関数、エラーハンドリング、標準ライブラリ

前回は、「ループ」と「条件分岐」を学び、プログラムに複雑な流れや判断をさせることができるようになりましたね！これで、面倒な繰り返し作業や、状況に応じた処理の切り替えはお手の物になったはずです。

さあ、今回はプログラミングの世界で「整理整頓の達人」とも言える「**関数（かんすう）**」と、Pythonが最初から用意してくれている超便利な道具箱「**標準ライブラリ**」について学びます！ 関数をマスターすれば、あなたの書くプログラムはもっとスッキリ整理され、同じ処理を何度も書く手間が省けます。そして標準ライブラリを知れば、複雑な計算や日付の扱い、ランダムなことまで、まるで魔法のように簡単に実現できるようになります。 AI開発においても、これらの知識は非常に重要です。自分で処理を部品化（関数化）したり、既存の便利なライブラリを使いこなしたりすることで、より高度なプログラムを効率的に組むことができるようになります。 覚えることは少し多いかもしれませんが、一つ一つが強力な武器になります。じっくり、そして楽しみながら進んでいきましょう！

目次

9. 演習
10. まとめと次回予告

---

1. 関数って何？ なぜ作るの？

プログラムの部品化：同じ処理を何度も書きたくない

プログラミングをしていると、「あれ、この処理、さっきも書いたな…」「ここ、ちょっとだけ変えてまた同じようなことしてる…」という場面がよく出てきます。そんなとき、同じようなコードを何度もコピペするのは、手間がかかるし、もし修正が必要になったら全部の箇所を直さないといけなくて大変ですよね。

そこで登場するのが「**関数**」です！ 関数とは、**一連の処理をひとまとめにして名前をつけたもの**、いわばプログラムの「部品」のようなものです。 例えば、「挨拶をする」という処理や、「2つの数値を受け取って合計を計算する」という処理をそれぞれ関数として作っておけば、必要な時にその関数の名前を呼ぶだけで、その処理を実行できるようになります。

> 

関数のメリット：整理整頓、再利用、読みやすさアップ

関数を使うと、たくさんのいいことがあります。

- **整理整頓（構造化）**: 長いプログラムも、機能ごとに小さな関数に分けることで、全体の見通しが良くなり、どこで何をしているのかが分かりやすくなります。
- **再利用性アップ**: 一度作った関数は、プログラムの別の場所や、さらには別のプログラムからでも何度でも呼び出して使えます。同じコードを何度も書く必要がなくなります。
- **読みやすさアップ**: 処理の塊に意味のある名前（関数名）をつけることで、コードが何をしようとしているのかが直感的に理解しやすくなります。
- **保守性アップ**: もし処理内容に変更が必要になった場合、その関数の中だけを修正すればOKです。コピペしたたくさんの箇所を修正する必要がなくなります。

料理に例えるなら、よく作る「野菜を切る」「炒める」といった手順をそれぞれ「レシピカード（関数）」にしておけば、新しい料理を作るときにそのカードを組み合わせるだけで済む、というイメージです。便利ですよね！

---

2. 基本的な関数の作り方と使い方（`def`）

では、実際にPythonで関数をどうやって作るのか見ていきましょう。

`def` で関数を定義する

関数を作ることを「**関数を定義する**」と言います。Pythonでは `def` というキーワードを使って関数を定義します。

基本的な形はこうです。

```
def 関数名():
    # この関数が呼び出された時に実行される処理
    # この部分はインデント（字下げ）されていることに注目！
    print(”関数が呼び出されました！”)
```

- `def`：「これから関数を定義しますよ」という合図のキーワードです。
- `関数名`：あなたが自由に決められる関数の名前です。変数名と同じように、分かりやすい名前をつけるのがコツです（例: `say_hello`, `calculate_sum` など）。
- `()`：関数名の後には必ず丸括弧をつけます。この括弧の中に、後で説明する「引数」を書くことができます。
- `:`（コロン）: 丸括弧の後には必ずコロンをつけます。忘れやすいので注意！
- **インデントされた処理ブロック**: `def` 文の次の行から、この関数が実行する処理を書きます。`if` 文や `for` ループの時と同じように、この部分は必ず**インデント**（半角スペース4つが一般的）します。このインデントされたまとまりが関数の本体です。

関数を呼び出す

関数を定義しただけでは、その中の処理は実行されません。関数を使うには、「**関数を呼び出す**」必要があります。 呼び出しは簡単で、関数名の後に丸括弧 `()` をつけて書くだけです。

```
# まず関数を定義する
def greet():
    print(”こんにちは！”)
    print(”良い一日を！”)

# 関数を呼び出す
greet()  # これで greet 関数の処理が実行される
greet()  # 何度でも呼び出せる
```

実行結果：

```
こんにちは！
良い一日を！
こんにちは！
良い一日を！
```

このように、`greet()` と書くだけで、定義した2行の `print` 処理が実行されます。

何もしない関数 `pass`

時には、「関数の名前と形だけ先に決めておいて、中身の処理は後で書きたい」ということがあります。そんな時、関数の処理ブロックを空にしておくとエラーになってしまいます。 そこで使うのが `pass` 文です。

`pass` は「何もしない」という特別な命令で、構文上何かを書かないといけないけれど、まだ処理内容が決まっていない場所で使えます。

```
def do_something_later():
    pass # とりあえず何もしないでおく

do_something_later() # 呼び出しても何も起こらない
print(”passのテスト完了”)
```

実行結果：

```
passのテスト完了
```

---

3. 関数に情報を渡す：引数（ひきすう）

関数は、ただ決まった処理をするだけでなく、呼び出される時に外部から情報を受け取って、その情報に基づいて処理を変えることができます。この関数に渡す情報のことを「**引数（ひきすう）**」と呼びます。

> 

引数とは？

関数を定義するときに、丸括弧 `()` の中に変数名を書くことで、その関数が受け取る引数を指定できます。この定義時に書く引数のことを「**仮引数（parameter）**」と呼びます。 そして、関数を呼び出す時に実際に渡す値のことを「**実引数（argument）**」と呼びます。

引数を使って関数に値を渡す

```
# name という仮引数を持つ関数を定義
def greet_person(name): # name が仮引数
    print(f”こんにちは、{name}さん！”)

# 関数を呼び出す時に実引数を渡す
greet_person(”山田”)   # ”山田” が実引数として name に渡される
greet_person(”鈴木”)   # ”鈴木” が実引数として name に渡される

my_friend = ”加藤”
greet_person(my_friend) # 変数を実引数として渡すこともできる
```

実行結果：

```
こんにちは、山田さん！
こんにちは、鈴木さん！
こんにちは、加藤さん！
```

`greet_person` 関数は、呼び出されるたびに異なる `name` を受け取り、それに応じたメッセージを表示していますね。

複数の引数を使う

関数は複数の引数を受け取ることもできます。定義時にカンマ `,` で区切って仮引数を並べます。

```
def add_numbers(x, y): # x と y という2つの仮引数
    result = x + y
    print(f"{x} + {y} = {result}")

add_numbers(5, 3)    # 5がxに、3がyに渡される
add_numbers(100, 200)
```

実行結果：

```
5 + 3 = 8
100 + 200 = 300
```

引数の順番が大切：位置引数

複数の引数がある場合、関数を呼び出す時に渡す実引数の**順番**が重要になります。実引数は、定義された仮引数の順番通りに対応付けられます。これを「**位置引数 (positional argument)**」と呼びます。

```
def describe_pet(animal_type, pet_name):
    print(f"私は {animal_type} を飼っています。")
    print(f"名前は {pet_name} です。")

describe_pet("ハムスター", "ジャンゴ")
# 順番を間違えると...
describe_pet("ポチ", "犬") # 意図しない結果になる！
```

実行結果：

```
私は ハムスター を飼っています。
名前は ジャンゴ です。
私は ポチ を飼っています。
名前は 犬 です。
```

2回目の呼び出しでは、`animal_type` に "ポチ" が、`pet_name` に "犬" が入ってしまっていますね。

名前で指定する：キーワード引数

引数の順番を気にせずに、また、どの値がどの仮引数に対応するのかを分かりやすくするために、「**キーワード引数（keyword argument）**」という渡し方があります。 これは `仮引数名=値` の形で実引数を指定する方法です。

```
def describe_pet(animal_type, pet_name):
    print(f"私は {animal_type} を飼っています。")
    print(f"名前は {pet_name} です。")

# キーワード引数を使えば、順番は自由
describe_pet(pet_name="タマ", animal_type="猫")
describe_pet(animal_type="犬", pet_name="コロ")
```

実行結果：

```
私は 猫 を飼っています。
名前は タマ です。
私は 犬 を飼っています。
名前は コロ です。
```

キーワード引数を使うと、コードが読みやすくなり、引数の意味も明確になります。位置引数とキーワード引数は混ぜて使うこともできますが、その場合は**位置引数を先に書き、その後にキーワード引数を書く**というルールがあります。

最初から値が決まっている：デフォルト引数値

関数の仮引数には、あらかじめ「**デフォルト値**」を設定しておくことができます。もし関数呼び出し時にその引数が省略された場合、このデフォルト値が使われます。

デフォルト値は、関数定義時に `仮引数名=デフォルト値` のように書きます。

```
def greet_with_country(name, country="日本"): # countryのデフォルト値は "日本"
    print(f"こんにちは、{name}さん！")
    print(f"{country}から来ましたか？")

greet_with_country("田中") # country引数は省略したのでデフォルト値 "日本" が使われる
greet_with_country("John", "アメリカ") # country引数を指定したので "アメリカ" が使われる
```

実行結果：

```
こんにちは、田中さん！
日本から来ましたか？
こんにちは、Johnさん！
アメリカから来ましたか？
```

デフォルト引数値は、よく使われる値を設定しておくと、関数呼び出しがシンプルになるので便利です。 **注意点：** デフォルト引数値を持つ仮引数は、持たない仮引数の**後ろに**定義する必要があります。

```
# これはOK
# def func(a, b=10):
#     pass

# これはエラー！
# def func_error(a=10, b):
#     pass
```

---

4. 関数から結果を受け取る：戻り値（もどりち）

関数は、処理を実行するだけでなく、その結果を呼び出し元に返すことができます。この返される値のことを「**戻り値（もどりち）**」または「**返り値（かえりち）**」と呼びます。

> 

`return` で値を返す

関数から値を返すには `return` というキーワードを使います。

```
def add(x, y):
    calculation_result = x + y
    return calculation_result # 計算結果を返す

sum_result = add(10, 20) # add関数の戻り値が sum_result に代入される
print(f"10 + 20 の結果は {sum_result} です。")

print(f"5 + 8 の結果は {add(5, 8)} です。") # 直接printの中で使うこともできる
```

実行結果：

```
10 + 20 の結果は 30 です。
5 + 8 の結果は 13 です。
```

`return` の後に書かれた式の値が、関数の呼び出し元に返されます。`return` が実行されると、その時点で関数の処理は終了し、呼び出し元に戻ります。

戻り値を変数に代入する

上の例のように、関数の戻り値は変数に代入して、後で使うことができます。これは非常によく行われる操作です。

複数の値を返す（タプルで返る）

関数は `return` で複数の値を返すこともできます。その場合、値はカンマ `,` で区切って書きます。 複数の値が返されると、それらは「**タプル（tuple）**」という形でまとめられて返されます。（タプルは、中身を変更できないリストのようなものだと考えてください。詳しくは以前の資料「リストと辞書」の辞書の `.items()` のところで少し触れましたね。）

```
def get_name_and_age():
    name = ”高専花子”
    age = 19
    return name, age # name と age を返す

info = get_name_and_age() # 戻り値はタプル（name, age）
print(f”受け取った情報（タプル）: {info}”)
print(f”名前: {info[0]}, 年齢: {info[1]}”)

# タプルはアンパッキングして複数の変数で直接受け取ることもできる
student_name, student_age = get_name_and_age()
print(f”名前（アンパッキング）: {student_name}, 年齢（アンパッキング）: {student_age}”)
```

実行結果：

```
受け取った情報（タプル）: (’高専花子’, 19)
名前: 高専花子, 年齢: 19
名前（アンパッキング）: 高専花子, 年齢（アンパッキング）: 19
```

戻り値がない関数（`None` が返る）

関数の中で `return` 文が実行されなかったり、`return` の後に何も書かれなかったりした場合、その関数は特別な値 `None`（ノン）を返します。`None` は「何もない」ということを表すPythonの特別な値です。

```
def simple_greet(name):
    print(f”やあ、{name}！”)
    # return がない

result = simple_greet(”ともこ”)
print(f”simple_greet関数の戻り値: {result}”)
print(f”戻り値の型: {type(result)}”)
```

実行結果：

```
やあ、ともこ！
simple_greet関数の戻り値: None
戻り値の型: <class ’NoneType’>
```

`print()` 関数自体も、画面に文字を表示しますが、戻り値としては `None` を返します。

5. 変数が使える範囲：スコープを意識しよう

プログラムの中で定義した変数は、どこからでも自由に使えるわけではありません。変数が有効な範囲（つまり、その変数にアクセスできる範囲）のことを「**スコープ**」と呼びます。

ローカル変数：関数の中でだけ使える

関数の中で定義された変数は、その関数の中だけで有効です。これを「**ローカル変数（local variable）**」と呼びます。関数の外からローカル変数にアクセスしようとするとエラーになります。

```
def my_function():
    message = ”これはローカル変数です。” # messageはmy_function内のローカル変数
    print(f”関数の中: {message}”)

my_function()
# print(f”関数の外: {message}”) # これはエラーになる！ (NameError: name ’message’ is not
defined)
```

実行結果（エラー行をコメントアウトした場合）:

```
関数の中: これはローカル変数です。
```

ローカル変数は、関数が呼び出されるたびに新しく作られ、関数が終了すると消えてしまいます。これにより、他の関数やプログラムの他の部分で同じ変数名を使っていても、互いに影響し合わないというメリットがあります。

グローバル変数：どこからでも使える（でも注意が必要！）

関数の外（通常はプログラムの最も上のレベル）で定義された変数は「**グローバル変数（global variable）**」と呼ばれ、プログラムのどこからでもアクセスできます。

```
global_message = ”これはグローバル変数です。”

def show_global_message():
    print(f”関数の中からグローバル変数: {global_message}”)

def try_to_modify_global():
    # global_message = ”グローバル変数を変更しようとしました” # これだけだと新しいローカル変数が
作られる
    # print(f”関数の中で変更試行後: {global_message}”)

    # グローバル変数の値を関数内で変更したい場合は global キーワードが必要
```

```
    global global_message # 「global_messageというグローバル変数をここで使います」と宣言
    global_message = "グローバル変数が変更されました！"
    print(f"関数の中で変更後: {global_message}")


show_global_message()
print(f"関数の外（変更前）: {global_message}")

try_to_modify_global()
print(f"関数の外（変更後）: {global_message}")
```

実行結果：

```
関数の中からグローバル変数: これはグローバル変数です。
関数の外（変更前）: これはグローバル変数です。
関数の中で変更後: グローバル変数が変更されました！
関数の外（変更後）: グローバル変数が変更されました！
```

注意点：

- 関数の中でグローバル変数と同じ名前の変数に値を代入しようとすると、Pythonはデフォルトで新しいローカル変数を作成しようとします。
- もし関数の中でグローバル変数の値を**変更したい**場合は、その変数の前に `global` キーワードを付けて宣言する必要があります。
- グローバル変数は便利に見えますが、多用するとプログラムのどこで値が変更されたのか追跡しにくくなり、バグの原因になることがあります。できるだけ関数に必要な情報は引数で渡し、結果は戻り値で受け取るように設計するのが良い習慣です。

スコープは最初は少しややこしく感じるかもしれませんが、プログラムが大きくなってくると非常に重要な概念になります。

---

6. 関数の説明書：ドキュメンテーション文字列（docstring）

自分で作った関数が、何をするためのものなのか、どんな引数を受け取り、何を返すのか、といった情報を関数自身に持たせておくことができます。これを「**ドキュメンテーション文字列（documentation string）**」、略して「**docstring（ドックストリング）**」と呼びます。

docstringの書き方と役割

docstringは、関数定義（`def`文）の直後の行に、トリプルクォート（`"""` または `'''`）で囲んで記述します。

```
def calculate_rectangle_area(width, height):
    """
    長方形の面積を計算して返す関数。
```

```
    Args:
        width (int or float): 長方形の幅。
        height (int or float): 長方形の高さ。

    Returns:
        int or float: 計算された長方形の面積。
                      幅か高さが0以下の場合は0を返す。
    """
    if width <= 0 or height <= 0:
        return 0
    return width * height

# docstringの例（一行の場合）
def say_hello(name):
    '''指定された名前で挨拶をします。'''
    print(f"こんにちは、{name}さん。")
```

docstringには、以下のような情報を書くのが一般的です。

- 関数が何をするのかの簡潔な説明（1行目が要約になることが多い）。
- 引数（`Args` や `Parameters` などと書くことが多い）:
    - 各引数の名前とデータ型。
    - 各引数が何を表すかの説明。
- 戻り値（`Returns` などと書くことが多い）:
    - 戻り値のデータ型。
    - 戻り値が何を表すかの説明。

良いdocstringを書くことは、将来の自分や他の人がその関数を理解しやすくするために非常に重要です。

`help()` 関数でdocstringを見る

定義した関数のdocstringは、Pythonの組み込み関数 `help()` を使って確認することができます。

```
# 上で定義した calculate_rectangle_area 関数があるとして

help(calculate_rectangle_area)

# help(say_hello)
```

これを実行すると、ターミナルに `calculate_rectangle_area` 関数のdocstringが表示されます。

```
Help on function calculate_rectangle_area in module __main__:

calculate_rectangle_area(width, height)
    長方形の面積を計算して返す関数。

    Args:
        width (int or float): 長方形の幅。
```

```
        height (int or float): 長方形の高さ。

    Returns:
        int or float: 計算された長方形の面積。
                      幅か高さが0以下の場合は0を返す。
```

VSCodeなどの高機能なエディタでは、関数名にマウスカーソルを合わせるだけでdocstringをポップアップ表示してくれる機能もあります。

---

7. もしもの時に備える：エラーと例外処理

さて、関数も作れるようになり、プログラムもだんだん複雑なことができるようになってきましたね！でも、プログラミングをしていると、どうしても避けて通れないのが「**エラー**」です。 「あれ？プログラムが動かない！」「何か赤い文字がいっぱい出てきた！」なんて経験、もうすでにあるかもしれませんね。

でも大丈夫！エラーは敵ではありません。むしろ、プログラムが「ここがちょっとおかしいよ！」と教えてくれる大切なサインなんです。 このセクションでは、そんなエラーと上手に付き合っていくための「**例外処理（れいがいしょり）**」という仕組みを学びましょう。これをマスターすれば、予期せぬエラーでプログラムが突然止まってしまうのを防いだり、エラーが起きても適切に対処したりできるようになります。

プログラムのエラーってどんな時？

エラーが起きる原因は様々です。

- **書き間違い（構文エラー）**: 例えば、`print` を `prnt` と打ち間違えたり、括弧を閉じ忘れたり。これはプログラムを実行する前にPythonが見つけてくれることが多いです。

  ```
  # prnt(”Hello”) # SyntaxError: invalid syntax
  # if x > 0 # SyntaxError: expected ’:’
  ```

- **実行してみたらダメだった（実行時エラー / 例外）**: 文法は合っているけれど、実際に動かしてみると問題が起きるケースです。
  - 0で割り算しようとした（`ZeroDivisionError`）
  - 文字と数字を足そうとした（`TypeError`）
  - リストの範囲外の要素にアクセスしようとした（`IndexError`）
  - 存在しないファイルを開こうとした（`FileNotFoundError`）

  ```
  # print(10 / 0) # ZeroDivisionError: division by zero
  # print(”Hello” + 5) # TypeError: can only concatenate str (not ”int”) to str
  # my_list = [1, 2, 3]
  # print(my_list[5]) # IndexError: list index out of range
  ```

このような実行時エラーのことを、Pythonでは特に「**例外（exception）**」と呼びます。例外が発生すると、プログラムはそこで処理を中断してしまいます。

Pythonのエラーメッセージを読んでみよう

エラーが発生すると、Pythonは「**トレースバック（Traceback）**」と呼ばれるエラーメッセージを表示します。最初は難解に見えるかもしれませんが、実は問題解決のための重要な情報がたくさん詰まっています。

```
Traceback (most recent call last):
  File "c:/Users/e2210/ProgrammingProject/nitac_jyoken_public_materials/AI班/資
料/test_error.py", line 3, in <module>
    result = 10 / number
             ~~~~~~~~~~
ZeroDivisionError: division by zero
```

注目するポイントは：

1. **エラーが発生したファイル名と行番号**: 上の例だと `test_error.py` の `line 3` でエラーが起きたことが分かります。
2. **エラーの種類**: `ZeroDivisionError` と表示されています。これがエラーの具体的な名前です。
3. **エラーの詳細メッセージ**: `division by zero`（0による割り算）と、何が問題だったのかを教えてくれます。

エラーメッセージをしっかり読むことが、バグ修正の第一歩です！

`try` と `except`：エラーをキャッチする

「この処理はもしかしたらエラーが起きるかもしれないな…」という部分を、あらかじめ「エラーが起きても大丈夫なように見張っておく」仕組みが `try` と `except` です。

基本的な形はこうです。

```
try:
    # エラーが発生するかもしれない処理
    x = int(input("数字を入力してください: "))
    result = 10 / x
    print(f"10を{x}で割った結果: {result}")
except ZeroDivisionError:
    # ZeroDivisionError が発生した場合に実行される処理
    print("おっと！0で割ることはできませんよ。")
except ValueError:
    # ValueError（例: 文字列をintに変換しようとした）が発生した場合に実行される処理
    print("うーん、それは数字じゃないみたいですね。数字を入力してください。")
except Exception as e:
    # 上記以外の何らかの予期せぬエラーが発生した場合
    print(f"予期せぬエラーが発生しました: {e}")

print("プログラムは続いています...")
```

- **`try` ブロック**: この中に、エラーが発生する可能性のあるコードを書きます。
- **`except` ブロック**: `try` ブロック内で特定の種類の例外が発生した場合に、このブロックのコードが実行されます。

- `except 例外の種類:` のように、どのエラーをキャッチするか指定できます。
- 複数の `except` ブロックを書くことで、エラーの種類に応じた異なる対処ができます。
- `except Exception as e:` と書くと、あらゆる種類の例外をキャッチし、エラー情報 `e` を使うことができます。これは最後の砦として便利ですが、具体的なエラーを個別にキャッチする方が良い場合が多いです。

もし `try` ブロック内でエラーが発生しなければ、`except` ブロックは無視されて、その後の処理に進みます。エラーが発生して `except` ブロックが実行された場合も、プログラムはそこで止まらずに、`except` ブロックの後の処理に進むことができます（`raise` しない限り）。

よく見るエラーの種類

Pythonにはたくさんの種類の組み込み例外があります。いくつか代表的なものを知っておくと、エラーメッセージを見たときに「ああ、あれか！」と見当がつきやすくなります。

- `SyntaxError`: Pythonの文法が間違っている。
- `IndentationError`: インデント（字下げ）が正しくない。
- `NameError`: 定義されていない変数や関数を使おうとした。
- `TypeError`: 演算や関数に不適切な型のオブジェクトを渡した。（例: 文字列と数値を `+` で繋ごうとした）
- `ValueError`: 型は適切だが、値が不適切。（例: `int()` に数字以外の文字列を渡した）
- `IndexError`: シーケンス（リストや文字列など）の範囲外のインデックスを指定した。
- `KeyError`: 辞書に存在しないキーを指定した。
- `ZeroDivisionError`: 0で割り算しようとした。
- `FileNotFoundError`: 存在しないファイルを開こうとした。
- `AttributeError`: オブジェクトが持っていない属性（メソッドや変数）にアクセスしようとした。（例: `int` 型の変数にリストの `append` メソッドを使おうとした）
- `ImportError`: `import` しようとしたモジュールが見つからない。

これら以外にもたくさんあります。公式ドキュメントで「組み込み例外 (Built-in Exceptions)」を調べてみると良いでしょう。

`else` と `finally`：エラー処理の追加機能

`try...except` 文には、さらに `else` ブロックと `finally` ブロックを追加することができます。

```
try:
    # エラーが発生するかもしれない処理
    num_str = input("割られる数を入力してください: ")
    num = int(num_str)
    result = 100 / num
except ValueError:
    print(f"「{num_str}」は有効な数値ではありません。")
except ZeroDivisionError:
    print("0で割ることはできません。")
else:
    # tryブロックで例外が発生しなかった場合に実行される
    print(f"計算結果: {result}")
finally:
    # 例外が発生したかどうかに関わらず、必ず最後に実行される
```

```
    print("計算処理を終了します。")

print("プログラムの次のステップへ...")
```

- `else` **ブロック**: `try` ブロックで**例外が発生しなかった場合**に実行されます。エラーが起きなかった時にだけ行いたい処理を書くのに便利です。
- `finally` **ブロック**: `try` ブロックで例外が発生したかどうか、`except` ブロックが実行されたかどうかに関わらず、**必ず最後に実行されます**。ファイルやネットワーク接続を閉じるなど、後片付け処理を書くのによく使われます。

自分でエラーを発生させる：`raise`

時には、プログラムの特定の条件が満たされない場合に、意図的にエラーを発生させたいことがあります。例えば、関数の引数が不正な値だった場合などです。 そのために使うのが `raise` 文です。

```
def calculate_discount_price(price, discount_rate):
    if not (0 <= discount_rate <= 1):
        raise ValueError("割引率は0から1の間でなければなりません。") # ここでエラーを発生させる
    if price < 0:
        raise ValueError("価格は0以上でなければなりません。")

    return price * (1 - discount_rate)

try:
    final_price = calculate_discount_price(1000, 0.2) # 正常なケース
    print(f"割引後の価格: {final_price}")

    # final_price_error = calculate_discount_price(500, 1.5) # 不正な割引率
    # print(f"割引後の価格（エラーケース): {final_price_error}")

    final_price_error2 = calculate_discount_price(-100, 0.1) # 不正な価格
    print(f"割引後の価格（エラーケース2): {final_price_error2}")

except ValueError as e:
    print(f"エラー: {e}")
```

`raise` の後に例外クラスのインスタンス（例: `ValueError("エラーメッセージ")`）を指定することで、その例外を発生させることができます。 自分で定義した関数の中で、不正な使われ方をされたくない場合に、このように `raise` を使ってエラーを明示するのは良い習慣です。

エラーハンドリングは、堅牢で使いやすいプログラムを作るためには欠かせない技術です。最初は少し難しく感じるかもしれませんが、実際にエラーに遭遇し、それを解決していく中でだんだんと身についていきます。 エラーメッセージを恐れずに、じっくり読んでみてくださいね

---

8. Pythonの便利な道具箱：標準ライブラリを使ってみよう

Pythonの大きな魅力の一つは、最初からたくさんの便利な機能（モジュール）が「**標準ライブラリ**」として提供されていることです。これらを使えば、複雑な数学計算、ランダムな数値の生成、日付や時刻の操作、ファイルの読み書きなど、様々なことを簡単に行うことができます。 自分で一から全部作る必要はなく、先人たちが作ってくれた便利な道具を借りるイメージですね！

標準ライブラリとは？

標準ライブラリは、Pythonをインストールした時に一緒にインストールされる、あらかじめ用意されたモジュール（関数やクラスなどが集まったファイル）の集まりです。 「モジュール」とは、関連する機能がまとめられたPythonのファイル（`.py`ファイル）のことです。

ライブラリを使う準備：`import`

標準ライブラリの機能を使うには、まず「このライブラリを使いますよ」とPythonに教えてあげる必要があります。そのために使うのが `import` 文です。

`import` の仕方にはいくつかバリエーションがあります。

- **`import ライブラリ名`**: ライブラリ全体を読み込みます。機能を使うときは `ライブラリ名.機能名` のように書きます。

```
import math

print(math.pi) # mathライブラリの円周率pi
print(math.sqrt(16)) # mathライブラリの平方根を計算するsqrt関数
```

- **`from ライブラリ名 import 特定の機能`**: ライブラリの中から、特定の関数や変数だけを読み込みます。この場合、機能を使うときはライブラリ名を付けずに直接機能名を書けます。

```
from random import randint, choice

print(randint(1, 6)) # 1から6までのランダムな整数（random.randintではない）
my_list = ['apple', 'banana', 'cherry']
print(choice(my_list)) # リストからランダムに1つ選択
```

- **`from ライブラリ名 import *`**: ライブラリの中の全ての機能を直接使えるように読み込みます。しかし、どの機能がどのライブラリから来たのか分かりにくくなったり、名前の衝突が起きやすくなったりするため、**あまり推奨されません。**

- **`import ライブラリ名 as 別名`**: ライブラリ名が長かったり、他の名前と区別しやすくしたい場合に、別名（エイリアス）をつけて読み込むことができます。

```
import datetime as dt # datetimeライブラリをdtという別名でインポート

now = dt.datetime.now()
print(now)
```

どの `import` 方法を使うかは状況や好みによりますが、`import ライブラリ名` や `from ライブラリ名 import 特定の機能` がよく使われます。

それでは、代表的な標準ライブラリをいくつか見ていきましょう！

`math` ライブラリ：数学計算はおまかせ

`math` ライブラリは、平方根、三角関数、対数、円周率など、様々な数学的な計算を行うための関数や定数を提供しています。

```
import math

# 定数
print(f"円周率 pi: {math.pi}")
print(f"ネイピア数 e: {math.e}")

# 平方根
print(f"16の平方根: {math.sqrt(16)}") # 4.0

# べき乗
print(f"2の10乗: {math.pow(2, 10)}") # 1024.0

# 三角関数（角度はラジアンで指定）
angle_rad = math.pi / 4 # 45度をラジアンで
print(f"sin(pi/4): {math.sin(angle_rad)}")
print(f"cos(pi/4): {math.cos(angle_rad)}")

# 度とラジアンの変換
print(f"90度をラジアンに: {math.radians(90)}")
print(f"piラジアンを度に: {math.degrees(math.pi)}")

# 切り上げ、切り捨て、整数部
num = 3.14159
print(f"{num}の切り上げ（ceil): {math.ceil(num)}")   # 4
print(f"{num}の切り捨て（floor): {math.floor(num)}") # 3
print(f"{num}の整数部（trunc): {math.trunc(num)}") # 3（0に近い整数）
print(f"-{num}の整数部（trunc): {math.trunc(-num)}") # -3

# 対数
print(f"log10(100): {math.log10(100)}") # 2.0（底が10の対数）
print(f"log(e^2)（自然対数): {math.log(math.e ** 2)}") # 2.0
```

`math` ライブラリには他にもたくさんの関数があります。数学的な計算が必要になったら、まず `math` ライブラリに便利な関数がないか調べてみると良いでしょう。`help(math)` で一覧を見ることもできます。

`random` ライブラリ：サイコロを振ってみよう

`random` ライブラリは、乱数（ランダムな数）を生成したり、シーケンス（リストなど）からランダムに要素を選んだりするための機能を提供します。ゲームを作ったり、データをランダムに並び替えたりする時に便利で

す。

```
import random

# 0.0以上、1.0未満のランダムな浮動小数点数
print(f"0.0以上1.0未満の乱数: {random.random()}")

# 指定した範囲のランダムな整数（a <= N <= b）
print(f"1から6までのランダムな整数（サイコロ）: {random.randint(1, 6)}")

# 指定した範囲のランダムな整数（start <= N < stop）
print(f"0から9までのランダムな整数: {random.randrange(0, 10)}")
print(f"1から10までの奇数からランダム: {random.randrange(1, 11, 2)}") # stepも指定可能

# リストなどのシーケンスからランダムに1つの要素を選択
fruits = ["apple", "banana", "cherry", "orange"]
print(f"ランダムに選ばれた果物: {random.choice(fruits)}")

# リストの要素をランダムに並び替える（元のリストが変更される！）
numbers = [1, 2, 3, 4, 5]
print(f"シャッフル前のリスト: {numbers}")
random.shuffle(numbers)
print(f"シャッフル後のリスト: {numbers}")

# シーケンスから指定した個数の要素をランダムに重複なく抽出（元のシーケンスは変更されない）
deck = list(range(1, 53)) # 1から52までのトランプのカード
hand = random.sample(deck, 5) # 5枚のカードを引く
print(f"引いた5枚のカード: {hand}")
```

`random` ライブラリを使うと、プログラムに予測不可能な動きを加えることができて面白いですよ！

`datetime` ライブラリ：日付や時刻を扱おう

`datetime` ライブラリは、日付や時刻を扱いたいときに非常に強力なツールです。現在の日時を取得したり、特定の日付を表現したり、日付同士の計算をしたりできます。

```
import datetime

# 現在の日時を取得
now = datetime.datetime.now()
print(f"現在の日時（datetimeオブジェクト）: {now}")

# 今日の日付を取得
today = datetime.date.today()
print(f"今日の日付（dateオブジェクト）: {today}")

# 特定の日時を指定してオブジェクトを作成
specific_datetime = datetime.datetime(2025, 12, 24, 18, 30, 0)
print(f"特定の日時: {specific_datetime}")
```

```
# 日付や時刻の各要素にアクセス
print(f"年: {now.year}")
print(f"月: {now.month}")
print(f"日: {now.day}")
print(f"時: {now.hour}")
print(f"分: {now.minute}")
print(f"秒: {now.second}")
print(f"マイクロ秒: {now.microsecond}")
print(f"曜日 (0:月曜, 1:火曜, ..., 6:日曜): {now.weekday()}") # 月曜日が0

# 日付や時刻を好きな書式の文字列に変換 (strftime: string format time)
formatted_now = now.strftime("%Y年%m月%d日 %H時%M分%S秒")
print(f"書式指定された現在の日時: {formatted_now}")
# %Y: 4桁の年, %m: 2桁の月, %d: 2桁の日
# %H: 24時間表記の時, %M: 2桁の分, %S: 2桁の秒
# 他にもたくさん書式コードがあります (例: %Aで曜日のフルネーム、%Bで月のフルネーム)

# 文字列からdatetimeオブジェクトに変換 (strptime: string parse time)
date_string = "2024-07-15 10:00:00"
parsed_datetime = datetime.datetime.strptime(date_string, "%Y-%m-%d %H:%M:%S")
print(f"文字列から変換した日時: {parsed_datetime}")

# 時間の差 (timedelta)
one_day = datetime.timedelta(days=1)
tomorrow = now + one_day
print(f"明日の同じ時間: {tomorrow}")

three_hours = datetime.timedelta(hours=3)
three_hours_ago = now - three_hours
print(f"3時間前: {three_hours_ago}")

# 2つの日時の差
diff = specific_datetime - now
print(f"特定の日時までの残り時間: {diff}")
print(f"残り日数: {diff.days}")
```

`datetime` ライブラリは機能が豊富なので、最初は少し難しく感じるかもしれませんが、ログの記録やイベントのスケジュール管理など、多くの場面で役立ちます。

✏️ 本日の演習

さあ、今日学んだ関数と標準ライブラリの知識を使って、実際に手を動かしてみましょう！

1. VSCodeで新しいPythonファイル（例: `practice06.py`）を作成してください。
2. **課題1：挨拶関数**
    - 名前を引数として受け取り、「こんにちは、〇〇さん！」と表示する関数 `greet(name)` を作成してください。

- 作成した関数を、あなたの名前や友人の名前で何度か呼び出してみてください。

3. **課題2：長方形の面積を計算する関数**
    - 長方形の幅 `width` と高さ `height` を引数として受け取り、その面積を計算して返す関数 `calculate_rectangle_area(width, height)` を作成してください。
    - 関数を呼び出し、戻り値を変数に格納して表示するか、直接 `print()` の中で呼び出して結果を表示してください。(例: 幅5、高さ8の長方形)
4. **課題3：リストの合計と平均を計算する関数**
    - 数値のリストを引数として受け取り、そのリスト内の数値の合計と平均を計算して、**両方の値を返す**関数 `calculate_sum_and_average(numbers)` を作成してください。
        - ヒント1: 合計は `sum()` 関数が使えます。平均は 合計 / 要素数 です。要素数は `len()` で取得できます。
        - ヒント2: リストが空の場合にエラーにならないように注意しましょう（例えば、空なら合計0、平均0を返すなど）。
    - 関数を呼び出し、返ってきた2つの値（タプル）をそれぞれ別の変数に代入して表示してください。
5. **課題4：サイコロゲーム**
    - `random` ライブラリの `random.randint(1, 6)` を使って、1から6までのランダムな整数を2回生成し（サイコロを2回振るイメージ）、その合計点を表示するプログラムを作成してください。
    - もし合計点が7なら「ラッキーセブン！」、合計点がゾロ目（1と1、2と2など）なら「ゾロ目！」と追加で表示するようにしてみましょう。
6. **課題5：今日の曜日を表示**
    - `datetime` ライブラリを使って、今日の日付と曜日を表示するプログラムを作成してください。
    - ヒント: `datetime.datetime.now().weekday()` は月曜日を0とする数値を返します。これを使って、日本語の曜日（「月曜日」「火曜日」など）を表示するにはどうすればよいか考えてみましょう（リストやif文が使えると良いですね）。
    - `strftime('%A')` を使うと英語の曜日名が取得できるので、それも試してみてください。

演習の解答例

```
# practice06.py

import math
import random
import datetime

# --- 課題1：挨拶関数 ---
print("--- 課題1：挨拶関数 ---")
def greet(name):
    """指定された名前で挨拶を表示する関数"""
    print(f"こんにちは、{name}さん！")

greet("高専太郎")
greet("AI花子")
print("-" * 30)

# --- 課題2：長方形の面積を計算する関数 ---
print("--- 課題2：長方形の面積を計算する関数 ---")
def calculate_rectangle_area(width, height):
```

```
    """長方形の面積を計算して返す関数"""
    if width <= 0 or height <= 0:
        return 0 # 不正な値の場合は面積0とする
    return width * height

area1 = calculate_rectangle_area(5, 8)
print(f"幅5、高さ8の長方形の面積: {area1}")
print(f"幅10、高さ3の長方形の面積: {calculate_rectangle_area(10, 3)}")
print(f"幅-2、高さ5の長方形の面積: {calculate_rectangle_area(-2, 5)}")
print("-" * 30)

# --- 課題3：リストの合計と平均を計算する関数 ---
print("--- 課題3：リストの合計と平均を計算する関数 ---")
def calculate_sum_and_average(numbers):
    """数値のリストを受け取り、合計と平均をタプルで返す関数"""
    if not numbers: # リストが空の場合
        return 0, 0.0

    total_sum = sum(numbers)
    average = total_sum / len(numbers)
    return total_sum, average

my_scores = [80, 95, 72, 88, 90]
s, avg = calculate_sum_and_average(my_scores)
print(f"リスト {my_scores} の合計: {s}, 平均: {avg:.2f}")

empty_list = []
s_empty, avg_empty = calculate_sum_and_average(empty_list)
print(f"空のリストの合計: {s_empty}, 平均: {avg_empty}")
print("-" * 30)

# --- 課題4：サイコロゲーム ---
print("--- 課題4：サイコロゲーム ---")
dice1 = random.randint(1, 6)
dice2 = random.randint(1, 6)
total_dice_score = dice1 + dice2

print(f"サイコロ1の目: {dice1}")
print(f"サイコロ2の目: {dice2}")
print(f"合計点: {total_dice_score}")

if total_dice_score == 7:
    print("ラッキーセブン！")
if dice1 == dice2: # ゾロ目かどうかの判定
    print("ゾロ目！")
print("-" * 30)

# --- 課題5：今日の曜日を表示 ---
print("--- 課題5：今日の曜日を表示 ---")
now = datetime.datetime.now()
```

```
# 方法1: weekday() とリストを使う
weekdays_jp = ["月曜日", "火曜日", "水曜日", "木曜日", "金曜日", "土曜日", "日曜日"]
today_weekday_index = now.weekday()
print(f"今日の日付: {now.year}年{now.month}月{now.day}日")
print(f"今日の曜日は {weekdays_jp[today_weekday_index]} です。 (weekday()使用)")

# 方法2: strftime('%A') を使う (英語表記)
print(f"今日の曜日は {now.strftime('%A')} です。 (strftime('%A')使用)")

# おまけ: strftimeで日本語の曜日 (環境による、Windowsでは期待通りにならないことも)
# import locale
# try:
#     locale.setlocale(locale.LC_TIME, 'ja_JP.UTF-8') # または 'Japanese_Japan.932'
#     print(f"今日の曜日は {now.strftime('%A')} です。 (strftime日本語ロケール設定後)")
# except locale.Error:
#     print("日本語ロケールの設定に失敗しました。")

print("-" * 30)
print("¥n演習お疲れ様でした！")
```

今回は、プログラムを部品化する「関数」と、そして安全な実装のために必須の「エラーハンドリング」、Pythonの強力な武器である「標準ライブラリ」について学びました。 関数を使うことで、あなたのコードはもっと整理され、読みやすく、そして再利用しやすくなります。標準ライブラリを使いこなせば、複雑な処理も数行のコードで実現できるようになり、プログラミングの可能性がぐっと広がります。

最初はたくさんの関数やライブラリがあって戸惑うかもしれませんが、全てを一度に覚える必要はありません。「こんなことができるんだな」という引き出しをたくさん作っておいて、必要になった時に調べながら使っていくのがコツです。 実際に手を動かして、色々な関数を試したり、ライブラリのドキュメントを眺めてみたりすると、新しい発見があって楽しいですよ！

次回の資料では、今まで学んできたことを振り返り、知識を整理しましょう。そして、実際にAI開発を行うために利用していくライブラリに立ち向かう前準備を指定行きます。

分からないこと、もっと知りたいことがあれば、いつでも遠慮なく部長のカトに気軽に質問してくださいね！みんなで一緒にスキルアップしていきましょう！