



파이썬 코드로 web site에 로그인 해서 사이트를 캡처하는 프로그램이 가능할까?

네, 파이썬 코드로 웹사이트에 로그인 후 특정 페이지를 캡처하는 프로그램을 충분히 만들 수 있습니다. 일반적으로 다음과 같은 라이브러리를 조합하여 사용합니다:

✓ 사용 기술 스택 개요

목적	라이브러리
브라우저 자동 제어	selenium, playwright, 또는 requests+BeautifulSoup
화면 캡처	selenium/playwright의 screenshot() 기능
로그인 자동화	ID/PW 입력 후 버튼 클릭 (또는 쿠키 이용)

🔑 예시: Selenium + ChromeDriver로 로그인 후 캡처

```
python

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
import time

# 1. 크롬 브라우저 열기
options = webdriver.ChromeOptions()
options.add_argument("--headless") # GUI 없이 실행
driver = webdriver.Chrome(options=options)

# 2. 웹사이트 접속
driver.get("https://example.com/login")

# 3. 로그인 정보 입력
driver.find_element(By.ID, "username").send_keys("your_id")
driver.find_element(By.ID, "password").send_keys("your_password")
driver.find_element(By.ID, "password").send_keys(Keys.RETURN)

# 4. 로그인 완료까지 대기 (페이지 전환 대기)
time.sleep(3)

# 5. 캡처할 페이지로 이동
driver.get("https://example.com/dashboard")
time.sleep(2)

# 6. 화면 캡처
driver.save_screenshot("dashboard.png")

# 7. 종료
driver.quit()
```

🔑 주의할 점

1. 사이트에 따라 로그인 방식이 다름

- ID/PW 방식 (폼 로그인)
- 소셜 로그인 (Google, Kakao 등은 자동화 어려움)
- CAPTCHA가 있으면 자동화 불가능 또는 회피 필요

2. 보안정책에 따라 자동 접속 차단

- User-Agent 변경, 쿠키 활용, 로그인 세션 유지 등 우회 필요

3. 정적 웹사이트 vs 동적 웹사이트

- 단순한 **requests**로 가능한 경우도 있음 (예: API 기반 사이트)
- JavaScript로 렌더링하는 사이트는 **selenium/playwright** 필요

playwright를 쓸 수도 있습니다

playwright는 속도와 안정성 면에서 **selenium**보다 뛰어난 경우가 많습니다. 예시 필요하시면 제공해드릴게요.

마무리

수행자님이 캡처하려는 사이트가 어떤 로그인 구조인지, 보안 요소가 있는지(CAPTCHA, MFA 등) 알려주시면 더욱 정확한 예제를 드릴 수 있습니다.

또한 GUI 없이 서버에서 자동 캡처가 목적이라면 **headless** 모드로 구성하는 게 유리합니다.

필요하시면 **playwright**, **requests** 기반 코드도 함께 드릴게요. 원하시는 방향 있으신가요?



airflow scheduler 는 30초에 한번씩 dag를 검사(파싱)해?

네, 정확히 말씀드리면 **Airflow Scheduler**는 기본적으로 30초마다 DAG 디렉토리를 스캔하여 DAG 파일을 파싱합니다. 이 주기를 조절할 수 있는 설정도 함께 알려드릴게요.

기본 동작: DAG 파싱 주기

- 기본값: 30초
- Airflow는 기본적으로 **dags_folder** 경로에 있는 DAG 파일들을 30초마다 다시 **스캔(scan)** 하고 **파싱(parse)** 하여 새로운 DAG가 등록되었는지, 수정되었는지를 확인합니다.
- 이 주기를 조절하는 설정이 바로:

```
ini
```

```
[scheduler]
dag_dir_list_interval = 30 ← (기본값: 30초)
```

🧠 관련 개념 정리

항목	설명
<code>dag_dir_list_interval</code>	DAG 디렉토리 스캔 주기 (default: 30초)
<code>min_file_process_interval</code>	동일 DAG 파일을 다시 파싱하기 위한 최소 대기 시간
<code>scheduler_heartbeat_sec</code>	스케줄러 heartbeat 주기 (UI에서 "alive" 표시용)
<code>parsing_processes</code>	병렬 DAG 파싱 프로세스 개수 (성능 향상에 사용됨)

🔧 설정 위치

- 이 값들은 `airflow.cfg` 또는 Helm 환경에서는 `values.yaml` 안의 `scheduler` 블록에 설정되어 있습니다:

```
yaml

scheduler:
  dagDirListInterval: 30
  minFileProcessInterval: 30
  schedulerHeartbeatSec: 5
```

📌 성능 튜닝 팁

- DAG 파일이 많고 복잡할수록 파싱 시간이 길어집니다.
- 다음을 권장합니다:
 - DAG 파싱 로직에 복잡한 함수 호출, DB 연결, API 요청 금지
 - `if __name__ == "__main__":` 또는 DAG 선언 외 코드 최소화
 - 커스텀 Python 코드 모듈은 DAG 내부가 아닌 외부 함수로 분리

🕒 Airflow 2.0 이후의 구조 변화

Airflow 2.x 이상에서는 스케줄러가 멀티프로세싱 기반으로 동작하며, 여러 DAG를 동시에 파싱하고 스케줄링할 수 있습니다. (`scheduler.use_job_schedule = True`)

👤 마무리

따라서 질문 요약:

"Airflow 스케줄러는 30초마다 DAG를 파싱하나요?"

👉 네, 기본값은 30초마다 DAG 디렉토리를 스캔하고 변경된 DAG를 감지하여 파싱합니다. 이 주기는 `dag_dir_list_interval`로 조절할 수 있습니다.

필요하시면 Helm 설치 환경에서의 설정 방법이나 파싱 최적화 방법도 안내드릴게요. 도와드릴까요?



그러면, dag를 검사할때, dag 상단에 subprocess.check_call 같은 방식으로 모듈을 설치하는 코드가 있으면 30초 한번씩 수행하면서 설치하는거야?

네, 맞습니다. 매우 위험한 상황입니다.

Airflow의 Scheduler는 **DAG 파일을 30초마다 스캔하고 파싱**하면서, DAG 파일 안에 있는 코드를 **실제로 실행**합니다. 따라서 DAG 상단에 `subprocess.check_call(...)` 같은 설치 코드가 있으면, **30초마다 해당 명령이 실행되는 매우 심각한 부작용**이 발생할 수 있습니다.

🔥 문제 예시

```
python

import subprocess

# ✅ 이 부분이 문제!
subprocess.check_call(["pip", "install", "some_package"])

with DAG(...) as dag:
    ...
```

이렇게 작성하면 Airflow는:

1. `days_folder`를 30초마다 스캔
2. 이 DAG 파일을 다시 파싱
3. DAG 파일을 import 하면서 위의 `pip install` 코드 실행 🌞
4. 계속 반복...

🚫 결과적으로 생기는 문제

문제 항목	설명
🦎 파싱 속도 저하	매번 설치 작업으로 DAG 파싱이 느려지고 scheduler가 병목됨
🔄 무한 설치 반복	pip install이 계속 반복되어 CPU/디스크 낭비
💣 스케줄러 오작동	scheduler timeout, zombie process, pod OOM 등 발생 가능
☁️ 클러스터 자원 낭비	파드 재시작, Image pull, 로그 오염 등

✅ 해결 방법 (Best Practice)

🔪 1. DAG 파일은 정의만 해야 합니다

- DAG은 단순히 **Task 정의**만 있어야 하며, 그 외 **설치, 실행, I/O, 연결** 등은 절대 금지입니다.
- 아래처럼 DAG 안에서 설치하게 하세요:

```
python

from airflow.operators.bash import BashOperator

install = BashOperator(
    task_id="install_some_package",
    bash_command="pip install some_package",
)
```

또는 PythonOperator로:

```
python

from airflow.operators.python import PythonOperator

def install_package():
    import subprocess
    subprocess.check_call(["pip", "install", "some_package"])

task = PythonOperator(
    task_id="install_package",
    python_callable=install_package,
)
```

🔥 2. 또는 Docker Image에 미리 포함

가장 권장되는 방식은:

- `some_package`를 미리 포함한 Docker Image를 만들고
- Airflow의 `pod_template`, `KubernetesExecutor`, `SparkSubmitOperator` 등에서 사용

🔥 3. 조건문으로 임시 차단도 가능

긴급하게 우회할 경우 다음과 같이 막을 수 있습니다:

```
python

if __name__ == "__main__":
    # 로컬 테스트 시에만 동작
    subprocess.check_call(["pip", "install", "some_package"])
```

👤 정리

질문	답변
DAG 상단에 <code>pip install</code> 코드가 있으면?	✅ 30초마다 실행됩니다. 매우 위험합니다.
해결 방법은?	✅ 설치하는 Operator로 분리하거나, Image에 미리 포함시켜야 합니다.
왜 안 좋은가요?	🔥 Scheduler 파싱 단계에서 코드가 실행되면, 반복적인 설치와 시스템 오작동을 초래할 수 있습니다.

필요하시면 해당 DAG 구조를 함께 리팩토링해 드릴 수 있어요. DAG 코드를 공유해주시면, 설치 코드를 Task로 분리하거나 다른 방법으로 개선해드릴게요.



만약에 매번 pip 로 모듈이 설치되면서 파일 경합이 일어날수도 있네, 모듈을 사용하는 코드가 수행될때 scheduler는 모듈을 설치한다면....

네, 수행자님 말씀 정확히 맞습니다.

Scheduler가 DAG 파싱 중 pip install을 실행하고,
같은 시점에 Worker(Pod)가 그 모듈을 import해서 사용하는 경우,
 실제로 파일 경합(file race condition) 또는 모듈 손상/ImportError가 발생할 수 있습니다.

! 이런 상황이 실제로 위험한 이유

▶ 예: 이런 상황이 가능함

1. scheduler가 DAG 파일을 파싱하면서 `pip install xxx`를 실행 중
2. 동시에 worker는 DAG 실행 중에 `import xxx`를 시도
3. 이 순간, 해당 모듈의 설치가 끝나지 않았거나 `.pyc` 등 캐시 파일이 깨져 있는 상태일 수 있음
4. 결과적으로:
 - `ModuleNotFoundError`
 - `ImportError: partially initialized module`
 - `pip install` 중인 파일이 lock되어 있어서 설치 실패
 - Airflow task가 실패하거나 scheduler가 crash

☀ 실제 증상 예시

text

```
ImportError: cannot import name 'XYZ' from partially initialized module 'abc' (most likely due to a circular import)
```

text

```
pip._internal.exceptions.InstallationError: ERROR: Could not install packages due to an OSError: [Errno 13] Permission denied: '...'
```

text

```
airflow.exceptions.AirflowException: Scheduler heartbeat failure
```

🔥 왜 이런 문제가 생길까?

- `pip install`은 실제로 다음 작업을 합니다:
 - `.whl` 다운로드 → 압축 해제 → `site-packages` 복사 → `.pyc` 생성 → 캐시 파일 생성
- 이 모든 과정이 완료되기 전에 다른 프로세스가 해당 모듈을 로드하려 하면 **경합(race condition)** 발생

✅ 안전한 모듈 배포 방식

방법	설명
✅ Docker Image 에 사전 포함	가장 안전하고 안정적인 방식. 모듈은 빌드 시 포함되어야 함.
✅ PythonOperator 내부에서만 <code>pip install</code>	Scheduler가 아니라 Task 내부에서 설치. 단, 이것도 권장되지 않음.
❌ DAG 파일 상단에서 <code>pip install</code>	절대 금지! 위 문제의 원인.
✅ requirements.txt + init container	Airflow Pod 시작 전에 필요한 모듈을 설치하도록 구성 가능

🧠 Kubernetes 환경이라면?

- **KubernetesExecutor**나 **KubernetesPodOperator** 환경에서는:
 - 파드마다 독립된 환경이므로 `pip install`이 내부에서는 안전할 수 있음
 - 하지만 여러 파드가 공유 volume (예: `hostPath`, `NFS`, `PV` 등)을 사용하는 경우 여전히 경합 가능성 존재
 - `site-packages`가 shared되면 위험도 증가

✅ 정리

항목	설명
Scheduler가 <code>pip install</code> 할 수 있나요?	❗ DAG 파일에 코드가 있으면 매번 실행됩니다.
동시에 Worker가 모듈을 쓰면?	❗ 설치 중 모듈을 import 하면서 파일 경합 발생 가능
결과는?	❌ <code>ImportError</code> , Module 손상, Scheduler 장애 등
해결책은?	✅ 설치 코드는 DAG 밖으로 빼고, 이미 설치된 환경에서 작업 수행해야 안전합니다.

필요하시면 다음도 도와드릴 수 있어요:

- `pip install`을 제거한 안전한 DAG 코드로 리팩토링
- **Dockerfile**로 커스텀 Airflow 이미지 구성 예시
- **KubernetesPodOperator**에서 `pip` 설치 시 캐싱 전략

어떤 방식으로 사용 중인지 알려주시면, 상황에 맞춰 드리겠습니다.



각각의 업무에서 사용하는 dag들을 주기적으로 (일주일에 한번, 한달에 한번)정도 취합해서 pip 로 개별적으로 설치하던 것을 base 이미지에 적용시켜 주는 전략에 대해 어떻게 생각해?

수행자님 질문에 대한 답변을 한 마디로 요약하자면:

✅ "주기적으로 사용하는 모듈을 수집하여 base 이미지에 포함시키는 전략은 아주 훌륭하고 실무적으로 권장되는 Best Practice입니다."

Airflow DAG 운영환경에서 모듈 설치를 다음과 같이 바꾸는 건 **안정성, 성능, 관리 효율성** 측면 모두에서 매우 효과적인 전략입니다.

🎯 전략 설명

🔄 기존 방식: 매번 pip install

- DAG 내부 또는 Task 시작 시 **pip install**로 필요한 모듈 설치
- 매번 재설치 → 시간 낭비, 경합 발생, 불안정성 증가
- 모듈 버전 일관성 관리 어려움

💡 개선안: base 이미지에 포함

- 모든 DAG에서 공통적으로 사용하는 모듈을 수집 (주 1회, 월 1회)
- 이를 **Dockerfile**에 미리 반영하여 base 이미지를 빌드
- 모든 DAG/Task는 동일한 base image 사용

✅ 장점 요약

항목	설명
⚡ 성능 향상	Task 실행 시 pip 설치 없이 바로 시작 가능 (Pod startup time ↓)
🎯 안정성 향상	모듈 설치 실패, 버전 불일치 등 문제 사전 제거
🧩 버전 관리 일원화	특정 모듈 버전을 이미지에서 고정 가능 → QA 및 재현성↑
☁ 리소스 절약	파드마다 중복 설치 방지, 네트워크 및 디스크 사용량↓
🔒 보안 강화	공식 이미지 빌드 시 모듈 검증 및 취약점 검사 가능
🔧 유지보수 편의	Airflow 환경의 일관성 유지, 운영 중 장애 대응 용이

🔧 전략 구현 예시

1. 🔍 모듈 수집 자동화 (예: 주간 스크립트)

bash

```
# 매주 사용하는 DAG에서 모듈 목록 추출
grep -hr "import " /opt/airflow/dags/ | sort | uniq > used_modules.txt
# 또는 requirements_dag1.txt + requirements_dag2.txt 등 수동 통합

# pip freeze와 diff하여 새로운 모듈만 추출
pip freeze > current_env.txt
comm -13 <(sort current_env.txt) <(sort used_modules.txt) > to_be_added.txt
```

2. 🐳 base 이미지 빌드

dockerfile

```
FROM apache/airflow:2.9.2-python3.11

# 주기적으로 수집한 공통 모듈 설치
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
```

3. 🔄 주기적 빌드 자동화

- GitLab CI, Jenkins, Argo Workflows 등으로 주기적 base 이미지 빌드
- 버전 태깅: **my-airflow:2.9.2-py311.2025.09**
- ArgoCD로 rollout

📌 추가 팁

항목	내용
DAG별 특수 모듈	base 이미지에 안 넣고 KubernetesPodOperator 의 이미지에서만 추가 가능
옵션 1	여러 base 이미지: airflow-base-data , airflow-base-ml , airflow-base-finance
옵션 2	pip install -t /opt/airflow/.venv ... 등 isolated 경로도 가능 (shared PVC 사용 시)

⚠️ 주의사항

항목	설명
🎯 모듈 호환성 테스트 필수	빌드 전에 QA 환경에서 DAG 정상 작동 여부 확인
🐳 base 이미지 사이즈 주의	너무 많은 모듈 설치 → 이미지 용량 증가 가능
🔒 보안 점검 필수	PyPI 패키지에 취약점이 있을 수 있으므로 이미지 스캔 (Trivy 등) 병행 권장

👤 마무리

✅ **Airflow** 환경에서 주기적으로 사용하는 모듈을 base 이미지에 통합하는 전략은 대규모 운영환경에서 매우 안정적이고 추천되는 방식입니다.

필요하시면:

- 🛠 커스텀 Airflow Dockerfile 예제
- 🐛 CI 기반 자동 이미지 빌드 스크립트
- 📊 DAG별 모듈 사용 내역 자동 스캐닝 도구

등도 함께 도와드릴 수 있습니다.

이 전략을 실현할 때 어떤 환경(예: Helm chart, K8s executor, 내부 registry 등)에서 사용하시는지 알려주시면 더 구체적으로 도와드릴 수 있어요.



너가 말한 아래 방식에 대해서 상세히 어떻게 적용하는지 알려줘. 도커 빌드할때 이전 모듈 현황 및 빌드하고 나서 새로운 모듈현황과 새롭게 변경된 내역등등을 자동화 하는 방한에 대해서

```
# 매주 사용하는 DAG에서 모듈 목록 추출
grep -hr "import " /opt/airflow/dags/ | sort | uniq > used_modules.txt
# 또는 requirements_dag1.txt + requirements_dag2.txt 등 수동 통합

# pip freeze와 diff하여 새로운 모듈만 추출
pip freeze > current_env.txt
comm -13 <(sort current_env.txt) <(sort used_modules.txt) > to_be_added.txt
```

수행자님, 아래는 Airflow DAG 환경에서 주기적으로 사용되는 모듈을 자동 수집하고, 이전/신규 모듈 현황을 비교하며 Docker base 이미지를 갱신하는 전체 자동화 흐름입니다.

✓ 목표

- DAG 코드에서 사용된 **import** 문 기반으로 **used_modules.txt** 생성
- 현재 이미지의 **pip freeze** 결과 (**current_env.txt**) 와 비교
- 신규 모듈 추출 → **to_be_added.txt**로 정리
- 신규 모듈만 **requirements.txt**로 사용하여 Docker 이미지 빌드
- 모든 결과를 자동 리포팅

🕒 전체 흐름 요약 (자동화 파이프라인 설계)

text

- | | |
|-----------|-----------------------|
| 📁 dags/ | ← DAG 코드 저장소 |
| 📁 docker/ | ← Dockerfile, 빌드 스크립트 |
| 📁 logs/ | ← 변경 리포트 저장소 |

빌드 파이프라인 단계:

1. used_modules.txt 생성
2. current_env.txt (이전 이미지 기준) 준비
3. 신규 모듈만 to_be_added.txt 생성
4. requirements.txt로 빌드
5. 리포트 자동 생성
6. Docker 이미지 빌드 & 푸시

🔧 1단계: DAG 코드에서 모듈 추출

```
bash

# extract_imports.sh
#!/bin/bash
set -e

echo "[1] DAG import 문 스캔 중..."
grep -horP '^import\s+\w+|^from\s+\w+' ./dags \
| awk '{print $2}' \
| sort -u > logs/used_modules.txt
```

- `awk '{print $2}': import abc, from xyz import ... 형식 중 모듈명만 추출`

🔧 2단계: 현재 환경 모듈 저장

```
bash

# extract_current_env.sh
#!/bin/bash
set -e

echo "[2] 현재 pip freeze 결과 수집 중..."
docker run --rm your-airflow:latest pip freeze | sort > logs/current_env.txt
```

- 이 결과는 현재 사용 중인 이미지 기준입니다 (`your-airflow:latest`는 이전 빌드된 이미지)

🔧 3단계: 신규 모듈 추출 (diff)

```
bash

# diff_modules.sh
#!/bin/bash
set -e

echo "[3] DAG import와 현재 환경 비교 중..."
comm -23 logs/used_modules.txt logs/current_env.txt > logs/to_be_added.txt
```

- `comm -23 A B`: A에는 있고 B에는 없는 줄만 출력 (신규 모듈)

🔧 4단계: requirements.txt 생성 (or merge)

```
bash

# build_requirements.sh
#!/bin/bash
set -e

echo "[4] requirements.txt 생성 중..."
cat logs/to_be_added.txt > docker/requirements.txt

# 추가: 기본 공통 모듈 미리 포함 시
cat docker/base_requirements.txt >> docker/requirements.txt
```

5단계: Dockerfile 예시

```
dockerfile

# docker/Dockerfile

FROM apache/airflow:2.9.2-python3.11

COPY requirements.txt /tmp/requirements.txt
RUN pip install --no-cache-dir -r /tmp/requirements.txt
```

6단계: Docker 이미지 빌드 & 푸시

```
bash

# build_and_push.sh
#!/bin/bash
set -e

TAG=$(date +%Y%m%d-%H%M)
echo "[5] Docker 이미지 빌드 중: airflow-base:${TAG}"

docker build -t your-registry/airflow-base:${TAG} -f docker/Dockerfile docker/
docker push your-registry/airflow-base:${TAG}
```

7단계: 변경 리포트 자동 생성

```
bash


# report_changes.sh
#!/bin/bash
set -e

echo "[6] 변경 리포트 생성 중..."

echo "NEW 추가된 모듈 목록:" > logs/report_$(date +%Y%m%d).md
cat logs/to_be_added.txt >> logs/report_$(date +%Y%m%d).md
```

예시 출력:

md

 추가된 모듈 목록 (2025-09-30)

- boto3
- pandas
- google-cloud-storage





전체 실행 스크립트 예시

bash

```
bash extract_imports.sh
bash extract_current_env.sh
bash diff_modules.sh
bash build_requirements.sh
bash build_and_push.sh
bash report_changes.sh
```

CI 도구(Jenkins, GitLab CI, ArgoCD 등)로 주간 자동 실행 가능

추가 아이디어

기능	설명
 Slack/Teams 리포트 전송	새로 추가된 모듈 변경 내역 자동 알림
 PR 자동 생성	새로운 requirements.txt로 PR 오픈
 모듈 검증	pip check, trivy 로 보안 점검 포함
 DAG별 모듈 사용 추적	DAG별 import 매핑하여 모듈 정리 가능

마무리

수행자님의 “운영 안정성 + 자동화된 모듈 통합 전략”은 매우 이상적입니다.

위 스크립트들을 기반으로 매주 자동 베이스 이미지 관리 + 변경 리포트 추적 + CI/CD 연동이 가능합니다.

필요하시면 위 스크립트를 **.sh** 파일로 정리해서 드리거나, GitLab CI 파이프라인 또는 ArgoCD 배포와 통합하는 흐름도 함께 설계해 드릴게요.

어떤 CI/CD 환경에서 운영 중이신가요? (예: GitLab, Jenkins, ArgoCD 등)