オブジェクト指向プログラミング

第５回

箕原辰夫

リスト

- 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
  - 例：[1, 2, 3, 4, 5]

    ["A", 34, "文字列", 45.2e-3]

リスト・タプル・文字列と変数

- 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、インデックスとスライスという記法を用いる（詳細は後のスライドで説明する）
- 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
```

スライス

- リスト[最初 : 終わりの次]
  - 部分的なリストが生成される
  - 例：alist[4: 7]
- リスト[: 終わりの次]
  - 最初の要素の位置は、0と仮定される
  - 例：alist[: 7]
- リスト[最初:]
  - 指定された最初の位置から、最後までになる
  - 例：alist[4:]
- リスト[-インデックス]
  - 最後の要素の次から、順次インデックスの値が引かれていく
  - 例：alist[-5], alist[-5:]

スライス式

- スライスで複数の要素を取り出すことができる
- 結果は、新たなリストとして返される
- 例：
  - numlist[2:3] # 1個の要素の場合でもリストとして返す
  - numlist[5:9] # ５番目から９番目の前まで
  - numlist[: 5] # 最初の５個の要素
  - numlist[-5:] # 最後の５個の要素
  - numlist[2:7:2] # ２番目から６番目まで、１つ飛ばし
  - numlist[::-1] # 逆順に取り出す
  - numlist[::-2] # １つ飛ばしで逆順
  - numlist[0:-1:-2] は動かない
  - numlist[1:-1][::-2]

for文とリスト・タプル・文字列

- **for**文の書式

  for 変数名 **in** リスト または タプル または 文字列:

  繰返したい内容

  [**else**: 一度も実行しなくても、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

```
for c in "いろはにほ":
        print( c, end= "  " )
```

- 最初に変数cが用意され、最初の1文字が代入される（この場合、"い"）
- print関数が呼ばれ、cの値が表示される
- 最後の文字まで繰返しを続ける

異種の型を持つリスト・タプルの場合

- 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
- 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、終了数に達したら終わり、差分がマイナスの値の場合は、終了数より大きい間は羅列が作られる
  例： range(1, 10, 2)…1, 3, 5, 7, 9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：

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

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
```

総和を求める・カウントする

- 総和を求めるための変数を用意する

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

- 特定の値をカウントするための変数を用意する

```
count = 0
for n in range( 200, 301 ):
   if n % 3 == 0: count += 1
print( count )
```

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*summ*の変化にも注目する

リストのインデックスで要素をアクセスしたい

- 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
- 例：

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

enumerate関数とリスト

- 組込み関数のenumerate関数は、リストに対して適用され、そのリストの要素とインデックスの対（タプル）から構成される新しいリストを返してくれる。
- 例：list(enumerate(["A", "B", "C"]))

  → [(0, 'A'), (1, 'B'), (2, 'C')]
- enumerate関数は、enumerateオブジェクト返してくるので、list, tuple, set, dictでそれぞれの型に変換する必要がある
  - enumerate([1, 2, 3]) → <enumerate object at 0x105940ae0>
  - tuple(enumerate([1, 2, 3])) → ((0, 1), (1, 2), (2, 3))
  - list(enumerate([1, 2, 3])) → [(0, 1), (1, 2), (2, 3)]
  - set(enumerate([1, 2, 3])) → {(0, 1), (1, 2), (2, 3)}
  - dict(enumerate([1, 2, 3])) → {0: 1, 1: 2, 2: 3}

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

```
for i, n in enumerate( nlist ):
    if n%2==1:   nlist[ i ] = n * 2
# 結果：[12, 46, 34, 90, 56, 134, 78]
```

- enumerate(リスト, start=開始番号)で、開始番号を指定できる

zip関数とリスト

- 組込み関数のzip関数は、複数のリストに適用することができ、各リストの先頭から要素の対のリストを生成できる。
- 例：list(zip(['Kobe', 'Kyoto', 'Osaka'], ['神戸', '京都', '大阪']))

  → [('Kobe', '神戸'), ('Kyoto', '京都'), ('Osaka', '大阪')]
- zip関数とfor文を組み合わせて、複数のリストを先頭から探索するようにすることができる
- 例： **for** en, jp **in** zip(['Kobe', 'Kyoto'], ['神戸', '京都']):

  print(en, jp)

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)

リスト・タプル・集合をfor文で作る

- **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] あるいは [式 **for**文 **if**文] あるいは[式 **for**文 **for**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]
  - 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]

リストをfor文で作る（続き）

- for文とif式を組み合わせた場合
  - ループ変数の値によって、要素の値の計算を変えることができる
  - 例：[n **if** n <= 3 **else** n+10 **for** n **in** range(1, 7)]
    ⇒ [1, 2, 3, 14, 15, 16]
- for文とif文を組み合わせた場合
  - ループ変数の値によって、要素の値の出力を抑制することができる
  - 例：[n**2 **for** n **in** range(1, 10) **if** n**2 > 30 **and** n**2 < 70]
    ⇒ [36, 49, 64]
- for文とfor文を組み合わせた場合
  - 周期的な値の変更を要素の値に加味することができる
  - 例：[n+m **for** n **in** range(5, 20, 5) **for** m **in** range(1, 4)]
    ⇒ [6, 7, 8, 11, 12, 13, 16, 17, 18]

リストをfor文で作る（更に続き）

- for文とenumerate関数を用いる
  - 例：[i * n **for** i, n **in** enumerate(range(1, 20, 3), start=2)]
  - ⇒ [2, 12, 28, 50, 78, 112, 152]
- for文とzip関数を用いる
  - 例：[m * n **for** m, n **in** zip(range(1, 20, 3), range(5, 50, 7))]
  - ⇒ [5, 48, 133, 260, 429, 640, 893]
- for文と整数除算・剰余を用いる
  - 例：[n//3*6 + n%3 **for** n **in** range(12)]
  - ⇒ [0, 1, 2, 6, 7, 8, 12, 13, 14, 18, 19, 20]

tuple, dict, setもfor文で作成できる

- 一般にジェネレータ式（generator expression）と呼ばれる
- 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) } あるいは 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) } あるいは set(n**2 for n in range(3, 8)) →
    {36, 9, 16, 49, 25}

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値を足したり、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

break文

- **break**文に出会うと、一番内側の繰返しのブロックから脱出する
- for文やwhile文は、break文で脱出した場合、else句がついていた場合は、そのブロックの実行はされない
- 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 脱出のための条件式 : break
    B
```

A→B→ A→B→ A→B→… A→B→A

break文と入力のガード

- 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

break文とelse句

- break文で抜けるとelse句は無視される
  - 例：

```
for n in range( 10 ):
    print( n, end=" " )
    if n > 7 : print(); break
else:
    print( " 終 " )
```

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

```
for _ in range( 10 ):
    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文とeval関数

- assert文
  - 書式：**assert** 式
- 式がPythonの構文規則にあっているかどうか、および条件式などの場合は、Trueに評価されるかどうかをチェックする
- 構文規則にあっていないと、エラーを発生する
- 例：

  x = 14
  assert x*12
  assert x==13 ⇒ AssertionErrorになる

- eval組込み関数
  - 書式：eval("Pythonの式")
- 文字列中に書かれた、Pythonの式を評価して、結果を返す
- 例：

  eval("x==13") ⇒ Falseに評価される（x=14のとき）

-

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：

  try: 文またはブロック
  except [式]: 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の文またはブロックを実行する、エラーが起こったらexceptの文またはブロックをする
- exceptの例外ですべて引っ掛からなかった場合は、elseの文またはブロックを実行する
- finallyは、エラーがあってもエラーが起こらなくても、最後に必ず実行する
- 例：

  try: *ix = 45 // 0*
  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

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++言語系のswitch文の発展形になっている。
- 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

```
match status:
    case 400:    return "Bad request"
    case 404:    return "Not found"
    case 418:    return "I'm a teapot"
```

```
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文とネストされたパターン

- リストの要素の細かな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 [(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 [(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 [(0, y1), (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、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." )
```

- 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 ((x1, y1), (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 (x, y) if x == y:
        print(f"The point is located on the diagonal Y=X at {x}.")
    case (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" )
```