Pythonプログラミング補足資料２：ステップアップ編

この資料は、「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（2/3）」の演習を進める上で役立つ、少し進んだPythonの機能や考え方について解説します。

1. 関数のさらなる活用（ステップ8, 9 に関連）

演習1で関数の基本を学びましたが、ここではもう少し便利な使い方を見ていきましょう。

- **複数の戻り値:** 関数は複数の値を返すことができます。これは `return 値1, 値2, ...` のように記述し、受け取る側も `変数1, 変数2, ... = 関数名()` のように対応させます。 演習2の `one_match` 関数では、プレイヤーの出目、コンピュータの出目、勝者の3つの情報を返しています。

```
def get_player_info():
    name = "Hero"
    level = 10
    hp = 100
    return name, level, hp

player_name, player_level, player_hp = get_player_info()
print(f"名前: {player_name}, レベル: {player_level}, HP: {player_hp}")
# 出力例: 名前: Hero, レベル: 10, HP: 100
```

- **引数のデフォルト値:** 関数の引数には、あらかじめデフォルトの値を設定しておくことができます。関数呼び出し時にその引数が省略されると、デフォルト値が使われます。 演習2の `roll_die(sides=6)` は、`sides` 引数にデフォルト値 `6` を設定しています。

```
def greet(name, message="こんにちは"): # message引数にデフォルト値を設定
    print(f"{name}さん、{message}！")

greet("山田") # messageを省略するとデフォルト値が使われる
# 出力: 山田さん、こんにちは！
greet("佐藤", "こんばんは") # messageを指定するとその値が使われる
# 出力: 佐藤さん、こんばんは！
```

- **型ヒント（Type Hints）:** 必須ではありませんが、関数の引数や戻り値に「どのような型のデータか」を示す情報（型ヒント）を付加することができます。これにより、コードの可読性が向上し、エラーを発見しやすくなることがあります。 演習2の `def roll_die()->int:` は、「この関数は整数(int)を返す」という意味です。 `one_match()->[int, int, str]:` は、整数のリスト、整数のリスト、文字列を返すことを示唆していますが、より正確には `Tuple[int, int, str]` や、Python 3.9以降では `tuple[int, int, str]` と書くのが一般的です（複数の値を返す場合、実際にはタプルという型で返されるため）。

```
def add(a: int, b: int) -> int:
    return a + b

result: int = add(5, 3)
print(result)
```

　型ヒントはあくまで「ヒント」であり、プログラムの実行自体には直接影響しません（型チェックツールを使うと検証できます）。

2. リストの高度な使い方（ステップ10 に関連）

リストは非常に強力なデータ構造で、様々な応用が可能です。

- **リストにリストを格納する（ネストしたリスト）:** リストの要素として、さらにリストを持つことができます。これにより、表のような二次元的なデータや、より複雑な構造のデータを表現できます。 演習2の `match_history` は、各対戦の結果 `[プレイヤー出目, コンピュータ出目, 勝者]` というリストを、さらに大きなリストに格納しています。

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print(matrix[0])    # 出力: [1, 2, 3]
print(matrix[1][1]) # 出力: 5 (2行目2列目の要素)
```

- **リスト内包表記（List Comprehensions）:** 既存のリストや他の反復可能なオブジェクトから、新しいリストを簡潔に作成するための強力な機能です。 演習2では、 `match_history` からプレイヤーの出目だけを取り出す際に使われています。

```
numbers = [1, 2, 3, 4, 5]
# 各要素を2乗した新しいリストを作成
squared_numbers = [x**2 for x in numbers]
print(squared_numbers) # 出力: [1, 4, 9, 16, 25]

# 偶数だけを取り出す
even_numbers = [x for x in numbers if x % 2 == 0]
print(even_numbers)    # 出力: [2, 4]

# 演習2の例
# match_history = [[6, 1, "player"], [3, 5, "computer"], [4, 4, "draw"]]
# player_rolls = [match[0] for match in match_history]
# print(player_rolls) # 出力: [6, 3, 4]
```

`for` ループと `if` 文を組み合わせることもできます。

- **`enumerate()` 関数の活用:** 演習1でも触れましたが、ループ処理でリストのインデックスと要素を同時に取得したい場合に非常に便利です。`start` 引数で開始インデックスを指定できます。

```
fruits = ["apple", "banana", "cherry"]
for i, fruit in enumerate(fruits, start=1):
    print(f"{i}番目の果物: {fruit}")
# 出力例:
# 1番目の果物: apple
# 2番目の果物: banana
# 3番目の果物: cherry
```

- **`zip()` 関数の紹介:** 複数のリスト（や他の反復可能なオブジェクト）の要素を、同時に順番に取り出して処理したい場合に便利です。それぞれのリストの同じインデックスにある要素をペアにしてくれます。

```
names = ["Alice", "Bob", "Charlie"]
scores = [85, 92, 78]

for name, score in zip(names, scores):
    print(f"{name}さんの点数は {score}点です。")
# 出力例:
# Aliceさんの点数は 85点です。
# Bobさんの点数は 92点です。
# Charlieさんの点数は 78点です。
```

リストの長さが異なる場合、短い方のリストの要素が尽きた時点で処理は終了します。

3. 辞書（dict）の基本と活用（ステップ11 に関連）

辞書は、キーと値のペアでデータを格納する非常に便利なデータ型です。リストが順番（インデックス）で要素を管理するのに対し、辞書は一意な「キー」で値を管理します。

- **辞書の作成とアクセス:** 辞書は `{}` を使って作成し、`キー: 値` の形で要素を記述します。値へのアクセスは `辞書名[キー]` で行います。

```
# 作成
player_stats = {"name": "Hero", "level": 10, "hp": 100, "gold": 500}

# アクセス
print(player_stats["name"])  # 出力: Hero
print(player_stats["hp"])    # 出力: 100

# 存在しないキーでアクセスしようとするとエラー (KeyError)
# print(player_stats["mp"])
```

- **辞書への要素の追加と更新:** 新しいキーと値のペアを追加したり、既存のキーの値を更新したりできます。

```
player_stats = {"name": "Hero", "level": 10}

# 追加
player_stats["job"] = "Warrior"
print(player_stats) # 出力: {'name': 'Hero', 'level': 10, 'job': 'Warrior'}

# 更新
player_stats["level"] = 11
print(player_stats) # 出力: {'name': 'Hero', 'level': 11, 'job': 'Warrior'}
```

- **辞書を使った設定管理:** 演習2の `game_config` のように、ゲームの設定値などを辞書でまとめて管理すると、コードの見通しが良くなり、変更も容易になります。

```
game_settings = {
    "difficulty": "normal",
    "sound_volume": 7,
    "show_tips": True
}
if game_settings["difficulty"] == "hard":
    print("ハードモードで開始します。")
```

- **辞書とループ:** 辞書のキー、値、またはキーと値のペアをループで取り出すことができます。

```
player_stats = {"name": "Hero", "level": 10, "hp": 100}

# キーだけを取り出す
for key in player_stats.keys(): # .keys() は省略可能 (for key in player_stats:) でも同じ
    print(key)
# 出力例: name, level, hp (順不同の可能性あり)

# 値だけを取り出す
for value in player_stats.values():
    print(value)
# 出力例: Hero, 10, 100 (順不同の可能性あり)

# キーと値のペアを取り出す
for key, value in player_stats.items():
    print(f"{key}: {value}")
# 出力例: name: Hero, level: 10, hp: 100 (順不同の可能性あり)
```

Python 3.7以降では、辞書の要素の順序は挿入された順序が保持されるようになりましたが、それ以前のバージョンでは順不同でした。

4. 定数とマジックナンバー（ステップ11 に関連）

- **定数 (Constants):** プログラムの中で値が変わらない特定の意味を持つデータには、名前を付けて管理することが推奨されます。Pythonには厳密な意味での「定数（再代入不可）」を定義する仕組みはありませんが、慣習として **大文字のスネークケース**（例: `PLAYER_WIN`）で変数名を記述し、これが変更すべきでない値であることを示します。 演習2では `PLAYER_WIN`, `COMPUTER_WIN`, `DRAW` がこれに該当します。

- **マジックナンバー / マジックストリング:** コードの中に直接書かれた、意味が自明でない数値や文字列のことを指します。

```
# マジックナンバーの例
if status == 0: # 0 が何を意味するかわかりにくい
    print("処理成功")

# 定数を使った改善例
STATUS_SUCCESS = 0
if status == STATUS_SUCCESS:
    print("処理成功")
```

定数を使うことで、コードの可読性が向上し、値の変更が必要になった場合も一箇所修正するだけで済むため、保守性が高まります。演習2の `game_config` のキーとして定数を使っているのは良い例です。

これらの知識を活用して、「さいころ演習２」をさらにスムーズに進めていきましょう！