スクリプト言語プログラミング
Pythonによる数値解析

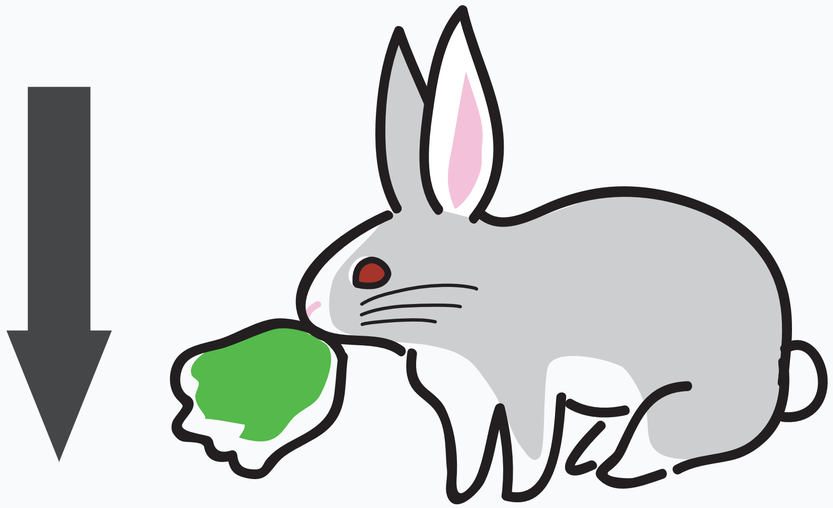
第４回講義資料

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　　または　関数呼出し
- 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
- 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
  - (x, y) = (10, 20)　# Python以外にC#, Swift, Lua, Julia, CLU, Occamなどの場合
- 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

代入と参照

- 変数名 = 最初に代入される値の式
  - x = 100 # xに100を代入
  - y = x * 200 # 代入された変数を利用して代入も可能
  - 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, y = 20, 30 # xに20とyに30が代入される
  - x = 20, 30 # xに(20, 30)が代入される
  - x, _, _, _ = 20, 30, 40, 50 # xに20が代入される
- 取り換えも可能
  - x, y = y, x　　(x, y) = (y, x)　# Julia, C#, Swiftはこの方式

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 ← 右辺の式の評価値

- 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

自己参照代入文の省略形

+= $x = x + 5 \Rightarrow x \mathrel{+}= 5$

–= $y = y - 5 \Rightarrow y \mathrel{-}= 5$

*= $z = z * (x+5) \Rightarrow z \mathrel{*}= x+5$

/= $w = w / (x-5) \Rightarrow w \mathrel{/}= x-5$

//= $u = u // (v + 2) \Rightarrow u \mathrel{//}= v + 2$

**= $v = v ** (w-5) \Rightarrow v \mathrel{**}= w-5$

%= $u = u \% 5 \Rightarrow u \mathrel{\%}= 5$

=+ $x =+ 5 \Rightarrow x = +5$ # 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

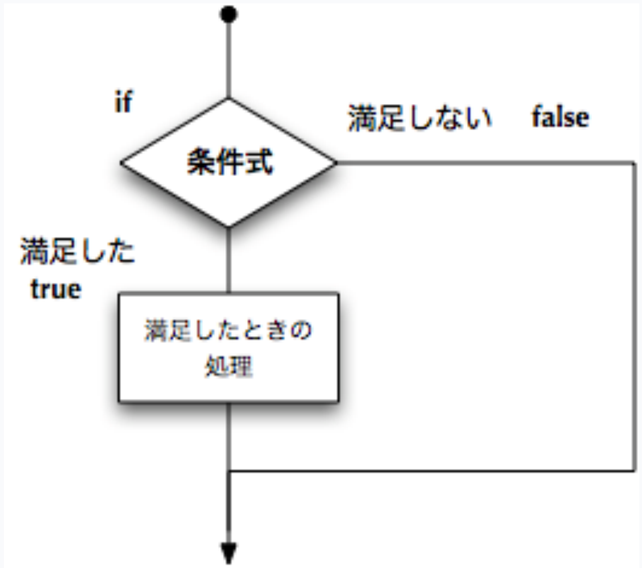
    else: しないとき
- 書式３：上から条件を満たすものを当てはめていく
  - **if** 条件: 満足するとき

    elif 次の条件: …

if文

if 論理値に評価される条件式 :

　条件式が満足されたときに実行されること



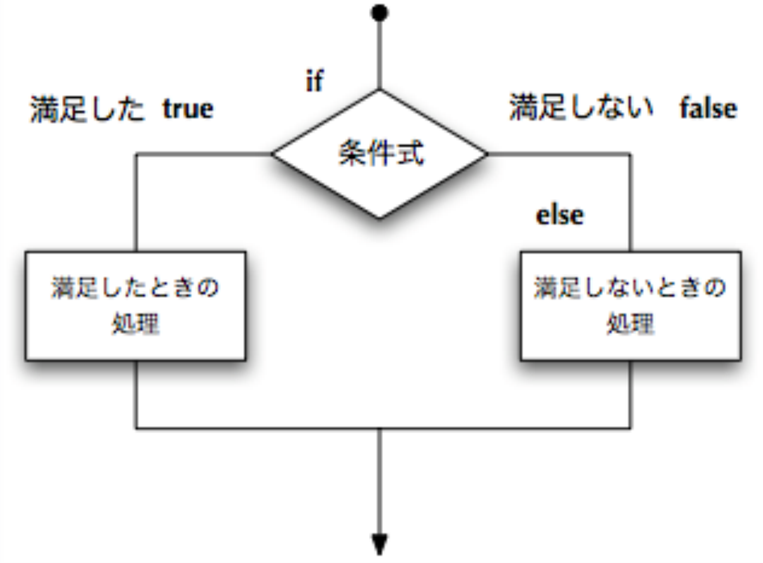
if - else 文

if 論理値に評価される条件式:

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



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:

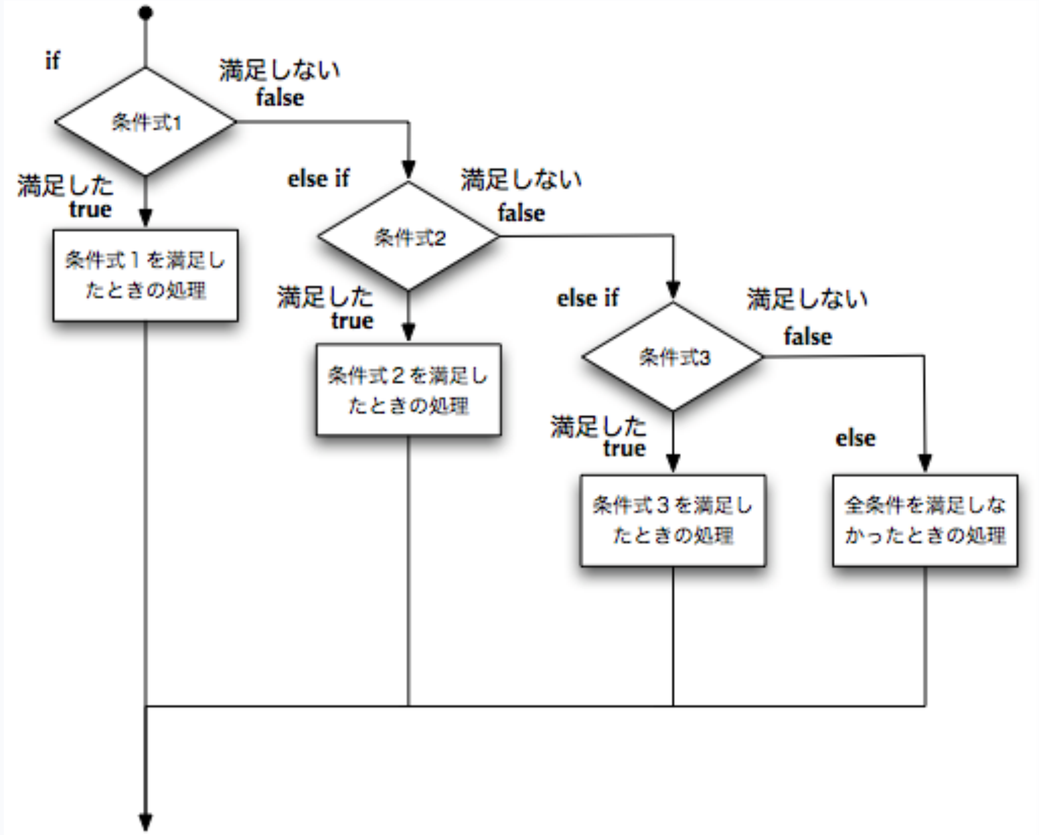
　条件式2が満足されたときに実行されること

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



条件式の書き方

- 論理値に評価される条件式を記述する
  - 式　比較演算子　式
  - 比較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
- 比較演算子は6つある（2文字演算子は、空白不可）
  - ==, !=, >, <, <=, >=, **is, is not**
  - <=と>=は、不等号を先に書く
  - 等しいは、==（=が２つ）, あるいは**is**　（isは、構造的な値の場合はオブジェクト識別子の等価性が評価される）
  - 等しくないは、!=、あるいは **is not**（is notは、構造的な値の場合はオブジェクト識別子の等価性が評価される）

条件式とリスト・タプル・文字列

- **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集合あるいは辞書

    …式で表わされる要素が含まれているかどうか判断する
  - 文字列 [**not**] **in** 文字列

    …最初の文字列が、2番目の文字列に含まれるかどうか判断する
  - 例： x **in** [12, 13, 18, 19]

    12 **in** (45, 56, 78) → Falseに評価

    "a" **in** "sample" → Trueに評価

    "pl" **in** "sample" → Trueに評価

リスト・文字列・タプルと比較演算子

- リスト・タプルで、比較演算子を使うと要素の個数が異なる場合、先頭の要素から順に評価される
  - 例：[34, 45] < [12, 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
- 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、先頭の文字から、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

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
- **elif** ではなく、**else if** と書いてしまう
- 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：＝＝　＞＝　＜＝

代入演算子とif文の条件式（Python 3.8より）

- if文の条件式の中に、代入演算子を記述することが可能になった。代入演算子の場合は、()で囲むことが必要なので注意すること
- 例：

```
if (n := len(a)) > 10:
    print( f"Listが長すぎます。{n}要素あります, 要素数は10以下にしてください。" )

if (value := int( input( "整数を入力: " ) ) ) < 0:
    print( f"{value}は負です。正の整数を入力して下さい" )
```

- 代入された変数は、それ以降も有効となる

型の取出しと比較演算子

- 変数や式の型を取り出すことができるtype( )組込み関数がある
- 比較演算子 ==, !=で比較することができる
- 比較対象として、型名 bool, int, float, complex, strを用いることができる、また構造型の型名として、list, tuple, set, dictを用いることができる。構造型の場合、変数の要素の型名までは取り出す組込み関数はない。
- 例：

```
if type( value ) == int: print( f"{value}は整数" )
elif type( value ) == float: print( f"{value}は実数" )
elif type( value ) == complex: print( f"{value}は複素数" )
# Pythonの場合、それぞれの型の包含関係は定義されていない
```

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：

```
if value > 0:
  if value % 2 == 0: print( "正の偶数" )
  else: print( "正の奇数" )
elif value == 0: print( "零" )
else: print( "負の数" )
```

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
  - 論理和（∨）　**or**　どちらかの条件を満たす
  - 否定（¬）　**not**　反対の条件にする
- 論理式の書式：
  - 条件式
  - (論理式)
  - **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。

andは、全部の項がTrueのときだけ、評価結果がTrueになる

orは、全部の項がFalseのときだけ、評価結果がFalse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

- 複数の条件式は、必ず論理積・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

    (Python 3, Juliaでは機能するが、C/C++/Java/Python 2では文法エラーになるか、エラーにならない場合は、間違った動作をしてしまう)

  - $100 < x$ **and** $x < 200$　　と記述した方が無難
- 次のような条件式も記述できるが、andで繋げた方が無難
  - x%2==x%3==0　→ x%2==0 **and** x%3==0
- それぞれの条件を記述していく
  - x ==1 **or** 2 間違った動作になる→ x==1 **or** x==2　これを書きたいのであれば、x **in** [1, 2]

ド モルガンの法則

- 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

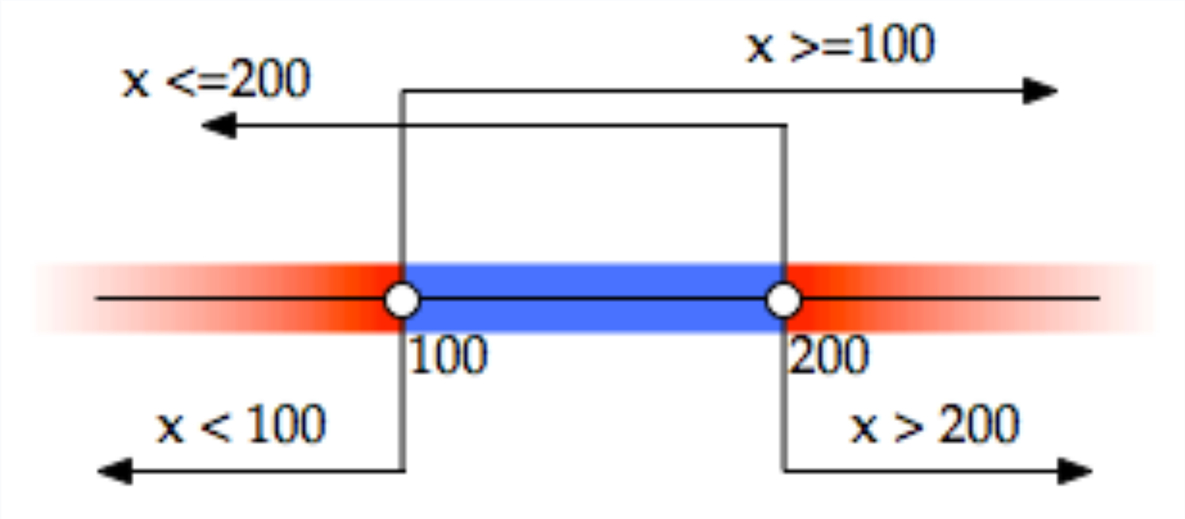
- 適用例

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

  ⇒ x != 1 **and** x != 2

ド・モルガン則と数直線

- 100 <= x **and** 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　a **is not** b
  - **not**(a != b) ⇔ a == 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)
- in 演算子の否定
  - **not** a **in** b ⇔ a **not in** b

論理和・論理積の評価

- orで結ばれた条件式は、左から評価されて、最初の条件でTrueになった場合は、あとは評価せずにTrueになる
  - x % 2 == 0 **or** x % 8 == 0
  - 偶数の時点で**True**に評価される

- andで条件式は、左から評価されて、最初の条件でFalseになった場合は、あとは評価せずにFalseになる
  - 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

- orではTrueに評価された時点で、あるいは、andではFalseに評価された時点、残りの条件は評価されないので注意すること
  - 例：(value:=2) == 2 **or** (value:= 3) == 3 # valueの値は2

    (value:=2) == 2 **and** (value:= 3) == 3 # valueの値は3

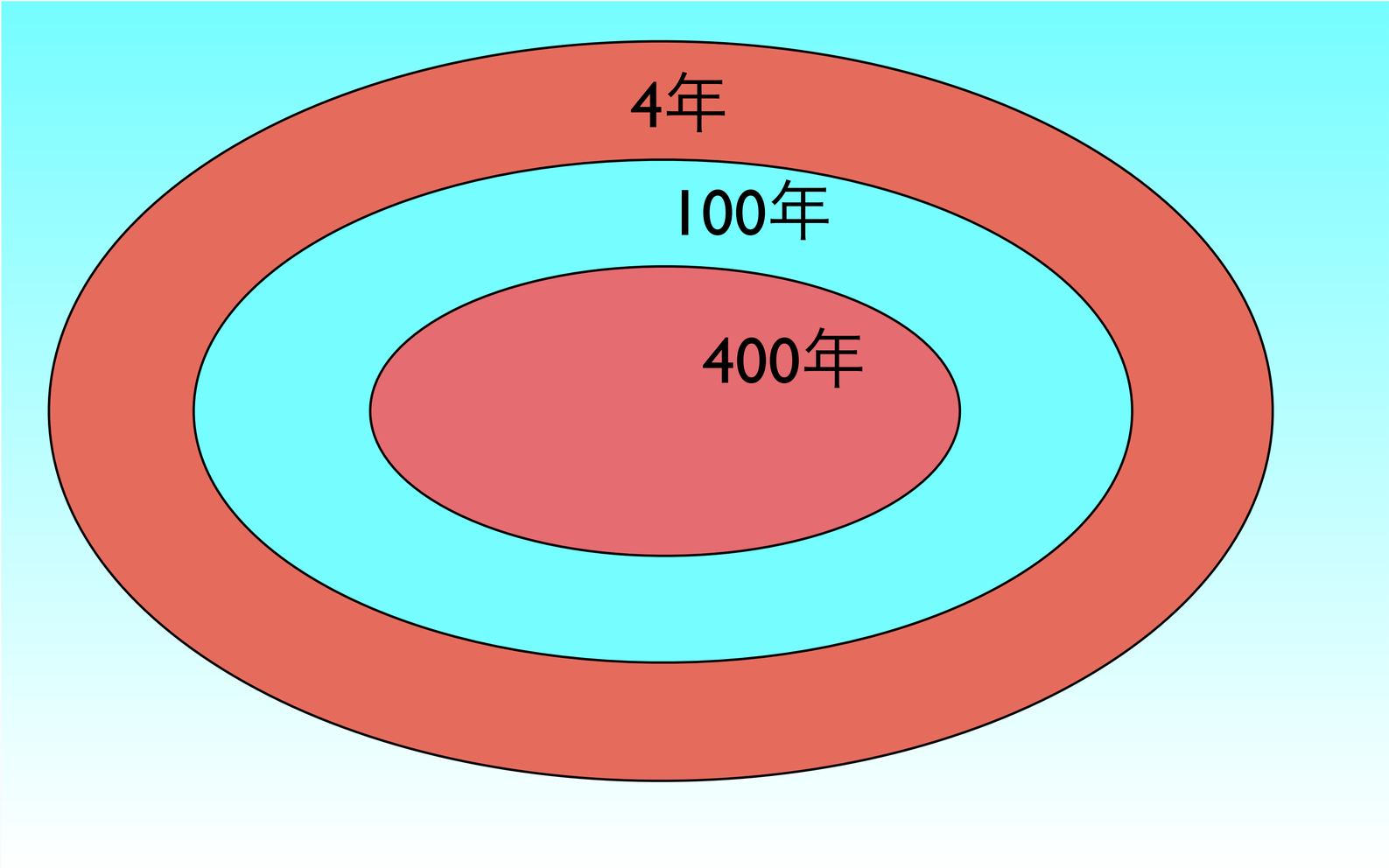
    (value:=2) == 3 **and** (value:= 3) == 3 # valueの値は2

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
  - ただし、4で割り切れる年の中で、100で割り切れる年は、平年　例：1900
  - ただし、100で割り切れる年の中で、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 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

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

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_rectangle(…., 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, C/C++, C#, JavaScript, Julia, Swift, Dart, Rust
- `y = if x > 100 then 10 else 20 end` # Julia, (AppleScript)
- `=IF( x > 100, 10, 20 )` EXCEL

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'

c = ( 'A' if  y >= 80  else ( 'B' if  y >=60
      else ( 'C' if  y >= 40 else  '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, Swift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。これに対応するfor文は、Java/C#/JavaScriptなどでは用意されている

    for (**var** a : nlist) → JavaのPythonに対応するfor文

    for (**var** a **of** nlist) → JavaScriptでPythonに対応するfor文

    for (**var** a **in** nlist) → JavaScriptでは、aは、nlistのインデックスになる

    foreach a **in** nlist → C#の場合

    nlist.forEach(コールバック関数) → Java/JavaScriptのリストに対する繰返しの関数メソッド（C#でも、ForEachメソッドがある）

while文による繰返し

while 継続条件:

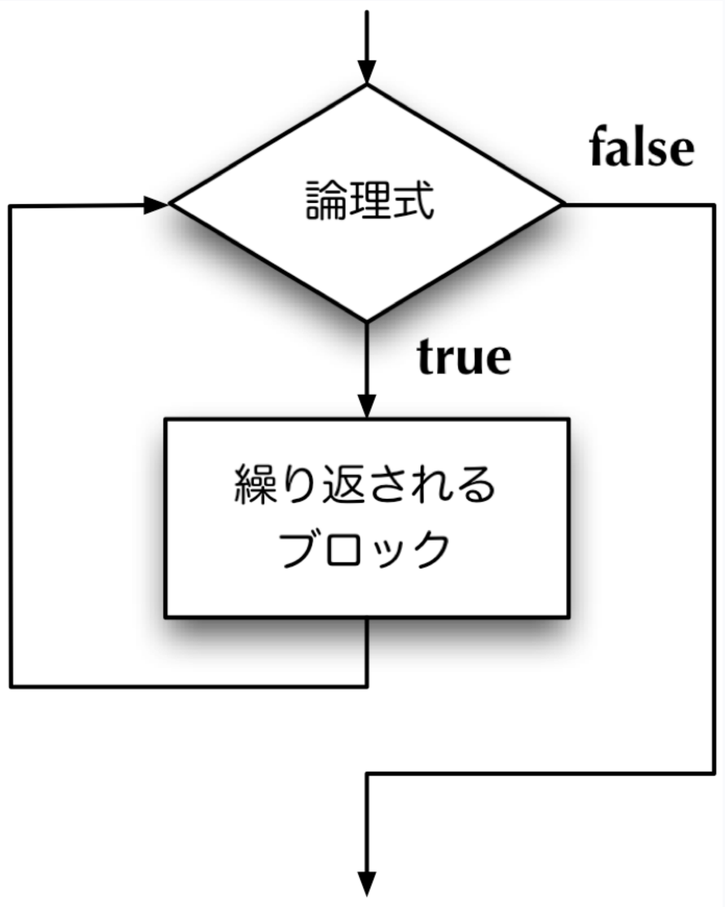
　　繰り返したいこと

[**else**:
　　必ず最後に実行されること
]

- **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、VSCodeではTabとShift+Tab

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
- 変数をいくつ使うべきかは、表わしたいデータの種類による

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

- VSCodeでは、停めたい行の行番号ところでクリックして、停めたい行（その行の実行の前で止まる）を指定する。⇒赤く表示される

- VSCodeの「実行」メニューの「デバッグの開始」のところから実行開始
- 「Python Debugger」で、「開いているPythonモジュールを実行」を選ぶ
- 「デバッグ」バーでは、、Step（ipdbではnext）は 1 行ずつ実行するが、Step Into（ipdbではstep）は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、Step Return（ipdbではreturn）を使う

- 次の止めたい行まで実行したい場合は、Continue（ipdbではcontinue）を使う
- 終了は、「実行」メニューの『デバッグの停止」から（ipdbではexit）

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いように条件を作る
- 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

- １回も繰返さない場合もある

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

繰返しの例題

- if文と組み合わせて、
  - 12ヶ月分の小の月・大の月を求める
  - 1900年から2100年までの閏年の一覧を出す
- 合同数を求める
- 任意の桁でn進数の文字列に変換する

Pythonのリストと他の言語

- Pythonのリストは、配列とリストの両方の性格を持っている。
  - JavaScriptのArrayに似ている
  - Javaだと、ArrayListに該当するが、配列にも該当
  - C/C++だと配列に該当するが、C/C++には、標準ではリストの機能がない
- 複数の値を[ ]で括って持っておくことができる
- それぞれの値は要素と呼ばれるが、Pythonでは要素の型は統一されていなくても良い、値の型を求めるのは、type( )組込み関数を使う
- 例：[1, 2, 3, 4, 5]

  ["A", 34, "文字列", 45.2e-3]

リストと変数

- 変数には、リストを代入することもできる
  - 例： xlist = [2, 3, 4, 5]

- 各要素を取り出したいときには、インデックス式という記法を用いる
- インデックス式で 1 つの要素を取り出したいときは、インデックスは0から始まる
  - 例： xlist = [3, 4, 5, 6]
  - xlist[2]　　# 5が取り出される
  - xlist[0]　　# 3が取り出される

for文とリスト

- **for**文の書式

  for 変数名 **in** リスト:

    繰返したい内容

  [**else**: 必ず最後に実行される]

- 意味

1. リストの各要素が、先頭から順番に、変数に代入される
2. その状態で、「繰り返したい内容」が実行される
3. 最後の要素まで代入されて実行されたら終了

for文の例

```
for  n  in [ 4, 3, 2, 4, 6 ]:
        print( n, end= "  " )
```

- 最初に変数nが用意され、最初の要素が代入される（この場合、整数の4）
- print関数が呼ばれ、nの値が表示される
- 最後の要素まで繰返しを続ける

異種の型を持つリストの場合

- 異種リストの場合は、要素の型を判定するのにtype組込み関数を使う
- 整数はint、実数はfloat、文字列はstr、論理値はbool、複素数はcomplexが返される
- 要素が構造型のリテラルの場合は、リストはlist、タプルはtuple、集合はset、辞書はdict、rangeクラスのオブジェクトはrangeが返される,
- 例：

```
for n in [ "John", 23, True, 45.3 ]:
    if type( n ) == int or type( n ) == str:
        print( n*2, end=" " )
print()
```

Rangeクラスのオブジェクト

- range関数でRangeクラスのオブジェクトが返される
- range(終了数) …0 ～ 終了数-1までの羅列

  例： range(10)…0 ～ 9までの羅列
- range(開始数, 終了数) …開始数～ 終了数-1までの羅列

  例： range(1, 10)… 1～9までの羅列
- range(開始数, 終了数, 差分)…開始数から始まり、差分が足されていった羅列ができる、差分が+の場合、終了数未満の間、差分が–の場合は、終了数より大きい間は羅列が作られる

  例： range(2, 10, 2)…2, 4, 6, 8の羅列

  range(10, 0, -2)…10, 8, 6, 4, 2の羅列

Rangeクラスとリスト

- rangeクラスのオブジェクトによって、作られる羅列は、list関数によって、リストに変換することができる
- 例：

  list(range(1, 9, 2)) ⇒ [1, 3, 5, 7]
- これを用いて、リストとしても利用することが可能になる
- 例：

  numlist = list(range(1, 9, 2))
  print(list[2])　# 5が表示される

Rangeクラスのオブジェクトと演算子・組込み関数

- **in / not in** 演算子
  - 7 **in** range(3, 10) ⇒ True
  - 12 **not in** range(4, 12) ⇒ True
- *による字面展開
  - print(*range(5, 10)) ⇒ 5 6 7 8 9
  - printの引数以外の場所では、外側に()や[], あるいは{}が必要
- rangeクラスのオブジェクトにもインデックス式、スライス式が適用できる。スライス式の結果は、rangeクラスのオブジェクトのまま
  - range(12, 23)[4] ⇒ 16
  - range(11)[4:6] ⇒ range(4, 6)
  - range(1, 10)[4:6] ⇒ range(5, 7)
- 加算演算子はない替わりに、itertoolsのchain関数が使える
  - **from** itertools **import** chain
  - list(chain(range(2, 4), range(6, 9))) ⇒ [2, 3, 6, 7, 8]

Rangeクラスに適用可能な組込み関数

- all / any関数
  - all(range(-4, 2)) ⇒ False,

    any(range(-4, 2)) ⇒ True
- enumerate 関数
  - list(enumerate(range(3, 5))) ⇒

    [(0, 3), (1, 4)]
- len 関数
  - len(range(4, 9)) ⇒ 5
- max / min 関数
  - min(range(3, 10)), max(range(3, 10)) ⇒ (3, 9)
- sorted 関数 → 結果はリストになる
  - sorted(range(4, 1, -1)) ⇒ [2, 3, 4]
- sum 関数
  - sum(range(4, 9)) ⇒ 30
- zip 関数
  - list(zip(range(23, 30), range(4, 7))) ⇒ [(23, 4), (24, 5), (25, 6)]

for文とrange

- **for**文の**in**の後には、Rangeクラスのオブジェクトを指定することが可能になる
- 書式は、**for** 変数 **in** Rangeクラスのオブジェクト:
- 例：

  for *n* **in** range(12): print(*n*) # 0〜11まで表示

  for *n* **in** range(1, 10): print(*n*) # 1〜9まで表示

  for *n* **in** range(5, -2, -2): print(*n*) # 5, 3, 1, -1を表示

総和・階乗を求める

- 総和

```
summ = 0
for i in range( 1, 11 ):
    summ = summ + i
    print( summ )
```

- 階乗

```
factorial = 1
for i in range( 1, 11 ):
    factorial = factorial * i
    print( factorial )
```

総和を求める

- ループ変数の値の変化に注目
- 足し合わされる変数*sum*の変化にも注目する

無名変数による回数繰返し

- 単にn回繰返しをしたいだけで、その間のループ変数を参照しないような場合は、無名変数（_ アンダーバー）を用いることができる
- 例：

```
for _ in range( 10 ):  # 10回繰り返したい
    print( "WOW! ", end="" )
```

リスト・タプル・文字列・rangeのリテラルの要素をインデックスでアクセス

- 組込み関数のlen関数がリストの長さを返してくれる
- len関数とrange関数を組み合わせる
- pythonでは、enumerate関数を使うことも多い
- 例：

```
xlist = [ 2, 3, 4, 5 ]
for  i  in range( len( xlist ) ):
    print( xlist[ i ] )

for i, n in enumerate( xlist ) ):
    xlist[ i ] = n**2
```

リストをfor文で作る（ジェネレータ式）

- **for**文を使って初期化されたリストを作成することができる
  - 書式：'[' 式 **for**文 [**for**文...] [**if**文] ']'
  - 例：[*x* **for** *x* **in** range(1, 10)] ⇒

    [1, 2, 3, 4, 5, 6, 7, 8, 9]

- このときに**if**文や**if**式も利用することが可能
  - 例：[*x* **for** *x* **in** range(1, 10) **if** *x* % 2 == 0] ⇒[2, 4, 6, 8]
  - 例：a = [*n* **if** *n* % 2 == 0 **else** *n**10 **for** n **in** range(1, 11)] ⇒[10, 2, 30, 4, 50, 6, 70, 8, 90, 10]
  - 例：[(*x*, *y*) **for** *x* **in** range(1, 4) **for** *y* **in** range(1, 4)] ⇒ [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]

tuple, dict, setもfor文で作成できる

- listの生成
  - 例：list(n**0.5 **for** n **in** [1, 4, 9, 16, 25]) → [1.0, 2.0, 3.0, 4.0, 5.0]
- tupleの生成
  - 例：tuple(n **for** n **in** range(5)) → (0, 1, 2, 3, 4)
- dictの生成
  - 例：{ n: n**2 **for** n **in** range(3, 8) } ≡ dict([n, n**2] **for** n **in** range(3, 8)) →

    {3: 9, 4: 16, 5: 25, 6: 36, 7: 49}
- setの生成
  - 例：{ n**2 **for** n **in** range(3, 8) } ≡ set(n**2 **for** n **in** range(3, 8)) →

    {36, 9, 16, 49, 25}

展開の*演算子

- 構造型をシーケンスに字面展開するための、*演算子がある
  - 例： *[1, 2, 3] ⇒ 1, 2, 3に展開される
- それぞれの構造型で展開してみる
  - print(*[23.4, 45.6, 56.7]) ⇒ print(23.4, 45.6, 56.7)
  - print(*(23.4, 45.6, 56.7)) ⇒ print(23.4, 45.6, 56.7)
  - print(*{ 23.4, 45.6, 56.7 }) ⇒ print(23.4, 45.6, 56.7)
  - print(*{ "two": 2, "four":4, "five":5 }) ⇒ print("two", "four", "five")
  - print(*"ABCDE") ⇒ print("A", "B", "C", "D", "E")
  - print(*range(3, 7)) ⇒ print(3, 4, 5, 6)

enumerate関数とリスト・タプル・文字列

- 組込み関数のenumerate関数は、リスト・タプル・文字列に対して適用され、そのリスト・タプルの要素（文字列の場合は、各文字）とインデックスの対（タプル）から構成されるenumerateオブジェクトを返してくれる。
- enumerateオブジェクトは、list関数やtuple関数で、リストやタプルにすることができる
- 書式：
  - enumerate(シーケンス型のオブジェクト)
- 例：
  - list(enumerate(["A", "B", "C"])) → [(0, 'A'), (1, 'B'), (2, 'C')
  - list(enumerate("ABCDEF")) → [(0, 'A'), (1, 'B'), (2, 'C'), (3, 'D'), (4, 'E'), (5, 'F')
  - tuple(enumerate([12, 23, 34])) → ((0, 12), (1, 23), (2, 34))

- enumerate(シーケンス型のオブジェクト, start=インデックスの開始番号)も可能（start=は省略可能）
- 例：
  - list(enumerate("ABC", start=1)) → [(1, 'A'), (2, 'B'), (3, 'C')]
  - tuple(enumerate(("ABC", 3+4j, 23.4), 1001)) → ((1001, 'ABC'), (1002, (3+4j)), (1003, 23.4))

enumerateオブジェクトと演算子・関数

- *演算子による字面シーケンス展開（ *enumerate(構造型)で、字面展開）
  - print(*enumerate("ABC")) ⇒ (0, 'A') (1, 'B') (2, 'C')
  - print(*enumerate(enumerate(enumerate("ABC")))) ⇒ (0, (0, (0, 'A'))) (1, (1, (1, 'B'))) (2, (2, (2, 'C')))
  - print(*enumerate((n**2 **for** n **in** range(1, 5)), start=1)) ⇒ (1, 1) (2, 4) (3, 9) (4, 16)
- **in** / **not in** 演算子
  - (3, "E") **in** enumerate("CODE") ⇒ True
  - (1, "A") **not in** enumerate("ABCDE") ⇒ True
- max / min関数
  - min(enumerate("FGHIKLMA")) ⇒ (0, 'F')
  - max(enumerate("EDCBA")) ⇒ (4, 'A')

enumerate関数とfor文

- enumerate関数を用いて、インデックスと一緒にリストを探索することができるfor文を生成できる
- 例：**for** n, value **in** enumerate(["A", "B", "C"]):

```
    print( n, value, end="  " )   # 0 A   1 B   2 C
```

- 例：nlist = [12, 23, 34, 45, 56, 67, 78, 89]

```
for i, n in enumerate( nlist ):
    nlist[ i ] = n**2  # 代入は、インデックス式やスライス式でしかできない
```

- 例：for i, (n, m) in enumerate(enumerate([12, 23, 34], 101), 1):

```
    print( i, n, m, end="   " )   # 1 101 12   2 102 23   3 103 34
```

zip関数とリスト・タプル・文字列

- 組込み関数のzip関数は、複数のリスト・タプル・文字列に適用することができ、各リストの先頭から、順番に要素の対（タプル）から構成されるzipオブジェクトを生成する。
- zipオブジェクトは、list関数やtuple関数でリストやタプルに変換できる。
- 書式：
  - zip(シーケンス型のオブジェクト [, シーケンス型のオブジェクト]...)
- 例：
  - list(zip(['Kobe', 'Kyoto', 'Osaka'], ['神戸', '京都', '大阪']))
    → [('Kobe', '神戸'), ('Kyoto', '京都'), ('Osaka', '大阪')]
  - list(zip("ABCDEF", "あいうえお", "一二三"))
    →[('A', 'あ', '一'), ('B', 'い', '二'), ('C', 'う', '三')]　# 一番少ない個数だけが生成される
- itertoolsモジュールのzip_longestは、一番長い個数のものにあわせて、足りない部分には、Noneを要素としていれられる
  - **from** itertools **import** *
  - list(zip_longest("ABC", [1, 2], (5.2, 1.2, 4.3, 1.1))) → [('A', 1, 5.2), ('B', 2, 1.2), ('C', None, 4.3), (None, None, 1.1)]
- 要素の個数が違うのを許さない場合は、zip(..., strict=True)とする必要がある

zipオブジェクトと演算子・関数

- *演算子による字面シーケンス展開
  - print(*zip("ABC", "DEF", "HIJKLM")) ⇒ ('A', 'D', 'H') ('B', 'E', 'I') ('C', 'F', 'J')
  - print(*zip("ABC", zip("DEF", "XYZ"))) ⇒ ('A', ('D', 'X')) ('B', ('E', 'Y')) ('C', ('F', 'Z'))
  - print(*zip(range(1, 5), (n**2 **for** n **in** range(1,5)))) ⇒ (1, 1) (2, 4) (3, 9) (4, 16)
- **in** / **not in** 演算子
  - ("B", "E") **in** zip("ABC", "DEF") ⇒ True
  - ('Kobe', '京都') **not in** zip(['Kobe', 'Kyoto'], ['神戸', '京都']) ⇒ True
- max / min関数
  - max(zip("DEF", "BAC")) ⇒ ('F', 'C')
  - min(zip(["C", "D", "A"], [2, 4, 3])) ⇒ ('A', 3)

zip関数とfor文

- 例： **for** en, jp **in** zip(['Kobe', 'Kyoto'], ['神戸', '京都']):

  print(en, jp, end=" ") # Kobe 神戸 Kyoto 京都

- 例：**for** num, (name, age) **in** enumerate(zip(['周', '金', '龍', '毛'], [22, 23, 32, 33]), 1):

  print(num, name, age, end=" ") # 1 周 22 2 金 23 3 龍 32 4 毛 33

- 例：**for** num, name, age **in** zip(range(1, 5), ['周', '金', '龍', '毛'], [22, 23, 32, 33])):

  print(num, name, age, end=" ") # 1 周 22 2 金 23 3 龍 32 4 毛 33

while文とfor文との互換性

- **while**文を**for**文で書き直す場合は、ループ変数に一定の変数を足したり、引いたりしている場合に限られる

  n=A
  while n < B : 文; n += C

  → **for** n **in** range(A, B, C): 文

- **for**文を**while**文で書き直す

  for n **in** range(A, B, C): 文

  → n= A

  while n < B: 文; n += C

動く設計

- 初期値と継続条件の設定の仕方による
  - 1 回も実行されない

    for n **in** range(3, 0, 10) : print(n)
  - 1 回も実行されない

    for n **in** range(3, 10, -4) : print(n)

break文

- **break**文に出会うと、一番内側のブロックから脱出する
- 入力のガード（既定値以外の入力をさせないようにする）にも**while**文と共に良く用いられる。
- 途中だけ処理をしたい場合に使われる
- **while True**:

  A

  if 条件式 : **break**

  B
- **break**文で抜けた場合は、**while**文や**for**文に**else**句がついていても、その**else**句のブロックは実行されない

break文とガード

- 必要以外の値を入力しないようにする

- **while True**:
  value = int(input("入力: "))
  if value >= 0: **break** # 0以上なら脱出
  print("正の数を入力のこと", end=" ")

- while文を抜けた段階では、0以上の値であることが保証されている

continue文

- **continue**文は、次の繰返しにいく
- インデントを深くしない場合に使うことが多い

- **while True:**
  if 除外する条件 : **continue**
  繰り返す処理

pass文

- 何もしないで次の文に制御を移す
- 制御構文や関数の定義で、内容が未定の場合に、pass文を使って、見た目をごまかすのに使われる
- 例：

```
for _ in range( 10 ):
    pass   # 10回何かをやる予定

def some_function( arg ): pass   # 中身未定の関数
```

- あとでpassの部分のプログラムを書き換える

assert文

- assert文は、デバッグ用に、式がインタープリタに受け取られるかどうかをテストするために用いられる
  - 書式：**assert**　式 [, 式]
  - 1番目の式がFalseであったり、文法規則にあっていないとエラーが発生する
  - 2番目の式がある場合は、assertで、最初の式が満たされない場合、エラーが発生し、エラーの結果として、2番目の値が返される
  - 例：assert 5==8, "Equality Error" ⇒ AssertionError: Equality Error
  - 更に2番目の式でエラーが発生する場合、その部分でもエラーが発生する
  - 例：**assert** 12==34, 555%2==3x

    ⇒ 3xの部分で、SyntaxError: invalid decimal literalが表示される

try except文

- 例外（実行時に起こるエラー）を処理するために使われる文
- 書式： [ ]は省略可能、except節は1回以上の繰返し可能、

  try: 文またはブロック

  except [式 [**as** 変数名]] : 文またはブロック

  [**else** : 文またはブロック]

  [**finally** : 文またはブロック]

- tryの中で書かれているブロックでエラーが発生したら、exceptの方に制御を移す
- except節の「式」で例外の範囲を指定することが可能
- except節の「式 as 変数名」で、指定された変数に、起こった例外のオブジェクトを代入することが可能
- 例：

  try: *ix* = *s*.index("A")
  except: *ix* = -1

try except文でエラーを受けとる

- except文では、受け取ったエラーを変数に保存し、表示させることができる
  - 例：

```
try:
    value = int( input( "Number: " ) )
except Exception as error:
    value = 0
    print( error )
```

- その後も実行を続けることが可能となる

組込み例外の基底クラス一覧

- BaseException
  - すべての組込み例外の基底クラスになっている（通常は以下のExceptionの方を使う）
- Exception
  - システム終了以外の全ての組込み例外の元となっているクラス
- ArithmeticError
  - 算術例外
- BufferError
  - バッファエラー
- LookupError
  - インデックスなどが範囲を超えたとき
- 具象例外
  - 通常のプログラム実行で起こりうる例外
- OS例外
  - 実行環境のオペレーティングシステムに依存して起こりうる例外
- 参照：
  - https://docs.python.org/ja/3/library/exceptions.html

sqlite3コマンドツール

- コマンドツールは、Mac OS Xでは、python.orgからPython IDLEをダウンロードすると自動的にインストールされる
- Windows およびPython IDLEをインストールしていない場合は以下のページからsqlite toolsをダウンロードする
  - https://sqlite.org/download.html
- Windowsは、インストールするときに、PATHに追加するをチェックする

Macの場合

Precompiled Binaries for Mac OS X (x86)

sqlite-tools-osx-x86-3350500.zip (1.42 MiB)	A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff program, and the sqlite3_analyzer program. (sha3: d2b8b59f8b321f8f19373b4ed0a544b0e8eef7c371d0285acbe5ce9c7dfd6e4d)

Precompiled Binaries for Windows

sqlite-dll-win32-x86-3350500.zip (538.88 KiB)	32-bit DLL (x86) for SQLite version 3.35.5. (sha3: 55b49ce165984865d62918cbd0ad34fa7af82820e4f5d10c899e2191dc63877c)
sqlite-dll-win64-x64-3350500.zip (879.80 KiB)	64-bit DLL (x64) for SQLite version 3.35.5. (sha3: 15ba94e68c6596bb292ba6a4ce303f5beaf731754846d01fd7a55a6edf11c4d5)
sqlite-tools-win32-x86-3350500.zip (1.81 MiB)	A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff.exe program, and the sqlite3_analyzer.exe program. (sha3: 295214b0c3d6bfef3240063 2fc5237e1cb57c07ba0b76e94c98fb0c3fe92075c)

Windowsの場合

sqlite3を利用する

- ターミナルやコマンドツールなどのターミナルソフトウェア上でカレントフォルダを移動
  - 例：cd /Users/ユーザ名/Desktop/Script 2021　Mac/Linuxの場合

    cd C:\Users\ユーザ名\Desktop\Script 2021　Windowsの場合
- 「sqlite3」あるいは「sqlite3 データベース名」で起動
  - 例：sqlite3 test.db
    - そのフォルダにtest.dbというファイルが生成される
- コマンド入力が出てくるので、コマンドを入れる
  - 例：.help　　→　ヘルプ画面が表示
  - .open データベース名
  - .read SQLが書かれたファイル名
  - .quit　　→　終了
  - .tables　　→　データベース内のテーブル一覧
  - .schema　→ スキーマ情報一覧
  - select * from sqlite_master;　　→　データベースの情報一覧

VSCodeからの利用・SQLコマンドマニュアル

- VSCodeでは、機能拡張でSQLiteを実行するための機能拡張がある。

- sqlite3か ページを参照
  - http://rktsqlite.osdn.jp
  - https://www.sqlite.org/lang.html

python3からsqlite3を利用する

- pythonをインストールすれば、標準でsqlite3のライブラリがインストールされている
- データベースに接続して、カーソルを作る
  - 例：**import** sqlite3

    conn = sqlite3.connect("test.db")　# 接続

    cur = conn.cursor( )　# カーソルを作る
- カーソルに対して、execute( )関数を呼ぶと実行ができる
  - 例：cur.execute("select * from sqlite_master")

実行結果の受け取り

- sqlite3では、select文などのSQLの実行結果を、タプルのリストとして返してくる
- for文で 1 行ずつ受け取る
  - 例：**for** row **in** cur.execute("select * from sample"):
    print(row)
- なお、データの更新や追加、削除などテーブルに変更を加えた場合は、最後にSQLでcommitコマンドを発行する必要がある
  - 例：cur.execute("commit")

try except文で実行時例外をキャッチ

- sqlite3側で、エラーが発生したら、それをtry except文で受け取る
  - 例：

```
try:
    result = cur.execute( command )
except Exception as error:
    print( error )      #    errorを表示する
```

raise文

- エラーを送出させる
- 書式：**raise** [式 [**from** 式]]
- 最初の式で示される例外クラスのエラーを送出する
- 例:

```
raise Exception("Some error occurred")
```

match文（Python 3.10から）

- Python以降のSwiftやRust, Juliaなどのプログラミング言語に導入されてきたmatch文がPythonにもフィードバックされて使えるように仕様がアップデートされた。
- 基本的には、C/C++言語のswitch文の発展系になっている。
- 基本文法は以下のようになっている
  - **match** 式:

    case 該当する式: ブロック

    :

    case _: ブロック ＃_は、それ以外のすべてに該当する
  - 最後の **case _**:の節がない場合は、何もしない
  - 該当する式を複数指定する場合は、|で区切る、範囲指定はガードで行なう

match文の例（1）

- 基本的なmatch文の例

```
def errorMessage( status ):
    match status:
        case 400:    return "Bad request"
        case 404:    return "Not found"
        case 418:    return "I'm a teapot"
        case  _:     return None


m = int( input( "Month: " ) )
match m:
    case 2 | 4 | 6 | 9 | 11:
        print( f"Small moon :{m}" )
    case _:
        print( f"Big moon: {m}" )
```

- タプルに対しての使うmatch文の例

```
# point は2つの要素から構成
  される (x, y) の形のタプル

match point:
    case (0, 0):    print("Origin")
    case (0, y):    print(f"Y={y}")
    case (x, 0):    print(f"X={x}")
    case (x, y):    print(f"X={x}, Y={y}")
    case _:  raise ValueError("Not a point")
```

match文とクラス

- クラスが次のように定義されているとする

```
class Point:
    def __init__( self, x: int, y: int ):
        self.x, self.y = x, y
```

- match文では、次のようにオブジェクトのインスタンス変数を指定して照合できる

```
def location(point):
    match point:
        case Point(x=0, y=0):  print("Origin is the point's location.")
        case Point(x=0, y=y):  print(f"Y={y} and the point is on the y-axis.")
        case Point(x=x, y=0):  print(f"X={x} and the point is on the x-axis.")
        case Point():          print("The point is located somewhere else on the plane.")
        case _:                print("Not a point")
```

match文とネストされたパターン

- リストの要素のオブジェクトを指定することができる
- 例：

```
match pointlist:
    case []:  # 空リスト
        print( "リストにPointがありません" )
    case [Point(0, 0)]:  # リストに要素が1つだけで、しかも0, 0
        print( "リストに要素が 1 つだけで、原点を指しています" )
    case [Point(x, y)]:  # リストに要素が1つだけ
        print( f"座標が {x}, {y}の1つの要素がリストにあります" )
    case [Point(0, y1), Point(0, y2)]: # リストに要素が2つだけ
        print( f"リストに2つの要素があり、両方共Y軸上にあり、そのY座標は {y1}, {y2} です" )
    case _:
        print( "それ以外の要素があります" )
```

match文と要素のワイルドカード

- 要素を指定する際に、ワイルドカードの_を利用することができる
- 例：

```
match test_tuple:
    case ( 'warning', code, 40 ):
        print( "A warning has been received." )
    case ('error', code, _ ):
        print( f"An error {code} occurred with three element tuple." )
    case ('error', code, *_ ):
        print( f"An error {code} occurred." )
```

- サブパターンをasで変数に代入しておくことができる
- 例：

```
case (Point(x1, y1), Point(x2, y2) as p2): ...
```

match文とガード、範囲指定

- case節の後に、ifをつけて、照合についてガード（制限）を掛けることができる
- 例：

```
match point:
    case Point(x, y) if x == y:
        print(f"The point is located on the diagonal Y=X at {x}.")
    case Point(x, y):
        print(f"Point is not on the diagonal.")
```

- ガードを用いて範囲指定をすることが可能になっている
- 例：

```
match value:
    case a if a in range( 1, 101 ):  print( f"{a} is within 100" )
    case a: print( f"{a} is over 100" )
```