**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN, ĐHQG-HCM**

**KHOA CÔNG NGHỆ THÔNG TIN**

---o0o---

Logo

Description automatically generated with medium confidence

**BÁO CÁO**

**ĐỒ ÁN LẬP TRÌNH SOCKET:**

**ĐIỀU KHIỂN TỪ XA BẰNG EMAIL**

**Thành viên nhóm MSSV**

Đỗ Phan Tuấn Đạt 22127057

Phạm Thành Đạt 22127064

Lê Hồ Phi Hoàng 22127123

Trần Nguyễn Minh Hoàng 22127131

**Lớp:** 22CLC02

**Môn học:** Mạng máy tính

**Học kỳ:** 1

**Năm học:** 2023-2024

# Mục lục

[I. Mục lục 2](#_Toc151024724)

[II. Môi trường làm việc, thư viện hỗ trợ 3](#_Toc151024725)

[1. Môi trường làm việc: 3](#_Toc151024726)

[2. Ngôn ngữ: 3](#_Toc151024727)

[3. Thư viện hỗ trợ 3](#_Toc151024728)

[III. Danh sách file trong source code 3](#_Toc151024729)

[IV. Phân tích, giải thích source code 3](#_Toc151024730)

[V. Hướng dẫn sử dụng 3](#_Toc151024731)

[VI. Đóng góp 3](#_Toc151024732)

[VII. Tài liệu tham khảo 3](#_Toc151024733)

# Môi trường làm việc, thư viện hỗ trợ

## Môi trường làm việc:

Visual Studio Code

## Ngôn ngữ:

Python

## Thư viện hỗ trợ

* **os**: Cho chức năng hệ điều hành
* **signal**: Gửi tín hiệu đến các tiến trình
* **psutil**: Quản lí tiến trình
* **subprocess**: Chạy lệnh bên ngoài
* **tabulate**: Định dạng dữ liệu thành bảng
* **BeautifulSoup4**: Xử lí chuỗi có chứa HTML tag

# Danh sách file trong source code

* appController.py: Chứa class AppController cùng các hàm dùng trong các hoạt động liên quan đến app
* processController.py: Chứa class ProcessController cùng các hàm dùng trong các hoạt động liên quan đến process

# Phân tích, giải thích source code

## appController.py

Class **AppController()**: bao gồm các hàm liên quan đến điều khiển ứng dụng:

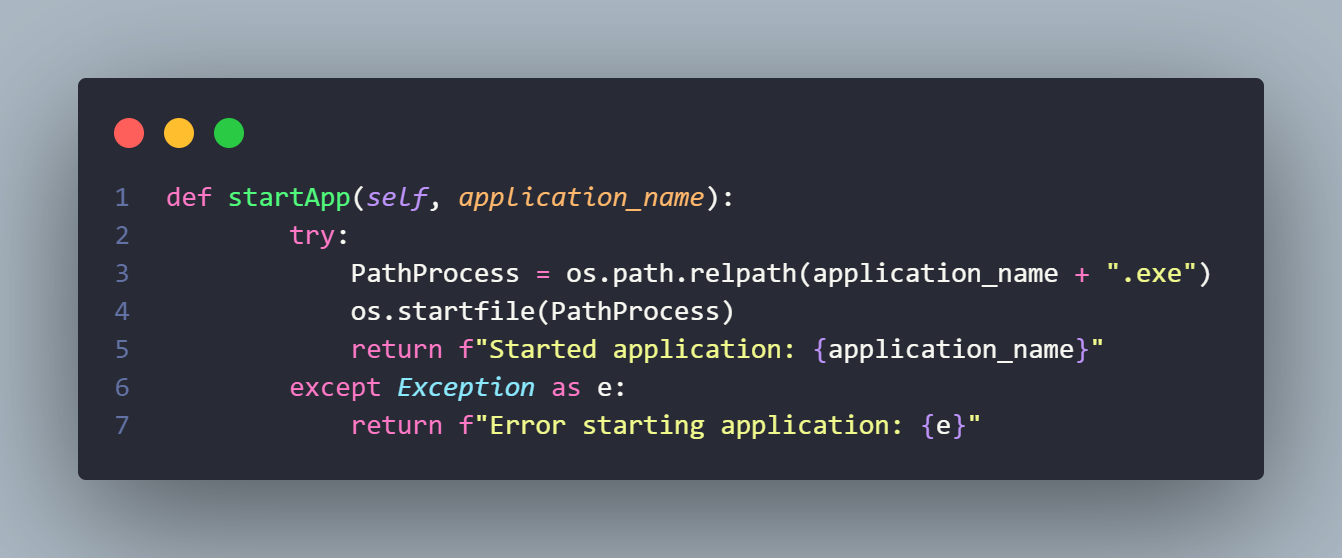
* *process2List*:



* **a = processes.decode().strip()**: Chuyển đổi chuỗi byte thành chuỗi văn bản thông qua phương thức **decode()**, sau đó loại bỏ các khoảng trắng ở đầu và cuối chuỗi bằng phương thức **strip()**. Kết quả là chuỗi văn bản đã được chuẩn hóa.Giải mã chuỗi byte thành chuỗi thông thường, loại bỏ khoảng trắng dẫn/dẫn cuối.
* **b = a.split("\r\n")**: Tách chuỗi thành một danh sách các dòng, sử dụng ký tự xuống dòng (**"\r\n"**) làm điểm tách.Tách mỗi dòng thành một danh sách các từ.
* **b = [" ".join(x.split()) for x in b]**: Cho mỗi dòng trong danh sách, loại bỏ các khoảng trắng thừa ở giữa bằng cách sử dụng **split()** để tách từng từ trong dòng, sau đó **join()** để kết hợp lại thành một chuỗi mới. Kết quả là danh sách các dòng đã được chuẩn hóa.
* **c = [x.split() for x in b][2:]**: Cho mỗi dòng trong danh sách sau bước trước, tách chuỗi thành các phần tử và lưu thành danh sách con. Bỏ qua hai phần tử đầu tiên của danh sách (hàng tiêu đề không cần thiết).
* **return c**: Trả về danh sách cuối cùng, đại diện cho thông tin chi tiết về các tiến trình, được biểu diễn dưới dạng danh sách các danh sách.
* *viewList*:



* **subprocess.check\_output**: Là một hàm trong thư viện **subprocess** của Python, được sử dụng để thực thi một lệnh hệ thống và trả về kết quả của lệnh đó dưới dạng chuỗi byte.
* "**powershell gps | where {$\_.MainWindowTitle} | select Name,Id,@{Name='ThreadCount';Expression={$\_.Threads.Count}}**"
  + **powershell gps**: Chạy cmdlet gps trong PowerShell để lấy thông tin về các tiến trình (Get-Process).
  + **where {$\_.MainWindowTitle}**: Lọc các tiến trình dựa trên có cửa sổ chính hay không.
  + **select Name,Id,@{Name='ThreadCount';Expression={$\_.Threads.Count}}**: Chọn các thuộc tính Name, Id và tạo một thuộc tính mới là ThreadCount, thể hiện số lượng luồng cho mỗi tiến trình.
* Kết quả của lệnh PowerShell được trả về dưới dạng chuỗi byte và được lưu vào biến **app**
* self.appList = self.process2List(app): Kết quả được truyền vào hàm process2List để chuyển đổi từ chuỗi byte sang danh sách các tiến trình.
* **table\_headers = ["Process", "PID", "Thread Count"]**: Định nghĩa tiêu đề cho bảng dữ liệu.
* **formatted\_table = tabulate(self.appList, headers=table\_headers, tablefmt="pretty")**: Sử dụng thư viện **tabulate** để định dạng danh sách các tiến trình thành một bảng dễ đọc. Kết quả được trả về dưới dạng một chuỗi chứa bảng được định dạng.
* **return formatted\_table**: Trả về bảng dữ liệu đã được định dạng.
* *startApp*:



* **PathProcess = os.path.relpath(application\_name + ".exe")**: Xây dựng đường dẫn tương đối đến tập tin thực thi của ứng dụng. Chuỗi **".exe"** được thêm vào tên ứng dụng để đảm bảo rằng extension của tập tin thực thi đã được xác định.
* **os.startfile(PathProcess)**: Sử dụng hàm **startfile** từ module **os** để mở tập tin hoặc ứng dụng với chương trình mặc định được liên kết với loại tập tin đó trên hệ điều hành. Trong trường hợp này, nó sẽ mở ứng dụng có đường dẫn là **PathProcess**.
* **return f"Started application: {application\_name}"**: Trả về một chuỗi thông báo thành công khi ứng dụng được khởi động. Chuỗi này chứa tên ứng dụng để người dùng biết ứng dụng nào đã được bắt đầu.
* **except Exception as e: return f"Error starting application: {e}"**: Nếu có bất kỳ lỗi nào xảy ra trong quá trình khởi động ứng dụng, chương trình sẽ rơi vào khối except. Nó sẽ trả về một chuỗi thông báo lỗi chứa thông tin chi tiết về lỗi đó, giúp người dùng xác định vấn đề
* *endApp*:

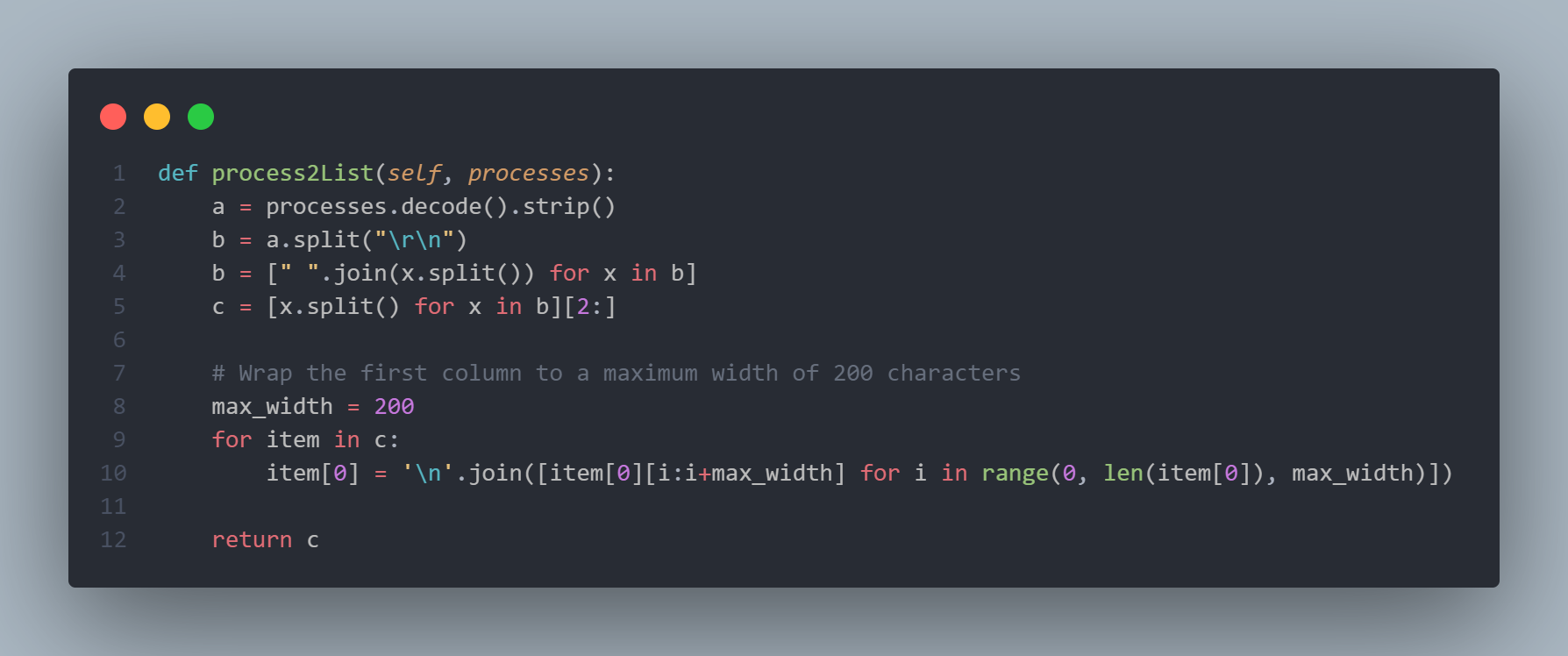


* **check = False**: Tạo một biến **check** và khởi tạo giá trị là **False**. Biến này được sử dụng để kiểm tra xem có tìm thấy và kết thúc ứng dụng hay không.
* **for process in psutil.process\_iter(attrs=['pid', 'name']):**: Sử dụng vòng lặp để duyệt qua tất cả các tiến trình đang chạy trên hệ thống sử dụng **psutil.process\_iter**. Thông tin về tiến trình được thu thập bao gồm **pid** (Process ID) và **name** (tên tiến trình).
* Trong **try**, **process\_info = process.info** lấy thông tin về tiến trình.
* **if process\_info['name'] == application\_name + ".exe":**: Kiểm tra xem tên của tiến trình có khớp với tên ứng dụng được đưa vào hay không.
* **pid = process\_info['pid']**: Lấy Process ID (PID) của tiến trình.
* **process = psutil.Process(pid)**: Tạo một đối tượng **Process** từ PID để thao tác với tiến trình.
* **process.terminate()**: Gọi phương thức **terminate()** để kết thúc tiến trình.
* **check = True**: Đặt giá trị của biến **check** thành **True** để cho biết tiến trình đã được kết thúc thành công.
* **except (psutil.NoSuchProcess, psutil.AccessDenied, psutil.ZombieProcess):**: Bắt các ngoại lệ có thể xảy ra trong quá trình lấy thông tin về tiến trình, như khi không tìm thấy tiến trình (**NoSuchProcess**), truy cập bị từ chối (**AccessDenied**), hoặc tiến trình là Zombie (**ZombieProcess**).
* **if(check == True): return f"End Application {application\_name} Successfully!"**: Kiểm tra giá trị của **check**. Nếu **check** là **True**, thì trả về một chuỗi thông báo cho biết ứng dụng đã được kết thúc thành công.
* **else: return f"No App {application\_name} Found."**: Nếu **check** là **False**, thì trả về một chuỗi thông báo cho biết không tìm thấy ứng dụng với tên đã đưa vào.

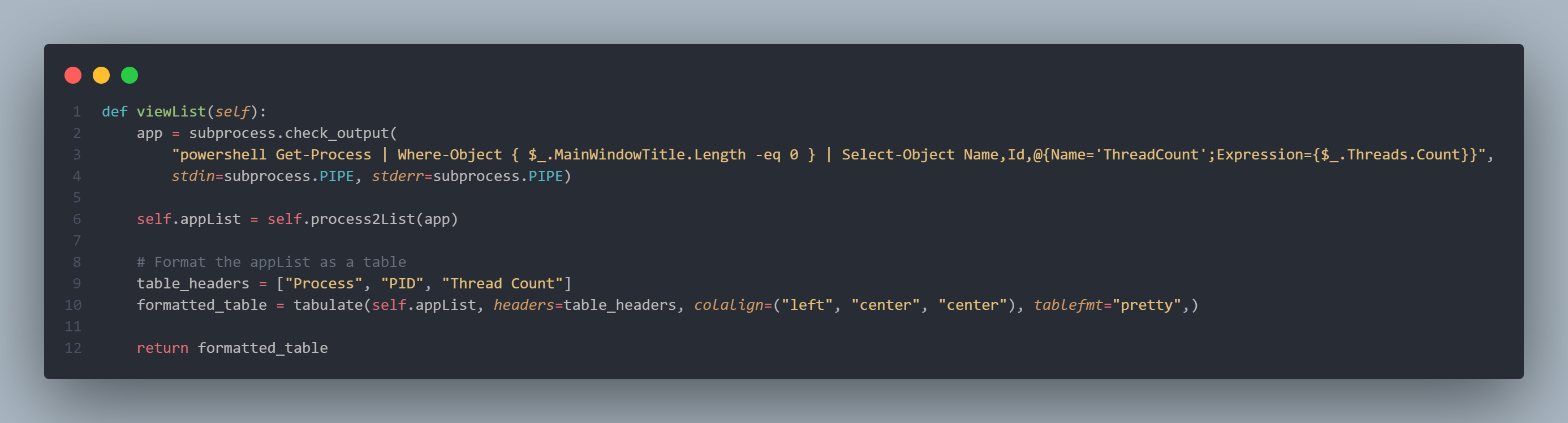
## processController.py

Class **ProcessController()**: chứa các hàm dùng để tương tác với process, bao gồm:

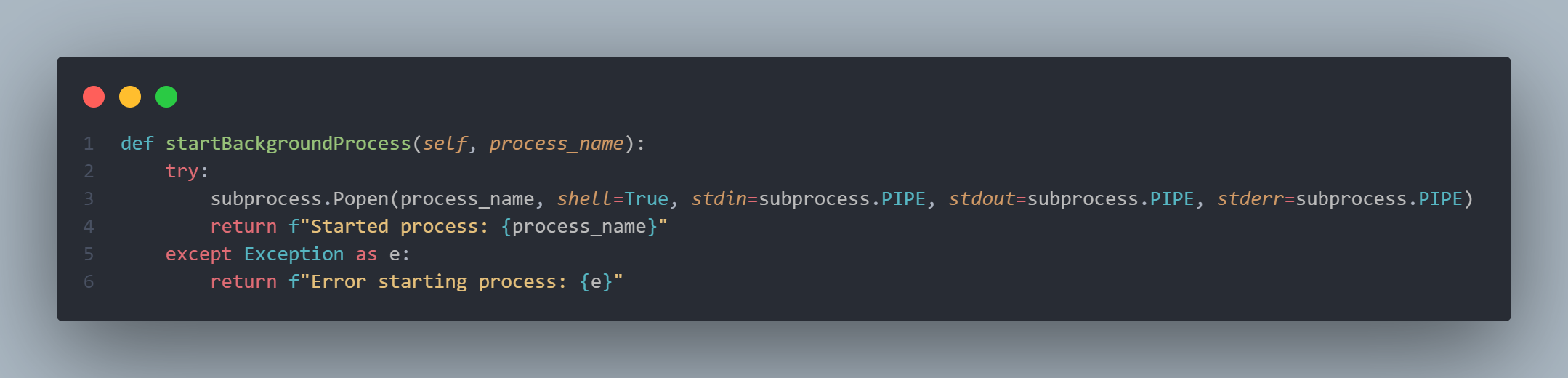
* *process2List*:



* Giống như trong **AppController**, hàm **process2List** của **ProcessController** nhận đầu vào là kết quả của lệnh PowerShell được gọi bằng **subprocess.check\_output**.
* Kết quả này được chuyển thành một danh sách các process với thông tin như tên process, ID và số lượng luồng.
* Do tên của process có thể rất dài, đối với mỗi tiến trình, cột đầu tiên (tên process) được chia thành các dòng có chiều dài tối đa là 200 ký tự.
* *viewList*:



* **viewList** sử dụng **subprocess.check\_output** để chạy lệnh PowerShell, lấy danh sách các tiến trình không có cửa sổ chính (**Where-Object { $\_.MainWindowTitle.Length -eq 0 }**).
* Sau đó, gọi **process2List** để chuyển đổi kết quả thành danh sách và sử dụng **tabulate** để định dạng danh sách này thành một bảng.
* *startBackgroundProcess*:



* Sử dụng **subprocess.Popen** để bắt đầu một process mới dựa trên tên process được truyền vào:
  + **shell=True**: Đặt giá trị này thành True cho phép sử dụng PowerShell để thực hiện câu lệnh.
  + **stdin=subprocess.PIPE**: Thiết lập luồng đầu vào (stdin) của tiến trình. Trong trường hợp này, **subprocess.PIPE** được sử dụng để tạo một ống dẫn đầu vào từ quy trình Python chính đến tiến trình con.
  + **stdout=subprocess.PIPE**: Thiết lập luồng đầu ra (stdout) của tiến trình. Tương tự như stdin, **subprocess.PIPE** tạo một ống dẫn đầu ra từ tiến trình con đến quy trình Python chính.
  + **stderr=subprocess.PIPE**: Thiết lập luồng lỗi (stderr) của tiến trình. Cũng giống như stdin và stdout, subprocess.PIPE tạo một ống dẫn đến quy trình Python chính.
* Nếu có lỗi, hàm sẽ trả về một thông báo lỗi.
* *endProcess*:



* Sử dụng **subprocess.run** để chạy lệnh **taskkill** để kết thúc một process dựa trên tên được truyền vào
  + **f"taskkill /f /im {process\_name}"**: Chuỗi lệnh được tạo ra bằng cách sử dụng f-string để chèn tên tiến trình vào lệnh **taskkill**. Đối số **/f** yêu cầu kết thúc tiến trình mạnh mẽ (force), và **/im** chỉ định tiến trình bằng tên của nó.
  + **shell=True**: Cho phép sử dụng PowerShell để thực hiện câu lệnh.
  + **check=True**: Nếu đặt thành True, nếu lệnh thực thi trả về mã lỗi khác 0, một **subprocess.CalledProcessError** sẽ được ném. Điều này có nghĩa là nếu lệnh **taskkill** thất bại, nó sẽ ném một ngoại lệ.
  + **text=True**: Đảm bảo rằng đầu ra và lỗi của quá trình con được trả về dưới dạng chuỗi thay vì byte.
  + **capture\_output=True**: Bật chế độ bắt đầu đầu ra từ quá trình con. Điều này làm cho đầu ra của lệnh **taskkill** có sẵn trong thuộc tính stdout của đối tượng **CompletedProcess**.
  + **return result.stdout.strip()**: Trả về đầu ra của lệnh taskkill dưới dạng chuỗi đã được loại bỏ khoảng trắng ở đầu và cuối chuỗi. Điều này có thể là một chuỗi thông báo xác nhận từ **taskkill** hoặc thông báo lỗi nếu có.
* Nếu có lỗi, hàm sẽ trả về một thông báo lỗi.

# Hướng dẫn sử dụng

# Đóng góp

# Tài liệu tham khảo