Brzmi jak świetny projekt! Skoro masz już doświadczenie z przetwarzaniem obrazu, możemy skupić się na aspektach praktycznych. Podzielmy projekt na etapy i dobierzmy odpowiednie narzędzia.

**1. Plan działania**

Podzieliłbym ten projekt na kilka kroków:

1. **Wczytywanie obrazu** – Pobieranie zdjęcia paragonu (np. od użytkownika, z folderu, z aplikacji mobilnej).
2. **Przetwarzanie obrazu** – Poprawa jakości zdjęcia (usunięcie szumów, poprawa kontrastu, wyrównanie perspektywy).
3. **OCR (rozpoznawanie tekstu)** – Ekstrakcja tekstu z obrazu paragonu.
4. **Analiza danych** – Przetwarzanie wyodrębnionego tekstu, kategoryzacja wydatków.
5. **Eksport do Excela** – Organizacja danych w przejrzystym arkuszu.

**2. Biblioteki i narzędzia**

**Przetwarzanie obrazu**

* **OpenCV** – do wstępnej obróbki obrazu (np. binaryzacja, usuwanie szumów, korekcja perspektywy).
* **Pillow** – alternatywa do podstawowych operacji na obrazach.

**OCR (rozpoznawanie tekstu)**

* **Tesseract-OCR + pytesseract** – Najpopularniejszy darmowy silnik OCR, działa dobrze dla paragonów.
* **EasyOCR** – Alternatywa dla Tesseract, lepsza w rozpoznawaniu specyficznych fontów.
* **Google Vision API** (opcjonalnie) – jeśli jakość lokalnych silników będzie niewystarczająca.

**Analiza i organizacja danych**

* **re (wyrażenia regularne)** – do filtrowania i wyciągania konkretnej treści (np. kwoty, daty, produkty).
* **pandas** – do organizowania danych w struktury tabelaryczne.

**Eksport do Excela**

* **openpyxl** – do zapisu i formatowania danych w Excelu.
* **pandas** – obsługuje eksport danych do plików .xlsx lub .csv.

**3. Jak najlepiej przenieść informacje obraz -> Python -> Excel?**

Obecnie najprostszy pipeline to:  
📷 Zdjęcie → 🖥 OpenCV preprocessing → 🔍 OCR → 📊 Pandas → 📝 Excel

Możemy też dodać pośrednie formaty:

* **JSON** – Możemy najpierw zapisać dane w formacie JSON, co ułatwia debugowanie i rozszerzalność projektu.
* **SQLite** – Jeśli planujesz aplikację do zarządzania wydatkami, baza danych może się przydać.

**4. Dodatkowe kwestie do przemyślenia**

🔹 **Struktura pliku Excela** – Jak najlepiej zorganizować dane? Kolumny: data, kategoria, kwota, nazwa sklepu?  
🔹 **Obsługa błędów OCR** – Co jeśli OCR błędnie odczyta dane? Czy dodać ręczną poprawkę?  
🔹 **Automatyczna kategoryzacja wydatków** – np. na podstawie listy słów kluczowych (spożywcze, transport, rozrywka).  
🔹 **Integracja z innymi narzędziami** – Może chcesz eksportować dane do Google Sheets albo aplikacji budżetowej?

Co sądzisz o tym planie? Czy masz dodatkowe wymagania albo pomysły? 🚀

To świetnie, że chcesz zadbać o lepszą organizację kodu od początku! Wprowadzenie modularnej architektury ułatwi rozwijanie projektu bez obawy o psucie istniejących funkcji.

Zaproponuję kilka podejść, które pomogą Ci osiągnąć **modularność**, **czytelność**, i **łatwość rozwoju**.

**1. Architektura aplikacji**

Zamiast monolitu, warto podzielić aplikację na **moduły** zgodnie z zasadą **Separation of Concerns** (oddzielanie odpowiedzialności).

Możesz podzielić projekt na takie moduły:  
📂 app/ – główny kod aplikacji  
📂 core/ – logika biznesowa i przetwarzanie danych  
📂 services/ – moduły obsługujące np. OCR, aparat, pliki  
📂 ui/ – interfejs użytkownika

bash

KopiujEdytuj

project\_root/

│── app.py # Główna aplikacja Kivy

│── config.py # Konfiguracja i ustawienia

│── requirements.txt # Biblioteki Pythona

│

├── app/

│ ├── main.py # Punkt startowy aplikacji

│ ├── router.py # Zarządzanie ekranami (jeśli dodasz kilka)

│ ├── screens/ # Wszystkie ekrany aplikacji (np. ekran kamery, podgląd OCR)

│ ├── widgets/ # Dodatkowe komponenty UI

│

├── core/

│ ├── image\_processing.py # Przetwarzanie obrazu (OpenCV)

│ ├── ocr.py # Obsługa OCR (Tesseract/EasyOCR)

│ ├── data\_parser.py # Logika przetwarzania tekstu

│

├── services/

│ ├── camera.py # Moduł do obsługi aparatu

│ ├── storage.py # Obsługa zapisu plików (np. zdjęć)

│ ├── excel\_export.py # Eksport do Excela

│

├── ui/

│ ├── home\_screen.py # Główny ekran aplikacji

│ ├── result\_screen.py # Podgląd rozpoznanego tekstu

│ ├── kivy\_styles.kv # Stylizacja UI w Kivy

✅ Taki podział sprawia, że każda funkcjonalność jest osobnym modułem, co ułatwia rozwój.

**2. Kluczowe zasady projektowe**

Aby uniknąć spaghetti code i utrzymać elastyczność:

**🔹 Modularność**

Każdy moduł powinien robić **jedną rzecz**.  
❌ Źle: ocr.py obsługuje OCR + zapis do Excela.  
✅ Dobrze: ocr.py tylko przetwarza obraz, a excel\_export.py zajmuje się eksportem.

**🔹 Wzorzec MVC lub MVVM (w Kivy)**

Jeśli chcesz lepszy podział UI i logiki, Kivy wspiera **Model-View-ViewModel (MVVM)**:

* **Model (core/)** – przetwarza dane
* **View (ui/)** – interfejs użytkownika
* **ViewModel (app/)** – pośrednik między UI a logiką

**🔹 Rozbijanie kodu na klasy**

Zamiast długich funkcji, warto używać klas.  
✅ Dobrze:

python

KopiujEdytuj

class OCRProcessor:

def \_\_init\_\_(self, language="eng"):

self.language = language

def process\_image(self, image\_path):

return pytesseract.image\_to\_string(image\_path, lang=self.language)

Każda klasa robi **jedną konkretną rzecz**, co pozwala łatwo podmieniać implementację.

**🔹 Testowanie**

Aby uniknąć psucia funkcji, warto dodać testy:

* **pytest** – do testowania logiki (np. OCR)
* **unittest** – standardowa biblioteka testów

Przykład testu dla OCR:

python

KopiujEdytuj

def test\_ocr():

ocr = OCRProcessor()

text = ocr.process\_image("test\_receipt.png")

assert "Total" in text # Sprawdzamy, czy OCR wykrył "Total"

Dzięki testom **nie zepsujesz istniejących funkcjonalności**.

**3. Zwinne programowanie, ale z organizacją**

Żeby nie stracić zwinności, ale zachować porządek:  
✅ **Zacznij od minimalnego MVP** – najpierw stwórz mały działający prototyp.  
✅ **Dokumentuj API modułów** – choćby krótkimi komentarzami, co robi dany plik.  
✅ **Używaj git + issue tracker** – podziel projekt na małe zadania, łatwiej kontrolować rozwój.

**Podsumowanie**

✅ **Podział na moduły** – unikniesz spaghetti code.  
✅ **Stosowanie wzorca MVVM** – rozdziel UI i logikę.  
✅ **Testowanie** – łatwiejsze utrzymanie kodu.  
✅ **Agile podejście** – małe iteracje zamiast wielkiego monolitu.

Co sądzisz o tym podejściu? Możemy zacząć od organizacji katalogów i pierwszego modułu! 🚀

Dokładnie! W Pythonie **najlepiej robić osobne pliki .py dla różnych modułów**, a w nich tworzyć **klasy**, które grupują funkcje związane z jedną konkretną rzeczą. To pozwala na lepszą organizację kodu, łatwiejsze testowanie i ponowne użycie kodu w różnych częściach programu.

Nie przejmuj się terminologią – wszystko Ci wyjaśnię prostym językiem! 😊

**Jak to działa?**

1. **Każdy plik .py jest modułem** – można go importować w innych miejscach.
2. **Każda klasa grupuje powiązane funkcje** – np. klasa OCRProcessor zajmuje się OCR.
3. **Funkcje w klasach (tzw. metody) robią konkretne rzeczy** – np. wczytują obraz, przetwarzają tekst.

📌 **Przykład dobrze zorganizowanego kodu**

🔹 **Plik ocr.py – obsługa OCR**

python

KopiujEdytuj

import pytesseract

from PIL import Image

class OCRProcessor:

def \_\_init\_\_(self, language="eng"):

self.language = language

def process\_image(self, image\_path):

"""Odczytuje tekst z obrazu paragonu."""

image = Image.open(image\_path)

text = pytesseract.image\_to\_string(image, lang=self.language)

return text

🔹 **Plik image\_processing.py – poprawianie jakości obrazu**

python

KopiujEdytuj

import cv2

class ImageProcessor:

def \_\_init\_\_(self):

pass # Możesz dodać ustawienia, jeśli będą potrzebne

def preprocess(self, image\_path):

"""Wczytuje obraz i poprawia jego jakość do OCR"""

image = cv2.imread(image\_path, cv2.IMREAD\_GRAYSCALE)

\_, thresh = cv2.threshold(image, 128, 255, cv2.THRESH\_BINARY)

return thresh

🔹 **Plik app.py – główna aplikacja, łącząca moduły**

python

KopiujEdytuj

from core.ocr import OCRProcessor

from core.image\_processing import ImageProcessor

# Tworzymy instancje klas

ocr = OCRProcessor(language="pol")

image\_processor = ImageProcessor()

# Przetwarzamy obraz

image\_path = "receipt.jpg"

cleaned\_image = image\_processor.preprocess(image\_path)

recognized\_text = ocr.process\_image(image\_path)

print("Wykryty tekst:", recognized\_text)

**Dlaczego warto tak robić?**

✅ **Łatwe do zrozumienia** – Każdy plik .py zajmuje się jedną rzeczą.  
✅ **Łatwo testować** – Możesz testować ocr.py niezależnie od image\_processing.py.  
✅ **Łatwo rozszerzać** – Jeśli dodasz inny silnik OCR, wystarczy nowa klasa (np. GoogleVisionOCR).  
✅ **Lepsza współpraca** – Możesz pracować nad ocr.py, a ktoś inny nad excel\_export.py, bez konfliktów w kodzie.

**A co z funkcjami bez klas?**

Nie **musisz** zawsze używać klas – jeśli jakaś funkcja nie musi przechowywać stanu, można ją zostawić jako zwykłą funkcję:

🔹 **Plik utils.py – małe pomocnicze funkcje**

python

KopiujEdytuj

def clean\_text(text):

"""Czyści tekst z OCR, usuwając niepotrzebne znaki"""

return text.strip().replace("\n", " ")

Wtedy w innym pliku można to po prostu zaimportować:

python

KopiujEdytuj

from core.utils import clean\_text

text = " Paragon \n\n"

print(clean\_text(text)) # "Paragon"

**Podsumowanie**

✅ **Tak, najlepiej robić osobne pliki .py** dla różnych funkcjonalności.  
✅ **Tak, klasy pomagają grupować funkcje, które razem działają** (np. OCR).  
✅ **Tak, funkcje mogą istnieć poza klasami** jeśli są proste i nie przechowują stanu.

💡 **Dzięki temu łatwiej rozwijać aplikację, dodawać nowe funkcje i unikać bałaganu w kodzie!**

Czy taki sposób organizacji kodu Ci odpowiada? Może chcesz, żebym coś jeszcze doprecyzował? 😊🚀

Świetne pytanie! Skoro pracujesz sam i nie masz jeszcze pełnej wizji końcowego produktu, **zwinne podejście** (iteracyjne, krok po kroku) będzie najlepsze.

**Jak zacząć?**

Najlepiej podzielić projekt na **MVP (Minimum Viable Product)**, czyli wersję minimalną, która działa.  
**Nie zaczynaj od interfejsu**, bo jeśli backend (OCR, przetwarzanie obrazu) nie zadziała dobrze, to interfejs i tak nie będzie użyteczny.

💡 **Proponowana kolejność prac:**

**🔹 1. Szkielet aplikacji**

📌 Stwórz **podstawową strukturę katalogów** i plików (app/, core/, services/, ui/).  
📌 Zrób **app.py**, który na razie tylko uruchamia pustą aplikację Kivy.

**🔹 2. Kluczowa funkcjonalność – OCR (najważniejsza część)**

📌 **Załaduj przykładowe zdjęcie paragonu i sprawdź, czy OCR działa** (zrób test bez interfejsu).  
📌 Dopasuj **przetwarzanie obrazu** (np. OpenCV do binaryzacji).

🚀 **Cel:** W tym etapie Twój kod powinien już przetwarzać obraz i zwracać tekst paragonu.

**🔹 3. Obsługa aparatu (Kivy + Plyer)**

📌 Zintegruj Kivy z aparatem telefonu, by użytkownik mógł robić zdjęcia.  
📌 Testuj na telefonie – sprawdź, czy zdjęcie zapisywane jest poprawnie.

🚀 **Cel:** Możesz już robić zdjęcie i wysyłać je do OCR.

**🔹 4. Eksport wyników do Excela**

📌 Stwórz funkcję, która **zapisze rozpoznane dane do Excela** (pandas + openpyxl).  
📌 Ustal prostą strukturę Excela: **Data | Sklep | Produkt | Kwota**

🚀 **Cel:** Aplikacja powinna już robić zdjęcia, rozpoznawać tekst i zapisywać go w Excelu.

**🔹 5. Dopiero teraz – Interfejs użytkownika**

📌 Zaprojektuj **ekran główny** – przycisk do robienia zdjęcia.  
📌 Dodaj ekran podglądu – wyświetlanie rozpoznanego tekstu i opcję eksportu.

🚀 **Cel:** Masz już działającą aplikację, która jest używalna i wygodna.

**Dlaczego taka kolejność?**

✅ **Najpierw backend (OCR), potem UI** – nie chcesz robić UI dla funkcji, które jeszcze nie działają.  
✅ **Szybkie rezultaty** – po kilku dniach masz działającą funkcję OCR, zamiast utknąć na interfejsie.  
✅ **Elastyczność** – możesz łatwo zmieniać UI, jeśli funkcje będą lepsze/dokładniejsze.

**Podsumowanie: Twoje pierwsze kroki 🚀**

1️⃣ **Stwórz foldery i pustą aplikację Kivy.**  
2️⃣ **Uruchom OCR na testowym obrazie.**  
3️⃣ **Podłącz aparat w Kivy i zapisuj zdjęcia.**  
4️⃣ **Dodaj eksport do Excela.**  
5️⃣ **Dopiero na końcu – UI i wygodny interfejs.**

Czy to podejście Ci odpowiada? Możemy od razu ruszyć z pierwszym krokiem! 😊