# Питон в коробке – venv в python 3.3

Python

Tutorial

Наверняка, большинство из тех, кто разрабатывает или деплоит Python приложения, использует виртуальные окружения. В частности через virtualenv, написанный Ian Bicking.

Идея оказалась так хороша и распространена, что нечто похожее теперь присутствует в Python 3.3 из коробки в виде модуля **venv**. Он почти такой же, как virtualenv, только **немного лучше**.

## Как это работает?

Основное отличие venv в том, что он встроен в интерпретатор и может отрабатывать ещё до загрузки системных модулей. Для этого, при определении базовой директории с библиотеками, используется примерно такой алгоритм:

- в директории с интерпретатором или уровнем выше ищется файл с именем `pyvenv.cfg`;
- если файл найден, в нём ищется ключ home, значение которого и будет базовой директорией;
- в базовой директории идёт поиск системной библиотеки (по спец. маркеру `os.py`);
- если что-то пошло не так – всё откатывается к захардкоженному в бинарнике значению.

Вот и вся суть venv, всё остальное уже обёртка над этим.

## Как создать?

Всё очень просто, нужно вызвать через ключ `-m` модуль `venv`, либо использовать встроенный скрипт pyvenv:

```
pyvenv /path/to/new/venv
```

Скрипт создаст указанную директорию, вместе со всеми родительскими директориями, если потребуется, и построит виртуальное окружение. Это можно делать и в Windows, только вызов будет чуть более многословным:

```
c:\Python33\python -m venv /path/to/new/venv
```

При создании можно добавлять различные параметры, как, например, включение системных site-packages или использование symlink вместо копирования интерпретатора.

В отличии от virtualenv новый venv требует чтобы создаваемая директория не существовала, либо была пустой. ~~Вероятно, это сделано, чтобы не допускать конфликтов с существующими файлами.~~ Это бага в python 3.3, в 3.4 уже исправлено. (Спасибо, svetlov).

## Как использовать?

Можно использовать старый добрый метод активации через bin/activate (Scripts/activate в windows):

```
cd /path/to/new/venv
. bin/activate
python3 some_script.py
```

А можно и **не использовать**, достаточно лишь вызвать интерпретатор из окружения и всё сработает автоматически:

```
/path/to/new/venv/bin/python3 some_script.py
```

Это конечно не сработает для скриптов, запускаемых напрямую через `#!/usr/bin/env python3`, для них всё равно нужно будет, как и раньше, делать активацию. Решение есть – о нём чуть ниже.

## Обновление

Если в вашей системе обновилась версия python, то виртуальное окружение иногда тоже нужно обновить.
Всё просто – вызываем venv аналогично созданию окружения, добавив ключ `--upgrade`:

```
pyvenv --upgrade /path/to/new/venv
```

Это произойдёт автоматически, если использовать symlink, но если вы хотите кроме изоляции делать фиксацию версии python и библиотек, я бы рекомендовал делать

обновление вручную.

## Расширение EnvBuilder

Вся работа по созданию окружения падает на класс `venv.EnvBuilder`, этот класс написан так, чтобы его можно было расширять.

Например, можно при инициализации окружения ставить туда `distribute`, `pip` и необходимые начальные зависимости из `requirements.txt`. Более сложную логику лучше оставить на совести более предназначенных для этого инструментов, типа buildout или make, но первоначальную настройку можно провести и на уровне EnvBuilder.

При создании окружения используется метод `create(self, env_dir)`, в исходном классе он выглядит так:

```
    def create(self, env_dir):
        env_dir = os.path.abspath(env_dir)
        context = self.ensure_directories(env_dir)
        self.create_configuration(context)
        self.setup_python(context)
        if not self.upgrade:
            self.setup_scripts(context)
            self.post_setup(context)
```

Метод описывает суть всего процесса: создание директории (`ensure_directories`), конфигурацию (`create_configuration`), добавление бинарников питона (`setup_python`) и добавление скриптов активации (`setup_scripts`).

В конце вызывается хук `post_setup`, в который вы можете добавлять свои действия. Видно, что `post_setup` выполняется только при создании окружения, а при `--upgrade` он выполняться не будет. Это легко исправить, добавив ещё один хук:

```
class ImprovedEnvBuilder(venv.EnvBuilder):

    def create(self, env_dir):
        """Overwrite create method (add more hooks)"""
        env_dir = path.abspath(env_dir)
        context = self.ensure_directories(env_dir)
        self.create_configuration(context)
        self.setup_python(context)
        if not self.upgrade:
            self.setup_scripts(context)
            self.post_setup(context)
        else:
```

```
            self.post_upgrade(context)

    def post_upgrade(self, context):
        pass
```

В качестве параметров при вызове методов после `ensure_directories` будет передаваться `context` — объект, содержащий в виде атрибутов всю необходимую информацию о создаваемом окружении. Почему-то в документации пока эти ключи не описаны, но вы легко сможете понять всё самостоятельно, заглянув в код метода `ensure_directories` в базовом классе. Приведу самые полезные из атрибутов:

- context.bin_path — путь к директории с бинарниками и исполняемыми скриптами,
- context.env_dir — путь к директории с созданным окружением,
- context.env_exe — путь к бинарнику внутри окружения.

Соответственно, для запуска python скрипта внутри окружения, можно сделать:

```
import subprocess
import venv
class MyEnvBuilder(venv.EnvBuilder):
    def post_setup(self, context):
        script = '/path/to/some_script.py'
        subprocess.call([context.env_exe, script])
```

## Исполняемые скрипты внутри venv

Вернёмся к проблеме с исполняемыми скриптами внутри виртуального окружения.

В virtualenv для них достаточно было указать интерпретатор через `#/usr/bin/env python3` и использовать, не забывая сделать `. bin/activate`. Если вас такой подход устраивал, то вы можете им продолжать пользоваться и в `venv`.

Есть и новый путь. Внутри EnvBuilder реализован метод `install_scripts(self, context, path)`, который автоматизирует копирование скриптов и бинарников в создаваемое окружение. В `path` необходимо передать путь к директории с вложенными поддиректориями «common», «nt», «posix» и т.д. В поддиректории, в свою очередь, положить необходимые скрипты или бинарники. В «common» скрипты для всех платформ, в «nt» – для Windows, «posix» – для Linux, Mac OS X и других posix систем.

Кроме того, для текстовых файлов выполняется постановка значений. Из коробки поддерживаются:

- `__VENV_DIR__`
- `__VENV_NAME__`
- `__VENV_BIN_NAME__`
- `__VENV_PYTHON__`

Пример шаблона запускаемого python скрипта:

```
#!__VENV_PYTHON__

import sys
import my_module

if __name__ == '__main__':
    sys.exit(my_module.run(sys.argv))
```

`__VENV_PYTHON__` будет заменено на полный путь к интерпретатору python в виртуальном окружении.

После установки такого скрипта через `install_scripts`, его можно будет запускать, без необходимости активации окружения через bin/activate.

…

- Документация по модулю venv
- PEP-405 – Python Virtual Environments
- Небольшой репозиторий на github, в котором я сделал пример дополнения EnvBuilder автоматической установкой distribute, pip, списка зависимостей из requirements.txt и набором исполняемых скриптов.