PYTHON

Лекция 12

ПЛАН ЛЕКЦИИ

- Дескрипторы

МОТИВАЦИЯ

- Уже видели два механизма работы с классами из простых атрибутов
- `namedtuple` и `dataclass`
- А что, если класс сложный, но в нем есть пара простых атрибутов ?
- Или мы на лету хотим вычислять аттрибут из других атрибутов ?

МОТИВАЦИЯ

- У `ndarray` в `numpy` есть операция транспонирования
- Один из способов ее записи - `.T`
- Формально выглядит как атрибут
- Но выглядит так, как будто он что-то делает при обращении
- Как ему это удается ?

ФУНКЦИЯ `property`

- Python предоставляет функцию `property`
- Чисто формально ей можно передать три метода
- Она создаст псевдоатрибут с заданным именем
- И будет их вызывать по факту чтения, записи и удаления этого атрибута

ПРИМЕР

```
 1 class C:
 2     def __init__(self):
 3         self._x = None
 4
 5     def getx(self):
 6         return self._x
 7
 8     def setx(self, value):
 9         self._x = value
10
11 # ......................
```

ПРИМЕР

```
# .....................

    def delx(self):
        del self._x

    x = property(getx, setx, delx,
                 "I'm the 'x' property.")

c = C()
c.x = 5
print(c.x)
print(c._x)
c._x = 10
print(c.x)
```

РАЗБЕРЕМ

- Функция `property` создает нечто, что умеет делегировать операции с собой методам класса
- Благодаря этому может управлять чтением, изменением и удалением
- Как минимум - запретить удалять или модифицировать
- Но не очень понятно, как оно работает

ПРИМЕР

```
 1 class Ten:
 2     def __get__(self, obj, objtype=None):
 3         return 10
 4
 5 class A:
 6     x = 5
 7     y = Ten()
 8
 9
10 a = A()
11 print(a.x)
12 print(a.y)
```

РАЗБЕРЕМ

- Есть механизм делегирования
- Обнаружив объект с методом `__get__`, Python вызывает его
- И его результат будет значением объекта
- Но вне класса это не работает

ФОРМАЛЬНО

- Дескриптором называем любой объект, который определяет методы `__get__`, `__set__` или `__delete__`
- Сам по себе это обычный объект
- Особое поведение проявляется только тогда, когда он - атрибут класса
- Если его присвоить непосредственно атрибуту объекта - поведение будет обычное

ПРИМЕР

```
 1 class Ten:
 2     def __get__(self, obj, objtype=None):
 3         return 10
 4
 5 class A:
 6     x = 5
 7     y = Ten()
 8
 9     def __init__(self):
10         self.z = Ten()
11
12 # ........................
```

ПРИМЕР

```
 1 # ........................
 2
 3 a = A()
 4 print(a.x)
 5 print(a.y)
 6 print(a.z)
 7 print(A.y)
 8
 9 t = Ten()
10 print(t)
```

ЗАВИСИМЫЙ АТРИБУТ

- Можно реализовать зависимый атрибут
- Без `property`, на дескрипторах
- Например, есть имя каталога в файловой системе
- Хотим свойство, возвращающее количество файлов в нем
- Свойство над имеющимся состоянием, не порождающее нового

ПРИМЕР

```
1 import os
2 import sys
3
4 class DirectorySize:
5
6     def __get__(self, obj, objtype=None):
7         return len(os.listdir(obj.dirname))
8
9 # .....................
```

ПРИМЕР

```
 1 # ......................
 2
 3 class Directory:
 4
 5     size = DirectorySize()
 6
 7     def __init__(self, dirname):
 8         self.dirname = dirname
 9
10 s = Directory(sys.argv)
11 print(s.size)
```

ДЕСКРИПТОРЫ С СОСТОЯНИЕМ

- Читать каталог может быть дорого
- В некоторых ОС можно подписываться на обновления каталога
- Дескриптор может это сделать, чтобы отслеживать судьбу файлов каталога
- И возвращать ее в методе `__get__`

ДЕСКРИПТОРЫ С СОСТОЯНИЕМ

- Можно мониторить и/или профилировать обращение к свойству
- В чем-то это похоже на декораторы к функциям
- Только без синтакической поддержки
- И нет встроенной поддержки для вложенности

ЛОГИРОВАНИЕ ДОСТУПА

- Хотим логировать доступ к атрибуту объекта класса
- Тоже без `property`, на дескрипторах
- Дескриптор будет участвовать в хранении состояния
- Для начала - завязавшись на имя поля

ПРИМЕР

```
 1 import logging
 2 logging.basicConfig(level=logging.INFO)
 3
 4 class LoggedAgeAccess:
 5     def __get__(self, obj, objtype=None):
 6         value = obj._age
 7         logging.info('Accessing %r giving %r',
 8                      'age', value)
 9         return value
10
11     def __set__(self, obj, value):
12         logging.info('Updating %r to %r', 'age', value)
13         obj._age = value
14 # ............................
```

ПРИМЕР

```
# ............................

class Person:
    age = LoggedAgeAccess()

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

    def birthday(self):
        self.age += 1

# ............................
```

ПРИМЕР

```
# ............................

mary = Person('Mary M', 30)
dave = Person('David D', 40)

print(vars(mary))
print(vars(dave))

v = mary.age
mary.birthday()
v = dave.name
v = dave.age

print(dave.name)
```

УЛУЧШИМ

- Не хотим завязываться на имя свойства
- Для этого воспользуемся еще одним контрактом
- В момент присваивания будет вызван `__set_name__` в десткрипторе
- И там мы узнаем наше имя

ПРИМЕР

```
 1 import logging
 2
 3 logging.basicConfig(level=logging.INFO)
 4
 5 class LoggedAccess:
 6
 7     def __set_name__(self, owner, name):
 8         self.public_name = name
 9         self.private_name = '_' + name
10
11 # ..............................
```

ПРИМЕР

```
# ..............................

    def __get__(self, obj, objtype=None):
        value = getattr(obj, self.private_name)
        logging.info('Accessing %r giving %r',
                     self.public_name, value)
        return value

    def __set__(self, obj, value):
        logging.info('Updating %r to %r',
                      self.public_name, value)
        setattr(obj, self.private_name, value)

# ......................
```

ПРИМЕР

```
 1 # ..........................
 2
 3 class Person:
 4
 5     name = LoggedAccess()
 6     age = LoggedAccess()
 7
 8     def __init__(self, name, age):
 9         self.name = name
10         self.age = age
11
12     def birthday(self):
13         self.age += 1
14 # ..........................
```

ПРИМЕР

```
 1 # ..........................
 2
 3 mary = Person('Mary M', 30)
 4 dave = Person('David D', 40)
 5
 6 print(vars(mary))
 7 print(vars(dave))
 8
 9 v = mary.age
10 mary.birthday()
11 v = dave.name
12 v = dave.age
13
14 print(dave.name)
```

РЕАЛИЗУЕМ ОГРАНИЧЕНИЯ

- Хотим для полей класса указать, какие данные мы там ожидаем
- Например, целое число из заданного диапазона
- Или одна из перечисленных строк
- Реализуем целый набор декрипторов с общим интерфейсом

ПРИМЕР

```
 1 from abc import ABC, abstractmethod
 2
 3 class Validator(ABC):
 4
 5     def __set_name__(self, owner, name):
 6         self.private_name = '_' + name
 7
 8     def __get__(self, obj, objtype=None):
 9         return getattr(obj, self.private_name)
10
11 # ....................................
```

ПРИМЕР

```
# ....................................

    def __set__(self, obj, value):
        self.validate(value)
        setattr(obj, self.private_name, value)

    @abstractmethod
    def validate(self, value):
        pass
```

ONEOF

```
 1 class OneOf(Validator):
 2     def __init__(self, *options):
 3         self.options = set(options)
 4
 5     def validate(self, value):
 6         if value not in self.options:
 7             raise ValueError(
 8     f'Expected {value!r} to be one of {self.options!r}')
 9
10 test = OneOf('green', 'red', 'blue')
11 test.validate('green')
12 test.validate('red')
13 test.validate('blue')
```

NUMBER

```
 1 class Number(Validator):
 2
 3     def __init__(self, minvalue=None, maxvalue=None):
 4         self.minvalue = minvalue
 5         self.maxvalue = maxvalue
 6
 7     def validate(self, value):
 8         if not isinstance(value, (int, float)):
 9             raise TypeError(
10                 f'Expected {value!r} to be an int or float'
11             )
12
13 # ............................
```

NUMBER

```
1 # ............................
2         if self.minvalue is not None and
3            value < self.minvalue:
4             raise ValueError(
5     f'Expected {value!r} to be at least {self.minvalue!r}'
6             )
7 # ............................
```

NUMBER

```
# ............................

        if self.maxvalue is not None and
           value > self.maxvalue:
            raise ValueError(
                f"Expected {value!r} "
                 "to be no more than {self.maxvalue!r}"
            )

test = Number()
test.validate(12345)
test.validate(-12.345)
```

STRING

```
1 class String(Validator):
2
3     def __init__(self, minsize=None, maxsize=None,
4                  predicate=None):
5         self.minsize = minsize
6         self.maxsize = maxsize
7         self.predicate = predicate
8
9 # ................................
```

STRING

```
# ................................

    def validate(self, value):
        if not isinstance(value, str):
            raise TypeError(
                f'Expected {value!r} to be an str'
            )
        if self.minsize is not None and
           len(value) < self.minsize:
            raise ValueError(
                f'Expected {value!r} to be no smaller '
                 'than {self.minsize!r}'
            )
# ................................
```

STRING

```
# ................................

        if self.maxsize is not None and
           len(value) > self.maxsize:
            raise ValueError(
                f'Expected {value!r} to be no bigger '
                 'than {self.maxsize!r}'
            )
        if self.predicate is not None and
           not self.predicate(value):
            raise ValueError(
                f'Expected {self.predicate} to be true '
                 'for {value!r}'
            )
```

STRING

```
 1 class Component:
 2     name = String(minsize=3, maxsize=10,
 3                   predicate=str.isupper)
 4     kind = OneOf('wood', 'metal', 'plastic')
 5     quantity = Number(minvalue=0)
 6
 7     def __init__(self, name, kind, quantity):
 8         self.name = name
 9         self.kind = kind
10         self.quantity = quantity
11
12 c = Component('WIDGET', 'metal', 5)
13
14 print(c)
```

ОСМЫСЛИМ

- Выглядит как крен в сторону классического ООП
- Мы фактически задаем схему класса
- Именно на уровне класса
- Только работает она все равно динамически

ОСМЫСЛИМ

- Это не отказ от гибкости и динамичности
- Это попытка привнести лучшее из мира типизации
- Но "типы" существуют как объекты
- Их можно создавать, переиспользовать, настраивать

ЕЩЕ ОДИН TRADE-OFF

- В примерах в каждом классе создавался новый дескриптор-объект
- Это задавало декларативный стиль
- Но мы могли вы создать объект где-то в другом месте
- А в описании класса - сослаться на него
- Особенно если класс создается в рамках мета-фреймворка как результат работы функции

ЕЩЕ ОДИН TRADE-OFF

- Пример: переиспользование логирующего дескриптора
- Мы могли бы его переиспользовать с пользой
- Например, если хотим логировать доступ полю age в разных классах
- С одинаковыми параметрами логирования
- Которые бы отражались в параметрах конструктора

ЕЩЕ ОДИН TRADE-OFF

- Это бы работало нормально, если вы один экземпляр привязывали к одному и тому же имени
- Но мы можем это проверять в `__set_name__`
- Другой вариант: разрешать привязывать один экземпляр внутри одного класса
- Пусть даже на разные имена
- И хранить знания об этих именах
- Особая аккуратность, если у дескриптора - свое состояние

ВАЖНОЕ ОГРАНИЧЕНИЕ

- В `__set__`/`__get__` не приходит имя, инициировавшее операцию
- Это ограничивает переиспользование одного экземпляра дескриптора внутри одного класса
- Только в случаях, когда поведение не привязано к полю
- Или мы намеренно хотим сделать одно поле псевдонимом другого

LOGGEDACCESS

```
 1 import logging
 2
 3 logging.basicConfig(level=logging.INFO)
 4
 5 class LoggedAccess:
 6     def __init__(self):
 7         self._owner = None
 8
 9     def __set_name__(self, owner, name):
10         if not self._owner:
11             self.public_name = name
12             self.private_name = '_' + name
13             self._owner = owner
14 # .....................................
```

LOGGEDACCESS

```
# ................................

    def __get__(self, obj, objtype=None):
        value = getattr(obj, self.private_name)
        logging.info('Accessing %r giving %r',
                     self.public_name, value)
        return value

    def __set__(self, obj, value):
        logging.info('Updating %r to %r',
                     self.public_name, value)
        setattr(obj, self.private_name, value)
# ................................
```

LOGGEDACCESS

```
 1 # ..................................
 2 class Person:
 3
 4     name = LoggedAccess()
 5     age = LoggedAccess()
 6     summers = age
 7
 8     def __init__(self, name, age):
 9         self.name = name
10         self.age = age
11
12     def birthday(self):
13         self.age += 1
14 # ..................................
```

LOGGEDACCESS

```
1 # ..................................
2 mary = Person('Mary M', 30)
3 dave = Person('David D', 40)
4
5 print(dave.age, mary.summers)
6
7 dave.summers = 35
8 mary.age -= 1
9 print(dave.summers, mary.age)
```

ORM

- ORM - object-relational mapping
- Шлюз между понятиями ООП и реляционной моделью данных
- В реляционной модели однотипные объекты хранятся в таблицах
- Поля таблицы соответствуют свойствам объекта

ORM

- Создание объекта - INSERT-запрос
- Чтение объекта - SELECT
- Изменение свойств - UPDATE
- Это в первом приближении

ORM

- Во втором приближении - нужно межтабличные взаимодействия поддерживать
- Делать JOIN под капотом объектной абстракции
- Обновлять записи как-то оптимальнее, чем одиночными UPDATE/INSERT-ами
- И какое-то итерирование

ORM

- В любом случае - хотим работать с объектами
- И чтобы это отображалось в какую-то вспомогательную логику
- Частью которой являются SQL-запросы
- И здесь помогают декораторы и дескрипторы

СХЕМА РЕАЛИЗАЦИИ

- Нужно знание о том, в какой базе все хранится
- Если это SQL, то какой именно (от этого зависит синтаксис)
- Знание о деталях доступа: хост, порт, пароли (не в исходниках)
- Это может быть класс с аттрибутами уровня класса
- С декоратором вроде `@Repository`

СХЕМА РЕАЛИЗАЦИИ

- Такой декоратор зарегистрирует репозиторий
- И зафиксирует конфигурацию для доступа
- Для классов, которые хотим хранить, будет свой декоратор, вроде `Entity`
- Декларация желания хранить объекты этого класса в репозитории

СХЕМА РЕАЛИЗАЦИИ

- Каждому классу выделяем свою таблицу
- По умолчанию имя таблицы - имя класса
- Но можно указать параметром декоратора
- Как и какие-то опции для создания таблицы

СХЕМА РЕАЛИЗАЦИИ

- Через дескрипторы указываем поля таблицы
- Там же можно указывать ограничения
- Какие поля и как индексировать
- Кто будет первичным ключом

КАК-ТО ТАК

```
class Field:
    def __set_name__(self, owner, name):
        self.fetch = f'SELECT {name} FROM '
                     f'{owner.table} WHERE {owner.key}=?;'
        self.store = f'UPDATE {owner.table} '
                     f'SET {name}=? WHERE {owner.key}=?;'

    def __get__(self, obj, objtype=None):
        return conn.execute(self.fetch,
                            [obj.key]).fetchone()[0]

    def __set__(self, obj, value):
        conn.execute(self.store, [value, obj.key])
        conn.commit()
```

КАК-ТО ТАК

```
 1 class Movie:
 2     table = 'Movies'          # Table name
 3     key = 'title'             # Primary key
 4     director = Field()
 5     year = Field()
 6
 7     def __init__(self, key):
 8         self.key = key
 9
10 # ........................
```

КАК-ТО ТАК

```
# ........................

class Song:
    table = 'Music'
    key = 'title'
    artist = Field()
    year = Field()
    genre = Field()

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

PROPERTY

- `property` реально реализована на C
- Можно реконструировать его логику через Python-класс
- И использовать механизм дескрипторов

PROPERTY

```
 1 class Property:
 2     def __init__(self, fget=None, fset=None,
 3                  fdel=None, doc=None):
 4         self.fget = fget
 5         self.fset = fset
 6         self.fdel = fdel
 7         if doc is None and fget is not None:
 8             doc = fget.__doc__
 9         self.__doc__ = doc
10         self._name = ''
11
12     def __set_name__(self, owner, name):
13         self._name = name
14 # ..................
```

PROPERTY

```
# ..................
    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError(
        f'property {self._name!r} '
        f'of {type(obj).__name__!r} object has no getter'
             )
        return self.fget(obj)
# ..................
```

PROPERTY

```
# ..................
    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError(
        f'property {self._name!r} of '
        f'{type(obj).__name__!r} object has no setter'
             )
        self.fset(obj, value)
# ..................
```

PROPERTY

```
# ..................
    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError(
        f'property {self._name!r} of '
        f'{type(obj).__name__!r} object has no deleter'
             )
        self.fdel(obj)
```

PROPERTY

```
 1     def getter(self, fget):
 2         prop = type(self)(fget, self.fset,
 3                           self.fdel, self.__doc__)
 4         prop._name = self._name
 5         return prop
 6
 7     def setter(self, fset):
 8         prop = type(self)(self.fget, fset,
 9                           self.fdel, self.__doc__)
10         prop._name = self._name
11         return prop
12
13 #...................
```

PROPERTY

```
1 #..................
2
3     def deleter(self, fdel):
4         prop = type(self)(self.fget, self.fset,
5                           fdel, self.__doc__)
6         prop._name = self._name
7         return prop
```