2024-10-02

Объективно-ориентированное программирование в Python

Объектно-ориентированная парадигма представляет собой методологию разработки, в которой семантически значимые элементы (сущности) представляются в виде объектов со своим набором свойств и каким-либо образом между собой взаимодействуют.

Каждый объект принадлежит к какому-либо классу.

Класс — модель объекта определенного типа, которая описывает его внутреннюю структуру и то, какие методы и алгоритмы доступны при взаимодействии объекта этого типа с другими объектами (как такого же типа, так и иных типов).

Базовым классом в Питоне является `object`. От него наследуются все классы (в том числе происходит неявно наследование, если вы создаете свой класс и явно не наследуете его ни от чего).

Чтобы посмотреть набор свойств и методов любого класса, можно вызывать метод `dir()`

```
dir(object)

'''
[out]:

['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__']
'''
```

```
class Pet:  # в отличие от имен переменных, функций, методов и т.п.,
имена классов в Питоне подчиняются стилю CamelCase
    def __init__(self, tail:str='black', ears:tuple[str]=('black',
```

```
'black')) -> None:
        self.tail = tail
        self.ears = ears

class Cat(Pet):
    def __init__(self, tail:str, ears:tuple[str], whiskers:str='white')
-> None:
        super().__init__()
        self.whiskers = whiskers
```

Классы могут наследоваться друг от друга — **дочерний класс** получает доступ к структуре и методам **родительского класса**, при этом может добавлять в свою структуру новые элементы и методы.

Объединенные такими связями наследования классы образуют **иерархию классов** в рамках отдельно взятого программного продукта.

Три важнейших принципа ООП — **инкапсуляция**, **полиморфизм** и **наследование**.

> **Об абстракции**
>
> Если нужны абстрактные классы, то они вынесены в отдельный модуль стандартной библиотеки — `abc` (Abstract Base Classes).
> Документация: abc — Abstract Base Classes — Python 3.12.7 documentation

Полиморфизм — способность метода обрабатывать данные разных типов.

```
def divider(a: int | float | str, b: int | float | str) -> float:
    if isinstance(a, str):
        a = float(a)
    if isinstance(b, str):
        b = float(b)
    return a / b
```

Инкапсуляция — возможность изолирования/сокрытия конкретных элементов структуры и/или методов класса от внешних воздействий. По сути, это разделяет методы и свойства класса на публично доступные (иногда их можно считать интерфейсами) и скрытыми, поддерживающими непосредственно модель объекта.

Явные модификаторы защиты (`protected`, `private`, `public`) в Питоне отсутствуют. Для имитации их поведения предназначены специальные правила именования методов класса.

- `_single_leading_underscore`: указание на то, что объект должен использоваться только внутри класса или модуля. `from M import *` не импортирует такие объекты (хотя использование `*` само по себе является плохой практикой), а многие среды разработки скрывают эти свойства и методы при подсказах
- - `single_trailing_underscore_`: конвенциональные имена для атрибутов, чьи названия совпадают со встроенными объектами языка (например `class_` вместо `class` — `Tkinter.Toplevel(master, class_='ClassName')`)
- - `__double_leading_underscore`: name mangling — внутри класса такие атрибуты в памяти всегда записываются как `_ИмяКласса__Метод`, например: `class FooBar`, метод `__tea` в памяти и во всех ссылках будет `_FooBar__tea`, вы не сможете обратиться к нему как `__tea`, только как к `_FooBar__tea`
- - `__double_leading_and_trailing_underscore__`: «магические» методы, объекты и атрибуты. НИКОГДА не изобретать свои магические методы, только использовать уже определенные в языке. К ним относятся, например, `__init__`, `__import__` or `__file__`, `__name__`, `__str__`, `__repr__`, все методы, определяющие поведение операторов сравнения, арифметических операторов, оператора присваивания и т.п.

```
class Foo:
    def __init__(self):
        self.public_var = 'Публичная (public) переменная'
        self._protected_var = 'Защищенная (protected) переменная'
        self.__private_var = 'Приватная/частная (private) переменная'

['_Foo__private_var', '__class__', '__delattr__', '__dict__', '__dir__',
```

```
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', '_protected_var', 'public_var']

a = Foo()
a._protected_var = 'Типа защищенная переменная'
print(a._protected_var) # 'Типа защищенная переменная'
```

Если нам надо изменить «реальный» конструктор, то используем переопределение метода `__new__`:

```
class MyClass:
    def __new__(cls) -> MyClass:
        print('Создаем экземпляр класса в памяти')
        return super(MyClass, cls).__new__(cls)
    def __init__(self) -> None:
        print('Инициализируем')

a = MyClass()


'''
[out]:
Создаем экземпляр класса в памяти
Инициализируем
'''
```

Практика 3

1. Реализовать структуру данных «очередь» (первый пришел, первый ушел) с помощью класса с возможностью просмотра, добавления и удаления элементов.
2. Реализовать структуру данных «стек» (последний пришел, первый ушел) с помощью класса с возможностью просмотра, добавления и удаления элементов.

3. Реализовать иерархию классов, описывающих разные виды объектов одного типа (**например, сервоприводов (синхронный/асинхронный/линейный и т.п.**). Реализовать минимум 3 уровня иерархии. Реализовать возможность задания характеристик (например, для двигателя это угол поворота, скорость вращения, ускорение и т.п.). Реализовать строковое представление классов «магическими» методами `__str__()` и `__repr__()`, быть готовым пояснить различия этих методов. Перегрузить условные операторы (см. магические методы `__eq__()`, `__ne__()`, `__lt__()`, `__gt__()`, `__le__()`, `__ge__()`) для реализации возможности сравнения экземпляров класса.
4. Реализовать упрощенную модель некоего объекта (например, шестизвенного манипулятора с сервоприводами) при помощи иерархии классов. Реализовать функции объекта (например, перемещение манипулятора в пространстве) через перегрузку арифметических операторов (`__add__()` и т.д.).

Лабораторная работа 2

1. Реализовать в консоли таск-трекер через классы. Данные хранить в объекте во время работы программы, выгружать список задач в JSON-файл, при запуске загружать файл (используя модуль `json`). Реализовать возможность ввода произвольной строки с описанием задачи, возможность отметки задания выполненным, возможность ввода произвольных категорий. *Бонус (опциональный): поиск по задачам; вывод всех задач в категории.*

```
- [x] Задание выполнено #pstu
- [ ] Еще задача #work
- [ ] И еще задача #pstu
```

2. Реализовать в консоли трекер бюджета через классы. Данные хранить в объекте во время работы программы, выгружать список задач в JSON-файл, при запуске загружать файл (используя модуль `json`). Реализовать возможность ввода произвольной строки с описанием операции и суммой расхода/дохода, возможность ввода произвольных категорий. *Бонус: установка лимитов на категории.*

Датаклассы, type hinting и валидация моделей

С 2014-го года активно внедряются элементы валидации типов.

PEP 484 – Type Hints | peps.python.org

Также для более сложных типов есть модуль `typing` стандартной библиотеки.

Все это направлено на более структурный подход и более жесткие требования к типизации в Питоне, хотя на уровне интерпретатора все еще типизация была и остается динамической.

Такой подход окупает себя при разработке с проектированием структур данных и их валидацией.

Для этого есть ряд библиотек и модулей (как в стандартной поставке Питона, так и сторонние):

- dataclasses — Data Classes — Python 3.12.7 documentation
- attrs 24.2.0 documentation
- Welcome to Pydantic - Pydantic
- marshmallow: simplified object serialization — marshmallow 3.22.0 documentation

```python
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

```python
from attrs import define, field

@define
class Coordinates:
    x: int
```

```
    y: int

@define
class Coordinates:
    x = field()
    y = field()
```

Welcome to Pydantic - Pydantic

```
from datetime import datetime
from typing import Tuple

from pydantic import BaseModel


class Delivery(BaseModel):
    timestamp: datetime
    dimensions: Tuple[int, int]


m = Delivery(timestamp='2020-01-02T03:04:05Z', dimensions=['10', '20'])
print(repr(m.timestamp))
#> datetime.datetime(2020, 1, 2, 3, 4, 5, tzinfo=TzInfo(UTC))
print(m.dimensions)
#> (10, 20)
```

Пример успешной валидации:

```
from datetime import datetime
from pydantic import BaseModel, PositiveInt


class User(BaseModel):
    id: int
    name: str = 'John Doe'
    signup_ts: datetime | None
    tastes: dict[str, PositiveInt]


external_data = {
```

```
    'id': 123,
    'signup_ts': '2019-06-01 12:22',
    'tastes': {
        'wine': 9,
        b'cheese': 7,
        'cabbage': '1',
    },
}

user = User(**external_data)

print(user.id)
#> 123
print(user.model_dump())
"""
{
    'id': 123,
    'name': 'John Doe',
    'signup_ts': datetime.datetime(2019, 6, 1, 12, 22),
    'tastes': {'wine': 9, 'cheese': 7, 'cabbage': 1},
}
"""
```

Пример ошибки валидации:

```
# continuing the above example...

from datetime import datetime
from pydantic import BaseModel, PositiveInt, ValidationError


class User(BaseModel):
    id: int
    name: str = 'John Doe'
    signup_ts: datetime | None
    tastes: dict[str, PositiveInt]


external_data = {'id': 'not an int', 'tastes': {}}

try:
```

```
    User(**external_data)
except ValidationError as e:
    print(e.errors())
    """
    [
        {
            'type': 'int_parsing',
            'loc': ('id',),
            'msg': 'Input should be a valid integer, unable to parse
string as an integer',
            'input': 'not an int',
            'url': 'https://errors.pydantic.dev/2/v/int_parsing',
        },
        {
            'type': 'missing',
            'loc': ('signup_ts',),
            'msg': 'Field required',
            'input': {'id': 'not an int', 'tastes': {}},
            'url': 'https://errors.pydantic.dev/2/v/missing',
        },
    ]
    """
```