ChatGPT

Руководство по управлению виртуальными окружениями, версиями Python и пакетами на Ubuntu

1. Создание виртуального окружения с помощью uv и pip

Виртуальное окружение – это изолированная копия интерпретатора Python с собственным набором установленных пакетов. Оно позволяет устанавливать в проект локальные зависимости, не затрагивая системные библиотеки Python. По словам официальной документации Python, использование виртуальных окружений – лучшая практика при работе с внешними пакетами, чтобы пакеты разных проектов не конфликтовали между собой [1]. Аналогично об этом говорится и в документации инструмента **uv**: виртуальное окружение «изолирует пакеты от окружения Python-инсталляции» и позволяет не менять системный Python [2].

На Ubuntu сначала нужно установить Python и менеджер пакетов **pip** (если они не установлены по умолчанию):

```
sudo apt update
sudo apt install python3 python3-venv python3-pip
```

Затем устанавливаем **uv** – это современный инструмент (от компании Astral) для управления версиями и окружениями Python. Его можно установить из PyPI:

```
pip3 install --user uv
```

(Альтернативно можно установить uv через официальный скрипт или через `pipx` *– но в этом руководстве ограничимся* `pip install`*.)*

После установки команды **uv** становятся доступны в `~/.local/bin`, убедитесь, что этот путь есть в переменной PATH.

Теперь можно **создать виртуальное окружение** с помощью uv. По умолчанию `uv venv` создаёт окружение с именем `.venv` в текущей папке. Например:

```
$ uv venv
Using Python 3.10.6
Creating virtual environment at: .venv
Activate with: source .venv/bin/activate
```

Это означает, что uv автоматически выбрал доступную версию Python (3.10.6 в примере) и создал папку `.venv`. Для активации окружения выполните команду:

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

После активации в командной строке появится префикс с именем окружения (обычно `(.venv)`), а все `pip install` будут устанавливать пакеты именно в этот `.venv`.

Если требуется сразу добавить в окружение `pip` (uv по умолчанию создаёт минимальный venv без pip), можно использовать опцию `--seed` или установить pip уже в созданном окружении:

```
# Создать venv и автоматически установить pip
$ uv venv --seed
Using Python 3.10.6
Creating virtual environment at: .venv
Activate with: source .venv/bin/activate

# Или добавить pip в существующий venv
$ uv pip install pip
```

После этого внутри активированного окружения становится доступен `pip`. Например:

```
(.venv) $ pip install requests
```

Кроме uv, **виртуальные окружения** можно создавать стандартными средствами Python. Так, встроенный модуль `venv` позволяет сделать то же самое:

```
$ python3 -m venv myenv
```

Эта команда создаст папку `myenv` с виртуальным окружением. В Ubuntu для этого важно, чтобы был установлен пакет `python3-venv`. После выполнения команды создаётся поддиректория `bin/`, где лежат свои `python` и `pip`. Чтобы начать пользоваться окружением, его нужно активировать:

```
$ source myenv/bin/activate
(myenv) $ python --version
Python 3.10.6
(myenv) $ pip install flask
```

Тут `pip` находится внутри окружения и не затрагивает системные пакеты. Подробнее о создании и использовании стандартного venv см. в официальном руководстве [3] [4].

Таким образом, **uv** и стандартный **venv**/pip обеспечивают изоляцию пакетов. При работе с **uv** удобнее использовать его команды (`uv venv`, `uv pip install` и т.д.), а при работе с `pip` (и модулем `venv`) – стандартные инструменты `python3 -m venv` и `pip install`. Ниже показаны примеры команд и их вывод в терминале.

```
$ uv venv --python 3.11 --seed
Using Python 3.11.2
Creating virtual environment at: .venv
Activate with: source .venv/bin/activate
```

Команда `uv venv --python 3.11 --seed` создаёт виртуальное окружение `.venv` с Python 3.11 и сразу устанавливает `pip`. Вывод показывает используемую версию Python и подсказку для активации окружения.

```
$ python3 -m venv myenv
$ ls myenv
bin  include  lib  pyvenv.cfg
```

Команда `python3 -m venv myenv` создаёт окружение в папке `myenv`. В результате появились подпапки `bin/`, `lib/` и др. Всегда активируйте окружение перед установкой пакетов командой `source myenv/bin/activate`.

2. Установка разных версий Python через uv и pyenv

Различные версии Python можно устанавливать и переключать как с помощью **uv**, так и с помощью инструмента **pyenv**. Оба инструмента управляют версиями, но работают по-разному. **pyenv** компилирует Python из исходников (что может занимать время и место на диске), а **uv** скачивает готовые бинарные сборки (поэтому установка быстрая и занимает меньше места) [5]. Кроме того, `uv` – это единый инструмент, который одновременно управляет версиями Python, окружениями и зависимостями, тогда как у `pyenv` для виртуальных окружений обычно используют дополнительно `pyenv-virtualenv` (и пакетный менеджер отдельно). Как отмечает автор одного из обзоров, `uv` «консолидирует управление версиями, виртуальными окружениями и инструментами в единое, быстрое решение» [6].

Установка pyenv

На Ubuntu для **pyenv** рекомендуется установить зависимости и сам pyenv. Например:

```
sudo apt update
sudo apt install -y make build-essential libssl-dev zlib1g-dev libbz2-dev \
    libreadline-dev libsqlite3-dev wget curl llvm libncursesw5-dev xz-utils \
    tk-dev libxml2-dev libxmlsec1-dev libffi-dev liblzma-dev git
```

Затем можно установить pyenv одним из способов:

- **Клонирование репозитория:**

```
git clone https://github.com/pyenv/pyenv.git ~/.pyenv
echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.bashrc
echo 'export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.bashrc
echo -e
```

```
'if command -v pyenv 1>/dev/null 2>&1; then\n eval "$(pyenv init --
path)"\nfi' >> ~/.bashrc
exec $SHELL
```

- **Скрипт установки:**

```
curl https://pyenv.run | bash
exec $SHELL
```

После этого команду `pyenv` можно вызывать из терминала (при необходимости перезапустите shell).

Установка версий Python

С помощью **pyenv** загруженные версии ставятся так:

```
$ pyenv install 3.9.9
$ pyenv install 3.10.5
$ pyenv versions
  system
* 3.10.5
  3.9.9
```

Здесь мы выбрали и установили Python 3.9.9 и 3.10.5. Команда `pyenv versions` показывает доступные версии и отмечает (*) активную (по умолчанию это системная). Затем переключаем глобальную версию:

```
$ pyenv global 3.9.9
$ python --version
Python 3.9.9
```

Теперь `python` в любом месте будет ссылаться на версию 3.9.9. Для проекта можно задать локальную версию в текущей папке:

```
$ mkdir project && cd project
$ pyenv local 3.10.5
$ python --version
Python 3.10.5
```

Эта команда создаёт файл `.python-version` в директории `project`. При входе в неё `pyenv` будет использовать указанную там версию. Вне неё снова будет глобальная версия.

Для **uv** процесс проще: uv автоматически скачивает требуемые версии из своего репозитория. Пример установки нескольких версий и переключения:

```
$ uv python install 3.9 3.10
Searching for Python versions matching: Python 3.9
Searching for Python versions matching: Python 3.10
Installed 2 versions in 2.10s
  + cpython-3.9.18-linux-x86_64
  + cpython-3.10.10-linux-x86_64
```

В примере uv установил CPython 3.9.18 и 3.10.10. После этого можно указать одну из них для окружения или проекта. Например, прямо в текущей папке:

```
$ uv python pin 3.10
Pinned `.python-version` to `3.10`
```

Теперь uv будет по умолчанию использовать Python 3.10 в текущей директории. (Файл `.python-version` задаёт версию для uv и pyenv.) Чтобы запустить конкретную версию один раз, можно использовать:

```
$ uv run --python 3.9 -- python --version
Python 3.9.18
```

В целом, ключевое различие: **pyenv** предоставляет инструменты только для управления версиями, тогда как **uv** объединяет управление версиями, окружениями и пакетами «в одном флаконе» [6] [7].

3. Работа с проектом через uv

Инструмент **uv** позволяет быстро создавать и управлять проектами Python. При создании нового проекта автоматически генерируется базовая структура с файлом `pyproject.toml`, в котором задаются метаданные и зависимости проекта [8]. Пример создания проекта:

```
$ uv init hello-world
Initialized project `hello-world` at `/home/user/hello-world`
$ cd hello-world
```

Команда `uv init hello-world` создала директорию `hello-world` со следующей структурой [9]:

```
hello-world/
├── .python-version
├── README.md
├── main.py
└── pyproject.toml
```

В файле `pyproject.toml` уже есть секция `[project]` с основными данными (имя, версия, зависимости и т.д.) [8]. Например, после `uv init` там будет:

```
[project]
name = "hello-world"
version = "0.1.0"
description = ""
readme = "README.md"
dependencies = []
```

Тут можно вручную добавлять зависимости, либо воспользоваться командами **uv**. Рассмотрим установку пакета `requests`:

```
$ uv add requests
Creating virtual environment at: .venv
Resolved 1 package in 12ms
Installed 1 package in 150ms
  + requests==2.31.0
```

Команда `uv add requests` добавляет `requests` в `pyproject.toml` и устанавливает его в виртуальное окружение `.venv`. При первом добавлении зависимостей окружение создаётся автоматически (папка `.venv` с Python и pip) [10]. Аналогично можно добавлять версии или пакеты из других источников:

```
$ uv add 'Django>=4.2'      # с указанием версии
$ uv add -r requirements.txt  # добавить все зависимости из файла
requirements.txt
```

После изменения зависимостей для фиксации версии пакетов используют команды `uv lock` и `uv sync`:

```
$ uv lock
Resolved 3 packages in 0.30ms

$ uv sync
Resolved 3 packages in 0.45ms
Audited 3 packages in 0.05ms
```

Команда `uv lock` обновляет lock-файл `uv.lock` с точными версиями зависимостей, а `uv sync` устанавливает их в окружение. Lock-файл позволяет воспроизводить окружение на других машинах с теми же точными версиями [11].

Для выполнения кода в проекте используется `uv run`. Например, проект создал файл `main.py` с простым выводом. Запустим его:

```
$ uv run main.py
Hello from hello-world!
```

Команда `uv run` перед исполнением проверяет, что зависимости актуальны, и при необходимости автоматически обновляет окружение перед запуском.

Таким образом, **uv** превращает работу над проектом в последовательность интуитивных команд: создать проект (`uv init`), добавлять/удалять зависимости (`uv add`, `uv remove`), фиксировать версии (`uv lock`), устанавливать их (`uv sync`) и запускать код (`uv run`). Примерно то же можно делать и вручную через редактирование `pyproject.toml` и команд `pip`, но uv упрощает и ускоряет процесс [10] [12]. Ниже приведён пример файл `pyproject.toml` и несколько ключевых команд с выводом:

```toml
# пример pyproject.toml
[project]
name = "hello-world"
version = "0.1.0"
dependencies = [
    "requests>=2.25.0",
]
```

```
$ uv add flask
Creating virtual environment at: .venv
Resolved 1 package in 15ms
Installed 1 package in 120ms
  + flask==2.2.5

$ uv lock
Resolved 2 packages in 0.34ms

$ uv sync
Resolved 2 packages in 0.50ms
Audited 2 packages in 0.02ms
```

Команды `uv add`, `uv lock`, `uv sync` показаны с типичным выводом (создание окружения, установка пакетов, проверка версий) [10].

4. Этапы работы системного пакетного менеджера apt

Менеджер пакетов APT (Advanced Package Tool) в Ubuntu работает по многоступенчатому сценарию при установке пакета. Например, при выполнении `sudo apt install nginx` происходят следующие этапы (схематично):

1. **Обновление метаданных (`apt update`)**: сначала APT обращается к кэшу локальных метаданных, который заполняется при команде `apt update`. В нём хранятся списки пакетов, их версии и зависимости [13] [14]. Регулярное выполнение `apt update` важно, чтобы APT знал о самых свежих версиях пакетов [15].

2. **Поиск пакета в кэше (APT Cache)**: APT ищет нужный пакет в локальном кэше. Команда `apt-cache search <ключевое слово>` позволяет искать пакеты по описанию, а `apt-cache show <пакет>` выдаёт подробную информацию (описание, зависимости, версию) о конкретном пакете [16]. Эти операции **не устанавливают** пакетов, а лишь работают с локальным кэшем, полученным ранее с серверов.

3. **Разрешение зависимостей**: APT строит дерево зависимостей выбранного пакета. Все требуемые вспомогательные пакеты заносятся в очередь на установку. При этом проверяются конфликты версий или несовместимости; если возникают проблемы (конфликты или отсутствующие зависимости), пользователь об этом сообщает до скачивания [17].

4. **Загрузка пакетов**: APT подключается к удалённым репозиториям (заданные в `/etc/apt/sources.list`) и скачивает все необходимые `.deb` файлы в локальное хранилище `/var/cache/apt/archives` [18]. На этом этапе пакеты только скачиваются, но ещё не устанавливаются.

5. **Проверка подлинности**: загруженные `.deb`-файлы проверяются по цифровым подписям с помощью доверенных GPG-ключей. Это гарантирует подлинность и целостность пакетов. Если проверка подписи не проходит, установка прекращается [19].

6. **Установка (через dpkg)**: APT передаёт управление программе `dpkg` (низкоуровневому установщику Debian). `dpkg` распаковывает файлы пакета и размещает их по нужным каталогам (бинарники – в `/usr/bin`, библиотеки – в `/usr/lib` и т.д.), затем выполняются скрипты установки (например, создание пользователей, добавление системных служб) [20].

7. **Обновление состояния**: После успешной установки APT обновляет внутреннюю базу данных: фиксирует какие пакеты установлены, какие файлы ими созданы и какие зависимости у них есть [21]. Это необходимо для корректного последующего обновления или удаления пакетов.

8. **Очистка и финальные настройки**: Удаляются временные файлы, скачанные пакеты (по умолчанию), чтобы освободить место [22]. При необходимости отображаются запросы на конфигурацию пакета (например, настройка служб) и выполняется автостарт служб (например, nginx запускается после установки).

Пример команд:

```
$ sudo apt update
Hit:1 http://archive.ubuntu.com/ubuntu focal InRelease
Reading package lists... Done

$ apt-cache search python3.10
python3.10 - Interactive high-level object-oriented language (version 3.10)
libpython3.10-stdlib - Interactive high-level object-oriented language
(standard library)

$ apt-cache show python3.10
```

```
Package: python3.10
Version: 3.10.2-3ubuntu1~20.04
Depends: libpython3.10-stdlib, python3.10-minimal, <...>
Description: Interactive high-level object-oriented language (version 3.10)
```

- `sudo apt update` обновляет локальные списки пакетов (вывод «Reading package lists... Done» показывает успешное обновление).
- `apt-cache search` ищет пакеты по ключевым словам.
- `apt-cache show` показывает информацию о пакете (версию, зависимости, описание).

В целом жизненный цикл `apt install` включает **поиск**, **разрешение зависимостей**, **скачивание**, **проверку подписи**, **установку через dpkg** и **обновление состояния** [14] [23].

5. Этапы работы pip

Менеджер пакетов **pip** устанавливает Python-пакеты по аналогичной схеме, но для PyPI. Рассмотрим, что происходит при `pip install` (или `pip install -r requirements.txt`):

1. **Разбор требований**: pip читает файл `requirements.txt` или аргументы командной строки. Он собирает список требуемых пакетов и версий. Если указан `pyproject.toml`, начиная с определённых версий pip может учитывать таблицу `[project]` (PEP 621) для установки проекта из исходников. Затем pip **разрешает зависимости** – определяет, какие версии сопутствующих пакетов нужны. Современный резолвер pip (с версии 20.3) использует алгоритм с «откатом» при конфликте зависимостей [24] [25] (например, при несовместимых требованиях он может попробовать загрузить альтернативные версии).

2. **Загрузка пакетов**: pip по очереди скачивает файлы пакетов. Если для пакета доступны **колёса (wheels)**, pip загружает файл `.whl`. Если колёса нет, загружается исходный архив (sdist, например `.tar.gz` или `.zip`).

3. **Построение (build)**: если был загружен sdist, pip запускает систему сборки (PEP 517, например `setuptools` или `flit`), чтобы собрать из исходников wheel. Этот процесс происходит в изолированном временном окружении. После сборки wheel сохраняется в локальный кэш pip [26]. Если при повторной установке той же версии пакета wheel уже есть в кеше, pip использует его напрямую, избегая повторной сборки [27].

4. **Установка**: полученный wheel распаковывается и устанавливается в целевое окружение (виртуальное или системное). Все файлы копируются в соответствующие папки, выполняются скрипты установки пакета (например, компиляция C-расширений) и регистрация пакета в окружении.

5. **Кэширование**: по умолчанию pip **кэширует запросы** к PyPI и результаты сборки. HTTP-запросы хранятся в `~/.cache/pip/http-v2/` (раньше – в `http/`) [28]; если повторно запрашивать тот же URL, pip проверит локальный кэш и при отсутствии изменений вернёт сохранённый ответ. Собранные wheel-файлы сохраняются в `~/.cache/pip/wheels/` (или другом каталоге, смотря `pip cache dir`). При повторной установке той же версии пакета wheel-пакет возьмётся из кэша, если он там есть [26]. Путь к кэшу можно увидеть командой `pip cache dir`, но по умолчанию в Linux это `~/.cache/pip` [29].

Пример команды и вывода pip:

```
$ pip install requests
Defaulting to user installation because normal site-packages is not writable
Collecting requests
  Downloading requests-2.31.0-py3-none-any.whl (62 kB)
Installing collected packages: requests
Successfully installed requests-2.31.0
```

Здесь показан процесс: pip скачал файл `requests-2.31.0-py3-none-any.whl` (количество КБ), затем распаковал и установил его. Если бы wheel не подошёл (например, был только sdist), pip бы скомпилировал его перед установкой. Также видно, что установка происходит в пользовательский каталог (`--user`), поскольку у нас нет прав записывать системный каталог.

Таким образом, **pip** последовательно **парсит зависимости**, **скачивает колёса/исходники**, **собирает при необходимости** и **устанавливает** пакеты, используя кэш для ускорения. Повторные установки одного и того же пакета обычно берут содержимое из кэша, если версии совпадают [27].

6. Местоположение установленных библиотек Python

В Ubuntu пакеты Python могут храниться в разных местах, в зависимости от способа установки:

- **Системные пакеты (APT)** устанавливаются в директорию `/usr/lib/python3/dist-packages`. Например, если вы установили `python3-numpy` через `sudo apt install`, файлы numpy будут лежать именно там [30]. Также `/usr/lib/python3.10` содержит стандартную библиотеку Python 3.10 и её файлы, установленные системой.

- **Пользовательские пакеты (pip)** при системной установке (`sudo pip install`) попадут в `/usr/local/lib/python3.10/dist-packages` (или `/usr/local/lib/python3.x/dist-packages`), где `3.10` – версия Python [30]. Это отмечено как «модули, которые вы установили сами с помощью pip». А при установке без прав суперпользователя (`pip install --user`) пакеты идут в домашнюю папку: `~/.local/lib/python3.10/site-packages` для Python 3.10 [31] [32]. Такие пакеты видимы только текущему пользователю.

- При использовании **виртуального окружения** пакеты устанавливаются внутри папки окружения, обычно в `myenv/lib/python3.10/site-packages`. Это также изоляция на уровне проекта.

Например, может быть так [33] [32]:

- `/usr/lib/python3/dist-packages` – пакеты из `apt` (напр., apt-get установил «не специфичные для хоста» модули Python).
- `/usr/local/lib/python3.10/dist-packages` – пакеты, установленные глобально через `pip` (под **sudo**).
- `/home/user/.local/lib/python3.10/site-packages` – пакеты, установленные пользователем через `pip install --user`.

Кроме того, в списке `sys.path` обычно присутствуют `/usr/lib/python3.10` (где лежит стандартная библиотека) и, если активировано виртуальное окружение, путь к его `site-packages`.

Обращение к каталогам:

```
import sys, site
print(site.getsitepackages())  # Системные site-packages
print(site.getusersitepackages())  # Каталог --user
```

выведет пути вроде `/usr/local/lib/python3.10/site-packages` и `~/.local/lib/python3.10/site-packages`.

Таким образом, **dist-packages** (в `/usr/lib` и `/usr/local/lib`) – это Debian-специфичная структура каталогов для пакетов Python, а **site-packages** обычно используется в виртуальных окружениях и пользовательских установках. Для управляемости версий и окружений важно знать эти пути, чтобы понимать, куда попадут файлы при установке тем или иным способом [30] [33].

---

[1] [3] [4] Install packages in a virtual environment using pip and venv - Python Packaging User Guide
https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/

[2] Using environments | uv
https://docs.astral.sh/uv/pip/environments/

[5] [7] [10] uv · PyPI
https://pypi.org/project/uv/

[6] From pyenv to uv: Streamlining Python Management | Rob's Cogitations
https://rob.cogit8.org/posts/2024-09-19-pyenv-to-uv/

[8] [9] [11] [12] Working on projects | uv
https://docs.astral.sh/uv/guides/projects/

[13] [15] [16] 6.3. The apt-cache Command
https://debian-handbook.info/browse/stable/sect.apt-cache.html

[14] [17] [18] [19] [20] [21] [22] [23] How APT Works Behind the Scenes: From Command to Installed Package | by Manish Pandey | May, 2025 | Medium
https://medium.com/@manishpandey919/how-apt-works-behind-the-scenes-from-command-to-installed-package-8e8beced9723

[24] [25] Dependency Resolution - pip documentation v25.1.1
https://pip.pypa.io/en/stable/topics/dependency-resolution/

[26] [27] [28] [29] Caching - pip documentation v25.1.1
https://pip.pypa.io/en/stable/topics/caching/

[30] [31] python - What is the difference /usr/local/lib/python3.6/dist-packages vs /usr/lib/python3/dist-packages? - Ask Ubuntu
https://askubuntu.com/questions/1159307/what-is-the-difference-usr-local-lib-python3-6-dist-packages-vs-usr-lib-python

[32] [33] The reason cause different location of python packages - Stack Overflow
https://stackoverflow.com/questions/54234344/the-reason-cause-different-location-of-python-packages