Глава 19. Мета программирование

Динамические атрибуты и свойства

> Ценность свойств заключается в том, что благодаря им можно совершенно безопасно — и это даже рекомендуется — раскрывать атрибуты-данные как часть открытого интерфейса класса[1].
>
> — Алекс Мартелли, один из разработчиков Python и автор книги.

Атрибуты-данные и методы в Python носят общее название «атрибуты»; **метод** — это просто *вызываемый* атрибут. Помимо атрибутов-данных и методов, мы можем создавать ещё свойства, позволяющие заменить открытые атрибуты-данные методами-акцепторами (т.е. методами чтения и установки), не изменяя интерфейс класса. Это согласуется с *принципом единообразного доступа*.

Все сервисы, предоставляемые модулем, должны быть доступны с помощью единообразной нотации, скрывающей механизм реализации: хранение или вычисление[2].

Помимо свойств, Python предлагает богатый API для управления доступом к атрибутам и реализации динамических атрибутов. Интерпретатор вызывает специальные методы `__getattr__` и `__setattr__` при использовании нотации доступа к атрибутам с помощью точки (например, `obj.attr`). Пользовательский класс, в котором имеется метод `__getattr__`, может реализовать "виртуальные атрибуты", вычисляемые "на лету", когда программа пытается прочитать несуществующий атрибут.

Динамические атрибуты — вид метапрограммирования, обычно применяемый создателями каркасов.

osconfeed.py: *загрузка файла osconfeed.json*

```
import json
import os
import warnings
from urllib.request import urlopen

URL = 'http://www.oreilly.com/pub/sc/osconfeed'
JSON = 'data/app.json'


def load():
    if not os.path.exists(JSON):
        msg = 'downloading {URL} to {JSON}'
        warnings.warn(msg)
        with urlopen(URL) as remote, open(JSON, 'wb') as local:
            local.write(remote.read())

    with open(JSON) as fp:
        return json.load(fp)


if __name__ == '__main__':
    load()
```

Исследование JSON-подобных данных с динамическими атрибутами

Класс `FrozenJSON` позволяет читать атрибуты, например `name`, и вызывать методы, например `.keys()` и `.items()`

```
from collections import abc


class FrozenJSON:
    """
    Допускающий только чтение фасад для навигации по
    JSON-подобному объекту с применением нотации атрибутов
    """

    def __init__(self, mapping):
        """
        Строим объект dict по аргументу mapping. Тем самым
        мы решаем 2 задачи: проверяем, что получили словарь
        и для безопасности делаем его копию.
        """
        self.__data = dict(mapping)

    def __getattr__(self, item):
        """
        Метод вызывается только когда не существует атрибута
        с таким именем.
        """
        if hasattr(self.__data, item):
            return getattr(self.__data, item)
        else:
            return FrozenJSON.build(self.__data[item])

    @classmethod
    def build(cls, obj):
        if isinstance(obj, abc.Mapping):
            return cls(obj)
        elif isinstance(obj, abc.MutableSequence):
            return [cls.build(item) for item in obj]
        else:
            return obj
```

Проблема не допустимого элемента

Суть простая, если атрибут - это зарезервированное слово из Python, то при попытке запросить данный атрибут мы получим синтаксическую ошибку. Что бы поправить этот косяк необходимо для таких атрибутов добавлять в конце символ "_".

Для этого достаточно изменить однострочный класс `__init__`:

```
import keyword

from explore0 import FrozenJSON


class FrozenJSON1(FrozenJSON):

    def __init__(self, mapping):
        self.__data = {}
        for key, value in mapping.items():
            if keyword.iskeyword(key):
                key += '_'
            self.__data[key] = value
```

Гибкое создание объектов с помощью метода `__new__`

Мы часто называем `__init__` конструктором, но это только потому, что позаимствовали терминологию из других языков. На самом деле конструирует экземпляр специальный метод `__new__`. Этот метод класса (однако он обрабатывается особым образом, поэтому декоратор `@classmethod` не используется), и возвращать он должен экземпляр. Этот экземпляр затем передаётся в качестве первого аргумента `self` методу `__init__`. Поскольку `__init__` при вызове уже получает экземпляр, что-то возвращать ему запрещено, по существу, метод `__init__` является "инициатором". *Настоящий конструктор* - это метод `__new__`, но мы о нём редко вспоминаем, потому что реализации, унаследованной от класса `object`, обычно достаточно.

Описанный только что путь — от `__new__` к `__init__` — самый распространённый, но не единственный. Метод `__new__` может возвращать и экземпляр другого класса; если такое происходит, то интерпретатор не вызывает `__init__`.

Псевдокод конструирования объекта

```
def object_maker(the_class, some_arg):
    new_object = the_class.__new__(some_arg)
    if isinstance(new_object, the_class):
        the_class.__init__(new_object, some_arg)
    return new_object

# следующие предложения приблизительно эквивалентны
x = Foo('bar')
x = object_maker(Foo, 'bar')
```

`explore2.py`: *использование* `new` *вместо* `build` *для конструирования новых объектов, которые могут быть или не быть экземплярами FrozenJSON.*

```
from collections import abc
from keyword import iskeyword
```

```
class FrozenJSON:
    """
    Допускающий только чтение фасад для навигации по
    JSON-подобному объекту с применением нотации атрибутов.
    """

    def __new__(cls, arg):
        """
        Будучи методом класса, __new__ получает в качестве
        первого аргумента сам класс, а остальные аргументы
        - те же, что получает __init__, за исключением self.
        """
        if isinstance(arg, abc.Mapping):
            """
            По умолчанию работа делегируется методу __new__
            суперкласса. В данном случае мы вызываем метод
            __new__ из базового класса object, передавая
            ему FrozenJSON в качестве единственного аргумента.
            """
            return super().__new__(cls)
        elif isinstance(arg, abc.MutableMapping):
            """
            Оставшаяся часть __new__ ничем не отличается от
            прежнего метода build.
            """
            return [cls(item) for item in arg]
        else:
            return arg

    def __init__(self, mapping):
        self.__data = {}
        for key, value in mapping.items():
            if iskeyword(key):
                key += '_'
            self.__data[key] = value

    def __getattr__(self, item):
        if hasattr(self.__data, item):
            return getattr(self.__data, item)
        else:
            """
            Здесь раньше вызывался метод FrozenJSON.build,
            а теперь мы просто вызываем конструктор FrozenJSON.
            """
            return FrozenJSON(self.__data[item])
```

Использование свойств для контроля атрибутов

LineItem, попытка №1: класс строки заказа

Представим себе приложение для магазина, который продаёт натуральные пищевые продукты на развес. В такой системе заказ состоит из последовательности строк, а каждую строку можно представить классом, показанным в примере:

`bulkfood_v1.py`: *простейший класс* `LineItem`

```
class LineItem:
    def __init__(self, description, weight, price):
        self.description = description
        self.wight = weight
        self.price = price

    def subtotal(self):
        return self.wight * self.price
```

Красиво и просто. Пожалуй, слишком просто. Ниже обозначим проблемку:

```
>>> from source.bulkfood_v1 import LineItem
>>> raisins = LineItem('Golden raisins', 10, 6.95)
>>> raisins.subtotal()
69.5
>>> raisins.wight = -20 # Мусор на входе
>>> raisins.subtotal()  # Мусор на выходе
-139.0
```

Как это исправить? Можно было бы изменить интерфейс класса `LineItem`, добавив методы чтения и установки атрибута `weight`. Так поступают в Java, и ничего плохого в этом нет.

С другой стороны, было бы естественно устанавливать атрибут `weight` элементы заказа, просто присваивая ему значение, да и не исключено, что в других частях эксплуатируемой системы уже встречается прямой доступ к атрибуту вида `item.weight`. В таком случае следовало бы заменить атрибут-данные свойством — это было бы в духе Python.

LineItem, попытка №2: контролирующее свойство

bulkfood_v2.py: *класс* LineItem *со свойством* weight

```
class LineItem:

    def __init__(self, description, weight, price):
        """
        Здесь уже используется метод установки свойства,
        который гарантирует, что не будет создан экземпляр
        с отрицательным значением weight
        """
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.price * self.weight

    @property   # Декоратор @property обозначает метод чтения свойства.
    def weight(self):
        """
        Имена всех методов, реализующих свойство, совпадают с именем
        открытого атрибута: weight
        Фактическое значение хранится в закрытом атрибуте self.__weight
        """
        return self.__weight

    @weight.setter
    def weight(self, value):
        """
        У декорированного метода чтения свойства имеется атрибут
        .setter, который является также и декоратором; тем самым
        методы чтения и установки связываются между собой.
        """
        if value > 0:
            self.__weight = value
        else:
            raise ValueError('value must be > 0')
```

Теперь объект LineItem *с недопустимым весом создать не возможно*

```
>>> from source.bulkfood_v2 import LineItem
>>> apples = LineItem('Ligol apple', 1.5, 140)
>>> apples.subtotal()
210.0
>>> apples.weight = -2
Traceback (most recent call last):
...
    raise ValueError('value must be > 0')
ValueError: value must be > 0
```

Было бы неплохо защитить от подобных ошибок и поле с ценой и преобразовать в `price` в свойство, но это бы повлекло за собой частичное повторение кода. Лекарство от повторения — абстрагирование.

Существует 2 способа абстрагировать определения свойств: фабрика свойств и дескрипторный класс. Продолжим наше исследование и реализуем фабрику свойств в виде функции.

Правильный взгляд на свойства

Встроенная сущность `property` часто используется как декоратор, но в действительности она является классом.

Сигнатура конструктора класса `property`:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

Все аргументы необязательны; если для какого-то из них не указана функция, то результирующий объект свойства не поддерживает соответствующую операцию.

Тип `property` появился в версии Python 2.2, но синтаксис декоратора был добавлен только в версии Python 2.4, т.е. на протяжении нескольких лет свойства нужно было определять, передавая функции-акцессоры в первых двух аргументах.

`bulkfood_v2b.py`: *то же, что и в примере выше, но без декоратора*

```
class LineItem:
    def __init__(self, description, weight, price):
        self.description = description
        self.wight = weight
        self.price = price

    def subtotal(self):
        return self.wight * self.price

    def get_weight(self):
        # Простой метод чтения
        return self.__weight

    def set_weight(self, value):
        # Простой метод установки
        if value > 0:
            self.__weight = value
        else:
            raise ValueError('value must be > 0')

    # Строим свойство и присваиваем его открытому атрибуту класса.
    weight = property(get_weight, set_weight)
```

В некоторых случаях классическая форма удобнее синтаксиса декораторов, одним из примеров является код фабрики свойств, который мы вскоре обсудим. С другой стороны, в теле класса, где много методов, декораторы позволяют сразу опознать методы чтения и установки, не полагаясь на соглашение о префиксах `get` и `set` в именах.

Свойства переопределяют атрибуты экземпляра

Если экземпляр и его класс оба имеют атрибут-данные с одним и тем же именем, то атрибут экземпляра переопределяет или маскирует атрибут класса - по крайней мере, когда мы обращаемся к атрибуту от имени этого экземпляра.

Атрибут экземпляра маскирует атрибут-данные класса

```
class Class:
    """
    Определяем Class с двумя атрибутами класса:
    атрибутом-данными data и свойством prop.
    """
    data = 'the class data attr'
    @property
    def prop(self):
        return 'the prop value'
```

```
>>> obj = Class()
>>> vars(obj)         # vars возвращает атрибут __dict__ объекта obj;
{}                    # как видим, атрибутов экземпляра в нём нет.

>>> obj.data          # Чтение из obj.data возвращает значение Class.data.
'the class data attr'

>>> obj.data = 'bar'  # Запись в obj.data создаёт атрибут экземпляра.
>>> vars(obj)         # Инспектируем, чтобы узнать, какие у него атрибуты.
{'data': 'bar'}

>>> obj.data          # Теперь, читая obj.data, мы получим значение атрибута/
'bar'

>>> Class.data        # Атрибут Class.data не изменился.
'the class data attr'

>>> Class.prop        # Чтение prop из Class возвращает сам объект свойств,
...                   # при этом его метод чтения не выполняется.
<property object at 0x0000018E39C619E0>

>>> obj.prop          # Чтение obj.prop приводит к выполнению метода чтения.
'the prop value'

>>> obj.prop = 'foo'  # Попытка установить атрибут экземпляра prop завершается
ошибкой.
Traceback (most recent call last):
...
  File "<input>", line 1, in <module>
AttributeError: can't set attribute 'prop'

>>> obj.__dict__['prop'] = 'foo'    # Запись 'prop' напрямую в obj.__dict__ работает.
```

```
>>> vars(obj)         # Теперь у obj есть два атрибута экземпляра: data и prop.
{'prop': 'foo', 'data': 'bar'}

>>> obj.prop          # Однако при чтении obj.prop по-прежнему выполняется метод
'the prop value'      # чтения свойства. Свойство не маскируется атрибутом экземпляра.

>>> Class.prop = 'baz'# В случае перезаписывания Class.prop объект свойств
уничтожается.
>>> obj.prop          # Теперь чтение obj.prop возвращает атрибут экземпляра.
'foo'                 # Class.prop больше не является свойством.

>>> obj.data          # obj.data возвращает атрибут экземпляра data.
'bar'
>>> Class.data        # Class.data возвращает атрибут класса data.
'the class data attr'
                      # Перезаписываем Class.data новым свойством
>>> Class.data = property(lambda self: 'the "data" prop value')
>>> obj.data          # Теперь Class.data маскирует obj.data.
'the "data" prop value'
>>> del Class.data    # Удаляем свойство.
>>> obj.data          # Теперь obj.data снова возвращает атрибут экземпляра data.
'bar'
```

В этом разделе мы, хотели показать, что при вычислении выражения вида `obj.attr` поиск `attr` начинается не с `obj`. На самом деле, поиск начинается с `obj.__class__` и, только если в классе не существует свойства с именем `attr`, то Python заглядывает в сам объект `obj`. Это правило применимо не только к свойствам, но и к целой категории дескрипторов: *переопределяющим дескрипторам*.

Документирование свойств

Когда функция оболочки `help()` или интегрированной среде разработки нужно вывести документацию по свойству, она получает информацию из атрибута свойства `__doc__`.

Конструктор класса `property` может получить строку документации в виде аргумента `doc`:

```
weight = property(get_weight, set_weight, doc='Вес в килограммах')
```

Если свойство объявлено с помощью декоратора, то строка документации метода чтения - того, который снабжён декоратором `@property`, — становится документацией свойства в целом.

```
class Foo:
    @property
    def bar(self):
        """The bar attribute"""
        return self.__dict__['bar']


    @bar.setter
    def bar(self, value):
        self.__dict__['bar'] = value
```

Строка документации для класса из примера

```
>>> help(Foo.bar)
Help on property:
    The bar attribute

>>> help(Foo)
Help on class Foo in module __main__:
class Foo(builtins.object)
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  bar
 |      The bar attribute
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __annotations__ = {}
```

Программирование фабрики свойств

Мы создадим фабрику свойств `quantity`.

Свойства - атрибуты класса. При создании каждого свойства с помощью `quantity` мы должны передать им имя атрибута `LineItem`, который будет управляться этим свойством.

`bulkfood_v2prop.py`: *фабрика свойств* `quantity` *в действии*

```
def quantity(storage_name):
    """
```

```
    :param storage_name: определяет, где хранятся
    данные свойства; в случае свойства weight данные
    будут храниться в атрибуте с именем 'weight'.

    :return: Конструируем и возвращаем объект свойства.
    """
    def qty_getter(instance):
        """
        Называть первым аргумент метода qty_getter
        именем self было бы не совсем правильно,
        т.к. это не тело класса;

        :param instance: ссылается на экземпляр
        LineItem, в котором будет храниться объект.

        :return: метод ссылается на storage_name,
        поэтому будет сохранён в замыкании этой функции;
        значение берётся непосредственно из instance.__dict__,
        чтобы обойти свойство и избежать бесконечной рекурсии.
        """
        return instance.__dict__[storage_name]

    def qty_setter(instance, value):
        if value > 0:
            instance.__dict__[storage_name] = value
        else:
            raise ValueError('value must be > 0')

    return property(qty_getter, qty_setter)


class LineItem:
    """
    Используем фабрику для определения первого
    свойства, weight, в виде атрибута класса.
    """
    weight = quantity('weight')
    # Здесь создаётся второе свойство, price.
    price = quantity('price')

    def __init__(self, description, weight, price):
        self.description = description
        """
        Здесь свойство уже работает, и поэтому
        попытка присвоить weight нулевое или
        отрицательное значение отвергается.
        """
        self.weight = weight
        self.price = price

    def subtotal(self):
```

```
        """
        Здесь свойства также работают: с их
        помощью производится доступ к значениям,
        хранящимся в экземпляре.
        """
        return self.weight * self.price
```

bulkfood_v2prop.py: *фабрика свойств* quantity

```
>>> nutmeg = LineItem('Moluccan nutmeg', 8, 13.95)
>>> nutmeg.weight, nutmeg.price   # Чтение weight и price с помощью свойств
...                               # маскирует одноимённые атрибуты экземпляра.
(8, 13.95)
>>> sorted(vars(nutmeg).items())  # Используйте метод vars, чтобы проинспектировать
...                               # экземпляр nutmeg: видно, в каких точно атрибутах
...                               # экземпляра хранится значения.
[('description', 'Moluccan nutmeg'), ('price', 13.95), ('weight', 8)]
```

Удаление атрибутов

Напомним, что в учебном пособии по Python описано предложение `del` *для удаления атрибутов объекта.*

```
del my_object.an_attribute
```

В определении свойства декоратор `@my_property.deleter` используется, что бы обернуть метод, отвечающий за удаление атрибута, управляемого свойством.

`blackknight.py`: *идея подсказана персонажем Black Knight из скетча "Monty Python and the Holy Grain".*

```
class BlackKnight:

    def __init__(self):
        """
        >>> knight = BlackKnight()
        >>> knight.member
        следующий член:
        'рука'
        >>> del knight.member
        ЧЕРНЫЙ РЫЦАРЬ (утрачена рука)
        -- Это всего лишь царапина.
        >>> del knight.member
        ЧЕРНЫЙ РЫЦАРЬ (утрачена вторая рука)
        -- Это всего лишь поверхностная рана.
        >>> del knight.member
        ЧЕРНЫЙ РЫЦАРЬ (утрачена нога)
        -- Я неуязвим!
        >>> del knight.member
        ЧЕРНЫЙ РЫЦАРЬ (утрачена вторая нога)
        -- Ну ладно, пусть будет ничья.
        """
        self.members = [
            'рука', 'вторая рука',
            'нога', 'вторая нога'
        ]
        self.phrases = [
            'Это всего лишь царапина.',
            'Это всего лишь поверхностная рана.',
            'Я неуязвим!',
            'Ну ладно, пусть будет ничья.'
        ]

    @property
    def member(self):
        print('следующий член:')
        return self.members[0]

    @member.deleter
    def member(self):
        print(f'ЧЕРНЫЙ РЫЦАРЬ (утрачена {self.members.pop(0)})\n--
{self.phrases.pop(0)}')


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Если используетесь не декоратор, а классический синтаксис, то для задания метода удаления применяется именованный аргумент `fdel`.

```
member = property(member_getter, fdel=member_deleter)
```

Если вы не пользуетесь свойствами, то для удаления атрибута можно было бы также реализовать низкоуровневый метод `__delattr__`, описанный в разделе *"Специальные методы для управления атрибутами"* ниже.

Важные атрибуты и функции для работы с атрибутами

Специальные атрибуты, влияющие на обработку атрибутов

`__class__`

Ссылка на класс объекта (т.е. `obj.__class__` — то же самое, что `type(obj)`). Python ищет специальные методы, например `__getattr__`, только в классе объекта, а не в самих экземплярах.

`__dict__`

Отображение, в котором хранится изменяемые атрибуты объекта или класса. Если у объекта есть этот атрибут, то его можно в любой момент наделить новыми атрибутами. Если в классе есть атрибут `__slots__`, то у его экземпляра не может быть атрибута `__dict__`.

`__slots__`

Этот атрибут можно определить в классе, чтобы ограничить состав атрибутов у экземпляров этого класса. `__slots__` представляет собой кортеж строк с именами допустимых атрибутов[3].

Встроенные функции для работы с атрибутами

`dir([object])`

Перечисляет большую часть атрибутов объекта. В официальной документации сказано, что дананя функция предоставляет не все атрибуты, а наиболее интересные[4].

`getattr(object, name[, default])`

Получает атрибут, идентифицируемый строкой `name`, объекта `object`. В результате может быть найден атрибут, определённый в классе или супер-классе объекта. Если такого атрибуты не существует, возбуждается исключение `AttributeError`, либо возвращает значении `default`, если оно задано.

`hasattr(object, name)`

Возвращает `True`, если атрибут с указанным именем существует в объекте `object` или может быть найден с его помощью (например, в результате наследования).

`setattr(object, name, value)`

Присваивает значение `value` поименованному атрибуту `object`, если это допускается. В

результате может быть создан новый атрибут или переименован старый.

`vars([object])`

Возвращает атрибут `__dict__` объекта `object`; функция `vars` не умеет работать с классами, в которых определён атрибут `__slots__` и нет атрибута `__dict__` (в отличии от фунции `dir`, которая справляется с такими экземплярами). Без аргумента `vars()` делает то же самое, что `locals(): возвращает словарь, описывающий локальную область видимости.

Специальные методы для работы с атрибутами

Специальные методы, описанные ниже, отвечают за чтение, установку, удаление и получение списка атрибутов(если они реализованы в пользовательском классе).

`__delattr__(self, name)`

Вызывается при любой попытке удалить атрибут в предложении `del`, например, `del obj.attr` приводит к вызову `Class.__delattr__(obj, 'attr')`.

`__dir__(self)`

Вызывается при вызове `dir` для объекта с целью получить список атрибутов.

`__getattr__(self, name)`

Вызывается только тогда, когда попытка найти поименованный атрибут в obj.Class и супер-классах завершается неудачно.

`__getattribute__(self, name)`

Вызывается при любой попытке получить поименованный атрибут за исключением случаев, когда искомый атрибут является специальным атрибутом или методом. К вызову этого метода приводит использование нотации с точкой и встроенной функции `getattr` и `hasattr`.

`__setattr__(self, name, value)`

Вызвается при любой попытке установить поименованный атрибут. К вызову этого метода приводит использование нотации с точкой и встроенной функции `setattr`.

[1] Alex Martelli «Python in a Nutshell», издание 2 (O'Reilly), стр 101.

[2] Bertrand Mayer, Object-Oriented Software Construction, издание 2, стр. 57.

[3] Алекс Мартелли отмечает, что `__slots__` может быть и списком, но лучше не оставлять места для недоразумений и всегда использовать кортеж, потому что изменение списка, хранящегося в `__slots__`, после обработки тела класса интерпретатором, не возымеет никакого эффекта, так что использование здесь изменяемой последовательности лишь стало бы причиной вредных иллюзий.

[4] **Примечание** Поскольку `dir()` предоставляется в первую очередь для удобства использования в интерактивной подсказке, он пытается предоставить интересный набор имен больше, чем пытается предоставить строго или последовательно определенный набор имен, и его детальное поведение может меняться в разных выпусках. Например, атрибуты метакласса отсутствуют в списке результатов, если аргументом является класс.