
Яндекс

Яндекс

Лекция 2. ООП

Немного истории

Немного истории

Развитие языков программирования началось с появления **ассемблера** – символического представления машинного языка, в котором команды языка соответствуют командам процессора.

К началу 70-х годов, в связи с ростом сложности программ, возникла необходимость увеличить производительность труда программистов. Программы стали слишком сложными для проектирования, кодирования и отладки – возникла необходимость переосмыслить подходы к созданию больших программных комплексов.

Над решением этих проблем бились лучшие программисты 70-х годов, такие как Дейкстра, Вирт, Майерс, Йордан и другие. В результате были разработаны строгие правила ведения проектов, которые получили название структурной методологии.

Программы начали представлять собой иерархические структуры блоков. В соответствии с данной методологией любая программа строится без использования оператора `goto` из трех базовых управляющий структур: ветвление, цикл и функции.

Немного истории

Структурное программирование принесло выдающиеся результаты, но даже оно оказалось несостоятельным, когда программа достигала определенной длины. Чтобы писать более сложную программу, необходим был новый подход к программированию. В итоге были разработаны принципы объектно-ориентированного программирования, которое аккумулировало лучшие идеи, воплощенные в структурном программировании.

Для небольших программ возможностей структурного программирования обычно достаточно. Однако крупные проекты, работу над которыми ведут группы людей, намного рациональней выполнять используя парадигму объектно-ориентированного программирования.

Объектно-ориентированное программирование (ООП) — это парадигма (совокупность понятий и идей) программирования, в рамках которой «во главу угла» ставят понятия объектов и классов.

ООП приобрело популярность во второй половине 80-х вместе с такими языками, как C++, Objective C (другое расширение C), Object Pascal и Turbo Pascal.

Основные понятия

Основные понятия

Класс – это описание еще не созданного объекта. Своеобразный шаблон или чертеж, по которому будет строиться экземпляр класса.

Класс содержит данные, которые описывают состав объекта, его атрибуты и методы работы с ним. Служит фабрикой объектов своего типа.

```
class MyClass:
    class_attribute = 'I am a class attribute
'

    def __init__(self, param1, param2):
        self.param1 = param1
        self.custom_name = param2
        self.name = 'world'

    def say_hello(self):
        return 'hello ' + self.name
```

Основные понятия

Объект или **экземпляр класса** – то, что инициализировано по описанию из класса.

Имеет состояние, структуру и поведение, описанные в классе, а также средство для его однозначной идентификации.

```
>>> instance = MyClass('abc', 42)
>>> id(instance)
4491646672
>>> instance.say_hello()
'hello world'
>>> instance.param1
'abc'
>>> instance.class_attribute
'I am a class attribute'
```

Основные понятия

Возникновение в ООП понятия класса связано с желанием иметь множество объектов со сходным поведением.

Класс в ООП – это абстрактный тип данных, задающий внутреннюю структуру и набор операций, который могут быть выполнены над этим типом. Объекты являются значениями данного абстрактного типа.

Класс можно рассматривать как чертеж для изготовления детали, объект – как выточенную по этому чертежу деталь.

Основные принципы

Основные принципы

- Инкапсуляция
- Полиморфизм
- Наследование

Инкапсуляция

Инкапсуляция – это объединение данных и функций в единый компонент – пользовательский тип данных. Обеспечивает модульность и абстракцию данных в языках ООП.

Зачем нужна инкапсуляция? Для сокрытия деталей реализации и данных объекта от внешних ресурсов, открывая наружу лишь интерфейс, дающий возможность оперировать над этими данными. Кроме того, инкапсуляция упрощает процесс отладки, помогая локализовать возможные ошибки в коде программы.

Аналогия из жизни – автомобиль. Мы можем не знать, как работает коробка передач, или что происходит, когда нажимаем педали тормоза или газа, но это не мешает нам водить машину. Мы не знаем реализации, но знакомы с интерфейсом, предоставляемым автомобилем.

Полиморфизм

Полиморфизм позволяет одному и тому же коду работать с данными разных типов и единообразно ссылаться на объекты различных классов (обычно внутри некоторой иерархии).

Бьёрн Страуструп определил полиморфизм как «один интерфейс – много реализаций»

Например, у вас есть базовый класс `Shape`, у которого определен метод `draw`. Все наследники этого класса реализуют этот метод и при его вызове вам не важно, объекту какого подкласса он принадлежит.

Наследование

Наследование — способ определения нового типа, когда новый тип
наследует элементы (свойства, методы и атрибуты) существующего, модифицируя или расширяя их.

Точно также дети наследуют черты лица и характер, от родителей.

- Классы, которые наследуются от другого, называются производными классами или подклассами.
- Классы, из которых получены другие классы, называются базовыми классами или суперклассами.
- Считается, что производный класс порождает, наследует или расширяет базовый класс.

Множественное наследование. MRO

Множественное наследование. MRO

Язык Python поддерживает множественное наследование. Это значит, что класс может иметь более одного родителя.

Каким же образом интерпретатор понимает, в каком классе-предке искать метод, если он не объявлен в классе-потомке?

Множественное наследование. MRO

```
class A:
    def method(self):
        print('A')


class B:
    def method(self):
        print('B')


class C(A, B):
    pass


class D(B, A):
    pass
```

Множественное наследование. MRO

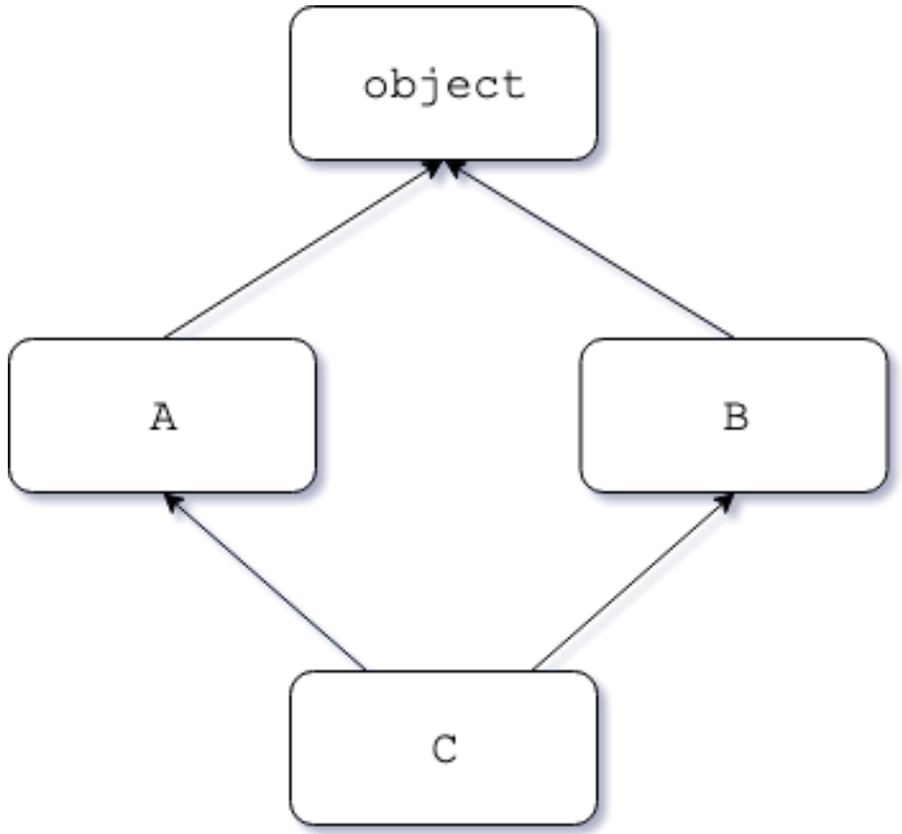
```
>>> c = C()
>>> c.method()
A

>>> d = D()
>>> d.method()
B
```

Множественное наследование. MRO

Видно, что интерпретатор ищет метод в списке родителей слева направо. Но как он поведет себя при более сложной иерархии наследования?

Для решения этой задачи придумали алгоритм `mro(method resolution order)`. Этот алгоритм появился еще в версии 2.3, когда ввели `new style classes`, отличительной чертой которых стало наследование от базового класса `object`. Начиная с третьей версии языка, новые классы полностью заменили старые, а неявное наследование осталось. Потребность в новом алгоритме появилась для решения проблемы ромбовидного наследования. Старый алгоритм поиска в глубину продолжил бы поиск в `object`, а не в классе `B`.



Множественное наследование. MRO

Новый алгоритм строит список предков класса, включая сам класс, отсортированный в порядке "удаленности". Такой список называют **линеаризацией**. Например, линеаризация класса `C` из примера выше будет `[A, B, object]`. Для класса `D` — `[B, A, object]`. Для линеаризации в питоне используется алгоритм `C3 linearization`.

> **Линеаризацией данного класса называется слияние линеаризаций его предков.**

Соглашения об именовании

Дандеры

В сообществе Python двойные символы подчеркивания часто называют «дандерами» (dunders — это сокращение от англ. double underscores). Причина в том, что в исходном коде Python двойные символы подчеркивания встречаются довольно часто, и, чтобы не изнурять свои жевательные мышцы, питонисты нередко сокращают термин «двойное подчеркивание», сводя его до «дандера».

Одинарный перед именем: _var.

Переменная или метод, начинающиеся с одинарного символа подчеркивания, предназначаются для внутреннего пользования. Эта договоренность определена в PEP 8, руководстве по стилю оформления наиболее широко применяемого исходного кода Python

Если для импорта всех имен из модуля вы будете использовать подстановочный импорт (wildcard import) (`from module import *`), то Python не будет импортировать имена с начальным символом подчеркивания (если только в модуле не определен список `__all__`, который отменяет такое поведение)

> **PEP8. _single_leading_underscore: weak "internal use" indicator.**

Одинарный в конце имени: var_.

Иногда самое подходящее имя переменной уже занято ключевым словом языка Python. По этой причине такие имена, как `class` или `def`, в Python нельзя использовать в качестве имен переменных. В этом случае можно в конец имени добавить символ одинарного подчеркивания, чтобы избежать конфликта из-за совпадения имен. Эта договоренность определена и объяснена в PEP 8.

> **PEP8. single_trailing_underscore_: used by convention to avoid conflicts with Python keyword**

Двойной в начале имени: __var.

Префикс, состоящий из двойного символа подчеркивания, заставляет интерпретатор Python переписывать имя атрибута для того, чтобы в подклассах избежать конфликтов из-за совпадения имен.

Такое переписывание также называется искажением имени (name mangling) — интерпретатор преобразует имя переменной таким образом, что становится сложнее создать конфликты, когда позже класс будет расширен.

> PEP8. __double_leading_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, __boo becomes _FooBar__boo)

Двойной в начале и в конце имени: __var__.

Имена, у которых есть начальный и замыкающий двойной символ подчеркивания, в языке зарезервированы для специального применения. Эти дандер-методы часто упоминаются как магические методы.

Name mangling к таким именам не применяется.

> PEP8. __double_leading_and_trailing_underscore__: "magic" objects or attributes that live in user-controlled namespaces. E.g. __init__, __import__ or __file__. Never invent such names; only use them as documented.

Одинарный символ подчеркивания: _.

По договоренности одинарный автономный символ подчеркивания иногда используется в качестве имени, чтобы подчеркнуть, что эта переменная временная, незначительная или вовсе не используется.

```
for _ in range(5):
    pass

a, _, _, b = (1, 'hello', [], 1+2)
```

Магические методы. Метод super()

object.__new__(*cls*[,...])

Вызывается при создании нового экземпляра класса. Первый аргумент – класс, объект которого создается. Остальные аргументы передаются в конструктор класса.

Возвращаемым значением должен быть экземпляр класса (обычно экземпляр `cls`, но не обязательно).

Если `__new__()` возвращает объект типа `cls`, то далее вызывается конструктор класса.

Применяется для кастомизации создания экземпляров.

```
class Speedometer:
    def __init__(self, max_speed, units):
        print('__init__')
        self.max_speed = max_speed
        self.units = units

    def __new__(cls, max_speed, units):
        print('__new__')
        if max_speed > 250:
            return None
        return super().__new__(cls)
```

```
>>> s1 = Speedometer(200, 'km')
__new__
__init__
>>> s1.max_speed
200
>>> s2 = Speedometer(350, 'miles')
__new__
>>> s2 is None
True
```

object.__init__(*self*[,...])

Конструктор, выполняется при инициализации экземпляра класса непосредственно перед возвратом экземпляра вызывающему.

Классы-наследники должны вызывать конструкторы своих базовых классов для корректной инициализации.

Не должен возвращать ничего, в противном случае будет выброшено исключение `TypeError`.

```
class MyClass:
    def __init__(self, param1, param2, **kwargs):
        super().__init__(param1, param2, **kwargs)
        self.param1 = param1
        self.param2 = param2
```

object.__del__(*self*)

Вызывается перед удалением объекта, должен вызывать методы `__del__()` базовых классов.

Гарантий того, что интерпретатор вызовет `__del__()` для всех существующих объектов в момент выхода нет.

Важно: `del x` не вызывает напрямую `x.__del__()`, а лишь уменьшает счетчик ссылок на него на 1. `x.__del__()` будет вызван, когда счетчик ссылок на объект станет равным нулю.

```
class MyClass:
    def __del__(self):
        ...
        super().__del__()
        ...
```

object.__call__(*self*[, *args...*])

Выполняется, когда объект вызывают как функцию.

```
class MyClass:
    def __call__(self, new_param='default'):
        print(
            f'Теперь я веду себя как функция! '
            f'Мне передали на вход {new_param}!'
        )

>>> obj = MyClass()
>>> obj()
Теперь я веду себя как функция! Мне передали на вход default!
>>> obj(42)
Теперь я веду себя как функция! Мне передали на вход 42!
```

Строковое представление

› `object.__repr__(self)`

Вызывается встроенной функцией `repr()` и является «официальным» строковым представлением объекта.

Если возможно, должен возвращать строку, по которой можно воссоздать тот же самый объект. В противном случае, строку вида `<...описание объекта...>`. Используется для отладки.

› `object.__str__(self)`

Вызывается функциями str(), format() и print(). Используется для возврата неформального представления объекта. Если в классе объявлена функция `__repr__()` и отсутствует `__str__()`, вместо нее будет использоваться `__repr__()`.

Декораторы

Декораторы

Декоратор – это шаблон проектирования, предназначенный для статического или динамического подключения дополнительного поведения к объекту, не изменяя при этом код самого объекта, не прибегая к наследованию и не затрагивая поведения других объектов того же класса. Еще декоратор – это одноименный элемент языка Python, который позволяет реализовать упомянутый паттерн. По сути, декораторы – это функции-обертки над другими функциями или даже классами.

В Python есть пара особенностей, которые позволяют определять и использовать декораторы.

Связывание функции с именем переменной

```
def multiply(x, y):
    return x * y

>>> a = multiply
>>> a(2, 2)
4
```

Определение одной функции внутри другой

```
def multiply(x, y):
    def inner(a, b):
        return a * b
    return inner(x, y)

>>> multiply(2, 2)
4
```

Передача функции в качестве аргумента

```
def multiply(x, y):
    return x * y

def caller(func, *args, **kwargs):
    return func(*args, **kwargs)

>>> caller(multiply, 2, 2)
4
```

Функция в качестве возвращаемого значения

```
def action(name):
    def add(x, y):
        return x + y

    if name == 'add':
        return add

>>> func = action('add')
>>> func(5, 5)
10
```

Классический пример

```
import time

def measure_time(func):
    def inner(*args, **kwargs):
        start_time = time.time()
        try:
            return func(*args, **kwargs)
        finally:
            ex_time = time.time() - start_time
            print('Execution time: {ex_time:.2f} seconds')
    return inner
```

Что происходит?

Давайте разберемся, как это работает. Пусть у нас есть некоторая функция, которая получает на вход URL, делает по нему запрос и выводит статус ответа, его кодировку и длину.

```
>>> fetch_url('https://python.org')
Status: 200 OK
Encoding: utf-8
Content Length: 47433 bytes
```

Что происходит?

Продекорируем эту функцию и посмотрим, как это повлияет на ее вывод.

```
>>> fetch_url = measure_time(fetch_url)
>>> fetch_url('https://python.org')
Execution time is 1.20 seconds
Status: 200 OK
Encoding: utf-8
Content Length: 47433 bytes
```

Что происходит?

Продекорируем эту функцию и посмотрим, как это повлияет на ее вывод.

```
>>> fetch_url = measure_time(fetch_url)
>>> fetch_url('https://python.org')
Execution time is 1.20 seconds
Status: 200 OK
Encoding: utf-8
Content Length: 47433 bytes
```

Что происходит?

Видно, что в дополнение к запрашиваемым данным, мы получили время выполнения функции в секундах. А теперь давайте разберем работу декоратора по шагам.

Итак, каждый раз при вызове `measure_time`, внутри нее будет определяться функция `inner`. Внутри `inner` в замыкании находится функция `func` (наша `fetch_url`), передаваемая аргументом в `measure_time`. Вся дополнительная полезная работа происходит внутри `inner`, которая возвращается в результате вызова `measure_time` и будет использоваться вместо оригинальной `fetch_url`.

Теперь наша продекорированная `fetch_url` будет выполняться так:

1. Сохранение текущего времени в переменной `start_time` перед вызовом оригинальной `fetch_url`
2. Вызов оригинальной `fetch_url` с аргументами и возврат результата вызова
3. Вывод времени выполнения

Что происходит?

Стоит обратить внимание на то, что в `try`-блоке нет инструкции `except`, а только `finally`. Это сделано по двум причинам:

1. Нам интересен `finally` с точки зрения выполнения кода после выполнения тела в блоке `try`.

2. Отлов и обработка исключений внутри декоратора (кроме редких случаев) – это очень плохо. Делать это следует в вызывающем коде.

Цепочки из декораторов

А что, если мы захотим применить к функции два декоратора? А три? Еще больше? Возможно ли это? Конечно, это возможно. Но только при выполнении одного условия: из декоратора необходимо вернуть функцию, которая пойдет на вход следующему декоратору в цепочке. Для наглядности напишем еще один декоратор, который буд ет кешировать вызовы функций.

```
def memoize(func):
    _cache = {}
    def wrapper(*args, **kwargs):
        name = func.__name__
        key = (name, args, frozenset(kwargs.items()))
        if key in _cache:
            return _cache[key]
        result = func(*args, **kwargs)
        _cache[key] = result
        return result
    return wrapper
```

Цепочки из декораторов

```
>>> fetch_url = measure_time(memoize(fetch_url))
>>> print(fetch_url('https://python.org'))
Execution time is 1.10 seconds
Status: 200 OK
Encoding: utf-8
Content Length: 47433 bytes

>>> print(fetch_url('https://python.org'))
Execution time is 0.00 seconds
Status: 200 OK
Encoding: utf-8
Content Length: 47433 bytes
```

Щепотка сахара

Существует более простой способ декорирования функций.
Символ `@` – это синтаксический сахар для применения декоратора к функции. В случае с несколькими декораторами, они применяются последовательно снизу вверх.

```
@memoize
@measure_time
def fetch_url(url):
    ...
```

Когда использовать?

- логирование
- трассировка вызовов
- замер времени выполнения функции
- кеширование результата вызова функции
- валидация входных параметров функции
- ленивые вычисления
- установка/проверка условий до/после выполнения функции
- classmethod, staticmethod, property
- и т.д.

Статические методы и методы класса

Статический метод

Объявляется в классе через декоратор `@staticmethod`.

Статические методы не принимают `self` первым аргументом и используются в тех случаях, когда в класс нужно внести функцию, которая имеет отношение к классу, но не оперирует непосредственно над экземпляром класса.

```
class APIClient:
    def __init__(self, *args, **kwargs):
        ...

    @staticmethod
    def get_default_headers():
        return {'Content-Type': 'application/json'}
```

Метод класса

Объявляется в классе через декоратор `@classmethod`.

Первым аргументом принимает `cls` – ссылку на экземпляр своего класса. Используется в тех случаях, когда у вас есть методы, не привязанные к конкретному экземпляру, но каким-то образом оперирующие над классами.

Классические примеры – паттерны фабрика и конструктор.

```
class Pizza:
    def __init__(self, size):
        self.size = size

    @classmethod
    def get_large_pizza(cls):
        return cls(42)

    @classmethod
    def get_standard_pizza(cls):
        return cls(30)
```

```
>>> large_pizza = Pizza.get_large_pizza()
>>> large_pizza.size
42
>>> standard_pizza = Pizza.get_standard_pizza()
>>> standard_pizza.size
30
```

@property и приватные атрибуты

Для работы с приватным атрибутами класса используется декоратор `@property`.

В языке Python, вообще, нет таких атрибутов, к которым вы бы не могли получить доступ напрямую, а приватные атрибуты существуют лишь на уровне соглашения об именовании (имена начинаются с подчеркивания).

> **Общее правило: если вам приходится работать с частными атрибутами напрямую, то, либо вы что-то делаете не так, либо класс плохо спроектирован и нуждается в доработке**

@property и приватные атрибуты

```
class C:
    def __init__(self, value):
        self._value = value

    @property
    def value(self):
        print('getter')
        return self._value

    @value.setter
    def value(self, value):
        print('setter')
        self._value = value

    @value.deleter
    def value(self):
        print('deleter')
        del self._value
```

@property и приватные атрибуты

```
>>> c = C('Hello World')
>>> c.value
getter
'Hello World'
>>> c.value = 'Goodbye World'
setter
>>> c.value
getter
'Goodbye World'
>>> del c.value
deleter
>>> c.value
getter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in value
AttributeError: 'C' object has no attribute '_value'
```

Атрибуты класса

Атрибуты класса

```
class A:
    attr = 123

>>> obj1 = A()
>>> obj2 = A()
>>> obj1.attr
123
>>> obj2.attr
123
>>> A.attr = 321
>>> obj1.attr
321
>>> obj2.attr
321
```

Атрибуты класса

```
>>> obj1.attr = 42
>>> obj1.attr
42
>>> obj2.attr
321
>>> A.attr
321
>>> obj1.__dict__
{'attr': 42}
>>> obj2.__dict__
{}
```

Метаклассы

Метаклассы

В Python все является объектами. Пришло время разобраться в этом глубже и понять, объектами каких классов является это **всё.**

Метадетектив

```
>>> type(1)
<class 'int'>

>>> type('str')
<class 'str'>

>>> class A: pass
>>> a = A()
>>> type(a)
<class '__main__.A'>
```

Метадетектив

```
>>> def f(): pass
>>> f.__class__
<class 'function'>

>>> class A: pass
>>> A.__class__
<class 'type'>
```

Метадетектив

Функция `f` является экземпляром `function`. А класс `A` – экземпляром `type` (или проще сказать типом).

Для определения типа мы использовали функцию `type` (которая, на самом деле не функция, а самый настоящий класс). С помощью `type` можно создавать и объекты классов.

Итак, метакласс – это объект, управляющий созданием классов.

Metatype

Принимает на вход 3 параметра:

- Имя создаваемого класса
- Список его наследников, в порядке наследования
- Словарь с атрибутами класса

Metatype

```
class A:
    a = 1

>>> C = type('C', (A,), {})
>>> c = C()
>>> c.a
1

>>> D = type('D', (A,), {'a':3, 'b': lambda self: print('Я метод b')})
>>> d = D()
>>> d.a
3
>>> d.b()
Я метод b
```

Метапример

Как интерпретатор обрабатывает конструкцию `class`:

- Изменяет имена атрибутов, начинающихся с префикса `__`
- Выполняет тело класса как обычный код инструкция за инструкцией
- Передаёт имя класса, список базовых классов и словарь атрибутов конструктору метакласса (нашему `type`)
- Метакласс создает на основе полученных данных объект класса

Метапример

```
class Template(metaclass=_TemplateMetaclass):
    """A string class for supporting $-substitutions."""

    delimiter = '$'
    idpattern = r'[_a-z][_a-z0-9]*'
    flags = _re.IGNORECASE

    def __init__(self, template):
        self.template = template
```

Метапример

```
class _TemplateMetaclass(type):
    pattern = r"""
    %(delim)s(?:(?P<escaped>%(delim)s) | (?P<named>%(id)s) |
      {(?P<braced>%(id)s)} | (?P<invalid>))
    """

    def __init__(cls, name, bases, dct):
        super(_TemplateMetaclass, cls).__init__(name, bases, dct)
        if 'pattern' in dct:
            pattern = cls.pattern
        else:
            pattern = _TemplateMetaclass.pattern % {
                'delim' : _re.escape(cls.delimiter),
                'id'    : cls.idpattern,
                }
        cls.pattern = _re.compile(pattern, cls.flags | _re.VERBOSE)
```

Метапример

Метод `__init__` метакласса, берёт значение некоторых атрибутов класса (`pattern`, `delimiter`, `idpattern`) и использует их для построения сложного регулярного выражения, которое, в свою очередь сохраняется в атрибуте класса `pattern`.

Можно ли было обойтись без метакласса? Можно, но в этом случае сложный `pattern` будет компилироваться каждый раз при создании нового объекта `Template`.

В случае же использования метакласса, происходит некоторая оптимизация и регулярное выражение компилируется только однажды, в момент создания самого класса `Template` то есть, при загрузке модуля.

Если ранее мы рассматривали класс и экземпляр, как чертеж и деталь, то метакласс можно сравнить с инженером, который чертит эти чертежи.

Итак, основная цель метаклассов – автоматически изменять класс в процессе создания и контролировать сам этот процесс.

Метадзен от гуру питона

Метаклассы - это очень глубокая материя, о которой 99% пользователей даже не нужно задумываться. Если вы не понимаете, зачем они вам нужны – значит, они вам не нужны (люди, которым они на самом деле требуются, точно знают, что они им нужны, и им не нужно объяснять - почему). - Тим Питерс (Tim Peters)

Дескрипторы

Дескрипторы

Дескрипторы – это классы, реализующие определенный протокол.

```
descr.__get__(self, obj, type=None) -> value
descr.__set__(self, obj, value) -> None
descr.__delete__(self, obj) -> None
```

Дескрипторы разделяют на две группы:

- дескриптор данных (data descriptor) – определены `__get__` и `__set__`
- дескриптор не данных (non-data descriptor) – только `__get__`

Дескрипторы

`__get__` предназначен для получения значения дескриптора. Входные параметры: `instance` - инициализированный объект класса, `type` - тип класса.

`__set__` предназначен для установки значения дескриптора. Входные параметры: `instance` - инициализированный объект класса, `value` – сохраняемое значение.

`__delete__` предназначен для удаления значения дескриптора. Входные параметры: `instance` - инициализированный объект класса.

Дескрипторы

Допустим, у нас есть задача, в которой требуется описать автомобиль – марку и год выпуска, при этом марка может быть только строкой, а год выпуска – целым числом.

Простая структура данных не подойдет, поскольку не учитывает наших ограничений.

```
class Car:
    brand = None
    year = None

>>> Car.brand = 2019
>>> Car.year = 'Honda'
```

Дескрипторы

Класс с проверками в `__init__` тоже не подойдет, поскольку никто не помешает переопределить значения у экземпляра. Да и кучу лишней логики тоже не хотелось бы там держать.

```
class Car:
    def __init__(self, brand, year):
        if isinstance(brand, str):
            self.brand = brand
        else:
            raise TypeError('brand is not a string')
        if isinstance(year, int):
            self.year = year
        else:
            raise TypeError('brand is not an integer')

>>> car = Car('Honda', 2019)
>>> car.year = '2010'
```

Дескрипторы

Для контроля над вводимыми данными воспользуемся дескрипторами.

Определим по классу-дескриптору для каждого типа данных, которыми мы оперируем.

```
class StringType:
    def __init__(self, name):
        self.name = name

    def __set__(self, instance, value):
        if isinstance(value, str):
            instance.__dict__[self.name] = value
        else:
            raise TypeError(f'{value} is not a string')

    def __get__(self, instance, class_):
        return instance.__dict__[self.name]
```

Дескрипторы

Для контроля над вводимыми данными воспользуемся дескрипторами.

Определим по классу-дескриптору для каждого типа данных, которыми мы оперируем.

```
class IntegerType:
    def __init__(self, name):
        self.name = name

    def __set__(self, instance, value):
        if isinstance(value, int):
            instance.__dict__[self.name] = value
        else:
            raise TypeError(f'{value} is not an int')

    def __get__(self, instance, class_):
        return instance.__dict__[self.name]
```

Дескрипторы

Или даже один для всего.

```
class TypeChecker:
    def __init__(self, name, value_type):
        self.name = name
        self.value_type = value_type

    def __set__(self, instance, value):
        if isinstance(value, self.value_type):
            instance.__dict__[self.name] = value
        else:
            raise TypeError(
              'f{value} must be {self.value_type}'
            )

    def __get__(self, instance, class_):
        return instance.__dict__[self.name]
```

Дескрипторы

```
class Car:
    brand = StringType('brand')
    year = IntegerType('year')

    def __init__(self, brand, year):
        self.brand = brand
        self.year = year
```

```
class Car:
    brand = TypeChecker('brand', str)
    year = TypeChecker('year', int)

    def __init__(self, brand, year):
        self.brand = brand
        self.year = year
```

Дескрипторы

```
>>> car = Car('Toyota', 2015)
>>> car.brand
'Toyota'
>>> car.year
2015
>>> car.year = '2015'
TypeError                                 Traceback (most recent call last)
----> 1 car.year  = '2015'
---> 10                 raise TypeError(f'{value} must be {self.value_type}')
TypeError: 2015 must be <class 'int'>
```

Оптимизация. Слоты

Оптимизация. Слоты

Бывают моменты, когда нужно создать много одинаковых объектов. Для экономии памяти в таких случаях и улучшения производительности придумали `__slots__`. Наличие `__slots__` ограничивает возможные имена атрибутов объекта теми, которые там указаны. `__dict__` в таком случае не создается, поскольку все атрибуты и так уже известны.

Оптимизация. Слоты

```
class Slotter:
    __slots__ = ['a', 'b']

    def get_values(self):
        return (self.a, self.b,)
```

```
>>> s = Slotter()
>>> s.a = 123
>>> s.b = 'qwe'
>>> s.a
123
>>> s.b
'qwe'
>>> s.c = 42
AttributeError: 'Slotter' object has no
attribute 'c'
>>> s.get_values()
(123, 'qwe')
```

Яндекс

Благодарю за внимание. Вопросы?

Валерий Лисай

Группа серверной разработки клиентского
продукта Яндекс.Такси

hairspray@yandex-team.ru

hairspray