

DS_PROD-2. Воспроизводимость и контейнеризация приложений

1. Введение

В предыдущем модуле мы рассмотрели базовые принципы работы с моделями на этапе продакшена.

МЫ НАУЧИЛИСЬ:

- сохранять обученные модели в виде бинарных файлов, а также файлов других форматов, которые поддерживаются языками программирования, отличными от Python;
- локально разворачивать собственный веб-сервис с помощью фреймворка Flask, встраивая в его работу свою модель машинного обучения;
- оптимизировать работу этого веб-сервиса, увеличивая его пропускную способность с помощью таких инструментов, как uWSGI и NGINX.

Однако, как мы уже упомянули в конце прошлого модуля, осталась одна проблема: наш сервис (да и сама модель) работает только на нашем компьютере. Когда мы зальём исходный код на GitHub, а наш коллега Василий склонирует себе репозиторий и попытается запустить сервис на своём «боевом» сервере, нет гарантии, что приложение запустится.

Что, если на сервере Василия нет необходимых зависимостей, таких как Scikit Learn, Flask и так далее? Установить библиотеки несложно, но какие именно их версии нужны Василию? А что, если у него уже стоят некоторые библиотеки, которые вы использовали, но в совершенно другой версии, которая не совместима с вашим приложением?

Кроме того, для обеспечения работы сервиса в связке uWSGI + NGINX Василию придётся проделывать все те манипуляции, которые мы производили в прошлом модуле. Вы уже должны были убедиться, что процесс настройки взаимодействия uWSGI + NGINX + Flask не из лёгких, пусть его и можно обернуть в некоторую инструкцию.

А что, если, ко всему прочему, на сервере Василия стоит другая операционная система, например Windows, которая не поддерживает работу с uWSGI?

Так много вопросов... И у нас есть на них ответы. Проблема, к которой мы подошли, называется **проблемой воспроизводимости**. О ней, как правило, не задумываются новички, однако она часто проявляется на этапе продакшена. Например, мы хотим перенести проект с локальной машины, где вели разработку, на реальный сервер, но из-за разницы в настройках и конфигурациях этот процесс становится болезненным и требует затрат времени, которого всегда остаётся очень мало на этапе продакшена.

В этом модуле мы разберём, как сделать свой код воспроизводимым и как изолировать свой сервис так, чтобы его можно было запустить на любой машине, а самое главное — как сделать это быстро и просто.

ЦЕЛИ МОДУЛЯ:

- Узнать, что такое воспроизводимость и какими инструментами её можно обеспечить.
- Познакомиться с терминами «виртуализация» и «контейнеризация».
- Научиться создавать виртуальные окружения, изолировать среду разработки, устанавливать и фиксировать зависимости в виртуальных окружениях.
- Познакомиться с инструментом контейнеризации Docker.
- Научиться писать Dockerfile, создавать образы контейнеров, запускать их, а также делиться ими с помощью хостинга docker-образов Docker Hub.
- Создать docker-образ для нашего веб-сервиса и запустить его в контейнере.

С КАКИМ ПРОЕКТОМ БУДЕМ РАБОТАТЬ?

В предыдущем модуле мы написали маленький веб-сервис. В нём функционирует модель машинного обучения, которая выполняет предсказания для данных, поступающих через POST-запросы по эндпоинту `/predict`.

Перед прохождением модуля давайте зафиксируем, как должна выглядеть директория нашего проекта:

```
├─web
│   └─models
│       └─model.pkl
│   └─client.py
│   └─server.py
```

Проект будет располагаться в директории `web`, в которой находятся файлы:

- `server.py` — содержит интерфейс сервера, реализованный на Flask. В интерфейсе предусмотрено два эндпоинта:
 - `'/'` — корневой, по обращению к которому пользователю возвращается тестовое сообщение;
 - `'/predict'` — предназначенный для обработки POST-запросов. Запросы приходят в виде списка из четырёх чисел в формате `json()`. Результатом выполнения запроса является JSON-словарь с ключом `'prediction'` и

значением-предсказанием модели. Вы писали функцию для обработки этого запроса в [прошлом модуле](#).

- `client.py` — скрипт для тестирования POST-запросов на сервер.
- `models` — папка с моделями, в которой находится файл `model.pkl` с моделью.

Файл `server.py`

```
from flask import Flask, request, jsonify
import pickle
import numpy as np

# загружаем модель из файла
with open('models/model.pkl', 'rb') as pkl_file:
    model = pickle.load(pkl_file)

# создаём приложение
app = Flask(__name__)

@app.route('/')
def index():
    msg = "Тестовое сообщение. Сервер запущен!"
    return msg

@app.route('/predict', methods=['POST'])
def predict():
    #ваш код здесь
    pass

if __name__ == '__main__':
    app.run('localhost', 5000)
```

Файл `client.py`

```
import requests

if __name__ == '__main__':
    # выполняем POST-запрос на сервер по эндпоинту add с параметром json
    r = requests.post('http://localhost:5000/predict', json=[1, 0, 1, 24])
    # выводим статус запроса
    print('Status code: {}'.format(r.status_code))
    # реализуем обработку результата
    if r.status_code == 200:
        # если запрос выполнен успешно (код обработки=200),
        # выводим результат на экран
        print('Prediction: {}'.format(r.json()['prediction']))
    else:
        # если запрос завершён с кодом, отличным от 200,
```

```
# выводим содержимое ответа
print(r.text)
```

В этом модуле мы сделаем наш сервис изолированным и поместим его в контейнер.

2. Воспроизводимость

Код, разрабатываемый дата-сайентистом, должен быть воспроизводимым в постоянно меняющихся в условиях:

- меняются данные, поступающие на вход;
- трансформируются пайплайны предобработки данных;
- постоянно изменяются гиперпараметры алгоритмов;
- иногда модифицируются или вовсе удаляются алгоритмы популярных библиотек.

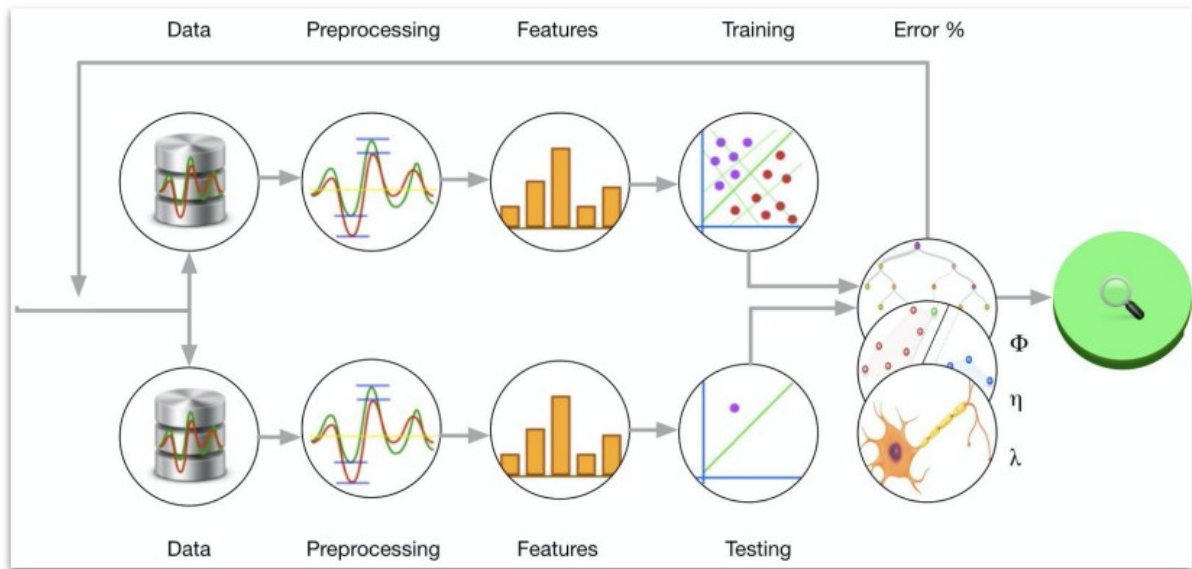
В идеале ожидается, что выполнение кода должно приводить к одинаковым результатам в различных условиях. Именно с этой целью в некоторых заданиях вы встречали требование зафиксировать параметр `random seed` — таким образом, вы уже неоднократно сталкивались с проблемой воспроизводимости на практике.

Воспроизводимость результатов является одним из главных показателей качества модели. Именно поэтому обеспечение воспроизводимости — одна из важнейших проблем в современном ML-инжиниринге.

Мы можем доверять результату, полученному с помощью модели, только когда доподлинно известно, на какой выборке и с каким пайплайном предобработки она была построена. Кроме того, ожидается, что результаты будут консистентными (согласованными с исходными данными), так как необходимо однозначно понимать, какое изменение привело к улучшению или ухудшению результатов.

Наиболее остро вопрос воспроизводимости стоит в случае, когда над одной моделью работает большая DS-команда. Каждый участник команды должен воспроизвести весь пайплайн и получить тот же самый результат. При этом возможные изменения должны доставляться в общий код так же быстро, сохраняя все требования к воспроизводимости.

На схеме приведён классический пайплайн работы над моделью:



ИНСТРУМЕНТЫ ОБЕСПЕЧЕНИЯ ВОСПРОИЗВОДИМОСТИ:

- **Версионирование кода.**

Обычно весь пайплайн представлен в виде частей кода, для работы с которым используется знакомая нам распределённая система управления версиями Git в совокупности с хостингом GitHub.

- **Версионирование артефактов.**

В процессе работы над проектом появляются различные артефакты: датасеты, модели, файлы конфигурации и прочее. Для их версионирования обычно используются такие инструменты, как [DVC](#) и [Sonatype Nexus](#).

- **Виртуализация и контейнеризация.**

Одним из важнейших аспектов воспроизводимости является настройка окружения.

Виртуализация — это технология изоляции, сохранения состояния и воспроизведения окружения. Благодаря ей вы можете воссоздать на другом компьютере точную копию вашего окружения и тем самым обеспечить воспроизводимость.

Однако иногда воссоздания окружения недостаточно и необходимо обеспечить не только воспроизводимость версий библиотек, но и воспроизводимость среды выполнения программы — операционной системы, на которой реализован проект. Для этого прибегают к инструментам **контейнеризации**, таким как Docker.

Все перечисленные выше инструменты мы подробно обсудим далее.

- **Управление экспериментами.**

Для каждого запуска обучения необходимо фиксировать настройки гиперпараметров и сохранять их. В этом помогают системы управления экспериментами, например [MLflow](#).

Над созданием универсального стандарта воспроизводимости работают многие крупные компании. Это означает, что у каждой компании пока что существуют собственные требования к обеспечению воспроизводимости. Когда вы начнёте работать в DS-команде, ваши коллеги будут требовать от вас их соблюдения. Обычно чем крупнее компания, тем более серьёзные требования к воспроизводимости необходимо соблюдать. Поскольку единого стандарта пока не существует, пути и используемые в разных компаниях инструменты могут кардинально отличаться.

Рекомендуем вам всегда помнить о задаче воспроизводимости результатов модели. Это позволит вам постепенно привить себе тот уровень культуры кода, который будет удовлетворять самые требовательные компании.

3. Виртуализация и изолированность.

Virtualenv

При обсуждении воспроизводимости мы упоминали, что одним из важнейших её требований является **настройка окружения**. Разные версии библиотек в разных проектах могут конфликтовать друг с другом, и становится сложно поддерживать работоспособность всех сервисов. Поэтому необходимо обеспечить не только одинаковые операционные системы, но и установку всех нужных библиотек и их версий.

Почти в 100 % случаев при работе над реальным проектом вам потребуется воссоздать точное окружение на сервере, чтобы обеспечить единообразие среды выполнения. Это включает в себя не только стандартные установленные зависимости и интерпретатор Python, но и специфические зависимости, которые могут работать по-разному на разных операционных системах или на разных их версиях.

Несколько лет назад для этих задач использовали программные решения для удалённого управления конфигурациями. Они создавали **большой конфигурационный файл**, в котором описывались все настройки, зависимости и библиотеки, и на его основе настраивались все удалённые машины. Одним из таких инструментов является [Ansible](#). Он позволяет через [SSH-соединение](#) «проталкивать» файлы конфигурации на множество машин и таким образом обеспечивать единообразие.

Другие системы, например [Chef](#), обычно поступают наоборот: узлы «тянут» (pull) конфигурацию с главной машины. Chef используется и сейчас, когда необходимо привести парк машин к одинаковой конфигурации.

ПРОБЛЕМЫ

- Когда в системе много сервисов и при этом каждый сервис запускается на разных машинах, становится тяжело следить за полнотой и правильностью конфигурационных файлов.
- На локальном компьютере разработчика может быть не установлено (или наоборот установлено) ПО, напрямую влияющее на стабильность работы приложения.

Намного проще создать для каждого проекта виртуальную среду, изолированную от основной системы. Тогда каждый из проектов будет иметь свои **независимые настройки**, в том числе разные библиотеки и их версии.

Когда мы говорим про **изолированность**, мы имеем в виду, что приложение будет запускаться в своей отдельной среде и таким образом не зависеть от ОС и настроек системы, на которой мы его разворачиваем

VIRTUALENV



Одним из наиболее популярных инструментов для создания изолированных сред в Python является `virtualenv`. Он обеспечивает работоспособность сервисов вне зависимости от того, какие они имеют зависимости.

Примечание. Сначала мы будем изучать основы работы с виртуальными окружениями на «игрушечных» примерах, а затем рассмотрим, как настроить виртуальное окружение для нашего веб-сервиса.

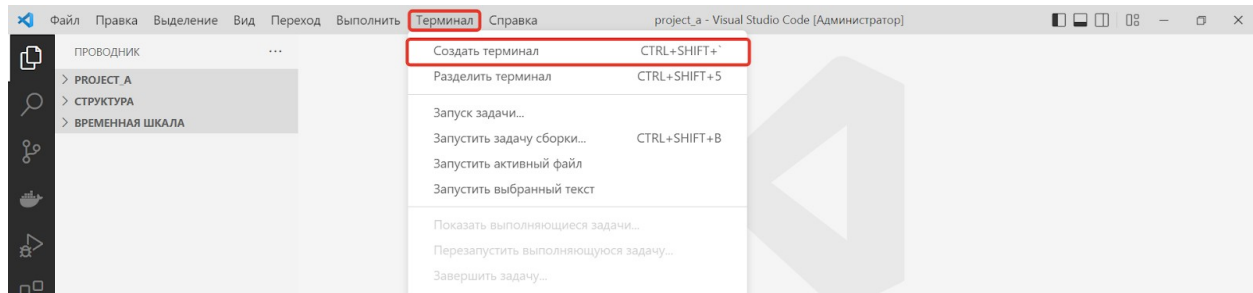
Предварительно создайте в своей операционной системе папку с именем `project_a`. Эта папка будет имитировать папку проекта, и на её основе мы продемонстрируем работу виртуальных окружений. Перейдите в созданную папку в терминале (команда `cd /путь/до/папки`) или откройте её в вашей IDE.

Чтобы установить virtualenv, необходимо воспользоваться стандартной командой:

```
$ pip install virtualenv
```

Примечание. Здесь и далее символ \$ будет означать, что команды выполняются в терминале (для Windows — в командной строке). **Вводить его не нужно** — вводите только текст команды!

Напомним, что в VS Code терминал (в Windows — командную строку) можно открыть с помощью кнопки в верхнем меню IDE:



Так как принципы работы с инструментом незначительно, но отличаются для различных UNIX-систем (Linux и MacOS) и Windows, мы приведём инструкции по работе для каждой из ОС.

Примечание. Чтобы показать, что команды в терминале выполняются в виртуальном окружении, мы будем перед символом \$ писать в скобках имя виртуального окружения, в котором происходит работа. Например:

```
(project_a_venv) $ pip install pandas
```

- **Установка на WINDOWS**

Чтобы создать новую среду, необходимо набрать в терминале команду:

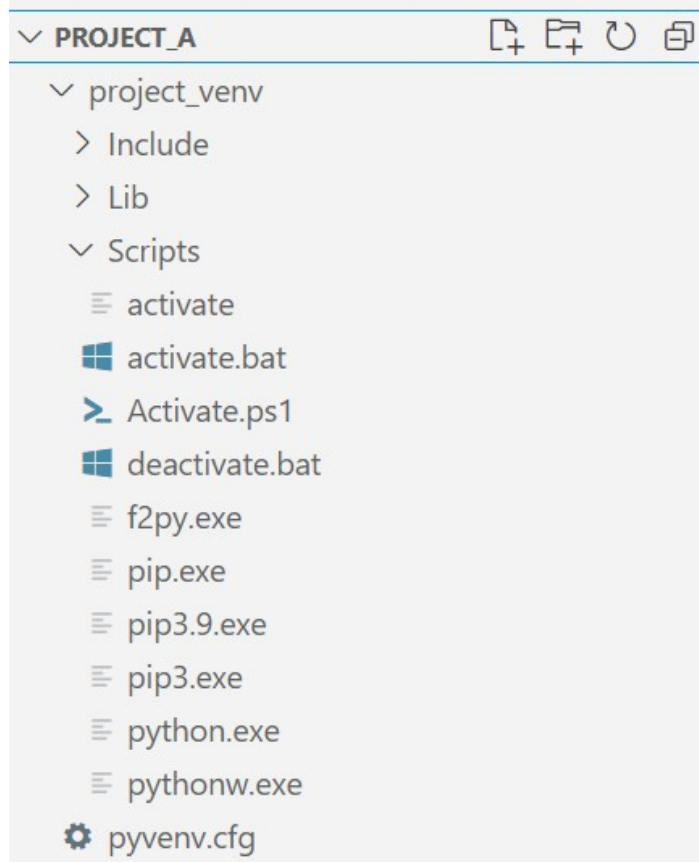
```
$ python -m venv <название сервиса>
```

Например, следующая команда создаёт виртуальное окружение с именем project_a_venv:

```
$ python -m venv project_a_venv
```

После этого в вашей текущей директории появится папка проекта с именем, которое вы указали в команде venv.

Папка с настройками виртуальной среды будет иметь следующий вид (имена файлов могут незначительно отличаться в зависимости от версии Python, которую вы используете):



Директория Scripts — это аналог директории bin в UNIX-системах: в ней лежат файлы, которые взаимодействуют со средой, например файлы для активации и деактивации среды.

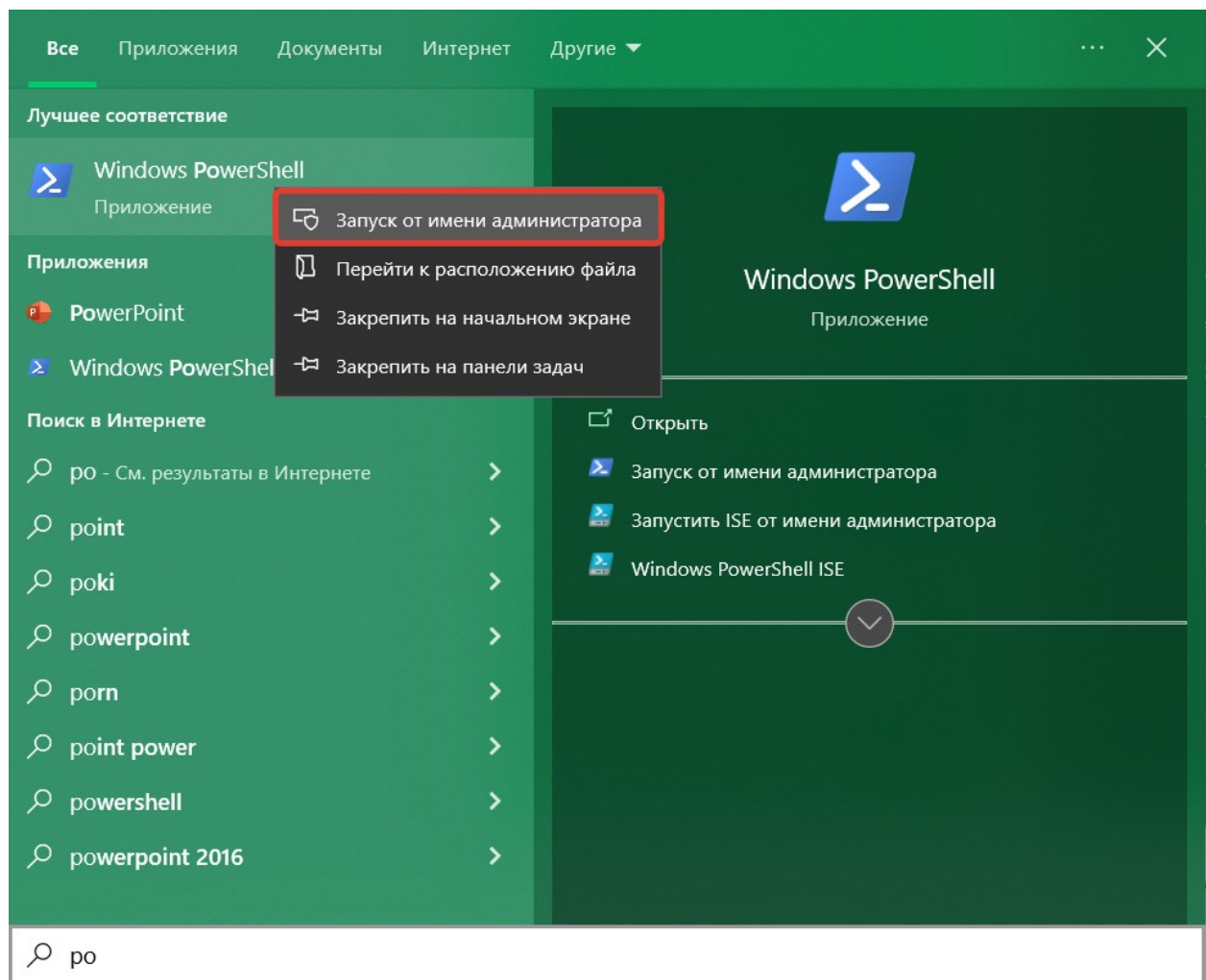
В папке Lib содержится копия python и все зависимости (библиотеки и их версии).

Чтобы активировать виртуальное окружение в Windows, необходимо в командной строке запустить файл Activate.ps1:

```
$ project_a_venv/Scripts/Activate.ps1
```

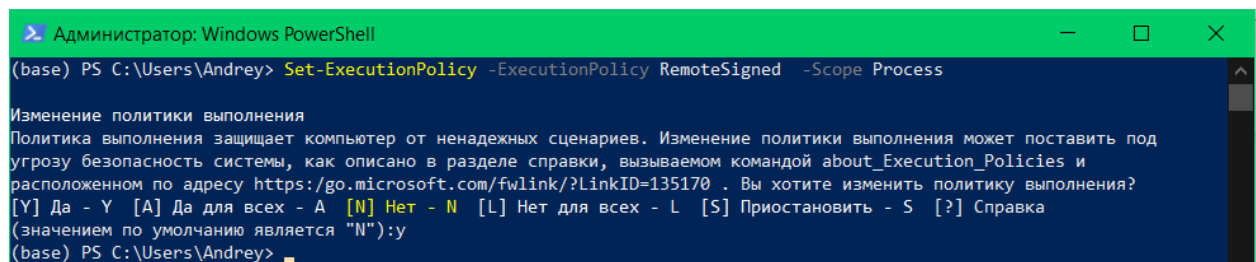
Если не получается активировать среду в Windows

У вас могут возникнуть трудности с активацией среды в VS Code, в частности это может быть связано с тем, что для успешного запуска несистемных скриптов в Windows необходимо активировать политику выполнения для пользователя. Для этого в меню «Пуск» найдите приложение PowerShell и запустите его от имени администратора:



В открывшемся терминале введите следующую команду:

```
$ Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process
```



Если после выполнения описанных выше действий активация среды продолжает вызывать ошибку, вам могут помочь следующие ссылки: [один](#), [два](#).

- Установка на UNIX

Чтобы создать новую среду, необходимо набрать в терминале команду:

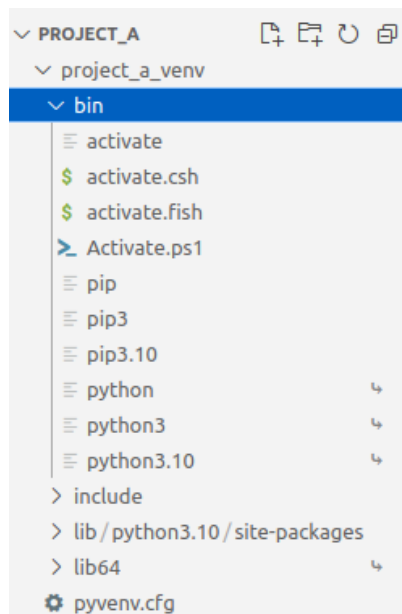
```
$ python3 -m venv <название сервиса>
```

Например, следующая команда создаёт виртуальное окружение с именем `project_a_venv`:

```
$ python3 -m venv project_a_venv
```

После этого в вашей текущей директории появится папка проекта с именем, которое вы указали в команде `venv`.

В UNIX-системах эта директория будет выглядеть примерно следующим образом (имена файлов могут незначительно отличаться в зависимости от версии Python, которую вы используете):



В директории `bin` лежат файлы, которые взаимодействуют с виртуальной средой, а в `lib` и `lib64` содержится копия версии Python и все зависимости (библиотеки и их версии).

Чтобы активировать виртуальную среду и зайти в неё в UNIX-системах, необходимо запустить в терминале команду:

```
$ source project_a_venv/bin/activate
```

Если команда выполнена успешно, вы увидите, что перед приглашением в командной строке появилась дополнительная надпись, совпадающая с именем виртуального окружения. В результате выполнения команды мы увидим следующее:

```
PS A:\Курс DS-3.0\PROD-2\project_a> python -m venv project_a_venv
PS A:\Курс DS-3.0\PROD-2\project_a> project_a_venv/Scripts/Activate.ps1
(project_a_venv) PS A:\Курс DS-3.0\PROD-2\project_a> █
```

Выделенная красным строка говорит нам, что мы находимся в изолированном окружении `project_a_venv`.

Таким образом вы полностью изолируете окружение своего проекта и можете установить все необходимые для работы проекта версии пакетов — эти версии не будут отражены в глобальном окружении Python и будут зафиксированы только в активированном виртуальном окружении.

Например, выполним команду для установки библиотеки `scikit-learn`:

```
(project_a_venv) $ pip install scikit-learn
```

После установки библиотеки вы увидите, что в папке `Lib/site-packages` появится `scikit-learn`, а также зависимости, необходимые для работы этой библиотеки (например, `pumpy`, `scipy` и `joblib`) — они устанавливаются вместе с ней автоматически.

Чтобы выйти из виртуального окружения в область глобального окружения, необходимо ввести в терминале команду:

```
$ deactivate
```

```
• (project_a_venv) andrey@andrey-VirtualBox:~/prod-2/project_a$ deactivate
○ andrey@andrey-VirtualBox:~/prod-2/project_a$ █
```

VIRTUALENV И VS CODE

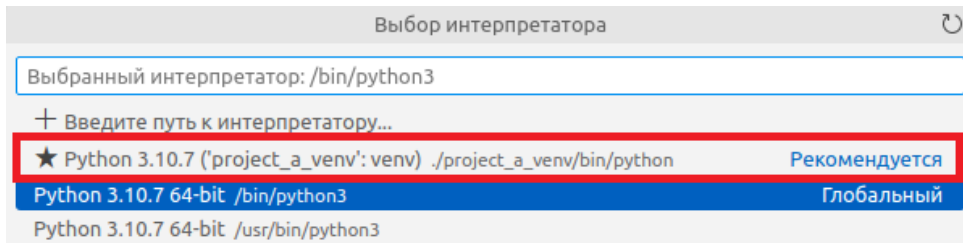
Если вы разрабатываете свои программы в IDE, например в VS Code, то перемещаться между виртуальными окружениями становится совсем просто.

Давайте предварительно создадим в папке нашего проекта пустой `py`-файл, чтобы VS Code понял, что мы работаем с языком Python. Назовём этот файл `app.py`.

Чтобы переключиться между окружениями в VS Code, необходимо перейти в раздел выбора интерпретатора Python (правый нижний угол):



По умолчанию используется глобальное окружение. Нам нужно переключиться на только что созданное виртуальное окружение:



После этого необходимо перезапустить терминал (если он был открыт).

Результат будет тем же, что и после активации виртуального окружения через командную строку:

```
ВЫХОДНЫЕ ДАННЫЕ  КОНСОЛЬ ОТЛАДКИ  ПРОБЛЕМЫ  ТЕРМИНАЛ  JUPYTER
● andrey@andrey-VirtualBox:~/prod-2/project_a$ source /home/andrey/prod-2/project_a/project_a_venv/bin/activate
○ (project_a_venv) andrey@andrey-VirtualBox:~/prod-2/project_a$ █
```

Примечание. Если вы устанавливали среду Anaconda, то вашим окружением по умолчанию будет среда base. В пакет Anaconda входит инструмент conda, который одновременно является и менеджером пакетов (как pip), и менеджером среды (как venv). С его помощью также можно создавать виртуальные окружения и управлять ими. Подробнее о conda и её различиях с pip и venv вы можете узнать [здесь](#).

ИЗОЛЯЦИЯ ЗАВИСИМОСТЕЙ

Теперь давайте на примере рассмотрим, как работать с виртуальными окружениями.

Рядом с папкой project_a создайте ещё одну папку проекта и назовите её project_b. В этой папке также создайте пустой файл app.py.

Откройте два терминала: в первом перейдите в папку project_a, а во втором — в project_b. Также можно открыть эти папки в двух окнах VS Code.

В папке с проектом A создадим виртуальное окружение с именем project_a_venv, активируем его и установим scikit-learn (если вы не делали этого ранее):

```
$ python -m venv project_a_venv
$ project_a_venv/Scripts/Activate.ps1
(project_a_venv)$ pip install -q scikit-learn
```

Примечание. Ключ -q предназначен для установки без вывода справочной информации — «тихая» установка (от англ. quiet — тихий).

Затем создадим виртуальное окружение в папке project_b с именем project_b_venv, активируем его и установим пакет pandas.

```
$ python3 -m venv project_b_venv
$ source project_b_venv/bin/activate
```

```
(project_b_venv)$ pip install -q pandas
```

Давайте посмотрим, какие библиотеки доступны внутри каждого из окружений. Для этого воспользуемся командой `pip freeze`, которая выводит список установленных пакетов с указанием номера их версии. Выполните в каждом из окружений команду:

```
(project_{}_venv)$ pip freeze
```

Для проекта А мы увидим примерно следующую картину:

```
• (project_a_venv) andrey@andrey-VirtualBox:~/prod-2/project_a$ pip freeze
joblib==1.2.0
numpy==1.23.4
scikit-learn==1.1.3
scipy==1.9.3
threadpoolctl==3.1.0
```

Для проекта В список будет выглядеть так:

```
• (project_b_venv) andrey@andrey-VirtualBox:~/prod-2/project_b$ pip freeze
numpy==1.23.4
pandas==1.5.1
python-dateutil==2.8.2
pytz==2022.6
six==1.16.0
```

Примечание. На скриншотах приведены результаты работы команд только в UNIX-системах, так как они совпадают с результатами в Windows.

Что мы видим? Списки установленных пакетов отличаются: например, библиотеки `scikit-learn` нет в виртуальном окружении проекта В, а библиотеки `pandas` нет в виртуальном окружении проекта А. Списки пакетов для проектов А и В пересекаются только в одном — библиотеке `numpy`. Так происходит потому, что и `scikit-learn`, и `pandas` требуют для своей стандартной работы пакет `numpy`.

Теперь давайте взглянём на список глобально установленных пакетов. Для этого откроем ещё один терминал и, не активируя никаких окружений, напомним в нём команду для вывода списка установленных пакетов:

```
$ pip freeze
```

```
• andrey@andrey-VirtualBox:~/prod-2$ pip freeze
bcrypt==3.2.0
blinker==1.4
Brlapi==0.8.4
certifi==2022.9.24
chardet==4.0.0
charset-normalizer==2.1.1
click==8.0.3
colorama==0.4.5
command-not-found==0.3
cryptography==3.4.8
cupshelpers==1.0
dbus-python==1.2.18
defer==1.0.6
distlib==0.3.6
```

В глобальном окружении находится совершенно другой список зависимостей.

Примечание. Пакеты в глобальном окружении на вашем компьютере могут отличаться от приведённых на скриншоте.

Только что мы посмотрели на пример **изоляции**: мы создали отдельные виртуальные окружения для каждого из проектов, зависимости которых изолированы друг от друга. Из приведённого примера становится интуитивно понятно, что виртуальные окружения существуют независимо от глобального и установка пакета в одно из них не означает, что пакет будет установлен в какое-то другое окружение, и наоборот.

Например, если мы установим в глобальное окружение библиотеку numpy версии 1.19.2,

```
$ pip install numpy==1.19.2
```

то для виртуальных окружений проектов А и В версия numpy не изменится. Это утверждение справедливо и в обратную сторону. Такой механизм позволяет нам работать с проектами А и В независимо друг от друга и даже независимо от глобального окружения, тем самым гибко управляя проектами.

Примечание. Очевидно, что попытка запустить в окружении код, использующий библиотеку, которая в нём не установлена, приведёт к ошибке. Например, если попробовать импортировать библиотеку pandas в файле app.py проекта А,

Файл project_a/app.py

```
import pandas as pd

df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
```

а после запустить этот файл из-под соответствующего виртуального окружения, мы получим ошибку импорта:

```
```py
(project_a_venv) $ python3 app.py

Traceback (most recent call last):
 File "/home/andrey/prod-2/project_a/app.py", line 1, in <module>
 import pandas as pd
ModuleNotFoundError: No module named 'pandas'
```

Поэтому стоит иметь в виду, что для каждого создаваемого виртуального окружения необходимо отдельно установить все зависимости, которые не входят в [стандартную библиотеку](#) Python.

Однако это ещё не всё. Когда разработка проекта завершена и мы готовы загрузить его на свой GitHub и поделиться им с коллегами, мы можем сохранить все те версии библиотек, которые использовали при разработке, в файл. Для этого применяется всё та же команда `pip freeze`, только с указанием имени файла, в который необходимо произвести запись. Традиционно такой файл называют `requirements.txt` и располагают в корневой директории проекта. Для указания файла используется ключ `-r` или оператор `>`:

```
(project_a_venv) $ pip freeze -r requirements.txt
```

или

```
(project_a_venv) $ pip freeze > requirements.txt
```

В результате создаётся текстовый файл requirements.txt, который вы можете поместить в свой репозиторий на GitHub.

Когда коллеги будут клонировать ваш проект, им не нужно будет разбираться, какие библиотеки вы использовали в проекте, а тем более — каких версий. Они смогут создать на своём компьютере собственное виртуальное окружение и установить в него все необходимые зависимости, используя лишь одну команду:

```
(имя_виртуального_окружения) $ pip install requirements.txt
```

Удобно, не правда ли?

Теперь давайте заглянем «под капот» и поймём, как именно работает изоляция.

## КАК РАБОТАЕТ ИЗОЛЯЦИЯ

Чтобы понять, как работает изоляция, давайте проверим, где располагаются исполняемые файлы Python в случае глобального и виртуального окружений.

Для начала посмотрим на глобальное окружение. С деактивированной средой запускаем команду:

```
UNIX: $ which python3
```

```
WINDOWS: $ Get-Command python | Format-Table Source
```

Данная команда выводит расположение скрипта, который выполняется при вызове команды python.

Теперь попробуйте запустить ту же самую команду, но уже находясь в одном из виртуальных окружений, например в виртуальном окружении проекта A:

```
UNIX: (project_a_venv) $ which python3
```

```
WINDOWS: (project_a_venv) $ Get-Command python | Format-Table Source
```

```
py #...\project_a\project_a_venv\Scripts\python.exe
```

Видно, что, активировав среду, мы получаем другой путь к интерпретатору Python. Но как подключение библиотек зависит от расположения интерпретатора? Напрямую.

Мы не будем подробно останавливаться на том, как происходит внутреннее управление путями в операционных системах, так как это довольно обширная тема и её изложение будет уникально для каждого типа операционных систем. Если говорить вкратце, при активации виртуального окружения вы меняете специальные переменные среды ОС, в результате чего при запуске команды вызова интерпретатора используете не глобальную версию интерпретатора Python, который имеет доступ ко всем установленным пакетам, а его копию, которая лежит в папке с виртуальным окружением. Этот интерпретатор не имеет доступа к папке, где хранятся библиотеки, установленные в глобальное или любое



другое виртуальное окружение — у него она своя. Это же правило работает и в обратную сторону.

Примечание. Подробнее о том, как меняются переменные среды, можно узнать в [этой статье](#).

## ПРИЧИНЫ ИСПОЛЬЗОВАНИЯ ВИРТУАЛЬНЫХ ОКРУЖЕНИЙ

У вас мог возникнуть вопрос: зачем это нужно? Зачем так сложно? Раньше мы спокойно работали в глобальном окружении и даже не знали, что оно глобальное, а сейчас для каждого проекта придётся создавать отдельное виртуальное окружение?

Если вы хотите стать профессиональными разработчиками, работать в команде и обеспечивать воспроизводимость библиотек с минимальной затратой времени, то да — создавать виртуальные окружения вам придётся чуть ли не в каждом проекте.

Представим следующую ситуацию: у вас есть два проекта — проект А и проект В. Они оба имеют зависимость от одной и той же библиотеки. Проблема становится явной, когда мы начинаем запрашивать разные версии этой библиотеки. Например, может случиться так, что проект А запрашивает версию 1.0.0, а проект В — версию 2.0.0, причём версия 2.0.0 настолько сильно отличается от 1.0.0, что для адаптации проекта А под новую версию придётся его полностью переписывать. Это большая проблема для Python, ведь работая только в глобальном окружении, мы не можем использовать обе версии библиотеки. Так или иначе, возникнет конфликт, который могут решить виртуальные окружения.

Вторая причина использования виртуальных окружений — удобная коммуникация внутри команды. Разрабатывая проект в виртуальном окружении, мы можем сохранить **только те зависимости и их версии**, которые использовали в проекте, например в файл `requirements.txt`.

Затем мы можем передать разработку своим коллегами (например, через GitHub), и они смогут установить только те зависимости, которые необходимы для работы нашего кода.

**Примечание.** Важно отметить, что папку самого виртуального окружения в GitHub помещать не нужно. Всегда добавляйте эту папку в файл `.gitignore`.

Например, мы можем прописать в файле `.gitignore` строку `*venv/`, которая будет означать, что всё содержимое папок, названия которых оканчиваются на `venv`, будет игнорироваться при коммитах:



Подробнее о том, как сочетать использование виртуальных окружений с .gitignore, вы можете почитать [здесь](#).

## ВИРТУАЛЬНОЕ ОКРУЖЕНИЕ ДЛЯ FLASK-ПРИЛОЖЕНИЯ

Пришло время создать виртуальное окружения для нашего веб-сервиса.

Предварительно перейдите в каталог, где расположен код вашего проекта из предыдущего модуля.

В корневой директории проекта (у нас она называется web) создадим виртуальное окружение с именем project\_venv.

### UNIX, WINDOWS:

```
$ python3 -m venv project_venv
```

Локальные копии Python и pip будут установлены в каталог project\_venv в каталоге вашего проекта.

Активируем виртуальное окружение:

### UNIX:

```
$ source project_venv/bin/activate
```

### WINDOWS:

```
$ project_venv/Scripts/Activate.ps1
```

Теперь мы можем переходить к установке пакетов в наше окружение. Сначала установим wheel с локальным экземпляром pip, чтобы убедиться, что наши пакеты будут устанавливаться даже при отсутствии архивов wheel:

```
(project_venv) $ pip install wheel
```

Затем установим Flask, requests и scikit-learn. Чтобы установить несколько пакетов сразу, можно просто перечислить их через пробел после команды install.

```
(project_venv) $ pip install flask requests scikit-learn
```

Затем запустим наш сервер:

### UNIX, WINDOWS:

```
(project_venv) $ python ./server.py
```

Давайте проверим, что мы установили все необходимые для работы веб-сервиса и его тестирования зависимости. Через браузерную строку зайдите по адресу <http://localhost:5000> или <http://127.0.0.1:5000>. Там должно быть выведено сообщение, что ваш сервер запущен.

Также попробуйте отправить POST-запрос на ваш сервер, выполнив клиентский скрипт в соседнем терминале.

## UNIX, WINDOWS:

```
(project_venv) $ python ./client.py
```

В результате работы скрипта должно быть выведено сообщение о статусе обработки запроса (он должен быть равен 200) и предсказание модели для отправленных данных.

Если всё работает корректно, GET- и POST-запросы отработали без ошибок, то мы можем зафиксировать версии наших зависимостей и поместить их в файл `requirements.txt` в корневой директории проекта:

```
(project_venv) $ pip freeze > requirements.txt
```

В нашей директории появится файл `requirements.txt` — он ещё пригодится в следующих юнитах.

Теперь, если мы загрузим наш код на GitHub (предварительно добавив папку `project_venv` в файл `.gitignore`), нашему коллеге Василию необходимо будет клонировать себе репозиторий, а после этого создать виртуальное окружение и активировать его. Чтобы в точности воссоздать все версии зависимостей, которые мы использовали, Василию будет достаточно набрать в терминале следующую команду:

```
$ pip install requirements.txt
```

После этого зависимости внутри виртуального окружения, созданного Василием, будут совпадать с нашими.

Virtualenv — полезный инструмент. Однако не всё так гладко, ведь он работает только с Python и не обеспечивает полную изоляцию. Также он не позволяет ограничивать ресурсы для каждого сервиса: например, иногда бывает необходимо разрешить одному сервису использование всех ядер процессора и ограничить — другому. Если мы также хотим автоматически балансировать нагрузку между сервисами, то и тут `virtualenv` нам не помощник.

На эту тему есть [прекрасная статья](#) на Хабре.

Ещё один недостаток технологии виртуальных окружений состоит в том, что они не помогут, если разработка проекта ведётся на одной операционной системе, а эксплуатация — на другой. В частности, вы могли заметить, что при создании виртуального окружения для нашего веб-сервиса мы ничего не сказали о связке Flask, uWSGI и NGINX — как мы помним по прошлому модулю, эта связка поддерживается только для UNIX-систем. Поэтому нам необходимо создать не просто виртуальное окружение, где мы будем хранить необходимые для работы приложения библиотеки, а что-то вроде карманной операционной системы, внутри которой уже настроено взаимодействие всех необходимых инструментов для работы сервиса, включая установленные внутри версии библиотек. Причём хотелось бы, чтобы эту карманную ОС можно было запускать на любом устройстве, то есть чтобы наш сервис можно было переносить между платформами. Тут к нам на помощь приходят **системы контейнеризации**, в частности **Docker**.

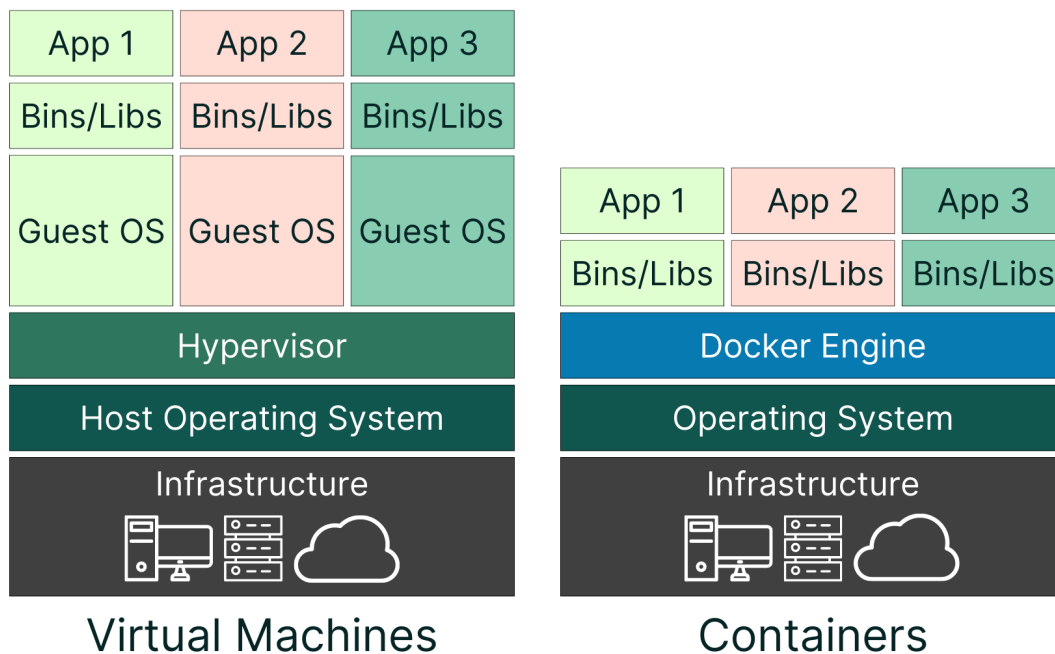
## 4. Контейнеризация. Docker и Docker Hub

В 2012–2013 гг. появилось несколько систем контейнеризации, которые позволяли иметь дополнительную операционную систему в изолированном от основной ОС виде и не использовали много ресурсов.

**Контейнеризация** — это метод виртуализации, при котором ядро операционной системы поддерживает несколько изолированных экземпляров приложений.

Наиболее популярной системой контейнеризации оказался Docker. Сегодня, если кто-то говорит о контейнерах, скорее всего, имеется в виду именно Docker.

По сути, Docker — это программное обеспечение, которое позволяет в автоматическом режиме создавать **виртуальные машины** и управлять ими. При этом Docker делает это более элегантно по сравнению с обычными средствами виртуализации, например VirtualBox.



Что изображено на схеме выше?

Данные схемы демонстрируют отличия с точки зрения архитектуры информационной системы при использовании виртуальных машин и контейнеров. Схемы стоит читать снизу вверх.

**В случае использования виртуальных машин (Virtual Machines) мы имеем:**

- Инфраструктуру системы (Infrastructure) — рабочие компьютеры пользователей, серверы, облачные технологии и т. д.
- Операционную систему устройства (Host Operating System) — операционные системы (Windows/Unix/Mac), на которых работают компьютеры;

- Гипервизор (Hypervisor) — процесс, который отделяет операционную систему компьютера от физического оборудования. Проще говоря, это специальное приложение, которое позволяет создавать и управлять виртуальными машинами и запускать на одном компьютере множество различных виртуальных операционных систем. Например, для Windows таким приложением является Hyper-V. Подробнее о гипервизорах и их устройствах вы можете прочитать [здесь](#).
- Гостевые/виртуальные операционные системы (GuestOS) — операционные системы, которые работают внутри виртуальных машин. Их может быть сколько угодно (зависит от возможностей железа, установленного в компьютере, на котором запускаются виртуальные машины).
- Бинарные файлы и библиотеки (Bins/Libs) — пакеты, которые необходимы для работы приложений, запущенных на каждой из гостевых ОС.
- Приложения (App) — приложения, которые мы запускаем внутри гостевых ОС. Это может быть несколько серверных приложений, которые ожидают поступающих интернет-запросов, при этом каждое из них запущено под определённой ОС.

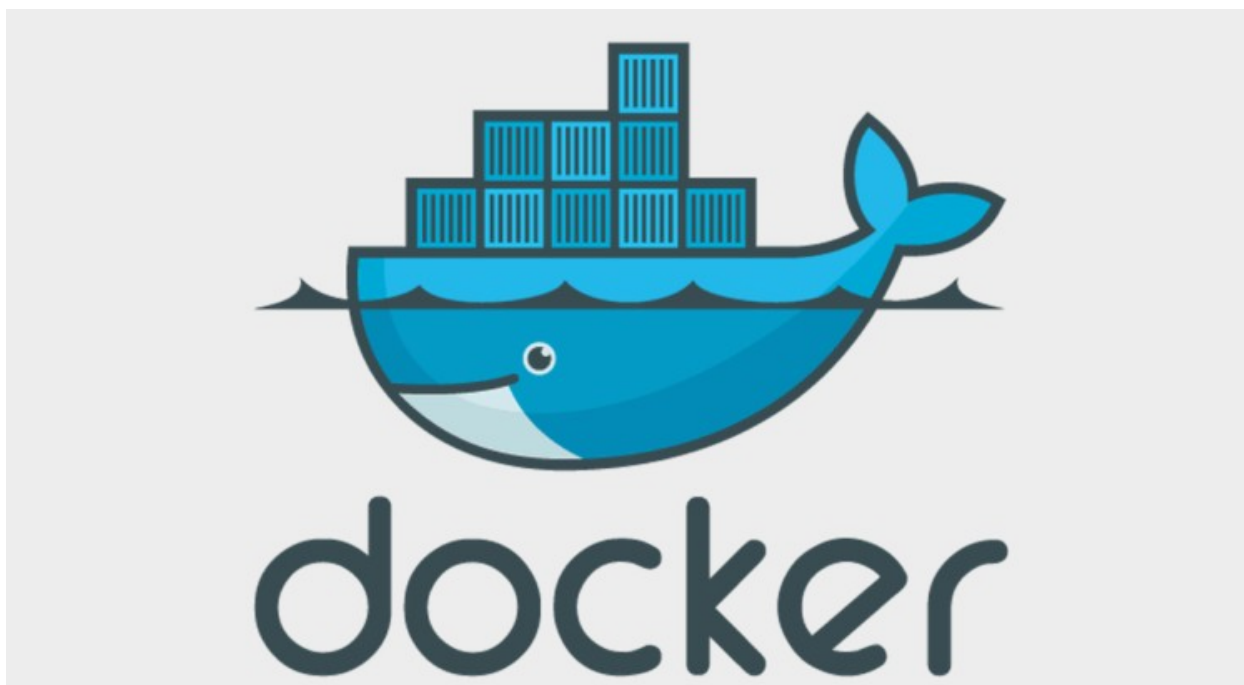
#### **В случае использования контейнеров (Containers) мы имеем:**

- Инфраструктуру системы (Infrastructure).
- Операционную систему устройства (Operating System).
- Движок Докер (Docker Engine) — специальное программное обеспечение для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации. Проще говоря, это контейнеризатор приложений. Он позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут на любой операционной системе. В каждом контейнере запущены свои приложения (как правило, один контейнер — одно приложение) со своими зависимостями.
- Бинарные файлы и библиотеки (Bins/Libs).
- Приложения (App).

Таким образом, из схемы видно, что, в отличие от виртуальных машин, контейнерам не нужна установка отдельных гостевых операционных систем с ненужными функциями, такими как графический интерфейс, встроенные в ОС приложения и т. д. В каждом контейнере содержится своя микро-ОС, в которой можно изолированно запускать отдельные приложения.

Докер позволяет собрать приложение со всем его окружением и зависимостями в **контейнер**. В нашем случае приложением может быть модель ML, предсказывающая стоимость авто, его зависимости — библиотеки `sklearn`, `numpy` и `pandas`, а окружением — ОС, на которой работает приложение.

За счёт того что Docker потребляет не очень много ресурсов машины, на которой находится, можно запускать сразу несколько контейнеров даже на среднестатистическом компьютере. Из-за этого стало принято использовать **небольшие контейнеры** для каждого конкретного сервиса: например, если у нас есть Django-приложение с базой данных, то сам сервер будет находиться в одном контейнере, а база — в другом. Изоляция часто позволяет добиться улучшения производительности и упрощения миграции сервисов.



У Docker неслучайно такая эмблема и название!

Проще всего это представить, воспользовавшись метафорой кораблей и контейнеров. Если необходимо перевезти несколько типов грузов (продукты, тяжёлые машины и химикаты), то сделать это на одном корабле можно, только используя контейнеры и таким образом изолируя грузы друг от друга. Когда грузы находятся в контейнерах, уже не очень важно, что внутри — оно может быть погружено на корабль.

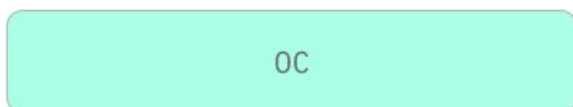
Таким образом, контейнеризация — это создание некоторого «ящика» для вашего приложения, в который будут сложены ядро, ОС, библиотеки, ПО и само приложение.



Приложение А    Приложение В    Приложение С



Ядро



ОС

Docker работает с функциями системы Linux и её ядрами, поэтому при работе на других ОС он использует небольшую хитрость: каждый раз в систему устанавливается виртуальная машина с Linux, и Docker работает уже в ней, но мы этого даже не замечаем.

## ПОДВЕДЁМ ПРОМЕЖУТОЧНЫЙ ИТОГ

Контейнер Docker, по сути, представляет собой «виртуальную» файловую систему, в которую вы устанавливаете всё необходимое для запуска вашего приложения. Это «всё необходимое» включает в себя даже ядро системы Linux. При запуске такого Docker ваша базовая система поднимает контейнер с этой файловой системой, и получается легковесная виртуальная машина, что-то вроде карманной операционной системы.

## ОСНОВНЫЕ КОМПОНЕНТЫ DOCKER

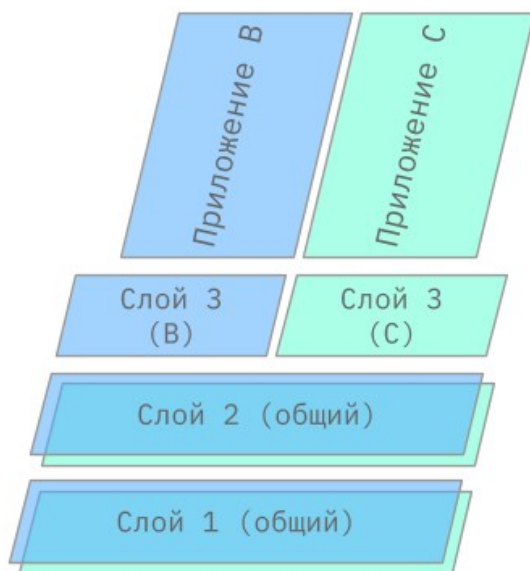
Поговорим об основных компонентах Docker.

**Образы (images)** — это основные строительные блоки, на основании которых создаются контейнеры (а в них впоследствии упаковываются приложения).

Образ, по своей сути — это шаблон, своеобразный чертёж или рецепт, в котором содержится образ базовой операционной системы, код приложения и библиотеки.

Запуская на его основе контейнер, мы создаём исполняемый экземпляр, который **инкапсулирует** требуемое программное обеспечение.

Ключевую роль в устройстве образа играет идея о слоях.



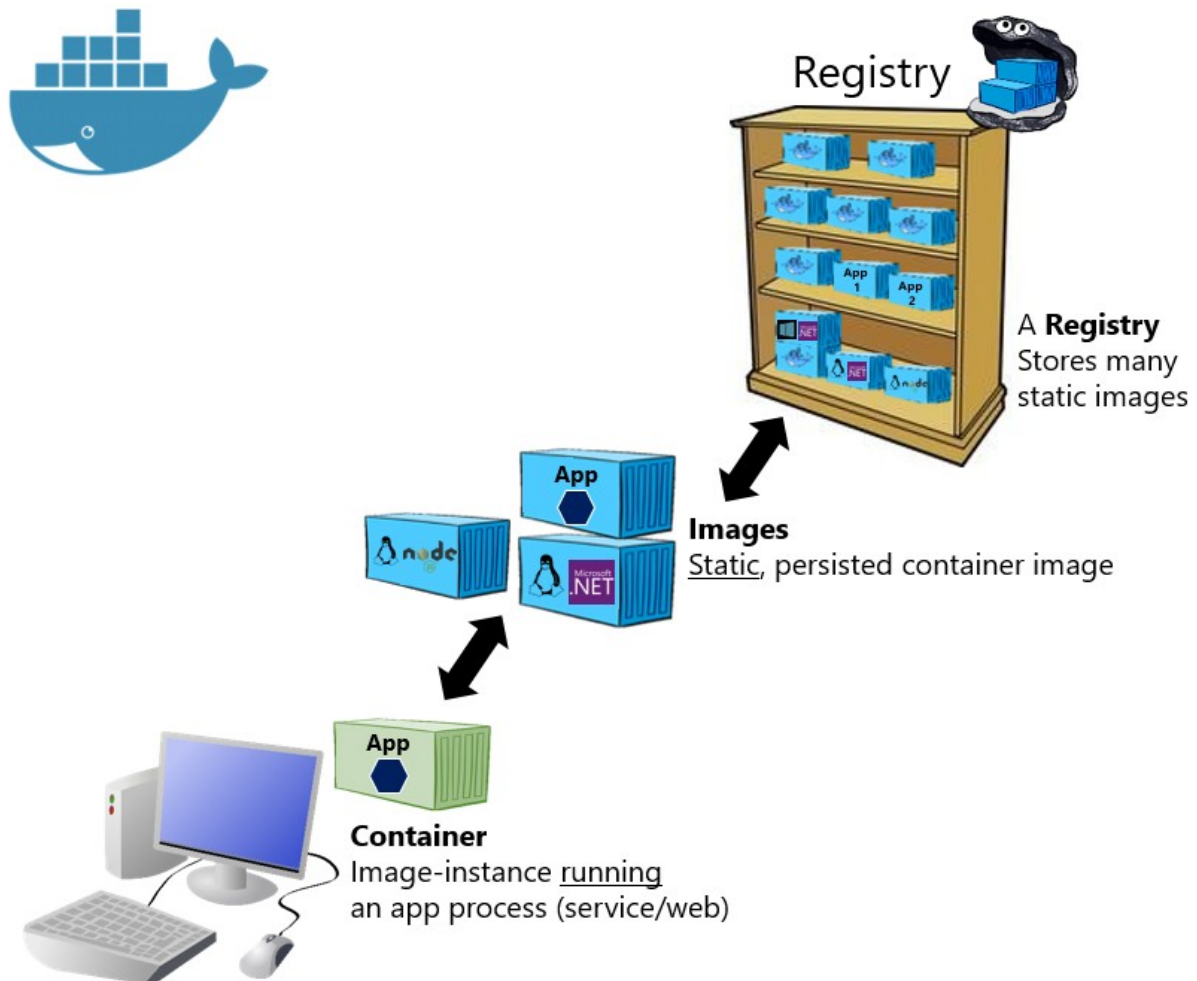
В основе каждого docker лежит базовый образ с операционной системой. С каждым новым слоем в ОС добавляются другие компоненты. Каждый слой представляет из себя подобие diff файловой системы. Например, вы взяли образ системы Ubuntu и поставили туда Python. В таком случае ваш образ будет состоять из двух слоёв — самой ОС и файлов Python поверх неё.

В качестве базового образа для docker можно использовать не только ОС, но и готовые образы с нужными вам компонентами. Кто угодно может добавлять свои образы в **общий регистр**, поэтому в нём очень много готовых образов, доступных для расширения. Таким образом, можно взять готовый образ из публичного репозитория в качестве базового и

добавлять в него дополнительные слои. Это делается в соответствии с инструкциями из Dockerfile, которые мы подробно рассмотрим дальше.

Готовые образы хранятся в **Docker Registry**. Они могут быть публичными или приватными: например, официальное публичное хранилище — это [Docker Hub](#). Это аналог GitHub, но если последний используется для хранения кода и работы с Git, то Docker Hub используется для хранения docker-образов и работы с Docker.

В качестве базового можно использовать абсолютно любой образ. Однако стоит помнить о безопасности и удостовериться, что образ не навредит.



Основное преимущество Docker заключается в том, что с помощью Docker Registry мы можем делиться образами, а значит, легко переносить созданные нами приложения.

Для работы с Docker Registry достаточно знать две команды (они очень похожи на команды Git) — подробнее о них мы поговорим далее:

1. `docker push` — отправить собранный образ в Docker Registry.
2. `docker pull` — скачать готовый образ из Docker Registry.



Существует множество [официальных образов](#). Например, можно найти [образ, где в качестве ОС используется Ubuntu](#), или, например, [Linux с уже установленным Python](#). Есть даже [Docker в Docker](#)! Очень удобно, не правда ли?

Docker-образ управляется Daemon, который отвечает за все действия, связанные с контейнерами, и, конечно, самим клиентом для взаимодействия с ним.

**Важно.** Зарегистрируйтесь на [Docker Hub](#). Для этого вам понадобится только e-mail. Ваш аккаунт пригодится вам далее при выполнении заданий, а также при работе с самим Docker.

Прежде чем перейти к созданию собственного Docker-контейнера, давайте установим сам Docker.

## УСТАНОВКА DOCKER

Существует две версии Docker: [Docker Community Edition \(CE\)](#) и [Docker Business](#). Версия Community содержит бесплатный набор продуктов Docker. Корпоративная (Business) версия является сертифицированной и представляет собой контейнерную платформу, предоставляющую своим пользователям дополнительные платные функции, например управление образами, безопасность образов, оркестрирование и управление средой выполнения контейнеров.

Для наших базовых целей будет достаточно и Community-версии, но имейте в виду, что большие компании, для которых важны безопасность и сертификация, а также гибкость работы с внутренней частью контейнеров, как правило, работают в Business-версии.

Исконно Docker предназначался только для операционных систем Linux. Контейнеры Docker, созданные в конкретной операционной системе, используют ядро ОС. Иначе говоря, это означает, что мы не можем использовать ядро Windows для запуска контейнеров Linux или наоборот.

Однако для систем Windows и MacOS есть обходной путь — приложение **Docker Desktop**. Поэтому, прежде чем начинать работу с Docker, пользователям этих ОС необходимо установить это приложение.

Хитрость Docker Desktop заключается в том, что для своих контейнеров он запускает Docker на виртуальной машине под операционной системой Linux. То есть, по сути, при сборке контейнеров вы будете использовать ядро ОС Linux, но даже не заметите этого. Благодаря этой особенности ваши контейнеры можно будет запускать на других компьютерах с ОС Linux или ОС, где установлен Docker Desktop.

Мы будем учиться работать с контейнерами, используя инструменты командной строки (терминала) и не прибегая к графическому интерфейсу Docker Desktop. Однако если вы изучите основные концепции работы с консольным Docker, для вас не составит труда разобраться и в графическом интерфейсе этого приложения.

Итак, давайте перейдём к установке. Все её шаги уже описаны в инструкциях с официального сайта — мы лишь приведём их здесь и обратим ваше внимание на некоторые нюансы.

**Примечание.** Если вы являетесь пользователем Windows, но работаете с дистрибутивами Linux через [WSL](#), то для вас актуальна установка Docker Desktop для Windows с последующей интеграцией Docker в дистрибутив Linux через WSL. Более подробную информацию о работе с Docker через Windows WSL вы можете найти [здесь](#).

## Для пользователей Linux

Для дистрибутивов Linux нет необходимости устанавливать Docker Desktop, так как Docker является «родным» для ОС данного типа и все контейнеры запускаются без использования дополнительных виртуальных машин, которые нужны для Windows и MacOS. Кроме того, чаще всего мы работаем с Docker, используя команды в терминале и не прибегая к графическому интерфейсу. В рамках данного курса мы также будем управлять контейнерами только с помощью интерфейса командной строки.

Однако если вы вдруг захотите познакомиться с интерфейсом Docker Desktop и управлять контейнерами с помощью графического интерфейса, то вам повезло: в 2022 году Docker Desktop стал доступен и для Linux (считается, что для Linux ОС этот инструмент избыточен). Ознакомиться с инструкциями по установке и работе с Docker Desktop для Linux вы можете [здесь](#).

Мы же рассмотрим установку «чистого» Docker и его движка для Linux. Сделаем это на примере дистрибутива Ubuntu, следуя этой инструкции. Инструкции по установке на другие дистрибутивы Linux будут отличаться незначительно, вы можете ознакомиться с ними [здесь](#).

Установка будет состоять из ряда этапов (все команды выполняются в терминале):

1. Сначала необходимо выполнить подготовительные шаги.
  - Обновите индекс пакета apt и установите пакеты, чтобы разрешить apt использовать репозиторий через HTTPS:

```
$ sudo apt-get update
$ sudo apt-get install \
 ca-certificates \
 curl \
 gnupg \
 lsb-release
```

- Добавьте официальный GPG-ключ Docker:

```
$ sudo mkdir -p /etc/apt/keyrings
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
gpg --dearmor -o /etc/apt/keyrings/docker.gpg
```

- Используйте следующую команду для настройки репозитория:

```
$ echo \
"deb [arch=$(dpkg --print-architecture)
signed-by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu \
```

```
$(lsb_release -cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null
```

1. Теперь, когда все подготовительные шаги выполнены, мы можем установить движок Docker — Docker Engine. Для этого установите Docker Community Edition (Docker-CE) и ещё несколько компонентов:

```
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
docker-compose-plugin
```

2. Убедимся, что установка Docker прошла успешно, запустив образ hello-world:

```
$ sudo docker run hello-world
```

Данная команда запускает образ контейнера с именем hello-world. По завершении работы контейнера вы должны увидеть в терминале примерно следующий текст:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 (amd64)
 3. The Docker daemon created a new container from that image which runs the
 executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
 to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

1. Изначально после установки Docker на Linux ОС команда docker может быть запущена только от имени пользователя root или от имени пользователя, указанного в автоматически создаваемой при установке группе пользователей — docker. Это означает, что попытка запустить команду docker без префикса sudo или от имени пользователя, который не находится в группе docker, приведёт к ошибке:

#### Output

```
docker: Cannot connect to the Docker daemon. Is the docker daemon running on this host?.
See 'docker run --help'.
```

Давайте сделаем так, чтобы не нужно было вводить `sudo` каждый раз при запуске Docker. Для этого выполним следующие шаги:

- a. Добавьте пользователя, под которым мы сейчас работаем, в список пользователей docker с помощью следующей команды:

```
$ sudo usermod -aG docker ${USER}
```

- b. Далее необходимо выйти из системы и войти заново, чтобы изменения вступили в силу. Или вы можете запустить следующую команду, чтобы активировать изменения в группах:

```
$ newgrp docker
```

- c. Проверьте, что вы имеете права доступа к docker без указания `sudo`. Для этого запустите образ hello-world:

```
$ docker run hello-world
```

- d. Вы должны получить тот же результат, что и выше.

## Для пользователей Windows 10/11

Ещё несколько лет назад мы бы сказали, что работать с Docker на ОС Windows невозможно либо возможно, но очень неудобно — гораздо проще это делать на Linux. Почему?

- Во-первых, большинство Docker-контейнеров работают на системе Linux.
- Во-вторых, Windows с трудом совместима с удалёнными серверами (вы помните, что почти все они работают на UNIX-системах) и некоторыми библиотеками для их настройки (например, uWSGI и NGINX).

Сейчас Docker неплохо встраивается в экосистему Windows, и Microsoft активно пытаются заставить контейнеры работать на ядре своей операционной системы.

Интересный факт для сравнения: в 2018 году на Docker Hub можно было найти лишь 13 официальных лицензированных образов, работающих на ядре Windows. Сейчас их более 140.

Microsoft уже выпустили множество программ с графическим интерфейсом для работы с Docker (например, Docker Desktop) и инструкций к ним. Инструкции на русском языке вы можете найти [здесь](#). Рекомендуем вам с ними ознакомиться, если вы собираетесь использовать Docker из-под Windows. А мы пока перейдём к установке Docker Desktop.

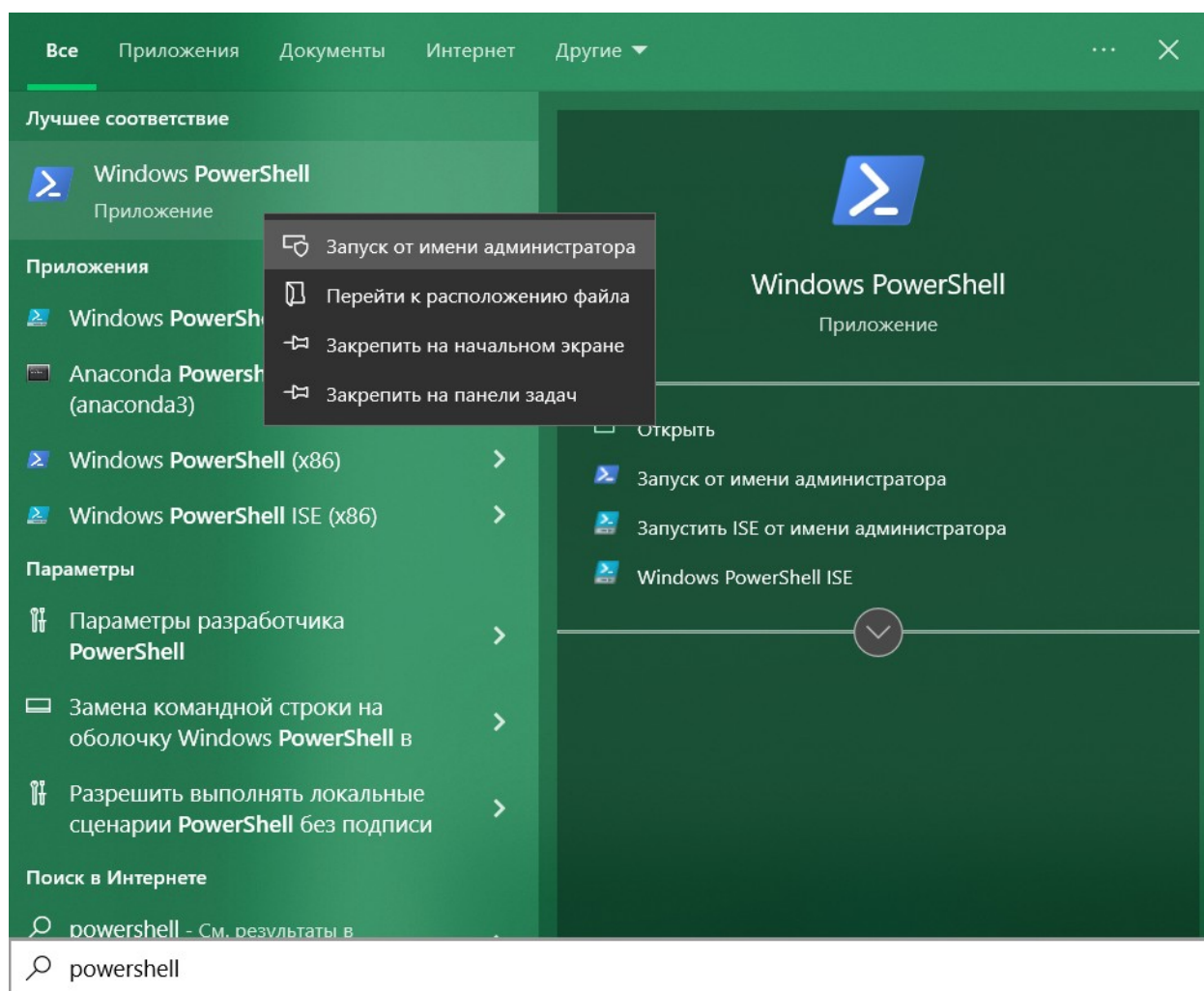
Будем следовать руководству с официального сайта Microsoft.

1. Первым делом для систем Windows необходимо включить технологию поддержки виртуализации Hyper-V.

Hyper-V позволяет запускать виртуализированные компьютерные системы поверх физического узла. Эти виртуализированные системы можно использовать и контролировать как физические компьютерные системы, но они находятся в виртуализированной и изолированной среде.

Как вы понимаете из описания, Hyper-V просто необходим для функционирования Docker.

Первый способ включить Hyper-V в Windows — через командную строку. Наберите в поиске powershell и запустите командную строку от имени администратора.



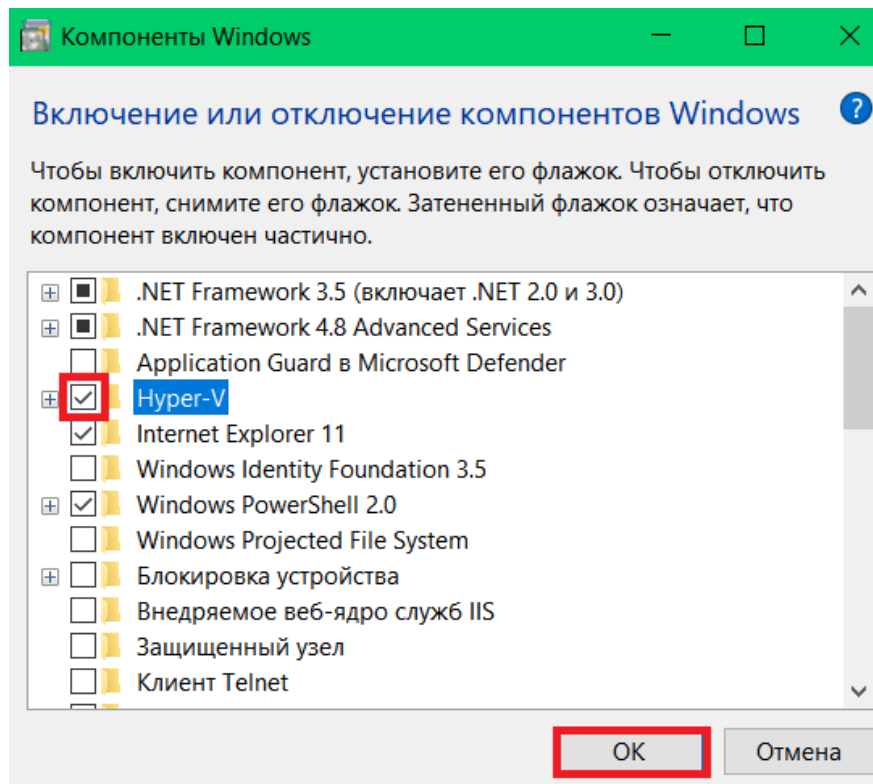
Далее в открывшейся командной строке запустите следующую команду:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -All
```

Если не удаётся найти команду, убедитесь, что вы используете PowerShell от имени администратора.

После завершения установки выполните перезагрузку.

Другой вариант запустить Hyper-V — через графический интерфейс. Для этого наберите в поиске «Включение или отключение компонентов Windows». В открывшемся окне поставьте галочку напротив Hyper-V и нажмите ОК:



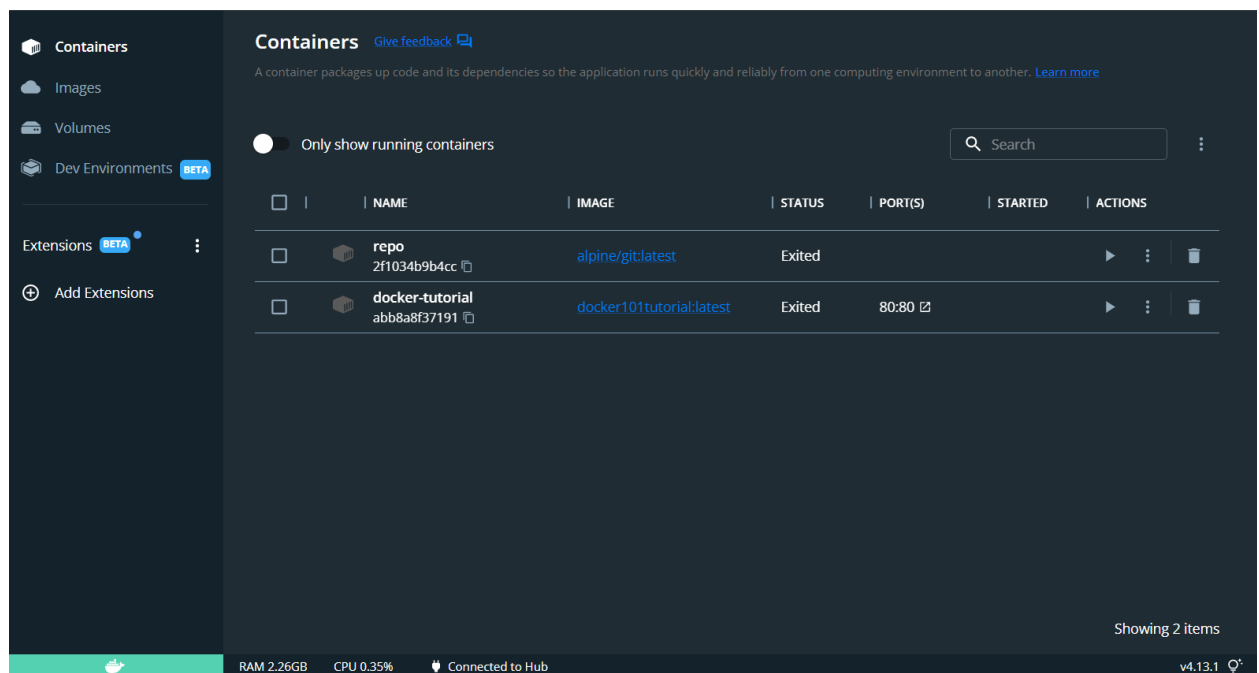
После этого система предложит перезапустить компьютер.

1. Теперь всё готово для установки самого Docker. Загрузите и установите Docker Desktop с [официального сайта](#). В установке нет ничего специфического — просто нажмите везде ОК.

После установки запустите Docker Desktop, найдя его в поиске. Ознакомьтесь с лицензионным соглашением и примите его.

2. После принятия лицензионного соглашения запустится сам Docker Desktop. Приложение предложит пройти ознакомительный туториал — мы рекомендуем выполнить все шаги.

3. Когда установка будет завершена, Docker Desktop предложит вам создать учётную запись Docker. Если у вас уже есть учётная запись на Docker Hub, просто введите её. Если вы ещё не обзавелись таковой, зарегистрируйтесь — учётная запись понадобится вам для использования всех возможностей Docker, в частности для отправки образов и их скачивания.
4. После регистрации вам будет доступно использование графического интерфейса Docker Desktop. С его помощью вы сможете управлять вашими контейнерами — запускать и останавливать их, а также создавать на их основе образы.



Так как в этом модуле мы будем работать с Docker через командную строку, то обзор интерфейса мы опустим — вы можете ознакомиться с ним самостоятельно.

5. Теперь давайте проверим, что Docker установлен и готов к работе — для этого запустим образ hello-world:

```
$ docker run hello-world
```

Данная команда запускает образ контейнера с именем hello-world. По завершении работы контейнера вы должны увидеть в терминале примерно следующий текст:



```
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

```
To generate this message, Docker took the following steps:
```

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.  
(amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

```
To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash
```

```
Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/
```

```
For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

## Для пользователей MacOS

**Системные требования.** Для успешной установки Docker Desktop ваш компьютер Mac должен соответствовать [определённым требованиям](#).

Для установки Docker Desktop на MacOS предварительно скачайте установочный файл Docker.dmg с [официального сайта](#).

Доступно две версии (для Apple- и Intel-чипсетов процессора) — выберите файл в соответствии со своим чипсетом.

О том, как узнать информацию о своём процессоре, в том числе о чипсете, можно прочитать [здесь](#).

После скачивания установочного файла вы можете воспользоваться любым из двух путей установки Docker Desktop: интерактивный (через графический интерфейс) или через командную строку.

### ИНТЕРАКТИВНАЯ УСТАНОВКА

1. Дважды кликните мышью на Docker.dmg, чтобы открыть программу установки. Затем перетащите значок Docker в папку Applications.
2. Дважды кликните мышью на Docker.app в папке «Приложения» (папка для запуска Docker).
3. В меню настройки (img) отобразится окно соглашения об обслуживании подписки Docker. Ознакомьтесь с лицензионным соглашением и примите его.
4. После принятия всех условий запустится Docker Desktop.



## УСТАНОВКА ИЗ КОМАНДНОЙ СТРОКИ

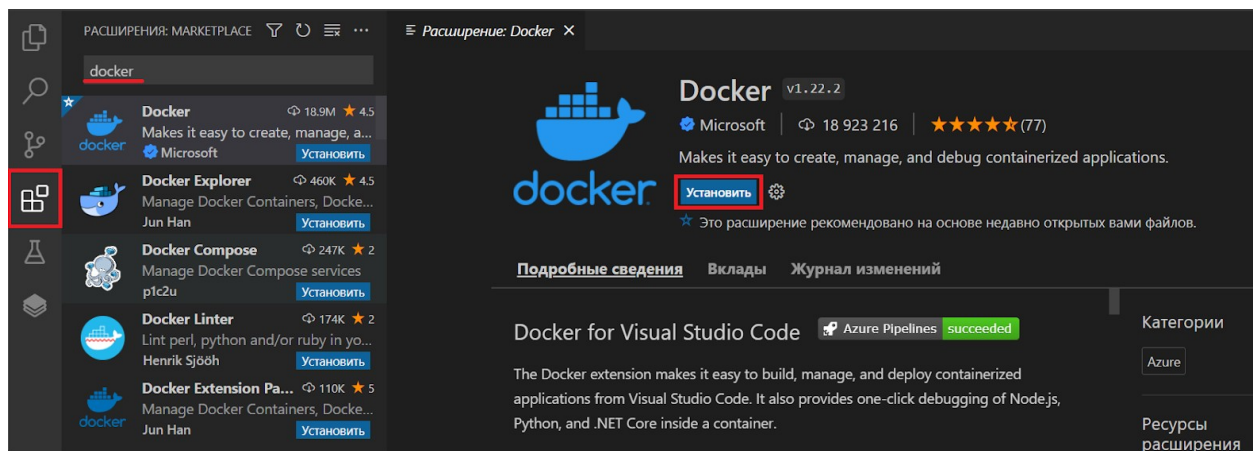
После загрузки Docker.dmg выполните следующие команды в терминале, чтобы установить Docker Desktop в папку Applications:

```
$ sudo hdiutil attach Docker.dmg
$ sudo /Volumes/Docker/Docker.app/Contents/MacOS/install
$ sudo hdiutil detach /Volumes/Docker
```

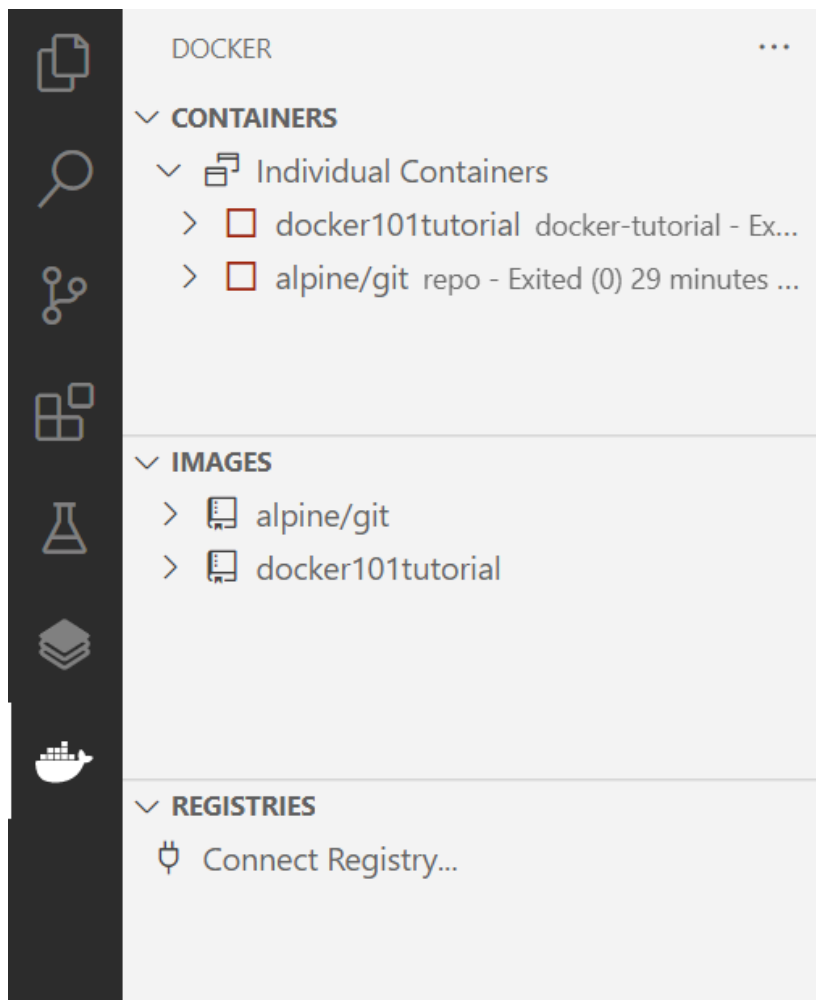
Поскольку MacOS обычно выполняет проверку безопасности при первом использовании приложения, выполнение команды install может занять несколько минут.

## DOCKER В VS CODE

В IDE VS Code есть специальные инструменты для удобной работы с контейнерами. Для их подключения к IDE необходимо установить расширение Docker. Однако установка самого Docker также требуется:

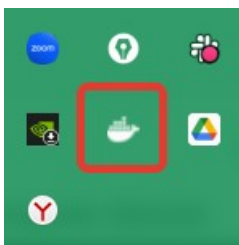


Если процесс Docker запущен, VS Code автоматически подключится к нему и вы сможете управлять вашими уже собранными контейнерами прямо из IDE.



**Примечание.** Если во вкладке Docker в VS Code возникают ошибки, это означает, что не запущен фоновый процесс Docker.

Чтобы удостовериться, что он работает, откройте скрытые значки на панели инструментов Windows — там должен быть логотип Docker.



При работе с Docker в Windows и MacOS приложение Docker Desktop должно быть всегда запущено.

В этом юните мы рассмотрели основные компоненты Docker и разобрали, как создаётся образ контейнера. Также мы установили Docker на свой компьютер и даже запустили свой первый контейнер. В следующем юните мы поговорим о том, как создавать собственные образы, познакомимся с синтаксисом Dockerfile и основными командами Docker.

## 5. Создание docker-образов. Dockerfile

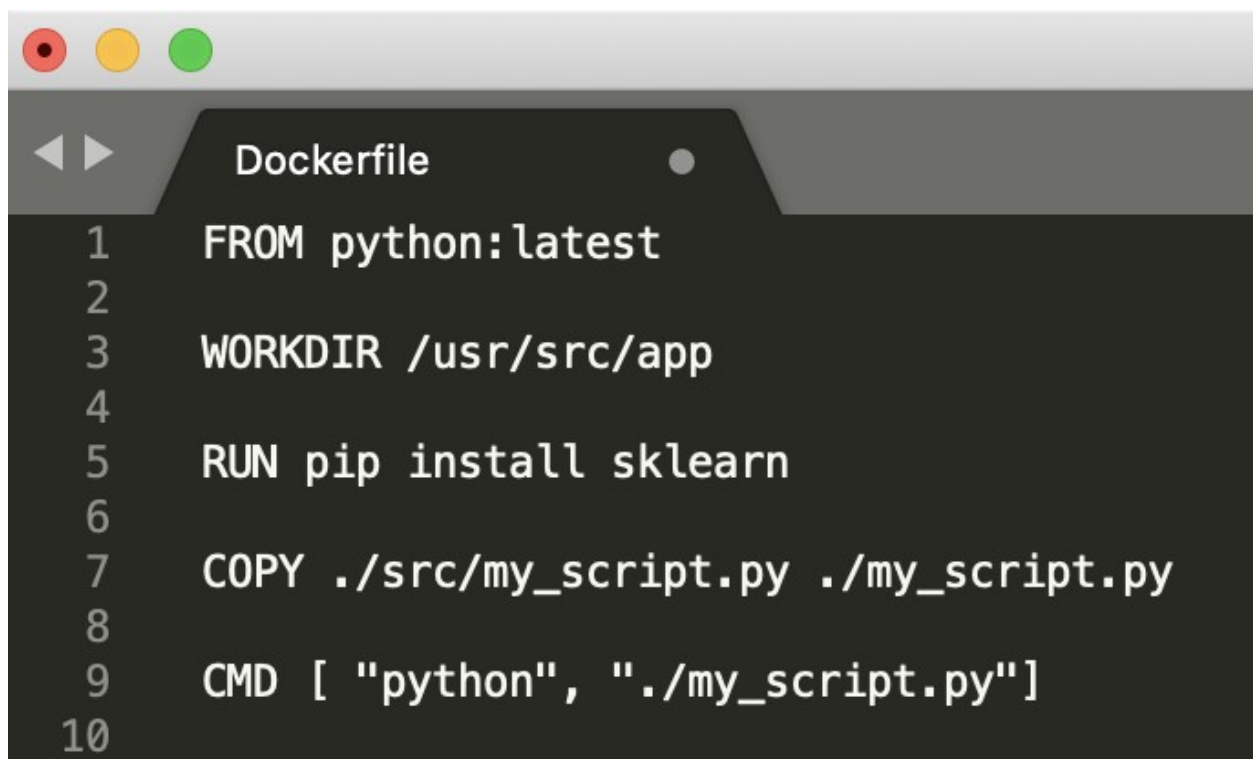
Нам уже пора учиться создавать собственные образы, запускать контейнеры и управлять ими. В этом юните мы рассмотрим основной синтаксис работы с Docker на примере приложения для визуализации данных, а уже в следующем юните применим технологию контейнеров для нашего веб-сервиса и заставим его работать на любой машине.

**Dockerfile** — это специальный файл, в котором содержатся инструкции по сборке контейнера (а точнее, его слоёв): какой тип контейнера и операционную систему использовать, какие дополнительные пакеты установить, какие команды запустить.

Так, например, если нам нужен образ для решения задач машинного обучения, то в Dockerfile мы пропишем инструкцию включить в образ библиотеки `sklearn` или `tensorflow`.

Каждая такая команда приводит к созданию отдельного слоя: diff слоя при добавлении файлов будет состоять из файлов, при выполнении команды — из разницы в файловой системе до выполнения и после.

Типичный Dockerfile выглядит примерно так:

A screenshot of a code editor window titled "Dockerfile". The editor has a dark background and shows a series of Docker build instructions. On the left side of the code, line numbers 1 through 10 are visible. The instructions are: 1. FROM python:latest, 2. (blank line), 3. WORKDIR /usr/src/app, 4. (blank line), 5. RUN pip install sklearn, 6. (blank line), 7. COPY ./src/my\_script.py ./my\_script.py, 8. (blank line), 9. CMD [ "python", "./my\_script.py"], 10. (blank line).

```
1 FROM python:latest
2
3 WORKDIR /usr/src/app
4
5 RUN pip install sklearn
6
7 COPY ./src/my_script.py ./my_script.py
8
9 CMD ["python", "./my_script.py"]
10
```

Давайте на практике разберёмся, что всё это значит, и соберём свой образ контейнера.

# ПИШЕМ DOCKERFILE ДЛЯ ПРИЛОЖЕНИЯ ВИЗУАЛИЗАЦИИ ДАННЫХ

Мы хотим написать сервис, который создаёт две случайные подвыборки данных из нормальных распределений: первая выборка — с параметрами  $\mu=0$  и  $\sigma=0$  (параметры стандартного нормального распределения), вторая — с параметрами, которые ввёл пользователь.

Для каждой выборки должны строиться и сохраняться в файл `plot.png` графики плотности распределений. Все файлы с графиками будем помещать в папку `output`.

Мы хотим запускать это приложение в контейнере и обеспечивать его работу на любом компьютере.

После окончания работы наш проект будет содержать:

- само приложение `plot.py`;
- папку `output`, в которой будут храниться результаты работы;
- приложения, то есть графики плотности распределений в формате PNG;
- `Dockerfile` — описание контейнера, чтобы обеспечить работу приложения на любом компьютере;
- файл `requirements.txt` — зависимости приложения (библиотеки, которые мы используем).

**Примечание.** Для простоты изложения в данном проекте не используется виртуальное окружение, но мы уверены, что вы самостоятельно можете его создать и установить в него все необходимые зависимости (`numpy`, `matplotlib`, `seaborn`).

## ШАГ 1. ПИШЕМ ПРИЛОЖЕНИЕ

Начнём с написания самого приложения. Пусть сначала исходный код в файле `plot.py` и папка `output` будут находиться в корневой директории `my_first_container`:

```
my_first_container
├── output
└── plot.py
```

### Файл `plot.py`

```
import sys
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

Создаём бесконечный цикл
while True:
 # Блок обработки исключений на случай неверного ввода
 try:
```

```

Считываем среднее с клавиатуры
mean = input('Введите среднее для нормального распределения:')
Пытаемся преобразовать введённую строку в число float
mean = float(mean)
Если код преобразования во float отработал, выходим из цикла
break
Если возникло исключение, выводим предупреждение об ошибке
except:
 print('Это не число!')
Создаём бесконечный цикл
while True:
 # Блок обработки исключений на случай неверного ввода
 try:
 # Считываем стандартное отклонение с клавиатуры
 deviation = input('Введите стандартное отклонение для
нормального распределения:')
 # Пытаемся преобразовать введённую строку в число float
 deviation = float(deviation)
 # Если код преобразования во float отработал, выходим из цикла
 break
 # Если возникло исключение, выводим предупреждение об ошибке
 except:
 print('Это не число!')
Генерируем стандартное нормальное распределение
distribution_n1 = np.random.normal(0,1,1000)
Генерируем нормальное распределение с параметрами, введёнными
пользователем
Дополнительно умножаем результат на 2
distribution_n2 = np.random.normal(mean,deviation,1000)*2

sns_plot = sns.histplot(distribution_n1, kde=True, color="orange")
sns_plot = sns.histplot(distribution_n2, kde=True, color="skyblue")
plt.savefig('/usr/src/app/output/plot.png')

print('Файл успешно сохранен')

```

Чтобы проверить работу приложения, запустите в терминале (находясь в корневой директории my\_first\_container) команду:

#### UNIX:

```
$ python3 ./plot.py
```

#### WINDOWS:

```
$ python ./plot.py
```

Укажите параметры распределения (помните о том, что стандартное отклонение всегда > 0). В результате выполнения кода в папке src/output/ должен появиться файл plot.png.

## ШАГ 2. СОЗДАЁМ DOCKER-ОБРАЗ

Наше мини-приложение работает. Теперь мы можем завернуть его в контейнер, поэтому следующим шагом будет создание образа контейнера. Для этого создайте (если вы этого ещё не сделали) файл Dockerfile (без расширения) в корне папки, в которой лежит ваше приложение.

Традиционно код самого приложения помещают в папку src (от англ. source — источник), а Dockerfile и файл с зависимостями requirements.txt (о последнем мы поговорим далее) располагают рядом с этой папкой в корневой директории проекта. Тогда директория нашего проекта будет выглядеть следующим образом:

```
my_first_container
├── src
│ ├── output
│ └── plot.py
└── Dockerfile
```

Откройте Dockerfile в любом текстовом редакторе или в привычной IDE.

Переходим к описанию нашего контейнера. Первое, что нужно обязательно указать в нашем новом Dockerfile, — какой образ мы берём за основу. Базовый образ будет задавать файловую систему контейнера и многие другие его составные конфигурации.

Давайте загрузим в качестве основы [образ Linux с уже установленным Python](#). Для указания базового образа, на основе которого будет собираться контейнер, используется ключевое слово FROM. Итак, вот первая строка нашего Dockerfile:

```
FROM python:3.9
```

**Примечание.** Здесь :3.9 в названии базового образа указывает на его версию. В данном случае мы используем версию 3.9 — с ней в нашем коде не возникнет предупреждений. Но вы можете использовать другие версии: например, python:latest означает использование последней доступной версии.

**Примечание.** В [описании образа](#) можно ознакомиться с информацией о том, какие бывают сборки и версии. Например, существуют обычная сборка python: и сборка python:-alpine, созданная на базе Alpine Linux. Последняя весит намного меньше, чем большинство базовых образов дистрибутива (~ 5 МБ). Кроме того, есть сборка python:-slim, которая содержит только минимальные пакеты, необходимые для запуска Python, и не включает в себя стандартные пакеты.

Теперь укажем путь к рабочей папке нашего приложения внутри docker-контейнера (вместо /usr/src/app вы можете прописать любой путь, по которому хотите поместить файлы внутри контейнера). Для этого используется ключевое слово WORKDIR.

```
WORKDIR /usr/src/app
```

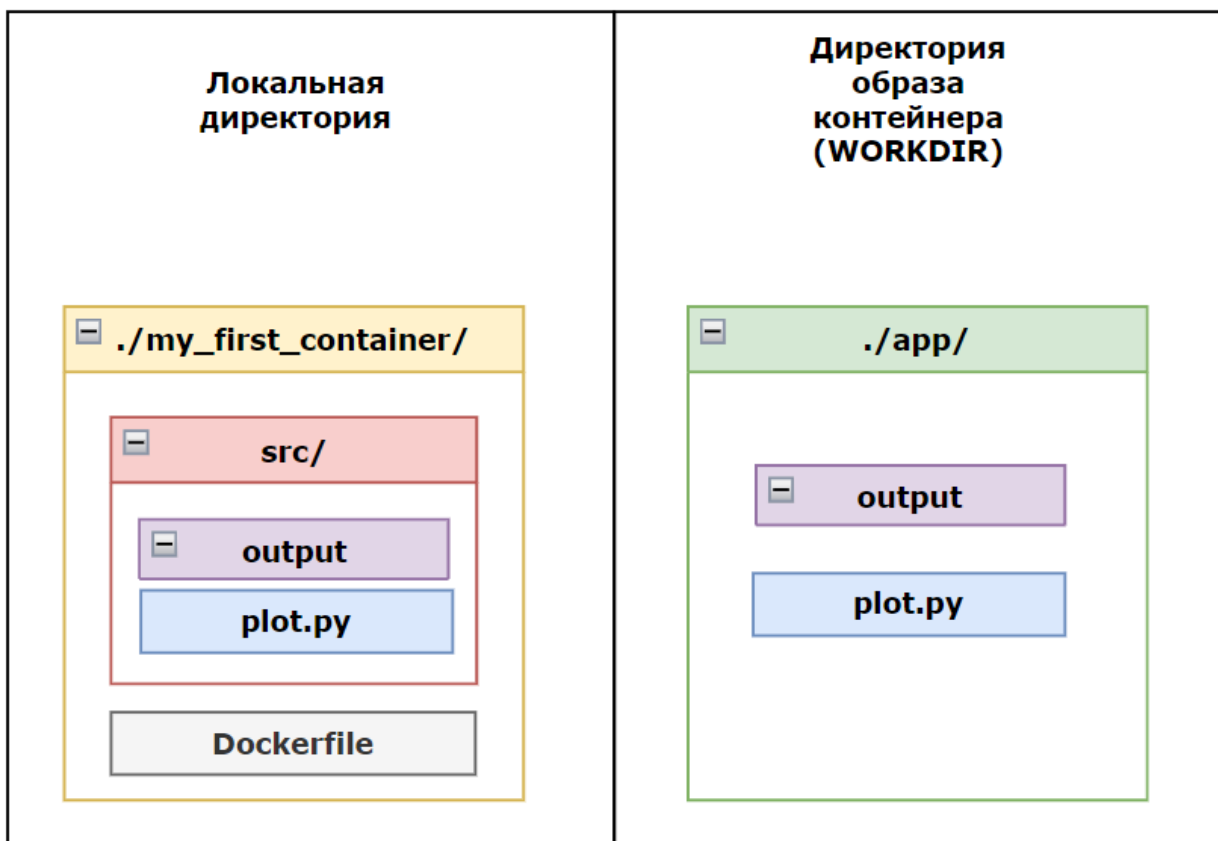
**Примечание.** Важно понимать, что файловая система контейнера существует отдельно от вашей. Так как образ python, который мы используем в качестве базового, собран на основе Linux, то и файловая система контейнера будет, как в ОС Linux. Ключевое слово

WORKDIR говорит контейнеру, какой каталог будет использоваться при его запуске. В данном случае мы говорим, что та директория, с которой по умолчанию будет запускаться контейнер, — это `/usr/src/app`.

Далее скопируем в рабочую директорию файлы из корня нашего приложения. Для этого используется ключевое слово `COPY`. В качестве первого аргумента этой директивы указываются папки или файлы, которые мы хотим скопировать с локальной машины, в качестве второго аргумента — куда мы хотим поместить эти копии в контейнере.

```
COPY ./src/ ./
```

Что здесь происходит? Мы указываем, чтобы при создании образа содержимое папки `./src/`, в которой находится исходный код нашего приложения, было перемещено в рабочую папку (корневой каталог `./`). В результате при сборке образа контейнера в рабочей директории образа появятся файл `plot.py` и папка `output`.



Наконец, напомним команду для запуска скрипта, который будет выполняться вместе с запуском контейнера. Для этого используется директива `CMD` (от англ. *command* — команда):

```
CMD ["python", "./plot.py"]
```

Таким образом, промежуточный `Dockerfile` будет выглядеть так:

**Файл `Dockerfile`**

```
FROM python:3.9
WORKDIR /usr/src/app
COPY ./src/ .
CMD ["python", "./plot.py"]
```

Теперь откроем терминал, перейдём в папку с нашим приложением и запустим команду для сборки контейнера (docker build):

```
$ docker build -t my_first_image .
```

Разберём команду на составляющие:

- build сообщает docker, что мы хотим создать образ;
- ключ -t указывает на название образа;
- . в конце означает, что Dockerfile нужно искать именно в корне. Так как мы запускаем команду из директории my\_first\_container/, а в ней находится Dockerfile, то Docker автоматически найдёт его. Если ваш Dockerfile находится в директории, отличной от той, в которой вы запускаете команду docker build, то вместо "." вам необходимо будет указать путь до него.

Запускаем команду и видим, что образ успешно создан. В справочной информации отражены все наши сборки. Если на каком-то этапе произойдёт ошибка при сборке, docker уведомит вас об этом в терминале.

```
Sending build context to Docker daemon 35.33kB
Step 1/4 : FROM python:3.9
3.9: Pulling from library/python
17c9e6141fdb: Already exists
de4a4c6caea8: Already exists
4edced8587e6: Already exists
a7969cfff46: Already exists
74fbfde6af91: Already exists
16fe51aed899: Already exists
b8127bbe87b: Pull complete
f3b0eb11a359: Pull complete
07d95c49563f: Pull complete
Digest: sha256:475fe86ebf1da48ea27009a8f7d7e96231af4142de918a68010d48d0abb9c9c5
Status: Downloaded newer image for python:3.9
--> ab0d2f900193
Step 2/4 : WORKDIR /usr/src/app
--> Running in 305e680d41fd
Removing intermediate container 305e680d41fd
--> 2b1424d835d2
Step 3/4 : COPY ./src/ .
--> 641c9f1705c8
Step 4/4 : CMD ["python", "./plot.py"]
--> Running in 4e78476651e0
Removing intermediate container 4e78476651e0
--> a49cc0cbfd1a
Successfully built a49cc0cbfd1a
Successfully tagged my_first_image:latest
```

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

Чтобы убедиться, что образ собран, а также посмотреть список доступных образов, можно использовать команду:

```
$ docker images
```



Если вы ранее не работали с Docker, скорее всего, сейчас вы получите следующие образы: `my_first_image`, `python` (он использовался в качестве базового для `my_first_image`) и `hello-world` (мы запускали его на этапе установки Docker).

Команда `docker images` выводит список собранных образов в виде таблицы со следующими столбцами: `REPOSITORY` (имя образа), `TAG` (тег образа, в котором обычно указывается его версия), `IMAGE ID` (идентификатор образа, по которому его можно однозначно найти, — у вас он может отличаться), `CREATED` (как давно образ был собран), `SIZE` (размер образа).

После выполнения команды `docker images` мы увидим на экране примерно следующую таблицу:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<code>my_first_image</code>	<code>latest</code>	<code>6187f2d2e322</code>	<code>22 minutes ago</code>	<code>932MB</code>
<code>python</code>	<code>latest</code>	<code>00cd1fb8bdcc</code>	<code>2 weeks ago</code>	<code>932MB</code>
<code>hello-world</code>	<code>latest</code>	<code>feb5d9fea6a5</code>	<code>13 months ago</code>	<code>13.3kB</code>

Из неё видно, что образу `my_first_image` соответствует тег `latest`.

Примечание. Так как образы занимают место в памяти компьютера, неактуальные образы можно удалять с помощью команды `docker rmi` (от англ. `remove image`):

```
$ docker rmi <image_id>
```

где `image_id` — идентификатор образа (столбец `IMAGE_ID`).

## ШАГ 3. ЗАПУСКАЕМ DOCKER-КОНТЕЙНЕР

Попробуем запустить контейнер на основе нашего образа — для этого используется команда `docker run`:

```
$ docker run -it --rm --name=my_first_container my_first_image
```

Расшифруем ключи и аргументы этой команды:

- `-it` объединяет команды: `-i` оставляет строку для ввода, а `-t` выделяет терминал;
- параметр `--rm` автоматически удаляет контейнер после завершения его работы (в том числе при завершении с ошибкой) — это позволяет не хранить неактивные контейнеры;
- параметр `--name` назначает docker-контейнеру имя (мы задали имя `my_first_container`).

Но вот незадача — после запуска мы увидим ошибку Python:

```
Traceback (most recent call last):
 File "./plot.py", line 4, in <module>
 import seaborn as sns
ModuleNotFoundError: No module named 'seaborn'
```

Эта ошибка говорит нам, что какие-то библиотеки не установлены. Почему это произошло? Ответ — в самом определении Docker: это инструмент виртуализации. То есть

окружение Docker полностью изолировано от нашей операционной системы. Более того, внутри нашего образа и вовсе используется другая операционная система, а это значит, что внутри контейнера нет тех библиотек, которые есть на нашем компьютере.

В таком случае давайте добавим нужные зависимости в Dockerfile.

## ШАГ 4. ДОБАВЛЯЕМ БИБЛИОТЕКИ В КОНТЕЙНЕР

Для этого создадим файл requirements.txt рядом с Dockerfile. Укажем в нём необходимые библиотеки и их версии:

```
my_first_container
├── src
│ ├── output
│ └── plot.py
├── Dockerfile
└── requirements.txt
```

### *Файл requirements.txt*

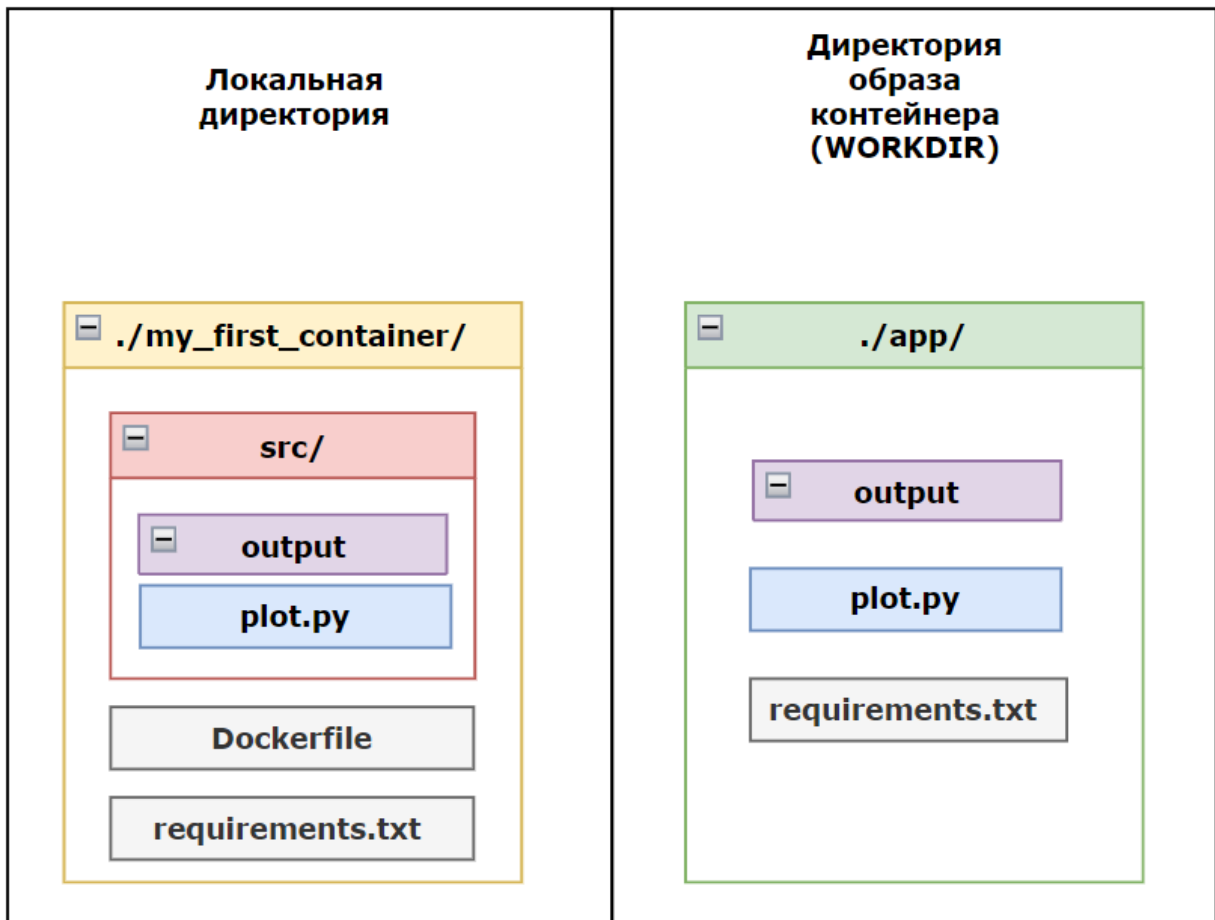
```
numpy == 1.23.4;
matplotlib == 3.6.0;
seaborn == 0.11.2.
```

Теперь необходимо переместить файл с зависимостями в наш контейнер, а после запустить команду для установки зависимостей.

Для этого напишем перед командой CMD, запускающей выполнение скрипта в Dockerfile, строки:

```
COPY ./requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
```

Директива COPY нам уже знакома: она позволяет копировать файлы из локальной директории в файловую систему контейнера.



Директива RUN позволяет запускать любые команды по аналогии с терминалом. Например, поскольку мы работаем на основе базового образа Python, мы можем воспользоваться менеджером пакетов `pip` для установки зависимостей.

Также для команды `pip install` мы указываем:

- параметр `--no-cache-dir` — позволяет не использовать кэш, а скачать пакеты заново. Вы можете не указывать этот параметр, если уверены, что в дальнейшем версии используемых библиотек не изменятся.
- ключ `-r` — указывает на файлы с зависимостями.

Тогда наш итоговый `Dockerfile`, на основе которого будет собираться образ контейнера, будет выглядеть следующим образом:

#### Файл `./Dockerfile`

```
FROM python:3.9
WORKDIR /usr/src/app
COPY ./src/ ./
COPY ./requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
CMD ["python", "./plot.py"]
```

Теперь нам необходимо заново собрать образ (сборка займёт некоторое время):

```
$ docker build -t my_first_image .
```

После этого можно запустить контейнер на основе этого образа:

```
$ docker run -it --rm --name=my_first_container my_first_image
```

**Примечание.** Для того чтобы посмотреть список запущенных контейнеров, можно воспользоваться командой `docker ps`. По умолчанию она выводит список активных контейнеров. Для вывода списка всех контейнеров используйте ключ `-a`, но предварительно перезапустите контейнер без ключа `--rm`, так как данный ключ удаляет контейнеры.

```
$ docker ps -a
```

Запустите команду в соседнем терминале одновременно с контейнером.

Она выводит на экран информацию о контейнерах в виде таблицы со следующими столбцами:

- CONTAINER ID — идентификатор контейнера;
- IMAGE — имя образа, на основе которого запущен контейнер;
- COMMAND — команда, используемая внутри контейнера (то, что мы прописали в директиве CMD в Dockerfile);
- CREATED — когда был запущен контейнер;
- STATUS — статус контейнера;
- PORTS — порты, которые использует контейнер (о них поговорим в следующем юните — сейчас наше приложение не использует веб-интерфейс, и портов у него не будет);
- NAMES — имя контейнера.

Теперь всё работает корректно. После запуска контейнера вас попросят ввести среднее и стандартное отклонение. После исполнения контейнера на экран должна быть выведена фраза "Файл успешно сохранен".

Однако файл `plot.png` в папке `output` не обновился с выполнением скрипта в контейнере. Куда же он тогда «успешно сохранён»?

## ШАГ 5. СИНХРОНИЗАЦИЯ ПУТЕЙ, ИЛИ КУДА ПРОПАЛ PLOT.PNG

Технически файл `plot.png` удалился вместе с контейнером после того, как запустился и выполнен скрипт `plot.py`. Docker не сохраняет файлы внутри контейнера, так как внутри контейнера своя отдельная файловая система.

Что же делать? Ответ очень прост — нам нужно связать контейнер с локальной файловой системой на нашем компьютере.

Для этого укажем параметр `--volume` или ключ `-v` в команде `docker run`.

Ключ `-v` требует указания путей, которые записываются в формате `::`

```
$ docker run -it --rm -v $PWD/src/output:/usr/src/app/output --name=my_first_container my_first_image
```

Если всё сделано правильно, после выполнения команды в папке `output` появится нужный график.

## ШАГ 6. ЗАГРУЖАЕМ ОБРАЗ НА DOCKER HUB

Большой успех Docker во многом обусловлен возможностью делиться образами контейнеров в Docker Hub.

Принцип загрузки образа в Docker Hub очень схож с загрузкой кода на GitHub:

- создаем локальный образ контейнера;
- делаем `push` образа на Docker Hub.

Пользователь со своей стороны может:

- сделать `pull` нашего образа;
- запустить образ контейнера на своей локальной машине.

Если мы правильно собрали образ контейнера, то благодаря технологии виртуализации пользователю не нужно знать, как устроено наше приложение, на какой операционной системе оно работает, какие зависимости в нём установлены — ему нужно просто запустить наш образ на своей локальной машине. Это значительно упрощает разработку внутри команды и выведение конечных решений в продакшн.

Рассмотрим механизм загрузки образов в Docker Hub по шагам.

1. Первым делом зарегистрируйтесь на Docker Hub, если не сделали этого ранее. Далее необходимо залогиниться под своей учётной записью в самом Docker. Для этого наберите в терминале команду:

```
$ docker login
```

Если вы ранее залогинились в Docker Desktop, данные учётной записи подгрузятся автоматически. Если вы не устанавливали Docker Desktop и не логинились в нём, Docker попросит вас ввести данные учётной записи на Docker Hub.

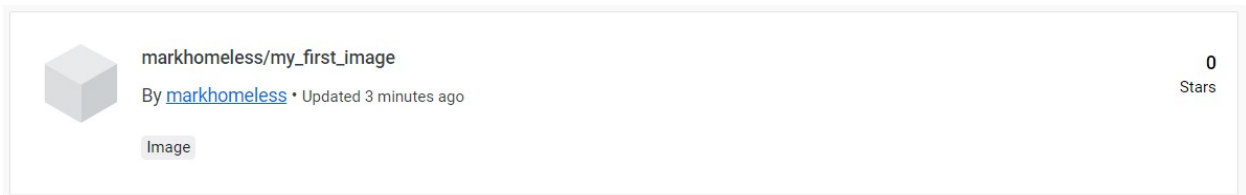
2. При отправке образа в Docker Hub необходимо указать имя пользователя как часть имени образа, так как Docker Hub организует репозитории по имени пользователя. Любой репозиторий, созданный под учётной записью, включает имя пользователя в имя образа Docker. Поэтому нам необходимо пересобрать наш образ `my_first_image`, задав ему имя в формате `/server_name`, где — это ваше имя пользователя в профиле на Docker Hub.

```
$ docker build -t <username>/my_first_image .
```

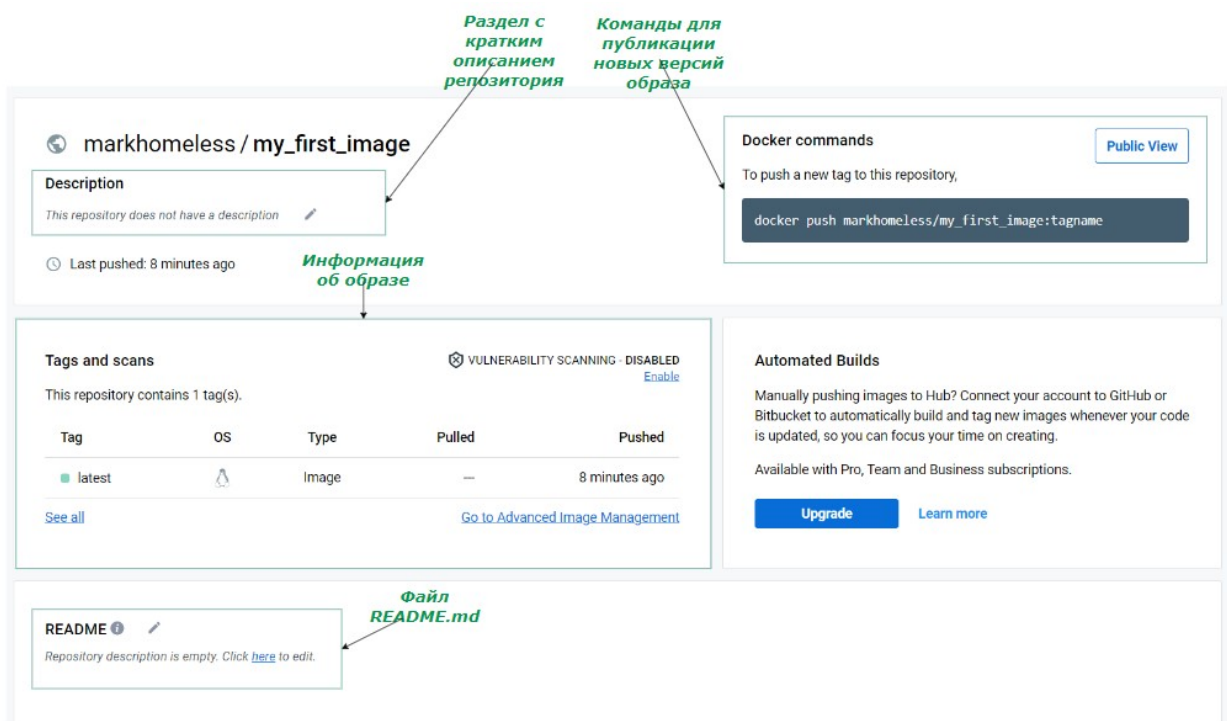
- Далее делаем push нашего образа на Docker Hub. В аргументах команды необходимо указать имя образа, которым мы хотим поделиться:

```
$ docker push <username>/my_first_image
```

- Далее откройте ваш профиль на Docker Hub — в нём должен был появиться новый репозиторий с именем /my\_first\_image:



В открывшемся окне вы увидите страницу своего docker-репозитория. Здесь вновь всё очень похоже на GitHub: есть раздел с кратким описанием образа (Description), а также файл README.md.



**Примечание.** При желании в файле README.md вы можете на языке Markdown описать суть образа, а также команды, которые необходимы для его запуска — всё как с GitHub-репозиториями.

После того как образ выложен на Docker Hub, любой пользователь может скачать его и воспользоваться им. Для этого используется команда `docker pull`, в аргументах которой указывается имя репозитория на Docker Hub:

```
$ docker pull <username>/my_first_image
```

После этого образ контейнера станет доступным для запуска (это можно отследить с помощью команды `docker images`), и его можно запустить стандартной командой `run`:

```
$ docker run -it --rm -v $PWD/src/output:/usr/src/app/output --name=my_first_container my_first_image
```

Обратите внимание: для запуска образов, которые мы получаем через `pull`, не нужно их собирать, то есть нам не нужен `Dockerfile` (данный файл по определению предназначен для сборки контейнера). Контейнер уже собран, и нам достаточно запустить его.

НА ЭТОМ ВСЁ?

Иногда нам может потребоваться задать переменные среды, которые используются для определения констант. Чтобы задать их, в `Dockerfile` существует инструкция `ENV` — благодаря ей значение будет доступно в контейнере во время его выполнения.

**Переменными средами** в Linux-подобных системах называются переменные, которые определяются на уровне оболочки и используются различными приложениями во время выполнения. Подробнее о них можно узнать в [статье](#).

Синтаксис `ENV` можно найти в [официальной документации](#).

В `Dockerfile` нужно добавить:

```
ENV <key> <value>
```

или

```
ENV <key>=<value> ...
```

Или, к примеру, при запуске контейнера:

```
$ docker run --env <key>=<value>
```

Например, мы хотим, чтобы в нашем контейнере появилась переменная среды `NAME`, в которой будет указываться имя разработчика контейнера. К этой переменной среды можно будет обратиться из любой части контейнера. Тогда в `Dockerfile` необходимо добавить объявление такой переменной среды:

**Файл `./Dockerfile`**

```
FROM python:3.9
ENV NAME="Skillfactory"
WORKDIR /usr/src/app
COPY ./src/ ./
COPY ./requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
CMD ["python", "./plot.py"]
```

## ПРОМЕЖУТОЧНЫЙ ВЫВОД ПО РАБОТЕ С DOCKER

В этом юните мы на примере маленького приложения рассмотрели, как создавать собственные образы, а также основные директивы `Dockerfile`. Запоминать последние не

обязательно — вы всегда можете найти их в [документации](#) или в этом [гайде](#). Также мы изучили основные команды, необходимые для работы с Docker.

Перечислим их (и пару дополнительных) ещё раз:

- `docker build` — создать образ контейнера;
- `docker images` — посмотреть список доступных образов;
- `docker rmi` — удалить образ;
- `docker run` — запустить контейнер;
- `docker stop` — остановить контейнер;
- `docker rm` — удалить контейнер;
- `docker ps` — посмотреть список запущенных контейнеров;
- `docker login` - залогиниться в учётной записи Docker Hub;
- `docker push` - отправить образ на Docker Hub;
- `docker pull` - скачать образ с Docker Hub;
- `docker logs` — посмотреть логи;
- `docker kill` — экстренно завершить процесс.

**Полезно!** У Docker достаточно богатый CLI (Command-Line Interface), который содержит множество команд, помимо тех, что перечислены в модуле. Больше команд и информации о них вы можете увидеть в [официальной документации](#).

## Задание 5.5

Теперь вам предстоит выполнить упражнение с Docker самостоятельно.

1. Мы подготовили для вас специальный базовый образ `mike0sv/sf_docker_base`, в котором зашит секретный код. Чтобы его узнать, вам необходимо создать свой образ на основе базового и добавить в него текстовый файл с фразой `I know how to use docker`.
2. Создайте переменную среды с названием (key) `DATA_PATH`. Переменная должна содержать добавленный вами путь к файлу (value) внутри контейнера (относительный или абсолютный).
3. Запустите контейнер. Если вы всё сделали правильно, он выведет секретный код.

**text.txt:**

```
I know how to use docker
```

**Dockerfile:**

```
FROM mike0sv/sf_docker_base
COPY ./text.txt ./
ENV DATA_PATH ./text.txt
```

**Команда для сборки образа:**



```
$ docker build -t task_image .
```

**Команда для запуска контейнера:**

```
$ docker run --rm task_image
```

## 6. Создаём образ веб-сервиса

Теперь давайте завернём в контейнер веб-сервис для нашего коллеги Василия, а затем загрузим его на Docker Hub, чтобы этот сервис был доступен для использования и Василий смог развернуть его у себя.

### ШАГ 1. ЗАВОРАЧИВАЕМ FLASK-ПРИЛОЖЕНИЕ В КОНТЕЙНЕР

Прежде всего, давайте ещё раз договоримся о расположении файлов в нашей директории. Вынесем файлы с кодом сервера (server.py и папка models) и клиентским приложением (client.py) в папки app и test соответственно. Это нужно для удобства, чтобы не прописывать несколько команд COPY в Dockerfile.

```
web
├── app
│ ├── models
│ │ └── model.pkl
│ └── server.py
├── test
│ └── client.py
├── Dockerfile
└── requirements.txt
```

**Примечание.** Также в папке web может находиться папка project\_venv с конфигурацией вашего виртуального окружения — мы намеренно её опустили, так как она не будет участвовать в создании контейнера. Нам понадобится только файл requirements.txt, который мы создавали ранее.

**Момент, который необходимо иметь в виду:** когда мы запускаем веб-сервис в контейнере, для тестирования сервера необходимо пользоваться широковещательным адресом (0.0.0.0) — это позволит вам обеспечить доступ к вашему контейнеру. Таким образом, измените в файле server.py следующий код, запускающий сервер:

```
if __name__ == '__main__':
 app.run(host='0.0.0.0', port=5000)
```

Теперь, когда мы зафиксировали все вводные, можно переходить к созданию Dockerfile. Это довольно просто, поэтому мы просто приведём описание шагов ниже, а вы с их помощью самостоятельно составите Dockerfile:

1. Оставить те же базовый образ (python:3.9) и рабочую директорию (/usr/src/app).

2. Скопировать локальную папку ./app в рабочую директорию контейнера.
3. Скопировать файл с зависимостями requirements.txt в рабочую директорию проекта.
4. Выполнить команду python ./server.py для запуска сервера.

```
FROM python:3.9
WORKDIR /app
COPY ./app ./
COPY ./requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
CMD ["python", "./server.py"]
```

Собираем образ:

```
docker build -t flask_image .
```

Теперь, когда у нас есть собранный контейнер, нам осталось лишь запустить его. Как мы знаем, это делается с помощью команды docker run. Единственное отличие от предыдущего контейнера: так как в нашем контейнере будет работать веб-приложение, нам необходимо открыть внешним приложениям доступ к порту, на котором запущено приложение. Это делается с помощью указания ключа --publish или его сокращённой версии -p. В аргументе этой команды указывается диапазон из номеров портов контейнера, которые мы хотим сделать публичными. В нашем приложении мы указали, что нам нужен порт 5000, а значит, диапазон будет выглядеть как 5000:5000. Общий синтаксис команды:

```
$ docker run -it --rm --name=server_container -p=5000:5000 flask_image
```

После запуска контейнера вы должны увидеть примерно следующее сообщение:

```
* Serving Flask app 'server'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.17.0.2:5000
```

- Первый адрес — localhost (<http://127.0.0.1:5000> или <http://localhost:5000>), к нему мы уже привыкли.
- Второй адрес — адрес вашего контейнера (у вас он может отличаться). По нему контейнеры могут «общаться» друг с другом.

Далее веб-сервер перейдёт в режим прослушивания сети и вы сможете проверить, что он работает, с помощью GET-запроса, обратившись по адресу <http://127.0.0.1:5000> через поисковую строку браузера. В браузере должно быть выведено сообщение "Test message. The server is running".

После этого выполните POST-запрос, запустив код клиентского приложения в соседнем терминале:

- UNIX:

```
$ python3 tests/client.py
```

- WINDOWS:

```
$ python tests/client.py
```

Если в результате выполнения скрипта вернулся код 200 и было выведено сообщение с предсказанием модели, поздравляем — ваше Flask-приложение запущено в контейнере, и вы можете к нему обращаться.

**Примечание.** Кстати, можно запускать контейнер в фоновом режиме. Для этого добавьте к команде `docker run` ключ `-d` (от англ. `detach` — отделить):

```
$ docker run -it --rm --name=flask_container -publish=5000:5000 flask_image
```

Итак, наше Flask-приложение запущено, но, как мы знаем из предыдущего модуля, обычного Flask-приложения недостаточно для полноценного веб-сервера. В продакшене используются WSGI-серверы, такие как uWSGI + NGINX (см. модуль [«PROD-1. Подготовка модели к продакшену и деплой»](#)).

Давайте реализуем связку Flask + uWSGI + NGINX в контейнере.

**Примечание.** Перед тем как перейти к следующему шагу, остановите контейнер `flask_image` (CTRL + C), так как он будет конфликтовать с контейнером, который мы создадим далее.

## ШАГ 2. FLASK + UWSGI + NGINX + DOCKER

Если вы проходили [юнит](#) предыдущего модуля, посвящённый uWSGI и NGINX, вы должны были понять, что настройка этих инструментов — непростая задача, как минимум потому, что запустить uWSGI можно только в UNIX-системах, что ограничивает разработку. К тому же, необходимо прописывать конфигурации серверов uWSGI и NGINX, связывать их работу через сокеты или TCP-порты и многое другое.

Конечно, всё это можно сделать и в контейнере, базовым образом которого является операционная система Linux (например, Ubuntu). Однако в этом случае Dockerfile был бы очень сложным и нам потребовалось бы разбираться в операционной системе внутри контейнера.

К счастью, на Docker Hub выложены образы контейнеров, в которых уже настроена вся необходимая конфигурация uWSGI и NGINX. Всё, что от нас требуется — заменить файл конфигурации uWSGI, который лежит внутри контейнера, на тот, который нужен нам, а также поместить в контейнер исходный код с интерфейсом сервера, который будет выполняться сервером uWSGI.

Давайте перепишем наш Dockerfile.

Базовый образ, который мы будем использовать для своего контейнера, — [tiangolo/uwsgi-nginx-flask](#). Это не официальный образ, но, несмотря на это, у него более миллиона скачиваний. Не волнуйтесь — мы проверили образ, он безопасен.

Образ `tiangolo/uwsgi-nginx-flask` создан на базе ОС Linux, и в нём уже настроено взаимодействие серверов uWSGI и NGINX через сокеты. Более подробно о нём вы можете прочитать в [документации](#).

В рабочей директории образа уже лежат файлы некоторого приложения. Давайте проверим, что образ работает — напомним Dockerfile, состоящий только из импорта базового образа (через ":" указывается версия Python, которую мы будем использовать):

#### Файл `./Dockerfile`

```
FROM tiangolo/uwsgi-nginx-flask:python3.9
```

Соберём образ контейнера и назовём его `server_image`:

```
$ docker build -t server_image .
```

Примечание. При выполнении этой команды в ОС Linux у вас может возникнуть ошибка примерно следующего содержания:

```
ERROR: UtilConnectToInteropServer:307: connect failed 2
Got permission denied while trying to connect to the Docker daemon
socket at unix:///var/run/docker.sock: Post "http://%2Fvar%2Frun
%2Fdocker.sock/v1.24/build?buildargs=%7B%7D&cachefrom=%5B
%5D&cgroupparent=&cpuperiod=0&cpuquota=0&cpusetcpus=&cpusetmems=&cpush
ares=0&dockerfile=Dockerfile&labels=%7B
%7D&memory=0&memswap=0&networkmode=default&rm=1&shmsize=0&t=server_ima
ge&target=&ulimits=null&version=1": dial unix /var/run/docker.sock:
connect: permission denied
```

Она говорит о том, что у контейнера недостаточно прав для доступа к сокетам, через которые общаются uWSGI и NGINX. Для исправления ошибки просто добавьте к команде префикс `sudo`, чтобы она выполнялась от имени главного пользователя `root`:

```
$ sudo docker build -t server_image .
```

Затем запустим контейнер на порте 80:

```
$ docker run -it --rm --name=server_container -p=80:80 server_image
```

В результате выведется огромное информационное сообщение, в котором описаны все настройки, с которыми был запущен сервер. После этого сервер уйдёт в режим ожидания запросов.

Проверим, что сервер работает. Для этого перейдём по адресу <http://localhost> или <http://127.0.0.1>. Обратите внимание, что номер порта указывать необязательно, так как мы запустили сервер на порте 80.

В результате вы должны увидеть в браузере примерно следующее сообщение:



## Hello World from Flask in a uWSGI Nginx Docker container with Python 3.9 (default)

Итак, внутри этого образа уже работает какое-то Flask-приложение. Нам лишь нужно заменить его на своё.

Для начала давайте выясним, что представляет собой содержимое этого контейнера. Для этого, не останавливая работу контейнера, в соседнем терминале наберём следующую команду:

```
$ docker exec -it server_container bash
```

После её выполнения мы попадём в файловую систему, окажемся в рабочей директории нашего контейнера и сможем проанализировать его содержимое. Например, выполним команду `ls`:

```
$ ls
__pycache__ main.py prestart.sh uwsgi.ini
```

**Примечание.** Чтобы посмотреть содержимое файлов, можно воспользоваться командой `cat`. Например, следующая команда позволит вывести содержимое файла `main.py`:

```
$ cat main.py
```

Что мы видим? Внутри контейнера есть:

- `main.py` — файл с кодом Flask-приложения;
- `uwsgi.ini` — файл с настройками uWSGI-сервера;
- `prestart.sh` — bash-скрипт для запуска;
- `__pycache__` — папка с кэшем.

Итак, сервер uWSGI в качестве своего интерфейса использует код из файла `main.py`. Наша задача — заменить файл `uwsgi.ini` на свой, где в качестве исполняемого приложения будет наш файл `server.py`.

Давайте создадим в корневом каталоге нашего проекта файл `uwsgi.ini`. В этом файле мы пропишем, что будем использовать объект `app` из модуля `server.py`, количество процессов, обрабатывающих поступающие на сервер запросы, а также добавим `master`-поток, который будет координировать работу потоков.

### Файл `./uwsgi.ini`

```
[uwsgi]

module = server:app
processes = 4
master = true
```

**Примечание.** Остальные параметры uWSGI, такие как сокет для взаимодействия с NGINX, уже заданы в образе по умолчанию — изменять их не нужно.

Теперь допишем наш Dockerfile.

### Файл ./Dockerfile

```
Задаём базовый образ
FROM tiangolo/uwsgi-nginx-flask:python3.9
Копируем содержимое папки ./app в рабочую директорию контейнера
COPY ./app ./
Копируем файл requirements.txt в рабочую директорию контейнера
COPY ./requirements.txt ./
Копируем файл uwsgi.ini в рабочую директорию контейнера
COPY ./uwsgi.ini ./
Запускаем установку необходимых зависимостей
RUN pip install -r requirements.txt
Заново соберём наш образ:
```

```
$ docker build -t server_image .
```

В результате копирования файла uwsgi.ini из локальной директории в файловую систему контейнера мы перезаписали файл uwsgi.ini, который находился в контейнере, на свой вариант, и теперь сервер uWSGI будет работать по нашим правилам.

Давайте убедимся в этом, перезапустив контейнер:

```
$ docker run -it --rm --name=server_container -p=80:80 server_image
```

Теперь в соседнем терминале посмотрим содержимое контейнера:

```
$ docker exec -it server_container bash
```

```
$ ls
__pycache__ main.py models prestart.sh requirements.txt server.py
uwsgi.ini
```

Как видите, теперь на сервере для обработки поступающих запросов вместо файла main.py используется файл server.py.

Давайте убедимся, что наш сервер работает. Протестируем GET-запросы: для этого через браузер зайдём по адресу <http://localhost> или <http://127.0.0.1>. В результате должно быть выведено сообщение, которое мы прописывали в файле server.py, — "Test message. The server is running". Далее выполним POST-запрос. Помним, что теперь наш сервер работает на порте 80, а не на порте 5000, поэтому в скрипте для отправки запросов изменим URL, по которому происходит отправка запросов на '<http://localhost/predict>'. После внесения изменений запустим код клиентского приложения в соседнем терминале.

- UNIX:

```
$ python3 tests/client.py
```

- WINDOWS:

```
$ python tests/client.py
```

Если в результате выполнения скрипта вернулся код 200 и было выведено сообщение с предсказанием модели, поздравляем — ваша связка Flask + uWSGI + NGINX работает в контейнере Docker.

Подведём **промежуточный итог**. В результате объединения Flask, uWSGI и NGINX мы получили мощный веб-сервер, способный обрабатывать несколько запросов одновременно, обладающий высокой пропускной способностью, но, что самое важное — этот сервер функционирует в контейнере, а значит, мы легко можем переносить его с одной машины на другую, не беспокоясь о вопросах воспроизводимости и совместимости операционных систем. Для этого нам достаточно разместить результирующий образ на Docker Hub, а затем с любой машины, на которой установлен Docker и есть интернет, можно скачать этот образ, собрать из него контейнер и запустить его.

Давайте разметим наш образ на Docker Hub.

## ШАГ 3. ДЕЛАЕМ PUSH ВЕБ-СЕРВИСА НА DOCKER HUB

Залогиньтесь в учётной записи Docker, если вы не сделали этого ранее, с помощью команды `docker login`.

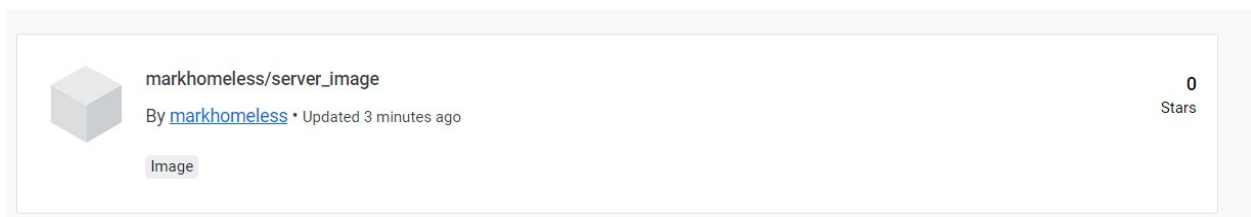
Нам необходимо пересобрать образ, задав ему имя в формате `/server_name`, где — имя пользователя в профиле на Docker Hub.

```
$ docker build -t <username>/server_image .
```

Образ собран — осталось только загрузить его на Docker Hub:

```
$ docker push <username>/server_image
```

Откройте ваш профиль на Docker Hub — в нём должен был появиться новый образ с именем `/server_image`:



Теперь, чтобы наш коллега Василий смог запустить веб-сервис со своей ML-моделью, ему будет достаточно сделать pull нашего образа контейнера и запустить его. Для справки приведём команды, которые должны быть выполнены на его компьютере:

```
$ docker pull <username>/server_image $ docker run -it --rm --name=server_container -p=80:80 server_image
```

Согласитесь, очень просто и удобно!

## 7. Итоги

Мы с вами проделали большую работу для обеспечения воспроизводимости своего кода.

### В ЭТОМ МОДУЛЕ МЫ:

- узнали, что такое воспроизводимость и какими инструментами её можно обеспечить;
- познакомились с терминами «виртуализация» и «контейнеризация»;
- научились создавать виртуальные окружения, изолировать среду разработки, устанавливать и фиксировать зависимости в виртуальных окружениях;
- рассмотрели инструмент контейнеризации Docker;
- научились писать Dockerfile, создавать образы контейнеров, запускать их, а также делиться ими с помощью хостинга docker-образов Docker Hub;
- создали docker-образ для нашего веб-сервиса и запустили его в контейнере.

Поместить модель и использующий её веб-сервис в контейнер — это не единственная цель Docker. В реальных системах такой сервис будет лишь частью глобальной системы, состоящей из нескольких или даже нескольких десятков подобных сервисов. Компоненты, из которых будет состоять эта система, называются **микросервисами**, и архитектура такой системы будет по аналогии называться **микросервисной**. Следующий модуль мы посвятим организации микросервисной архитектуры и взаимодействия между её компонентами: мы увидим, как сервисы, внутри которых функционируют модели машинного обучения, становятся частью большой системы, и поговорим о том, как управлять такой системой.

### ДОПОЛНИТЕЛЬНО

- [Официальная документация Virtualenv](#).
- [Обзор инструментов для создания виртуальных окружений в Python \(помимо Virtualenv\)](#).
- [Список команд Docker](#).
- [Материалы по работе с Docker от его амбассадоров](#).
- [Цикл статей «Разворачиваем модель машинного обучения с Docker»](#).
- [Цикл видео на Youtube, посвящённый работе с Docker](#).
- [Видео “Docker Compose in 12 Minutes” \(на английском языке\)](#).