Poetry 치트시트

프로젝트 초기화 및 관리

- `poetry new <project_name>`: 표준적인 프로젝트 디렉토리 구조(소스 코드, 테스트, 설정 파일 등)와 함께 새 프로젝트를 생성합니다.
- `poetry init`: 이미 존재하는 디렉토리에서 대화형으로 `pyproject.toml` 파일을 생성합니다.
- `poetry check`: `pyproject.toml` 파일의 구문 오류나 의존성 정의의 유효성을 검사합니다.
- `poetry show`: 현재 프로젝트에 설정된 모든 의존성 목록을 나열합니다.
  - `--tree`: 의존성 간의 관계를 트리 형태로 시각화하여 보여줍니다.
  - `--outdated`: 설치된 패키지 중 더 높은 버전이 있는지 확인합니다.
- `poetry search <package>`: PyPI에서 패키지를 검색합니다.

의존성(Dependencies) 관리

- `poetry add <package>`: 프로젝트에 새 의존성을 추가하고 `pyproject.toml`과 `poetry.lock` 파일을 자동으로 업데이트합니다.
  - `poetry add "pandas@^1.0"`: 버전 제약 조건을 명시하여 패키지를 추가합니다.
  - `poetry add --group dev <package>`: 개발용 의존성 그룹(예: `pytest`, `black`)에 추가합니다.
  - `poetry add "git+https://github.com/user/repo.git"`: Git 리포지토리 주소로부터 직접 패키지를 추가합니다.
- `poetry remove <package>`: 설치된 의존성을 제거합니다.
- `poetry install`: `poetry.lock` 파일이 존재하면 명시된 정확한 버전의 패키지들을 설치합니다. 락 파일이 없다면 `pyproject.toml`을 기반으로 의존성 관계를 계산(resolve)하고 락 파일을 새로 생성합니다.
  - `--no-dev`: 개발용 의존성을 제외하고 기본 패키지만 설치합니다. (운영 환경 권장)
  - `--with <group>`: 특정 의존성 그룹을 포함하여 설치합니다.
  - `--sync`: 실제 설치된 환경을 `poetry.lock` 파일의 상태와 정확히 일치시킵니다. (목록에 없는 패키지 제거)
- `poetry update`: `pyproject.toml`의 제약 조건 내에서 의존성 패키지들을 최신 버전으로 업데이트하고 락 파일을 갱신합니다.
- `poetry lock`: 패키지를 실제로 설치하지 않고 `pyproject.toml` 설정을 바탕으로 `poetry.lock` 파일만 생성하거나 업데이트합니다.

가상 환경 및 실행 제어

- `poetry run <command>`: 프로젝트 전용 가상 환경 내에서 명령어를 실행합니다. (예: `poetry run pytest`)
- `poetry shell`: 프로젝트 가상 환경이 활성화된 새로운 셸 세션을 시작합니다.
- `poetry config virtualenvs.in-project true`: 가상 환경 폴더(`.venv`)를 프로젝트 디렉토리 내부에 생성하도록 설정합니다. (전역 설정)
- `poetry env info`: 현재 사용 중인 가상 환경의 상세 정보(경로, 파이썬 버전 등)를 확인합니다.
- `poetry env list`: 현재 프로젝트와 연결된 모든 가상 환경 목록을 보여줍니다.
- `poetry env use <python_executable>`: 프로젝트에서 사용할 파이썬 인터프리터를 변경합니다.
- `poetry env remove <python_version>`: 생성된 특정 가상 환경을 삭제합니다.

빌드 및 배포

- `poetry build`: 배포 가능한 소스 패키지(sdist)와 휠(wheel) 파일을 `dist/` 디렉토리에 빌드합니다.
- `poetry publish`: 빌드된 패키지를 PyPI 또는 설정된 저장소에 배포(publish)합니다.
  - `--build`: 배포 직전에 빌드 작업을 먼저 수행합니다.
  - `--repository <name>`: 전역 설정에 등록된 특정 저장소로 배포합니다.
- `poetry config repositories.<name> <url>`: 전역 설정에 외부 패키지 저장소 주소를 등록합니다.

`pyproject.toml` 설정 파일 상세 분석

```toml
[tool.poetry]
name = "my-awesome-project"
version = "0.1.0"
description = "프로젝트에 대한 간단한 설명"
authors = ["홍길동 <gildong@example.com>"]
license = "MIT"
readme = "README.md"

# ^: 호환되는 버전 범위 (예: ^1.2.3 -> >=1.2.3, <2.0.0)
# ~: 근사 버전 범위 (예: ~1.2.3 -> >=1.2.3, <1.3.0)
[tool.poetry.dependencies]
python = "^3.9"
requests = "^2.25.1"
# 선택적 의존성 (extras)
scipy = { version = "^1.7.3", optional = true }

# 개발 도구용 의존성 그룹
[tool.poetry.group.dev.dependencies]
pytest = "^6.2.2"
black = {version = "^22.3.0", allow-prereleases = true}

# 문서 생성용 의존성 그룹
[tool.poetry.group.docs.dependencies]
mkdocs = "^1.2.3"

# Extras 정의 (필요한 경우에만 설치)
[tool.poetry.extras]
science = ["scipy"]

# 명령줄에서 바로 실행할 스크립트 정의
# poetry run my-script 형식으로 실행 가능
[tool.poetry.scripts]
my-script = "my_awesome_project.main:app"

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

- Extras 패키지 설치 방법: `poetry install --extras "science"`

권장되는 Poetry 워크플로우

1. **프로젝트 시작**: `poetry new my-project` 명령으로 기초 구조 생성
2. **가상 환경 경로 설정**: `poetry config virtualenvs.in-project true` 설정 (IDE 연동에 유리)
3. **의존성 구성**: `poetry add pandas`, `poetry add --group dev pytest` 등으로 패키지 추가
4. **개발 진행**: `poetry shell`로 가상 환경에 진입하거나 `poetry run`을 통해 코드 실행
5. **검증 및 테스트**: `poetry run pytest` 명령으로 테스트 수행
6. **버전 관리**: `poetry.lock` 파일을 포함하여 Git에 커밋 (팀원 간 동일 환경 보장)
7. **패키지 배포**: `poetry publish --build` 명령으로 PyPI 등에 최종 결과물 배포