Pythonサイコロゲームで学ぶ！基礎からオブジェクト指向プログラミング（1/3）

この資料では、簡単なサイコロゲームを作りながら、Pythonプログラミングの基礎からオブジェクト指向（OOP）の考え方までを学びます。

___や日本語になっている部分を埋めてください

ステップ1：サイコロを振って結果を表示する

まずは、コンピュータにサイコロを1回振ってもらい、その結果を表示するプログラムを作りましょう。

考えてみよう（コードの穴埋め）：

```
import random

roll = random.randint(1, 6)  # 1から6の間のランダムな整数を生成
出た目は~ですと出力
```

ヒント：

- `print()` で文字と数値を一緒に出すときは、数値を `str()` を使って文字列に変換する必要があります。

解答例：

```
import random

# サイコロを振る
roll = random.randint(1, 6)
print(”出た目は”+str(roll)+”です。”)
```

ポイント：

- `import random`：`random` モジュールを使えるようにする。

- `random.randint(1, 6)`：1から6までのランダムな整数を生成する。
- 変数 `roll`：生成されたサイコロの目を保存する。
- `print()`：文字列や変数の値を画面に表示する。
- `str(roll)`：数値である `roll` を文字列に変換する。
- `#`：「コメント」と言い、プログラムの説明などを書くために使います。`#` から行末まではプログラムの動作に影響しません。

> コラム print(f”出た目は{roll}です。”) のように書いても同じです。(f-string)。より簡潔な記法なので、覚えよう！

ステップ2：もう一度振るか聞いてみる（if文）

次に、サイコロを振った後、ユーザーにもう一度振るかどうかを尋ね、その答えによって動作を変えるようにしてみましょう。 ここでは `if` 文という、条件によって処理を分ける方法を学びます。

考えてみよう（コードの穴埋め）：

```
import random

# サイコロを振る
roll = random.randint(1, 6)
print(”出た目は”+str(roll)+”です。”)

# もう一度振るかどうか確認
y_or_n = ____(”もう一度振りますか？ (y/n): ”) # ユーザーからの入力を受け取る

もしy_or_nが ”y” だったら:
    roll = random.randint(1, 6)
    print(”二回目の出た目は”+str(roll)+”です。”)
そうではなくもしy_or_nが ”n” だったら:
    print(”ゲームを終了します。”)
それ以外の場合:
    print(”無効な入力です。’y’ または ’n’ を入力してください。”)
```

ヒント：

- ユーザーにキーボードから文字を入力してもらうには `input(”表示するメッセージ”)` を使います。

- `if 条件:`：もし「条件」が正しい場合に、次の行からの処理（インデントされたブロック）を実行します。
- `elif 条件:`：`if` の条件が正しくなく、こちらの「条件」が正しい場合に処理を実行します。
- `else:`：`if` や `elif` のどの条件にも当てはまらなかった場合に処理を実行します。
- `==`：「等しいかどうか」を比べるための記号です。

解答例:

```
import random

# サイコロを振る
roll = random.randint(1, 6)
print("出た目は"+str(roll)+"です。")

# もう一度振るかどうか確認
y_or_n = input("もう一度振りますか？ (y/n): ")

if y_or_n == "y":
    roll = random.randint(1, 6)
    print("二回目の出た目は"+str(roll)+"です。")
elif y_or_n == "n":
    print("ゲームを終了します。")
else:
    print("無効な入力です。'y' または 'n' を入力してください。")
```

ポイント:

- `input()`：ユーザーからのキーボード入力を受け取る。入力されたものは文字列になる。
- `if` 文：条件によって処理を分岐させる。
  - `if 条件1:`
  - `elif 条件2:` (else if の略)
  - `else:`
- インデント（字下げ）：Pythonでは、`if` 文などで処理のまとまりを表すのに重要。

ステップ3: 何度でも振れるようにする（whileループ）

「y」と答えている間は何度でもサイコロを振れるように、繰り返し処理を導入しましょう。 ここでは `while` ループという、特定の条件が満たされている間、処理を繰り返す方法を学びます。

考えてみよう（コードの穴埋め）：

```
import random

ずっと繰り返す:
    roll = random.randint(1,6)
    print(”出た目は”+str(roll)+”です。”)
    y_or_n = input(”もう一度振りますか？ (y/n): ”)
    if y_or_n == ”y”:
        ループの最初に戻って処理を続ける
    elif y_or_n == ”n”:
        print(”ゲームを終了します。”)
        ループを抜ける
    else:
        print(”無効な入力です。’y’ または ’n’ を入力してください。無効なのでゲームを終了します。”)
        ループを抜ける
```

ヒント：

- `while 条件:`：「条件」が正しい間、次の行からの処理（インデントされたブロック）を繰り返します。`True` を条件にすると無限に繰り返します（ループを抜ける処理が必要）。
- `continue`：`while` ループの中で使うと、それ以降の処理を飛ばして、ループの最初（条件判定）に戻ります。
- `break`：`while` ループの中で使うと、ループを強制的に終了します。

解答例：

```
import random

while True:
    roll = random.randint(1,6)
    print(”出た目は”+str(roll)+”です。”)
    y_or_n = input(”もう一度振りますか？ (y/n): ”)
    if y_or_n == ”y”:
        continue
    elif y_or_n == ”n”:
        print(”ゲームを終了します。”)
        break
    else:
        print(”無効な入力です。’y’ または ’n’ を入力してください。無効なのでゲームを終了します。”)
        break
```

ポイント：

- `while` ループ：条件が真の間、処理を繰り返す。
- `True`：「真」を表す特別な値。`while True` で無限ループを作れる。

- `continue`：ループの現在の回を中断し、次の回の先頭へ進む。
- `break`：ループを即座に終了する。

ステップ4: 出た目を記録する（リスト）

これまでのサイコロの出た目をすべて記録し、最後にまとめて表示できるように
しましょう。 複数の値をまとめて管理するには「リスト」を使います。

考えてみよう（コードの穴埋め）：

```
import random

roll_history_list = ____  # 出た目を記録するための空のリストを用意

while True:
    roll = random.randint(1, 6)
    roll_history_list.____(roll)  # 出た目をリストの末尾に追加
    print(”出た目は”+str(roll)+”です。”)
    y_or_n = input(”もう一度振りますか？ (y/n): ”)
    if y_or_n == ”y”:
        continue
    elif y_or_n == ”n”:
        print(”ゲームを終了します。”)
        break
    else:
        print(”無効な入力です。’y’ または ’n’ を入力してください。無効なのでゲームを終了します。”)
        break

print(”これまでの出た目:”, roll_history_list) # 記録したリスト全体を表示
```

ヒント：

- リストは `[]` で作ります。空のリストは `[]` です。
- リストに要素を追加するには、`リスト名.append(追加する要素)` というメソッドを使います。

解答例：

```
import random

roll_history_list = []  # 出た目を記録するリスト

while True:
```

```
    roll = random.randint(1, 6)
    roll_history_list.append(roll)  # 出た目をリストに追加
    print("出た目は"+str(roll)+"です。")
    y_or_n = input("もう一度振りますか？ (y/n): ")
    if y_or_n == "y":
        continue
    elif y_or_n == "n":
        print("ゲームを終了します。")
        break
    else:
        print("無効な入力です。'y' または 'n' を入力してください。無効なのでゲームを終了します。")
        break

print("これまでの出た目:", roll_history_list)
```

ポイント：

- リスト：複数のデータを順番に格納できるデータ構造。`[]` で作成。
- `roll_history_list = []`：空のリストを作成し、変数 `roll_history_list` に代入。
- `roll_history_list.append(roll)`：`roll_history_list` の末尾に `roll` の値を追加する。
- リストの表示：`print(リスト名)` でリストの内容を表示できる。

ステップ5：forループを使って、出た目の履歴を見やすく表示しよう

これまでのステップでは、サイコロを振った結果をリストに保存しました。このステップでは、保存された履歴を `for` ループを使って、より分かりやすく表示する方法を学びます。

考えてみよう（コードの穴埋め）：

```
import random

roll_history_list = []  # 出た目を記録するリスト

while True:
    roll = random.randint(1, 6)
    roll_history_list.append(roll)  # 出た目をリストに追加
    print("出た目は"+str(roll)+"です。")
    y_or_n = input("もう一度振りますか？ (y/n): ")
```

```
    if y_or_n == ”y”:
        continue
    elif y_or_n == ”n”:
        print(”ゲームを終了します。”)
        break
    else:
        print(”無効な入力です。’y’ または ’n’ を入力してください。無効なのでゲームを終了します。”)
        break

print(”これまでの出た目:”)
iを 0 から roll_history_list の長さ - 1 までの範囲で繰り返す:
    「i+1 回目: (roll_history_listのi番目の値)」という形式で出力する
```

ヒント:

- リストの長さを取得するには `len()` 関数を使います。
- `for` ループと `range()` 関数を組み合わせることで、特定の回数だけ処理を繰り返せます。
- リストの要素には `リスト名[インデックス]` でアクセスできます。インデックスは0から始まります。
- f-string `f”文字列 {変数}”` を使うと、文字列の中に変数の値を埋め込めて便利です。

解答例:

```
import random

roll_history_list = []  # 出た目を記録するリスト

while True:
    roll = random.randint(1, 6)
    roll_history_list.append(roll)  # 出た目をリストに追加
    print(”出た目は”+str(roll)+”です。”)
    y_or_n = input(”もう一度振りますか？ (y/n): ”)
    if y_or_n == ”y”:
        continue
    elif y_or_n == ”n”:
        print(”ゲームを終了します。”)
        break
    else:
        print(”無効な入力です。’y’ または ’n’ を入力してください。無効なのでゲームを終了します。”)
        break

print(”これまでの出た目:”)
for i in range(len(roll_history_list)):
    print(f” {i+1}回目: {roll_history_list[i]}”)
```

ポイント:

- `for i in range(len(リスト名)):` : リストの各要素に順番にアクセスするための一般的な方法です。`i` は0から始まり、リストの最後の要素のインデックスまで変化します。

- `リスト名[i]`: リストの `i` 番目の要素を取得します。
- `f-string`: `f"{i+1}回目: {roll_history_list[i]}"` のように、文字列に変数の値を埋め込むことで、見やすい出力を作成できます。`i+1` とすることで、回数表示を1から始めることができます。

> コラム `enumerate` 関数を使うと、リストのインデックスと要素を同時に取得できて便利です。
>
> ```
> print("これまでの出た目:")
> for i, roll in enumerate(roll_history_list, start=1):
>     print(f"{i}回目: {roll}")
> ```
>
> `start=1` を指定することで、インデックス `i` が1から始まります。この方が `i+1` と書くよりもスッキリしますね。

ステップ6: forループを使って、出た目の合計値を計算しよう

リストに保存された出た目の履歴を使って、今度はそれらの合計値を計算してみましょう。ここでも `for` ループが活躍します。

考えてみよう（コードの穴埋め）:

```
import random

roll_history_list = []  # 出た目を記録するリスト

while True:
    roll = random.randint(1, 6)
    roll_history_list.append(roll)  # 出た目をリストに追加
    print("出た目は"+str(roll)+"です。")
    y_or_n = input("もう一度振りますか？ (y/n): ")
    if y_or_n == "y":
        continue
    elif y_or_n == "n":
        print("ゲームを終了します。")
        break
    else:
        print("無効な入力です。'y' または 'n' を入力してください。無効なのでゲームを終了します。")
        break

print("これまでの出た目:")
```

```
for i in range(len(roll_history_list)):
    print(f"{i+1}回目: {roll_history_list[i]}")

total = 0
roll_history_list の中の各要素を roll に取り出すループ:
    total += roll
print("これまでの出た目の合計は:", total)
```

ヒント:

- 合計を計算し始める前に、合計値を保存する変数を `0` で初期化します。
- `for 変数名 in リスト名:` という構文を使うと、リストの各要素を順番に取り出して処理できます。
- `変数 += 値` は `変数 = 変数 + 値` と同じ意味で、変数に値を加算します。

解答例:

```
import random

roll_history_list = []  # 出た目を記録するリスト

while True:
    roll = random.randint(1, 6)
    roll_history_list.append(roll)  # 出た目をリストに追加
    print("出た目は"+str(roll)+"です。")
    y_or_n = input("もう一度振りますか？ (y/n): ")
    if y_or_n == "y":
        continue
    elif y_or_n == "n":
        print("ゲームを終了します。")
        break
    else:
        print("無効な入力です。'y' または 'n' を入力してください。無効なのでゲームを終了します。")
        break

print("これまでの出た目:")
for i in range(len(roll_history_list)):
    print(f"{i+1}回目: {roll_history_list[i]}")

# 合計値を計算
total = 0
for roll in roll_history_list:
    total += roll
print("これまでの出た目の合計は:", total)
```

ポイント:

- **変数の初期化**: 合計やカウントなど、値を累積していく変数は、ループの前に適切な初期値（通常は0や空のリストなど）で初期化することが重要です。

- **`for 要素 in リスト名:`**：リストの各要素を順番に取り出して `要素` という変数に代入し、ループ内の処理を実行します。ステップ5の `range(len())` を使う方法よりもシンプルに書けます。
- **累算代入演算子 `+=`**：`total = total + roll` を短く書いたもので、変数に値を足しこむ際によく使われます。

> コラム Pythonには、リストなどの数値の集まりの合計を簡単に計算できる `sum()` 関数があります。
>
> ```
> total = sum(roll_history_list)
> print(”これまでの出た目の合計は:”, total)
> ```
>
> このように書けば、 `for` ループを使って自分で合計を計算するコードを書く必要がありません。とても便利ですね！

ステップ7: 関数を使って処理をまとめよう

これまでのコードは長くなってきましたね。同じような処理を何度も書いたり、どこで何をしているのか分かりにくくなったりしていませんか？ このステップでは、「関数」という機能を使って、処理を部品のようにまとめ、コードをスッキリさせます。

考えてみよう（コードの穴埋め）：

```
import random

# サイコロを1個振って、出た目を返す関数
def roll_die()-> int:
    # 1から6までのランダムな整数を生成して返す
    return ___

# 「もう一度振りますか？」と聞いて、”y”ならTrue、”n”ならFalseを返す関数
def ask_and_check_if_continue()->bool:
    while True:
        # 「もう一度振りますか？ (y/n): 」と表示し、ユーザーの入力を受け取る
        y_or_n = input(”もう一度振りますか？ (y/n): ”)
        if y_or_n == ”y”:
            # ”y”が入力されたらTrueを返す
            return ___
        elif y_or_n == ”n”:
            # ”n”が入力されたらFalseを返す
            return ___
```

```
        else:
            # それ以外の入力ならエラーメッセージを表示
            print("無効な入力です。'y' または 'n' を入力してください。")

# 出た目の履歴リストを受け取って、見やすく表示する関数
def print_history(history_list):
    print("---これまでの出た目---")
    # enumerateを使って、インデックスと値を同時に取得して表示
    for i, roll in enumerate(history_list, start=1):
        print(f"{i}回目: {roll}")
    print("--------------------")
    # sumを使って合計値を計算して表示
    print(f"合計: {sum(history_list)}")

# サイコロゲームのソロプレイを実行する関数
def solo_play():
    roll_history_list = []
    while True:
        # roll_die関数を呼び出してサイコロを振る
        roll = ___
        roll_history_list.append(roll)
        print(f"出た目は {roll} です。")

        # ask_and_check_if_continue関数を呼び出して続けるか確認
        if not ___:
            print("ゲームを終了します。")
            break

    # print_history関数を呼び出して履歴を表示
    ___(roll_history_list)

# このスクリプトが直接実行されたときに、solo_play関数を呼び出す
if __name__ == "__main__":
    ___
```

ヒント:

- 関数を定義するには `def 関数名():` のように書きます。
- 関数の中で行う処理は、インデント（字下げ）します。
- 関数を呼び出すには `関数名()` のように書きます。
- 関数から値を返すには `return 値` のように書きます。
- `if __name__ == "__main__":` は、このPythonファイルが他のファイルから読み込まれたのではなく、直接実行された場合にのみ、その中の処理を実行するためのおまじないです。

解答例:

```
import random
```

```
# サイコロを1個振って、出た目を返す関数
def roll_die()->int:
    # 1から6までのランダムな整数を生成して返す
    return random.randint(1, 6)

# 「もう一度振りますか？」と聞いて、”y”ならTrue、”n”ならFalseを返す関数
def ask_and_check_if_continue()->bool:
    while True:
        # 「もう一度振りますか？（y/n）: 」と表示し、ユーザーの入力を受け取る
        y_or_n = input(”もう一度振りますか？（y/n）: ”)
        if y_or_n == ”y”:
            # ”y”が入力されたらTrueを返す
            return True
        elif y_or_n == ”n”:
            # ”n”が入力されたらFalseを返す
            return False
        else:
            # それ以外の入力ならエラーメッセージを表示
            print(”無効な入力です。’y’ または ’n’ を入力してください。”)

# 出た目の履歴リストを受け取って、見やすく表示する関数
def print_history(history_list):
    print(”---これまでの出た目---”)
    # enumerateを使って、インデックスと値を同時に取得して表示
    for i, roll in enumerate(history_list, start=1):
        print(f”{i}回目: {roll}”)
    print(”--------------------”)
    # sumを使って合計値を計算して表示
    print(f”合計: {sum(history_list)}”)

# サイコロゲームのソロプレイを実行する関数
def solo_play():
    roll_history_list = []
    while True:
        # roll_die関数を呼び出してサイコロを振る
        roll = roll_die()
        roll_history_list.append(roll)
        print(f”出た目は {roll} です。”)

        # ask_and_check_if_continue関数を呼び出して続けるか確認
        if not ask_and_check_if_continue():
            print(”ゲームを終了します。”)
            break

    # print_history関数を呼び出して履歴を表示
    print_history(roll_history_list)

# このスクリプトが直接実行されたときに、solo_play関数を呼び出す
if __name__ == ”__main__”:
    solo_play()
```

ポイント：

- **関数**を使うと、特定の処理をひとつの「部品」としてまとめることができます。

- 関数には **名前** をつけ、その名前を使って何度も呼び出すことができます（ **再利用性** ）。
- コードの役割分担が明確になり、 **見通しが良く** なります。
- `def` キーワードを使って関数を定義します。
- 関数に情報を渡すことを「 **引数（ひきすう）** を渡す」と言います。例: `print_history(roll_history_list)` の `roll_history_list`
- 関数から結果を受け取ることを「 **戻り値（もどりち）** を受け取る」と言います。例: `roll = roll_die()` の `roll`
- `if __name__ == "__main__":` は、Pythonスクリプトが「プログラムとして直接実行された」場合にのみ、そのブロック内のコードを実行するための決まり文句です。他のPythonファイルから部品として呼び出された場合には実行されません。

> コラム 関数定義の際に、引数や戻り値の「型ヒント」を書くことがあります。例えば `def roll_die() -> int:` のように書くと、「この関数は整数(int)を返す」という意味になります。これはコードの可読性を高め、間違いを減らすのに役立ちますが、必須ではありません。 また、関数の最初の行に `"""関数の説明文"""` のように三重クォートで囲んで説明文（docstring）を書く習慣もあります。これも他の人がコードを理解しやすくするために役立ちます。