Python 問題(2)

＜for 文による検索＞

enumerate[1]関数を使うと、検索した各要素にインデックスが付けられます。
次の例は、リスト interable に格納された文字 a,b,c に対して、インデックスを付けて表示するコード。

```
iterable = ['a', 'b', 'c']

for index, value in enumerate(iterable):
    print(index, value)
```

表示される結果はつぎのようになる。

```
0 a
1 b
2 c
```

enumerate 関数は、オプションで引数「start=」を持っている。これを指定すれば開始するインデックス番号を指定できる。

```
iterable = ['a', 'b', 'c']
for index, value in enumerate(iterable, start=1):
    print(index, value)
```

◆for else 文

繰り返し処理の中で、ある条件が成立しなかった場合だけ実行したい処理がある場合、for else 構文を使う。for ブロック内で break 文が実行されなかった場合、else ブロック内の処理 B が実行される。

```
文法：
for 繰り返し条件:
   ・・処理 A・・
   break
else:
    ・・処理 B・・
```

[1] 列挙する、番号づける。ɪn(j)úːmərèɪt

同じ働きをするものに、while else 構文がある。

次の問題は、リストにある名前から、指定された名前がある場合はインデックス番号を、そうでない場合には「見つかりませんでした」と表示する。

```
#問題４
def find_person(people, target):
    found = False
    for i, person in enumerate(people):
        if person == ア:
            print(f"{target}が i == {i}で見つかりました。")
            found = True
            イ
        ウ:
            print(f"{target} は見つかりませんでした。")

find_person(['Wilma', 'Woof', 'Wally'], 'Wally')
find_person(['Wenda', 'Odlaw', 'WatcherA', 'WatcherB'], 'Whitebeard')
```

ターゲットを発見したフラグとして変数 find を使う。変数 found の初期値は「False」（未発見）

関数の定義 def find_person(people, target):と
呼び出しの書き方 find_person(['Wilma', 'Woof', 'Wally'], 'Wally')に注目しよう！

正解：ア：target
　　イ：break
　　ウ：else

＜while 文の使い方＞

while 文は、簡単なようで難しい。
また、読みづらい、絡まったコード（「スパゲッティ・コード」と呼ばれる）を作る原因にもなるので、最近のプログラミングでは「while 文の使用は避けるように」とされてきた。しかし、Python 言語や他の比較的古い源流を持つ言語の普及とともに while 文が復活してきた。なので、今となっては避けて通れない道となってしまった。

そこで、while についてしっかりと学習する必要がおこる。

◆while ループ

while ループは、指定された条件が真（True）である間、コードブロックを繰り返し実行するために使用される。

条件式が True の間、ループ内の「コードブロック」が実行され、条件式が False になった時点で、ループは終了する。

文法：

```
while 条件式:
    # 実行するコード
```

・基本的なループ：

```
count = 0
while count < 5:
    print(count)
    count += 1
```

・無限ループからの脱出： break

ユーザーが「exit」を入力すると、break ステートメントによりループから抜けだす。

```
while True:
    user_input = input("Enter 'exit' to quit: ")
    if user_input == 'exit':
        break
```

・条件付きの継続：

```
count = 0
while count < 10:
    count += 1
    if count % 2 == 0:
        continue
    print(count)
```

この例では、count が「偶数」の場合、continue ステートメントがループの残りの部分をスキップして、次の繰り返しに直接進みます。その結果、「奇数のみ」が出力される。

◆append 末尾に追加

append メソッドはリストの「末尾」に要素を追加する。

例：リストの最後に「5」を追加するコード

```
my_list = [2, 3, 4]
my_list.append(5)
```

逆に、リストの先頭に追加する場合は、insert メソッドを使う。
これは、リストのインデックスのある位置に「挿入」するメソッドです。

文法：
リスト名.insert(インデックス番号, 挿入要素)

insert()でインデックスを指定すればどこにでも挿入ができる。

```
my_list = [2, 3, 4]
my_list.insert(0, 1)
```

◆reverse　リストの逆順を返す

reverse()はリストメソッドで、リストの逆順を返します。

たとえば、リストが[1, 2, 3, 4, 5]であった場合、その結果は 5，4，3，2，1 が返されます。

```
my_list = [1, 2, 3, 4, 5]
my_list.reverse()
print(my_list)
```

場合によっては、リストの中身はそのままに保ったまま、逆順を返してもらいたい場合もあります。

そのような場合には、reversed()を使い、リストの中身はそのままに、出力だけを入れ替えます。

```
my_list = [1, 2, 3, 4, 5]
new_list = list(reversed(my_list))
print(new_list)
```

この場合、new_list は「5，4，3，2，1」を表示するが、もとの my_list の内容は変更されない。

◆join　結合

join()は、リスト内の要素を結合させて出力します。

```
文法：
separator = '区切り文字'
my_list = ['要素 1', '要素 2', '要素 3'・・・]
joined_string = separator.join(my_list)
```

たとえば、join を使って、年、月、日を日本式に「/」（スラッシュ）で区切って表示するコード。

```
date_parts = ['2024', '01', '14']
date_string = '/'.join(date_parts)
print(date_string)
```

join()で、’’.join()の場合は、要素を区切り文字なしで連結する。

```
#問題 5
def to_binary_str(num):
    digit_list = []
    while num >0:
        digit_list.append(str(ア))
        num = int(num / 2)
    digit_list.reverse()
    return ''.join(digit_list)

print(to_binary_str(19))
```

(解説)

to_binary_str 関数は、整数 num を引数として受け取り、その整数を 2 進数表現（バイナリ形式）の文字列に変換して返します。

1. digit_list という空のリストを作成します。このリストは後に 2 進数の各桁を格納するために使用する。
2. while ループを使用して、引数 num が 0 より大きい間、次の処理を繰り返します：
3. num % 2 を計算して、num の 2 進数表現の最下位桁（最後の桁）を取得します。
この桁を文字列に変換し、digit_list に追加します。
4. num を 2 で割り、整数部分のみを新しい num 値として更新します。
5. while ループが終了したら、digit_list に格納されている桁は逆順になっていますので、reverse() メソッドを使用して桁の順序を正しい 2 進数表現に修正します。
6. 最後に、join() メソッドを使用して digit_list の各要素（桁）を結合し、一つの文字列にします。

正解：ア：num%2

＜再帰＞

「**再帰**」とは自分自身を呼び出す関数のことです。

例えば、階乗の計算のコード

```
def factorial(n):
    # 基底ケース：0! = 1
    if n == 0:
        return 1
    # 再帰ステップ：n! = n × (n-1)!
    else:
        return n * factorial(n - 1)

# 例として 5! を計算
print(factorial(5))
```

（**問題**）次のコードは、文字列「hello」を再帰を使って反転させて表示するものです。

```
#問題６
def reverse_string(s):
    if len(s) <= 1:
        return s
    else:
        return reverse_string(s[1:]) + ア

print(reverse_string('hello'))
```

（**解説**）

関数 reverse_string は、文字列「ｓ」の長さが 1 以下の場合（基底ケース）、文字列をそのまま返します。これは再帰の終了条件。文字列の長さが 1 より大きい場合（再帰ステップ）、関数は自身を最初の文字を除いた残りの文字列に対して再帰的に呼び出し、その結果の末尾に最初の文字を追加します。

さらに詳しく、

文字列スライシング（s[1:]）：この部分は文字列「ｓ」の最初の文字を除いた残りの部分を取り出す。たとえば、「ｓ」が 'hello' の場合、s[1:] は'ello'になる。

再帰関数の呼び出し（reverse_string(s[1:])）：ここでは、スライスされた文字列（最初の文字を

除いた残りの部分）を `reverse_string` 関数に渡しています。このステップで関数は自身を再帰的に呼び出し、小さな部分文字列を反転させます。

文字列の連結`(+ s[0])`:最後に、再帰的に反転された文字列の末尾に、元の文字列の最初の文字（`s[0]`）を追加します。`s[0]`は元の文字列「 s 」の最初の文字です。

結果として、この式は文字列「 s 」の最初の文字を最後に移動させ、残りの部分に対して再帰的に同じ操作を行います。このプロセスは、文字列の長さが 1 以下になるまで続く（これが基底ケース）。この方法により、文字列は最終的に反転されます

正解：**ア**：`s[0]`

＜必要な小数点で示す＞

小数点以下 3 桁で表わす。その際に、「. 」の代わりに「,」（カンマ）で小数点を表わす。

```
#問題 7
def format_float(number):
    if type(number) is not float:
    raise TypeError("引数のタイプは float ではありません")
     number = str(number)
    integer_part, float_part = number.split(".")
    return ",".join([integer_part, float_part[0:3]])

print(format_float(3.1415926536))
print(format_float(1.5))
```

（解説）

関数 format_float の最初の if 文では、number の「型」をもとめ「is not」[2]で左右が同じかを調べ、float 型でない場合に True を返します。

raise ステートメントは、typeError の例外を発生させる。そして、エラーメッセージを発生させる。

次の number = str(number)では、型変換（キャスティング）を行っています。ここでは、変数 number が持つ値を文字列（str）型に変換し、同じ変数 number にその結果を再代入しています。

number を 「.」（ピリオド）を基準にして分割（split）し、その結果を 2 つの変数 integer_part と float_part に代入している。

2 つの文字列 integer_part と float_part[0:3]を「,」（カンマ） で結合している。float_part[0:3]は、インデックス番号で 0 から 3−1 まで、つまり、最初の 3 文字文を取得しています。その上で、integer_part（整数部分）と結合している。

123.456.789,00 €

[2] is not はれっきとした演算子。しかし、「!=」が推奨される

<クラスの基本>　65

```
#問題 8
ア Person:
    def __イ__(self, last_name, first_name, age):
        self.last_name = last_name
        self.first_name = first_name
        self.age = age

    def description(self):
        return self.last_name + " " + self.first_name + "," +
                                              str(self.age) + "歳です"

mika = Person("タナカ", "ミカ", 29)
print(mika.description())
```

正解：　ア：class
　　イ：__init__

　クラスの定義と基本的なメソッドの働きです。
クラスのインスタンスを初期化するコードは「__init__」メソッドの中に書きます。

＜継承＞

次のコードを実行した場合の結果を求めなさい。

```
#問題 10
class Person:
    def __init__(self, last_name, first_name):
        self.last_name = last_name
        self.first_name = first_name

    def __str__(self):
        return "{0}{1}".format(self.last_name, self.first_name)

class Composer(Person):
    def __init__(self, last_name, first_name, number):
        super().__init__(last_name, first_name)
        self.number = number

    def __str__(self):
        return "Composer {0}.{1}, {2} {1}".format(self.number, self.last_name,
self.first_name)


music = Composer("Mozart", "Amadeus", "Sym.40")
print(music)
```

クラス Composer はクラス Person を**継承**している。
さらに、クラス Compoer は__str__をオーバーライド（上書き）している。

◆__str__

特殊メソッドのひとつで文字列表現を定義するために使われる。str()関数や print()が呼び出されたときに自動的に呼び出される。

```
文法：
class MyClass:
    def __str__(self):
        # このオブジェクトの文字列表現を返す
        return "文字列表現"
```

例：名前と年齢の属性を持つ Person クラスに__str__メソッドを実装する。

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"名前: {self.name}, 年齢: {self.age}"
```

このクラスのインスタンスが str()か print()に渡されたとき、__str__メソッドが呼び出される。

```
person = Person("太郎", 30)
print(str(person))  # 名前: 太郎, 年齢: 30
print(person)       # 名前: 太郎, 年齢: 30
```