Python: ошибки, логи, .env | Я.Шпора

Хранение секретов

Хранить токены или другие критические данные (пароли, адреса серверов, ключи шифрования) прямо в коде приложения — плохая идея. Код может попасть в чужие руки, а вместе с кодом — и конфиденциальные данные.

Передавать такие данные в программу лучше из переменных окружения.

Переменную окружения можно создать прямо из bash-терминала:

```
# Создаём переменную TOKEN:
export TOKEN=123
```

> Созданная таким образом переменная существует только на протяжении одной сессии терминала — до тех пор, пока пока терминал открыт.

Переменные окружения можно хранить в обычном текстовом файле. При таком подходе удобно работать с библиотекой dotenv.

Установка библиотеки:

```
pip install python-dotenv
```

Для хранения переменных окружения принято создавать текстовый файл с именем *.env*. Пример содержимого такого файла:

```
# Файл .env
TOKEN=1896714192:AAFmXXXXPEn6KqewJ13kKKtlnBYqVC_XXXX

# В этом файле может быть несколько переменных,
# каждая - на отдельной строке:
```

```
# LOGIN=test
# PASSWORD=some_password
```

При разработке проекта файл *.env* должен быть добавлен в *.gitignore*, иначе файл с паролями и токенами может оказаться в общем доступе.

С помощью библиотеки dotenv можно получить доступ к переменным, сохранённым в файле *.env*:

```
import os

from dotenv import load_dotenv

load_dotenv()
# Теперь все переменные, описанные в файле .env,
# доступны в пространстве переменных окружения:

token = os.getenv('TOKEN')
print(token)  # 1896714192:AAFmXXXXPEn6KqewJ13kKKtlnBYqVC_XXXX
```

Файл `.env` должен лежать в той же директории, что и исполняемый файл программы.

Обработка исключений

Обработка ошибок — обязательная часть любого проекта. В Python обработка ошибок выполняется с помощью конструкции `try...except`:

```
try:
    # Выполняем этот блок кода
except <определённая разновидность исключений>:
    # Выполняем этот блок, если возникло определённое исключение;
    # таких блоков может быть много.
except <другая разновидность исключений>:
    # Выполняем этот блок, если возникло определённое исключение;
    # таких блоков может быть много.
except Exception:
    # Выполняем этот блок, если возникло исключение, отличное от
else:
    # Выполняем этот блок, если не возникло исключений (опциональ
```

```
finally:
    # Этот блок выполнится всегда (опциональный блок).
```

В блоке `try` пишется основной код, который может выполниться, а может вызвать исключение. За ним следует один или несколько блоков `except`, в которых описан код, который сработает при том или ином типе исключения. Лучше всегда указывать тип исключения явным образом — это хорошая практика.

Логирование

Логирование — это ведение «бортового журнала», автоматическая запись событий в специальный файл или вывод таких записей в терминал.

В Python есть встроенная библиотека для логирования — **logging**.

Настройка логов

Настроить логирование можно глобально, а можно определить настройки точечно, например — для каждого пакета в отдельности.

Глобальная настройка

Глобально определить настройки логирования можно через метод `basicConfig()`. У этого метода есть три основных параметра:

- `level` — уровень логирования;
- `filename` — файл, в котором будут сохраняться логи;
- `format` — вид, в котором будет сохранён результат.

Уровни логирования

- **DEBUG** — уровень отладки. На этом уровне выводится служебная информация о том, что происходит в коде; это информация для разработчика.
- **INFO** — информация о текущих событиях. Этот уровень применяют, если нужно убедиться, что всё идёт по плану: «Письмо отправлено», «Запись в базе создана».
- **WARNING** — «тревожный звоночек»: проблемы нет, но есть что-то, что может привести к проблеме; что-то, на что следует обратить внимание.

- **ERROR** — это ошибка: что-то работает не так, как нужно. Требуется вмешательство и исправление ошибки.
- **CRITICAL** — случилось что-то совсем критичное: надо всё бросать и бежать к компьютеру; всё сломалось. Не очень часто используется на практике, обычно бывает достаточно ERROR.

```
import logging


# Здесь задана глобальная конфигурация для логирования:
logging.basicConfig(level=logging.INFO)

logging.debug('123')  # Когда нужна отладочная информация.
logging.info('Сообщение отправлено')  # Когда нужна дополнительная
logging.warning('Большая нагрузка!')  # Когда что-то идёт не так,
logging.error('Бот не смог отправить сообщение')  # Когда что-то с
logging.critical('Всё упало! Зовите админа!1!111')  # Когда всё со
```

В официальной документации рекомендуется создавать отдельные **логгеры** для каждого модуля приложения.

Обычно пишут отдельный логгер для каждого пакета. Имя логгерам традиционно дают по имени `__name__` пакета, для которого создан логгер.

```
import logging

# В переменной __name__ хранится имя пакета;
# это же имя будет присвоено логгеру.
# Это имя будет передаваться в логи, в аргумент %(name)
logger = logging.getLogger(__name__)

...
```

Хэндлер — это обработчик логов, переданных в логгер. В листинге применён обработчик **RotatingFileHandler**, он управляет ротацией логов.

```
...
from logging.handlers import RotatingFileHandler
```

```
...

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)
handler = RotatingFileHandler('my_logger.log', maxBytes=50000000,
logger.addHandler(handler)
```

Обработчик **RotatingFileHandler** следит за объёмом и количеством лог-файлов. Файлы с логами со временем растут, занимают всё больше места и, в результате, могут забить всё дисковое пространство. Чтобы контролировать их объём, можно выставить ограничения количества этих файлов и на их размер.

Кроме RotatingFileHandler есть и другие — **FileHandler**, который отправляет записи в файл, и **StreamHandler**, который отправляет записи в стандартные потоки, такие как sys.stdout или sys.stderr.

Логирование исключений

Исключения тоже можно логировать:

```
# example_for_log.py

try:
    42 / 0
except Exception as error:
    logging.error(error, exc_info=True)

# Будет записано в лог:
# ERROR:root:division by zero
# Traceback (most recent call last):
#   File "/Users/stasbasov/dev/kittybot/example_for_log.py", line
#     42 / 0
# ZeroDivisionError: division by zero
```