# 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:

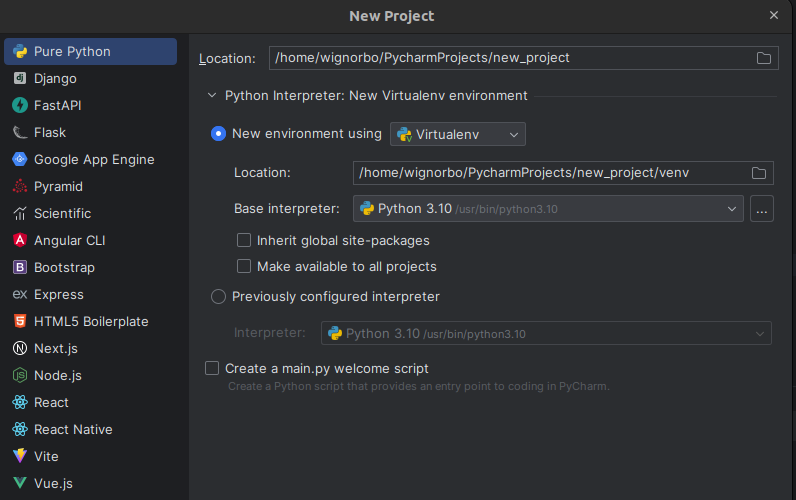
(.venv) $ deactivate

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.

# Что такое POST-запросы?

POST-запросы — это один из способов отправки данных через интернет от клиента (например, браузера) к серверу. Они используются, когда нужно передать какую-то информацию, например:

* Регистрационные данные (логин и пароль).
* Форму заказа на сайте.
* Загрузить файл на сервер.

Когда ты отправляешь POST-запрос, данные передаются в теле запроса, а не в адресной строке браузера. Это делает их чуть более безопасными, чем GET-запросы, где данные видны всем.

Пример:  
Ты заполняешь форму на сайте с полями «Имя» и «Сообщение», нажимаешь "Отправить", и эти данные отправляются на сервер с помощью POST-запроса.

В Python отправка POST-запроса обычно делается с помощью библиотеки requests:

***import requests***

***data = {'name': 'Alice', 'message': 'Hello, world!'}***

***response = requests.post('https://example.com/submit', data=data)***

***print(response.text) # Ответ сервера***

[Большой гайд по тестированию с Postman для начинающих](https://testengineer.ru/gajd-po-testirovaniyu-v-postman/?ysclid=luqqll00sd454434000)

[О максимальной длине HTTP GET запроса](https://stackoverflow.com/questions/2659952/maximum-length-of-http-get-request)

Команда для установки консольной утилиты curl:  
**$ sudo apt install curl**

Установка программы с графическим интерфейсом postman:  
**$ sudo snap install postman**

***Шпаргалка по curl***

* **Отправка GET-запроса:  
  curl <URL>**
* **Отправка GET-запроса с параметрами:  
  curl "<URL>?param1=value1&param2=value2"**
* **Отправка POST-запроса с данными в теле запроса:  
  curl -d "data" <URL>**
* **Отправка POST-запроса с данными в теле запроса в формате JSON:  
  curl -H "Content-Type: application/json" -d '{"key": "value"}' <URL>**

# ПОТОКИ И ПРОЦЕССЫ

В Python, как и в других языках программирования, важно понимать разницу между процессами и потоками, так как они используются для выполнения задач параллельно или конкурентно. Вот ключевые моменты, которые тебе нужно знать:

**1. Процесс**

* **Процесс** — это экземпляр программы, который выполняется независимо от других процессов.
* Каждый процесс имеет свое собственное пространство памяти, что делает их изолированными друг от друга.
* Процессы создаются с помощью модуля multiprocessing.
* Пример:

**from multiprocessing import Process**

**def worker():**

**print("Процесс работает")**

**if \_\_name\_\_ == "\_\_main\_\_":**

**p = Process(target=worker)**

**p.start()**

**p.join()**

**2. Поток (Thread)**

* **Поток** — это легковесный "подпроцесс", который выполняется внутри процесса.
* Потоки разделяют память и ресурсы процесса, в котором они созданы.
* Потоки создаются с помощью модуля threading.
* Пример:

**import threading**

**def worker():**

**print("Поток работает")**

**t = threading.Thread(target=worker)**

**t.start()**

**t.join()**

**3. Основные различия**

* **Память**: Процессы изолированы, потоки разделяют память.
* **Производительность**: Потоки легче и быстрее создаются, но из-за общего доступа к памяти могут возникать проблемы с синхронизацией (например, race conditions).
* **GIL (Global Interpreter Lock)**: В Python существует GIL, который ограничивает выполнение нескольких потоков на уровне интерпретатора. Это означает, что даже если у тебя несколько потоков, только один может выполнять Python-код одновременно. Это не касается процессов, так как каждый процесс имеет свой собственный интерпретатор.

**4. Когда использовать**

* **Процессы**: Когда задачи требуют полной изоляции и/или интенсивных вычислений (например, обработка больших данных).
* **Потоки**: Когда задачи связаны с I/O операциями (например, работа с сетью или файлами), где GIL не так сильно влияет на производительность.

**5. Синхронизация**

* При работе с потоками важно использовать механизмы синхронизации, такие как Lock, Semaphore, Event из модуля threading, чтобы избежать проблем с одновременным доступом к данным.
* Для процессов синхронизация может быть выполнена с помощью Queue, Pipe или Manager из модуля multiprocessing.

**6. Асинхронное программирование**

* В Python также существует асинхронное программирование с использованием asyncio, которое позволяет выполнять задачи конкурентно без создания потоков или процессов. Это полезно для I/O-bound задач, таких как сетевые запросы.

**Заключение**

* Процессы и потоки — это мощные инструменты для параллельного и конкурентного выполнения задач. Выбор между ними зависит от конкретной задачи и требований к производительности и изоляции.

**Архитектура UNIX** - <http://heap.altlinux.org/tmp/unix_base/ch01s03.html#small-pres-3-02>

# Запуск и контроль программ в linux

В Linux запуск и контроль программ (процессов) осуществляется через командную строку с помощью различных команд и инструментов. Вот основные моменты, которые тебе нужно знать:

**1. Запуск программ**

* **Обычный запуск**: Просто введи имя программы в терминале, и она запустится.

python script.py

* **Запуск в фоновом режиме**: Добавь & в конце команды, чтобы программа работала в фоне.

python script.py &

Это позволит тебе продолжать использовать терминал, пока программа выполняется.

* **Запуск с приоритетом**: Используй команду nice, чтобы задать приоритет процессу (от -20 до 19, где -20 — наивысший приоритет).

nice -n 10 python script.py

**2. Контроль процессов**

* **Просмотр запущенных процессов**: Используй команду ps для просмотра текущих процессов.

ps aux

* + a — процессы всех пользователей,
  + u — детальная информация,
  + x — процессы без терминала.
* **Поиск процесса по имени**: Используй pgrep для поиска PID (идентификатора процесса) по имени.

pgrep python

* **Завершение процесса**: Используй команду kill для завершения процесса по его PID.

kill <PID>

* + kill -9 <PID> — принудительное завершение процесса.
* **Завершение процесса по имени**: Используй pkill для завершения процесса по имени.

pkill python

**3. Управление задачами**

* **Приостановка процесса**: Нажми Ctrl + Z, чтобы приостановить выполнение программы.
* **Возобновление процесса**: Используй команду fg (foreground) или bg (background), чтобы вернуть процесс в работу.
  + fg — вернуть процесс на передний план,
  + bg — продолжить выполнение в фоне.
* **Просмотр задач**: Используй команду jobs, чтобы увидеть список приостановленных и фоновых задач.

jobs

**4. Мониторинг процессов**

* **Команда top**: Показывает список процессов в реальном времени с информацией о потреблении ресурсов (CPU, память и т.д.).

top

* **Команда htop**: Более продвинутая версия top с удобным интерфейсом (может потребовать установки).

htop

**5. Автозапуск программ**

* **Демоны (службы)**: В Linux программы могут запускаться как демоны (фоновые службы). Управление ими осуществляется через systemd.
  + Запуск службы:

sudo systemctl start <service\_name>

* + Включение автозапуска:

sudo systemctl enable <service\_name>

* + Проверка статуса:

sudo systemctl status <service\_name>

* **Добавление в автозагрузку**: Можно добавить команду в файл ~/.bashrc или ~/.profile, чтобы программа запускалась при входе в систему.

**6. Логи и диагностика**

* **Просмотр логов**: Логи процессов и системы хранятся в /var/log/. Для просмотра используй команду tail или cat.

tail -f /var/log/syslog

* **Проверка ошибок**: Используй команду dmesg для просмотра системных сообщений, связанных с оборудованием и драйверами.

dmesg | grep error

**7. Управление ресурсами**

* **Ограничение ресурсов**: Используй команду ulimit, чтобы ограничить ресурсы для процесса (например, память или количество открытых файлов).

ulimit -n 1024 # Ограничение количества открытых файлов

* **Мониторинг ресурсов**: Используй команду free для просмотра использования памяти и df для просмотра использования диска.

free -h

df -h

**Заключение**

* В Linux управление процессами и программами осуществляется через командную строку с помощью мощных инструментов, таких как ps, kill, top, systemctl и других.
* Понимание этих команд позволяет эффективно запускать, контролировать и диагностировать программы в Linux.
* Для более глубокого изучения используй man (например, man ps), чтобы получить подробную документацию по каждой команде.

Подробнее - <http://xgu.ru/wiki/Стандартные_потоки_ввода/вывода>

# Работа с объектом Popen

**Что такое Popen?**

Popen — это класс из модуля subprocess в Python, который позволяет создавать новые процессы, взаимодействовать с ними и управлять ими. С помощью Popen ты можешь запускать внешние команды, программы или скрипты прямо из своего Python-кода.

**Зачем это нужно?**

Иногда тебе нужно выполнить команду в командной строке (например, запустить другую программу или скрипт) и получить результат её выполнения. Popen позволяет сделать это программно, не выходя из Python.

**Основные моменты:**

1. **Импорт модуля**:  
   Для работы с Popen нужно импортировать модуль subprocess:

import subprocess

1. **Создание процесса**:  
   Чтобы запустить команду, нужно создать объект Popen. Например, запустим команду ls (список файлов в текущей директории):

process = subprocess.Popen(['ls', '-l'])

Здесь ['ls', '-l'] — это список, где первый элемент — это команда, а остальные — её аргументы.

1. **Ожидание завершения процесса**:  
   Если нужно дождаться завершения процесса, можно использовать метод wait():

process.wait()

1. **Получение вывода команды**:  
   Чтобы получить результат выполнения команды (например, вывод в консоль), можно использовать аргумент stdout=subprocess.PIPE:

process = subprocess.Popen(['ls', '-l'], stdout=subprocess.PIPE)

output, error = process.communicate()

print(output.decode('utf-8'))

Здесь communicate() возвращает кортеж из двух элементов: stdout и stderr (стандартный вывод и ошибки). Мы можем декодировать вывод в строку с помощью decode('utf-8').

1. **Передача ввода в процесс**:  
   Если нужно передать данные в процесс (например, ввод с клавиатуры), можно использовать stdin=subprocess.PIPE:

process = subprocess.Popen(['grep', 'python'], stdin=subprocess.PIPE, stdout=subprocess.PIPE)

output, error = process.communicate(input=b'some text with python\n')

print(output.decode('utf-8'))

Здесь мы передаем строку 'some text with python\n' в процесс grep, который ищет слово python.

1. **Обработка ошибок**:  
   Если команда завершилась с ошибкой, можно проверить код возврата с помощью process.returncode. Обычно код 0 означает успешное выполнение, а другие коды — ошибки.

**Пример:**

import subprocess

# Запускаем команду 'ls -l' и получаем её вывод

process = subprocess.Popen(['ls', '-l'], stdout=subprocess.PIPE, stderr=subprocess.PIPE)

output, error = process.communicate()

if process.returncode == 0:

print("Команда выполнена успешно:")

print(output.decode('utf-8'))

else:

print("Произошла ошибка:")

print(error.decode('utf-8'))

**Важные моменты:**

* **Безопасность**: Будь осторожен при использовании Popen с пользовательским вводом, чтобы избежать уязвимостей (например, инъекций команд).
* **Блокировка**: Метод wait() и communicate() блокируют выполнение программы до завершения процесса. Если нужно выполнять команду асинхронно, можно использовать Popen без вызова этих методов.
* **Кодировка**: Убедись, что ты правильно обрабатываешь кодировку вывода (например, utf-8).

# Метод subprocess.run

Метод run в модуле subprocess — это удобный высокоуровневый способ запуска внешних команд и управления ими. Он был добавлен в Python 3.5 как замена для более старых функций, таких как call, check\_call и check\_output. Давай разберём, как он работает, какие аргументы принимает и что умеет делать.

**Основное назначение**

Метод subprocess.run используется для запуска внешней команды, ожидания её завершения и получения результата. Он возвращает объект CompletedProcess, который содержит информацию о завершённом процессе (например, код возврата, стандартный вывод и стандартный вывод ошибок).

**Синтаксис**

subprocess.run(args, \*, stdin=None, input=None, stdout=None, stderr=None, capture\_output=False, shell=False, cwd=None, timeout=None, check=False, encoding=None, errors=None, text=None, env=None, universal\_newlines=None)

**Основные аргументы**

1. **args**:
   * Команда, которую нужно выполнить. Это может быть строка или список строк.
   * Пример:

subprocess.run(["ls", "-l"]) # Список аргументов

subprocess.run("ls -l", shell=True) # Строка (только с shell=True)

1. **stdin**:
   * Поток ввода для команды. Может быть файловым объектом, subprocess.PIPE или None.
   * Пример:

subprocess.run(["cat"], input=b"hello", stdout=subprocess.PIPE)

1. **stdout**:
   * Поток вывода команды. Может быть subprocess.PIPE, файловым объектом или None.
   * Пример:

result = subprocess.run(["echo", "hello"], stdout=subprocess.PIPE)

print(result.stdout) # b'hello\n'

1. **stderr**:
   * Поток ошибок команды. Может быть subprocess.PIPE, файловым объектом или None.
   * Пример:

result = subprocess.run(["ls", "nonexistent"], stderr=subprocess.PIPE)

print(result.stderr) # b'ls: nonexistent: No such file or directory\n'

1. **capture\_output**:
   * Если True, то stdout и stderr автоматически перенаправляются в subprocess.PIPE.
   * Пример:

result = subprocess.run(["echo", "hello"], capture\_output=True)

print(result.stdout) # b'hello\n'

1. **shell**:
   * Если True, команда выполняется через оболочку (например, /bin/sh в Unix или cmd.exe в Windows).
   * Пример:

subprocess.run("echo $HOME", shell=True)

1. **cwd**:
   * Текущий рабочий каталог для выполнения команды.
   * Пример:

subprocess.run(["ls"], cwd="/tmp")

1. **timeout**:
   * Максимальное время выполнения команды в секундах. Если команда не завершится за это время, будет вызвано исключение subprocess.TimeoutExpired.
   * Пример:

try:

subprocess.run(["sleep", "10"], timeout=5)

except subprocess.TimeoutExpired:

print("Команда не завершилась вовремя!")

1. **check**:
   * Если True, то при завершении команды с ненулевым кодом возврата будет вызвано исключение subprocess.CalledProcessError.
   * Пример:

try:

subprocess.run(["false"], check=True)

except subprocess.CalledProcessError:

print("Команда завершилась с ошибкой!")

1. **encoding и errors**:
   * Используются для декодирования вывода команды в строку.
   * Пример:

result = subprocess.run(["echo", "hello"], capture\_output=True, encoding="utf-8")

print(result.stdout) # 'hello\n'

1. **text**:
   * Если True, то stdin, stdout и stderr обрабатываются как текстовые строки (аналогично universal\_newlines=True).
   * Пример:

result = subprocess.run(["echo", "hello"], capture\_output=True, text=True)

print(result.stdout) # 'hello\n'

1. **env**:
   * Словарь с переменными окружения для команды.
   * Пример:

subprocess.run(["echo", "$MY\_VAR"], shell=True, env={"MY\_VAR": "hello"})

**Возвращаемое значение**

Метод run возвращает объект CompletedProcess, который имеет следующие атрибуты:

* **args**: Аргументы, переданные в команду.
* **returncode**: Код возврата команды. 0 обычно означает успешное выполнение.
* **stdout**: Стандартный вывод команды (если stdout=subprocess.PIPE).
* **stderr**: Стандартный вывод ошибок (если stderr=subprocess.PIPE).

result = subprocess.run(["echo", "hello"], capture\_output=True, text=True)

print(result.args) # ['echo', 'hello']

print(result.returncode) # 0

print(result.stdout) # 'hello\n'

**Примеры использования**

1. **Простой запуск команды**:

import subprocess

subprocess.run(["ls", "-l"])

1. **Захват вывода**:

result = subprocess.run(["echo", "hello"], capture\_output=True, text=True)

print(result.stdout) # 'hello\n'

1. **Проверка кода возврата**:

try:

subprocess.run(["false"], check=True)

except subprocess.CalledProcessError:

print("Команда завершилась с ошибкой!")

1. **Тайм-аут**:

try:

subprocess.run(["sleep", "10"], timeout=5)

except subprocess.TimeoutExpired:

print("Команда не завершилась вовремя!")

1. **Использование переменных окружения**:

subprocess.run(["echo", "$MY\_VAR"], shell=True, env={"MY\_VAR": "hello"})

**Преимущества run перед другими методами**

* **Удобство**: run объединяет функциональность call, check\_call, check\_output и других методов.
* **Гибкость**: Поддерживает множество параметров для управления вводом, выводом, тайм-аутами и т.д.
* **Читаемость**: Код с использованием run проще читать и поддерживать.

# Обработка ошибок в Flask. Flask Error Handlers

Обработка ошибок в Flask — это важная часть разработки веб-приложений, которая позволяет корректно управлять исключительными ситуациями и предоставлять пользователю понятные сообщения об ошибках. Flask предоставляет несколько способов для обработки ошибок, включая встроенные механизмы и возможность создания собственных обработчиков.

## Встроенные обработчики ошибок

Flask автоматически обрабатывает некоторые стандартные HTTP-ошибки, такие как 404 (Not Found) и 500 (Internal Server Error). Однако вы можете переопределить эти обработчики, чтобы кастомизировать ответы.

## Создание собственных обработчиков ошибок

Вы можете создать собственные обработчики ошибок с помощью декоратора **@app.errorhandler()**. Этот декоратор позволяет указать, какой код ошибки или тип исключения должен обрабатываться.

**Пример обработки ошибки 404**

from flask import Flask, render\_template

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

@app.errorhandler(404)

def page\_not\_found(error):

return render\_template('404.html'), 404

В этом примере, если пользователь запросит несуществующую страницу, Flask вызовет функцию page\_not\_found, которая вернет кастомный шаблон 404.html с кодом состояния 404.

**Пример обработки ошибки 500**

@app.errorhandler(500)

def internal\_server\_error(error):

return render\_template('500.html'), 500

Этот обработчик будет вызван, если на сервере произойдет внутренняя ошибка.

**Обработка исключений**

Вы также можете обрабатывать исключения Python, такие как ValueError, TypeError и другие.

@app.errorhandler(ValueError)

def handle\_value\_error(error):

return "Произошла ошибка значения: " + str(error), 400

**Глобальные обработчики ошибок**

Если вы хотите обрабатывать все ошибки в одном месте, вы можете использовать глобальный обработчик:

@app.errorhandler(Exception)

def handle\_all\_exceptions(error):

return "Произошла непредвиденная ошибка: " + str(error), 500

## Логирование ошибок

Для отладки и мониторинга важно логировать ошибки. Flask позволяет логировать ошибки с помощью модуля logging.

import logging

@app.errorhandler(404)

def page\_not\_found(error):

app.logger.error(f"404 error: {error}")

return render\_template('404.html'), 404

**Пример полного приложения**

from flask import Flask, render\_template

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

@app.route('/')

def index():

return "Добро пожаловать на главную страницу!"

@app.errorhandler(404)

def page\_not\_found(error):

return render\_template('404.html'), 404

@app.errorhandler(500)

def internal\_server\_error(error):

return render\_template('500.html'), 500

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

app.run(debug=True)

Иерархия ошибок - <https://docs-python.ru/tutorial/vstroennye-iskljuchenija-interpretator-python/ierarhija-vstroennyh-iskljuchenij/>

**Основные возможности модуля logging**

1. Уровни логирования : Модуль поддерживает несколько уровней логирования, которые позволяют классифицировать сообщения по степени важности:
   * **DEBUG**: Подробная информация, обычно интересна только при диагностике проблем.
   * **INFO**: Подтверждение того, что всё работает как ожидается.
   * **WARNING**: Указывает на неожиданное поведение или потенциальную проблему.
   * **ERROR**: Сообщает об ошибке, которая помешала выполнению какой-либо функции.
   * **CRITICAL**: Сообщает о серьёзной ошибке, из-за которой программа может быть неспособна продолжить работу.
2. Гибкая настройка : Вы можете настраивать формат вывода, место сохранения логов (консоль, файлы, удалённые серверы), фильтрацию сообщений по уровням и многое другое.
3. Многопоточность и многопроцессорность : Модуль безопасен для использования в многопоточных и многопроцессорных приложениях.
4. Расширяемость : Вы можете создавать собственные обработчики логов (**handlers**) и форматтеры для адаптации под свои нужды.

Про Logging - https://docs-python.ru/standart-library/paket-logging-python/

# Что такое addHandler в Python?

Метод **addHandler** является частью модуля **logging** и используется для добавления обработчиков (handlers) к объекту логгера (**Logger**). Обработчики определяют, куда будут отправляться лог-сообщения: в консоль, файл, сетевой сокет, электронную почту и т.д.

## Зачем нужен addHandler?

Когда вы используете метод **basicConfig**, он автоматически настраивает базовый обработчик (например, вывод в консоль или файл). Однако если вам нужно более сложное поведение, например:

* Логирование одновременно в несколько мест (например, в консоль и файл).
* Разные форматы логов для разных уровней.
* Логирование в удаленные системы (например, через HTTP).

Тогда вам нужно использовать **addHandler** для настройки нескольких обработчиков.

## Как работает addHandler?

1. Создание обработчика :
   * Вы создаете экземпляр класса обработчика (например, **StreamHandler**, **FileHandler**, **RotatingFileHandler** и т.д.).
2. Настройка обработчика :
   * Указываете формат сообщений, уровень логирования и другие параметры.
3. Добавление обработчика к логгеру :
   * Используете метод **addHandler**, чтобы связать обработчик с логгером.
4. Логирование :
   * Сообщения, соответствующие уровню логгера и обработчика, будут отправлены через этот обработчик.

## Пример использования addHandler

## **1.** Логирование в консоль и файл

import logging

# Создаем логгер

logger = logging.getLogger('my\_logger')

logger.setLevel(logging.DEBUG) # Устанавливаем минимальный уровень логирования

# Создаем обработчик для вывода в консоль

console\_handler = logging.StreamHandler()

console\_handler.setLevel(logging.INFO) # Только INFO и выше

console\_formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

console\_handler.setFormatter(console\_formatter)

# Создаем обработчик для записи в файл

file\_handler = logging.FileHandler('app.log')

file\_handler.setLevel(logging.DEBUG) # Все сообщения от DEBUG и выше

file\_formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

file\_handler.setFormatter(file\_formatter)

# Добавляем обработчики к логгеру

logger.addHandler(console\_handler)

logger.addHandler(file\_handler)

# Логирование сообщений

logger.debug('Это сообщение уровня DEBUG') # Будет записано только в файл

logger.info('Это сообщение уровня INFO') # Будет выведено в консоль и записано в файл

logger.warning('Это сообщение уровня WARNING')

2023-10-05 12:34:56 - INFO - Это сообщение уровня INFO

2023-10-05 12:34:56 - WARNING - Это сообщение уровня WARNING

Содержимое файла **app.log**:

2023-10-05 12:34:56 - my\_logger - DEBUG - Это сообщение уровня DEBUG

2023-10-05 12:34:56 - my\_logger - INFO - Это сообщение уровня INFO

2023-10-05 12:34:56 - my\_logger - WARNING - Это сообщение уровня WARNING

## 2. Логирование с разными уровнями для разных обработчиков

Вы можете настроить разные уровни логирования для разных обработчиков. Например:

* В файл записываются все сообщения (DEBUG и выше).
* В консоль выводятся только предупреждения и ошибки (WARNING и выше).

import logging

# Создаем логгер

logger = logging.getLogger('my\_logger')

logger.setLevel(logging.DEBUG)

# Обработчик для файла

file\_handler = logging.FileHandler('app.log')

file\_handler.setLevel(logging.DEBUG)

file\_formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

file\_handler.setFormatter(file\_formatter)

# Обработчик для консоли

console\_handler = logging.StreamHandler()

console\_handler.setLevel(logging.WARNING)

console\_formatter = logging.Formatter('%(levelname)s: %(message)s')

console\_handler.setFormatter(console\_formatter)

# Добавляем обработчики

logger.addHandler(file\_handler)

logger.addHandler(console\_handler)

# Логирование сообщений

logger.debug('Это сообщение уровня DEBUG') # Только в файл

logger.info('Это сообщение уровня INFO') # Только в файл

logger.warning('Это сообщение уровня WARNING') # В файл и консоль

logger.error('Это сообщение уровня ERROR') # В файл и консоль

Вывод в консоль:

WARNING: Это сообщение уровня WARNING

ERROR: Это сообщение уровня ERROR

Содержимое файла **app.log**:

2023-10-05 12:34:56 - DEBUG - Это сообщение уровня DEBUG

2023-10-05 12:34:56 - INFO - Это сообщение уровня INFO

2023-10-05 12:34:56 - WARNING - Это сообщение уровня WARNING

2023-10-05 12:34:56 - ERROR - Это сообщение уровня ERROR

## Типы обработчиков (Handlers)

Python предоставляет множество встроенных обработчиков. Вот некоторые из них:

1. **StreamHandler** :
   * Выводит логи в поток (например, в консоль).
2. **FileHandler** :
   * Записывает логи в файл.
3. **RotatingFileHandler** :
   * Записывает логи в файл с ротацией (ограничение размера файла, создание новых файлов).
4. **TimedRotatingFileHandler** :
   * Записывает логи в файл с ротацией по времени (например, каждый день новый файл).
5. **SocketHandler** :
   * Отправляет логи на удаленный сервер через сокет.
6. **HTTPHandler** :
   * Отправляет логи на удаленный сервер через HTTP.
7. **SMTPHandler** :
   * Отправляет логи по электронной почте.

**Пример с RotatingFileHandler**

import logging

from logging.handlers import RotatingFileHandler

# Создаем логгер

logger = logging.getLogger('rotating\_logger')

logger.setLevel(logging.DEBUG)

# Создаем обработчик с ротацией файлов

handler = RotatingFileHandler('app.log', maxBytes=1024, backupCount=3)

handler.setLevel(logging.DEBUG)

formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

handler.setFormatter(formatter)

# Добавляем обработчик

logger.addHandler(handler)

# Логирование сообщений

for i in range(100):

logger.debug(f'Лог номер {i}')

В этом примере:

* Логи записываются в файл **app.log**.
* Когда размер файла достигает 1024 байт, создается новый файл (до 3 файлов).

## Особенности addHandler

1. Порядок вызова :
   * Вызовите **addHandler** после создания и настройки обработчика, но до начала логирования.
2. Один обработчик — один логгер :
   * Если вы добавите один и тот же обработчик к нескольким логгерам, это может привести к дублированию сообщений.
3. Удаление обработчиков :
   * Вы можете удалить обработчик с помощью метода **removeHandler**.

# Основные плейсхолдеры (placeholders) для fmt

В строке формата (**fmt**) можно использовать специальные плейсхолдеры, которые заменяются соответствующими значениями при создании лог-сообщения. Вот наиболее часто используемые:

| **Плейсхолдер** | **Описание** |
| --- | --- |
| **%(asctime)s** | Время создания лога (формат времени настраивается через параметр**datefmt**). |
| **%(levelname)s** | Уровень логирования (например, DEBUG, INFO, WARNING, ERROR, CRITICAL). |
| **%(message)s** | Текст самого лог-сообщения. |
| **%(name)s** | Имя логгера (например, имя модуля или пользовательское имя). |
| **%(filename)s** | Имя файла, из которого вызван логгер. |
| **%(funcName)s** | Имя функции, из которой вызван логгер. |
| **%(lineno)d** | Номер строки, где произошло логирование. |
| **%(module)s** | Имя модуля, из которого вызван логгер. |
| **%(process)d** | ID процесса. |
| **%(thread)d** | ID потока. |
| **%(threadName)s** | Имя потока. |

# Декларативное конфигурирование логирования через dictConfig

**Что такое декларативное конфигурирование?**

Обычно настройка логирования в Python делается программно — вы пишете код, который создает логгеры, обработчики (handlers) и форматтеры. Но если проект большой, то такой подход становится неудобным:

* Код становится громоздким.
* Изменение настроек требует правки кода.
* Разные разработчики могут по-разному настраивать логирование.

Чтобы упростить жизнь, Python предоставляет возможность декларативного конфигурирования : вы описываете настройки логирования в виде словаря или файла (например, JSON или YAML), а затем загружаете их в программу. Это называется "декларативным", потому что вы просто *описываете* , как должно работать логирование, а Python сам применяет эти настройки.

## Что такое dictConfig?

**dictConfig** — это функция из модуля **logging.config**, которая позволяет настроить систему логирования, используя словарь Python . Этот словарь содержит все необходимые параметры: уровни логирования, обработчики, форматтеры и т.д.

Пример использования:

from logging.config import dictConfig

# Настройка логирования через словарь

dictConfig({

'version': 1, # Версия конфигурации (всегда 1)

'formatters': {

'simple': {

'format': '%(asctime)s - %(name)s - %(levelname)s - %(message)s'

}

},

'handlers': {

'console': {

'class': 'logging.StreamHandler',

'level': 'DEBUG',

'formatter': 'simple',

'stream': 'ext://sys.stdout'

}

},

'root': {

'level': 'DEBUG',

'handlers': ['console']

}

})

# Логирование

import logging

logger = logging.getLogger()

logger.debug('Это сообщение уровня DEBUG')

## Как работает dictConfig?

Словарь, который вы передаете в **dictConfig**, состоит из нескольких ключевых разделов:

### 1. version

* Обязательный ключ. Всегда равен **1**. Это указывает версию формата конфигурации.

### 2. formatters

* Здесь описываются форматы логов. Например:

'formatters': {

'simple': {

'format': '%(asctime)s - %(name)s - %(levelname)s - %(message)s'

}

}

Это означает, что логи будут выводиться в формате:

2023-10-05 12:34:56 - my\_logger - DEBUG - Это сообщение уровня DEBUG

### 3. handlers

* Здесь описываются обработчики (handlers), которые определяют, куда отправляются логи (в консоль, файл и т.д.). Например:

'handlers': {

'console': {

'class': 'logging.StreamHandler', # Вывод в консоль

'level': 'DEBUG', # Минимальный уровень логов

'formatter': 'simple', # Используемый форматтер

'stream': 'ext://sys.stdout' # Поток вывода (консоль)

},

'file': {

'class': 'logging.FileHandler', # Запись в файл

'level': 'INFO', # Минимальный уровень логов

'formatter': 'simple', # Используемый форматтер

'filename': 'app.log' # Имя файла

}

}

### 4. loggers

* Здесь описываются отдельные логгеры для разных частей программы. Например:

'loggers': {

'my\_module': {

'handlers': ['file'], # Используем обработчик 'file'

'level': 'INFO', # Минимальный уровень логов

'propagate': False # Не передавать логи родительскому логгеру

}

}

### 5. root

* Корневой логгер (основной логгер). Все логи, которые не попадают в другие логгеры, обрабатываются здесь. Например:

'root': {

'level': 'DEBUG', # Минимальный уровень логов

'handlers': ['console'] # Используем обработчик 'console'

}

# Кастомный Handler в Python Logging

Кастомные обработчики (custom handlers) позволяют создавать собственные способы обработки логов. Например, вы можете отправлять логи в базу данных, на удаленный сервер, в очередь сообщений или даже в графический интерфейс.

Для создания кастомного обработчика нужно унаследовать класс **logging.Handler** и реализовать его методы. Давайте разберем процесс пошагово.

## 1. Как создать кастомный Handler?

Чтобы создать кастомный обработчик, нужно:

1. Унаследовать класс **logging.Handler**.
2. Переопределить метод **emit(record)**, который определяет, как именно будут обрабатываться логи.
3. (Опционально) Переопределить другие методы, если требуется дополнительная функциональность.

**Пример: простой кастомный Handler для вывода логов в GUI**

Предположим, у нас есть графический интерфейс (например, с использованием **tkinter**), и мы хотим выводить логи в текстовое поле.

import logging

import tkinter as tk

from tkinter import scrolledtext

class TkinterHandler(logging.Handler):

def \_\_init\_\_(self, text\_widget):

"""

Инициализация обработчика.

:param text\_widget: Виджет, куда будут выводиться логи (например, ScrolledText).

"""

super().\_\_init\_\_()

self.text\_widget = text\_widget

def emit(self, record):

"""

Метод emit вызывается при каждом логировании.

Здесь мы форматируем сообщение и добавляем его в текстовое поле.

"""

try:

# Форматируем сообщение

msg = self.format(record)

# Добавляем сообщение в текстовое поле

self.text\_widget.configure(state='normal') # Разблокируем виджет для редактирования

self.text\_widget.insert(tk.END, msg + '\n') # Вставляем сообщение

self.text\_widget.configure(state='disabled') # Блокируем виджет

self.text\_widget.yview(tk.END) # Прокручиваем до конца

except Exception:

# Если что-то пошло не так, вызываем handleError

self.handleError(record)

# Создаем окно Tkinter

root = tk.Tk()

root.title("Логирование в GUI")

# Создаем текстовое поле для логов

text\_area = scrolledtext.ScrolledText(root, state='disabled', height=10)

text\_area.pack(padx=10, pady=10)

# Создаем логгер и добавляем кастомный обработчик

logger = logging.getLogger('gui\_logger')

logger.setLevel(logging.DEBUG)

handler = TkinterHandler(text\_area)

handler.setFormatter(logging.Formatter('%(asctime)s - %(levelname)s - %(message)s'))

logger.addHandler(handler)

# Логирование

logger.debug('Это сообщение уровня DEBUG')

logger.info('Это сообщение уровня INFO')

# Запускаем цикл событий Tkinter

root.mainloop()

**Что происходит в этом примере?**

1. Создание класса **TkinterHandler**:
   * Наследуется от **logging.Handler**.
   * Переопределяется метод **emit**, который отвечает за обработку каждого лог-сообщения.
   * Сообщение форматируется и добавляется в текстовое поле **text\_widget**.
2. Использование обработчика:
   * Создается экземпляр **TkinterHandler**, который принимает текстовое поле как аргумент.
   * Обработчик добавляется к логгеру с помощью метода **addHandler**.
3. Вывод логов:
   * Все логи выводятся в текстовое поле графического интерфейса.

## 2. Основные методы класса Handler

Когда вы создаете кастомный обработчик, вы можете переопределить следующие методы:

**a) emit(record) (обязательный)**

* Этот метод вызывается при каждом логировании.
* Параметр **record** — это объект **LogRecord**, содержащий информацию о логе (время, уровень, сообщение и т.д.).
* Вы должны реализовать этот метод, чтобы определить, как именно будут обрабатываться логи.

**b) handle(record)**

* Этот метод вызывает **emit(record)**, но также проверяет фильтры.
* Обычно его не нужно переопределять, если вы не хотите изменить стандартное поведение.

**c) flush()**

* Этот метод используется для очистки буфера (например, если логи записываются в файл).
* Если ваш обработчик не использует буферизацию, можно оставить его пустым.

**d) close()**

* Этот метод вызывается при закрытии обработчика (например, при завершении программы).
* Здесь можно освободить ресурсы (закрыть файл, соединение с сервером и т.д.).

## 3. Пример: Кастомный Handler для отправки логов на сервер

Предположим, вы хотите отправлять логи на удаленный сервер через HTTP.

import logging

import requests

class HTTPHandler(logging.Handler):

def \_\_init\_\_(self, url):

"""

Инициализация обработчика.

:param url: URL сервера, куда будут отправляться логи.

"""

super().\_\_init\_\_()

self.url = url

def emit(self, record):

"""

Метод emit отправляет логи на сервер.

"""

try:

# Форматируем сообщение

msg = self.format(record)

# Отправляем POST-запрос на сервер

response = requests.post(self.url, json={'message': msg})

if response.status\_code != 200:

print(f"Ошибка отправки лога: {response.status\_code}")

except Exception:

# Если что-то пошло не так, вызываем handleError

self.handleError(record)

# Создаем логгер и добавляем кастомный обработчик

logger = logging.getLogger('http\_logger')

logger.setLevel(logging.DEBUG)

handler = HTTPHandler(url='https://example.com/log')

handler.setFormatter(logging.Formatter('%(asctime)s - %(levelname)s - %(message)s'))

logger.addHandler(handler)

# Логирование

logger.debug('Это сообщение уровня DEBUG')

logger.info('Это сообщение уровня INFO')

**Что происходит в этом примере?**

1. Создание класса **HTTPHandler**:
   * Наследуется от **logging.Handler**.
   * Переопределяется метод **emit**, который отправляет логи на сервер через HTTP POST-запрос.
2. Использование обработчика:
   * Создается экземпляр **HTTPHandler**, который принимает URL сервера как аргумент.
   * Обработчик добавляется к логгеру с помощью метода **addHandler**.
3. Отправка логов:
   * Все логи отправляются на указанный URL.

## 4. Настройка фильтров и уровней

Вы можете дополнительно настроить кастомный обработчик:

* Установить минимальный уровень логирования с помощью метода **setLevel(level)**.
* Добавить фильтры с помощью метода **addFilter(filter)**.

## 5. Заключение

Кастомные обработчики позволяют гибко настраивать систему логирования под ваши нужды. Основные шаги для создания кастомного обработчика:

1. Унаследуйте класс **logging.Handler**.
2. Переопределите метод **emit**, чтобы определить, как обрабатываются логи.
3. (Опционально) Реализуйте дополнительные методы, такие как **flush** и **close**