

Изолированное окружение. Пакеты, модули





Модули и пакеты

Система модулей позволяет вам логически организовать ваш код на Python. Группирование кода в модули значительно облегчает процесс написания и понимания программы. Говоря простым языком, модуль в Python это **просто файл**, содержащий код на Python. Каждый модуль в Python может содержать **переменные, объявления классов и функций**. Кроме того, в модуле может находиться **исполняемый код**.



Команда **import**

Вы можете использовать любой питоновский файл как модуль в другом файле, выполнив в нем команду `import`. Команда `import` в Python обладает следующим синтаксисом:

```
import module_1[, module_2[, ... module_N]
```

Когда интерпретатор Python встречает команду `import`, он импортирует этот модуль, если он присутствует в пути поиска Python. Путь поиска Python это список директорий, в которых интерпретатор производит поиск перед попыткой загрузить модуль.



Например, чтобы использовать модуль `math` следует написать:

```
import math
```

```
# Используем функцию sqrt из модуля math
```

```
print(math.sqrt(9))
```

```
# Печатаем значение переменной pi, определенной в math
```

```
print(math.pi)
```

Важно знать! Модуль загружается лишь однажды, независимо от того, сколько раз он был импортирован. Это препятствует циклическому выполнению содержимого модуля.



from *module* **import** *var*

Выражение `from ... import ...` не импортирует весь модуль, а только предоставляет доступ к конкретным объектам, которые мы указали.

```
# Импортируем из модуля math функцию sqrt
from math import sqrt
# Выводим результат выполнения функции sqrt.
# Обратите внимание, что нам больше нечем указывать
# имя модуля
print (sqrt(144))
# Но мы уже не можем получить из модуля то, что не
# импортировали !!!
print (pi) # Выдаст ошибку
```



from *module* **import** *var, func, class*

Импортировать из модуля объекты можно через запятую.

```
from math import pi, sqrt  
print(sqrt(121))  
print(pi)  
print(e)
```



from *module* **import ***

В Python так же возможно импортировать всё (переменные, функции, классы) за раз из модуля, для этого используется конструкция **from ... import ***

```
from math import *
```

```
# Теперь у нас есть доступ ко всем функция и  
переменным, определенным в модуле math
```

```
print(sqrt(121))
```

```
print(pi)
```

```
print(e)
```

Не импортируются объекты с ***_var***

Повторяющиеся названия перезаписываются. Такое поведение нужно отслеживать при импорте нескольких модулей.



```
import module_1 [ ,module_2 ]
```

За один раз можно импортировать сразу несколько модулей, для этого их нужно перечислить через запятую после слова `import`

```
import math, os  
print (math.sqrt (121))  
print (os.env)
```



import *module* **as** *my_alias*

Если вы хотите задать псевдоним для модуля в вашей программе, можно воспользоваться вот таким синтаксисом

```
import math as matan  
print (matan.sqrt (121))
```




Местонахождение модулей в Python

Когда вы импортируете модуль, интерпретатор Python ищет этот модуль в следующих местах:

- Директория, в которой находится файл, в котором вызывается команда импорта
- Если модуль не найден, Python ищет в каждой директории, определенной в консольной переменной **PYTHONPATH**.
- Если и там модуль не найден, Python проверяет путь заданный по умолчанию

Путь поиска модулей сохранен в системном модуле `sys` в переменной `path`. Переменная `sys.path` содержит все три вышеописанных места поиска модулей.



Получение списка всех модулей Python установленных на компьютере

Для того, чтобы получить список всех модулей,
установленных на вашем компьютере достаточно
выполнить команду:

```
>>>help("modules")
```

Через несколько секунд вы получите список всех доступных
модулей.



Создание своего модуля в Python

Чтобы создать свой модуль в Python достаточно сохранить ваш скрипт с расширением `.py`. Теперь он доступен в любом другом файле. Например, создадим два файла: `module_1.py` и `module_2.py` и сохраним их в одной директории. В первом запишем:

```
# module_1.py
def hello():
    print("Hello from module_1")
```

А во втором вызовем эту функцию:

```
# module_2.py
from module_1 import hello
hello()
```




Пакеты модулей в Python

Отдельные файлы-модули с кодом на Python могут объединяться в пакеты модулей. Пакет это директория (папка), содержащая несколько отдельных файлов-скриптов.

Например, имеем следующую структуру:

my_file.py

my_package

__init__.py

inside_file.py

В файле `inside_file.py` определена некая функция `foo`. Тогда чтобы получить доступ к функции `foo`, в файле `my_file` следует выполнить следующий код:

```
from my_package.inside_file import foo
```



Функция **dir()**

Встроенная функция `dir()` возвращает отсортированный список строк, содержащих все имена, определенные в модуле.

```
# на данный момент нам доступны лишь встроенные функции
```

```
dir()
```

```
# импортируем модуль math
```

```
import math
```

```
# теперь модуль math в списке доступных имен
```

```
dir()
```

```
# получим имена, определенные в модуле math
```

```
dir(math)
```

Менеджер пакетов **pip** (Python Package Index)





Установка **pip**

Начиная с Python версии 3.4, **pip** поставляется вместе с интерпретатором Python. Метод универсален и подходит для любой операционной системы, если в ней уже установлена какая-либо версия Python

Открыть консоль (терминал)

Скачать файл `get-pip.py`:

```
wget https://bootstrap.pypa.io/get-pip.py
```

Установить **pip**:

```
python3 get-pip.py
```



Использование **pip**

Самый распространённый способ использования **pip** - это через консоль (терминал). Чтобы использовать **pip**, в консоли нужно вызвать команду **pip** для Python2 или **pip3** для Python3. Для того, чтобы узнать какие команды есть в **pip** нужно вызвать **pip3 -help**:

Usage:

pip3 <command> [options]

Commands:

install	Install packages.
download	Download packages.
uninstall	Uninstall packages.
freeze	Output installed packages in requirements format.
list	List installed packages.
show	Show information about installed packages.
check	Verify installed packages have compatible dependencies.
config	Manage local and global configuration.
search	Search PyPI for packages.
wheel	Build wheels from your requirements.
hash	Compute hashes of package archives.
completion	A helper command used for command completion.
help	Show help for commands.



install

Команда `install` позволяет установить какой-либо пакет.

```
pip3 install Flask==2.1
```

После Flask мы также указали версию пакета, которую мы хотим установить. Это необязательно, если мы не укажем версию, то установится самая последняя версия пакета, которая присутствует в репозитории.

```
pip3 install Flask
```

Также можно указывать ограничения на версии, к примеру, что хотим установить Django не старше версии

```
pip3 install Flask > 2.1
```



pip install -r *reqfile.txt*

Установка пакетов перечисленных в файле

```
pip3 install -r requirements.txt
```

Файл requirements.txt

```
Flask==2.0.2
```

```
Flask-JWT-Extended==4.3.1
```

```
Flask-RESTful==0.3.9
```

```
Flask-SQLAlchemy==2.5.1
```

```
passlib==1.7.4
```

```
pymongo==4.0.1
```

```
Werkzeug==2.0.2
```

Установленные пакеты будут храниться в папке
`/python3.X/site-packages`



Импортирование в скрипте

После установки пакета его можно импортировать в скрипт.

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```



Понижение версии `--force-reinstall`

А если пакет уже установлен и вы хотите понизить его версию добавьте `--force-reinstall` вот так:

```
pip install 'stevedore>=1.3.0,<1.4.0' --force-reinstall
```



Проблемы **--no-cache-dir**

Иногда ранее установленная версия кэшируется.

При установке новой указанной версии пакета

```
pip install pillow==5.2.0
```

pip возвращает следующее:

```
Требование уже выполнено: pillow==5.2.0 in  
/home/ubuntu/anaconda3/lib/python3.6/site-packages (5.2.0)
```

Мы можем использовать `--no-cache-dir` вместе с `-I`, чтобы перезаписать это

```
pip install --no-cache-dir -I pillow==5.2.0
```




uninstall - удаление пакета

Удаление установленного пакета

```
pip uninstall Flask
```

Удаление пакетов перечисленных в файле

```
pip uninstall -r requirements.txt
```

Удаление всех установленных пактов

```
pip freeze | xargs pip uninstall -y
```



download – *закачка без установки*

Позволяет скачать пакеты без установки.

```
pip3 download Flask
```

Пакеты скачиваются с зависимостями и имеют расширения .whl. Установить их в проект можно через install так:

```
pip install --find-links=/download Flask-2.1.1-py3-  
none-any.whl
```



pip list

Позволяет просмотреть список всех установленных в системе пакетов.

Пример:

```
pip3 list
```



pip show

Позволяет просмотреть информацию об установленном в системе пакете.

Пример:

```
pip3 show Flask
```

В дополнение к pip show есть пакет



Virtualenv

Когда программист Python работает над большим количеством проектов, со временем у него появляется потребность в том, чтобы не устанавливать все пакеты из всех проектов себе в систему, а разделить их друг от друга. Для решения этой проблемы был создан Virtualenv.

virtualenv — программа для создания и управления окружениями Python. Позволяет создать среду со своими отдельными модулями, настройками и программами. Среда ограничивается рамками одного каталога. Очень удобна для работы с различными версиями одних и тех же модулей, для создания проектов, у которых "всё с собой", которые не зависят от операционной системы.



Установка virtualenv

Установить virtualenv можно через менеджер пакетов pip.

```
pip3 install virtualenv
```



virtualenv [OPTIONS] DEST_DIR

Обязательным параметром является только DEST_DIR - путь к папке, в которой будет храниться виртуальное окружение.

Название папки лучше называть по имени проекта.

Пример:

```
virtualenv course
```



Версия Python

Вы можете указать нужную вам версию интерпретатора python, при этом он должен быть установлен в системе. Если вы опустили эту опцию, то будет использоваться умолчательный (Выполните `which python` чтобы узнать какой он у вас, но скорее всего это будет `/usr/bin/python`).

Пример:

```
virtualenv --python=python3.6 course
```



site-packages

Запретить использование системного site-packages (для полной изоляции вашего окружения от системы). Например у вас в системе установлена "Flask 2.1", если вы будете использовать эту опцию, то в созданном окружении эта "Flask" не будет доступна.

Пример:

```
virtualenv --no-site-packages course
```



--system-site-packages

Эта опция противоположна предыдущей, то есть заставляет окружение использовать установленные в системе пакеты, если не нашлись онные в окружении.

Пример:

```
virtualenv --system-site-packages course
```



--clear

Используется для очистки существующего окружения от пакетов и прочих изменений.

Пример:

```
virtualenv --clear existing_venv
```



Использование `virtualenv`

После создания виртуальное окружение надо запустить, иначе будет использоваться ваше системное окружение для Python

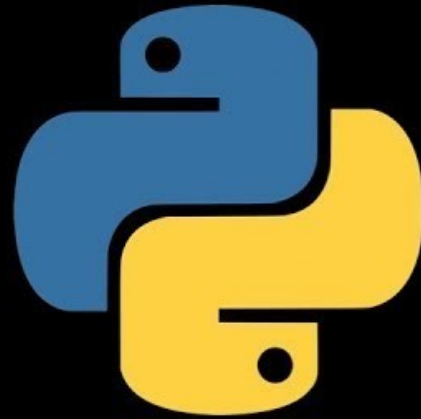
```
source venv/bin/activate
```

Когда активируется виртуальное окружение, строка приглашения к вводу в консоле (терминале) заменится на название виртуального окружения . После этого можно свободно пользоваться виртуальной средой, запускать модули `python` и пользоваться пакетами, установленными в виртуальном окружении. Чтобы выйти из виртуальной среды, нужно ввести:

```
deactivate
```



Спасибо за внимание!



PYTHON

PROGRAMMING