Задание 4.1 На основе материалов преподавателя, рекомендованной литературы, а также ресурсов сети интернет (видеозаписи выступления по теме на конференциях) разработать конспект с примерами по использованию различных систем виртуализации проектов на Python. Требуется рассмотреть virtualenv, pipfile, pyenv, pipenv, venv. Привести примеры с разбором конфигурационных файлов 2 систем.

Задание 4.2 Создать глоссарий команд, использующихся для создания виртуального окружения для одной из описанных систем.

virtualenv

virtualenv — это инструмент, позволяющий создавать виртуальные окружения с пакетами. Разные «песочницы» (согласитесь, напоминает sandbox’ы в [Cabal](https://eax.me/cabal/)) имеют разный набор пакетов разных версий. Работая над конкретным проектом, вы просто переключаетесь на подходящую песочницу, и проблема уходит. Заметьте, что в отличие от того, как это сделано в некоторых других языках программирования, одну и ту же песочницу можно смело использовать сразу в нескольких проектах. Плюс к этому уходит проблема засорения системы ненужными пакетами, так как песочницы можно легко создавать и удалять. Как я это вижу, программируя на Python в третьем тысячелетии, вообще все нужно делать через virtualenv!

Установка virtualenv и virtualenvwrapper, предоставляющего чуть более удобный интерфейс к virtualenv:

sudo pip3 install virtualenv virtualenvwrapper

В ~/.bashrc дописываем:

export VIRTUALENVWRAPPER\_PYTHON=/usr/bin/python3  
source /usr/bin/virtualenvwrapper.sh

Создаем новое окружение:

mkvirtualenv env-name

Смотрим список окружений:

workon

Меняем окружение:

workon env-name

Выходим из окружения:

deactivate

Удаляем окружение:

rmvirtualenv env-name

Находясь в одном из окружений, можно ставить пакеты через pip, как обычно:

pip3 install flask

Список зависимостей проекта, насколько я смог разобраться, принято сохранять в файл с именем requirements.txt:

pip3 freeze > requirements.txt

Этот подход позволяет одной командой установить все зависимости, необходимые проекту:

pip3 install -r requirements.txt

Заметьте, что если вы используете [bpython](http://www.bpython-interpreter.org/screenshots.html), его тоже придется установить в песочницу. После этого его можно запустить как обычно, сказав bpython. Подозреваю, что то же справедливо и в отношении некоторых других утилит, написанных на Python. В связи с этим поддержание минимального requirements.txt становится не такой уж простой задачей.

**Pipenv**

[**Pipenv**](https://github.com/pypa/pipenv) — это набирающий популярность пакет управления виртуальным окружением для Python, который решает некоторые распространенные проблемы, связанные с типичным рабочим процессом, в котором используется **pip**, **virtualenv** и старый добрый файл **requirements.txt**.

Как написано на официальном сайте: Pipenv — это инструмент, который призван привнести в мир Python лучшее из всех упаковочных миров (bundler, composer, npm, cargo, yarn и т. д.).

Помимо решения некоторых распространенных проблем, он стандартизицирует и упрощает процесс разработки с помощью единого инструмента командной строки.

В этом руководстве мы рассмотрим, какие проблемы решает Pipenv и как управлять зависимостями Python с помощью Pipenv.

Какие проблемы решает Pipenv?

Для того чтобы понять преимущества Pipenv, нужно изучить текущие методы работы с пакетами и управления зависимостями в Python.

Давайте начнем с типичной ситуации в работе со сторонними пакетами. Затем мы рассмотрим пример развертывания приложения Python.

Управление зависимостями через requirements.txt

Представьте, что вы работаете над проектом Python, в котором используется сторонний пакет, например такой как **flask**. Нам необходимо указать этот пакет в списке зависимостей, чтобы другие разработчики и автоматизированные системы могли использовать наше приложение.

Таким образом, мы решаем включить зависимость от **flask** в файл **requirements.txt**:

flask

Отлично, локально все работает, и после некоторого улучшения нашего приложения мы решили перенести его в продакшин. И вот тут все становится немного сложнее …

В приведенном выше файле **requirements.txt** нет указаний, какую версию **flask** нужно использовать. В этом случае **pip install -r requirements.txt** по умолчанию установит последнюю версию. Это нормально, если только в последней версии нет изменений интерфейса или поведения, которые нарушают наше приложение.

Для примера предположим, что вышла новая версия **flask**. Однако она не имеет обратной совместимости с версией, которую мы использовали во время разработки.

Теперь предположим, что мы развертываем свое приложение в рабочей среде и выполняете команду **pip install -r requirements.txt**. Pip инсталлирует последнюю, несовместимую с предыдущей версией версию **flask**, и именно так наше приложение выходит из строя ….

«Но эй, это работало на моей машине!» — я сам был, в такой ситуации много раз и это не очень приятное чувство.

На данный момент мы знаем, что версия **flask**, которую мы использовали во время разработки, работала нормально. Итак, чтобы исправить ситуацию, попытаемся быть немного более конкретным в наших требованиях в **requirements.txt**. Укажем конкретную версию flask. Это также называется «закреплением» (*pinning* a dependency) зависимости:

flask==0.12.1

«Закрепление» зависимости **flask** к конкретной версии гарантирует, что **install -r requirements.txt** установит точную версию flask, которую мы использовали во время разработки. **Но так ли это на самом деле?**

Имейте в виду, что у самого **flask** также есть зависимости (которые устанавливаются автоматически). Тем не менее, сам **flask** не определяет точные версии для своих зависимостей. Например, он допускает любую версию **Werkzeug >= 0.14**.

Опять же, ради этого примера, допустим, была выпущена новая версия **Werkzeug**, и предположим что эта версия вызывает ошибку в нашем приложения.

На этот раз, когда мы выполним **pip install -r requirements.txt** в рабочей среде, получим **flask == 0.12.1**, так как мы закрепили это в нашем **requirements.txt**. Однако, к сожалению, мы так же получили самую последнюю версию Werkzeug которая вызывает у нас ошибку. Опять неожиданная проблема в другом окружение.

Реальная проблема здесь в том, что зависимости нашего проекта **не являются детерминированными**. Под этим я подразумеваю, что при одинаковых входных данных (файл requirements.txt) pip не всегда создает одну и ту же среду. Получается что, в настоящее время мы не можем легко воспроизвести ту среду, которая есть у нас, на нашей локальной машине.

Типичным решением этой проблемы является использование **pip freeze**. Эта команда позволяет нам получить точные версии для всех сторонних библиотек, установленных в текущий момент времени, включая pip зависимостей, установленный автоматически. Таким образом, мы можем «заморозить» все зависимости в процессе разработки, чтобы обеспечить одинаковую среду в другом окружением.

Выполнение команды **pip freeze** приводит к «закреплению» зависимостей, которые мы можем добавить вручную в файл requirements.txt (или автоматически командой: pip freeze > requirements.txt):

click==6.7

Flask==0.12.1

itsdangerous==0.24

Jinja2==2.10

MarkupSafe==1.0

Werkzeug==0.14.1

С помощью закрепления зависимостей мы можем быть уверены, что пакеты, установленные в нашей производственной среде, будут точно соответствовать пакетам в нашей среде разработки, чтобы ваш проект неожиданно не ломался. Это «решение», к сожалению, приводит к совершенно новому набору проблем.

Теперь, когда мы указали точные версии каждого стороннего пакета, **мы несем ответственность за поддержание актуальности этих версий**, даже несмотря на то, что они являются зависимыми компонентами **flask**. Что если в **Werkzeug == 0.14.1** обнаружена дыра в безопасности, которую разработчики пакетов сразу же исправили в **Werkzeug == 0.14.2** ? Теперь нам нужно обновиться до Werkzeug == 0.14.2, чтобы избежать каких-либо проблем безопасности, возникающих в более ранней версии.

Проблема в том что, мы должны как то узнать, что есть проблема с версией, которая у нас установлена. Затем нам нужно заняться установкой новой версию в нашей производственной среде самим. Но это не то что нам нужно. Правда в том, что нам действительно все равно, какая версия **Werkzeug** установлена у нас, если она не нарушает наш код. Мы не хотим на себя брать контроль над всеми зависимостями проекта. Нам нужно чтобы последнии версии зависимостей гарантировали, что мы автоматически получаем все необходимые исправления ошибок, исправления безопасности, новые функции, улучшенную оптимизации и так далее.

Venv

virtualenv — это инструмент для создания изолированной среды Python. У такой среды есть отдельна установка python, при ее использовании загруженные библиотеки недоступны другим. Можно сделать так, чтобы у этой среды не было доступа к глобальным библиотекам.

Virtualenv — простой и рекомендованный способ настройки среды Python.

## Отличия virtualenv и venv

Venv — это пакет, который идет по умолчанию с Python 3.3+. В версии Python 2 его нет.

Virtualenv — более продвинутая библиотека. По [**ссылке**](https://virtualenv.pypa.io/en/stable/) можно ознакомиться с основными отличиями.

Виртуальную среду можно создать и с помощью venv, но все-таки рекомендуется установить и использовать virtualenv для полноценной работы.

## Установка virtualenv с помощью pip

Для установки virtualenv с Python нужно использовать pip. Желательно предварительно обновить этот инструмент.

python -m pip install --upgrade pip

Или используйте python3, если python ссылается на 2 версию.

После обновления можно установить и virtualenv:

pip install virtualenv # или pip3

## Создание виртуальной среды

1. Перейдите в директорию, в которой вы хотите создать виртуальную среду(например папка проекта).

~$ cd test

~/test$

2. Создайте виртуальную среду, при необходимости указав конкретную версию Python. Следующая команда создает virtualenv с именем venv\_name и использует флаг -p, чтобы указать полный путь к версии Python:

virtualenv -p /usr/bin/python venv\_name

# или

virtualenv venv\_name

Назвать среду можно как угодно

После выполнения команды вы увидите логи:

Running virtualenv with interpreter /usr/bin/python

Already using interpreter /usr/bin/python

Using base prefix '/usr'

New python executable in /home/username/test/venv\_name/bin/python

Installing setuptools, pip, wheel...

done.

Эта команда создает локальную копию среды. Работая с ней, важно не забывать об активации, чтобы использовались нужные версии конкретных инструментов и пакетов.

Если при установке возникла ошибка setuptools pip failed with error code 1` error, используйте следующую команду, чтобы решить проблему:

pip install --upgrade setuptools # или pip3

3. Для активации новой виртуальной среды используйте команду:

source venv\_name/bin/activate

После этого название текущей среды отобразится слева от символа ввода: (venv\_name) username@desctop:~/test$

Теперь при установке любого пакета с помощью pip он будет размещаться в папках этой среды, изолированно от глобальной установки.

После завершения работы деактивировать виртуальную среду можно с помощью команды deactivate.

Введите ее и приставка venv\_name пропадет. Вы вернетесь к использованию глобально версии python.

## Удаление виртуальной среды

Для удаления виртуальной среды достаточно просто удалить папку проекта. Для этого используется следующая команда:

rm -rf venv\_name

## Решение популярных ошибок

**Ошибки при создании virtualenv**. При попытке создать virtualenv с Python 3.7 могут возникнуть следующие ошибки.

AttributeError: module 'importlib.\_bootstrap' has no attribute 'SourceFileLoader'

OSError: Command /home/username/venv/bin/python3 -c "import sys, pip; sys...d\"] + sys.argv[1:]))" setuptools pip failed with error code 1

Для их исправления нужно добавить следующую строку в .bash\_profile.

export LC\_ALL="en\_US.UTF-8"

**Использование полного пути к виртуальной среде**. Может быть такое, что при использовании команды virtualenv будет использована не та версия. Для решения проблемы нужно лишь задать полные пути как к virtualenv, так и к Python в системе.

А получить их можно с помощью этой команды:

/home/username/opt/python-3.8.0/bin/virtualenv -p /home/username/opt/python-3.8.0/bin/python3 venv