ChatGPT

Подготовка к экзамену по Python: окружения, версии и пакеты

1. Создание виртуального окружения с использованием uv и pip

Виртуальное окружение (virtual environment) – это изолированная среда для Python-проектов, в которой можно устанавливать пакеты, не вмешиваясь в глобальные (системные) библиотеки. Это позволяет избежать конфликтов версий и зависимостей между разными проектами. Инструмент **pip** исторически используется для установки пакетов, однако сам по себе он не создаёт виртуальных окружений – обычно для этого применяются встроенный модуль `venv` или утилиты вроде `virtualenv`. С появлением инструмента **uv**, разработанного на Rust, управление окружениями упростилось: **uv по умолчанию требует использования виртуального окружения**, тогда как `pip` этого не делает. Ниже рассмотрим, как создавать виртуальные окружения традиционными средствами и с помощью uv.

Создание виртуального окружения через стандартные средства (venv/pip)

В Python 3 есть встроенный модуль `venv` для создания виртуальных окружений. Например, чтобы создать новое окружение в каталоге `env`, выполните команду:

```
$ python3 -m venv env
```

Эта команда скопирует необходимый интерпретатор Python и создаст в папке `env` изолированное окружение со своей директорией для пакетов. Начиная с Python 3.4, модуль `venv` автоматически устанавливает внутрь окружения pip, так что после создания окружения можно сразу использовать команду `pip` для установки зависимостей.

Активация окружения: после создания окружения необходимо “активировать” его, чтобы команды `python` и `pip` в терминале ссылались на интерпретатор и пакеты внутри этого окружения, а не на системные. На Linux и macOS активация выполняется командой:

```
$ source env/bin/activate
(env) $  # теперь префикс командной строки указывает на активное окружение
```

На Windows используется сценарий активации в `Scripts\activate.bat`. После активации в начале командной строки появляется название окружения (например, `(env)`), что говорит о том, что вы находитесь внутри него. Теперь установки пакетов через `pip install` будут происходить в `env`, изолированно от системы.

Деактивация окружения: чтобы выйти из виртуального окружения, выполните команду `deactivate`. После этого префикс `(env)` исчезнет, и команды `python`, `pip` снова будут соответствовать системному интерпретатору.

Пример работы: создадим и активируем окружение, установим пакет и деактивируем:

```
$ python3 -m venv env             # создание окружения
$ source env/bin/activate         # активация
(env) $ pip install requests     # установка пакета 'requests' внутри env
(env) $ pip list                 # просмотр установленных пакетов в env
(env) $ deactivate               # деактивация окружения
$
```

В данном примере пакет requests будет установлен в каталог `env/lib/pythonX.Y/site-packages`, не затрагивая глобальные библиотеки.

Создание виртуального окружения с помощью uv

Инструмент **uv** объединяет функциональность pip, virtualenv и других утилит, автоматизируя создание и использование окружений. **uv требует, чтобы вы работали в виртуальном окружении**, и потому предоставляет команду для его быстрой генерации. Чтобы создать виртуальное окружение в текущем проекте с помощью uv, достаточно выполнить:

```
$ uv venv
```

Эта команда создаст виртуальное окружение в подкаталоге по умолчанию (как правило, `.venv` в текущей директории). Если указать имя, uv создаст окружение с этим именем. Например, `uv venv myenv` создаст папку `myenv` с необходимыми файлами окружения. При необходимости можно указать версию Python: `uv venv --python 3.11` – uv попытается использовать Python 3.11 (скачает его, если он не установлен, подробнее об этом в разделе про версии Python).

После создания окружения **uv** также позволяет сразу устанавливать пакеты. Например, выполнив последовательно:

```
$ uv venv               # создать окружение .venv
$ uv pip install ruff   # установить пакет 'ruff' в окружение
```

будет создано окружение и в него установлен пакет **ruff**. Обратите внимание: если использовать **uv** для установки пакета без активного окружения, uv автоматически обнаружит и задействует окружение (по умолчанию `.venv`), поэтому ручная активация может и не понадобиться.

Активация и деактивация: хотя uv сам управляет окружениями, вы можете активировать созданное uv-окружение традиционным способом. Например, если uv создал директорию `.venv`, выполните `. .venv/bin/activate` (аналогично приведённой выше активации). Деактивация – командой `deactivate`, как обычно.

Особенность uv: uv предлагает альтернативу ручной активации – команду `uv run`. Она позволяет запустить скрипт внутри окружения без явной активации: uv сам **на время выполнения скрипта активирует окружение, а по завершении деактивирует** его. Например, если в проекте есть файл **hello.py**, можно выполнить `uv run hello.py` – скрипт запустится в контексте виртуального окружения, даже если вы его не активировали явно. При первом таком запуске uv автоматически создаст папку `.venv` (если её ещё нет) и файл `uv.lock` для фиксации зависимостей.

Вопросы для самопроверки:

- В чём преимущество использования виртуальных окружений при разработке Python-проектов?
- Как создать и активировать виртуальное окружение с помощью модуля `venv`? Как его деактивировать?
- Как команда `uv venv` упрощает создание окружения по сравнению с использованием pip и virtualenv?
- Что делает команда `uv run script.py` и почему она удобна?

2. Установка разных версий Python через uv и pyenv

Когда требуется работать с несколькими версиями Python (например, для тестирования или для разных проектов), на помощь приходят инструменты, позволяющие устанавливать и переключаться между версиями интерпретатора. Рассмотрим два подхода: менеджер версий **pyenv** и возможности самого **uv**.

Менеджер версий pyenv

pyenv – популярный инструмент для управления несколькими версиями Python на одной машине. С его помощью можно установить различные версии (включая альтернативные интерпретаторы, такие как PyPy, Jython и др.) и гибко переключаться между ними для разных проектов. По умолчанию **pyenv** хранит все установленные интерпретаторы в каталоге `$(pyenv root)/versions/` (обычно это `~/.pyenv/versions/`). При установке новой версии Python исходники скачиваются и компилируются, после чего интерпретатор размещается, например, в `~/.pyenv/versions/3.10.4/`.

После установки версий pyenv предоставляет несколько команд для выбора активной версии:

- `pyenv install <версия>` – установить указанную версию Python (можно посмотреть список доступных версий командой `pyenv install --list` [1]).
- `pyenv versions` – показать все установленные версии и текущую активную (отмечена звёздочкой).
- `pyenv global <версия>` – сделать указанную версию глобальной (по умолчанию используемой в системе).
- `pyenv local <версия>` – создать в текущем проекте файл `.python-version` с версией, которая должна использоваться в данном каталоге (включая поддиректории). При входе в этот каталог pyenv автоматически будет подставлять указанную версию Python.
- `pyenv shell <версия>` – временно использовать указанную версию в текущем shell-сессии.

Например, выполнив:

```
$ pyenv install 3.9.13      # установка Python 3.9.13
$ pyenv install 3.11.4      # установка Python 3.11.4
$ pyenv versions            # список установленных версий
  system
  3.9.13
* 3.11.4 (set by /home/user/.pyenv/version)
$ pyenv global 3.9.13       # переключение глобально на 3.9.13
$ python --version
Python 3.9.13
$ pyenv local 3.11.4        # в текущем проекте использовать 3.11.4
$ python --version
Python 3.11.4
```

В этом примере глобальной версией стала 3.9.13, но в директории проекта (где выполнен `pyenv local`) будет использоваться 3.11.4. Pyenv достиг этого, поместив файл **.python-version** с содержимым “3.11.4” в данный каталог. Pyenv при каждом запуске Python проверяет текущую директорию (и выше по иерархии) на наличие `.python-version` и, найдя его, подставляет указанную версию Python.

Важно отметить, что pyenv изменяет версию **интерпретатора**, но не изолирует пакеты по проектам автоматически. Обычно для каждого проекта всё равно рекомендуется создавать виртуальное окружение (например, через `python -m venv`), уже выбрав нужную версию через pyenv.

Управление версиями Python через uv

Инструмент uv объединяет функциональность pyenv, поэтому вы можете устанавливать и переключать версии Python непосредственно через uv. По умолчанию uv предпочитает “свои” версии интерпретатора, которые он устанавливает и хранит в специальном каталоге. На Linux uv сохраняет скачанные интерпретаторы в директории `~/.local/share/uv/python` (или в пути, заданном переменной окружения `XDG_DATA_HOME`). Например, при установке Python 3.12 через uv интерпретатор будет размещён по пути `~/.local/share/uv/python/cpython-3.12.x-...`.

Команды uv для управления версиями Python аналогичны pyenv:

- `uv python install <версия>` – установить указанную версию Python (если её нет в системе). Например, `uv python install 3.10` скачает и установит Python 3.10. Можно перечислить несколько версий через пробел, либо указать диапазон версий в кавычках (uv установит все подходящие версии) [2].
- `uv python list` – отобразить все доступные версии (как установленные локально, так и возможные для установки). С флагом `--only-installed` показывает только уже установленные версии.
- `uv python pin <версия>` – “закрепить” (использовать по умолчанию) определённую версию Python в текущем проекте. Эта команда обновляет файл `.python-version` в директории проекта на указанную версию (по сути аналог `pyenv local`, но привязанный к uv).
- Кроме того, команду `uv run` можно вызывать с параметром `--python <версия>`, чтобы выполнить скрипт разово под конкретной версией интерпретатора, не меняя версию по умолчанию в проекте [3].

Пример работы uv: допустим, у нас проект с минимальной требуемой версией Python 3.9 (это указано в `pyproject.toml` как `requires-python = ">=3.9"`). В системе сейчас Python 3.13. Создадим через uv виртуальное окружение и установим дополнительную версию:

```
$ uv venv                  # создаём окружение (.venv) для проекта
$ uv python install 3.10   # устанавливаем Python 3.10 через uv
$ uv python list --only-installed
cpython-3.13.x [default]
cpython-3.10.x             # uv показывает, что 3.13 и 3.10 установлены
(версия 3.13 сейчас дефолтная)
$ uv python pin 3.10       # переключаем проект на Python 3.10
Pinned .python-version to 3.10
$ uv run --python 3.13 main.py   # разово запустить скрипт под Python 3.13 [3]
```

После команды `uv python pin 3.10` в файле `.python-version` проекта будет записано “3.10”, и теперь при обычном `uv run` или других операциях uv будет использовать Python 3.10 по умолчанию для этого проекта. В то же время глобально (за пределами проекта) системный Python 3.13 остаётся неизменным. UV, в отличие от pyenv, **управляет версиями на уровне проекта**, записывая предпочтение в конфигурацию проекта, тогда как pyenv может глобально менять версию для shell.

Следует отметить, что **uv при необходимости автоматически скачивает интерпретатор**. Если вы создаёте новое окружение командой `uv venv`, а требуемая версия Python (например, новейшая) не установлена, uv сам её загрузит [4]. Это упрощает работу: нет нужды вручную устанавливать Python – uv делает это прозрачно. (Эту автозагрузку можно отключить опцией `--no-managed-python` или настройками, если требуется полный контроль [5].)

После установки нескольких версий Python через uv, вы можете видеть установленные интерпретаторы в каталоге `~/.local/share/uv/python`. Там они хранятся в виде папок с именами, включающими версию и платформу, например: `cpython-3.11.9-linux-x86_64-gnu`, `cpython-3.10.12-linux-x86_64-gnu` и т.д.. UV будет использовать нужный интерпретатор, исходя из требований проекта (`.python-version` и `requires-python` в pyproject).

Вопросы для самопроверки:

- Где менеджер pyenv хранит установленные версии интерпретаторов Python на вашем компьютере?
- Какими командами pyenv можно переключить версию Python для: а) всей системы по умолчанию, б) конкретного проекта?
- Как узнать, какие версии Python установлены через uv? Как uv поступит, если нужной версии интерпретатора нет на системе?
- Для чего предназначен файл `.python-version` в проекте и как с ним работают pyenv и uv?

3. Работа с Python-проектом через uv

Инструмент uv позиционируется как “всё в одном” для управления проектами Python – он может и создавать структуру проекта, и управлять зависимостями (как pip/pipenv/poetry), и работать с

виртуальным окружением. Рассмотрим, как с помощью uv можно инициализировать проект и выполнять в нём основные действия.

Инициализация проекта (структура и файлы)

Создать новый проект с помощью uv очень просто: достаточно выполнить команду `uv init <название>` в желаемой директории. Например:

```
$ uv init myproject
Initialized project myproject at /home/user/myproject
$ cd myproject
$ ls -a
.gitignore        .python-version    pyproject.toml
hello.py          README.md
```

При инициализации uv делает следующее: - Создаёт новую папку проекта (в примере `myproject`) и сразу настраивает Git-репозиторий в ней (создаёт пустой репозиторий с .gitignore и README.md). - Создаёт файл `pyproject.toml` – основной конфигурационный файл проекта. В нём прописывается базовая информация (имя, версия проекта и пр.) и, что важно, раздел **[project]** с зависимостями. Пока проект только создан, список зависимостей пустой. Например, сразу после `uv init` файл pyproject.toml содержит строчки:

```
[project]
name = "myproject"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.10"
dependencies = []
```

Здесь `requires-python = ">=3.10"` означает минимальную версию Python для проекта (uv подставляет версию текущего интерпретатора на момент создания проекта; например, если стоял Python 3.10 или 3.13, как в примере). `.python-version` содержит конкретную версию интерпретатора, используемую по умолчанию (например, "3.13" – она совпадает с текущей системой, но может быть изменена через `uv python pin`, как обсуждалось выше). - Создаёт простой скрипт **hello.py** с шаблонным содержимым (обычно программа выводит строку вроде "Hello from <project>!" для проверки). - Создаёт `.gitignore` с типичными исключениями (например, чтобы не коммитить виртуальное окружение и прочие служебные файлы).

Обратите внимание: пока что виртуальное окружение **не** создано. Uv инициализировал проект, но саму папку `.venv` внутри не сделал, ожидая первого запуска или установки пакетов.

Установка зависимостей и lock-файл

После создания проекта можно добавлять зависимости (пакеты). В uv для этого используется команда `uv add`. Она похожа на `pip install`, но не только устанавливает пакет, а еще и: 1. Вносит его в список зависимостей в `pyproject.toml` (в секцию `[project] dependencies`). 2.

Обновляет (перегенерирует) lock-файл `uv.lock` – аналог `poetry.lock` или `Pipfile.lock`, фиксируя точные версии всех установленных зависимостей и их транзитивных зависимостей.

Например, добавим библиотеку **requests** в проект:

```
$ uv add requests
```

UV скачает пакет requests (последней версии, если не указано иное) и установит его в виртуальное окружение проекта. **При первом добавлении пакета** uv обнаружит, что окружение еще не создано, поэтому **автоматически создаст виртуальное окружение** в папке `.venv` и установит туда requests. После выполнения команды структура каталога изменится – появится директория `.venv` и файл **uv.lock**:

```
myproject/
├── .venv/              # виртуальное окружение (создано автоматически)
├── .gitignore
├── .python-version
├── hello.py
├── pyproject.toml      # обновлён: requests добавлен в dependencies
├── README.md
└── uv.lock             # lock-файл с зафиксированными версиями
```

В `pyproject.toml` теперь можно увидеть записанную зависимость, например: `dependencies = [ "requests>=2.31.0" ]` (uv обычно указывает минимальную версию, удовлетворяющую установленной). Файл **uv.lock** будет содержать подробную информацию о конкретных версиях requests и его зависимостей (например, urllib3 и др.), хеши файлов и прочие данные для воспроизведения окружения. Этот lock-файл служит гарантией воспроизводимости: на другом компьютере, выполнив `uv sync` (команда для синхронизации окружения с lock-файлом), uv установит точно такие же версии пакетов.

Управление зависимостями: - Если нужно обновить пакет до новой версии, используется `uv add <package> --upgrade` (аналог `pip install -U`). Можно обновить сразу несколько или все зависимости. Например, `uv add requests --upgrade` обновит requests до версии, удовлетворяющей ограничениям (при необходимости перед этим можно изменить версию в pyproject, например с `==2.31.0` на `>=2.31.0` для разрешения обновления). - Для удаления зависимости – команда `uv remove <package>` (можно перечислить несколько). UV удалит пакет из окружения и уберет его из `pyproject.toml` и `uv.lock` соответственно.

Важно, что uv.lock обновляется на каждое изменение зависимостей, подобно тому как `poetry.lock` обновляется при `poetry add/remove`. Если сравнивать, **uv** по функционалу близок к **Poetry** или **Pipenv**: он так же хранит список необходимых пакетов и lock-файл, а также сам управляет виртуальным окружением проекта. Разница в том, что uv стремится быть быстрее и объединяет сразу много возможностей (утверждается, что uv работает быстрее pip и аналогичных инструментов на порядок). Кроме того, uv использует стандартный `pyproject.toml` (PEP 621) для хранения зависимостей, тогда как Pipenv использует Pipfile, а Poetry – свой раздел в pyproject или отдельный poetry.lock для версий.

Запуск скриптов и управление окружением

Для запуска Python-скриптов внутри проекта с uv можно использовать команду `uv run`, как отмечалось ранее. Например:

```
$ uv run hello.py
```

При выполнении этой команды uv автоматически активирует виртуальное окружение `.venv`, запускает скрипт **hello.py**, а затем деактивирует окружение. Это упрощает жизнь: нет необходимости вручную вызывать `source .venv/bin/activate` перед каждым запуском или пользоваться `python3 path/to/env/bin/python script.py`. UV делает это под капотом.

Если скрипту требуются какие-то аргументы командной строки, их можно добавить после имени файла, и uv передаст их вашему скрипту.

Кроме того, uv позволяет запускать любые консольные команды, установленные в окружении, через ту же команду `uv run` или сокращение `uvx`. Например, если вы установили в зависимостях линтер **black**, можно выполнить `uvx black .` – uv создаст временное окружение, установит туда black и выполнит его [6] [7]. Однако `uvx` (или `uv tool run`) скорее для одноразовых утилит; для основных скриптов проекта достаточно `uv run`.

Подводя итог, при работе над проектом с uv основной цикл выглядит так: 1. **Инициализация**: `uv init projectname` – создает проект со всеми метаданными. 2. **Установка пакетов**: `uv add package` – добавляет зависимость, обновляя конфиги и устанавливая пакет в окружение. 3. **Запуск**: `uv run script.py` – запускает ваш код в изолированном окружении. 4. **Обновление/удаление**: `uv add pkg --upgrade` / `uv remove pkg` – управление зависимостями. 5. **Репликация окружения**: коммитите `pyproject.toml` и `uv.lock` в VCS. На другом месте `uv sync` создаст такое же окружение.

Это соответствует современному подходу управления проектами, аналогичному связке Poetry + venv, но с единообразной командной строкой. При этом uv может работать и с привычными командами pip: например, `uv pip install` или `uv pip list` выполнятся внутри окружения uv-проекта, что удобно для тех, кто постепенно переходит на uv.

Вопросы для самопроверки:

- Какие файлы создаются при инициализации нового проекта командой `uv init`? Что хранят файлы `pyproject.toml` и `uv.lock`?
- Как добавить новую зависимость в проект uv и где она будет зафиксирована?
- Чем отличается запуск скрипта командой `uv run` от прямого запуска `python` внутри активированного окружения?
- Сравните подход uv с инструментами **Poetry** или **Pipenv**: что общего и какие отличия?
- Как удалить или обновить уже установленный пакет с помощью uv и что при этом происходит с lock-файлом?

4. Этапы работы системного пакетного менеджера на примере apt

Рассмотрим, как работает установщик пакетов на уровне системы Linux, используя APT (Advanced Package Tool) в Debian/Ubuntu в качестве примера. Когда пользователь выполняет команду вроде `sudo apt install <package>`, запускается цепочка действий, которая заканчивается появлением установленной программы или библиотеки в системе. В общих чертах, **apt-get/apt выполняет следующие шаги**:

1. **Обращение к спискам пакетов:** APT сначала проверяет свои локальные индексы доступных пакетов. Эти индексы хранятся в каталоге `/var/lib/apt/lists/` и обновляются командой `apt update`. Если список пакетов устарел, apt предложит выполнить `apt update` прежде, чем устанавливать что-либо.
2. **Поиск и разрешение зависимостей:** apt находит в индексах запись о запрашиваемом пакете. Эта запись содержит метаданные, включая список зависимостей. Менеджер пакетов выясняет, какие другие пакеты нужны для работы данного (и не установлены ли они уже). Apt автоматически включит в план установки все необходимые зависимости. Если какие-то зависимости конфликтуют или требуют удаления других пакетов, apt оповестит об этом. Пользователю выводится список изменений: какие пакеты будут установлены (или обновлены), а какие удалены (например, при конфликтах или заменах).
3. **Загрузка пакетов:** после подтверждения apt начинает скачивать `.deb`-пакеты (дистрибутивные бинарные пакеты Debian) для указанного и для всех необходимых зависимостей. Загрузка идёт из удалённых репозиториев, указанных в конфигурации (/etc/apt/sources.list и /etc/apt/sources.list.d/*). Сами репозитории представляют собой каталоги в интернете (или на локальном носителе), где хранится множество .deb-файлов и индексы. Apt знает URL нужного файла, исходя из данных индекса (поле* Filename *в списке пакетов). Скачанные файлы сохраняются во временный кэш (по умолчанию в* `/var/cache/apt/archives/` *).
4. **Проверка целостности и подлинности:** apt убедится, что загруженные .deb-файлы не повреждены и действительно из доверенного источника. Каждый репозиторий APT имеет файл **Release**, подписанный GPG-ключом разработчиков дистрибутива. В Release перечислены хеш-суммы всех индексных файлов (Packages, Sources). Когда мы делаем `apt update`, эти подписи и суммы проверяются с использованием локально установленных доверенных GPG-ключей (хранятся в /etc/apt/trusted.gpg.d/). Таким образом, apt доверяет содержимому индексов. При загрузке конкретного пакета apt сравнивает его контрольную сумму с записанной в индексе (файл *Packages* содержит для каждого .deb SHA256 и др.). Если сумма не совпадает, пакет не будет установлен. Подпись каждого .deb-файла отдельно обычно не проверяется, доверие основано на целостности индексов и общей подписи Release.
5. **Установка пакета (через dpkg):** после успешного скачивания и проверки apt передает каждый .deb-файл низкоуровневому установщику **dpkg** (Debian Package Manager) для распаковки и настройки. Процесс установки dpkg включает:
6. Распаковку содержимого .deb в системные директории (обычно файлы кладутся в /usr/bin, /usr/lib, /etc и т.п. согласно структуре пакета). Dpkg при этом следит за коллизиями файлов, сохранением старых конфигурационных файлов и т.д..
7. Выполнение скриптов обслуживания, которые содержатся в пакете (если они есть). Это скрипты, такие как **preinst**, **postinst**, **prerm**, **postrm**, которые могут выполнять необходимые действия до или после установки/удаления пакета. Например, *postinst* может запустить службу или сгенерировать какие-то кеши, *prerm* – остановить службу перед удалением старой версии, и т.д. Эти скрипты устанавливаются пакетом и хранятся в `/var/`

`lib/dpkg/info/<package>.<postinst/preinst/...>`. Apt/dpkg автоматически запускает их в нужные моменты.

8. Регистрацию установленного пакета: dpkg ведет базу данных всех установленных пакетов в файле `/var/lib/dpkg/status`. Туда добавляется запись о новом пакете, включая версию, список файлов, состояния конфигурационных файлов и пр.
9. **Завершение и отчёт:** apt обновляет свои внутренние данные о состоянии системы (какие пакеты установлены, какие были автоустановлены как зависимости и могут быть удалены при необходимости командой `autoremove` и т.п.). После этого apt сообщает пользователю об успешной установке. Если пакет включает службы, они могут быть запущены, если это прописано (например, через systemd triggers).

Приведённый процесс несколько упрощён, но отражает суть. Apt фактически выступает “оркестром”: он решает, **что** ставить и откуда скачать, а работу по распаковке делегирует dpkg. Таким образом, ответ на вопрос “что делает `apt-get install` под капотом” – *скачивает нужные .deb, проверяет их, а затем вызывает dpkg, который распаковывает файлы и выполняет необходимые скрипты*.

Структура репозиториев, ключи и метаданные

APT-репозиторий – это, по сути, набор файлов: коллекция .deb-пакетов плюс индексы (метаданные), организованные особым образом. Обычно в адресе репозитория есть поддиректория `dists/<кодовое_имя_релиза>/` (например, `dists/focal/` для Ubuntu 20.04). В ней лежат файлы **Release** и **Release.gpg/InRelease** (подпись), а также каталогов по компонентам (main, universe, etc.) и архитектурам:

- **Release** – текстовый файл с списком хешей всех индексов (Packages, Sources) данного репозитория, а также информацией о версии дистрибутива. Этот файл подписан GPG-ключом поставщика репозитория.
- **Packages** – индекс двоичных пакетов для каждой комбинации компонент+архитектура. Например, `dists/focal/main/binary-amd64/Packages.xz` содержит запись о каждом пакете: имя, версия, описание, размер, MD5/SHA256 и **зависимости** (поля Depends, Recommends, etc.). Аналогично binary-i386/Packages для 32-бит и т.д.
- **Sources** – индекс исходных пакетов (может не потребоваться при установке бинарных .deb).
- **Contents** – вспомогательные файлы, указывающие, какой пакет содержит тот или иной файл (для поиска, не всегда присутствуют локально).

APT скачивает эти индексы при `apt update`. Отметим: **ключи** GPG, которыми подписан Release, хранятся в системе (APT keyring). Без соответствующего ключа apt откажется принимать пакеты из репозитория (их будут помечены как недоверенные). Поэтому при добавлении нового внешнего репозитория обычно устанавливают его публичный ключ (например, командой `apt-key add` в старых версиях или помещением файла .gpg в /etc/apt/trusted.gpg.d/). Только при верификации подписи Release apt может быть уверен, что индексы не были подделаны.

Зависимости и версии пакетов указаны прямо в индексах (в полях Depends, Conflicts, Provides и др.). Apt имеет встроенный алгоритм разрешения зависимостей. В простых случаях он просто собирает всё, что требуется, как описано выше. В более сложных – может выбирать между альтернативными пакетами, предлагать установить рекомендуемые пакеты и т.д. Если зависимости не могут быть удовлетворены (например, требуется пакет недоступной версии), apt сообщит об ошибке и не станет делать частичную установку – в этом отличие от pip, который

может оставить систему в состоянии конфликтов версий, apt же не допустит нарушений целостности зависимостей.

Аналогия apt и pip

Хотя apt и pip оба занимаются установкой программ, они работают на разных уровнях и существенно отличаются:

- **Источник пакетов:** apt получает программы из дистрибутивных репозиториев (например, официальный репозиторий Ubuntu), где пакеты прошли проверку и скомпилированы для данной ОС. Pip же берёт **Python-пакеты** из индекса PyPI (или других индексов), куда обычно публикуют сами разработчики библиотек. PyPI не привязан к конкретной ОС – там исходники и колеса (wheel) для разных платформ.
- **Формат и содержимое пакета:** `.deb` пакет apt содержит скомпилированные бинарные файлы, скрипты, данные – всё, что нужно для работы приложения, а также метаданные для системы (какие файлы куда пойдут, сервисы и т.д.). Пакеты pip – это либо исходники (sdist), либо колеса (wheel) с Python-модулями. Они устанавливаются только в среду Python (в `site-packages`), и pip не знает о системных службах или файлах вне этой директории.
- **Уровень привилегий:** apt устанавливает пакеты системно (требует sudo), изменяя общую файловую систему (/usr, /etc). Pip обычно ставит в пользовательское или виртуальное окружение, не требуя прав суперпользователя (если не использовать `sudo pip`, что не рекомендуется). Поэтому apt используется для системных утилит и библиотек, доступных всем пользователям, а pip – для зависимостей внутри проектов.
- **Управление зависимостями:** apt строго контролирует версии – в репозитории обычно одна версия библиотеки, совместимая со всем остальным. Pip позволяет иметь разные версии библиотек в разных виртуальных окружениях, но внутри одного окружения конфликт версий двух пакетов может проявиться лишь во время исполнения (раньше pip просто брал последний совместимый, теперь у него есть базовое разрешение конфликтов, но не такое мощное). Apt не позволит установить два разных варианта одной и той же системной библиотеки одновременно (за исключением специальных случаев с разными именами пакетов).
- **Удаление и обновление:** apt может обновить систему до новой версии пакета или удалить пакет полностью, зная список файлов. Pip также может обновлять/удалять, но это ограничено Python-пакетами. Кроме того, apt хранит **статус** – какие пакеты установлены явно, а какие как зависимости (и не используются), предлагая чистить ненужные (`apt autoremove`). Pip такой автоматизации не имеет – ненужные зависимости остаются, пока вы их явно не удалите или не создадите новое окружение.

В некотором смысле, `pip install` похож на `apt install` для Python-библиотек, а **PyPI** – аналогично репозиторию (хранилищу) пакетов. Но apt – это **системный** менеджер, который заботится о всей ОС, а pip – инструмент в рамках интерпретатора Python. Они дополняют друг друга: например, apt можно установить сам Python и некоторые C-библиотеки, необходимые для Python-пакетов, а pip – взять специфичные Python-зависимости для проекта.

Примечание: Никогда не стоит путать область действия apt и pip. Не устанавливайте через pip то, что предназначено для установки apt'ом (например, системные утилиты), и наоборот. В Ubuntu, к примеру, есть пакет `python3-requests` для apt и есть `requests` в PyPI для pip – первый установит requests глобально для системы (в /usr/lib/python3/dist-packages), второй – в ваше виртуальное окружение или пользовательский каталог. Эти два менеджера не знают о пакетах друг друга.

Вопросы для самопроверки:

- Какие действия выполняет `apt` перед тем, как передать пакет на установку программе `dpkg`?
- Зачем нужна команда `apt update` и что произойдёт, если её не выполнить перед установкой нового пакета?
- Где хранятся сведения о доступных пакетах и их зависимостях на локальной машине?
- Как `apt` проверяет подлинность скачанных пакетов и защищает систему от установки поддельных данных?
- В чём разница между установкой библиотеки через `apt` (например, `python3-numpy`) и установкой через `pip install numpy`?

5. Этапы работы pip (Python Package Installer)

Рассмотрим теперь, как работает установка пакетов через **pip** – стандартный пакетный менеджер Python. Команда `pip install <package>` выполняет за кулисами следующий алгоритм:

1. **Поиск пакета в индексе (PyPI):** По умолчанию pip ищет указанный пакет в **Python Package Index (PyPI)** – центральном репозитории Python-пакетов. Для этого pip обращается к так называемому *simple index* PyPI по HTTP(S) [8]. Индекс – это упрощённый веб-страницы со списком файлов (дистрибутивов) доступных версий пакета. Pip получает этот список и выбирает подходящую версию: если версия не указана явно, обычно берётся последняя стабильная версия, удовлетворяющая возможным ограничениям (с учётом флагов `--pre` для pre-release и т.д.). Если указаны зависимости (например, через requirements.txt), pip будет стараться найти совместимые версии всех требуемых пакетов (сейчас pip использует решатель зависимостей, пытаясь подобрать сочетание версий, удовлетворяющее всем требованиям). **Важно:** pip по умолчанию ищет в PyPI, но может использовать альтернативные индексы (опции `-i/--index-url`) или локальные директории (через `--find-links`). Также можно отключить обращение в интернет (флаг `--no-index`) – тогда установка возможна только из указанных источников (например, wheel-файлов на диске).

2. **Выбор файла для загрузки:** найдя список доступных версий, pip определяет, **какой файл скачать**. Python-пакеты могут распространяться как исходники (Source Distribution, обычно архив .tar.gz) или как готовые колеса (Wheel, файл .whl). **Pip предпочитает скачать wheel-файл, если он есть для вашей платформы**, потому что колесо уже содержит скомпилированные бинарники (если нужны) и не требует сборки. Если подходящего wheel не найдено, pip скачает исходный архив и будет собирать пакет из исходников. Например, для пакета с расширением на C, PyPI может иметь wheel для Windows и Linux, но не для вашей редкой платформы – тогда pip возьмёт sdist и скомпилирует его самостоятельно. (Можно явно указать флаги `--no-binary` или `--only-binary` чтобы контролировать этот выбор.)

3. **Загрузка пакета:** pip скачивает выбранный файл по ссылке (обычно HTTPS URL на файлы PyPI). По умолчанию скачанный файл сохраняется в локальный кеш pip, расположенный в `~/.cache/pip` на Linux. Кеширование означает, что если вы уже качали этот пакет (тот же файл) ранее, pip может не скачивать его повторно при следующей установке – он возьмёт его из кеша, экономя время и трафик. (Кеш можно отключить флагом `--no-cache-dir`, или очистить командой `pip cache purge`.)

4. **Проверка и приготовление к установке:** pip проверяет целостность файла (на PyPI для каждого файла хранится хеш SHA256, pip проверяет его автоматически). Далее поведение зависит от типа файла:

5. Если это **wheel (.whl)**: этот формат – просто zip-архив с заранее собранными `.py` и `.so` (если есть расширения), плюс метаданные. Его **не нужно компилировать**, поэтому pip может сразу перейти к шагу установки.

6. Если это **исходник (sdist)**: pip должен сначала собрать из него wheel. Начиная с PEP 517, у пакета может быть указан «сборщик» (build backend) и зависимости для сборки. Pip создаст временную изолированную среду для сборочных зависимостей, установит туда, например, setuptools или другие инструменты, затем запустит процесс сборки – обычно `setup.py bdist_wheel` или аналог через указанное API. В результате получится файл .whl. **Pip кэширует собранные wheel-файлы** (в директории кеша pip, обычно в подпапке `wheels`), чтобы в будущем не повторять компиляцию. Таким образом, **если pip не находит wheel, он строит его локально и сохраняет для последующих установок**. Например, если вы установили пакет X версии 1.0 из исходников, при следующей установке X==1.0 на этой же машине pip уже возьмёт скомпилированный wheel из кеша, а не будет собирать заново.

7. **Установка (развёртывание):** на этом этапе у pip на руках wheel-файл – либо скачанный, либо только что собранный. Установка колеса означает:

8. Распаковать `.whl` (это zip-архив) в целевой каталог пакетов. Если вы в виртуальном окружении, то обычно это `<env>/lib/pythonX.Y/site-packages/`. Если это глобальная установка с правами, то каталог может быть `/usr/local/lib/pythonX.Y/dist-packages/` (подробнее в следующем разделе про пути).
9. При распаковке создаётся папка модуля/пакета, а также специальная директория с метаданными `<package>-<version>.dist-info`, содержащая сведения о установленном пакете: версия, список файлов, зависимые пакеты, лицензия и т.д.
10. Если пакет предоставляет консольные скрипты (через entry points), pip генерирует исполняемые файлы в директории `<env>/bin/` (или `Scripts\` на Windows). Эти файлы при запуске будут импортировать пакет и вызывать нужную функцию. Например, после `pip install flask` появится команда `flask` в Scripts, которая запускает `flask.cli` модуль.

11. Pip может также выполнять неявно некоторые post-install действия. Например, может компилировать `.py` файлы в `.pyc` (байткод) для ускорения импорта. Обычно это делается самими wheels (они могут содержать уже скомпилированные файлы), либо Python сам скомпилирует при первом запуске.

12. **Установка зависимостей:** Если устанавливаемый пакет **имеет зависимости**, pip по умолчанию также попытается их установить (если они не уже установлены). В metadata пакета указаны `install_requires` – минимальные версии необходимых библиотек. Pip решает эти зависимости рекурсивно. Раньше pip просто устанавлививал все указанные зависимости по очереди, сейчас же он старается избежать конфликтов версий: если два пакета требуют несовместимых версий одной библиотеки, pip обнаружит это и сообщит о невозможности разрешить зависимости (Dependency conflict). В таких случаях нужно либо явно указать версии, либо обновить один из пакетов. Но в отличие от apt, pip не всегда может идеально подобрать сочетание версий (нет централизованного списка совместимости), это остаётся задачей разработчика/пользователя или более

высокоуровневых средств вроде Poetry. Тем не менее, pip **никогда не "сломает" существующие пакеты самовольно** – если конфликт, он прекратит операцию, не удаляя ничего (или, если используете флаг `--upgrade`, то обновит только если все зависимости ок).

13. **Завершение:** после успешной установки pip выводит список установленных пакетов (или сообщает об успехе). Теперь пакет доступен для импорта в Python.

Стоит отметить, что pip не ведёт одну глобальную базу установленных пакетов, как apt. Он ориентируется на конкретное окружение. Список установленных можно увидеть командой `pip list` или `pip freeze` (последняя выводит формат пригодный для requirements.txt). Информацию о конкретном пакете – командой `pip show <name>`, которая, в частности, указывает путь, **где пакет установлен** [9] [10]. Например, `pip show requests` может вывести строку `Location: /home/user/venv/lib/python3.11/site-packages` – это путь, откуда Python будет импортировать requests.

Работа с `requirements.txt`: Requirements-файл – просто список зависимостей (с optional указанием версий) строка за строкой. Pip может установить всё, что перечислено в таком файле, командой `pip install -r requirements.txt`. По сути, pip просто последовательно обработает каждую строку как аргумент установки. Часто requirements.txt фиксирует версии (`==`), чтобы гарантировать всем разработчикам/серверу одинаковое окружение. Однако в отличие от uv.lock или poetry.lock, requirements.txt обычно создаётся вручную или через `pip freeze` и не содержит хешей – это не столь строгий lockfile, но при должном контроле версий он выполняет свою функцию.

Кэширование и офлайн-установка: как упомянуто, pip хранит кеш скачанных пакетов. При повторной установке того же пакета (особенно при CI/CD или сборках Docker) это экономит время. Если необходимо установить пакеты на машине без интернета, можно заранее скачать нужные wheel-файлы (например, командой `pip download`) и затем установить их локально с флагом `--no-index --find-links /path/to/wheels`. Pip также имеет команду `pip cache list` / `pip cache dir` / `pip cache purge` для управления своим кешем.

Сравнение wheel vs sdist: Колесо (.whl) – это pre-built пакет. Преимущество – быстрая установка (распаковал и готово). Недостаток – колесо может быть специфично для ОС/архитектуры/версии Python (например, содержит скомпилированный .so под конкретный ABI). Исходник (sdist) универсален, но требует выполнения setup.py и компиляции, что дольше и может потребовать установленного компилятора и dev-библиотек. Pip всегда пытается взять wheel, если доступно совместимое, и лишь при отсутствии – берёт исходник и строит локально. Это сделано и для скорости, и для безопасности: установка из исходников выполняет произвольный код (setup.py), тогда как wheel – это уже собранный артефакт, установка которого менее рискованна (но все равно требует доверия к пакету в целом).

Подключение к PyPI: pip использует HTTPS для связи с `pypi.org`. Он проверяет SSL-сертификаты (при необходимости можно настроить, например, доверять корпоративному прокси, используя опции – но в общем случае это прозрачно). PyPI – огромный репозиторий, поэтому при запросе пакета pip может быть перенаправлен на конкретный CDN. Существуют зеркала PyPI, и pip может быть настроен на них (например, через переменную окружения PIP_INDEX_URL или конфиг). Это бывает полезно, если у основного PyPI проблемы или нужна другая геолокация.

Внутренне pip реализован на Python и использует библиотеку **urllib/requests** для загрузки, распаковки файлов для установки и т.д. Весь процесс происходит в вашем пользовательском пространстве, без особых привилегий (если вы не ставите глобально с sudo). Поэтому **pip не может установить системные пакеты** (например, нельзя через pip поставить C-компонент в /usr/lib), он оперирует только в директориях, доступных интерпретатору Python для модулей.

Вопросы для самопроверки:

- Где pip ищет информацию о пакетах по умолчанию и как можно изменить или ограничить этот поиск?
- В каком случае pip будет загружать исходный код пакета, а не готовый wheel-файл? Что делает pip после загрузки исходников?
- Куда pip помещает скачанные файлы и с какой целью? Что произойдёт, если запустить установку уже однажды установленного пакета на той же машине?
- Как pip определяет, в какой каталог установить пакет? Чем отличаются пути установки при использовании виртуального окружения, глобально с sudo и с флагом `--user`?
- Что содержится в директории `<package>.dist-info` и для чего эта информация используется?
- Для чего предназначен файл requirements.txt и в чём его отличие от lock-файлов (например, uv.lock или poetry.lock)?

6. Физическое размещение Python-библиотек в Linux

При работе с Python на Linux-платформах важно понимать, **где располагаются установленные модули и пакеты**, и почему их несколько мест. Расположение модулей влияет на приоритет их загрузки и на то, каким образом устанавливать/обновлять пакеты. Рассмотрим три основных директории, связанных с Python 3 (предположим, версия Python 3.X используется):

- `/usr/lib/python3.X/` – базовая директория стандартной библиотеки Python. Содержит модули, идущие в комплекте с интерпретатором Python, установленным через систему. Например, там находятся папки `collections/`, `asyncio/`, файлы `random.py`, `os.py` и т.п., а также скомпилированные расширения `.so` стандартных модулей. *Замечание:* в некоторых дистрибутивах (Debian/Ubuntu) структура чуть сложнее: часть стандартных модулей может лежать непосредственно в `/usr/lib/python3.X`, а часть – в `/usr/lib/python3`, но для целей нашего обсуждения можно считать, что `/usr/lib/python3.X` – это “стандартная библиотека” данной версии Python.

- `/usr/lib/python3/dist-packages/` – каталог для **системно устанавливаемых** Python-пакетов. На Debian/Ubuntu именно сюда менеджер пакетов apt кладёт модули, установленные через `apt`. Например, если вы установили через apt пакет `python3-numpy`, его файлы окажутся в `/usr/lib/python3/dist-packages/numpy/`. Эта директория добавляется в `sys.path` Python’ом (смотрите модуль site). Она одна для всех Python 3 версий в системе (в пределах совместимости) – обычно дистрибутив стремится, чтобы одна директория dist-packages обслуживала текущую основную версию Python 3. **Итог:** `/usr/lib/python3/dist-packages` содержит пакеты, поставленные ОС (apt), “проверенные” и обычно одни на всю систему.

- `/usr/local/lib/python3.X/dist-packages/` – каталог для **локально устанавливаемых глобальных** пакетов. Сюда попадают библиотеки, которые **пользователь установил через pip глобально** (с правами root, без виртуального

окружения). Почему именно сюда? Дело в том, что директория `/usr/local` традиционно предназначена для локальной установки ПО вне управления пакетного менеджера ОС. Python (в Debian/Ubuntu) настроен так, что при запуске добавляет `/usr/local/lib/python3.X/dist-packages` в `sys.path` перед `/usr/lib/python3/dist-packages`. Таким образом, локально установленные пакеты **имеют приоритет** над системными, если имена совпадают. Например, если администратор установил через pip пакет “requests” глобально, а в системе уже был “python3-requests” из apt, то Python будет импортировать версию из /usr/local (та, что поставил pip), а не системную. Это сделано, чтобы не мешать пользователю использовать более новые версии библиотек, чем те, что поставляются с ОС (на свой страх и риск). В общем, **/usr/local/lib/python3.X/dist-packages – для pip install с sudo**.

Кроме этих, есть ещё `~/.local/lib/python3.X/site-packages/` – это **пользовательская директория** для установки пакетов без прав root (с флагом `--user` для pip, или если pip настроен на user install по умолчанию). Она находится в домашнем каталоге пользователя и тоже обычно включена в `sys.path` (но в Ubuntu, обратите внимание, переменная окружения может требоваться: начиная с Python 3.10+ на некоторых системах user-site может быть отключён по умолчанию, а включается при вызове pip с `--user` или если выставлена `PYTHONUSERBASE`). Тем не менее, **pip без sudo** часто пишет именно в `~/.local/lib/.../site-packages`. Например, `pip install --user safeeyes` положит пакет **safeeyes** в `~/.local/lib/python3.X/site-packages`. Эта директория видна только для вашего пользователя и не влияет на системные или глобальные установки.

Таким образом: - Apt-пакеты → `/usr/lib/python3/dist-packages` - Глобальный pip (sudo) → `/usr/local/lib/python3.X/dist-packages` - Локальный pip (–user) → `~/.local/lib/python3.X/site-packages` - Виртуальное окружение → `<env>/lib/python3.X/site-packages` (отдельно для каждого env)

Иногда возникает вопрос: *почему Debian использует dist-packages, а не стандартный site-packages?* Исторически, это особенности политики Debian. **dist-packages** отделены от **site-packages** чтобы разделить “дистрибутивные” пакеты и “локально установленные”. В CPython по умолчанию ожидался `.../lib/python3.X/site-packages` для внешних пакетов. Debian патчит модуль `site` так, чтобы вместо site-packages использовались dist-packages для системных путей. `/usr/local/lib/python3.X/dist-packages` по сути выполняет роль site-packages для глобальных установок, а `/usr/lib/python3/dist-packages` – место для apt-пакетов. Это защищает от ситуации, когда пользователь поставил пакет через pip, а apt потом установил другую версию – они лежат в разных папках. Однако Python всё равно видит оба, просто в определённом порядке.

Порядок поиска (sys.path): при запуске интерпретатора Python формирует список путей, откуда импортировать модули. Обычно он выглядит так (для Debian-подобной системы): 1. Текущая директория (или скрипт, если запущен файл – директория файла). 2. `/usr/local/lib/python3.X/dist-packages` (локальные глобальные пакеты). 3. `/usr/lib/python3/dist-packages` (системные пакеты). 4. Стандартная библиотека `/usr/lib/python3.X` (на самом деле она может добавляться раньше, но т.к. она находится внутри lib/python3.X, технически она в sys.path тоже). 5. Пользовательская директория `~/.local/lib/python3.X/site-packages` **может** добавляться, обычно после локальных и системных (при условии, что не отключена). В частности, когда вы запускаете Python, если переменная окружения `PYTHONNOUSERSITE` не установлена, модуль site добавит user-site в конец списка путей. 6. Также могут быть .pth файлы, которые добавляют дополнительные пути.

Чтобы увидеть этот список, можно в интерактивном интерпретаторе распечатать `import sys; print("\n".join(sys.path))`.

Как узнать, откуда импортируется модуль? Есть несколько способов: - Воспользоваться интерактивно: после `import <module>`, посмотреть атрибут `<module>.__file__` – он обычно содержит путь к файлу, откуда модуль загружен. Например:

```
>>> import requests
>>> print(requests.__file__)
/home/user/.local/lib/python3.10/site-packages/requests/__init__.py
```

Это укажет точную директорию. - Команда `pip show <package>` выдаёт строку **Location**, где находится пакет [9]. Это удобнее, когда пакет установлен, но модуль не загружен. Например, `pip show numpy` может показать `Location: /usr/local/lib/python3.10/dist-packages` – значит, numpy импортируется из глобальной установки pip. - Использовать утилиту `python -m site` – она выведет, какие директории считаются “site-packages” и включены в поиск (включая user-site). - Проверить `sys.path` вручную, как упоминалось.

Пример: представим, что вы установили пакет через apt и через pip одновременно (не рекомендуется, но бывает). Допустим, `apt install python3-pandas` положил pandas в `/usr/lib/python3/dist-packages/pandas`, а затем `pip install pandas` (с sudo) установил другую версию в `/usr/local/lib/python3.10/dist-packages/pandas`. В таком случае:

```
>>> import pandas
>>> pandas.__version__
'1.5.3'
>>> print(pandas.__file__)
/usr/local/lib/python3.10/dist-packages/pandas/__init__.py
```

Версия и путь укажут, что взята версия из /usr/local (pip-овская), потому что этот путь стоит выше. Если же вы хотите убедиться, что конкретно apt-овская версия используется, нужно либо удалить глобальный (pip) вариант, либо манипулировать `sys.path` (что делать нежелательно). В идеале избегайте такой конкуренции установок.

Различия между dist-packages директориями: как мы выяснили: - `/usr/lib/python3/dist-packages` – **дистрибутивные (apt)** пакеты. - `/usr/local/lib/python3.X/dist-packages` – **локальные (pip)** глобальные пакеты. - (A site-packages используется для виртуальных окружений и user-установок в домашней директории.)

Зная это, легко понимать сообщения и поведение. Например, `pip list` запущенный без окружения, покажет все пакеты, которые pip видит (это, скорее всего, объединение /usr/local и ~/.local, так как pip не отслеживает apt-пакеты). А вот `apt list --installed | grep python3-` покажет, что установлено через apt.

pip vs apt – определение местоположения: команду `pip show` мы уже обсудили – она хороший помощник. Также можно использовать утилиты типа `python -c "import pkgutil; print([m.module_finder.path for m in pkgutil.iter_modules() if m.name=='<module_name>'])"`, но это избыточно. В простых случаях `pip show` достаточно:

“Location: ...”. Помним только, что если один пакет установлен двумя способами, pip show может увидеть только тот, который установлен pip’ом. Apt-пакеты pip не “знает”, но Python всё равно их импортирует. В таких случаях лучше явно проверять `__file__` после импорта.

Вопросы для самопроверки:

- Чем отличаются директории **dist-packages** и **site-packages** в контексте дистрибутивов Debian/Ubuntu? Куда будут помещены файлы, если вы установите пакет командой `sudo pip install <pkg>`? А куда – если `pip install --user <pkg>`?
- Почему пакет, установленный через pip, может “затмить” версию пакета, установленную через системный менеджер apt? В каком порядке Python ищет модули при импорте?
- Как узнать, из какого каталога была загружена конкретная библиотека при выполнении скрипта?
- Какие есть способы избежать конфликтов между пакетами, установленными разными способами (глобально/через apt/в виртуальном окружении)?
- Что произойдет, если установить одну и ту же библиотеку и через apt, и через pip? Как Python решит, какую версию использовать?

---

[1] Managing Python versions with pyenv
https://thepythoncorner.com/posts/2022-05-07-managing-python-versions-with-pyenv/

[2] [3] [6] [7] Быстрый старт в мир Python окружений с uv / Хабр
https://habr.com/ru/articles/875840/

[4] [5] Installing and managing Python | uv
https://docs.astral.sh/uv/guides/install-python/

[8] pip install - pip documentation v25.1.1
https://pip.pypa.io/en/stable/cli/pip_install/

[9] [10] python - Where does pip install its packages? - Stack Overflow
https://stackoverflow.com/questions/29980798/where-does-pip-install-its-packages