

Модули

- [Описание модулей](#)
- [jsonschema](#)
- [Pydantic 2](#)

Описание модулей

Хранение конфигурации

Configparser стандартная библиотека для чтения и записи .ini файлов. [Инструкция 1](#)

Jsonschema модуль для проверки соответствия json существующей схеме. [Документация](#)

Libsettings модуль на основе [jsonschema](#) для чтения конфигурации из json файла и проверки конфигурации на соответствие схеме. [Gitverse проекта](#)

```
import logging
from libsettings import Jsettings

logging.basicConfig(level=logging.ERROR,
                    filename='error.log',
                    format="%(levelname)s %(message)s")
mysettings = Jsettings(settingsfname='mysettings.json',
                       schemafname='myschema.json')
mysettings.load_settings()
```

jsonschema

Используется для валидации json схемы. По умолчанию дополнительно указанные ключи (не существующие в схеме, но присутствующие в документе) не проверяются.

Установка

```
pip install jsonschema
```

Базовое использование

```
from jsonschema import validate

validate(instance=json_to_check, schema=schema)
```

Исключения

`jsonschema.exceptions.ValidationError` - если документ не соответствует структуре

`jsonschema.exceptions.SchemaError` - если сама схема некорректна

Пример схемы:

```
schema = {
    "type": "object",
    "properties": {
        "name": {"type": "string"},
        "age": {"type": "number"},
    },
    "required": ["name"],
}
```

Данная схема определяет json объект с 2 свойствами: name и age. Обязательное свойство name.

Ключевые слова

Для некоторых типов используются дополнительные ключевые слова.

Ключевое слово	Описание
type	Тип. Для корня часто object. string - строка number - число object - объект array - список
\$defs	Вложенный шаблон для случая, когда шаблон элемента встречается в нескольких местах.
\$ref	Подстановка вложенного шаблона.
\$schema	Ссылка на шаблон шаблона. При обновлении версии библиотеки будет использоваться новый шаблон шаблона, что может привести к проблемам. Желательно указывать.

Дополнительные ключевые слова для типов.

Тип array

Ключевое слово	Описание
items	Тип элементов списка. <div>"scores": { "type": "array", "items": {"type": "number"}, }</div>
minItems	Минимальное количество элементов

Тип object

Ключевое слово	Описание
required	Список обязательных ключей. <div>"required": ["name"]</div>

Ключевое слово	Описание
properties	Определяет ключи объекта и их тип. <div>"properties": { "name": {"type": "string"} },</div>
additionalProperties	Если True, то дополнительно указанные ключи приводят к исключению.

Локальные вложенные шаблоны.

Для определения используется переменная \$defs, для использования - \$ref.

```
schema = {
  "type": "object",
  "properties": {
    "address": {"$ref": "#/$defs/address"},
  },
  "$defs": {
    "address": {
      "type": "object",
      "properties": {
        "street": {"type": "string"},
      }
    },
  },
}
```

Pydantic 2

Описание

Библиотека валидации (проверка на соответствие типов) и трансформации (автоматическое приведение к нужным типам и форматам) данных.

Модели наследуются от класса `BaseModel`. Модель описывает набор полей, представляющих структуру данных и условия валидации.

Установка

```
pip install pydantic
```

Типизация:

- Простая: указание типа, например, `name: str`.
- Объект `Field()`: дополнительные параметры, например, значения по умолчанию, ограничения и т.д.

Внутри класса можно комбинировать способы типизации.

```
from pydantic import BaseModel, Field

class User(BaseModel):
    name: str
    email: str = Field(..., alias='email_address')
```

Валидация:

- Минимальная валидация: встроенные типы Python (например, `str`, `int`).
- Валидаторы: например `EmailStr` для проверки email-адресов. Требуется установка дополнительных зависимостей: `pydantic[email]` или `pydantic[all]`.
- `@field_validator` — добавляет логику валидации поля. Вызывается при создании или изменении модели.

```
from pydantic import BaseModel, field_validator

class User(BaseModel):
    age: int
```

```
@field_validator('age')
def check_age(cls, value):
    if value < 18:
        raise ValueError('Возраст должен быть больше 18 лет')
    return value
```

- `@computed_field` — вычисляемое поле на основе данных в модели. Его можно использовать для автоматической генерации значений, а также для валидации.

```
from pydantic import BaseModel, computed_field

class User(BaseModel):
    name: str
    surname: str

    @computed_field
    def full_name(self) -> str:
        return f"{self.name} {self.surname}"
```

- `@model_validator` - валидация всей модели.

Работает со всей моделью (а не с отдельными полями), может изменять данные перед валидацией (`mode='before'`) или после (`mode='after'`), полезен для комплексных проверок, где одно поле зависит от другого, может возвращать новую версию модели (если нужно модифицировать данные).

```
@model_validator(mode='before')
def validate_before(cls, data: dict):
    if 'username' not in data:
        data['username'] = "guest_" + str(data.get('id', 0))
    return data
```

```
@model_validator(mode='after')
def validate_after(self):
    if self.age < 18 and self.is_premium:
        raise ValueError("Minors cannot have premium accounts!")
    return self
```

При проверке `before` передается класс, при `after` - объект. Можно делать два валидатора: `before` для подстановки вычисляемых значений и `after` для финальной проверки

Интеграция с SQLAlchemy:

Для настройки используется параметр ConfigDict с флагом from_attributes=True.

```
from datetime import date
from pydantic import BaseModel, ConfigDict

class User(BaseModel):
    id: int
    name: str = 'John Doe'
    birthday_date: date

    config = ConfigDict(from_attributes=True)
```

Для создания модели Pydantic из объекта ORM используется метод from_orm.

```
user = User.from_orm(orm_instance)
```

Ссылки:

[Основа для текста](#)