Изоляция сред на уровне приложений

В Python есть встроенный инструмент для создания виртуальных сред — модуль `venv`, который можно запустить непосредственно из системной оболочки. Чтобы создать новую виртуальную среду, введите следующую команду:

```
$ python3.9 -m venv <имя_среды>
```

Здесь `имя_среды` заменяется именем новой среды (вместо него можно указать абсолютный путь). Обратите внимание, что используется команда `python3.9`, а не просто `python3`. Дело в том, что в зависимости от среды команда `python3` может быть привязана к разным версиям интерпретатора, так что всегда лучше предельно ясно указать, какую конкретно версию Python использовать для создания новой виртуальной среды. Команда `python3.9 -m venv` создает новый каталог `имя_среды` в текущем рабочем каталоге. В созданном каталоге находятся несколько подкаталогов:

- `bin/`: здесь хранится новый исполняемый файл Python и сценарии или исполняемые файлы, предоставляемые другими пакетами.

ПРИМЕЧАНИЕ ДЛЯ ПОЛЬЗОВАТЕЛЕЙ WINDOWS

В Windows модуль venv придерживается других соглашений об именах во внутренней структуре каталогов. Вместо bin/, lib/ и include/ следует использовать Scripts/, Libs/ и Include/ в соответствии с соглашениями, принятыми в этой ОС. Команды для активации и деактивации среды тоже различаются: вместо source со сценариями activate и deactivate нужно вызывать ИМЯ_СРЕДЫ/Scripts/activate.bat и ИМЯ_СРЕДЫ/Scripts/deactivate.bat.

- `lib/` и `include/`: в этих каталогах находятся вспомогательные библиотечные файлы для нового интерпретатора Python в виртуальной среде. Новые пакеты будут устанавливаться в каталог `ИМЯ_СРЕДЫ/lib/pythonX.Y/site-packages/`.

Разработчики часто хранят свои виртуальные среды вместе с исходным кодом и называют каталоги сред типовыми именами вида .venv или venv. Многие интегрированные среды разработки (IDE) для Python распознают эти имена и автоматически подгружают библиотеки для автодополнения кода. Типовые имена также позволяют автоматически исключать каталоги виртуальных сред из систем управления версиями, что обычно рекомендуется делать. Например, пользователи Git могут добавить эти имена в глобальный файл .gitignore с шаблонами путей, которые надо игнорировать при управлении версиями исходного кода.

После того как новая среда будет создана, ее надо активировать в текущем сеансе командной оболочки. Если вы используете оболочку Bash, виртуальную среду можно активировать командой `source`:

```
$ source имя_среды/bin/activate
```

Есть и более короткая версия команды, которая должна работать в любой POSIX-совместимой системе независимо от оболочки:

```
$ . имя_среды/bin/activate
```

Эта команда изменяет состояние текущего сеанса оболочки, затрагивая ее переменные окружения. Чтобы пользователь понимал, что виртуальная среда активирована, приглашение в командной строке изменяется: в его начало добавляется строка `(ИМЯ_СРЕДЫ)`. Вот пример, в котором создается и активируется новая среда:

```
$ python3 -m venv example
$ source example/bin/activate
(example) $ which python
/home/swistakm/example/bin/python
(example) $ deactivate
$ which python
/usr/local/bin/python
```

Важно заметить, что `venv` не предоставляет никаких дополнительных возможностей, чтобы отслеживать, какие пакеты должны быть установлены в виртуальной среде. Кроме того, виртуальные среды непереносимы и их не следует перемещать в другую систему, на другой компьютер и даже в другое место этой же файловой системы. Это значит, что каждый раз, когда вы захотите установить свое приложение на новом хосте, придется создавать новую виртуальную среду.

Из-за этого пользователи `pip` выработали полезный принцип: хранить спецификацию всех зависимостей проекта в одном месте. Проще всего для этого создать файл требований с общепринятым именем requirements.txt, пример содержимого которого показан ниже:

```
# Строки, начинающиеся с решетки (#), считаются комментариями.

# фиксированные спецификаторы версий лучше всего для воспроизводимости
eventlet==0.17.4
graceful==0.1.1

# для проектов, тщательно протестированных с разными
# версиями зависимостей, допускаются диапазоны версий
falcon>=0.3.0,<0.5.0

# следует избегать пакетов без указания версии, кроме ситуации,
# когда всегда желателен/необходим последний выпуск
pytz
```

С таким файлом все зависимости можно легко установить за один шаг. Команда `pip install` поддерживает формат подобных файлов требований. Путь к файлу можно указать с флагом `-r`, как в примере:

```
$ pip install -r requirements.txt
```

Учтите, что в файлах требований перечисляются только те пакеты, которые надо установить, а не те, которые уже установлены в вашей среде. Если вы установите какой-то пакет вручную, он не будет автоматически отражен в файле требований. Таким образом, нужно специально заботиться о том, чтобы файл требований поддерживался в актуальном состоянии, особенно в больших и сложных проектах.

Команда `pip freeze` выводит список всех пакетов в текущей среде вместе с их версиями, однако использовать ее следует с осторожностью. В список также попадут зависимости ваших зависимостей, так что в больших проектах он очень быстро разрастается. Вам придется тщательно проверять, нет ли в списке пакетов, установленных по ошибке или случайно.

Если проект требует улучшенной воспроизводимости виртуальных сред и жесткого контроля за установленными зависимостями, то вам может понадобиться более мощный инструмент. Один из таких инструментов — Poetry — будет рассмотрен в следующем разделе.

Poetry как система управления зависимостями

Poetry реализует относительно новый подход к управлению зависимостями и виртуальными средами в Python. Цель этого проекта с открытым кодом — предоставить более предсказуемую и удобную среду для работы с экосистемой пакетов Python.

Так как Poetry является пакетом PyPI, его можно установить командой `pip`:

```
$ pip install --user poetry
```

Обратите внимание, что Poetry создает виртуальные среды Python, поэтому этот пакет не следует устанавливать внутри самой виртуальной среды. Poetry можно установить в пользовательский или глобальный каталог пакетов, хотя пользовательский каталог считается предпочтительным (см. раздел «Изоляция среды выполнения».

Как уже упоминалось в разделе «Установка пакетов Python с помощью pip», эта команда устанавливает Poetry в каталог пакетов. В зависимости от конфигурации системы это будет либо глобальный, либо пользовательский каталог. Что-

бы избежать этой неоднозначности, создатели Poetry рекомендуют использовать альтернативный метод инициализации.

В macOS, Linux и других POSIX-совместимых системах Poetry можно установить с помощью утилиты `curl`:

```
$ curl -sSL https://raw.githubusercontent.com/python-poetry/poetry/
master/get-poetry.py | python
```

В Windows для установки можно использовать PowerShell:

```
> (Invoke-WebRequest -Uri https://raw.githubusercontent.com/python-
poetry/poetry/master/get-poetry.py -UseBasicParsing).Content | python -
```

После установки Poetry можно применять для следующих целей:

- Создавать новые проекты Python вместе с виртуальными средами.
- Инициализировать существующие проекты в виртуальной среде.
- Управлять зависимостями проектов.
- Упаковывать библиотеки.

Чтобы в Poetry создать совершенно новый проект, запустите команду `poetry new`, как в этом примере:

```
$ poetry new my-project
```

Эта команда создает новый каталог my-project, содержащий несколько исходных файлов. Структура каталога выглядит примерно так:

```
my-project/
├── README.rst
├── my_project
│   └── __init__.py
├── pyproject.toml
└── tests
    ├── __init__.py
    └── test_my_project.py
```

Как видите, создалось несколько файлов, которые можно использовать как заглушки для дальнейшей разработки. Чтобы инициализировать Poetry в уже существующем проекте, запустите команду `poetry init` в каталоге проекта. Она не создаст новых файлов проекта, кроме файла pyproject.toml.

Этот файл — pyproject.toml — является центральным компонентом Poetry и описывает конфигурацию проекта. Для `my-project` из предыдущего примера его содержимое может выглядеть так:

```
[tool.poetry]
name = "my-project"
version = "0.1.0"
description = ""
authors = ["Michał Jaworski <swistakm@gmail.com>"]

[tool.poetry.dependencies]
python = "^3.9"

[tool.poetry.dev-dependencies]
pytest = "^5.2"

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

Как видно из листинга, файл pyproject.toml делится на четыре раздела:

- `[tool.poetry]`: основные метаданные проекта: название, описание версии, автор и т. д. Эта информация необходима, если вы собираетесь опубликовать проект в виде пакета в PyPI.
- `[tool.poetry.dependencies]`: список зависимостей проекта. В новых проектах этот раздел содержит только версию Python, но в нем также могут перечисляться версии всех пакетов, которые обычно описываются в файле requirements.txt.
- `[tool.poetry.dev-dependencies]`: список зависимостей, необходимых для локальной разработки, например тестовые фреймворки или инструменты разработчика. На практике принято создавать отдельные списки таких зависимостей, потому что в средах реальной эксплуатации они обычно не нужны.
- `[build-system]`: описание Poetry как системы сборки, используемой для управления проектом.

Файл pyproject.toml определен в официальном стандарте Python (документ PEP 518). Подробнее ознакомиться с его структурой можно по адресу https://www.python.org/dev/peps/pep-0518/.

Если вы создаете новый проект или инициализируете существующий проект в Poetry, эта программа сможет в любой момент создать новую виртуальную среду в совместно используемом расположении. Ее можно активировать средствами Poetry, вместо того чтобы вызывать `source` для сценариев `activate`. Это удобнее, чем использовать модуль `venv`, потому что вам не придется запоминать, где фактически хранится виртуальная среда. Все, что вам нужно, — перейти в оболочке в любое место дерева исходного кода проекта и ввести команду `poetry shell`, как в этом примере:

```
$ cd my-project
$ poetry shell
```

С этого момента виртуальная среда Poetry будет активирована в текущем сеансе оболочки. Чтобы в этом убедиться, введите команду `which python` или `python -m site`.

С помощью Poetry также проще управлять зависимостями. Как уже упоминалось, файлы requirements.txt — очень примитивный механизм управления зависимостями. Они описывают, какие пакеты надо установить, но не отслеживают автоматически, что было установлено в среде в процессе разработки. Если вы установите какой-то пакет с помощью `pip`, но забудете отразить это изменение в файле requirements.txt, то другие программисты могут столкнуться с затруднениями, воспроизводя вашу среду.

С Poetry эта проблема исчезает. Добавлять зависимости в проект можно только одним способом, и этот способ — команда `poetry add <имя_пакета>`. Вот что она делает:

- Обрабатывает целые деревья зависимостей, если другие пакеты совместно используют зависимости.
- Устанавливает все пакеты из дерева зависимостей в виртуальной среде, связанной с проектом.
- Отражает изменения в файле pyproject.toml.

В следующем примере показан процесс установки фреймворка Flask в среде `my-project`:

```
$ poetry add flask
```

Результат выполнения команды выглядит так:

```
Using version ^1.1.2 for Flask

Updating dependencies
Resolving dependencies... (38.9s)

Writing lock file

Package operations: 15 installs, 0 updates, 0 removals

  • Installing markupsafe (1.1.1)
  • Installing pyparsing (2.4.7)
  • Installing six (1.15.0)
  • Installing attrs (20.3.0)
  • Installing click (7.1.2)
  • Installing itsdangerous (1.1.0)
```

```
  • Installing jinja2 (2.11.2)
  • Installing more-itertools (8.6.0)
  • Installing packaging (20.4)
  • Installing pluggy (0.13.1)
  • Installing py (1.9.0)
  • Installing wcwidth (0.2.5)
  • Installing werkzeug (1.0.1)
  • Installing flask (1.1.2)
  • Installing pytest (5.4.3)
```

А вот как выглядит файл pyproject.toml с выделенным изменением в зависимостях проекта:

```
[tool.poetry]
name = "my-project"
version = "0.1.0"
description = ""
authors = ["Michał Jaworski <swistakm@gmail.com>"]

[tool.poetry.dependencies]
python = "^3.9"
Flask = "^1.1.2"

[tool.poetry.dev-dependencies]
pytest = "^5.2"

[build-system]
requires = ["poetry-core>=1.0.0"]
```

В предыдущем примере Poetry установил 15 пакетов, хотя мы запросили всего одну зависимость. Дело в том, что у Flask есть свои собственные зависимости, а у этих зависимостей — тоже свои зависимости. Такие «зависимости зависимостей» называются **транзитивными зависимостями**. Для библиотек часто используются нестрогие спецификаторы версий вида `six >=1.0.0`, которые допускают широкий диапазон версий. Poetry реализует алгоритм разрешения зависимостей, чтобы составить набор версий, который удовлетворяет всем ограничениям транзитивных зависимостей.

Проблема транзитивных зависимостей — в том, что они могут меняться со временем. Вспомните, что библиотеки могут задавать нестрогие спецификаторы версий для своих зависимостей. Может оказаться, что в двух средах, созданных в разное время, используются разные финальные версии пакетов. Невозможность воспроизвести точные версии всех транзитивных зависимостей может создать проблемы для крупных проектов, а вручную отслеживать все зависимости в файле requirements.txt — неблагодарное занятие.

Poetry решает проблему транзитивных зависимостей с помощью так называемых **файлов блокировки зависимостей**. Когда вы уверены, что в вашей среде сло-

жился работоспособный и протестированный набор версий пакетов, можно запустить такую команду:

```
$ poetry lock
```

Команда создает чрезвычайно пространный файл poetry.lock, в котором содержится полный снимок процесса разрешения зависимостей. Затем этот файл используется для подбора версий транзитивных зависимостей вместо обычного процесса разрешения зависимостей.

Каждый раз, когда с помощью команды `poetry add` добавляются новые пакеты, Poetry анализирует дерево зависимостей и обновляет файл poetry.lock. На сегодняшний день файл блокировки — лучший и самый надежный способ управлять транзитивными зависимостями в проекте.

Дополнительную информацию о расширенных возможностях Poetry можно найти в официальной документации по адресу https://python-poetry.org.

Изоляция сред на уровне системы

Ключевой фактор быстрых итераций при разработке ПО — повторное использование существующих программных компонентов. «Не повторяйтесь» (DRY, Don't Repeat Yourself) — расхожая мантра среди программистов. Включение сторонних пакетов и модулей в кодовую базу — лишь часть этой идеологии. Повторно используемыми компонентами также можно считать двоичные библиотеки, базы данных, системные службы, сторонние API и т. д. Даже целые операционные системы можно рассматривать как повторно используемые компоненты.

Серверная часть (backend) веб-приложений — отличный пример того, насколько сложными могут быть такие приложения. Простейший программный стек обычно состоит из нескольких уровней. Рассмотрим воображаемое приложение, которое хранит некую информацию о своих пользователях и предоставляет доступ к ней через интернет по протоколу HTTP. На практике приложения такого рода содержат как минимум три уровня (начиная с низшего):

- База данных или другая разновидность хранилища.
- Код приложения, реализованный на Python.
- Сервер HTTP, работающий в режиме обратного прокси-сервера (например, Apache или NGINX).

Очень простые приложения могут быть одноуровневыми, но это обычно не относится к приложениям, которые сложно устроены или должны обрабатывать

интенсивный трафик. В реальности большие приложения иногда настолько сложны, что их нельзя представить в виде вертикального стека слоев, а можно только в виде «лоскутного одеяла» из взаимосвязанных служб. И маленькие и большие приложения могут использовать несколько разных баз данных, делиться на независимые процессы и задействовать другие системные службы для кэширования, управления очередями, журналирования, обнаружения служб и т. д. К сожалению, этим усложнениям нет предела.

Здесь действительно важно то, что не все элементы программного стека можно изолировать на уровне среды выполнения Python. HTTP-сервер (например, nginx), СУБД (например, PostgreSQL) или совместно используемая библиотека — все эти компоненты обычно не входят в дистрибутив Python или экосистему пакетов Python и не инкапсулируются в виртуальных средах Python. Поэтому они считаются внешними зависимостями программного продукта.

Проблема усугубляется тем, что внешние зависимости обычно доступны в разных версиях и модификациях под разными операционными системами. Например, если два разработчика используют совершенно разные дистрибутивы Linux (допустим, Debian и Gentoo), то вряд ли в каждый момент времени им будет доступна одна и та же версия nginx в репозиториях пакетов ОС. Более того, разные версии могут компилироваться с разными флагами времени компиляции (например, чтобы активировать специфические настройки для той или иной ОС), а также поставляться с особыми расширениями или исправлениями для конкретного дистрибутива.

Таким образом, без подходящих инструментов было бы трудно гарантировать, что все участники команды разработки используют одинаковые версии всех компонентов. Теоретически можно добиться, чтобы все, кто работает над одним и тем же проектом, использовали одинаковые версии служб на своих компьютерах. Однако все усилия окажутся напрасными, если они не работают под той же операционной системой, которая будет в среде эксплуатации. Не всегда можно заставить программиста работать в чем-то, кроме его любимой операционной системы.

Среда эксплуатации (production environment)[1] — реальная среда, в которой приложение устанавливается, чтобы работать по прямому назначению. Например, среда эксплуатации для настольного приложения — это настольный компьютер, на котором пользователь установил приложение. Для серверной части веб-приложения, доступного через интернет, средой эксплуатации обычно является удаленный сервер (иногда виртуальный), установленный в центре обработки данных.

[1] Иногда говорят просто «продакшен». — *Примеч. ред.*