Лабораторная работа 2.14 Установка пакетов в Python. Виртуальные окружения

Цель работы: *приобретение навыков по работе с менеджером пакетов pip и виртуальными окружениями с помощью языка программирования Python версии 3.x.*

Ход работы

Где взять отсутствующий пакет?

Необходимость в установке дополнительного пакета возникнет очень быстро, если вы решите поработать над задачей, за рамками базового функционала, который предоставляет *Python*. Например: работа с *web*, обработка изображений, криптография и т.п. В этом случае, необходимо узнать, какой пакет содержит функционал, который вам необходим, найти его, скачать, разместить в нужном каталоге и начать использовать. Все эти действия можно сделать вручную, но этот процесс поддается автоматизации. К тому же скачивать пакеты с неизвестных сайтов может быть довольно опасно.

К счастью для нас, в рамках *Python,* все эти задачи решены. Существует так называемый *Python Package Index (PyPI)* – это репозиторий, открытый для всех *Python* разработчиков, в нем вы можете найти пакеты для решения практически любых задач. Там также есть возможность выкладывать свои пакеты. Для скачивания и установки используется специальная утилита, которая называется *pip*.

Менеджер пакетов в Python – pip

Pip – это консольная утилита (без графического интерфейса). После того, как вы ее скачаете и установите, она пропишется в *PATH* и будет доступна для использования.

Эту утилиту можно запускать как самостоятельно:

```
$ pip <аргументы>
```

так и через интерпретатор *Python*:

```
$ python -m pip <аргументы>
```

Ключ *-m* означает, что мы хотим запустить модуль (в данном случае *pip*). Более подробно о том, как использовать *pip*, вы сможете прочитать ниже.

Установка pip

При развертывании современной версии *Python* (начиная с Python 2.7.9 и Python 3.4), *pip* устанавливается автоматически. Но если, по какой-то причине, *pip* не установлен на вашем ПК, то сделать это можно вручную. Существует несколько способов.

Универсальный способ

Будем считать, что *Python* у вас уже установлен, теперь необходимо установить *pip*. Для того, чтобы это сделать, скачайте скрипт *get-pip.py*

```
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
```

и выполните его.

```
$ python get-pip.py
```

При этом, вместе с *pip* будут установлены *setuptools* и *wheels*. *Setuptools* – это набор инструментов для построения пакетов *Python*. *Wheels* – это формат дистрибутива для пакета *Python*. Обсуждение этих составляющих выходит за рамки урока, поэтому мы не будем на них останавливаться.

Способ для *Linux*

Если вы используете *Linux*, то для установки *pip* можно воспользоваться имеющимся в вашем дистрибутиве пакетным менеджером. Ниже будут перечислены команды для ряда *Linux* систем, запускающие установку *pip* (будем рассматривать только *Python* 3, т.к. *Python* 2 уже морально устарел, а его поддержка и развитие будут прекращены после 2020 года).

Fedora

```
$ sudo dnf install python3 python3-wheel
```

openSUSE

```
$ sudo zypper install python3-pip python3-setuptools python3-wheel
```

Debian/Ubuntu

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

Arch Linux

```
$ sudo pacman -S python-pip
```

Обновление pip

Если вы работаете с *Linux*, то для обновления *pip* запустите следующую команду.

```
$ pip install -U pip
```

Для *Windows* команда будет следующей:

```
> python -m pip install -U pip
```

Использование pip

Далее рассмотрим основные варианты использования *pip*: установка пакетов, удаление и обновление пакетов.

Установка пакета

Pip позволяет установить самую последнюю версию пакета, конкретную версию или воспользоваться логическим выражением, через которое можно определить, что вам, например, нужна версия не ниже указанной. Также есть поддержка установки пакетов из репозитория. Рассмотрим, как использовать эти варианты.

Установка последней версии пакета

```
$ pip install ProjectName
```

Установка определенной версии

```
$ pip install ProjectName==3.2
```

Установка пакета с версией не ниже 3.1

```
$ pip install ProjectName>=3.1
```

Установка *Python* пакета из git репозитория

```
$ pip install -e git+https://gitrepo.com/ProjectName.git
```

Установка из альтернативного индекса

```
$ pip install --index-url http://pypackage.com/ ProjectName
```

Установка пакета из локальной директории

```
$ pip install ./dist/ProjectName.tar.gz
```

Удаление пакета

Для того, чтобы удалить пакет воспользуйтесь командой

```
$ pip uninstall ProjectName
```

Обновление пакетов

Для обновления пакета используйте ключ *–upgrade.*

```
$ pip install --upgrade ProjectName
```

Просмотр установленных пакетов

Для вывода списка всех установленных пакетов применяется команда *pip list*.

```
$ pip list
```

Если вы хотите получить более подробную информацию о конкретном пакете, то используйте аргумент *show*.

```
$ pip show ProjectName
```

Поиск пакета в репозитории

Если вы не знаете точное название пакета, или хотите посмотреть на пакеты, содержащие конкретное слово, то вы можете это сделать, используя аргумент *search*.

```
$ pip search "test"
```

Проблемы окружения приложений на Python

В системе для интерпретатора Python может быть установлена глобально только одна версия пакета. Это порождает ряд проблем.

1. Проблема обратной совместимости

Некоторые операционные системы, например, Linux и MacOs используют содержащиеся в них предустановленные интерпретаторы Python. Обновив или изменив самостоятельно версию какого-то установленного глобально пакета мы можем непреднамеренно сломать работу утилит и приложений из дистрибутива операционной системы.

Чем опасно обновление пакетов или версий интерпретатора? В новой версии пакета могут измениться названия функций или методов объектов и число и/или порядок передаваемых в них параметров. В следующей версии интерпретатора могут появиться новые ключевые слова, которые совпадают с именами переменных уже существующих приложений.

Эта же проблема затрагивает и процесс разработки. Обычно программист работает со множеством проектов, так как приходится поддерживать созданные ранее и не редко даже унаследованные от других: когда-то веб приложения создавались для одной версии фреймворка, а сегодня уже вышла новая, но переписывать все долго и дорого, поэтому проект и дальше поддерживается для старой.

2. Проблема коллективной разработки

Если разработчик работает над проектом не один, а с командой, ему нужно передавать и получать список зависимостей, а также обновлять их на своем компьютере таким образом, чтобы не нарушалась работа других его проектов. Значит нам нужен механизм, который вместе с обменом проектами быстро устанавливал бы локально и все необходимые для них пакеты, при этом не мешая работе других проектов.

Виртуальное окружение как решение

Если вы уже сталкивались с этой проблемой, то уже задумались, что для каждого проекта нужна своя "песочница", которая изолирует зависимости. Такая "песочница" придумана и называется "виртуальным окружением" или "виртуальной средой". Но прежде, чем мы научимся работать в виртуальном, нужно разобраться с реальным окружением.

Идея виртуального окружения родилась раньше, чем была реализована стандартными средствами Python. Попыток было несколько, но в основу PEP 405 легла утилита virtualenv Яна Бикинга. Были проанализированы возникающие при работе с ней проблемы. После этого в работу интерпретатора Python версии 3.3 добавили их решения. Так был создан встроенный в Python модуль venv, а утилита virtualenv теперь дополнительно использует в своей работе и его.

Как работает виртуальное окружение? Ничего сверхъестественного. В отдельной папке создаётся **неполная копия** выбранной установки Python. Эта копия является просто набором файлов (например, интерпретатора или ссылки на него), утилит для работы с собой и нескольких пакетов (в том числе pip). Стандартные пакеты при этом не копируются.

Чтобы начать работать с виртуальным окружением его нужно активировать. Активация меняет для сессии командной строки системную переменную PATH, добавляя папку копии интерпретатора в неё первой. Поэтому при вызове команды python находится первой его копия из виртуального окружения. Далее сам интерпретатор по понятному ему правилу создает список путей поиска пакетов от места его запуска. При этом он учитывает место расположения стандартных пакетов своей первоначальной установки, что решает проблему доступа к ним. Папка сторонних пакетов добавляется из виртуального окружения. Теперь мы можем работать с её содержимым используя стандартную утилиту pip.

После окончания работы в виртуальном окружении команда деактивации позволяет убрать из PATH путь к интерпретатору виртуального окружения. Стандартный интерпретатор Python опять становится доступен в сессии командной строки.

Вот основные этапы работы с виртуальным окружением:

1. **Создаём** через утилиту новое виртуальное окружение в отдельной папке для выбранной версии интерпретатора Python.
2. **Активируем** ранее созданное виртуального окружения для работы.
3. **Работаем** в виртуальном окружении, а именно управляем пакетами используя pip и запускаем выполнение кода.
4. **Деактивируем** после окончания работы виртуальное окружение.
5. **Удаляем папку** с виртуальным окружением, если оно нам больше не нужно.

Среда разработки PyCharm при создании нового проекта автоматически создает для него виртуальное окружение. Это удобно и избавляет вас от использования большинства команд из этого топика. Тем не менее, мы считаем работу с виртуальным окружением в “ручном” режиме полезным навыком.

Давайте пройдем каждый этап по отдельности со стандартной реализацией виртуального окружения Python через модуль venv.

Виртуальное окружение с venv

Стандартный модуль venv, был добавлен с версии Python 3.3. И первая ошибка, которая возникает - это попытка его использовать с версией Python 2.x, путь даже если она была выпущена после Python 3.3.

Руководство по работе с venv https://docs.python.org/3/tutorial/venv.html

Создание виртуального окружения

Для создания виртуального окружения достаточно дать команду в формате:

```
python3 -m venv <путь к папке виртуального окружения>
```

Обычно папку для виртуального окружения называют env или venv.

В описании команды выше явно указан интерпретатор версии 3.x. Под Windows и некоторыми другими операционными системами это будет просто python.

Если на Linux получена ошибка об отсутствии пакета venv, то его нужно установить, например, под Debian так:

```
sudo apt-get install python3-venv
```

Создадим виртуальное окружение в папке проекта. Для этого перейдём в корень любого проекта на Python >= 3.3 и дадим команду:

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

После её выполнения создастся папка env с виртуальным окружением. Вот пример структуры такой папки для Windows (скрыты файлы пакетов и папки кэширования Python):

```
└───env
    |   pyvenv.cfg
    |
    ├───Include
    ├───Lib
    |   └───site-packages
    |       |
    |       ├───pip
    |       ├───pip-19.2.3.dist-info
    |       ├───pkg_resources
    |       ├───setuptools
    |       ├───setuptools-41.2.0.dist-info
    |
    └───Scripts
            activate
            activate.bat
            Activate.ps1
            deactivate.bat
            easy_install-3.7.exe
            easy_install.exe
            pip.exe
            pip3.7.exe
            pip3.exe
            python.exe
            pythonw.exe
```

В Linux и macOS папки Lib и Scripts изменятся на lib и bin.

Активация

Чтобы активировать окружение под Linux и macOS нам нужно дать команду:

```
$ source env/bin/activate
```

Чтобы активировать виртуальное окружение под Windows команда выглядит иначе:

```
> env\\Scripts\\activate
Просто под Windows мы вызываем скрипт активации напрямую.
```

После активации приглашение консоли изменится. В его начале в круглых скобках будет отображаться имя папки с виртуальным окружением, например, возможный вариант для Linux:

```
(env) user@user:~/venv_test/proj3$
```

При размещении виртуального окружения в папке проекта стоит позаботится об его исключении из репозитория системы управления версиями. Для этого, например, при использовании Git нужно добавить папку в файл .gitignore. Это делается для того, чтобы не засорять проект разными вариантами виртуального окружения.

```
$ python3 -m venv /home/user/envs/project1_env
```

Виртуальное окружение благополучно создалось. Давайте активируем его:

```
$ source /home/user/envs/project1_env/bin/activate
```

Виртуальное окружение активировано.

Деактивация

Чтобы переключиться с одного окружения на другое нам нужно выполнить команду деактивации и команду активации другого виртуального окружения, например, так:

```
$ deactivate
$ source /home/user/envs/project1_env2/bin/activate
```

Команда `deactivate` всегда выполняется из контекста текущего виртуального окружения. По этой причине для неё не нужно указывать полный путь.

Ранее мы работали с интерпретаторами доступными через стандартные вызовы python. Если в системе установлена другая версия интерпретатора, которая не добавлена в переменную PATH, то для создания виртуального окружения нужно явно задать путь до исполняемого файла интерпретатора. Например, у нас Windows и ранее мы установили Python версии 3.7.6. Сегодня мы решили ознакомится с особенностями Python версии 3.8.2, так как нужно расти в качестве разработчика. Для этого скачали и установили новую версию в папку C:\Python38, но при этом не добавили в PATH (не установили соответствующую галочку в программе установки). Для создания виртуального окружения для Python 3.8.2 нам придется дать в папке проекта команду с указанием полного пути до интерпретатора, например:

```
> C:\\Python38\\python -m venv env
```

После выполнения команды мы получим виртуальное окружение для Python 3.8.2. Активация же будет происходить стандартно для Windows:

```
> env\\Scripts\\activate
```

Виртуальное окружение с virtualenv

Зачем нам нужно уметь работать с утилитой virtualenv? Ведь мы уже научились работать со стандартным модулем Python venv. Просто он очень распространён и поддерживает большее число вариантов и версий интерпретатора Python, например, PyPy и CPython.

Для начала пакет нужно установить. Установку можно выполнить командой:

```
# Для python 3
python3 -m pip install virtualenv

# Для единственного python
python -m pip install virtualenv
```

Создание виртуального окружения с утилитой virtualenv отличается от стандартного. Например, создание в текущей папке виртуального окружения для интерпретатора доступного через команду python3 с названием папки окружения env:

```
virtualenv -p python3 env
```

Активация и деактивация такая же, как у стандартной утилиты Python. Например, для Linux и macOS:

```
$ source env/bin/activate

(env) $ deactivate
```

Для операционной системы Windows:

```
> env\\Scripts\\activate

(env) > deactivate
```

С созданием, активацией и деактивацией виртуального окружения разобрались. А как обмениваться и поднимать чужое виртуальное окружение?

Перенос виртуального окружения

Само виртуальное окружение никуда переносить не нужно. Требуется возможность формирования и развертывания пакетных зависимостей. Для формирования и развертывания пакетных зависимостей используется утилита pip.

Просмотреть список зависимостей мы можем командой:

```
pip freeze
```

Что бы его сохранить, нужно перенаправить вывод команды в файл:

```
pip freeze > requirements.txt
```

Имя файла хранения зависимостей `requirements.txt` выбрано не зря. Оно является стандартной договоренностью и используется некоторыми утилитами автоматически.

Установка пакетов из файла зависимостей в новом виртуальном окружении так же выполняется одной командой:

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

Давайте рассмотрим строку выше в деталях. `freeze` - команда, используемая для получения всех установленных пакетов в формате требований. Таким образом, все пакеты, которые вы установили перед выполнением команды и предположительно использовали в каком-либо проекте, будут перечислены в файле с именем «requirements.txt». Кроме того, будут указаны их точные версии (см. Requirement Specifiers https://pip.pypa.io/en/stable/reference/pip_install/#requirement-specifiers).

Рассмотрим пример вывода команды `freeze`:

```
beautifulsoup4==4.7.1
nltk==3.4.1
numpy==1.16.3
scikit-learn==0.21.1
scipy==1.3.0
```

Важно понимать, что команда `freeze` на самом деле касается всех установленных библиотек, что не так часто используется на деле и может считаться плохой практикой. По этой причине мы рекомендуем использовать более осознанный подход, который заключается в том, чтобы самостоятельно пересмотреть полученный файл.

Управление пакетами с помощью conda

Все чаще среди Python-разработчиков заходит речь о менеджере пакетов conda, включенный в состав дистрибутивов Anaconda и Miniconda. JetBrains включил этот инструмент в состав PyCharm. Давайте разберемся, чем conda лучше pip и что представляет собой рабочий процесс с conda.

Чем conda лучше pip?

Основная проблема заключается в том, что `pip`, `easy_install` и `virtualenv` ориентированы на Python. Эти инструменты игнорируют библиотеки зависимостей, реализованные с использованием других языков. Например, `XSLT`, `HDF5`, `MKL` и другие, которые не имеют `setup.py` в исходном коде и не устанавливают файлы в директорию `site-packages`.

Conda же способна управлять пакетами как для Python, так и для C/ C++, R, Ruby, Lua, Scala и других. Conda устанавливает двоичные файлы, поэтому работу по компиляции пакета самостоятельно выполнять не требуется (по сравнению с pip).

Существуют также некоторые различия, если вы заинтересованы в создании собственных пакетов. Например, pip создан на основе setuptools, тогда как conda использует свой собственный формат, который имеет некоторые преимущества (например, статическая компиляция пакета).

Виртуальное окружение с conda

1. Начиная проект, создайте чистую директорию и дайте ей понятное короткое имя. Для Linux это будет соответствовать набору команд:

```
mkdir $PROJ_NAME
cd $PROJ_NAME
touch README.md main.py
```

Для Windows, если использьзуется дистрибутив Anaconda, то необходимо вначале запустить консоль Anaconda Powershell Prompt. Делается это из системного меню, посредством выбора следующих пунктов: *Пуск* → *Anaconda3 (64-bit)* → *Anaconda Powershell Prompt (Anaconda3)*. В результате будет отображено окно консоли, показанное на рисунке.

Обратите на имя виртуального окружения по умолчанию, которым в данном случае является *base*. В этом окне необходимо ввести следующую последовательность команд:

```
mkdir %PROJ_NAME%
cd %PROJ_NAME%
copy NUL > main.py
```

Здесь `PROJ_NAME` - это переменная окружения, в которую записано имя проекта. Допускается не использовать переменные окружения, а использовать имя проекта вместо `$PROJ_NAME` или `%PROJ_NAME%`.

2. Создайте чистое conda-окружение с таким же именем, как директория проекта, и затем активируйте его. Для Linux это последовательность команд:

```
source deactivate
conda create -n $PROJ_NAME python=3.7
source activate $PROJ_NAME
```

Тогда как для Windows эта последовательность будет несколько иной:

```
conda create -n %PROJ_NAME% python=3.7
conda activate %PROJ_NAME%
```

После выполнения этих команд в приглашении ввода должно отобразиться имя созданного виртуального окружения.

3. Установите пакеты, необходимые для реализации проекта.

```
conda install django, pandas
```

4. Периодически экспортируйте параметры окружения. Экспортируйте после установки, перед каждым большим или маленьким коммитом:

```
conda env export > environment.yml
```

5. Затем начните разработку. Как только понадобятся новые пакеты, вы можете выполнить `conda install` и `conda env export`. Когда пришло время прекратить разработку или

переключиться на новый проект, отключите среду. Для Linux это возможно с помощью команды:

```
source deactivate
```

Для Windows необходимо использовать следующую команду:

```
conda deactivate
```

Если вы хотите удалить только что созданное окружение, выполните:

```
conda remove -n $PROJ_NAME
```

6. Файл `environment.yml` позволит воссоздать окружение в любой нужный момент. Достаточно набрать:

```
conda env create -f environment.yml
```

Это краткое вступление, но достаточное для того, чтобы начать использовать conda. С подробной информацией всегда можно ознакомиться в документации https://conda.io/.

Указания по технике безопасности

При работе на ЭВМ без разрешения руководителя занятия запрещается:

- подавать (снимать) напряжение на ПЭВМ и электрические розетки с распределительного щита;
- включать и выключать блоки питания ПЭВМ и мониторы;
- извлекать ПЭВМ из защитного кожуха;
- устранять неисправности, возникшие в ходе выполнения лабораторной работы.

Методика и порядок выполнения работы

1. Изучить теоретический материал работы.
2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.
3. Выполните клонирование созданного репозитория.
4. Организуйте свой ре+позиторий в соответствие с моделью ветвления git-flow.
5. Создайте виртуальное окружение Anaconda с именем репозитория.
6. Установите в виртуальное окружение следующие пакеты: pip, NumPy, Pandas, SciPy.
7. Попробуйте установить менеджером пакетов conda пакет TensorFlow. Возникает ли при этом ошибка? Попробуйте выявить и укажите причину этой ошибки.
8. Попробуйте установить пакет TensorFlow с помощью менеджера пакетов pip.
9. Сформируйте файлы `requirements.txt` и `environment.yml`. Проанализируйте содержимое этих файлов.
10. Зафиксируйте сделанные изменения в репозитории.
11. Добавьте отчет по лабораторной работе в *формате PDF* в папку *doc* репозитория. Зафиксируйте изменения.
12. Выполните слияние ветки для разработки с веткой *master/main*.
13. Отправьте сделанные изменения на сервер GitHub.
14. Отправьте адрес репозитория GitHub на электронный адрес преподавателя.

Содержание отчета и его форма

Отчет по лабораторной работе оформляется электронно в формате PDF, должен содержать ответы на контрольные вопросы, ссылку на репозиторий с которым выполнялась работа, скриншоты терминала, отображающего процесс установки пакетов, содержимое файлов `requirements.txt` и `environment.yml`.

Вопросы для защиты работы

1. Каким способом можно установить пакет Python, не входящий в стандартную библиотеку?
2. Как осуществить установку менеджера пакетов pip?
3. Откуда менеджер пакетов pip по умолчанию устанавливает пакеты?
4. Как установить последнюю версию пакета с помощью pip?
5. Как установить заданную версию пакета с помощью pip?
6. Как установить пакет из git репозитория (в том числе GitHub) с помощью pip?
7. Как установить пакет из локальной директории с помощью pip?
8. Как удалить установленный пакет с помощью pip?
9. Как обновить установленный пакет с помощью pip?
10. Как отобразить список установленных пакетов с помощью pip?
11. Каковы причины появления виртуальных окружений в языке Python?
12. Каковы основные этапы работы с виртуальными окружениями?
13. Как осуществляется работа с виртуальными окружениями с помощью venv?
14. Как осуществляется работа с виртуальными окружениями с помощью virtualenv?
15. Изучите работу с виртуальными окружениями pipenv. Как осуществляется работа с виртуальными окружениями pipenv?
16. Каково назначение файла `requirements.txt`? Как создать этот файл? Какой он имеет формат?
17. В чем преимущества пакетного менеджера conda по сравнению с пакетным менеджером pip?
18. В какие дистрибутивы Python входит пакетный менеджер conda?
19. Как создать виртуальное окружение conda?
20. Как активировать и установить пакеты в виртуальное окружение conda?
21. Как деактивировать и удалить виртуальное окружение conda?
22. Каково назначение файла `environment.yml`? Как создать этот файл?
23. Как создать виртуальное окружение conda с помощью файла `environment.yml`?
24. Самостоятельно изучите средства IDE PyCharm для работы с виртуальными окружениями conda. Опишите порядок работы с виртуальными окружениями conda в IDE PyCharm.
25. Почему файлы `requirements.txt` и `environment.yml` должны храниться в репозитории git?