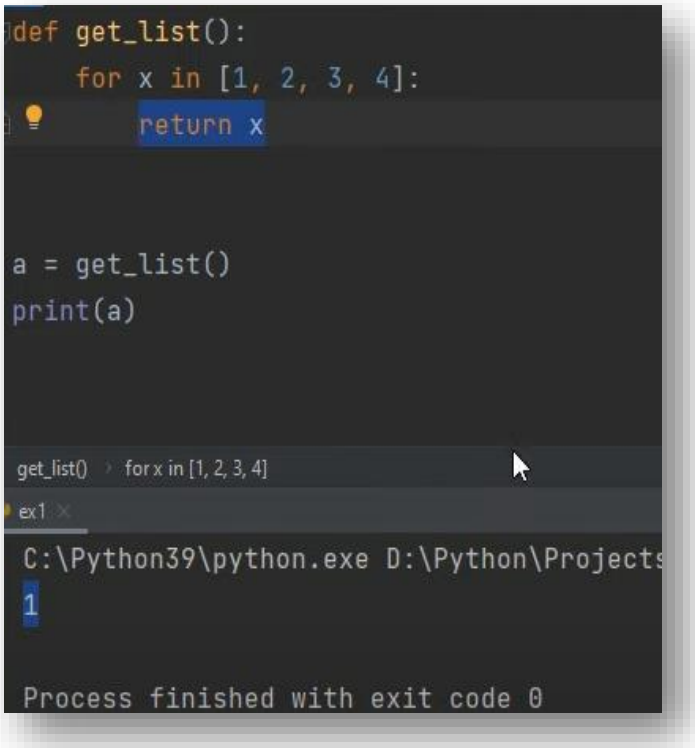
Yield

yield - ключевое слово, используемое для возврата из функции с сохранением состояния ее локальных переменных, и при повторном вызове функции выполнение продолжается с **yield**. Функция содержащая **yield** – генератор.

Пример с оператором **return** и **yield**



Преимущества **yield**:

🔷Автоматическое сохранение и управление состояниями локальных переменных, не нужно заботиться о выделении и освобождении памяти.

🔷Быстрота выполнения: при повторном вызове генератор возобновляет работу, а не начинает с начала.

Про генераторы и итераторы:

Итерация - процесс, где последовательно повторяется набор инструкций до тех пор, пока не будет выполнено условие (выполнение цикла —итерация. Выполнился цикл 5 раз, произошло 5 итераций)

Итератор - это объект, позволяющий «обходить» элементы последовательностей.

Генератор - функция, которая при каждом своём вызове возвращает объект (в функции-генераторе вызывается next())

next() - позволяет извлекать следующий объект из итератора (автоматически вызывается в while и for)

итератор — это механизм поэлементного обхода данных, а генератор позволяет отложено создавать результат при итерации. Генератор – это итератор, но не наоборот.

Пример **yield** с числами Фибоначчи

```
# Создаем бесконечный генератор чисел Фибоначчи
def fib():
    a, b = 0, 1
    while 1:
        yield a # исполнение продолжится отсюда
        a, b = b, a + b

# вызываем генераторную функцию
x = fib()

# итерируем генератор функцией next()
for i in range(6):
    print(next(x))

# 0
# 1
# 1
# 2
# 3
# 5
```

Depends

В FastAPI имеется внедрение зависимостей. Это означает, что в коде есть способ объявить, то что требуется для работы и использования зависимости.

Пример:

```python
from fastapi import FastAPI, Depends

app = FastAPI()

# yield


async def get_async_session():
    print("Получение сессии")
    session = "session"
    yield session
    print("Уничтожение сессии")


@app.get("/items")
async def get_items(session=Depends(get_async_session)):
    print(session)
    return [{"id": 1}]
```

1. Получаем session
2. Yield session отдается в endpoint
3. Мы с ней как-то работаем
4. Отдаем ответ пользователю
5. Функция замораживается
6. Уничтожение сессии

```
INFO:     Waiting for application startup.
INFO:     Application startup complete.
Получение сессии
session
INFO:     127.0.0.1:53186 - "GET /items HTTP/1.1" 200 OK
Уничтожение сессии
```

Depends(get_db)

Предпочтительным способом передачи сессии базы данных в функции обработки запроса представляет внедрение сессии баз данных в качестве зависимости. Поэтому далее создаем функцию get_db(), через которую объект сессии базы данных будет передаваться в функцию обработки

```python
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

здесь сначала создаем объект сессии базы данных. Затем в конструкции try..finally с помощью оператора yield возвращаем созданный объект. Таким образом, данный объект будет внедрен в функцию обработки запроса. Выражение yield будет выполняться при получении каждого нового запроса.

Помещение выражение yield в блок try позволяет получить и обработать любую ошибку, возникшую в процессе взаимодействия с базой данных.

После завершения операций с базой данных выполняется блок finally, в котором закрывается подключение к базе данных с помощью метода close()

```python
@router.post('/create/{token_limit}')
async def create(token_limit: int, db: Session = Depends(get_db)):
    return service.create_token(token_limit, db)


@router.get('/use/{idd}')
async def use(idd: int = None, db: Session = Depends(get_db)):
    return service.use(idd, db)


@router.get('/list')
async def get_list(db: Session = Depends(get_db)):
    return service.get_all(db)
```

С помощью класса Depends() в функцию передается результат функции get_db, то есть сессия базы данных, который передается параметру db. И через этот параметр мы сможем взаимодействовать с базой данных. Подобным образом сессия базы данных внедряется во все остальные функции-обработчики запросов.

Pidentic

Pydantic — это библиотека Python, которая упрощает процесс проверки данных. Это универсальный инструмент, который можно использовать в различных сферах, таких как создание API, работа с базами данных и обработка данных в проектах. Библиотека имеет простой и интуитивно понятный синтаксис, позволяющий легко определять и проверять модели данных. Она включает в себя возможность указывать типы, значения по умолчанию и ограничения непосредственно в коде, что делает его понятным и удобным в сопровождении.

Pydantic предоставляет ряд функций проверки и анализа данных

- Проверка данных: Pydantic включает функцию проверки, которая автоматически проверяет типы и значения атрибутов класса, гарантируя их правильность и соответствие любым заданным ограничениям. Также поддерживаются настраиваемые правила проверки с использованием подсказок, что позволяет пользователям определять дополнительные ограничения для своих данных.
- Поддержка комплексных чисел. Поддерживает широкий спектр типов данных, включая списки, словари и вложенные модели, что упрощает определение и проверку сложных структур данных.
- Эффективность и лёгкость: Pydantic спроектирован так, чтобы быть быстрым и эффективным, с небольшой кодовой базой и минимальным количеством зависимостей. Он также хорошо документирован и прост в использовании, что делает его популярным выбором для проверки и анализа данных в Python.
- Открытый исходный код: Pydantic — это библиотека с открытым исходным кодом, лицензированная в соответствии с лицензией MIT, которая активно разрабатывается и поддерживается сообществом участников.

Одним из основных преимуществ использования Pydantic является то, что она позволяет нам определять структуру наших данных декларативным способом, используя аннотации типов Python. Это означает, что мы можем указать типы, значения по умолчанию и ограничения наших данных непосредственно в коде, что упрощает его понимание и поддержку.

Например, мы хотим определить простую модель данных для пользователя с полями username, age, email и password. С Pydantic можно определить эту модель следующим образом:

```python
from pydantic import BaseModel

class User(BaseModel):
    username: str
    age: int
    email: str
    password: str
```

Каждое поле снабжено аннотацией типа, указывающей тип данных, которые оно может содержать. В этом случае имя пользователя, адрес электронной почты и пароль являются строками, а возраст — целым числом.

```python
class User(BaseModel):
    username: str
    age: conint(gt=18)
    email: EmailStr
    password: constr(min_length=8, max_length=16)
    phone: Optional[str] = 'Unknown'
```

В этом примере мы использовали валидатор constr для определения ограничений для полей age и password. Мы указываем, что пароль должен состоять не менее чем из 8 символов. Поле age определено как тип conint с условием, что его значение должно быть больше 18 символов, а поле password определено как тип constr с двумя условиями: значение должно иметь длину не менее 8 символов и не более 16 символов . Мы также использовали тип EmailStr, чтобы указать, что поле email должно содержать действительный адрес электронной почты. Наконец, было добавлено поле phone, которое является необязательным и имеет значение по умолчанию «Неизвестно».

В дополнение к встроенным валидаторам, мы также можем создавать свои собственные валидаторы. Это может быть полезно, когда нам нужно применить более конкретные или сложные правила к нашим данным, которые не охватываются встроенными средствами проверки.

Чтобы создать собственный валидатор, мы можем определить функцию, которая принимает значение и выполняет все необходимые проверки. Если значение допустимо, функция должна вернуть его. Если значение недействительно, функция должна вызвать ValidationError с сообщением, объясняющим проблему.

Пример использования пользовательских валидаторов для модели User, аналогичный предыдущему примеру, и показывающий, как данные передаются в класс Pydantic для проверки и что такое выходные данные Pydantic.

```python
import re
from pydantic import BaseModel, validator

class User(BaseModel):
    username: str
    age: int
    email: str
    password: str

    @validator("age")
    @classmethod
    def validate_age(cls, value):
        if value < 18:
            raise ValueError("User must be adult")
        return value

    @validator("email")
    @classmethod
    def validate_email(cls, value):
        if not bool(re.fullmatch(r'[\w.-]+@[\w-]+\.[\w.]+', value)):
            raise ValueError("Email is invalid")
        return value
```

```python
    @validator("password")
    @classmethod
    def validate_password(cls, value):
        password_length = len(value)
        if password_length < 8 or password_length > 16:
            raise ValueError("The password must be between 8 and 16 characters long")
        return value

# Valid User
valid_user = {
    'username': 'test_name',
    'age': 20,
    'email': 'name@test.gr',
    'password': '123456789'
}

# Invalid User
invalid_user = {
    'username': 'test_name',
    'age': 16,
    'email': 'name_test.gr',
    'password': '1234'
}

try:
    valid_result = User(**valid_user)
    print(valid_result)
    # Valid Output: username='test_name' age=20 email='name@test.gr' password='123456789'
    invalid_result = User(**invalid_user)
except ValueError as e:
    print(e.errors())
    # Invalid Output:
    [
      {
          "loc":"(""age",")",
          "msg":"User must be adult",
          "type":"value_error"
      },
      {
          "loc":"(""email",")",
          "msg":"Email is invalid",
          "type":"value_error"
      },
      {
          "loc":"(""password",")",
          "msg":"The password must be between 8 and 16 characters long",
          "type":"value_error"
      }

    ]
```