# Что такое Flask?

**Flask** — это легковесный веб-фреймворк для Python, который позволяет быстро создавать веб-приложения и API. Его основные преимущества — простота и гибкость.

**Установка Flask**

Чтобы начать работать с Flask, установите его через pip:

bash

pip install flask

# Настройка виртуального окружения

Использование одного и того же окружения в нескольких проектах может привести к ошибкам, например к несовместимости версий библиотек. Чтобы предотвратить подобные конфликты, используют механизм **виртуального окружения**.

Для этого в Python существует специальный модуль venv, который позволяет работать с виртуальным окружением.

Рассмотрим основные операции подробнее.

|  |  |  |
| --- | --- | --- |
| **Операция \ ОС** | **Windows** | **Linux / macOS** |
| Создание | python -m venv .venv | python3 -m venv .venv |
| Активация | .\.venv\Scripts\activate.bat | source ./.venv/bin/activate |
| Деактивация | deactivate | deactivate |

# **Создание виртуального окружения**

Модуль venv обычно поставляется вместе с интерпретатором Python на macOS и Windows. На Ubuntu может потребоваться отдельная установка:

$ sudo apt install python3-venv

Чтобы проверить, что модуль установлен, напишите следующую команду:

$ python -m venv *--help*  
*usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear] [--upgrade] [--without-pip] [--prompt PROMPT] [--upgrade-deps] ENV\_DIR [ENV\_DIR ...]*

Для создания виртуального окружения перейдите в директорию с проектом и воспользуйтесь следующей командой:

$ python -m venv .venv

В текущей рабочей директории будет создана папка .venv.

Активация виртуального окружения

Для активации воспользуйтесь следующей командой:

$ source ./.venv/bin/activate

или в Windows в консоли cmd:

> .\.venv\Scripts\activate.bat

После активации должен появиться специальный маркер такого вида:

(.venv) $

Теперь можно устанавливать библиотеки, которые будут храниться в этом виртуальном окружении.

(.venv) $ pip install Flask

Хорошая практика — указывать версию устанавливаемой библиотеки. Их разработчики тоже могут допускать ошибки, поэтому лучше довериться стабильному и проверенному временем варианту. Например, так можно установить версию 2.2.3:

(.venv) $ pip install Flask==2.2.3

# **Список установленных библиотек**

Для получения всех установленных библиотек в этом окружении можно использовать следующую команду:

(.venv) $ pip freeze  
click==8.1.3  
Flask==2.2.3  
itsdangerous==2.1.2  
Jinja2==3.1.2  
MarkupSafe==2.1.2  
Werkzeug==2.2.3

Библиотеки можно перенаправить в файл requirements.txt, используемый для хранения этих зависимостей.

(.venv) $ pip freeze > requirements.txt

С помощью знака > мы перенаправляем результат команды pip freeze в файл requirements.txt.

# **Установка зависимостей**

Если вы клонировали чужой проект, то все зависимости из файла requirements.txt можно установить одной командой:

(.venv) $ pip install -r requirements.txt

С помощью параметра -r мы говорим, что библиотеки, которые нужно установить, находятся в файле requirements.txt.

# **Деактивация виртуального окружения**

Чтобы выйти из виртуального окружения, достаточно ввести следующую команду:

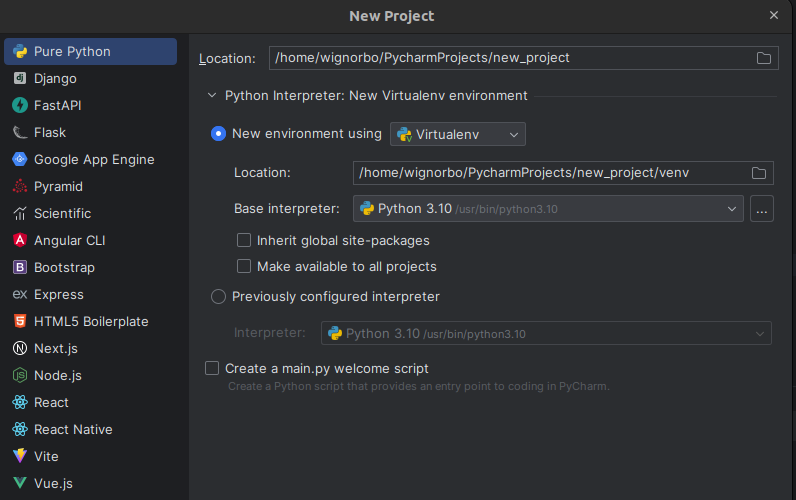
(.venv) $ deactivate

После исполнения должен пропасть маркер с названием виртуального окружения — это значит, что вы успешно вышли из него.

# **Работа с виртуальным окружением в PyCharm**

PyCharm позволяет упростить взаимодействие с виртуальным окружением.

При создании проекта вам сразу предложат выбрать интерпретатор:



Чтобы настроить виртуальное окружение в имеющемся проекте, нажмите File → Settings → Project Settings → Python Interpreter. Должно появиться такое окно:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Далее нажмите Add Interpreter → Add Local Interpreter. Появится следующее окно:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

В поле Location вы указываете путь к виртуальному окружению. В этом случае оно будет находиться в корне проекта и называться venv.

В поле Base interpreter вы выбираете интерпретатор, который будет лежать в основе вашего виртуального окружения. В данном случае — Python 3.10.

Нажмите OK, а затем Apply. Теперь все библиотеки будут устанавливаться в указанное виртуальное окружение.

Открыв терминал, вы увидите маркер, который указывает, что виртуальное окружение активировано:

Изображение выглядит как снимок экрана, текст, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

# **Как запустить приложение Flask на операционной системе Windows:**

1. Используя консоль cmd создайте виртуальное окружение с помощью команды python -m venv venv
2. Активируйте виртуальное окружение c помощью команды venv\Scripts\activate.bat
3. Установите Flask с помощью команды pip install Flask
4. Проверьте, что Flask действительно установлен в системе с помощью команды pip freeze | findstr /i flask
5. Перейдите в ваш личный репозиторий с кодом с помощью кнопки «Перейти в GitLab» в практической работе.
6. Клонируйте репозиторий с кодом себе на компьютер с помощью команды git clone <https://gitlab.skillbox.ru/><ВАШ\_НИКНЕЙМ>/python\_advanced.git
7. Перейдите в директорию first\_application с кодом Flask приложения с помощью команды cd module\_01\_flask\materials\first\_application
8. С помощью setx добавьте переменные окружения \*FLASK\_APP и FLASK\_DEBUG с помощью команд  
   setx FLASK\_APP "app.py"  
   setx FLASK\_DEBUG 1
9. Запустите Flask с помощью команды python -m flask run --port=5555
10. Перейдите на<http://127.0.0.1:5555/test>

# **unit тесты**

При запуске тестов из консоли вы можете столкнуться со следующей ошибкой:

$ python3 -m unittest test\_hello\_word\_with\_day.py  
...  
ModuleNotFoundError: No module named 'module\_03\_ci\_culture\_beginning'

Эта ошибка возникает, потому что Python не знает, где искать модуль module\_03\_ci\_culture\_beginning. Чтобы это исправить, нужно изменить переменную окружения PYTHONPATH.

Например, модуль module\_03\_ci\_culture\_beginning находится в директории python\_advanced. Тогда в PYTHONPATH нужно добавить путь к этой директории:

export PYTHONPATH=$PYTHONPATH:/home/wignorbo/PycharmProjects/python\_advanced

В Windows это можно сделать аналогичной командой set:

$ set PYTHONPATH=%PYTHONPATH%;C:\Users\wignorbo\PyCharmProjects\python\_advanced

Запустим тест и убедимся, что всё в порядке:

$ python3 -m unittest test\_hello\_word\_with\_day.py  
.  
----------------------------------------------------------------------  
Ran 1 test in 0.007s  
OK

# **Четыре метода класса TestCase**

На практике можно часто встретить методы setUp, setUpClass, tearDown и tearDownClass. Давайте узнаем, что они делают и для чего они нужны.

В качестве примера для тестирования будем использовать класс Student в файле models.py:

class Student:  
    def \_\_init\_\_(self, name=None, age=None):  
        self.name = name  
        self.age = age  
  
    def set\_age(self, value):  
        if value <= 0:  
            raise ValueError("Возраст должен быть положительным числом")  
        self.age = value

## setUp

Метод вызывается **перед запуском каждого теста** и позволяет не писать один и тот же код по созданию экземпляра тестируемого класса. Сравните код с setUp и без него.

**Код без использования setUp**

В каждом тесте создаём экземпляр студента.

from unittest import TestCase  
from models import Student  
class StudentTestCase(TestCase):  
    def test\_default\_name\_is\_none(self):  
        student = Student()  
        self.assertIsNone(student.name)  
    def test\_set\_invalid\_age(self):  
        student = Student()  
        with self.assertRaises(ValueError):  
            student.set\_age(-100)

**Код с использованием setUp**

Перед запуском каждого теста заново создаём экземпляр студента, который можно будет использовать в тестах.

from unittest import TestCase  
from models import Student  
  
class StudentTestCase(TestCase):  
    def setUp(self):  
        self.student = Student()  
  
    def test\_default\_name\_is\_none(self):  
        self.assertIsNone(self.student.name)  
  
    def test\_set\_invalid\_age(self):  
        with self.assertRaises(ValueError):  
            self.student.set\_age(-100)

Метод assertIsNone проверяет, что значение переменной равно None. Метод assertRaises проверяет, что определённое исключение было вызвано в процессе выполнения программы.

## Перечень наиболее популярных assert-методов:

|  |  |
| --- | --- |
| **Метод** | **Проверяет** |
| assertEqual(a, b) | a == b |
| assertNotEqual(a, b) | a != b |
| assertTrue(x) | bool(x) is True |
| assertFalse(x) | bool(x) is False |
| assertIs(a, b) | a is b |
| assertIsNot(a, b) | a is not b |
| assertIsNone(x) | x is None |
| assertIsNotNone(x) | x is not None |
| assertIn(a, b) | a in b |
| assertNotIn(a, b) | a not in b |
| assertIsInstance(a, b) | isinstance(a, b) |
| assertNotIsInstance(a, b) | not isinstance(a, b) |

Подробнее об assert-методах читайте в официальной документации.

## tearDown

Метод вызывается **после запуска каждого теста** и позволяет очистить или закрыть ресурсы. Например, удалить созданные в процессе тестирования файлы или закрыть подключение к базе данных.

В связке с setUp можно замерять время работы каждого теста:

import time  
from unittest import TestCase  
class PerformanceTest(TestCase):  
    def setUp(self):  
        self.start = time.perf\_counter()  
    def tearDown(self):  
        self.end = time.perf\_counter()  
        print(self.id(), self.end - self.start)  
    def test\_million\_appends(self):  
        N = 1\_000\_000  
        lst = []  
        for i in range(N):  
            lst.append(i)  
        self.assertListEqual(lst, list(range(N)))

time.perf\_counter() возвращает текущее время с наибольшим доступным разрешением.

self.id() выдаст название текущего теста в таком формате: \_\_main\_\_.PerformanceTest.test\_million\_appends

## setUpClass

Метод вызывается **перед запуском всех тестов** в этом классе. Он позволяет задать общие настройки и открыть ресурсы, которые будут использованы всеми тестами.

Здесь, например, мы можем создать тестовый клиент Flask-приложения:

from unittest import TestCase  
from hello\_word\_with\_day import app  
class TestHelloWorldWithDayApp(TestCase):  
    @classmethod  
    def setUpClass(cls):  
        app.config['TESTING'] = True  
        app.config['DEBUG'] = False  
        cls.app = app.test\_client()  
        cls.base\_url: str = '/hello-world/'  
    def test\_can\_get\_correct\_username\_with\_weekdate(self):  
        username: str = 'username'  
        response = self.app.get(self.base\_url + username)  
        response\_text: str = response.data.decode()  
        self.assertIn(username, response\_text)

Заметьте, что это классовый метод, поэтому мы используем декоратор classmethod.

## tearDownClass

Метод вызывается **после завершения всех тестов** в этом классе. Он позволяет закрыть ресурсы, которые были открыты в setUpClass. А также очистить данные, созданные в результате тестирования.

Например, в setUpClass мы можем открыть файл, в который будем складывать результаты тестирования. А в tearDownClass — закрыть его.

import time  
from unittest import TestCase  
class PerformanceTest(TestCase):  
    @classmethod  
    def setUpClass(cls):  
        cls.file = open("test\_log.txt", "a")  
    def setUp(self):  
        self.start = time.perf\_counter()  
    def test\_million\_appends(self):  
        N = 1\_000\_000  
        lst = []  
        for i in range(N):  
            lst.append(i)  
        self.assertListEqual(lst, list(range(N)))  
    def test\_sum\_of\_numbers(self):  
        N = 1\_000\_000  
        self.assertEqual(sum(range(N)), N \* (N + 1) // 2)  
    def tearDown(self):  
        self.end = time.perf\_counter()  
        print(self.id(), self.end - self.start, file=self.file)  
    @classmethod  
    def tearDownClass(cls):  
        cls.file.close()

В файле увидим следующий результат:

\_\_main\_\_.PerformanceTest.test\_million\_appends 0.09666280300007202

\_\_main\_\_.PerformanceTest.test\_sum\_of\_numbers 0.011858064999614726

## Заключение

Вспомним, для чего нужен каждый из рассмотренных методов:

* **setUp** запускается перед каждым тестом;
* **tearDown** запускается после каждого теста;
* **setUpClass** запускается перед всеми тестами;
* **tearDownClass** запускается после всех тестов.

В setUp и setUpClass мы открываем ресурсы, настраиваем тестируемый объект, создаём необходимые файлы, подключаемся к базе данных. В tearDown и tearDownClass — закрываем ресурсы, удаляем созданные файлы, отключаемся от базы данных.

Помимо четырёх рассмотренных методов, есть и другие. Почитайте о них в [документации](https://docs.python.org/3/library/unittest.html#unittest.TestCase).

# Pylint

Pylint — это мощный инструмент статического анализа кода для языка программирования Python. Он помогает разработчикам находить ошибки, улучшать качество кода и следовать стандартам кодирования, таким как PEP 8.

## Основные характеристики Pylint

* **Статический анализ**: Pylint анализирует код без его выполнения, что позволяет выявлять проблемы на этапе разработки.
* **Поддержка PEP 8**: Инструмент проверяет соответствие кода рекомендациям по стилю, описанным в PEP 8.
* **Обширные проверки**: Pylint выполняет множество проверок, включая:
  + Проверка длины строк.
  + Проверка имен переменных и классов.
  + Проверка наличия неиспользуемых переменных и импортов.
  + Определение логических ошибок и потенциальных уязвимостей.
* **Гибкость**: Pylint позволяет настраивать правила проверки, отключать определенные предупреждения и добавлять собственные.

## Установка

Для установки Pylint используйте pip:

bash

pip install pylint

Запуск Pylint

Для анализа файла с кодом выполните следующую команду в терминале:

bash

pylint имя\_файла.py

Пример вывода

При запуске Pylint вы получите отчет, содержащий:

* Список найденных проблем с указанием номера строки.
* Статистику по типам ошибок.
* Общую оценку качества кода по десятибалльной шкале.

## Интеграция с IDE

Pylint можно интегрировать с различными средами разработки (IDE), такими как:

* Visual Studio Code
* PyCharm
* Eclipse (с помощью PyDev)

Это позволяет получать мгновенные уведомления о проблемах прямо в процессе написания кода.

Преимущества использования Pylint

1. **Автоматизация**: Упрощает процесс проверки кода, позволяя разработчикам сосредоточиться на логике приложения.
2. **Улучшение качества кода**: Помогает поддерживать единообразие и чистоту кода, что облегчает его сопровождение.
3. **Обнаружение ошибок**: Позволяет находить ошибки до выполнения программы, что снижает вероятность возникновения проблем в рабочем коде.

## Заключение

Pylint является незаменимым инструментом для Python-разработчиков, желающих повысить качество своего кода и следовать лучшим практикам программирования. С его помощью можно эффективно выявлять ошибки и улучшать структуру проектов.

# Что такое GET-запросы?

**GET-запросы** — это тип HTTP-запросов, используемый для получения данных с сервера. Когда вы отправляете GET-запрос, сервер возвращает ответ, содержащий данные, которые были запрошены.

Например, при вводе URL в адресную строку браузера вы отправляете GET-запрос, чтобы получить содержимое страницы.

Пример GET-запроса:

***GET /index.html HTTP/1.1***

***Host: example.com***

**Особенности GET-запросов:**

1. **Параметры передаются в URL:** Они добавляются в строку запроса после ?, например:  
   https://example.com/search?query=python&lang=ru
2. **Данные доступны в адресной строке:** Они видимы, что делает GET неподходящим для передачи конфиденциальной информации.
3. **Запросы кэшируются браузером:** Это ускоряет повторные обращения.

**Создание простого Flask-приложения с GET-запросом**

Вот пример базового приложения Flask, которое обрабатывает GET-запросы:

python

КопироватьРедактировать

from flask import Flask, request

app = Flask(\_\_name\_\_)

# Обработка GET-запроса

@app.route('/greet', methods=['GET'])

def greet\_user():

# Получение параметров из строки запроса

name = request.args.get('name', 'Гость') # Параметр 'name', по умолчанию 'Гость'

return f'Привет, {name}! Добро пожаловать!'

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**Как это работает:**

1. **Маршрут /greet:** Указывает, что запросы по адресу /greet будут обрабатываться этой функцией.
2. **Метод GET:** Это означает, что обработка производится для GET-запросов.
3. **Получение параметров:** request.args.get('name') считывает параметр name из строки запроса. Например, при запросе:  
   http://127.0.0.1:5000/greet?name=Иван  
   сервер вернет: Привет, Иван! Добро пожаловать!

**Как запустить приложение:**

1. Сохраните код в файл, например, app.py.
2. Запустите приложение:

bash

python app.py

1. Откройте браузер и перейдите по адресу:  
   http://127.0.0.1:5000/greet?name=Иван

**Полезные советы:**

* **Для тестирования запросов** используйте инструменты, такие как [Postman](https://www.postman.com/) или curl в терминале.
* **Flask-обработчики поддерживают несколько методов (GET, POST и т.д.):** Если вам нужно обрабатывать разные типы запросов, укажите их через аргумент methods.
* Flask позволяет легко расширять функционал, подключая базы данных, шаблоны (например, Jinja2) и маршруты для обработки более сложных запросов.