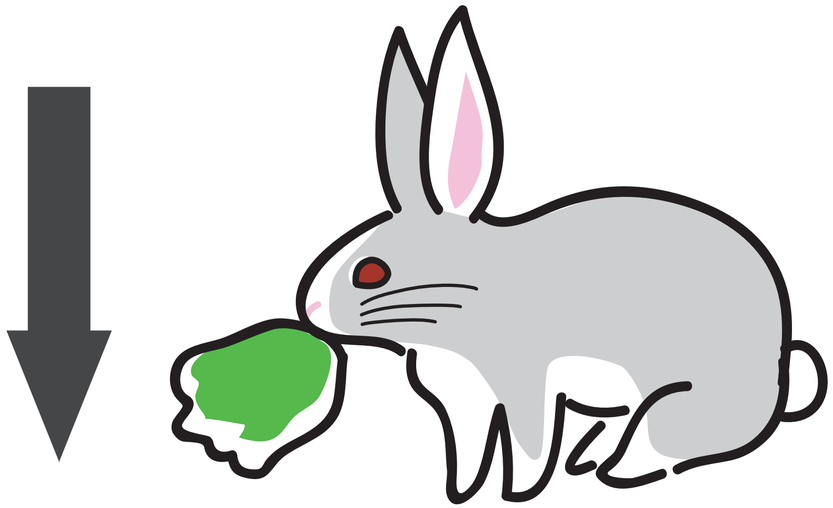
オブジェクト指向プログラミング

第４回

箕原辰夫

命令型プログラム(Imperative Program)

Go to the center of the garden ;
Find a rabbit ;
Hold her ;
Bring her to your home ;
Give her a bit of lettuce ;
Bring back her to the garden ;



Pythonの文について

- １行に１つの命令を記述する
- 上から順番に実行
  - 例：

```
print( 1 )
print( 2 )
print( 3 )
```

- １行の中で２つ以上の命令を記述する場合は、

  命令を　;（セミコロン）で区切る。 命令は、左から右に実行される

  - 例：print(1); print(2); print(3)

文とブロック

- 文
  - 式　または　代入文
- ブロック
  - 複数の文をまとめる
  - Pythonでは、空白あるいはTabによるインデントがあっているとブロックと見做される
- Pythonでは、左側に空白が空いているのがアウトラインのレベルを示す
  - ➡ TABキーを使う, Deleteで戻せる
  - ➡ Command + [(Control+[)とCommand +] (Control+])でインデントを調整できるエディタが多い
- インデントがあっていないと動作しない

代入文

- 変数名 = 最初に代入される値の式
  - 例： x = 0
- 変数名の並び = 式の並び
  - 変数名の個数分だけ、式が要求される
  - 並びは、カンマで区切られる
  - 例： x , y = 10, 20
- 変数名の並び = リストあるいはタプル
  - 各要素が、各変数に代入される

    x, y = [45, 56]
    x, y = (89, 78)

- 他のプログラミング言語では、以下をサポートしているものが多い
  - (x, y) = (10, 20)
  - [x,y,z] = [2,3,4]

- 変数名 = 変数名 = 最初に代入される値の式
  - この構文は、Python3.5ぐらいから導入された
  - 赤い部分は0個以上の繰返しが可能
  - 例：x = y = z = 0　#　3つの変数すべてに0が代入される
    - 代入演算子を使う場合は、以下のようになる
    - x = (y := (z := 0))あるいは、(x:= (y := (z:=0)))と等価

一括代入

- 変数名 = 最初に代入される値の式

  x = 100 # xに100を代入する

  y = x * 200 # 代入された変数を利用して代入も可能

  x = y = 20 # xとyに20が代入される

  x, y = 20 # yしか20が代入されない、エラーになる

  x, y = 20, 20 # xとyに20が代入される

  x = 20, 20 # これはxに(20, 20)が代入される

  x, y = 20, 30 # xに20とyに30が代入される

- 取換えも可能

  x, y = y, x

  (x, y) = (y, x) # 取換えが記述できる場合

- 取換えが記述できないプログラミング言語の場合は、以下のように記述する

```
temp = x
x = y
y = temp
```

自己参照代入文

- 左辺 = 右辺

  ➡ これは代入文であって、等しいということを示すものではない。

  ➡ 左辺の変数 ← 右辺の評価値

- そのため同じ変数が左辺にも右辺にも出てくる場合がある。

  ➡ $x = x + 1$

自己参照代入文（続き）

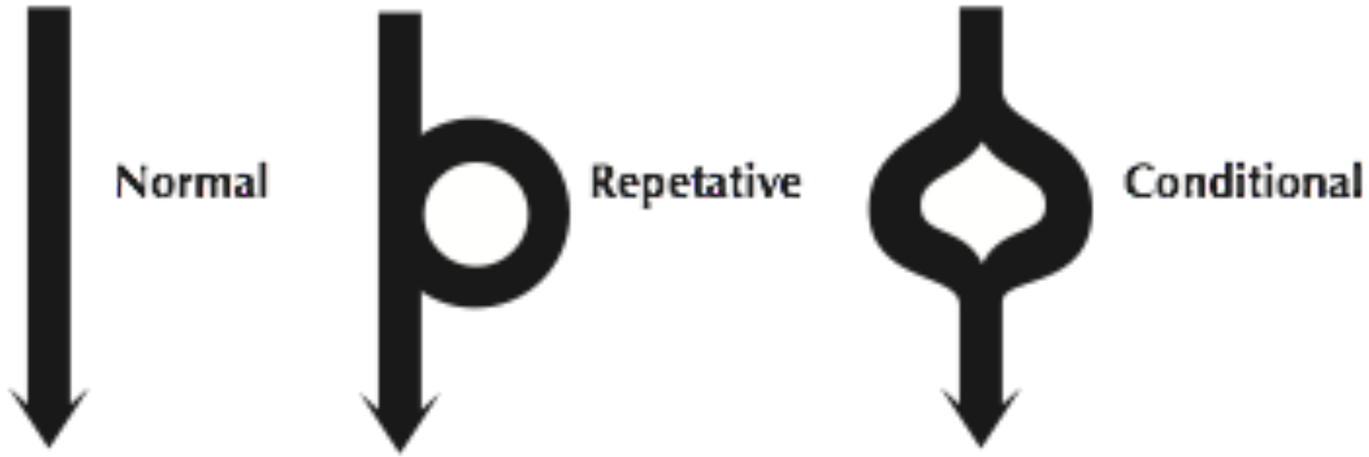
- x = x + 1
  - ➡ xのそれまで持っていた値が評価され、+1されて、新しいxの値として代入される。
  - ➡ 同じ変数が
    - ✳ 右辺に出てきたら、それまでの値
    - ✳ 左辺に出てきたら、新しく代入される
- x = x //10 * 10
  - xを10で割り切れる数に丸める
  - x ← 24　x // 10 * 10 → 24//10 * 10 → 2 * 10 → 20　x = 20

自己参照代入文の省略形

- +=　　$x = x + 5 \Rightarrow x \mathrel{+}= 5$
- −=　　$y = y - 5 \Rightarrow y \mathrel{-}= 5$
- *=　　$z = z * (x+5) \Rightarrow z \mathrel{*}= x+5$
- /=　　$w = w / (x-5) \Rightarrow w \mathrel{/}= x-5$
- //=　　$u = u // (v + 2) \Rightarrow u \mathrel{//}= v + 2$
- **=　　$v = v ** (w-5) \Rightarrow v \mathrel{**}= w-5$
- %=　　$u = u \% 5 \Rightarrow u \mathrel{\%}= 5$
- 間違えやすいもの
  - =+　　$x =+ 5 \Rightarrow x = +5$ # 単項演算子
  - =-　　$x =- 5 \Rightarrow x = -5$ # 単項演算子

制御構文（Control Statement）

- 通常の文の流れだけでなく、繰返し・条件分岐がある



条件分岐：３つのif文

- 書式１：条件を満足しなければスキップする
  - **if** 条件 : 満足するときにすること
- 書式２：条件を満足する場合としない場合
  - **if** 条件: 満足するとき

    else: しないとき
- 書式３：上から条件を満たすものを当てはめていく
  - **if** 条件: 満足するとき

    elif 次の条件: …

条件式の書き方

- 条件式を記述する
  - 式　条件演算子　式
  - 条件演算子の左右の式はどちらに書いても構わない
  - 例：x == 34　⇔　34 == x　　y * 2 >= 11 ⇔　11 <= y * 2
- 演算子は６つある（空白不可）
  - ==, !=, >, <, <=, >=
  - <=と>=は、不等号を先に書く
  - 等しいは、==（=が２つ）　**is**
  - 等しくないは、!= **is not**

リスト・文字列・タプル・集合と比較演算子

- リスト・タプルで、比較演算子を使うと要素の個数に関係なく、先頭の要素から大小の比較が行なわれる
  - 例：[34, 45] < [20, 34, 45, 89]　# Falseに評価
- リスト・タプルで、要素の個数が同じだと、先頭の要素から大小の比較が行なわれる
  - 例：[34, 45, 56] < [34, 45, 57]　# Trueに評価
- 集合の包含関係で大小の比較が行なわれる
  - 例：{34, 45, 56} < { 23, 34, 45, 78, 56 }　# Trueに評価
- 文字列で、比較演算子を使うと文字列の長さには関係なく、すべてアルファベット順（Unicode順）の評価になる
  - 例："あいう" < "かきく"　# Trueに評価

    "ABC" < "AE"　# Trueに評価

条件式とリスト・タプル・文字列・集合・辞書のキー

- **in**, **not in**も論理値を出すので用いることができる
  - 式 [**not**] **in** リストあるいはタプルあるいは文字列、集合、辞書
  - 文字列 [**not**] **in** 文字列
  - 辞書の場合には、キーの集合との比較になる
  - 例： x **in** [12, 13, 18, 19]

    "a" **in** "sample"　→ Trueに評価

    "sam" **in** "sample" → Trueに評価

    12 **in** (45, 56, 78) → Falseに評価

    21 **in** {56, 21, 34} → Trueに評価

    "one" **in** {"one":1, "two":2} → Trueに評価

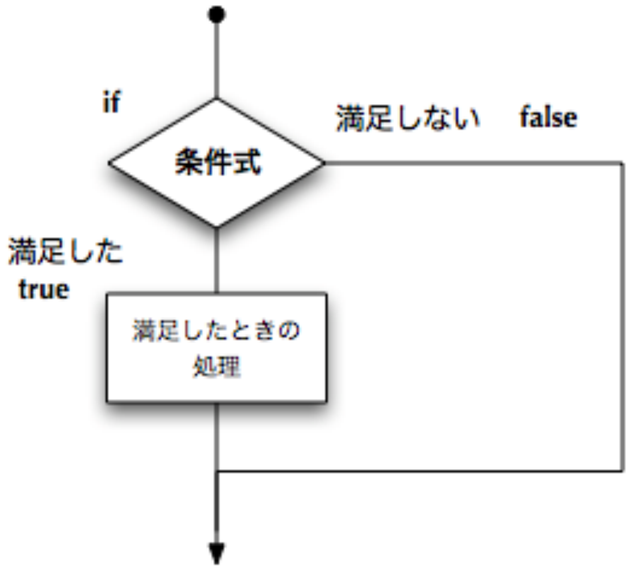
if文

if 条件式 :

　条件式が満足されたときに実行されること

例：

　　if value < 0: value = -value



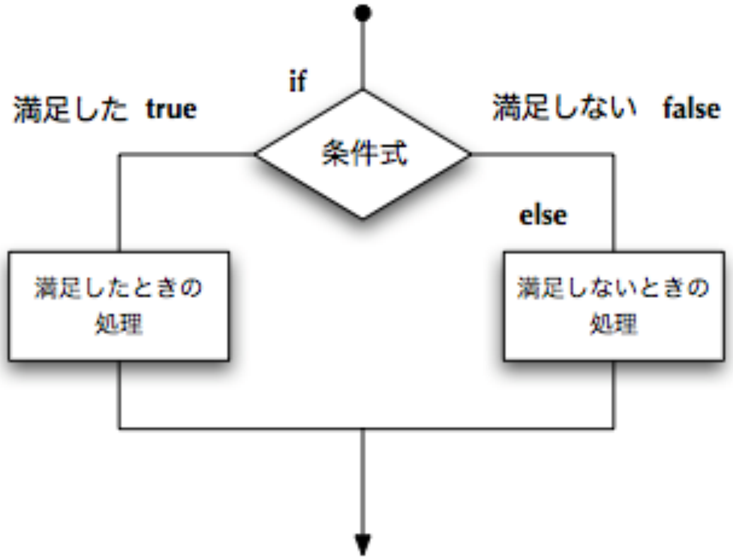
if - else 文

if 条件式:

条件式が満足されたときに実行されること

else:

満足されないときに実行されること



例：

if value % 2 == 0: print(f"{value}は偶数")

else: print(f"{value}は奇数")

if - elif - else 文

if 条件式1:

　条件式1が満足されたときに実行されること

elif 条件式2:

　条件式2が満足されたときに実行されること

else:

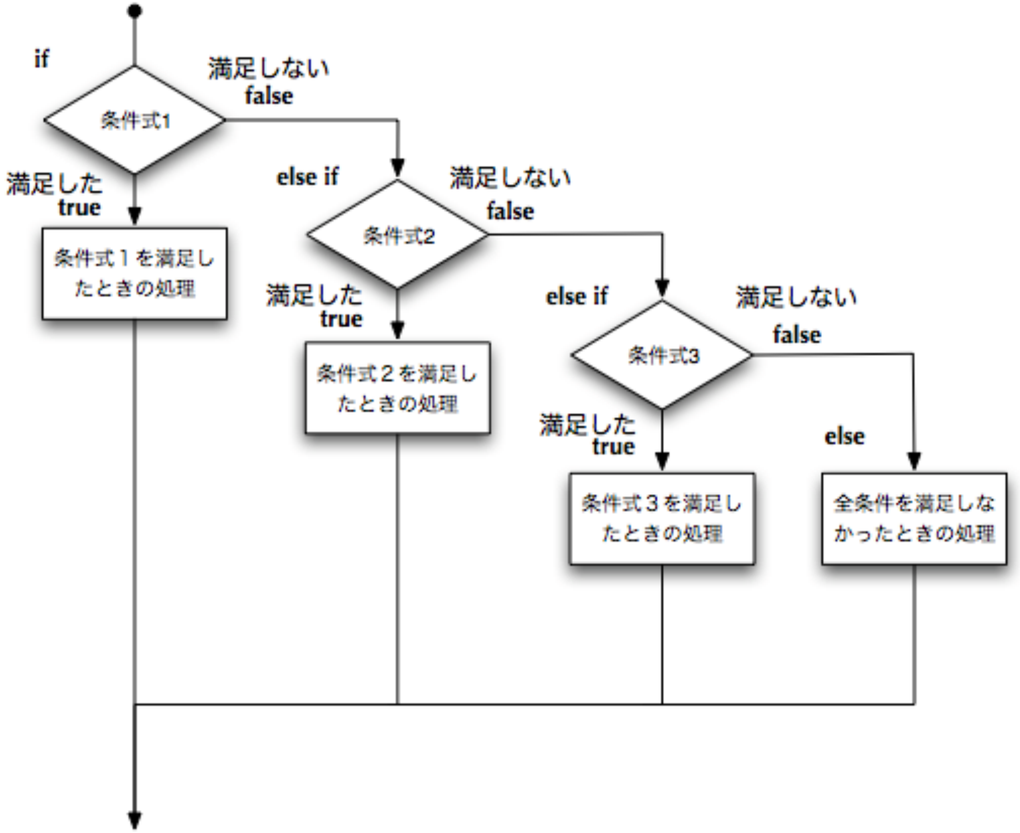
　全てが満足されないときに実行されること

青は 1 回以上あってもよい

紫は省略されてもよい

例：

```
if  value % 12 == 0: print( f"{value}は12の倍数" )
elif value % 3 == 0: print( f"{value}は3の倍数" )
else: print( f"{value}は3の倍数ではない" )
```



よくある文法エラー

- 反対の条件を書いてしまう
- 条件やelseの後に:（コロン）を忘れる
  - if文やelse文と関係ない独立ブロックになるが、その前に文法エラーになる
- elif ではなく、else if と書いてしまう
- インデントを忘れる
- 複合条件を,（カンマ）で記述する
- ２文字の演算子の間に空白をいれる
  - 例：＝＝　＞＝　＜＝

if文のネスト

- 外側のif文
  - 大きく分けたいときにつかう
- 内側のif文
  - その条件のなかで更に細かく分けたいときにつかう

例題：大の月・小の月

- 西向く侍、小の月　2, 4, 6, 9, 11(=士)
- 偶数の月と奇数の月で分かれる
  - 偶数：8よりも小さい月が小の月
  - 奇数：8よりも大きい月が小の月
- if文のネストで表現してみる
- ユーザから月を入力してもらい、その月が大の月か小の月かを判定する

論理式

- 条件式を複数使うことができる
  - 論理積（∧）　**and**　両方の条件を満たす　（C言語〜JavaScript: && ）
  - 論理和（∨）　**or**　どちらかの条件を満たす（C言語〜JavaScript: || ）
  - 否定（¬）　**not**　反対の条件にする　（C言語〜JavaScript: ! ）
- 論理式
  - 条件式
  - **not** 論理式
  - 論理式 **and** 論理式
  - 論理式 **or** 論理式

論理式の値

- 網が掛かっている部分がFalseになる。白い部分は、Trueになる。

not True	True **and** True	True **or** True
not False	True **and** False	True **or** False
	False **and** True	False **or** True
	False **and** False	False **or** False

論理式の記述の仕方

- かならず論理積・論理和などで結ぶ必要がある
  - $100 < x < 200$　は避けるようにする

    (Python3, Juliaでは機能するが、C/C++/C#/Java/JavaScript/Python2ではエラーになるか間違った動作をしてしまう)
    - $100 < x$ **and** $x < 200$
    - x **between** 100 **and** 200 (SQL)
  - $x \% 2 == x \% 3 == 0$　も避けるようにする
  - (Python3, Juliaでは機能するが、C/C++/C#/Java/JavaScript/Python2ではエラーになるか間違った動作をしてしまう
    - $x \% 2 == 0$ **and** $x \% 3 == 0$

ド・モルガン（De Morgan）の法則

- 否定のついた論理式を変換することが可能

  not (条件A **and** 条件B) ⇔ **not** 条件A **or not** 条件B

  not (条件A **or** 条件B) ⇔ **not** 条件A **and not** 条件B

- 適用例

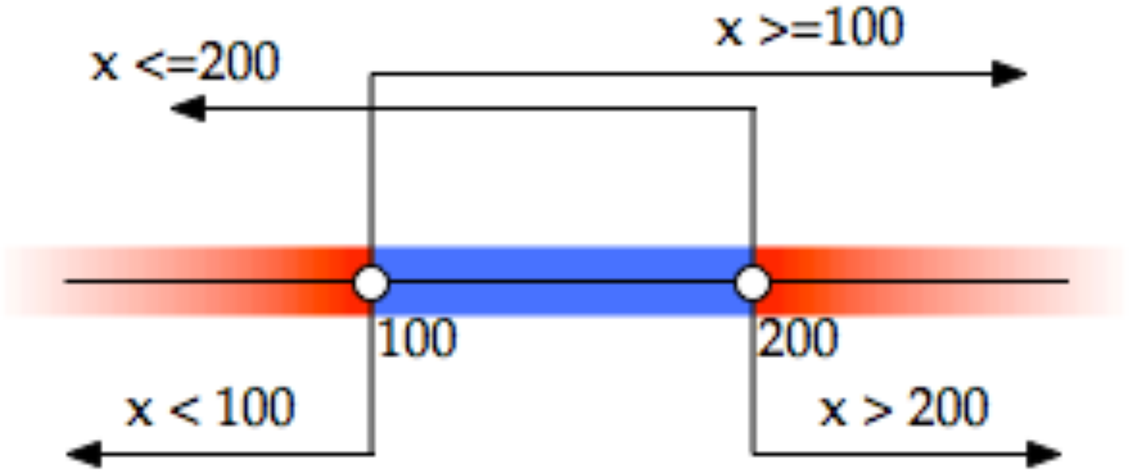
  not(x == 1 **or** x == 2)

  ⇒ **not**(x == 1) **and not**(x == 2)

  ⇒ x != 1 **and** x != 2　　x **is not** 1 **and** x **is not** 2

ド・モルガン則と数直線

- 100 <= x **and** x <= 200　　100 <= x <= 200
- **not**(100 <= x **and** x <= 200)
  - ➡ **not**(100 <= x) **or** **not**(x <= 200)
  - ➡ 100 > x **or** x > 200



それ以外の変換

- 否定と等号
  - **not**(a == b) ⇔ a != b　　**not**(a **is** b) ⇔ a **is not** b
  - **not**(a != b) ⇔ a == b　　**not**(a **is not** b) ⇔ a **is** b
- 不等号と否定
  - **not**(a >= b) ⇔ a < b　　**not**(a <= b) ⇔ a > b
  - **not**(a > b) ⇔ a <= b　　**not**(a < b) ⇔ a >= b
- 等号付き不等号の分解
  - a >= b ⇔ (a > b **or** a == b)　　a <= b ⇔ (a < b **or** a == b)

論理和・論理積の評価

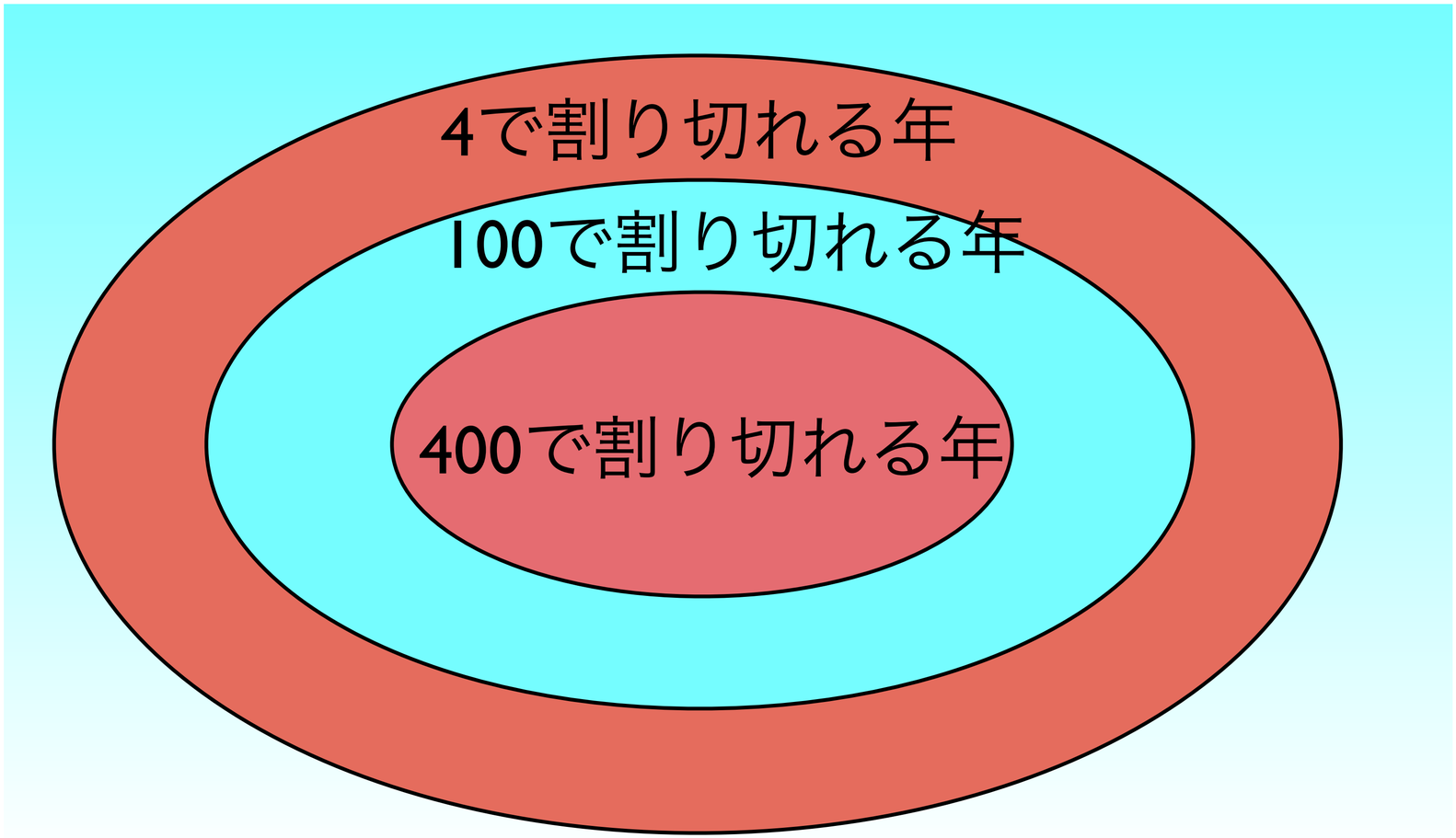
- 条件式は、左から評価される
  - x % 2 == 0 **or** x % 8 == 0
  - 偶数の時点で**True**に評価される
  - x % 2 == 0 **and** x % 3 == 0
  - 奇数の時点で**False**に評価される
- 同じ優先順位の演算は、左から評価される
  - x % 2 == 0 **and** x % 3 == 0 **and** x % 5 == 0
  - 左の条件から評価される

演習

- ユーザに西暦を入力してもらう
- その年が閏年(Leap Year)かどうか判定するプログラム
  - 4で割り切れない年は、平年　例: 2015
  - 4で割り切れる年は、閏年　例：2008
  - 100で割り切れる年は、平年　例：1900
  - 400で割り切れる年は、閏年　例：2000

閏年の求め方

- 集合の図で描くと下記のようになる



閏年の求め方

- ３つの考え方がある
  - Rare（起こりにくい）もの（内側）から記述する

    ✳ if elif 文でできる

  - 起こりやすいもの（外側）から記述する

    ✳ if文のネスト（多重化）を用いる

  - 集合の該当する部分だけを示す

    ✳ 論理式を用いる

閏年とは？

- 地球の公転周期は、365日ぴったりではない。
- 4年に 1 日

  1/4 … 0.25日
- 100年 1 日は省く

  1/100 … 0.01日
- 400年に 1 日

  1/400 … 0.0025日

- 365日+ 0.25 - 0.01 + 0.0025 = 365.2425日
- 本来は、365.2422日
- 古代マヤ文明は、この値に近づけるため、閏日を設定
- 陰暦は、月と年があわなくなるので、閏月を挿入

if式

- **if**で、評価する値をどちらかに決定できる

  真の場合の値 **if** 条件式や論理式 **else** 偽の場合の値

- 式なので、代入文の中やメソッド呼出しのパラメータの中でも使える

➡ x = 10 **if** y >= 100 **else** 20

➡ c.create_….(fill="red" **if** x>100 **else** "blue")

if文とif式の等価性

- **if** x > 100:
  y = 10
  else:
  y = 20

- y = 10 **if** x > 100 **else** 20 # Python
- y = (x > 100) ? 10 : 20; // Java, JavaScript, C/C++, C#
- =IF(x > 100, 10, 20) EXCEL

EXCELのIF関数

- =IF(条件式, 真のときの値, 偽のときの値)
- =AVERAGEIF(範囲, 条件式あるいは値)
- =SUMIF(範囲, 条件式あるいは値)
- =COUNTIF(範囲, 条件式あるいは値)

- =AND(条件式, 条件式)
- =OR(条件式, 条件式)
- =NOT(条件式)

if式（続き）

- if式をネストさせることもできる

  c = 'A' **if** *y* >= 80 **else** 'B' **if** *y* >=60
  else 'C' **if** *y* >= 40 **else** 'D'

  ```
  if y >= 80: c = 'A'
  elif y >= 60: c = 'B'
  elif y >= 40:  c = 'C'
  else: c = 'D'
  ```

- かなり多用するプログラマが多い

  if文が省略できる、短く書ける

  使い過ぎると何をしているのかわからないので、ほどほどに

Python演算子の優先順位（Python 3.8版）

- 上が一番優先順位が低く、
  下が一番優先順位が高い

演算子	説明
:=	代入式
lambda	ラムダ式
if -- else	条件式
or	ブール演算 OR
and	ブール演算 AND
not x	ブール演算 NOT
in, not in, is, is not, <, <=, >, >=, !=, ==	所属や同一性のテストを含む比較
|	ビット単位 OR
^	ビット単位 XOR
&	ビット単位 AND
<<, >>	シフト演算
+, -	加算および減算
*, @, /, //, %	乗算、行列乗算、除算、切り捨て除算、剰余 [5]
+x, -x, ~x	正数、負数、ビット単位 NOT
**	べき乗 [6]
await x	Await 式
x[index], x[index:index], x(arguments...), x.attribute	添字指定、スライス操作、呼び出し、属性参照
(expressions...), [expressions...], {key: value...}, {expressions...}	結合式または括弧式、リスト表示、辞書表示、集合表示

プログラムの状態遷移

- プログラムの状態(State)

  変数が一定の状態にあることを指す

- プログラムの状態遷移(State Transition)

➡ 変数の値が移り変わっていくことを指す

プログラムの状態

- プログラムの状態

  変数の値が一定の値にあること

- 変数の値が変われば

  状態遷移が起こる

```
public class Transient {
        public void main( String [ ] arg ) {
                int x;

                x = 10;
                x = x + 11;
                x = x - 78;
                x = 32 / 4;
        }
}
```

// x の値がどんどん変わっていく

繰返しを記述する構文

- **while文**

  どのような言語でもある

- **for文**

  Pythonでは、リストあるいは範囲指定のオブジェクトに対しての繰返しになる

while文による繰返し

while 継続条件を示す論理式:

繰り返したいこと

[**else:**

繰返しの最後に実行すること

（1回も繰返しされなくても実行される）

]

- **while**文の意味

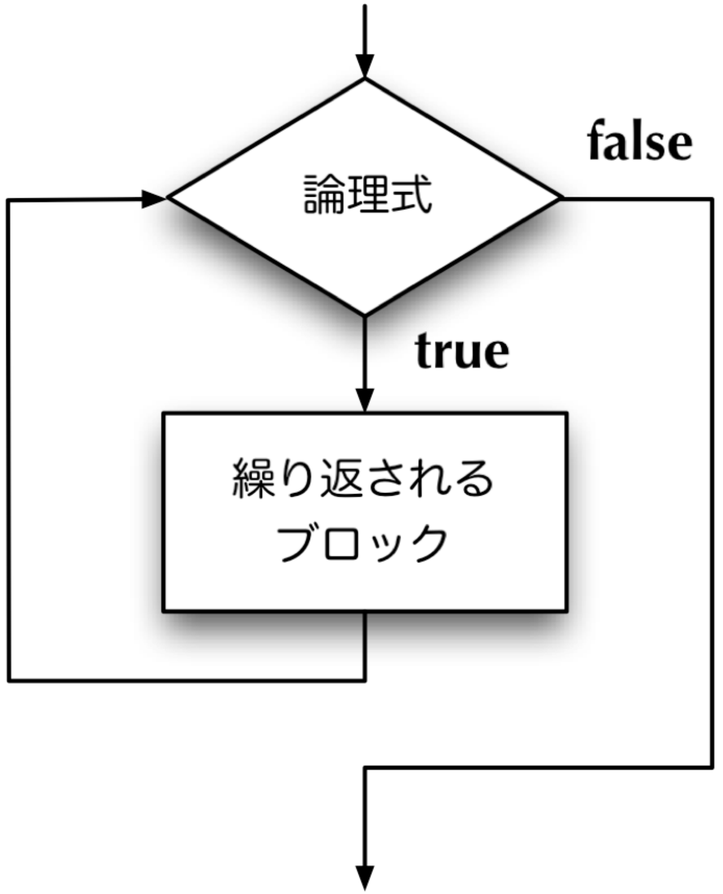
  継続条件が満たされている間、実行する

- **while**文を使うには、

  いつかは継続条件を満たさなくなるように状態遷移させる

while文の動き

- 繰り返すたびに、論理式を評価し、Falseになったら、次にいく。



繰返しを作るには

- 繰り返したい部分をブロックでまとめ、インデントする
- 状態遷移をさせる部分をブロックの中に入れる

- **while** 論理式:

  繰り返したい内容

  状態遷移させる内容

回数繰返し文

- 繰返しをしている部分がどの範囲か明確になる。

```
count=1
while count <= 10 :
    # 繰り返される内容
    count=count+1
```

- 字下げを手動で行なうには、TABキーを使う
- 字下げを戻すのは、deleteキー
- インデントの上げ下げは、⌘+]と⌘+[
  Windowsでは、control+]とcontrol+[

状態遷移は変数の値を使う

- 変数の値を使って指折り数えさせることができる（ループ変数と呼ばれる）

  このときの変数は整数型

- 変数を大きくしたり、小さくしたりして、いずれは終了させる

ループ変数の変化

- 変数の変化の仕方で繰返す回数が決まる

count = 初期値

while *count* < 最終値 :

count = *count* ＋ 差分

- 繰返しの回数はCeilを使って求められる

⌈（最終値 - 初期値）／差分　⌉

- ⌈a⌉…Ceil: aと等しいか、aよりも大きい最小の整数

ループ回数の例

```
m = 2
while m < 21:
  m = m + 3
```

- 回数は、⌈（21 - 2）/ 3　⌉で、7回

変数の値を使った繰返し

- ループ変数の値を変えていく
- 最終的に、継続条件を満たさなくなるようにする
- ループ変数の値を使ってメソッド呼出しのパラメータなどに使える
- 複数の変数が、繰返しの中で状態遷移する場合は、どれか1つをループ変数に割り当てる
- 変数をいくつ使うべきかは、表わしたいデータの種類の個数による

ループ変数のトレース

- ループ変数の値の変化を追う
  - 最初の値（初期値）
  - 途中の増分（差分）値
  - 継続条件が終わるときの値（最終値）
- 値の推移を追っていれば繰返しがどう動くかわかる

- IDLEでは、停めたい行のところで、右クリックして「Set breakpoint」で停めたい行（その行の実行の前で止まる）を指定。⇒黄色く表示される
  - IDLEのShellのDebugメニューのDebuggerのところで、デバッガを起動しておく
  - 現れたDebugパネルのsourceの部分をチェックしておく→実行されているプログラムのソースが表示される
  - Goで停めたい行の手前で止まる、Stepは 1 行ずつ実行するが、関数呼出しをしていると、そちらに行ってしまうので注意（print関数など）、知らない関数の内部に行ってしまったら、Outを使う
  - 関数の内部まで行かずに、 1 行ずつ実行したいときは、Overを使う

繰返しを作るコツ

- ループ変数がいつかは継続条件を満たさなくなるように条件を作る
- ループ変数がどの変数か注意する
- 条件はきつめに設定する

  n = 10 #9の場合は、0を飛び越えて1の後は、-1になる

  while n != 0: → n > 0とした方が良い

  　n -= 2

- 1回も繰返さない場合もある

  n = 10
  while n < 0: # while文は脱出条件を記述するのではなくて、継続条件を記述する

  　n = n - 1

繰返しの文法ミス

- **while** 条件式: 複数の文を書きたいときは ; で区切る

  while 条件式:

  　　インデントがあっていない文

  　　　インデントがあっていない文

- インデントがあっていないと、１つのブロックとしては認められず、実行前に怒られてしまう

多重の繰返し

- ループ変数を２つ使う

```
n = 1
while  n <= 10  :
    m = 1
    while  m <= 5  : m = m + 1
    n = n + 1
```

- 外側の繰返しと内側の繰返しに分かれる
- 内側の繰返しが行なわれるのは

  外側の繰返しの回数　×　内側の繰返しの回数

- 掛け算になることに注意する
- 内側の繰返しの継続条件を外側の繰返しのループ変数を使って制御できる

繰返しと条件分岐の組合せ

- 繰返しの中のブロック内に、if文を入れる
- ある回だけ、特別な制御をさせたいときに使う
- if文の条件に整数剰余を使うと周期的に何かをさせられる

```
n=1
while  n <= 10 :
    if  nを使った条件式 : ....

    n = n + 1
```

整数除算、整数剰余を使った繰返し

- 整数除算
  - 段階的になる

    count // 5 * 5　　(5の倍数のみ)

- 整数剰余
  - 周期的になる

    count % 7　(0〜6の値しか出てこない)

- 繰返しで値を見てみる

整数除算・剰余と繰返しの組合せ

- 整数除算
  - 段階的になにかをさせたいときに使う

    x // n * n　　→ nを１つの段階にできる
- 整数剰余
  - 周期的に何かをさせたいときに使う

    x % n →　nが周期になる

- カレンダーを表示させてみる

補数と逆数

- 補数
  - 足してその数になる対の数
  - 10の補数
    - 1と9　2と8　3と7　4と6　5と5

$$14 - 9 = 10 + 4 - 9 = 10 - 9 + 4 = 1 + 4 = 5$$
$$14 - 9 = 14 - (4+5) = 14 - 4 - 5 = 10 - 5 = 5$$

- 逆数
  - 掛け算すると 1 になる数
  - 4 → 1/4　5/6 → 6/5

補数を使ってみる

- 10000円で7892円のときのおつり
  - 10の補数 = 9の補数+1
  - 9の補数は、足して9になる組み合わせ

    9999
    – 7892
    = 2107

    2107 + 1 = 2108
- 14683円で7892円のときのおつり

  14683 = 10000 + 4683

  14683 – 7892 = 10000 – 7892 + 4683

  = 2108 + 4683 = 6791

補数・逆数を使った繰返し

- 補数
  - 定数からループ変数を引く
  - 100 - 10 * n
- 逆数
  - 定数をループ変数で割る
  - 100 / n