

# 파이썬을 처음 사용하는 동료와 효율적으로 일하는 방법

당근마켓 이태현 (Walter)

**BACK TO US,  
BACK TO PYTHON**

발표에서 사용된 소스 코드

> <http://bit.ly/3NCK306>



# 파이썬을 처음 사용하는 동료와 효율적으로 일하는 방법

**BACK TO US,  
BACK TO PYTHON**

# *No Silver Bullet*

**BACK TO US,  
BACK TO PYTHON**

# 파이썬을 처음 사용하는 동료와 효율적으로 일하는 방법

BACK TO US,  
BACK TO PYTHON

# 효율적

: 들인 노력에 비하여 얻는 결과가 큰 것

BACK TO US,  
BACK TO PYTHON

# 효율적

: 들인 노력에 비하여 얻는 결과가 큰 것

BACK TO US,  
BACK TO PYTHON

노력 = 문제 발굴 및 해결

결과 = 비즈니스 가치

BACK TO US,  
BACK TO PYTHON



노력 = 문제 발굴 및 해결

결과 = 비즈니스 가치

BACK TO US,  
BACK TO PYTHON

노력 = 문제 발굴 및 해결

결과 = 비즈니스 가치

BACK TO US,  
BACK TO PYTHON

노력 ↓

비즈니스 가치 ↑

BACK TO US,  
BACK TO PYTHON

비즈니스 가치를 창출할 수 있는 환경  
= 비즈니스 로직에 집중할 수 있는 환경

비즈니스 가치를 창출할 수 있는 환경  
= 비즈니스 로직에 집중할 수 있는 환경

# 모든 게 비즈니스 가치와 연관된 거 아니야?

**BACK TO US,  
BACK TO PYTHON**

# 스페이스(Space) vs 탭(Tab)

BACK TO US,  
BACK TO PYTHON

# 79글자 vs 80글자

BACK TO US,  
BACK TO PYTHON



파이썬은 모든 개발자가 좋아하는 언어예요.

파이썬만의 특징을 일컫는 파이써닉함에 매료된 개발자들이 무척 많죠.

그러나 단점도 존재하기 마련이다. 대표적으로 동적 타입 언어라는 특징 때문에 안정성이 떨어진다는 공격을 받기도 한다.

파이썬은 모든 개발자가 좋아하는 언어예요.

파이썬만의 특징을 일컫는 파이써닉함에 매료된 개발자들이 무척 많죠.

그러나 단점도 존재하기 마련이다. 대표적으로 동적 타입 언어라는 특징 때문에 안정성이 떨어진다는 공격을 받기도 한다.

# 좀 어색한데?

**BACK TO US,  
BACK TO PYTHON**

파이썬 = 프로그래밍 '언어'

BACK TO US,  
BACK TO PYTHON

파이썬 = 프로그래밍  
'언어'

스타일 고민  
→ 소통

BACK TO US,  
BACK TO PYTHON

파이썬 = 프로그래밍  
'언어'

규칙  
적용  
스타일 고민  
→ 소통

BACK TO US,  
BACK TO PYTHON

# *No Silver Bullet*

**BACK TO US,  
BACK TO PYTHON**

- 파이썬이라는 언어의 특징
- 효율적으로 업무하는 방법



- 파이썬이라는 언어의 특징
- 효율적으로 업무하는 방법

- 파이썬이라는 언어의 특징
- 효율적으로 업무하는 방법

# 파이썬이라는 언어의 특징

**BACK TO US,  
BACK TO PYTHON**

1. 가상환경 및 패키지 관리
2. 동적 타입 언어
3. 여러 패키지 활용

1. 가상환경 및 패키지 관리

2. 동적 타입 언어

3. 여러 패키지 활용

1. 가상환경 및 패키지 관리

2. 동적 타입 언어

3. 여러 패키지 활용

1. 가상환경 및 패키지 관리
2. 동적 타입 언어
3. 여러 패키지 활용

**FastAPI**

웹 프레임워크 패키지

관계형 데이터베이스  
마이그레이션 패키지

**Alembic**

**BACK TO US,  
BACK TO PYTHON**



FastAPI

Alembic

**BACK TO US,  
BACK TO PYTHON**

1. 가상환경 및 패키지 관리
2. 동적 타입 언어
3. 여러 패키지 활용

# 가상환경 생성

**BACK TO US,  
BACK TO PYTHON**

## 1. 내장된 *venv* 명령어 사용

```
$ python -m venv .venv
```

```
$ source .venv/bin/activate
```

```
$ deactivate
```

## 1. 내장된 *venv* 명령어 사용

```
$ python -m venv .venv
```

```
$ source .venv/bin/activate
```

```
$ deactivate
```

## 1. 내장된 *venv* 명령어 사용

```
$ python -m venv .venv
```

```
$ source .venv/bin/activate
```

```
$ deactivate
```

## 1. 내장된 *venv* 명령어 사용

```
$ python -m venv .venv
```

```
$ source .venv/bin/activate
```

```
$ deactivate
```

## 2. *conda* 명령어를 통한 아나콘다 또는 미니콘다 사용

```
$ conda create -n venv python=3.11
```

```
$ conda activate venv
```

```
$ conda deactivate
```



## 2. *conda* 명령어를 통한 아나콘다 또는 미니콘다 사용

```
$ conda create -n venv python=3.11
```

```
$ conda activate venv
```

```
$ conda deactivate
```

## 2. *conda* 명령어를 통한 아나콘다 또는 미니콘다 사용

```
$ conda create -n venv python=3.11
```

```
$ conda activate venv
```

```
$ conda deactivate
```

# 패키지 설치 및 관리

**BACK TO US,  
BACK TO PYTHON**

## 1. *pip* 명령어를 통한 패키지 설치

```
$ source .venv/bin/activate
```

```
$ pip install requests
```

## 1. *pip* 명령어를 통한 패키지 설치

```
$ source .venv/bin/activate
```

```
$ pip install requests
```

## 1. *pip* 명령어를 통한 패키지 설치

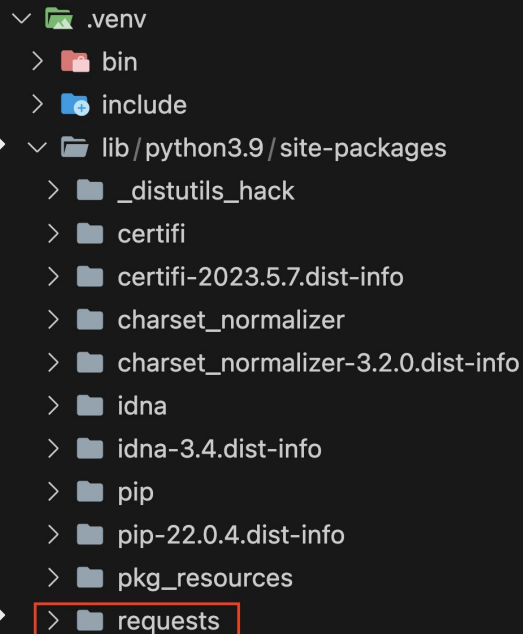
```
$ source .venv/bin/activate
```

```
$ pip install requests
```

## 패키지 설치 및 관리

설치되는  
패키지들이  
위치하는 곳

requests 패키지



```

└─ .venv
   ├── bin
   ├── include
   └─ lib/python3.9/site-packages
      ├── _distutils_hack
      ├── certifi
      ├── certifi-2023.5.7.dist-info
      ├── charset_normalizer
      ├── charset_normalizer-3.2.0.dist-info
      ├── idna
      ├── idna-3.4.dist-info
      ├── pip
      ├── pip-22.0.4.dist-info
      ├── pkg_resources
      └── requests

```

## 2. *requirements.txt* 파일을 통한 패키지 관리

```
$ pip freeze > requirements.txt
```

```
$ cat requirements.txt
```

```
> requests==2.31.0
```

```
> charset-normalizer==3.2.0
```



## 2. *requirements.txt* 파일을 통한 패키지 관리

```
$ pip freeze > requirements.txt
```

```
$ cat requirements.txt
```

```
> requests==2.31.0
```

```
> charset-normalizer==3.2.0
```

## 2. *requirements.txt* 파일을 통한 패키지 관리

```
$ pip freeze > requirements.txt
```

```
$ cat requirements.txt
```

```
> requests==2.31.0
```

```
> charset-normalizer==3.2.0
```

1. **source** 및 **pip** 등 익혀야 하는 명령어
2. 패키지 의존성 관리의 취약함

1. **source** 및 **pip** 등 익혀야 하는 명령어
2. 패키지 의존성 관리의 취약함

1. **source** 및 **pip** 등 익혀야 하는 명령어

2. 패키지 의존성 관리의 취약함

| 패키지 의존성 관리의 취약함

```
$ pip uninstall requests
```

```
$ pip freeze > requirements.txt
```

```
$ cat requirements.txt
```

```
> charset-normalizer==3.2.0
```

| 패키지 의존성 관리의 취약함

```
$ pip uninstall requests
```

```
$ pip freeze > requirements.txt
```

```
$ cat requirements.txt
```

```
> charset-normalizer==3.2.0
```

| 패키지 의존성 관리의 취약함

```
$ pip uninstall requests
```

```
$ pip freeze > requirements.txt
```

```
$ cat requirements.txt
```

```
> charset-normalizer==3.2.0
```



| 패키지 의존성 관리의 취약함

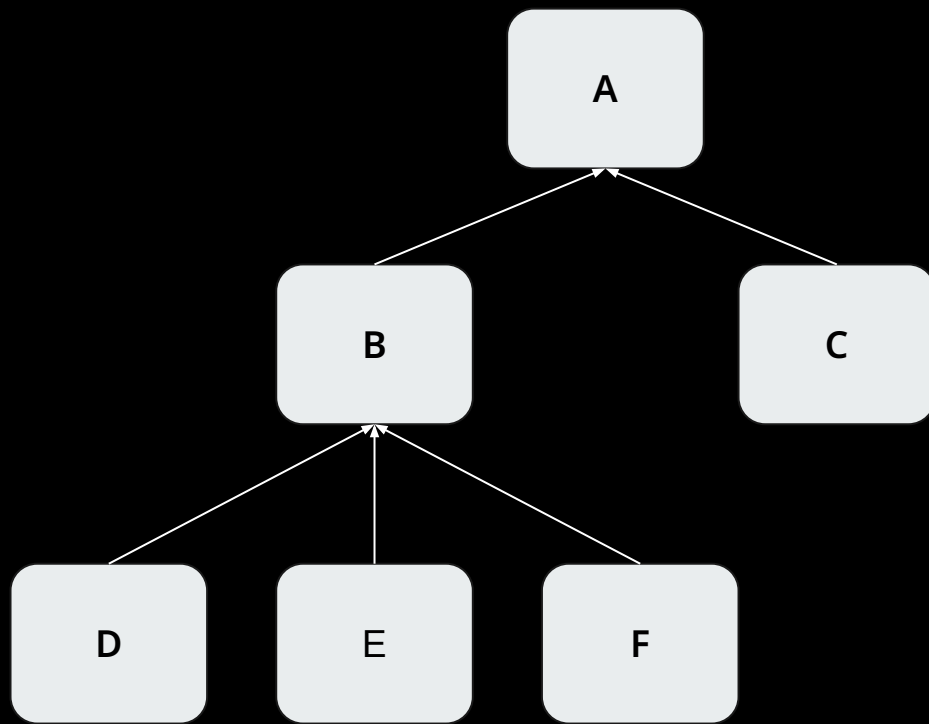
```
$ pip uninstall requests
```

```
$ pip freeze > requirements.txt
```

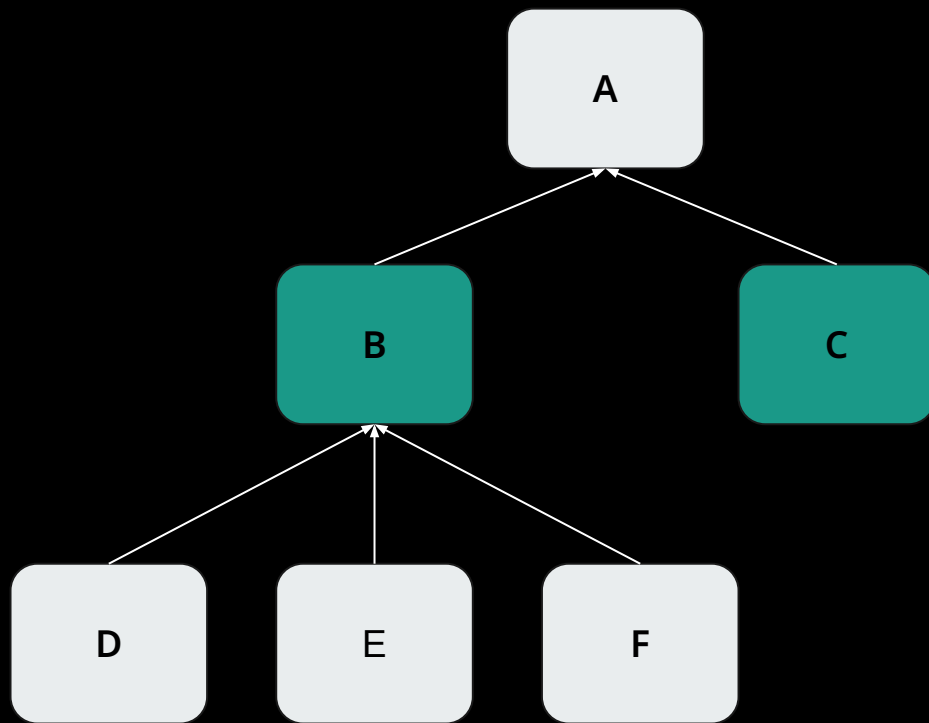
```
$ cat requirements.txt
```

```
> charset-normalizer==3.2.0
```

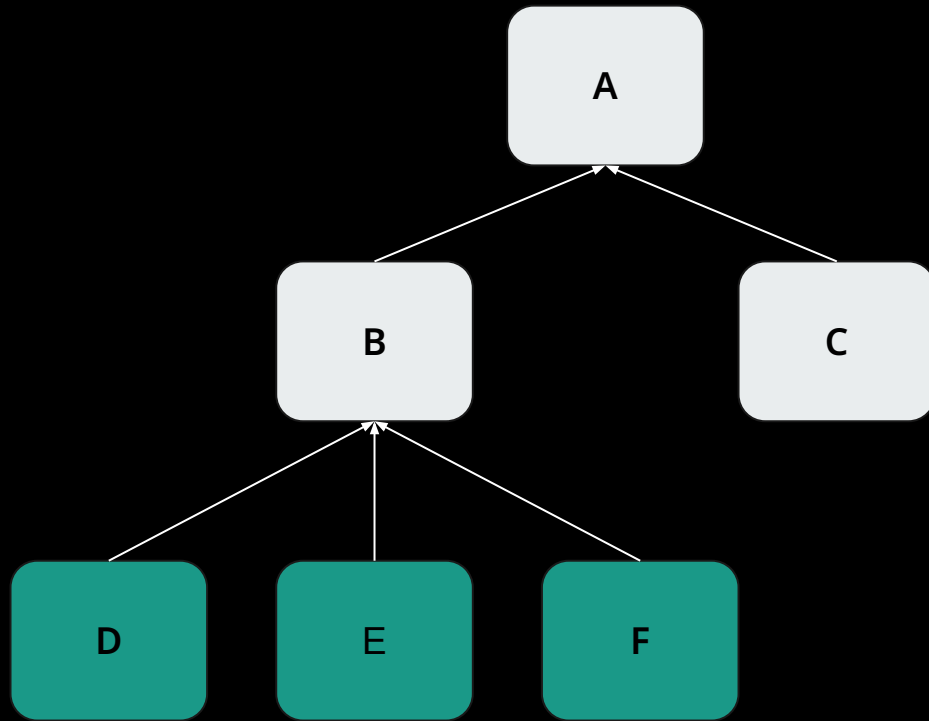
## 패키지 취약성



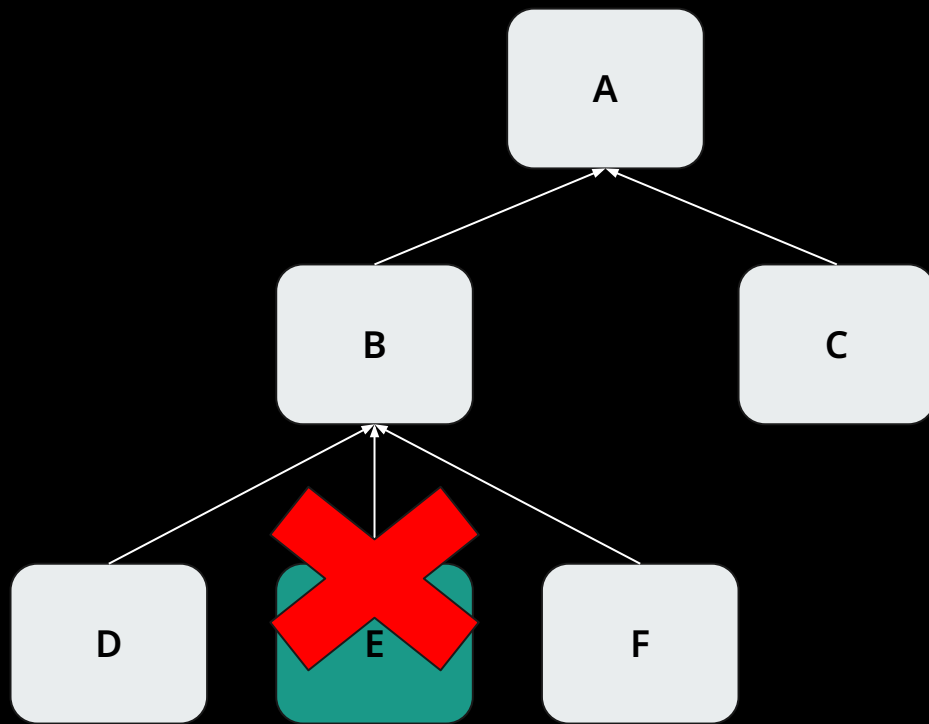
## 패키지 취약성



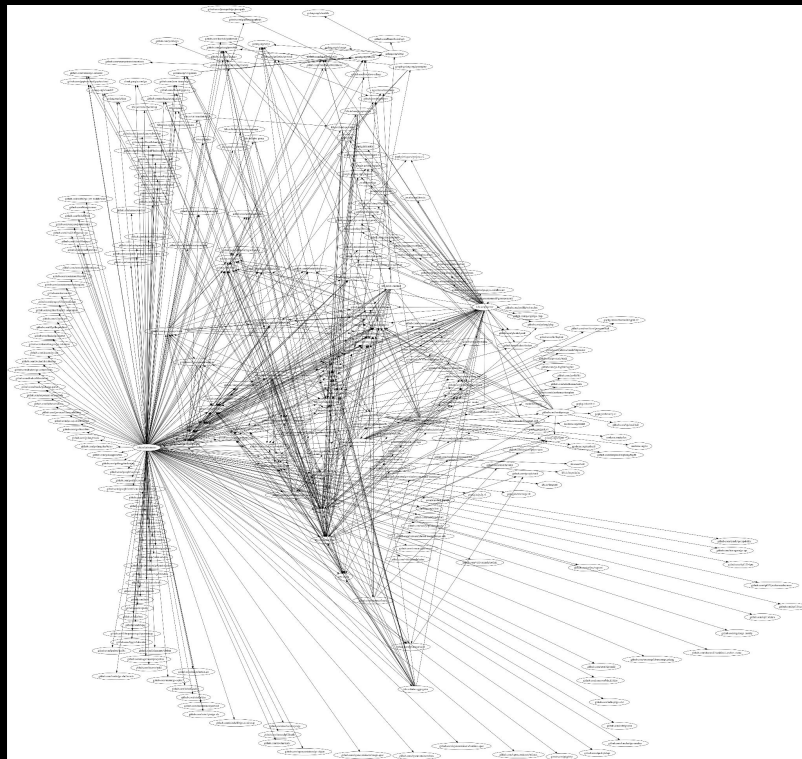
## 패키지 취약성



## 패키지 취약성



## 패키지 취약성



BACK TO US,  
BACK TO PYTHON

PYCON KOREA 2023

# Poetry 패키지 사용

**BACK TO US,  
BACK TO PYTHON**

## Poetry 패키지 사용

### 1. 가상환경 생성

```
$ poetry init
```

```
$ ls
```

```
> pyproject.toml
```



## Poetry 패키지 사용

### 1. 가상환경 생성

```
$ poetry init
```

```
$ ls
```

```
> pyproject.toml
```

## Poetry 패키지 사용

### 1. 가상환경 생성

```
$ poetry init
```

```
$ ls
```

```
> pyproject.toml
```

## 2. 패키지 설치 및 관리

```
$ poetry add requests
```

```
$ ls
```

```
> poetry.lock
```

```
> pyproject.toml
```

## 2. 패키지 설치 및 관리

```
$ poetry add requests
```

```
$ ls
```

```
> poetry.lock
```

```
> pyproject.toml
```

## 2. 패키지 설치 및 관리

```
$ poetry add requests
```

```
$ ls
```

```
> poetry.lock
```

```
> pyproject.toml
```

## Poetry 패키지 사용

패키지 정보



```
[[package]]
name = "requests"
version = "2.31.0"
description = "Python HTTP for Humans."
optional = false
python-versions = ">=3.7"
files = [
  {
    file = "requests-2.31.0-py3-none-any.whl",
    hash = "sha256:58cd2187c01e70e6e26505bca751777aa9f2ee0b7f4300988b709f44e013003f"
  },
  {
    file = "requests-2.31.0.tar.gz",
    hash = "sha256:942c5a758f98d790eaed1a29cb6eefc7ffb0d1cf7af05c3d2791656dbd6ad1e1"
  },
]
```

의존성 관리



```
[package.dependencies]
certifi = ">=2017.4.17"
charset-normalizer = ">=2,<4"
idna = ">=2.5,<4"
urllib3 = ">=1.21.1,<3"
```

## Poetry 패키지 사용

```
[package.dependencies]
```

```
certifi = ">=2017.4.17"
```

```
charset-normalizer = ">=2,<4"
```

```
idna = ">=2.5,<4"
```

```
urllib3 = ">=1.21.1,<3"
```

## Poetry 패키지 사용

```
[package.dependencies]
```

```
certifi = ">=2017.4.17"
```

```
charset-normalizer = ">=2,<4"
```

```
idna = ">=2.5,<4"
```

```
urllib3 = ">=1.21.1,<3"
```



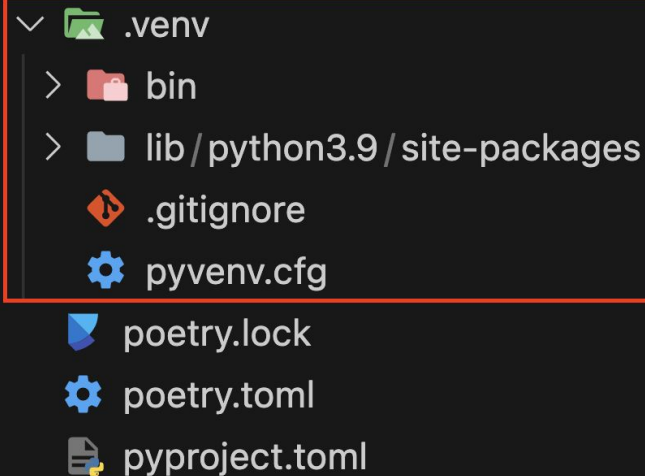
## Poetry 패키지 사용

[virtualenvs]

create = true

in-project = true

path = ".venv"



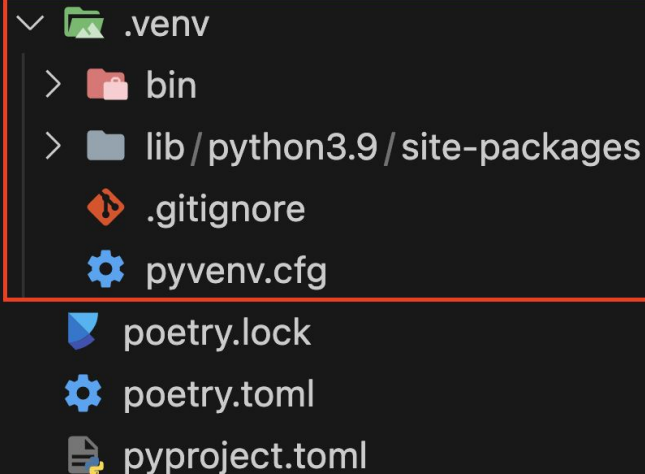
## Poetry 패키지 사용

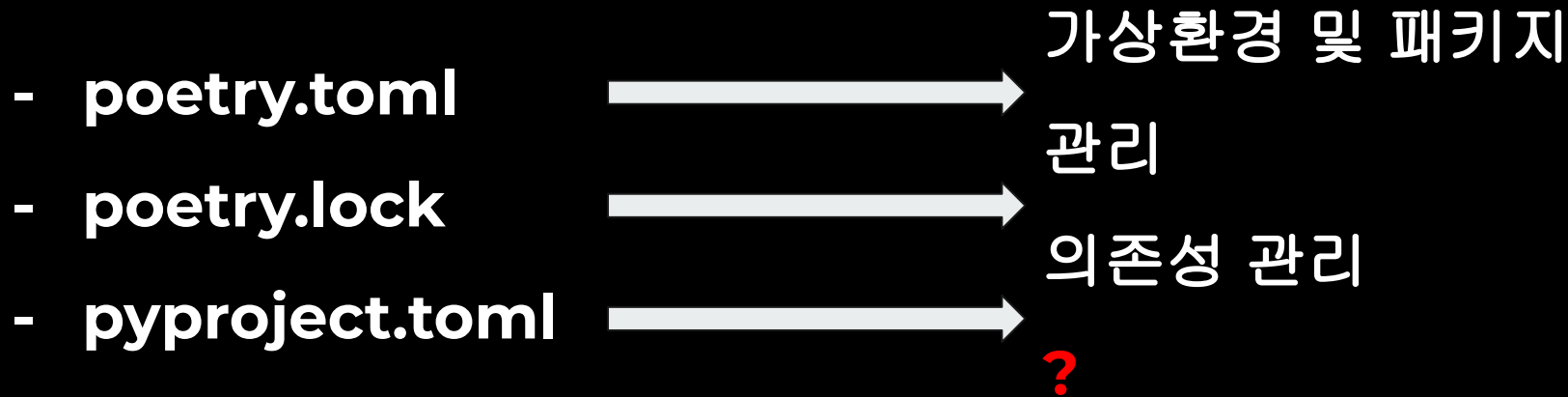
[virtualenvs]

**create** = true

**in-project** = true

path = ".venv"





# TOML

## : Tom's Obvious Minimal Language

BACK TO US,  
BACK TO PYTHON

## TOML 파일

```
[tool.poetry.dependencies]
```

```
python = "^3.11"
```

```
requests = "^2.31.0" ← $ poetry add requests
```

```
[tool.poetry.group.test.dependencies]
```

```
pytest = "^7.4.0" ← $ poetry add pytest --group test
```

## TOML 파일

```
[tool.poetry.dependencies]
```

```
python = "^3.11"
```

```
requests = "^2.31.0" ← $ poetry add requests
```

```
[tool.poetry.group.test.dependencies]
```

```
pytest = "^7.4.0" ← $ poetry add pytest --group test
```

## TOML 파일

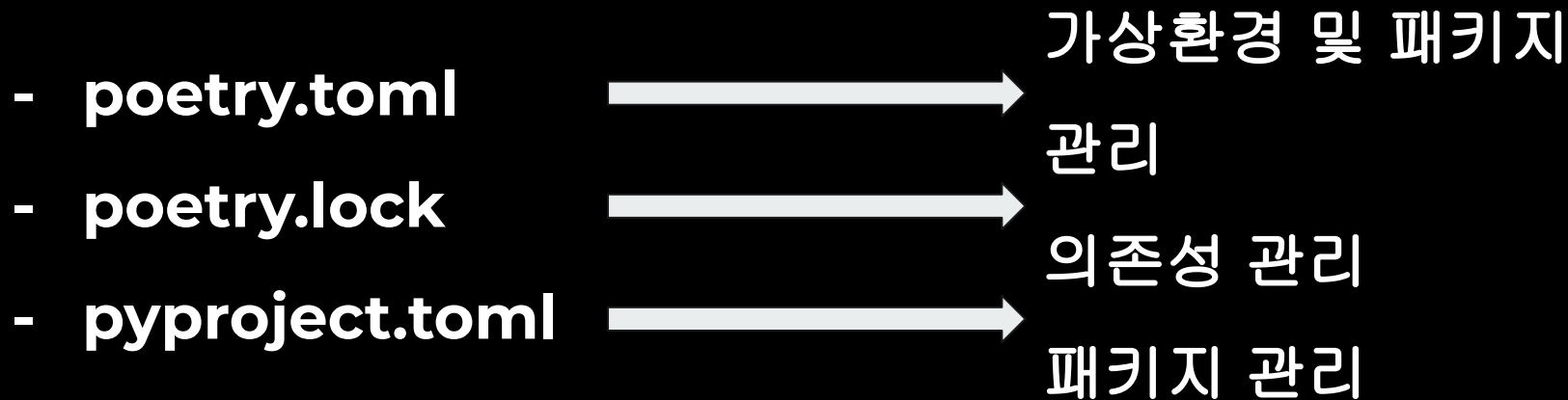
**[tool.poetry.dependencies]**

python = "^3.11"

requests = "^2.31.0" ← \$ poetry add requests

**[tool.poetry.group.test.dependencies]**

pytest = "^7.4.0" ← \$ poetry add pytest --group test





# 동적 타입 언어

**BACK TO US,  
BACK TO PYTHON**

## 동적 타입 언어

```
a = 1
```

```
type(a) # <class 'int'>
```

```
a = 'a'
```

```
type(a) # <class 'str'>
```

## 동적 타입 언어

```
a = 1
```

```
type(a) # <class 'int'>
```

```
a = 'a'
```

```
type(a) # <class 'str'>
```

## 동적 타입 언어

```
a = 1
```

```
type(a) # <class 'int'>
```

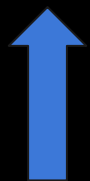
```
a = 'a'
```



다른 자료형의 값 재할당

```
type(a) # <class 'str'>
```

소스  
코드



안정성



BACK TO US,  
BACK TO PYTHON

소스  
코드

타입 힌팅은 말 그대로 힌트일 뿐!

안정성

BACK TO US,  
BACK TO PYTHON

## 동적 타입 언어

```
a: int = 1
```

```
type(a) # <class 'int'>
```

```
a: int = 'a'
```

```
type(a) # <class 'str'>
```

## 동적 타입 언어

```
a: int = 1
```

```
type(a) # <class 'int'>
```

```
a: int = 'a'
```

```
type(a) # <class 'str'>
```



## 동적 타입 언어

```
a: int = 1
```

```
type(a) # <class 'int'>
```

```
a: int = 'a'
```



타입 힌팅과 다른 자료형의 값 할당

```
type(a) # <class 'str'>
```

## 동적 타입 언어

```
a: int = 1
```

```
type(a) # <class 'int'>
```

```
a: int = 'a'
```



타입 힌팅과 다른 자료형의 값 할당

```
type(a) # <class 'str'>
```

# MyPy 패키지 사용

**BACK TO US,  
BACK TO PYTHON**

## MyPy 패키지 사용

```
$ poetry run mypy .
```

```
> error: incompatible types in assignment  
    (expression has type "int", variable has type "str")
```

## MyPy 패키지 사용

```
$ poetry run mypy .
```

```
> error: incompatible types in assignment
```

```
(expression has type "int", variable has type "str")
```

## | MyPy 패키지 사용

```
$ poetry run mypy .
```

```
> error: incompatible types in assignment
```

```
(expression has type "int", variable has type "str")
```

## MyPy 패키지 사용

```
$ poetry run mypy .
```

```
> error: incompatible types in assignment  
    (expression has type "int", variable has type "str")
```

## MyPy 패키지 사용

```
$ poetry run mypy . --exclude test/
```

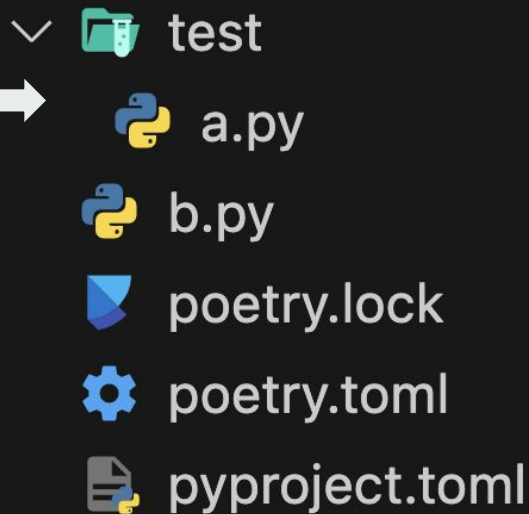
```
> error: incompatible types in assignment
```

```
(expression has type "int", variable has type "str")
```



## MyPy 패키지 사용

```
[tool.mypy]  
exclude = [  
    'test/',  
]
```



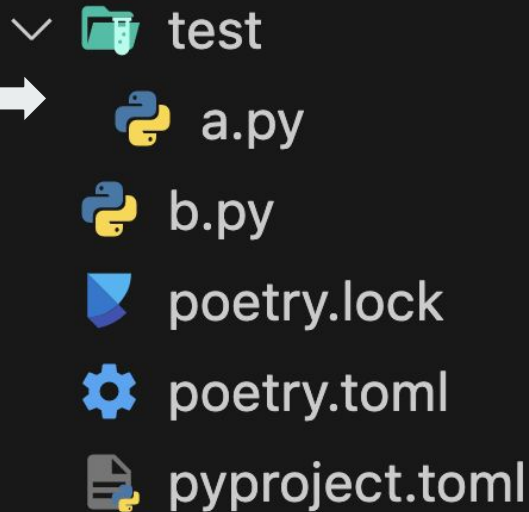
## MyPy 패키지 사용

```
[tool.mypy]
```

```
exclude = [
```

```
    'test/',
```

```
]
```



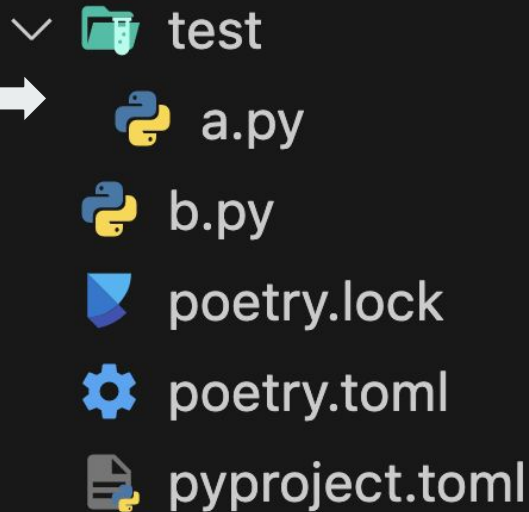
## MyPy 패키지 사용

```
[tool.mypy]
```

```
exclude = [
```

```
    'test/',
```

```
]
```



# 효율적으로 업무하는 방법

**BACK TO US,  
BACK TO PYTHON**

1. 가상환경 및 패키지 관리
2. 동적 타입 언어
3. 여러 패키지 활용

파이썬은 모든 개발자가 좋아하는 언어예요.

파이썬만의 특징을 일컫는 파이써닉함에 매료된 개발자들이 무척 많죠.

그러나 단점도 존재하기 마련이다. 대표적으로 동적 타입 언어라는 특징 때문에 안정성이 떨어진다는 공격을 받기도 한다.

# 파이썬의 대표적인 스타일 가이드

- **PEP8** (파이썬을 위한 스타일 가이드)
- 구글 파이썬 스타일 가이드

# 파이썬의 대표적인 스타일 가이드

- **PEP8** (파이썬을 위한 스타일 가이드)
- 구글 파이썬 스타일 가이드



# 파이썬의 대표적인 스타일 가이드

- PEP8 (파이썬을 위한 스타일 가이드)
- 구글 파이썬 스타일 가이드

# 파이썬의 대표적인 스타일 가이드

- PEP8 (파이썬을 위한 스타일 가이드) → 79 글자
- 구글 파이썬 스타일 가이드 → 80 글자

# 파이썬의 대표적인 스타일 가이드

- PEP8 (파이썬을 위한 스타일 가이드) → 79 글자
- 구글 파이썬 스타일 가이드 → 80 글자

# ***A Foolish Consistency is the Hobgoblin of Little Minds***

**BACK TO US,  
BACK TO PYTHON**

# Black 패키지와 Isort 패키지

BACK TO US,  
BACK TO PYTHON

| **Black** 패키지와 **Isort** 패키지

**[tool.black]**

**line-length = 30**

| **Black** 패키지와 **Isort** 패키지

**[tool.black]**

**line-length = 30**

**Black** 패키지와 **Isort** 패키지

```
import requests # 순서가 섞여 있는 패키지
```

```
import datetime
```

```
a: str = "abcdefg" + "hijklmnop" # 30 글자가 넘는 문자열
```



| Black 패키지와 Isort 패키지

```
import requests # 순서가 섞여 있는 패키지
```

```
import datetime
```

```
a: str = "abcdefg" + "hijklmnop" # 30 글자가 넘는 문자열
```

| **Black** 패키지와 **Isort** 패키지

```
import requests # 순서가 섞여 있는 패키지
```

```
import datetime
```

```
a: str = "abcdefg" + "hijklmnop" # 30 글자가 넘는 문자열
```

## Black 패키지와 Isort 패키지

```
import datetime # 내장 모듈
```

```
import requests # 외부 모듈
```

```
a: str = (  
    "abcdefg" + "hijklmnop"  
)
```

## Black 패키지와 Isort 패키지

```
import datetime # 내장 모듈
```

```
import requests # 외부 모듈
```

```
a: str = (  
    "abcdefg" + "hijklmnop"  
)
```

## Black 패키지와 Isort 패키지

```
import datetime # 내장 모듈
```

```
import requests # 외부 모듈
```

```
a: str = (  
    "abcdefg" + "hijklmnop"  
)
```

# 또다른 포매팅 패키지 Flake8

: `pyproject.toml` 파일 미지원

# 무수히 많은 명령어

**BACK TO US,  
BACK TO PYTHON**

FastAPI

MyPy

Alembic

Isort

Uvicorn

Black

BACK TO US,  
BACK TO PYTHON



FastAPI

MyPy

Alembic

Isort

Uvicorn

Black

BACK TO US,  
BACK TO PYTHON

| 무수히 많은 명령어

`$ poetry run mypy .`

`$ poetry run black .`

`$ poetry run isort .`

`$ poetry run alembic revision -m "마이그레이션"`

`$ poetry run alembic upgrade head`

`$ poetry run uvicorn run main:app --reload`

# Makefile 파일 사용

**BACK TO US,  
BACK TO PYTHON**

## Makefile 파일 사용

**.PHONY: lint**

**lint:**

poetry run mypy . && poetry run black . && poetry run isort .

**.PHONY: run**

**run:**

make lint && make migrate && poetry run uvicorn main:app --reload

**.PHONY: migrate**

**migrate:**

poetry run alembic revision -m "마이그레이션" && poetry run alembic upgrade head

## | Makefile 파일 사용

**.PHONY: lint**

**lint:**

poetry run mypy . && poetry run black . && poetry run isort .

**.PHONY: run**

**run:**

make lint && make migrate && poetry run uvicorn main:app --reload

**.PHONY: migrate**

**migrate:**

poetry run alembic revision -m "마이그레이션" && poetry run alembic upgrade head

## Makefile 파일 사용

**.PHONY:** lint

**lint:**

poetry run mypy . && poetry run black . && poetry run isort .

**.PHONY:** run

**run:**

make lint && make migrate && poetry run uvicorn main:app --reload

**.PHONY:** migrate

**migrate:**

poetry run alembic revision -m "마이그레이션" && poetry run alembic upgrade head

## Makefile 파일 사용

**.PHONY:** lint

**lint:**

poetry run mypy . && poetry run black . && poetry run isort .

**.PHONY:** run

**run:**

make lint && make migrate && poetry run uvicorn main:app --reload

**.PHONY:** migrate

**migrate:**

poetry run alembic revision -m "마이그레이션" && poetry run alembic upgrade head

## Makefile 파일 사용

\$ make lint



하나의 명령어로 묶이고,

\$ make migrate

**make**라는 접두어로 예측가능한

\$ make run


명령어



## Makefile 파일 사용

`$ make lint`

`$ make migrate`

`$ make run`  `make run` 명령어에는  
`make lint` 및 `make migrate` 명령어가  
선행되게 정의되어 있다.

## Makefile 파일 사용



잘못된 커밋



# Pre-commit 패키지 사용

**BACK TO US,  
BACK TO PYTHON**

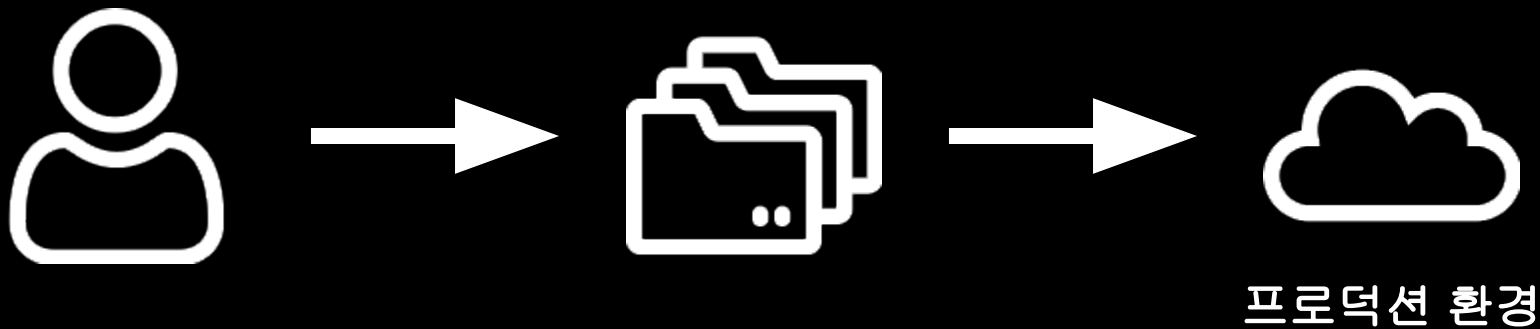
## Pre-commit 패키지 사용



## Pre-commit 패키지 사용



## Pre-commit 패키지 사용



## Pre-commit 패키지 사용



## Pre-commit 패키지 사용

repos:

- repo: <https://github.com/pre-commit/mirrors-mypy>

rev: 'v1.4.1'

hooks:

- id: **mypy**

- repo: <https://github.com/ambv/black>

rev: 23.3.0

hooks:

- id: **black**



## Pre-commit 패키지 사용

repos:

- repo: <https://github.com/pre-commit/mirrors-mypy>

rev: 'v1.4.1'

hooks:

- id: **mypy**

- repo: <https://github.com/ambv/black>

rev: 23.3.0

hooks:

- id: **black**

## Pre-commit 패키지 사용

repos:

- repo: <https://github.com/pre-commit/mirrors-mypy>

rev: 'v1.4.1'

hooks:

- id: **mypy**



- repo: <https://github.com/ambv/black>

rev: 23.3.0

hooks:

- id: **black**



**pyproject.toml**

파일에 정의된

설정값 조회 불가능

# 발표를 마치며

**BACK TO US,  
BACK TO PYTHON**

1. 가상환경 및 패키지 관리를 위한 **Poetry**
2. 동적 타입 언어의 한계를 극복하기 위한 **MyPy**
3. 통일된 코드 스타일을 위한 **Black / Isort**
4. 통일된 명령어 사용을 위한 **Makefile**
5. 인적 오류(**Human Error**)를 줄이기 위한 **pre-commit**

1. 가상환경 및 패키지 관리를 위한 **Poetry**
2. 동적 타입 언어의 한계를 극복하기 위한 **MyPy**
3. 통일된 코드 스타일을 위한 **Black / Isort**
4. 통일된 명령어 사용을 위한 **Makefile**
5. 인적 오류(**Human Error**)를 줄이기 위한 **pre-commit**

1. 가상환경 및 패키지 관리를 위한 **Poetry**
2. 동적 타입 언어의 한계를 극복하기 위한 **MyPy**
3. 통일된 코드 스타일을 위한 **Black / Isort**
4. 통일된 명령어 사용을 위한 **Makefile**
5. 인적 오류(**Human Error**)를 줄이기 위한 **pre-commit**

1. 가상환경 및 패키지 관리를 위한 **Poetry**
2. 동적 타입 언어의 한계를 극복하기 위한 **MyPy**
3. 통일된 코드 스타일을 위한 **Black / Isort**
4. 통일된 명령어 사용을 위한 **Makefile**
5. 인적 오류(**Human Error**)를 줄이기 위한 **pre-commit**

1. 가상환경 및 패키지 관리를 위한 **Poetry**
2. 동적 타입 언어의 한계를 극복하기 위한 **MyPy**
3. 통일된 코드 스타일을 위한 **Black / Isort**
4. 통일된 명령어 사용을 위한 **Makefile**
5. 인적 오류(**Human Error**)를 줄이기 위한 **pre-commit**



1. 가상환경 및 패키지 관리를 위한 **Poetry**
2. 동적 타입 언어의 한계를 극복하기 위한 **MyPy**
3. 통일된 코드 스타일을 위한 **Black / Isort**
4. 통일된 명령어 사용을 위한 **Makefile**
5. 인적 오류(**Human Error**)를 줄이기 위한 **pre-commit**

1. 가상환경 및 패키지 관리를 위한 **Poetry**
2. 동적 타입 언어의 한계를 극복하기 위한 **MyPy**
3. 통일된 코드 스타일을 위한 **Black / Isort**
4. 통일된 명령어 사용을 위한 **Makefile**
5. 인적 오류(**Human Error**)를 줄이기 위한 **pre-commit**