

プログラミングII 2025 課題レポート（第3回）

解答用紙 → このファイルにレポートを書き込み、Githubへ提出する

- 問題用紙をよく読み、この回答用紙にレポート記述して提出すること
- マークダウン記法を、しっかりプレビューして、記述ミスはすべて修正して完成させること
- 作成したコード、動作確認（実行結果）、分析・考察を自分の言葉で記入する
- 生成AIの回答、他人のレポートをコピーしてはいけない（判明し次第、減点ペナルティ対象とする）

レポート提出者：

クラス	学籍番号	氏名
B	20125058	森田巧成

解答編

- それぞれの問題をよく読み、以下の項目に自分の言葉でしっかり記述する
- マークダウン記法を、しっかりプレビューして、記述ミスはすべて修正して完成させる

選択する回答コース：「AB」コース

- 骨太コース：SA級**：生成AIを使わず、ノーヒントで自力で解く骨太な勇者・猛者・トリックスターコース
- 庶民コース：AB級**：生成AIを使わず、ヒントにしたがってアプローチする一般的コース
- ゆるふわコース：CD級**：生成AIや他力本願を前提とする ゆとりコース

問題1: ジェネレータを使ったコルーチンの設計と実装

- 概要:** Pythonジェネレータを使って生産者-消費者問題を解決するコードを設計せよ
- 課題内容:**
 - 要件：生産者コルーチンがデータを生成し、消費者コルーチンがデータを消費するコード
 - `send`と`receive = yield`を用いた協調的並列処理の実装
 - 分析:** なぜこの設計が必要か？実行結果の確認と検証
 - 考察:** コルーチンを用いることで得られる効率性や設計上の利点を論じる
 - 応用（チャレンジ）** ジェネレータに対する`StopIteration`例外を使うことで、生産者が生産を終了した後、消費者も消費を停止するコードに進化せよ

作成したpythonコード、特に空欄やポイントの説明

```
import time
import random
```

```

# 生産者Coroutine 無からアイテムを生産する専門家
def producer(consumer_coroutine):
    print("[生産者]: 生産準備OK...")
    consumer_coroutine.send(None) # 消費者コルーチンを開始 【空欄A】 コルーチンの初期化 = 実行をする必要がある。最初のyieldまで進める必要がある。
    for i in range(3): # アイテム（データ）を3個生産
        duration = random.uniform(0.5, 1.5)
        time.sleep(duration) # 生産に要した時間
        item = f"アイテム {i}"
        print(f"\n---[生産者]: 所要時間{duration:.2f}秒で、 {item} を生産")
        consumer_coroutine.send(item) # 生産したデータを消費者に送る 【空欄B】 消費者にitemのデータを送っている。yieldに値を送信、消費者が実行
    consumer_coroutine.close() # 消費者コルーチンを終了
    print("\n---[生産者]: 生産終了")

# 消費者Coroutine アイテムを受け取り、別の製品を生み出す専門家
def consumer():
    print("<消費者>: 準備OK...")
    while True:#永遠に続くため、closeを使っている。
        item = yield # データを受け取る 【空欄C】 yieldが値を受け取っている。
        print(f"<配送>{item}")
        duration = random.uniform(0.5, 1.5)
        print(f"<配送>: 配送時間{duration:.2f}秒で、{item} 配送完了" )
        print(f"<消費者>: {item} 受領... 消費中.....")
        time.sleep(duration) # 消費に要した時間
        print(f"<消費者>: 所要時間{duration:.2f}秒で、{item} 消費完了")
    print("\n<消費者>: 消費完了") # この出力を最後に出すためには、StopIteration例外が必要（応用問題）

if __name__ == "__main__":
    consumer_coroutine = consumer() # 消費者コルーチン の生成？ 【空欄D】
    producer(consumer_coroutine) # 生産者を起動 【空欄E】

#非同期：今の処理が終わらなくても次の処理が始まる

```

動作確認結果、確認ポイントの説明

[生産者]: 生産準備OK...

---[生産者]: 所要時間1.32秒で、 アイテム 0 を生産
 <配送>アイテム 0
 <配送>: 配送時間0.53秒で、アイテム 0 配送完了
 <消費者>: アイテム 0 受領... 消費中.....
 <消費者>: 所要時間0.53秒で、アイテム 0 消費完了

---[生産者]: 所要時間1.04秒で、 アイテム 1 を生産
 <配送>アイテム 1
 <配送>: 配送時間0.72秒で、アイテム 1 配送完了
 <消費者>: アイテム 1 受領... 消費中.....
 <消費者>: 所要時間0.72秒で、アイテム 1 消費完了

```
---[生産者]: 所要時間0.84秒で、アイテム 2 を生産
<配送>アイテム 2
<配送>:配送時間0.90秒で、アイテム 2 配送完了
<消費者>: アイテム 2 受領... 消費中.....
<消費者>: 所要時間0.90秒で、アイテム 2 消費完了

---[生産者]: 生産終了
```

分析

- このコルーチンは、ジェネレータベースであり、yieldとsend()をうまく活用して、手動でデータの送信や、受け取りを行えるようにすることや、コルーチンの初期化など。
- 二つのコルーチン順番に再開させ、同時に実行しているかのように見せる、疑似並列について。
- ・

考察

- まず、コルーチンを実行するには、初期化をする必要があるのと、どのようにして順番に実行させるのかを、yield、send()を活用して、データの送受信をできるようにするなど、コルーチンの仕組みを理解する
- 二つのコルーチンがデータのやり取りを行えるように、データの流れを追う
- 多くの処理を並列して処理できるので、センサーなどの多くの情報を処理する場面では、役に立ちそう。

改善点

-while Trueは無限ループになってしまってるので、終了条件をアイテム数までなど柔軟にしてつければいい感じた。

- ・

問題2: 基本的なオブジェクト指向の例題

- 概要:** 図形クラスとその派生クラスを設計し、オブジェクト指向の原則の理解度を確認する
- 課題内容:**
 - 抽象基底クラスShapeを定義し、以下のクラスを継承して設計実装せよ
 - 四角形クラスRectangle
 - 三角形クラスTriangle
 - 楕円クラスEllipse
 - 五角形クラスPentagon
 - 各クラスにrotateメソッドを実装し、「指定角度 (degree) だけ時計回りに回転させる」機能を追加しなさい（シミュレーションでよい、厳密な行列・ベクトルを用いた幾何学的回転の実装は不要）
 - ポリモーフィズムの概念を用いた操作を記述せよ

4. 考察: 継承、抽象基底クラスによるインターフェースの強制、ポリモーフィズムのメリットと、設計上の効果を分析しなさい

作成したpythonコード、特に空欄やポイントの説明

```
from abc import ABC, abstractmethod

# rotate()メソッドを義務づける抽象基底クラス Shape
class Shape(ABC):#ABCは、抽象クラスである。これにより、実装を強制することができる。
    @abstractmethod # rotate()インターフェースを強制する【空欄A】
    def rotate(self, angle):
        pass

# Shapeを継承した2次元図形クラス Shape2D
class Shape2D(Shape):#これが上位クラスとなって、メソッドを継承させている
    def __init__(self,angle):
        self.angle = angle
    def rotate(self, angle): # 強制されたインターフェースの実装 【空欄B】
        if self.angle>0:      #self.angleはインスタンス変数であり、インスタンスを作成時に、与えられる
            print("正の角度")  #angleは関数を実行する際に、与えられる動かす角度のこと
        else:
            print("負の角度")

        print(f'{self.__class__} @ angle({self.angle}) rotated by ({angle})degree ->',end=' ')
        self.angle += angle
        self.angle %= 360
        print(f'to angle({self.angle})')

# 長方形クラス Rectangle
class Rectangle(Shape2D):
    def __init__(self, width, height, angle=0):
        self.width = width
        self.height = height
        self.angle = angle
    def rotate(self, angle):
        super().rotate(angle) # 上位クラスのメソッド呼び出し【空欄C】
        print(f'calc rotate (width,height)={self.width,self.height}@{self.angle}'')

# 三角形クラス Triangle
class Triangle(Shape2D):
    def __init__(self, base, height, angle=0):
        self.base = base
        self.height = height
        self.angle = angle
    def rotate(self, angle):
        super().rotate(angle) # 上位クラスのメソッド呼び出し【空欄D】
        print(f'calc rotate (base,height)={self.base,self.height}@{self.angle}')
```

```

# 橋円クラス Ellipse
class Ellipse(Shape2D):
    def __init__(self, major_axis, minor_axis, angle=0):
        self.major = major_axis
        self.minor = minor_axis
        self.angle = angle
    def rotate(self, angle):
        super().rotate(angle) # 上位クラスのメソッド呼び出し【空欄E】
        print(f'calc rotate (major,minor)={self.major,self.minor}@{self.angle}'')

# 五角形クラス Pentagon
class Pentagon(Shape2D):
    def __init__(self, side_length, angle=0):
        self.side = side_length
        self.angle = angle
    def rotate(self, angle):
        super().rotate(angle) # 上位クラスのメソッド呼び出し【空欄F】
        print(f'calc rotate (side)={self.side}@{self.angle}'')

class square(Shape2D):
    def __init__(self, one_side, angle=0):
        self.one_side = one_side
        self.angle = angle
    def rotate(self, angle):
        super().rotate(angle)
        print(f'calc rotate (one_side)={self.one_side}@{self.angle}'')

# 多様な図形インスタンスのリスト： 長方形、三角形、橋円、五角形の個体
shapes = [
    Rectangle(10, 5, angle=270),
    Triangle(6, 8, angle=90),
    Ellipse(5, 3, angle=330),
    Pentagon(7, angle=-180),
    square(4,angle=50)
]

# 個体がどの図形インスタンスかを意識せず、
# rotate()メッセージを送信しメソッド起動する
# ポリモーフィズム（多相性）の効能
for shape in shapes:
    shape.rotate(60) # ポリモーフィズムの活用【空欄G】

#super()は上位クラスのメソッドにアクセスするための関数 繙承する

```

動作確認結果、確認ポイントの説明

正の角度

```
<class '__main__.Rectangle'> @ angle(270) rotated by (60)degree ->to angle(330)
calc rotate (width,height)=(10, 5)@330
正の角度
```

```
<class '__main__.Triangle'> @ angle(90) rotated by (60)degree ->to angle(150)
calc rotate (base,height)=(6, 8)@150
正の角度
<class '__main__.Ellipse'> @ angle(330) rotated by (60)degree ->to angle(30)
calc rotate (major,minor)=(5, 3)@30
負の角度
<class '__main__.Pentagon'> @ angle(-180) rotated by (60)degree ->to angle(240)
calc rotate (side)=7@240
正の角度
<class '__main__.square'> @ angle(50) rotated by (60)degree ->to angle(110)
calc rotate (one_side)=4@110
```

分析

- ABCクラスを継承して抽象基底クラスとして定義し、デコレータを使い抽象メソッドを定義することでサブクラスで用いる関数の実装を強制することができること。
- 子クラスが、親クラスを継承することで、親クラスの機能を引き継いたり、拡張することができるここと。
- インターフェースによって、異なるオブジェクトで、同じメソッドを活用しているが別々の処理をすることができること。

考察

- 抽象基底クラスを作る際に、ポリモーフィズムを使うため、メソッド名を統一しなければいけない。
- インスタンス化前の時点で、rotate()メソッドをそれぞれのクラスで活用したかったので、super()を用いて、親クラスのメソッドを呼び出すようにした。
- 引数と、インスタンス変数の値の、変化の流れをしっかりと理解するようにした

改善点

- 多角形に対応した、クラスを作ることで一つの図形にこだわらず柔軟に対応できると感じた。
- ・
- ・

問題3: コマンド(Command) デザインパターンの実装

- 概要:** テキストエディタに対する編集操作とUndo/Redo機能をコマンドデザインパターンで実装せよ
- 課題内容:**
 - 基本的なコマンドクラスCommandを設計
 - 具体的なコマンドとして、InsertText、DeleteTextを実装しなさい（シミュレーションでよい）
 - Undo/Redo機能を有効にする履歴管理を追加しなさい
 - 考察:** コマンドパターンがもたらす柔軟性と拡張性について分析

作成したpythonコード、特に空欄やポイントの説明

```
class Command: #抽象クラスの中に、これらを実装させる。
    """コマンドクラスがもつ共通インターフェース"""
    def execute(self): # 実行
        raise NotImplementedError

    def undo(self): # キャンセル=実行を打ち消す、逆操作
        raise NotImplementedError

class InsertText(Command):#特定の操作の内容
    """テキストを挿入するコマンド"""
    def __init__(self, document, text, position):#documentにインスタンスが引数として、受け取っている。
        self.document = document # 編集対象のドキュメントオブジェクト
        self.text = text # 追加するテキスト
        self.position = position # 追加する位置

    def execute(self): # 実行 (挿入) self.document=Document
        self.document.insert(self.text, self.position)

    def undo(self): # キャンセル=実行を打ち消す、逆操作 (削除)
        self.document.delete(self.position, len(self.text)) # 逆操作から推論【空欄A】

    def __repr__(self):#インスタンスの中身のインスタンス変数を返す 中身を確認できる。
        return f"(挿入:{self.text}@{self.position})"

class DeleteText(Command):#特定の操作の内容
    """テキストを削除するコマンド"""
    def __init__(self, document, position, length):
        self.document = document # 編集対象のドキュメントオブジェクト
        self.position = position # 削除する位置
        self.length = length # 削除する文字数
        self.deleted_text = "" # Redo用に、削除されたテキストを保存

    def execute(self): # 実行 (削除)
        self.deleted_text = self.document.content[self.position:self.position + self.length]
        self.document.delete(self.position, self.length)#実質ここは、Documentクラスの変数で、メソッドを活用している。

    def undo(self): # キャンセル=実行を打ち消す、逆操作 (挿入)
        self.document.insert(self.deleted_text, self.position) # 逆操作から推論【空欄B】

    def __repr__(self):#文字列表現を返す
        return f"(削除:{self.deleted_text}@{self.position}から{self.length}文字)"

class Document:#操作を実行するメソッドを定義しているクラス
    """テキストを管理するドキュメントクラス"""
    def __init__(self):
        self.content = "" # ドキュメントの本文

    def insert(self, text, position): # 文書へ、テキスト挿入 text=self.text
```

```
position=self.position
    self.content = self.content[:position] + text + self.content[position:]
    print(f"Inserted '{text}' at {position}.\nCurrent content:
'{self.content}'")

def delete(self, position, length): # 文書から、テキスト削除
    self.content = self.content[:position] + self.content[position + length:]
    print(f"Deleted text at {position} with length {length}.\nCurrent content:
'{self.content}'")

class CommandManager:#コマンドを蓄積
    """コマンドを管理し、Undo/Redo機能を提供するクラス"""
    def __init__(self):
        self.undo_stack = [] # Undo履歴 スタックにコマンドオブジェクトをそのまま積み上げ
        self.redo_stack = [] # Redo履歴 スタックにコマンドオブジェクトをそのまま積み上げ

    def execute_command(self, command): # 編集コマンドの実行 edが引数として入る
        command.execute() # コマンドデザインパターンの実行【空欄C】
        self.undo_stack.append(command)
        self.redo_stack.clear() # 新しい操作を行ったらRedo履歴をクリア

    def undo(self): # 編集コマンドの取消し
        if self.undo_stack:
            command = self.undo_stack.pop()#末尾の要素がcommandに代入される 末尾のインスタンスが削除され、代入される
            command.undo() # コマンドデザインパターンの実行【空欄D】
            self.redo_stack.append(command)
        else:
            print("Nothing to undo.")

    def redo(self):# 編集コマンドの再実行 = 取消を取り消す
        if self.redo_stack:#スタックがあれば、True
            command = self.redo_stack.pop()
            command.execute() # コマンドデザインパターンの実行【空欄E】
            self.undo_stack.append(command)
        else:
            print("Nothing to redo.")

# 活用ケース
if __name__ == "__main__":
    document = Document()#操作するメソッドを定義しているクラスをインスタンス化 インサートやデリートのメソッドが定義されている

    edit_actions = [ #挿入、挿入、挿入、削除 編集コマンドのインスタンス
        InsertText(document, "Take", 0),#documentはインスタンス それぞれのクラスで同じインスタンスを使っているためインスタンスの変数は変化する
        InsertText(document, "the ", 5),#__repr__メソッドの公式な文字列表現を返す
        InsertText(document, "risk", 8),
        DeleteText(document, 0, 5),
    ]
    print(edit_actions)

    manager = CommandManager()#コマンドを記録
```

```
for ed in edit_actions:  
    manager.execute_command(ed) # コマンドデザインパターンの実行【空欄G】  
  
    print("\nUndo操作を2回実施:")  
    # Undo, Undo  
    manager.undo()#削除前に戻る  
    manager.undo()#三回目の挿入前に戻る  
    manager.undo()  
  
    print("\nRedo操作を2回実施:")  
    # Redo, Redo  
    manager.redo()#三回前の挿入後に戻る  
    manager.redo()#削除後に戻る  
    manager.redo()  
  
# パラメータ化とは、処理の中の値を外部から渡せるようにすること
```

動作確認結果、確認ポイントの説明

```
[(挿入:'Take'@0) , (挿入:' the '@5) , (挿入:' risk'@8) , (削除:' '@0から5文字)]  
Inserted 'Take' at 0.  
Current content: 'Take'  
Inserted ' the ' at 5.  
Current content: 'Take the '  
Inserted ' risk' at 8.  
Current content: 'Take the risk '  
Deleted text at 0 with length 5.  
Current content: 'the risk '
```

Undo操作を3回実施:

```
Inserted 'Take ' at 0.  
Current content: 'Take the risk '  
Deleted text at 8 with length 5.  
Current content: 'Take the '  
Deleted text at 5 with length 5.  
Current content: 'Take '
```

Redo操作を3回実施:

```
Inserted ' the ' at 5.  
Current content: 'Take the '  
Inserted ' risk' at 8.  
Current content: 'Take th riske '  
Deleted text at 0 with length 5.  
Current content: ' th riske '
```

分析

- コマンドパターンによる、操作をオブジェクトとしてカプセル化することで、限定された操作のみでしか操作できないようにすること
- Undo/Redoの、操作を元に戻す、戻した操作のやり直しを行うために、スタックを用いて記録を管理すること
- クラスごとに役割を持たせ、どのクラスで何を行っているのかを理解すること

考察

- それぞれのクラスで、何をしたいのかを考える
- 実行部分から、どのように処理されていくのか、実際にデータの流れを頭の中で整理しながら理解していく
- Undo/Redo操作で、なんのメソッドを活用するのかを出力結果をたどりながら考える

改善点

- -
 -
-
-

問題4: オブザーバ(Observer) デザインパターンの実装

- **概要:** Pythonのダックタイピングを活用して**オブザーバパターン**を実装し、MVCモデルについて論じよ
 - **課題内容:**
 1. **Observer**と**Subject**を設計し、観察者-被観察者関係を構築
 2. サンプルアプリとして簡易的なMVCモデルを実装
 3. 各コンポーネントの役割を論述
 4. **考察:** ダックタイピングの利点と、オブザーバパターンによる柔軟なシステム設計についてパターンを使わない場合と比較し、効能・使い道、論じる
-

作成したpythonコード、特に空欄やポイントの説明

```
## Ovserver Design Pattern for Python MVC
#####
`Model`クラスの設計
class Model: # 被観察者は、周りの視線(観察者)を気にせず、自分の生き方を貫く（我が道をゆく）
    def __init__(self, name):#このクラスはデータを保持
        self._name = name# 例 モデル1の引数
        self._data = None

    def set_data(self, data):#例dateには、イベント通知
        self._data = data

    def get_data(self):
        return self._data#アンダーバー一つは、内部的に使う変数 外部には公開しないでほしいという合
    図

    def get_name(self):
        return self._name
```

```
def serialize(self): # Modelのデータを直列化するメソッド
    return {"data": self._data}

def deserialize(self, data): # 直列化されたデータからModelを復元するメソッド
    self._data = data["data"] # 引数dataのdateキーの値をインスタンス変数に格納

##### `Observer` クラスの設計
class Observer: # 観察する人 このクラスは表示
    def update(self, model): # 共通メソッド: 【空欄A】
        pass

##### `View` クラスの設計 == 観察する人 : Text, Web, GUI, Html, CSV, Excel, PDF, いろんな表現があります

class HTMLView(Observer): # updateは通知
    def update(self, model): # 例 モデルクラスのインスタンスのメソッドを活用していると見れる Model. メソッド
        print(f"<div> HTMLView: {model.get_name()} updated to {model.get_data()}" </div>")

class CSVView(Observer): # カンマで区切られたファイル形式
    def update(self, model):
        print(f"CSVView, {model.get_name()}, updated, to, {model.get_data()}, ")

class PDFView(Observer):
    def update(self, model):
        print(f"@PDFView: {model.get_name()} {model.get_data()} updated")

class GUIView(Observer):
    def update(self, model):
        print(f"● GUIView: TEXT_FIELD:{model.get_name()} updated to {model.get_data()} CheckBox ✓")

##### `Controller` クラスの設計

class Subject: # 観察者と被観察者を間接的に仲介する主体
    def __init__(self): # このクラスは、更新と通知
        self._observers = [] # 観察者の集団

    def register_observer(self, observer): # 観察者を（複数）登録する
        self._observers.append(observer) # 【空欄B】

    def unregister_observer(self, observer): # 観察を外す、観察者を削除
        self._observers.remove(observer) # 【空欄C】

    def notify_observers(self, model): # すべての観察者に変化を通知する
        for observer in self._observers:
            observer.update(model) # 【空欄D】

class Controller(Subject): # イベント契機（イベント発生をきっかけ）に、何等かの変化を被観察者、 観察者に伝える
    def __init__(self, model):
```

```
super().__init__()#継承されたSubjectクラスのメソッドを呼び出し、インスタンス変数を活用できるようにしている
    self.model = model#モデルインスタンス

    def set_data(self, data):
        self.model.set_data(data)#例self.modelにはdateの引数で、引数としてイベント通知が格納されているがModelインスタンスが格納されている
        self.notify_observers(self.model) # 【空欄E】インスタンス自身

##### オブザーバパターンによるMVCプログラム

if __name__ == "__main__":
    # モデルの作成
    model1 = Model("主人公キャラ")#インスタンス化 ? self.nameには主人公キャラが格納
    model2 = Model("仲間キャラ")
    model3 = Model("武器")

    # 各種ビューの作成
    html_view = HTMLView()#インスタンス化
    csv_view = CSVView()#観察者クラス
    pdf_view = PDFView()
    gui_view = GUIView()

    # コントローラの作成
    controller1 = Controller(model1)#引数model1のコントローラーインスタンスを作成
    controller2 = Controller(model2)
    controller3 = Controller(model3)

    # コントローラにビューを登録
    controller1.register_observer(html_view)#各クラスのインスタンス、観察者をリストに登録
    controller1.register_observer(csv_view)
    controller2.register_observer(pdf_view)
    controller2.register_observer(gui_view)
    controller3.register_observer(html_view)
    controller3.register_observer(pdf_view)

    # コントローラのイベント発生で、モデルのデータが変更 ← 観察者が反応する
    controller1.set_data("イベント通知:Hi, HTML and CSV表現")
    controller2.set_data("イベント通知: Hello, PDF and GUI表現")
    controller3.set_data("イベント通知: neko, HTML and CSV表現")
```

動作確認結果、確認ポイントの説明

```
<div> HTMLView: 主人公キャラ updated to イベント通知:Hi, HTML and CSV表現 </div>
CSVView,主人公キャラ,updated,to,イベント通知:Hi, HTML and CSV表現,
@PDFView: 仲間キャラ イベント通知: Hello, PDF and GUI表現 updated
□●GUIView: TEXT_FIELD:仲間キャラ updated to イベント通知: Hello, PDF and GUI表現
CheckBox ✓
<div> HTMLView: 武器 updated to イベント通知: neko, HTML and CSV表現 </div>
```

@PDFView: 武器 イベント通知: neko, HTML and CSV表現 updated

分析

- オブザーバーパターンとは、非観察者の状態変化を、複数の観察者に通知するという仕組みを理解すること
- MVCの三つの要素に役割に分けて、そこから関係性を整理し理解すること
- 抽象化と継承の使い方を理解すること

考察

- MVCの役割をそれぞれ、データの保持、表示、更新と通知を理解しながら、何をしたいのかを考える
- コードの流れを追うことで、どこで何が起こっているのかを理解していく
- 各クラスの継承を行う理由を考える

改善点

-
-
-

チャレンジ問題5: 非同期処理の実装と分析 (骨太コース必須)

- **概要:** `async`と`await`を使用した非同期処理の設計と実装を通じて、非同期プログラミングを理解する
- **課題内容:**
 1. `async`と`await`を使用して複数の非同期タスクを並列実行するコード（例：Webスクレイピング、ファイル入出力）のサンプル例を実行し、動作原理を理解する
 2. 実行時間の比較実験を行い、同期処理との差を示す
 3. 非同期処理の利点、欠点、適用例を分析し、同期処理との違いを考察して、自分の言葉で説明せよ

作成したpythonコード、特に空欄やポイントの説明

ここにコードを書く

動作確認結果、確認ポイントの説明

ここに実行結果を書く

分析

-
-
-

考察

-
-
-

改善点

-
-
-

所感・今後の目標課題

(自由記述欄、多様なご意見を尊重します)

- プログラミングに興味が湧かない原因
 - プログラミングには興味がないが、それ以外で没頭していること
 - 授業中に質問や意見が言えない理由
 - コードや英語や文字、活字が好きにならない理由
 - 私とプログラミング
 - なぜプログラミングを学ぶのだろうか？
 - これまで学んだことを振り返り
 - これから学びに活かすべき知見
 - 締め切りを日曜まで延ばしてほしい
-