プログラミング実習I クラス5 (井村担当)

知能・機械工学課程 井村 誠孝

m.imura@kwansei.ac.jp

繰り返し(その2): while文

while文による繰り返し処理

- whileを使うと条件が成立する間だけ繰り返すことができる．
  - 例: 乱数で発生させた値が4になるまで繰り返す．

```
import random
rand_num = 0
while rand_num != 4:
    rand_num = random.randint(0, 9)
    print(rand_num)
```

条件

同じ部分を何度も実行する
実行するたびに状態(変数の値)が変わる

  - 実行結果の例

```
6
1
4
```

while文による繰り返しの詳細

rand_num の値

```
import random
rand_num = 0
while rand_num != 4:
    rand_num = random.randint(0, 9)
    print(rand_num)
```

rand_num != 4

0
0 — True
6
6

繰り返し2回目

```
while rand_num != 4:
    rand_num = random.randint(0, 9)
    print(rand_num)
```

6
1 — True
1

繰り返し3回目

```
while rand_num != 4:
    rand_num = random.randint(0, 9)
    print(rand_num)
```

1
4 — True
4

繰り返し4回目

```
while rand_num != 4:
    rand_num = random.randint(0, 9)
    print(rand_num)
```

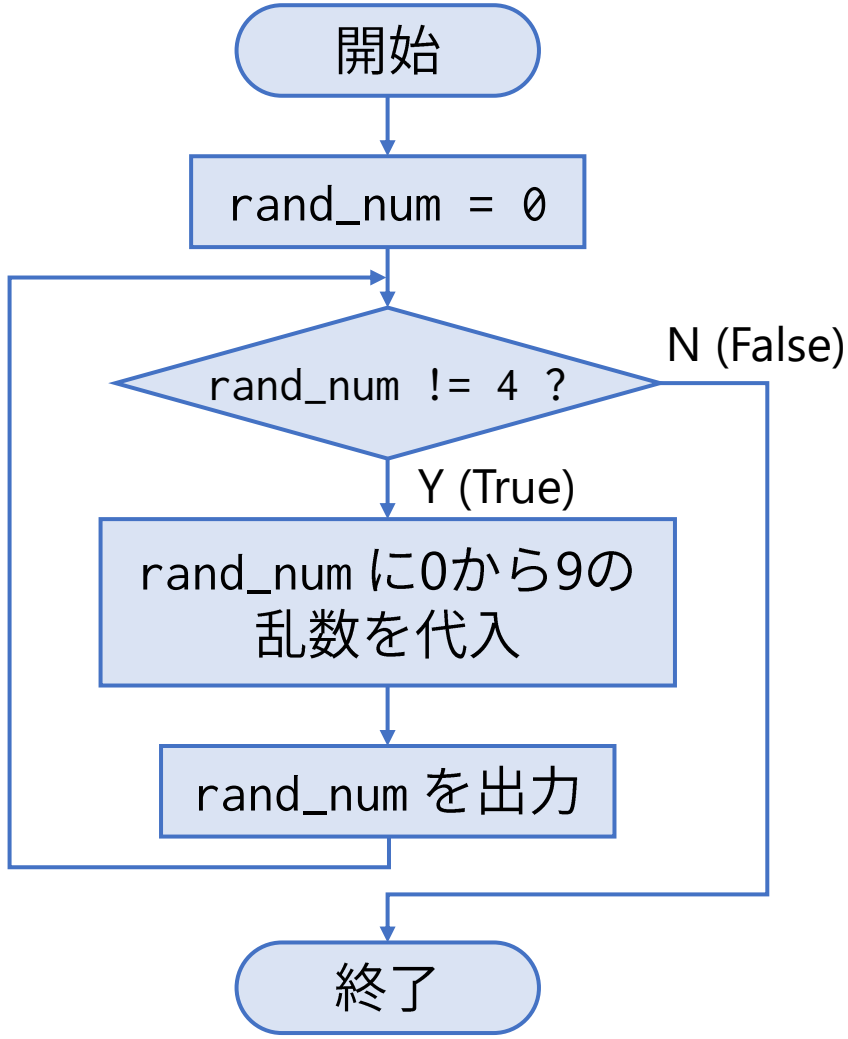
4 — False

ブロックの外に抜ける

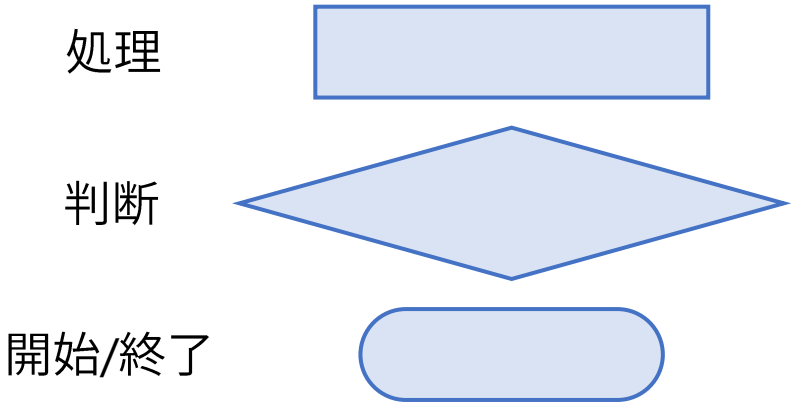
while文のフローチャートでの表現

- 処理の流れ(フロー)を図(チャート)で示す

```
import random
rand_num = 0
while rand_num != 4:
    rand_num = random.randint(0, 9)
    print(rand_num)
```

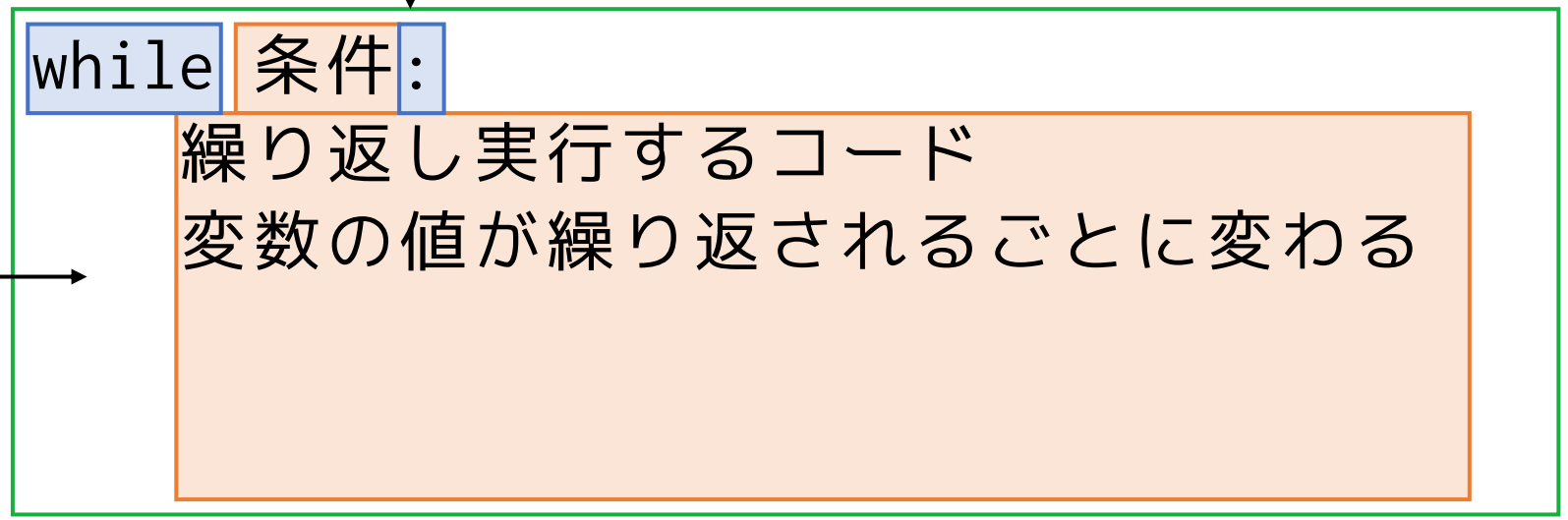


- 主な記号



while文の構造

コロンを忘れないように．
コロンが現れると，Pythonは「次はブロックだな」とわかる．



字下げ(インデント)で
ブロックを表現

お決まりの部分(常に同じ)

処理に合わせて変える

for と while の違い

- while
  - 繰り返し処理を継続する条件がわかっている．
  - 何回繰り返すかは実行してみないとわからない．
- for
  - 繰り返し処理する対象や回数があらかじめ定まっている．
    - ただし後述する break 文を使えば，繰り返し処理を中断することも可能である．

処理内容に応じて適切な方を選択しよう

繰り返しの流れを制御: break文とcontinue文

無限に繰り返す

- 条件として True を与えると，処理を無限に繰り返す．

```
import random
while True:
    rand_num = random.randint(0, 9)
    print(rand_num)
```

- このプログラムは停止しない…

一般的にプログラムが停止しない場合は，
Ctrl+Cで止めてください．

breakで繰り返しから抜ける

- 繰り返しブロックの中で break を使うと，以降のコードを無視して繰り返し処理を終了する．

```
import random
while True:
    rand_num = random.randint(0, 9)
    print(rand_num)
    if rand_num == 4:
        print('4が出たから終わるよ')
        break
    print('4じゃないから続けるよ')
print('whileから抜けてきました')
```

出力例

```
7
4じゃないから続けるよ
1
4じゃないから続けるよ
3
4じゃないから続けるよ
4
4が出たから終わるよ
whileから抜けてきました
```

教科書では説明のために break と continue を両方使っていますが，breakだけ使う方がスマート

continueで繰り返しの先頭に戻る

- 繰り返しブロックの中で continue を使うと，繰り返しの最初に戻る．

```
import math

print('log(x)を求めます．Enterのみで終了．')
while True:
    s = input('x = ')
    if s == '':
        break
    x = float(s)
    if x <= 0:
        print('xは正である必要があります．')
        continue
    log_x = math.log(x)
    print(f'log({x}) = {log_x}')

```

数学関数logの値を求めるためmathをインポート

入力を変数 s に代入．s は文字列(str)型

s が '' (空文字列)と等しければ，
while の外に抜ける

s を小数(float)型に変換し変数 x に代入．

x の値が0以下であれば，
logを計算できないことを表示して
先頭に戻る

log(x)を計算して変数 log_x に代入．
結果を表示．ブロック末尾なので，先頭に戻る．

ブロックから抜けてきたが，
続く命令は無いので，プログラム終了．

break と continue は for文でも使えます

- breakの例: 10以上の要素を探し，1つ見つけたら終了する

```
data = [1, 5, 3, 12, 6, 4, 7]
for n in data:
    if n >= 10:
        print(n)
        break
```

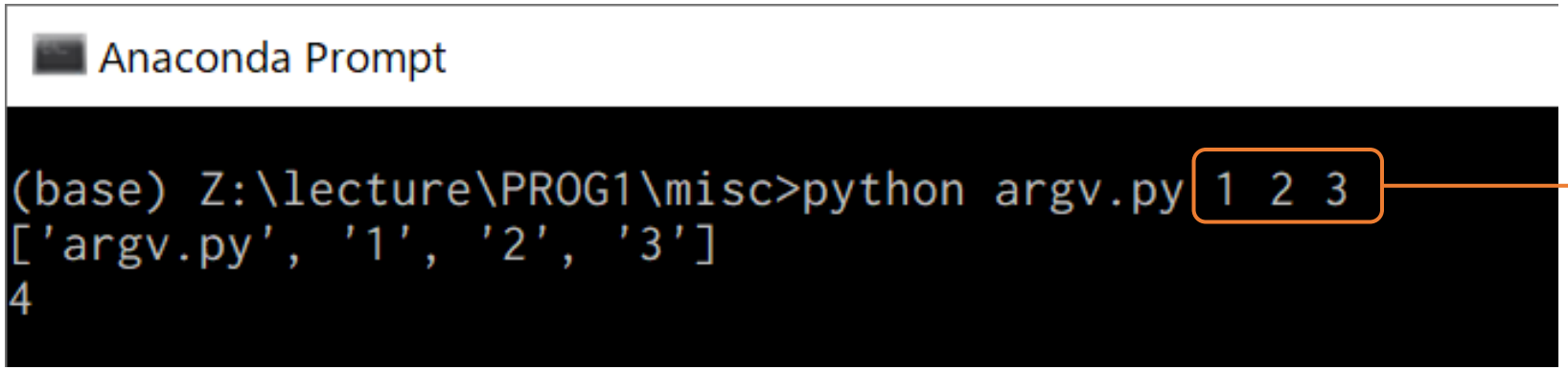
- continueの例: 3の倍数だけ表示しない

```
data = [1, 5, 3, 12, 6, 4, 7]
for n in data:
    if n % 3 == 0:
        continue
    print(n)
```

コマンドライン引数

プログラムにデータを入力する方法

1. スクリプト中にデータを埋め込む．
2. 関数 input() を使ってキーボードから入力する．
3. (ウィンドウを開くアプリケーションの場合) 入力欄(テキストボックス)に入力する．
4. ファイルに記述しておき，プログラムで読み込む．
5. コマンドライン引数として与える．



コマンドライン引数とは

- Pythonスクリプトを実行する際に，スクリプト名に続いて，空白で区切った引数を与えると，リストとして参照することができる．
  - モジュール sys のリスト sys.argv に格納される．

```
import sys
print(sys.argv)
print(len(sys.argv))
```

モジュール sys の import が必要

  - 実行例

```
>python argv.py
['argv.py']
1

>python argv.py a b 12
['argv.py', 'a', 'b', '12']
4
```

1つめの要素はスクリプト名

引数は空白で区切られる
全て文字列として格納される

コマンドライン引数に関するありがちな間違い

- リストの最初の要素はスクリプト名である．

```
import sys

name = sys.argv[0]
print(f'こんにちは，{name}さん！')
```

```
(base) Z:\lecture\PROG1\exercise\AB06>python command-line.py 井村
こんにちは，command-line.pyさん！
```

- 全て文字列としてスクリプトに渡される．

```
import sys

x = sys.argv[1]
y = sys.argv[2]
z = x + y
print(z)
```

```
(base) Z:\lecture\PROG1\exercise\AB06>python command-line.py 1 2
12
```

エラー処理

例: コマンドライン引数の2つの数値を除算

```
import sys

a = float(sys.argv[1])
b = float(sys.argv[2])
print(a / b)
```

```
>python argv.py 5 2
2.5
```

行儀のよいユーザの場合: 問題ない

ユーザがプログラムの想定していない
コマンドライン引数を与えたら?

今までプログラムがうまく動かないのは
全てプログラマ(あなた)の責任だった.

コマンドライン引数の場合は適切な値を
入力してくれないユーザに文句を言いたくなる.

でもユーザに責任を負わせるわけには
いかないので，やはりプログラムの方で
うまく対応する必要がある.

コマンドライン引数が不適切な場合

- 様々なエラー: どれがどのエラーを出す?

```
import sys

a = float(sys.argv[1])
b = float(sys.argv[2])
print(a / b)
```

```
>python argv.py 1 0
```

```
>python argv.py 1
```

```
>python argv.py a b
```

```
Traceback (most recent call last):
  File "argv.py", line 4, in <module>
    b = float(sys.argv[2])
IndexError: list index out of range
```

```
Traceback (most recent call last):
  File "argv.py", line 3, in <module>
    a = float(sys.argv[1])
ValueError: could not convert string to float: 'a'
```

```
Traceback (most recent call last):
  File "argv.py", line 5, in <module>
    print(a / b)
ZeroDivisionError: division by zero
```

いずれにしても，エラーが出たときに止まるのは困るなあ…

コマンドライン引数が不適切な場合

- 様々なエラー: どれがどのエラーを出す?

```
import sys

a = float(sys.argv[1])
b = float(sys.argv[2])
print(a / b)
```

```
>python argv.py 1 0
```

```
>python argv.py 1
```

```
>python argv.py a b
```

```
Traceback (most recent call last):
  File "argv.py", line 4, in <module>
    b = float(sys.argv[2])
IndexError: list index out of range
```

```
Traceback (most recent call last):
  File "argv.py", line 3, in <module>
    a = float(sys.argv[1])
ValueError: could not convert string to float: 'a'
```

```
Traceback (most recent call last):
  File "argv.py", line 5, in <module>
    print(a / b)
ZeroDivisionError: division by zero
```

いずれにしても，エラーが出たときに止まるのは困るなあ…

エラー処理

- エラー(例外)が生じた際にプログラムを停止させずに処理を継続する．

```
import sys

try:
    a = float(sys.argv[1])
    b = float(sys.argv[2])
    print(a / b)
except:
    print('Error!')

print('end')
```

try の後のブロックでエラー(例外)が発生すると，
exceptの後のブロックに処理が移る．

エラーが発生しなければ，exceptの後のブロックは
実行されない．

注意: エラーの発生が予見できる場合は，適切に処理しよう

- 入力された引数が足りているか: len(sys.argv)でわかる　条件分岐で対応可能
- 0で除算していないか: bを調べればわかる　条件分岐で対応可能
- 入力された引数が数値として扱えるか: 不可能ではないがやや難しい

例外で対応する

- 例外処理のメリット: エラー処理でプログラムがわかりにくくなることが防がれる，エラーが出た時点でエラー処理に移れる
- 例外処理のデメリット: プログラムの流れを突然変えてしまう，思ってもみないエラーに気付きにくくなる

例外処理の使用は限定的，最小限に留めよう．

私が妥当だと考える例

```
import sys

try:
    a = float(sys.argv[1])
    b = float(sys.argv[2])
    print(a / b)
except:
    print('Error!')

print('end')
```

exceptで処理する例外の種類を
指定するのが作法
(何でも例外で処理しない)

```
import sys
                                        引数不足は自分でチェック
if len(sys.argv) < 3:
    print(f'使い方: {sys.argv[0]} 割られる数 割る数')
    sys.exit()
try:
    a = float(sys.argv[1])
    b = float(sys.argv[2])      数値以外の入力は例外で対応
except ValueError:
    print('数値を入力してください')
    sys.exit()
if b == 0:
    print('0では割れない')
else:                                   割る数が0でないことをチェック
    print(a / b)

print('end')
```

体型判定プログラム

BMIを求めるプログラム(少し改変)の解説

```
while True:
    height_str = input('身長(m)?: ')
    if len(height_str) == 0:
        break
    height = float(height_str)

    weight_str = input('体重(kg)?: ')
    weight = float(weight_str)

    bmi = weight / (height ** 2)

    print(f'BMI値は{bmi:.1f}です．')

    if bmi < 18.5:
        print('すこしやせすぎです．')
    elif bmi < 25.0:
        print('標準的な体型です．')
    elif bmi < 30.0:
        print('すこし太っています．')
    else:
        print('高度の肥満です．')
```

入力は文字列なので，後で使う変数 height とは別の名前がよい．
長さ0，つまり入力無しだったら繰り返しから抜ける．

[試す] height_str == '' や，not height_str でも判定可能．

floatに変換する．

[拡張] try-exceptで変換できないエラーに対処
[拡張] 0や負の値に対処
[拡張] m と cm の自動判別

体重も同様に処理して変数 weight に格納．

二乗の計算手法はいろいろある．

f文字列でもフォーマット指定は使用可能．

変数 bmi の値によって処理を振り分ける．if〜elif〜elseの使用．

elifを使うので，下限を指定する必要はない．
数値を間違って入力すると気付きにくいバグの温床となるので，
むしろ指定しない方がよい．