↑ 목록으로 ← 이전 단원

6단원. 인프라 자동화를 위한 파이썬

> 대상: 전공자 · 정보처리기사 취득자 | 목적: 인프라 직무(네트워크·클라우드·시스템·보안) 기술 면접 대비 중요도: ★★★ 최빈출 · ★★ 자주 출제 · ★ 기본 개념

> **Q1. subprocess** 모듈로 외부 명령어를 실행하는 방법을 설명하세요. ★★★

답안 **`subprocess`** 모듈은 파이썬에서 셸 명령어나 외부 프로그램을 실행하고 그 결과를 제어하기 위한 표준 방법으로, 과거에 쓰이던 `os.system`이나 `os.popen`보다 더 세밀하고 안전한 제어를 제공해 현재는 이 모듈 사용이 권장됩니다. **`subprocess.run()`**은 명령을 실행하고 완료될 때까지 기다린 뒤 종료 코드, 표준 출력, 표준에러를 담은 결과 객체를 반환하는 가장 널리 쓰이는 고수준 API로, `capture_output=True`로 출력을 캡처하고 `check=True`로 지정하면 명령이 실패(0이 아닌 종료 코드)했을 때 `CalledProcessError` 예외를 자동으로 발생시켜 에러 처리를 강제할 수 있습니다. 명령어는 `["ls", "-l", "/tmp"]`처럼 리스트 형태로 전달하는 것이 권장되는데, `shell=True`와 함께 문자열을 그대로 전달하면 사용자 입력이 섞였을 때 셸 인젝션(**shell injection**) 취약점으로 이어질 수 있기 때문입니다. 더 정교한 제어(실시간 스트리밍 출력, 파이프 연결 등)가 필요하면 저수준 API인 `subprocess.Popen`을 직접 사용할 수 있습니다.

관련 개념 subprocess.run vs Popen, shell=True 인젝션 위험, check=True, CalledProcessError

---

Q2. os 모듈을 이용한 파일 시스템 조작 방법을 설명하세요. ★★

답안 **`os`** 모듈은 운영체제와 상호작용하는 다양한 기능을 제공하며, 파일 시스템 관점에서는 `os.listdir()` 로 디렉토리 내 목록을 조회하고, `os.mkdir()` / `os.makedirs()` 로 디렉토리를 생성(후자는 중첩 경로를 한 번에 생성 가능)하며, `os.remove()` / `os.rmdir()` 로 파일과 빈 디렉토리를 삭제합니다. 경로 관련 작업은 `os.path.join()` 으로 운영체제에 맞는 구분자를 사용해 경로를 안전하게 조합하고, `os.path.exists()` / `os.path.isfile()` / `os.path.isdir()` 로 존재 여부와 종류를 확인하며, `os.environ` 을 통해 환경변수를 읽거나 설정할 수 있습니다. 최신 파이썬에서는 객체지향적인 **`pathlib`** 모듈(`Path` 클래스)이 경로 조작을 더 직관적으로 표현할 수 있어 점차 많이 대체되고 있지만, `os` 모듈은 여전히 프로세스 정보(`os.getpid()`), 환경변수 등 폭넓은 시스템 인터페이스를 제공한다는 점에서 함께 알아두어야 합니다.

관련 개념 os.path.join, pathlib.Path, os.environ, os.walk 재귀 탐색

Q3. 파이썬 **logging** 모듈을 이용한 로깅 방법과 로그 레벨을 설명하세요. ★★★

답안 **`logging`** 모듈은 단순 `print` 문과 달리 로그 레벨별 필터링, 다양한 출력 대상(콘솔, 파일, 원격 서버 등), 형식(타임스탬프, 모듈명 등) 지정을 표준화된 방식으로 지원하는 파이썬 표준 로깅 프레임워크입니다. 로그 레벨은 심각도가 낮은 순서대로 **`DEBUG`** < **`INFO`** < **`WARNING`** < **`ERROR`** < **`CRITICAL`** 로 구성되며, 로거의 레벨을 특정 값으로 설정하면 그보다 낮은 레벨의 로그는 출력되지 않아 개발 중에는 `DEBUG` 로 상세히, 운영 환경에서는 `INFO` 나 `WARNING` 이상만 남기는 식으로 조절할 수 있습니다. `logging.getLogger(__name__)` 으로 모듈별 로거를 생성하고, `Handler` (예: `FileHandler` , `StreamHandler`)와 `Formatter` 를 조합해 로그를 파일에 남기면서 동시에 콘솔에도 출력하는 등 유연한 구성이 가능합니다. 운영 자동화 스크립트에서는 예외 발생 시 `logger.exception()` 을 사용하면 스택 트레이스까지 함께 기록되어 장애 원인 파악에 큰 도움이 됩니다.

관련 개념 DEBUG~CRITICAL 레벨, Handler/Formatter, logger.exception, getLogger(__name__)

> **Q4. requests** 라이브러리를 이용한 **API** 자동화 예시와 주의점을 설명하세요. ★★

답안 **`requests`**는 파이썬에서 HTTP 요청을 간결하게 다룰 수 있게 해주는 대표적인 서드파티 라이브러리로, `requests.get(url, params={...})`, `requests.post(url, json={...})`처럼 직관적인 함수로 REST API를 호출할 수 있습니다. 응답 객체의 `.status_code`로 HTTP 상태 코드를 확인하고, `.json()`으로 JSON 응답 본문을 파이썬 딕셔너리로 바로 변환할 수 있어 API 자동화에 매우 편리합니다. 실무에서 자주 놓치는 부분은 타임아웃 미설정인데, `timeout` 파라미터를 지정하지 않으면 상대 서버가 응답하지 않을 때 요청이 무한정 대기해 자동화 스크립트 전체가 멈출 수 있으므로 반드시 명시해야 합니다. 또한 API 인증 토큰이나 비밀번호 같은 민감 정보는 코드에 하드코딩하지 말고 환경변수나 별도 시크릿 관리 도구를 통해 주입하는 것이 보안상 바람직하며, 일시적 네트워크 오류에 대비해 재시도 로직(`requests`의 `Session`과 `HTTPAdapter`를 활용한 retry 전략 등)을 함께 구성하는 것이 안정적인 자동화의 핵심입니다.

관련 개념 timeout 설정, .json() 파싱, 인증 토큰 환경변수 관리, retry/backoff

> **Q5.** 파이썬으로 파일을 안전하게 읽고 쓰는 방법(**with**문, **csv/json)**을 설명하세요. ★★

답안 파일을 열 때는 `open()`을 직접 호출한 뒤 `close()`를 잊는 실수를 방지하기 위해 **`with open(...) as f:`** 구문(컨텍스트 매니저)을 사용하는 것이 표준적인 방법으로, 블록을 벗어나면 예외가 발생하더라도 파일이 자동으로 닫혀 자원 누수를 방지합니다. 텍스트 파일은 `f.read()`, `f.readlines()`, 또는 `for line in f:`로 한 줄씩 순회하며 읽을 수 있고, 인코딩 문제를 피하기 위해 `open(path, encoding="utf-8")`처럼 인코딩을 명시하는 것이 권장됩니다. 구조화된 데이터를

다룰 때는 표준 라이브러리인 **`csv`** 모듈로 `csv.reader` / `csv.writer` 또는 `csv.DictReader` / `DictWriter`를 사용해 CSV를 다루고, **`json`** 모듈로 `json.load()` / `json.dump()`를 사용해 JSON 파일을 파이썬 객체와 상호 변환할 수 있습니다. 설정 파일이나 API 응답 데이터를 자동화 스크립트에서 파싱할 때 이 두 모듈이 매우 자주 사용됩니다.

관련 개념 with문 컨텍스트 매니저, csv.DictReader, json.load/dump, 인코딩 명시

> **Q6.** 운영 자동화 스크립트를 작성할 때 지켜야 할 설계 패턴(멱등성, **dry-run)**을 설명하세요. ★★

답안 인프라 자동화 스크립트는 사람이 실수로 두 번 실행하거나 재시도 로직에 의해 여러 번 실행될 가능성이 항상 있으므로, 몇 번을 실행하든 결과가 동일하게 유지되는 멱등성(**idempotency**)을 설계 원칙으로 삼아야 합니다. 예를 들어 "리소스를 생성하라"는 스크립트라면 실행 전에 이미 존재하는지 확인한 뒤 없을 때만 생성하도록 작성하는 식입니다. 실제 변경을 가하기 전에 어떤 작업이 수행될지 미리 확인할 수 있도록 **`--dry-run`** 옵션을 제공하는 것도 실무에서 매우 중요한 패턴으로, 운영 환경에 영향을 주는 스크립트일수록 dry-run 모드로 먼저 검증한 뒤 실제 실행하는 습관이 사고를 예방합니다. 이 외에도 하드코딩을 피하고 설정값을 환경변수나 설정 파일로 분리하는 것, 각 단계의 성공/실패를 명확히 로깅하는 것, 그리고 부분 실패 시 이미 적용된 변경을 되돌릴 수 있는 롤백 경로를 고려하는 것이 견고한 운영 스크립트의 핵심 요소입니다.

관련 개념 dry-run 옵션, 롤백 설계, 설정 외부화, 사전 상태 확인

> **Q7.** 자동화 스크립트에서 재시도(**retry**)와 백오프(**backoff**) 전략을 설명하세요. ★★

답안 네트워크 호출이나 외부 API 연동처럼 일시적인 오류가 발생할 수 있는 작업에서는 실패 즉시 전체 스크립트를 중단시키기보다 일정 횟수만큼 재시도하는 것이 안

정성을 높입니다. 이때 실패할 때마다 곧바로 재시도하면 상대 서버에 부담을 가중시킬 수 있으므로, 재시도 간격을 점점 늘려가는 지수 백오프(**exponential backoff)**(예: 1초, 2초, 4초, 8초 대기)를 적용하는 것이 일반적이며, 여러 클라이언트가 동시에 같은 타이밍에 재시도하는 것을 막기 위해 대기 시간에 무작위성을 더하는 지터(**jitter**)를 함께 사용하기도 합니다. 파이썬에서는 이를 직접 구현하거나 `tenacity`, `backoff` 같은 서드파티 라이브러리, 또는 `requests`의 `HTTPAdapter`에 `Retry` 객체를 연결해 자동화할 수 있습니다. 다만 모든 실패를 무한정 재시도하면 근본 원인을 놓칠 수 있으므로, 최대 재시도 횟수를 제한하고 최종 실패 시에는 명확한 에러 로그와 알림을 남기는 것이 중요합니다.

관련 개념 exponential backoff, jitter, tenacity/backoff 라이브러리, 최대 재시도 횟수 제한

> **Q8.** 파이썬 스크립트를 **cron**이나 **systemd timer**와 연계해 정기 실행하는 방법을 설명하세요. ★

답안 파이썬 스크립트를 정기적으로 실행하는 가장 간단한 방법은 3단원/4단원에서 다룬 **cron**의 crontab 항목에 `python3 /절대경로/script.py`처럼 인터프리터 경로와 스크립트 경로를 모두 절대경로로 등록하는 것으로, cron 환경에서는 로그인 셸의 PATH나 가상환경이 활성화되어 있지 않으므로 가상환경을 사용하는 경우 `/venv/bin/python3`처럼 가상환경 내 인터프리터를 직접 지정해야 합니다. 좀 더 현대적인 방식으로는 **systemd timer**를 사용할 수 있는데, `.service` 유닛에 실행할 파이썬 스크립트를, `.timer` 유닛에 실행 주기를 정의하면 `journalctl`을 통한 통합 로그 조회, 실행 이력 관리, 의존성 설정 등 cron보다 풍부한 기능을 활용할 수 있습니다. 어느 방식을 쓰든 스크립트 내부에서 앞서 다룬 `logging` 모듈로 실행 시작/종료/에러를 남겨두어야 정기 실행 이력을 사후에 추적할 수 있습니다.

관련 개념 systemd .timer/.service, 가상환경 인터프리터 절대경로, journalctl 로그 조회

> **Q9.** 자동화 스크립트에서 민감한 설정값(비밀번호, **API** 키)을 관리하는 방법을 설명하세요. ★

답안 API 키나 데이터베이스 비밀번호 같은 민감 정보를 소스 코드에 직접 하드코딩하면 버전 관리 시스템에 그대로 노출되어 심각한 보안 사고로 이어질 수 있으므로, 이런 값들은 코드와 분리해 환경변수(`os.environ.get("API_KEY")`)나 `.env` 파일(단, `.gitignore`에 반드시 등록), 또는 클라우드 환경의 시크릿 관리 서비스(예: 파라미터 스토어, 시크릿 매니저류)를 통해 실행 시점에 주입하는 것이 표준적인 방법입니다. 파이썬에서는 `python-dotenv` 같은 라이브러리로 `.env` 파일의 값을 환경변수처럼 로드해 쓸 수 있어 로컬 개발 편의성과 보안을 동시에 챙길 수 있습니다. 설정값이 환경(개발/스테이징/운영)마다 달라지는 경우에는 환경별로 별도의 설정 파일이나 네임스페이스를 두어 실수로 운영 환경 값이 다른 환경에 섞이지 않도록 관리하는 것도 중요합니다. 로깅 시에도 민감 정보가 로그에 그대로 찍히지 않도록 마스킹 처리하는 습관이 필요합니다.

관련 개념 .env / python-dotenv, .gitignore, 시크릿 매니저, 로그 마스킹

> **Q10.** 자동화 작업에서 **threading**과 **multiprocessing**의 차이와 선택 기준을 설명하세요. ★★

답안 **`threading`** 모듈은 하나의 프로세스 안에서 여러 스레드를 생성해 동시성을 구현하지만, 5단원에서 다룬 **GIL**의 제약으로 인해 한 순간에 하나의 스레드만 파이썬 코드를 실행할 수 있어 CPU 연산이 많은 작업에는 큰 성능 향상을 주지 못합니다. 대신 여러 서버에 순차적으로 SSH 접속하거나 다수의 API를 호출하는 것처럼 네트워크/디스크 **I/O** 대기 시간이 많은 작업에서는 대기 중 다른 스레드가 실행될 수 있어 threading만으로도 충분히 효과적인 성능 향상을 얻을 수 있습니다.
`multiprocessing` 모듈은 여러 개의 독립된 파이썬 프로세스를 생성해 각자 별도의 GIL과 메모리 공간을 가지므로, 대량의 로그 파싱이나 데이터 변환처럼 **CPU** 연산이 많은 작업을 여러 CPU 코어에 분산시켜 실질적인 병렬 처리 성능을 얻을 수 있습니다. 다만 프로세스 간에는 메모리를 직접 공유할 수 없어 `Queue`나 `Pipe` 같은

별도의 IPC 메커니즘을 통해 데이터를 주고받아야 하고, 프로세스 생성 자체의 오버헤드가 스레드보다 크다는 점도 고려해야 합니다.

관련 개념 GIL 제약, I/O-bound vs CPU-bound 선택 기준, multiprocessing.Queue, concurrent.futures

---