07. ООПП

16 март 2026

07. ОщеООП

16 март 2026

Лирическо отклонение

- Не оставяйте тестов код в решенията си.
- Ще ви обясним причините за това до няколко лекции, но имайте предвид, че шансът да “замърсите” решението си, особено когато има някакъв вид persistence, е немалък.
- Като например паметта на `memnick`-а, а вие да сте оставили в решението си примерния код, който го извиква няколко пъти и добавя няколко фрази към паметта му.
- ~4 лекции и ще имате идея защо това се случва и как да го избегнете **елегантно**, но за момента просто не го правете.

Да си припомним

Миналия път казахме, че в Python:

1. Всичко е обект
2. Обектите са отворени
3. Класовете са отворени

Последните две с някои малки уговорки, които обаче рядко ще ви интересуват.

Класове

Класове се създават с ключовата дума `class`, след което всяка функция, дефинирана в тялото на класа, е метод, а всяка променлива е клас променлива.

Примерен Клас

```
class Vector:

    def __init__(self, x, y):
        self.x = x
        self.y = y

spam = Vector(1.0, 1.0)

print(spam.x)
```

1. „Конструктура“ се казва `__init__`, той не връща стойност
2. Първият аргумент на методите винаги е инстанцията, върху която се извикват, той може да се казва всякак, но винаги се казва `self`, иначе губите огромни количества точки/колегите ви ви мразят/никой не иска да си играе с вас в пясъчника
3. Атрибутите („член-променливите“/„член-данните“) не се нуждаят от декларации (обектите са отворени)
4. Инстанцираме клас, като го „извикаме“ със съответните аргументи, които очаква `__init__` метода му и като резултат получаваме новоконстурирания обект

Примерен Клас 2

```
import math


class Vector:

    def __init__(self, x, y): ...

    def length(self):
        return math.sqrt(self.x**2 + self.y**2)


spam = Vector(1.0, 2.0)

print(spam.length())
```

1. В методите атрибутите могат да се достъпват само през `self`, няма никакви магически имплицитни scope-ове
2. Методите се извикват през инстанцирания обект `обект.име_на_метод()`

Примерен Клас 3

```
class Vector:


    def __init__(self, x, y, z): ...


    def _coords(self):
        return (self.x, self.y, self.z)


    def length(self):
        return sum(_ ** 2 for _ in self._coords()) ** 0.5
```

1. `_coords` е protected метод
2. Отново, методите се извикват върху `self`
3. `_` е валидно име за променлива

Private / Protected

Казахме, че класовете са отворени. Това ще рече, че private и protected концепциите не са това, за което сте свикнали да мислите в езици като C++/Java/C#.

Ограниченията за използване на защитени и частни методи в класовете в Python е отговорност на програмиста, което по никакъв начин не прави живота ви по-труден.

Методи/атрибути започващи с _ са защитени, т.е. би следвало да се ползват само от методи на класа и наследяващи го класове.

Методи/атрибути започващи с __ са частни, т.е. би следвало да се ползват само от методи на класа.

Достатъчно очевидно е, а в някои много редки случаи може да се наложи тези ограничения да не се спазят.

Name mangling

- Това е по-забавно нагледно...
- И все пак, за да го имате в слайдовете - Python променя имената от тип `__name` до `_classname__name`
- Нарича се *name mangling*

Синтаксис VS поведение

- В други езици private / protected / public са синтаксис
- В Python са конвенция
- Много други неща в Python **не** са синтаксис
- ... например статичните методи

Статични методи

```
class Vector:

    def __init__(self, x, y, z): ...


    @staticmethod
    def from_list(numbers):
        if len(numbers) != 3:
            # error!
            …
        return Vector(numbers[0], numbers[1], numbers[2])



v = Vector.from_list([1, 2, 3])
```

Property Methods

```
import math


class Vector:

    def __init__(self, x, y): ...

    @property
    def length(self):
        return math.sqrt(self.x**2 + self.y**2)


spam = Vector(1.0, 2.0)

print(spam.length)
```

Декораторът `@property` може се използва за да накарате някой метод да се преструва на property

Property Methods 2

- Но можем ли да променим стойността на метод, който се преструва на атрибут?
- На помощ идват setter-ите

Property Methods 3

```
class Color:

    def __init__(self, rgba):
        self._rgba = tuple(rgba)

    @property
    def rgba(self):
        return self._rgba

    @rgba.setter
    def rgba(self, value):
        self._rgba = tuple(value)
```

```
>>> red = Color([255, 0, 0])
>>> red.rgba
(255, 0, 0)
>>> red.rgba = [127, 0, 0]
>>> red.rgba
(127, 0, 0)
```

Лирическо отклонение №2, но не толкова отклонено

Mutable vs. Immutable

- mutable са обекти, които променят вътрешното си състояние във времето
- immutable са обекти, които никога не променят вътрешното си състояние

Най-общо повечето обекти в Python са mutable, доколкото езика не ни забранява да ги променяме

Какво в Python знаем, че е immutable?

Mutation method

```
class Vector:

    def __init__(self, x, y, z): …

    def length(self): …

    def normalize(self):
        length = self.length()
        self.x /= length
        self.y /= length
        self.z /= length
```

Non-mutating method

```
class Vector:

    def __init__(self, x, y, z): …

    def length(self): …

    def normalized(self):
        length = self.length()
        return Vector(self.x / length,
                      self.y / length, self.z / length)
```

Normalize vs Normalized?

```
 def normalize(self):
    length = self.length()
    self.x /= length
    self.y /= length
    self.z /= length
```

```
def normalized(self):
    return Vector(self.x / self.length(),
                  self.y / self.length(),
                  self.z / self.length())
```

Сравняване на Обекти

- Можете да проверите дали два обекта са равни по стойност с `==`
- Можете да проверите дали две имена сочат към един и същи обект с `is`
- Можете да предефинирате равенството за обекти от даден клас с метода `__eq__`
- По подразбиране, `__eq__` е имплементирана с `is`

```
class Vector:

    def __init__(self, x, y, z):
        self._coords = (x, y, z)

    def __eq__(self, other):
        return self._coords == other._coords
```

Dunder methods a.k.a. “Magic methods”

Dunder (double under) методите в Python най-често предефинират някакъв аспект от поведението на обектите ни.

Аритметични оператори

- `__add__(self, other) - self + other`
- `__sub__(self, other) - self - other`
- `__mul__(self, other) - self * other`
- `__truediv__(self, other) - self / other`
- `__floordiv__(self, other) - self // other`
- `__mod__(self, other) - self % other`
- `__lshift__(self, other) - self << other`
- `__rshift__(self, other) - self >> other`
- `__and__(self, other) - self & other`
- `__xor__(self, other) - self ^ other`
- `__or__(self, other) - self | other`
- Нека да пробваме някои от математическите операции!

Преобразуване до стандартни типове

- `__int__(self) - int(обект)`
- `__float__(self) - float(обект)`
- `__complex__(self) - complex(обект)`
- `__bool__(self) - bool(обект)`

Обекти, които могат да бъдат извиквани като функции

Можете да дефинирате поведение на обектите си, когато биват извиквани като функции

```
class Stamp:

  def __init__(self, name):
      self.name = name

  def __call__(self, something):
      print(f"{something} was stamped by {self.name}")


>>> stamp = Stamp("The government")

>>> stamp("That thing there")

That thing there was stamped by The government
```

getattr / setattr

```
>>> v1 = Vector(1, 1, 1)
>>> getattr(v1, 'y')
1
>>> setattr(v1, 'z', 5)
>>> getattr(v1, 'z')
5
```

Класови методи

```
class Countable:

    _count = 0

    def __init__(self, data):
        self.data = data
        type(self).increase_count()

    @classmethod
    def increase_count(cls):
        cls._count += 1

    @classmethod
    def decrease_count(cls):
        cls._count -= 1
```

Можете да използвате и `@classmethod`, за да получите класа от който е извикан метода като първи аргумент

Деструктор

```
class Countable:

    _count = 0

    def __init__(self, data):
        self.data = data
        type(self).increase_count()

    @classmethod
    def increase_count(cls):
        cls._count += 1

    @classmethod
    def decrease_count(cls):
        cls._count -= 1

    def __del__(self):
        type(self).decrease_count()
```

Когато обект от даден тип бъде събран от garbage collector-а, се извиква неговия `__del__` метод.

Той не прави това, което прави деструктора в C++, а неща от по-високо ниво, например затваря мрежови връзки или файлове, изпраща съобщения до някъде, че нещо е приключило и прочее.

Vector Class again

```
def addition(a, b):
    return Vector(a.x + b.x, a.y + b.y, a.z + b.z)


class Vector:
    def __init__(self, x, y, z): …

    __add__ = addition


print(Vector(1.0, 2.0, 3.0) + Vector(4.0, 5.0, 6.0))
```

1. Функциите са първокласни обекти
2. Методите са атрибути на класа
3. Класовете са динамични
4. Ето защо `self` е явен

ООП е интересно и сложно и елегантно... ок?

Въпроси?