# はじめに

本書は、Python で GUI 操作を自動化して自作の RPA 自動化ツールを作成したい方に向けた入門書です。Python の入門書を読み終えたばかりの方でも、コード解説を読み、実際に手を動かして作りながら、明日から現場で使える無料の RPA 自動化ツールを開発するスキルを習得出来るように執筆しています。

本書を読み終えた後に実務で自動化 RPA ツールを開発して業務効率化を実現する事を目的としています。実践編では、実務で特に連携のニーズが高い Excel、ブラウザアプリ、機械翻訳ツール、OCR エンジンを連携させて 2 つの自動化ツールを作成します。単純に動作するプログラムのコード解説ではなく、RPA 自動化ツールを作るまでの一連の手順を踏んでツールを開発してきます。実践編を終える頃には、明日から実務で使える RPA 自動化ツールを開発の実践力が身に付いている事でしょう。

本書が皆様の RPA 自動化ツール開発のきっかけになれば、著者として望外の喜びです。

# 本書について

本書は、以下のような読者を対象としています。Python を全く学んだ事がない方でも、Python の入門書を片手に読み進めていただければ十分に理解出来る内容となっています。本書を読み進めて、わからない部分だけを入門書で確認すればよいでしょう。実践編を終える頃には、ご自分の RPA 自動化ツールを無料で作成する基礎を身に付ける事が出来ます。

・業務の自動化を RPA で実現したいけど、何から初めて良いか分からない方

・RPA に興味はあるけど、市販の RPA は高額で無料の自作 RPA を探している方

・Python の入門書を読み終えたので、次のステップアップを検討されている方

## 本書の概要

Python で RPA 自動化ツールを作成するには PyAutoGUI というライブラリを使用します。PyAutoGUI は Python で GUI 操作（マウスやキーボード操作）を自動化するための必須ライブラリであり、無料で使用することが出来ます。難しいプログラムを作成しなくても、マウスやキーボード操作で出来る事を簡単に自動化することが出来ます。実践編で紹介しますが、わずかな学習コストで Kindle 電子書籍の自動翻訳ツールを作成することも可能です。

プログラミングを習得するには、実際にコーディングしてプログラムを動かしながら学ぶことが重要です。Visual Studio Code 上で実際にプログラムを動かしながら学べるように、サンプルコードを多数掲載しています。実際にコーディングしながら学ぶ事をお勧めします。こちらのサイトからサンプルコードをダウンロード出来るようにしています。ご自身の PC 上でプログラムを実際に動かしながら使い方の理解を深めて下さい。サンプルコードは、書籍を購入いただいた方への特典となりますので使用するにはパスワードが必要です。パスワードは、本書の一番後ろに記載しております。

## 本書の構成

### 1 章

Python で RPA 自動化ツールを開発するための環境構築をします。PyAutoGUI というライブラリの導入とその事前準備について説明します。プログラミングを始める際の最初の障壁が開発環境の構築です。Python のインストールから、その他のライブラリを簡単にインストールするためのツール、アプリ開発の効率を上げてくれるツールの使い方などを解説します。また、今回は OCR 処理と機械翻訳とも連携させますので、OCR エンジンと機械翻訳アプリのインストールについても解説します。

### 2 章

Python の RPA ライブラリである PyAutoGUI の使い方について詳しく解説します。最初に市販の RPA と PyAutoGUI の違いや PyAutoGUI の優位性を解説します。次に、PyAutoGUI によるマ

ウス操作やキーボード操作などを実際に簡単なプログラムを作成しながら学びます。PyAutoGUI はシンプルなライブラリですので、この章で PyAutoGUI をマスターする事が出来ます。

**3 章**

実践編 1 では、Google Maps を使った走行距離チェックツールを作成します。このツールでは、ブラウザベースの Google Maps とエクセルの 2 つのアプリを連携させて自動化します。エクセルには、出発地と目的地が入力されており、それらの値を Google Maps に入力して走行距離を検索します。OCR で走行距離を抽出し、その結果をエクセルに転記する自動化ツールです。複数のアプリを Python の RPA 自動化ツールで連携させるための基礎を学ぶ事が出来ます。

**4 章**

実践編 2 では、本書の仕上げとして、Kindle 電子書籍の自動翻訳ツールを作成します。2 章、３章で学んだ内容と DeepL 翻訳アプリを組み合わせて、実践的な RPA 自動化ツールを開発します。このツールでは、電子書籍の画像データから文字情報を OCR で抽出し、DeepL アプリで機械翻訳を実施します。この操作を繰り返し実行する事で、電子書籍を丸々 1 冊自動翻訳する事も可能です。自作の RPA 自動化ツールの最大の利点は、キーボード操作やマウス操作などの GUI 操作を自在にコントロールして自動化出来る事であり、複数のアプリを繋げられる事です。実践編 1、2 を通じて複数のアプリを組み合わせるための考え方やプログラムの作成方法などを学び、実際に現場で使える RPA 自動化ツールを開発する基礎を身に付けることが出来ます。

## 対象 OS

著者が動作を確認している環境は次の通りです。

・Windows10 Home, Pro

・Python = 3.9

### 2.6.1 スクリーンショット関数

screenshot()関数は戻り値として、Pillow の Image オブジェクトを返します（詳しくは Pillow 公式ドキュメントを参照してください）。引数に画像のファイル名を指定すると、スクリーンショットを画像として保存し、戻り値として Image オブジェクトを返します。ファイル名の代わりにパスを渡せば、所定の場所に画像を保存出来ます。以下に screenshot()関数の使用例コードを示します。ただし、絶対パスを指定する場合にはパスの先頭に r を付けて raw 文字列とする事に注意して下さい。また、**username** のところはご自身のユーザーネームに書き換えて下さい。ご自身のユーザーフォルダ直下とプログラムファイルと同じ階層に my_screenshot.png という画像が作成され、スクリーンショットが保存されていると思います。

Code. 2.6.1.py

```
import pyautogui as ag


# スクリーンショット画像を変数 img に格納
img = ag.screenshot()
# プログラムファイルと同じ階層にスクリーンショット画像を保存
ag.screenshot('my_screenshot.png')
# username フォルダ直下にスクリーンショット画像を保存
ag.screenshot(r'C:\Users\username\my_screenshot.png')
```

PyAutoGUI 公式ドキュメントによると 1920 x 1080 の解像度では、screenshot()関数の処理におおよそ 100 ミリ秒かかるようです。早くも、遅くもありませんが反射力が求められるゲームなどで多用する場合には処理時間は気になります。オプションで任意の領域のみをキャプチャーする事も出来ます。引数 region にキャプチャーする領域の(左上の x 座標、左上の y 座標、幅、高さ)の 4 つの整数をタプル形式で指定します。
region を指定した方が処理速度も速くなりますので、取得したい領域が限られている場合には領域指定した方が良いでしょう。本書でも基本は **Region** を指定してスクリーンショットを使います。領域を指定することで余計な画像を取り込まないので、画像マッチングや OCR 処理の精度低下を防げます。例えば、画面左上（x = 0, y = 0）から幅 100 ピクセル、高さ 200 ピクセルの領域をキャプチャーしたい場合は、以下のようなコードになります。

```
import pyautogui as ag
img = ag.screenshot(region=(0,0, 100, 200))
```

## 2.6.2 画像マッチング関数

locateOnScreen()関数は、画面上の正確な座標が分からない場合に使用します。引数には画像マッチングさせたいファイル名を指定します。座標を検出したい画像を予め準備し、画像マッチングでその画像が表示されているスクリーン上の座標を検出出来ます。戻り値は、画像マッチで検出された領域の座標(left=領域左上の x 座標, top=領域左上の y 座標, width=領域の幅, height=領域の高さ) のタプルになります。画像がスクリーン上で見つけられなかった場合は、戻り値として ImageNotFoundException を返します。戻り値として、画像マッチで検出した領域の中心座標を取得したい場合は、locateCenterOnScreen()関数を使用します。オプションでキーワード引数 confidence に画像マッチングの信頼度を指定する事も出来ます。画像マッチングでは、信頼度が 100%になる事はほとんどありません。このため、confidence を 0.6（60%)以上などに指定する事で ImageNotFoundException を防ぐ事が出来ます。この値を下げ過ぎてしまうと間違った画像も検出してしまいますので微調整が必要です。この confidence を使うためには、OpenCV がインストールされている必要があります。本書では、1 章で OpenCV をインストールしていますので問題なく使えます(プログラム中で import する必要はありません)。

例えば、3 章では Google Maps で距離検索を実施しますが、交通手段を車で検索したい場合は車ボタンをクリックする必要があります。また、プログラムの途中で交通手段を徒歩に変更したい場合などにも、徒歩ボタンの座標が予め把握出来ていないと交通手段を変更する事が出来ません。このような場合に画像マッチングを使用すれば、選択したいボタンの画像を予め用意していれば画面上からボタンの座標を検出する事が出来ます。

図 2.6.2.1 から図 2.6.2.2 に示す car_button.png と walking_button.png の座標を画像マッチングで検出し、座標をターミナルに出力させるコードは以下の通りです。事前に図 2.6.2.3 のようにプログラムファイルと同じ階層に car_button.png と walking_button.png を保存しておきます。以下のコードでは、領域指定していませんので筆者の環境（ディスプレイの解像度：2560 x 1440）では処理に 0.7 秒ほどかかりました。

Code. 2.6.2.py #1

```
import pyautogui as ag


car_position = ag.locateOnScreen('car_button.png')
walking_position = ag.locateOnScreen('walking_button.png')


print(car_position)
print(walking_position)



参考ですが、出力結果は以下のようにターミナルに出力されます。

Box(left=108, top=133, width=31, height=24)
Box(left=188, top=127, width=40, height=37)
```

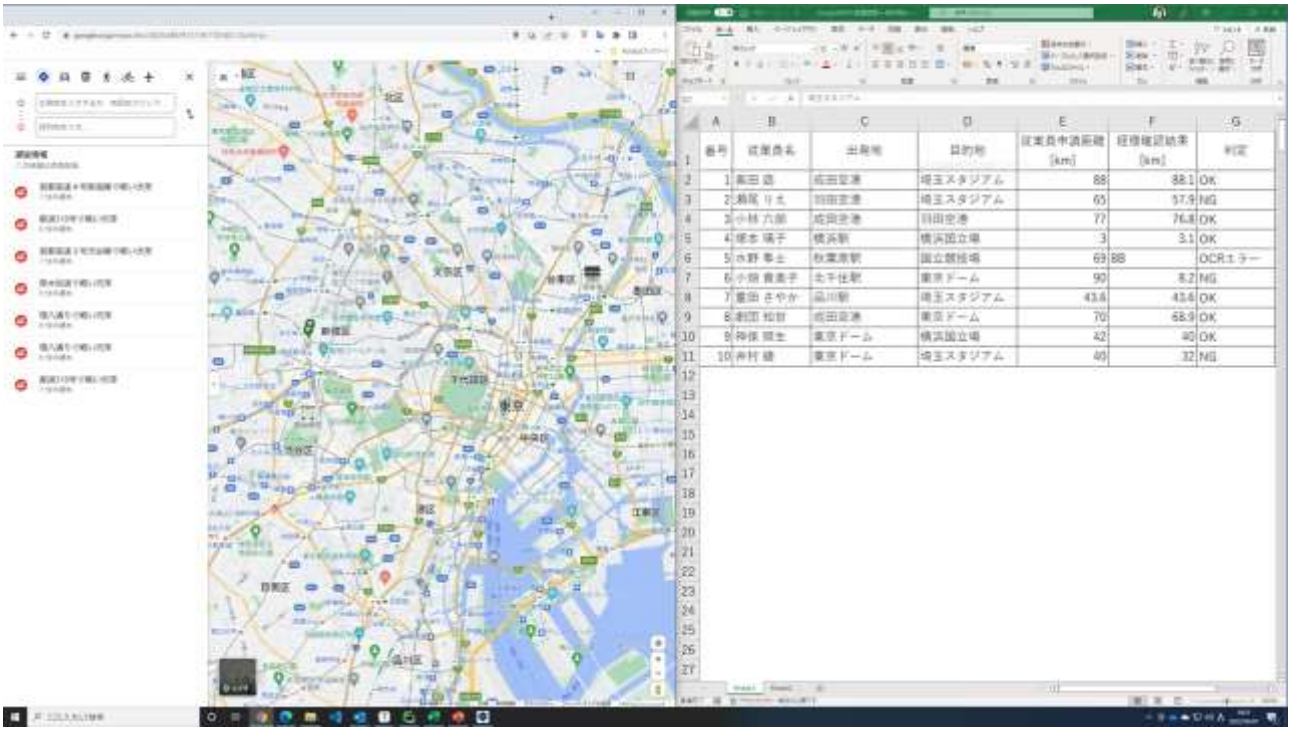

図 2.6.2.1 画像マッチングの対象領域


図 2.6.2.2 画像マッチング対象のボタン

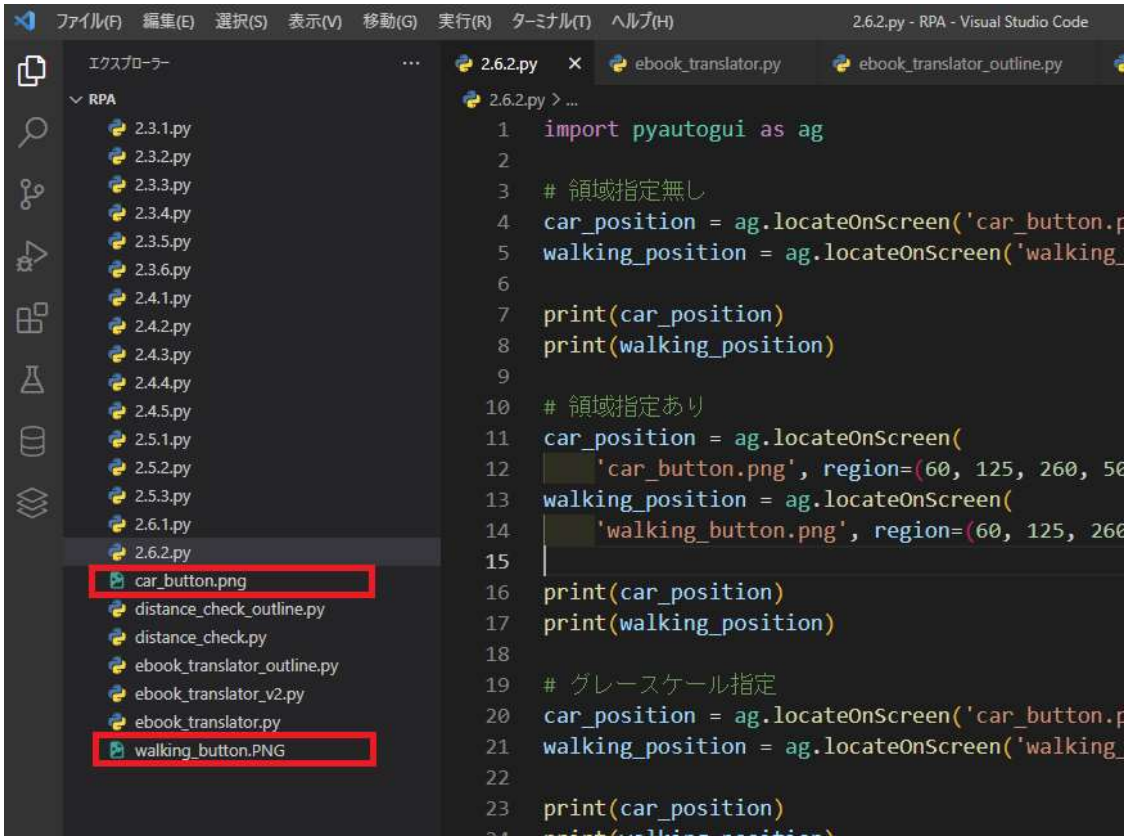


図 2.6.2.3 画像をプログラムファイルと同じ階層に保存

次に、処理時間を短くするため画像マッチングを実施する領域を図 2.6.2.4 に示すように指定します。コードは以下の通りです。この場合の処理時間は、筆者の環境で 0.15 秒ほどでした。画像マッチングする領域を（2560 x 1440）から（260 x 35）に制限したので、かなり高速化する事が出来ました。処理速度が気になるようであれば、領域を指定する事を検討下さい。

Code. 2.6.2.py #2

```
import pyautogui as ag


car_position = ag.locateOnScreen(
    'car_button.png', region=(60, 125, 260, 35))
walking_position = ag.locateOnScreen(
    'walking_button.png', region=(60, 125, 260, 35))


print(car_position)
print(walking_position)
```

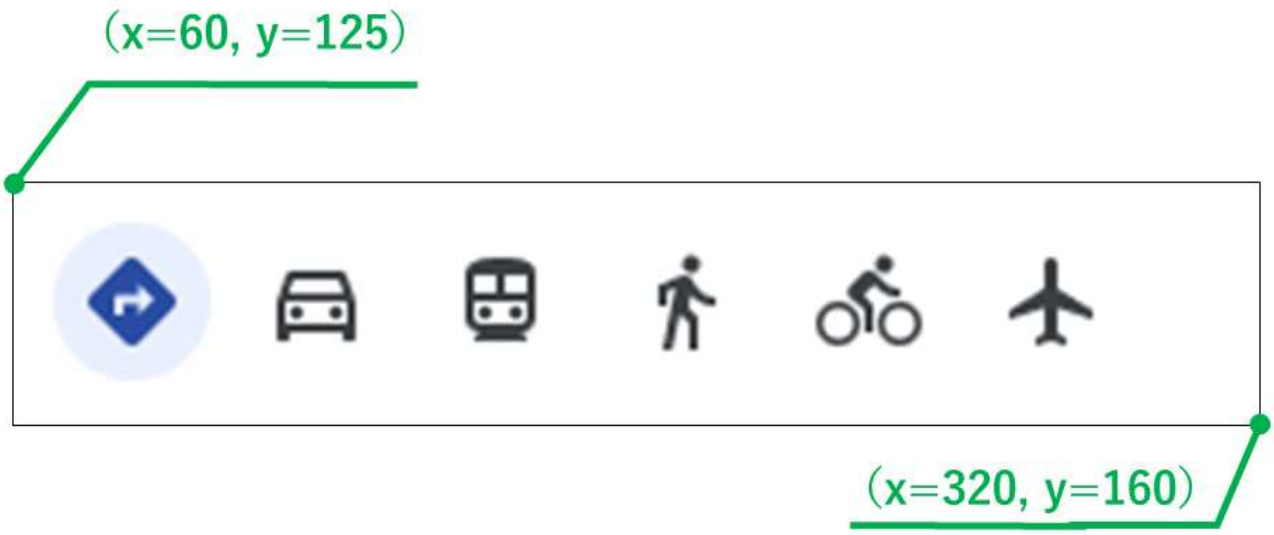


図 2.6.2.3 画像マッチングの領域指定

処理速度を高速化させるためのオプションとしてグレースケールでの画像マッチングを実施する事も出来ます。引数 grayscale=True を指定すると画像マッチングがグレースケールで処理されます。コードは以下の通りです。筆者の環境では、グレースケールを指定する事で処理速度が 30%～40%ほど高速化されました。

Code. 2.6.2.py #3

```
import pyautogui as ag


car_position = ag.locateOnScreen('car_button.png', grayscale=True)
walking_position = ag.locateOnScreen('walking_button.png', grayscale=True)


print(car_position)
print(walking_position)
```

検出したい画像がスクリーン上に複数ある場合には、locateAllOnScreen()関数を使用します。locateOnScreen()関数では、画像マッチングでスクリーン上で最初に検出された座標 1 つを出力しますが、locateAllOnScreen()関数では検出した複数の座標を出力出来ます。以下に使用例のコードを示します。ここでは、画像マッチングの信頼度を 0.6 以上にしています。

Code. 2.6.2.py #4

```
import pyautogui as ag


car_position = ag.locateAllOnScreen('car_button.png',confidence = 0.6)


for position in car_position:
    print(position)
```

locateAllOnScreen()関数の戻り値は、画像マッチングで信頼度 0.6 以上で検出された全ての座標のジェネレータになります。ジェネレータの概念は分かり難い部分もありますが、難しく考えずに検出したすべての座標を取得し、座標を取り出す時には以下のコードにあるように for ループ文で 1 つずつ取り出す事が出来ると覚えて下さい。

参考ですが、筆者の環境では出力結果は以下のようにターミナルに出力されました。信頼度を 0.6 まで下げましたので、1～2 ピクセルだけずれた座標が 5 つ出力されました。

```
Box(left=108, top=132, width=31, height=24)
Box(left=107, top=133, width=31, height=24)
Box(left=108, top=133, width=31, height=24)
Box(left=109, top=133, width=31, height=24)
Box(left=108, top=134, width=31, height=24)
```

# 3.実践編 1:Google Maps を使った走行距離チェックツール

## 3.1 Google Maps を使った走行距離チェックツールの概要

本章と次章では、PyAutoGUI を使った実践的な RPA 自動化ツールを開発します。本章では、図 3.1 で示すような経理部門で行っている Google Maps を使った走行距離のチェック業務を自動化します。ある会社では出張で私有車を使用した場合、走行距離に応じてガソリン代が支給されます。従業員は、出張管理システムに出張の出発地、目的地、走行距離を登録します。経理部門では、その申請内容をシステムからエクセル形式でダウンロードし、Google Maps を使って走行距離に大きな乖離が無いかをチェックしています。このチェック業務は単純作業ですが、人手で対応していたため多くの工数がかかっていました。また、あまりにも単調で誰でも出来る業務のため、誰もやりたがらないという事も問題でした。このような付加価値が低い単純作業は RPA による自動化に適しています。RPA 自動化ツールでエクセル形式の申請内容を読み込み、Google Maps で距離検索し、結果をエクセルに記入し、この作業がすべて終わるまで繰り返します。


図 3.1 経理部門のチェック業務自動化

## 3.2 自動化する作業の把握

図 3.1 で自動化する作業の概要は掴んでいただけたと思います。ここからは、PyAutoGUI で自動化するために、大まかに作業を分解していきます。図 3.2 に自動化する作業の順番に番号を振り可視化してみました。自動化を始める前に、画面右側にエクセルを開き、左側にブラウザで Google Maps を開きます。Google Maps では、走行距離を調べたいのでルート検索のページを表示させて車のマークをクリックしておきます。そして、① ～ ⑦の順で従業員が申請した距離が正しかをチェックします。この操作を PyAutoGUI で自動化します。大まかな自動化したい処理は把握出来ましたが、このままではまだ自動化出来ません。もう少し各作業を PyAutoGUI で自動化出来る程度に詳細に検討します。

① Excel 上で出発地を取得する。

② Google Maps 上で出発地を入力する。

③ Excel 上で目的地を取得する。

④ Google Maps 上で目的地を入力し、検索実行する。

⑤ Google Maps 上で検索結果（距離）を取得する。

⑥ Excel 上で経理確認結果に検索結果（距離）を入力する。

⑦ Excel 上で従業員申請距離と経理確認結果を比較して、判定結果を入力する。

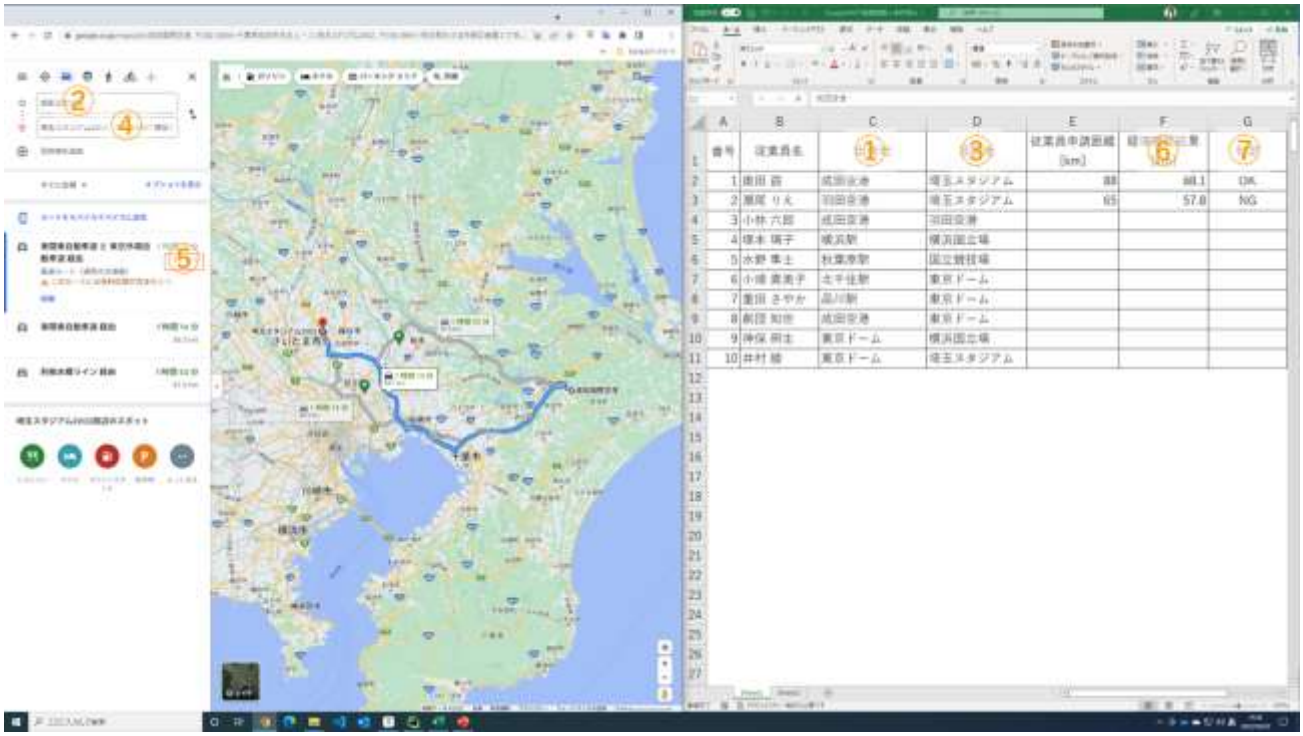


図 3.2. 自動化する作業の順番

# 4.実践編 2:Kindle 電子書籍の自動翻訳ツール

## 4.1 Kindle 電子書籍の自動翻訳ツールの概要

この章では、Kindle 電子書籍の自動翻訳ツールを開発します。Kindle 電子書籍にはコピー制限が付いている書籍が多いので、機械翻訳をしたくても直接文字をコピーする事が出来ません。このような場合には、OCR 機能で文字列に変換してから機械翻訳にかけます。機械翻訳には、世界一高精度の翻訳ツールである DeepL のデスクトップアプリを使用します。アプリでは、5000 文字までは無料で何度でも使用出来ますので、無料の自動翻訳ツールを PyAutoGUI と組み合わせて作成する事が出来ます。Kindle に限らず、PDF ファイルや画像などから文字列を抽出したい場合には OCR 機能は必要になります。機械翻訳作業も単純作業ですので RPA で自動化したい処理の一つだと思います。今回のプログラムを少し改良すれば、さまざまな用途に応用出来ますので是非とも活用してみて下さい。

今回は、図 4.1 に示すような電子書籍の翻訳作業を自動化します。Kindle などの電子書籍 Viewer で電子書籍を表示させ、OCR 機能で文字列に変換し、DeepL などの機械翻訳ツールで翻訳する作業を電子書籍の各ページで実施し、翻訳結果をファイルに書き出します。


図 4.1 電子書籍の翻訳作業の自動化

## 4.2 自動化する作業の把握

図 4.1 で自動化の概要は掴んでいただけたと思います。ここからは、PyAutoGUI で自動化するために、大まかに作業を分解していきます。図 4.2 に自動化する作業の順番に番号を振り可視化してみました。画面構成ですが、左半分に Kindle の電子書籍を表示させ、右半分に DeepL デスクトップアプリを表示させます。自動翻訳ツールの処理手順ですが、①～③の作業を繰り返し、その後④でファイルを出力します。この操作を PyAutoGUI で自動化します。大まかな自動化したい処理は把握出来ましたが、このままではまだ自動化出来ません。もう少し各作業を PyAutoGUI で自動化出来る程度に詳細に検討します。

① 電子書籍エリアを OCR 処理で文字列に変換。

② 原文エリアに OCR で抽出した文字を貼り付けし翻訳実行。

③ 訳文エリアから翻訳結果を取得。

④ 翻訳結果をテキストファイルに書き出し。

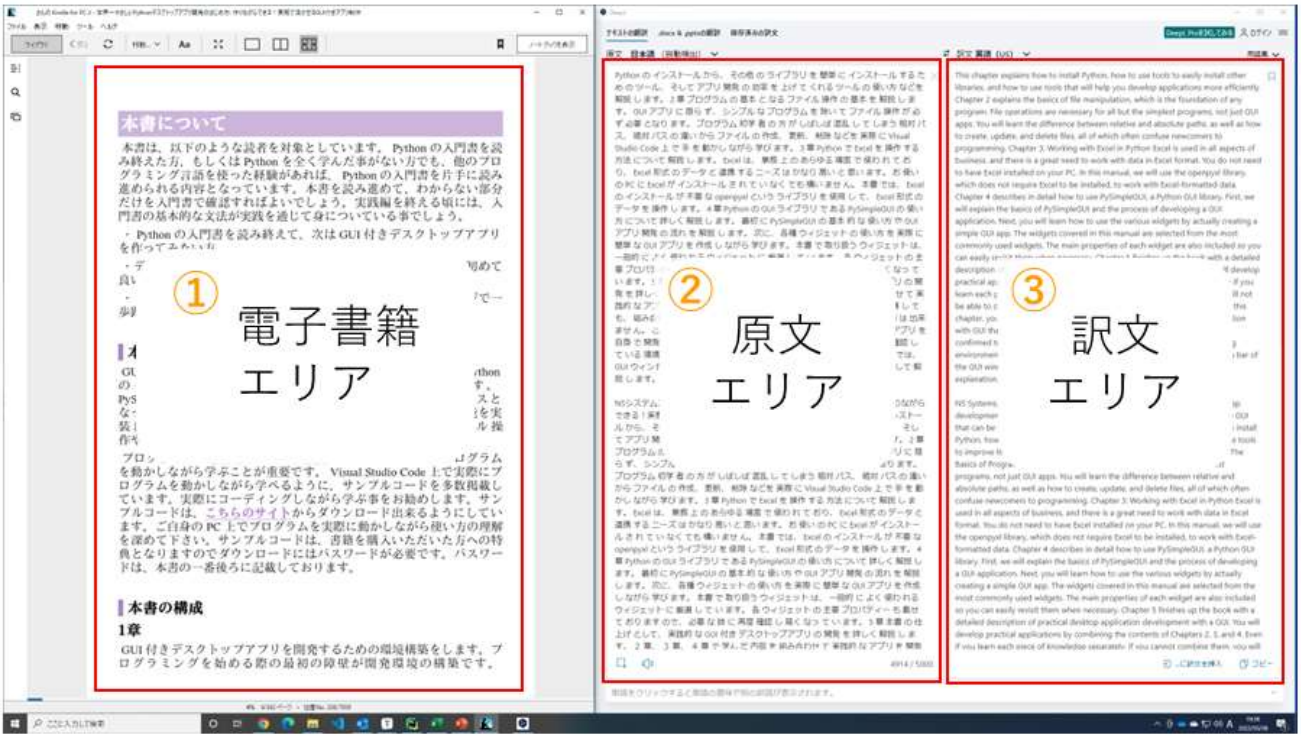


図 4.2. 画面構成と各エリアの説明 1