correct answer was "Python". Game over.
>>>
```

上記は**ユーザの回答が不正解に終わった場合にのみ正答を提示**する仕様のプログラム。

for文の効用

```
1  >>> for digit in '135':
2  ...     print(digit)
   ...
   1
   3
   5
3  >>>
```

命令は**2行**。反復回数やオフセットを指定する**必要なし**。無限ループの**心配なし**。

文字列型ではなく、**1, 3, 5** のように**数値の羅列**でイテラブルを与えても良い。

How Pythonic! You must be a Pythonista.

```
1  >>> digits = '135'
2  >>> offset = 0
3  >>> while offset < len(digits):
   ...     print(digits[offset])
   ...     offset += 1
   ...
   1
   3
   5
4  >>>
```

命令は**5行**。反復回数やオフセットを指定する**必要あり**。無限ループの**心配あり**。

Hmm... it's not Pythonic. I would like you to be a Pythonista.

- **for文**は**反復回数が決定している場合に利用する**と効果的。
- 反復の回数や対象はPythonがイテラブルから自動算出。無限ループの心配はない。
- **while文**は**反復回数がユーザに依存するなどの未定の場合に利用する**と効果的。

for文でbreak文を用いた例：最初の偶数を検出するプログラム

```
>>> for number in 1, 3, 5:
...     if number % 2 ==0:
...         print('Found even number', number)
...         break
...     print(number)
...
1
3
5
>>>
```

シーケンス。奇数ばかりで偶数がない。

実行結果からもシーケンスに偶数はないのでif文の中にある**break文が実行されていない**ことがわかる。

```
>>> for number in 1, 2, 5:
...     if number % 2 ==0:
...         print('Found even number', number)
...         break
...     print(number)
...
1
Found even number 2
>>>
```

シーケンス。奇数の1と5、偶数の2がある。

実行結果からも偶数が見つかるとif文の中に入り、**break文が実行されている**ことがわかる。
break文以降の命令は何も実行されていない。

break文：反復処理を中断し、ループから抜け出す命令。

for文でbreak文とcontinue文を用いた例：正の整数のみを足し算するプログラム

```
1  >>> total = 0
2  >>> for number in -3, 2, -1, 0, 1, 'injection', -2, 3:
   ...     if number == 'injection':
   ...         print('Injection found')
   ...         break
   ...     elif number <= 0:
   ...         continue
   ...     else:
   ...         total += number
   ...
   Injection found
3  >>> print(total)
   3
4  >>>
```

(参考) 想定外の値があればbreak文を実行。

値が0以下の整数であれば次の反復処理を開始。

正の整数のうち 'injection' 前の2と1は足されているが 'injection' 後の3は足されていないので足し算の結果は３となる。0以下の整数の場合は、**continue文**によって**反復処理を中断し、ループの先頭から命令を再開**。

continue文：反復処理を中断し、ループの先頭に戻る命令。

for-else文

```
1  >>> word = 'Pneumonoultramicroscopicsilicovolcanoconiosis'
2  >>> for letter in word:
   ...     if letter == '/':
   ...         break
3  >>>     print(letter)
   ... else:
   ...     print('No "/" in there.')
   ...
   P
   n
   (略)
   s
   No "/" in there.
4  >>>
```

構文

```
for 要素 in イテラブル:
    ループ命令
else:
    反復処理が全要素へ適用されたときに実行される命令
```

for-else文とは、反復処理の中断がある場合とない場合とで命令パターンを変える**構文**。

- while-else文と同様に、break文が実行されないことを前提としたプログラムを書くケースとbreak文が実行されることを前提としたプログラムを書くケースがある。
- **break文を書かなくてもエラーは生じない**。while-else文でも同様。
- break文を必要としないプログラムを作成するのであればelse文を書く意味はない。
- 反復処理後の順次処理（ループ外の処理）として命令を書く場合と同じ結果となる。

数値シーケンス

- これまでにfor文で用いたイテラブルな例として「`5, 4, 3, 2, 1`」や「`'135'`」や「`-3, 2, -1, 0, 1, 'injection', -2, 3 `」や「`'Pneumonoultramicroscopicsilicovolcanoconiosis' `」があった。
- 他にもイテラブルなオブジェクト（タプル、リスト、辞書、集合など）がある。
- イテラブルであれば、in演算子を用いたり、in演算子とfor文を併用したりできる。
- **イテラブル**なオブジェクトは**シーケンス**ともいえる。
- シーケンスのうち、数値リテラルの列であるものは**数値シーケンス**。
- range関数を用いることにより数値リテラルを羅列した場合と**同等な**数値シーケンスを**一括生成**できる。
- range関数を用いて生成したシーケンスもイテラブルであり、in演算子を用いたり、in演算子とfor文を併用したりできる。

range関数

- range関数はスライスとよく似た形式で利用する。

書式 `range(start, stop, step)`

- startを省略すると、代わりに値0が自動的に用いられる。
- stopは必ず指定する必要があり、スライスと同様に、作成される最後の値はstopの直前になる。
- stepのデフォルト値はこれもスライス同様に1となる。-1を指定すると逆順に数値シーケンスを生成できる。

range関数による数値シーケンスの生成

```
1  >>> for x in range(0, 3):
   ...     print(x)
   ...
   0
   1
   2
2  >>> for x in range(2, -1, -1):
   ...     print(x)
   ...
   2
   1
   0
3  >>>
```

0から2の数値シーケンスをステップ1で生成する。

逆に２から０の数値シーケンスをステップ-1で生成する。

- シーケンスの長さが小さい場合、リテラルの羅列でも対応できる。
- シーケンスの長さが大きい場合、range関数を用いることが効果的。
  例 range(0, 10000)、range(200000)など。
- 書式のルールはスライスと同じ。
- 値のパターンを変えて様々な数値シーケンスをrange関数で生成してみると良い。

反復処理の入れ子

```
1  >>> i = 1
2  >>> while i <= 9:
   ...     j = 1
   ...     while j <= 9:
   ...         result = i * j
   ...         print(f"{i} x {j} = {result}")
   ...         j += 1
   ...     i += 1
   ...
   1 x 1 = 1
   1 x 2 = 2
   (略)
   9 x 9 = 81
3  >>> for i in range(1, 10):
   ...     for j in range(1, 10):
   ...         result = i * j
   ...         print(f"{i} x {j} = {result}")
   ...
   1 x 1 = 1
   1 x 2 = 2
   (略)
   9 x 9 = 81
4  >>>
```

内側のwhile文

外側のwhile文

内側のfor文

外側のfor文

- このように、**ある文の中に同じ文が入っている状態**を**入れ子**という。
- 入れ子の深さに制限はないが、作りすぎは 難読化の元。

まとめ

- **反復処理**とは**同様の命令を2回以上連続的に実行**させること。
- 順次処理、分岐処理、反復処理を組み合わせることで**様々なプログラムを作成することができる。**
- 反復処理の構文として、**while文**、**for文**がある。
- while文は**反復処理の回数が不明**の場合に用いることが多い。
- プログラムが**無限ループ**にならないように注意。
- for文は**反復処理の回数が事前に決定**している場合に用いる。
- **break文**は反復処理を中断してループから抜け出す命令。
- **continue文**は反復処理を中断してループの先頭に戻る命令。
- **while-else文**や**for-else文**は、反復処理の中断がある場合とない場合とで命令パターンを変える構文。
- while文またはfor文においても**入れ子**を作ることができる。

次回予告　**タプル、リスト、辞書、集合**