Лекция 9
Менеджеры контекстов
Продвинутая работа с атрибутами

13 апреля 2017 г.

Менеджеры контекстов

Оператор with

```
with open('some-file.txt') as f:
    ...
```

- При выходе из контекста файл закрывается
- В т.ч. при исключениях

Оператор with

```
with open('some-file.txt') as f:
    ...
```

- При выходе из контекста файл закрывается
- В т.ч. при исключениях

`open` возвращает специальный объект — *context manager*.

Пример менеджера контекста

```
class Context(object):
    def __enter__(self):
        print '__enter__()'
    def __exit__(self, <...>):
        print '__exit__()'

with Context():
    print "Inside context."
```

```
__enter__()
Inside context.
__exit__()
```

Переменная контекста

```
class Context(object):
    def __enter__(self):
        print '__enter__()'
        return 'Some data'
    ...

with Context() as x:
    print "Inside context, x =", x
```

```
__enter__()
Inside context, x = Some data
__exit__()
```

Инициализация контекста

```
class Context(object):
    def __init__(self):
        print '__init__()'
    ...
```

Инициализация контекста

```
class Context(object):
    def __init__(self):
        print '__init__()'
    ...

with Context() as x:
    print "Inside context, x =", x
```

```
__init__()
__enter__()
Inside context, x = Some data
__exit__()
```

Исключения и контексты

```
class Context(object):
    ...
    def __exit__(self, exc_type, exc_val,
                 exc_tb):
        print '__exit__({}, {})'.format(
            exc_type.__name__, exc_val)


with Context() as x:
    print "Inside context, x =", x
    raise RuntimeError("Smth is wrong")
```

Исключения и контексты

```
with Context() as x:
    print "Inside context, x =", x
    raise RuntimeError("Smth is wrong")
```

```
__init__()
__enter__()
Inside context, x = Some data
__exit__(RuntimeError, Smth is wrong)

Traceback (most recent call last):
  <...>
RuntimeError: Smth is wrong
```

Пример: остановка исключений

```
class Context(object):
    def __init__(self, stop_exception):
        print '__init__()'
        self.stop_exception = stop_exception
    def __enter__(self):
        print '__enter__()'
        return 'Some data'
    def __exit__(self, exc_type, exc_val,
                 exc_tb):
        print '__exit__({}, {})'.format(
            exc_type.__name__, exc_val)
        return self.stop_exception
```

Пример: остановка исключений

```
with Context(False) as x:
    print "Inside context, x =", x
    raise RuntimeError("Smth is wrong")
```

```
__init__()
__enter__()
Inside context, x = Some data
__exit__(RuntimeError, Smth is wrong)

<...>
RuntimeError: Smth is wrong
```

Остановка исключений

```
with Context(True) as x:
    print "Inside context, x =", x
    raise RuntimeError("Smth is wrong")
```

```
__enter__()
Inside context, x = Some data
__exit__(RuntimeError, Smth is wrong)
```

Контекст без обработки исключений

```
class Context(object):
    def __exit__(self, exc_type,
                 exc_val, exc_tb):
        print '__exit__({}, {})'.format(
            exc_type, exc_val)

with Context(True) as x:
    print "Inside context, x =", x
```

```
__init__()
__enter__()
Inside context, x = Some data
__exit__(None, None)
```

Пример: временный файл

- Существует только внутри контекста
- Можно читать и писать

(есть стандартный модуль `tempfile`)

Использование временного файла

```
with TmpFile('some-dir/') as path:
    with open(path, 'w') as fout:
        fout.write('Some stuff.')
        ...
    ...
    with open(path) as fin:
        ... = fin.read()
```

Временный файл

```
class TmpFile(object):
    def __init__(self, folder):
        self.folder = folder
    def __enter__(self):
        name = generate_name()
        self.path = os.path.join(self.folder,
                                 name)
        # create empty file
        with open(name, 'w'): pass

        return self.path
    def __exit__(self, exc_type,
                 exc_value, exc_tb):
        os.remove(self.path)
```

Операторы доступа к атрибутам

Зачем нужны операторы доступа

Операторы доступа к атрибутам

```
a.x  # read
a.y = ...  # write
```

Примеры использования

- Не нужно всегда писать функции чтения и записи для всех атрибутов
  (если атрибут нужно будет превратить в метод, он все равно сможет «притворяться» атрибутом)
- Псевдо-атрибуты
  (для которых данные не хранятся, но вычисляются)

Оператор `__getattr__`

`__getattr__(self, name)`

- Срабатывает при чтении атрибута (`a.x`)
- Не вызывается, если атрибут существует
- Получает на вход имя атрибута

Оператор `__getattr__`

`__getattr__(self, name)`

- Срабатывает при чтении атрибута (`a.x`)
- Не вызывается, если атрибут существует
- Получает на вход имя атрибута
- Должен вернуть его значение
  или сгенерировать `AttributeError`

Оператор `__getattr__`

`__getattr__(self, name)`

- Срабатывает при чтении атрибута (`a.x`)
- Не вызывается, если атрибут существует
- Получает на вход имя атрибута
- Должен вернуть его значение
  или сгенерировать `AttributeError`
- Может использовать внутри другие атрибуты
- (Методы - тоже атрибуты)

Оператор __getattr__

```
class A(object):
    def __init__(self):
        self.x = 1
    def __getattr__(self, name):
        return self.x + 1

a = A()
print a.x
print a.y, a.z, a.qwerty
```

```
1
2 2 2
```

Эмуляция отсутствия атрибута

```
class A(object):
    def __init__(self):
        self.base = 1
    def __getattr__(self, name):
        if name == 'doubled':
            return self.base * 2
        else:
            raise AttributeError(
                "Attribute not found.")
```

Эмуляция отсутствия атрибута

```
a = A()
print a.base, a.doubled
```

```
1 2
```

```
print a.other
```

```
AttributeError: Attribute not found.
```

Оператор `__setattr__`

`__setattr__(self, name, value)`

- Вызывается при изменении атрибута (`a.x = ...`)
- Если определен, то вызывается всегда
- Получает на вход имя атрибута и значение

Оператор `__setattr__`

`__setattr__(self, name, value)`

- Вызывается при изменении атрибута (`a.x = ...`)
- Если определен, то вызывается всегда
- Получает на вход имя атрибута и значение
- Может изменять другие атрибуты через `super(C, self).__setattr__()`

Оператор __setattr__

```
class A(object):
    def __init__(self):
        self.x = 1
    def __setattr__(self, name, value):
        super(A, self).__setattr__(
            'x', value)
```

Оператор __setattr__

```
a = A()
print a.x
a.y = 2
print a.x
```

```
1
2
```

```
print a.y
```

```
AttributeError: 'A' object has no
attribute 'y'
```

Оператор `__getattribute__`

`__getattribute__(self, name)`

- Срабатывает при чтении атрибута (`a.x`)
- Если определен, то вызывается всегда
- Получает на вход имя атрибута
- Должен вернуть его значение
  или создать `AttributeError`

Оператор `__getattribute__`

`__getattribute__(self, name)`

- Срабатывает при чтении атрибута (`a.x`)
- Если определен, то вызывается всегда
- Получает на вход имя атрибута
- Должен вернуть его значение
  или создать `AttributeError`
- Может получать другие атрибуты через
  `super(C, self).__getattribute__()`

Оператор `__getattribute__`

`__getattribute__(self, name)`

- Срабатывает при чтении атрибута (`a.x`)
- Если определен, то вызывается всегда
- Получает на вход имя атрибута
- Должен вернуть его значение
  или создать `AttributeError`
- Может получать другие атрибуты через
  `super(C, self).__getattribute__()`

- Исключение — методы `__*__`
- (Более сложный вариант `__getattr__`)

Оператор __getattribute__

```
class A(object):
    def __init__(self):
        self.x = 1
    def __getattribute__(self, name):
        return (super(A, self).
            __getattribute__('x') + 1)

a = A()
print a.x
print a.y
```

Оператор __getattribute__

```
class A(object):
    def __init__(self):
        self.x = 1
    def __getattribute__(self, name):
        return (super(A, self).
            __getattribute__('x') + 1)

a = A()
print a.x
print a.y
```

```
2
2
```

Доступ к атрибутам

Оператор	**Код**	**Вызывается**
`__getattr__`	`a.x`	Если не сработал обычный механизм
`__getattribute__`	`a.x`	Всегда
`__setattr__`	`a.x =`	Всегда

Доступ по имени

Работа с атрибутами, имя которых хранится как строка.

```
c = SomeClass(...)

getattr(c, 'someattr')
# equivalent to
c.someattr

setattr(c, 'someattr', 1234)
# equivalent to
c.someattr = 1234

# is attribute present? (uses getattr)
hasattr(c, 'someattr')
```

«Модификаторы» доступа

Типы атрибутов

Тип	**Доступен снаружи**	**Доступен из наследников**
Открытый	Да	Да
Защищенный	Нет	Да
Закрытый	Нет	Нет

Модификаторы доступа в Python

Всегда есть доступ до всех атрибутов.

Специальные обозначения для закрытых и защищенных атрибутов.

- Нет «защиты от дурака»
- Есть защита от ошибок
- Полезно в некоторых программах (отладка, текстовые редакторы)

Защищенные атрибуты

Обозначаются префиксом _

```
class A(object):
    def __init__(self):
        self.normal_attribute = 0
        self._protected_attribute = 0
```

Защищенные атрибуты

Обозначаются префиксом _

```
class A(object):
    def __init__(self):
        self.normal_attribute = 0
        self._protected_attribute = 0
```

Доступ изнутри

- По имени (так же, как для открытых атрибутов)

Доступ извне

- По имени (так же, как для открытых атрибутов)
- Легко заметен в коде
- Обнаруживается редакторами и чекерами

Закрытые атрибуты

Обозначаются префиксом `__`

```
class A(object):
    def __init__(self):
        self.normal_attribute = 0
        self._protected_attribute = 0
        self.__private_attribute = 0

    def __private_method(self):
        ...
```

Закрытые атрибуты

Обозначаются префиксом `__`

```
class A(object):
    def __init__(self):
        self.normal_attribute = 0
        self._protected_attribute = 0
        self.__private_attribute = 0

    def __private_method(self):
        ...
```

Проблема: как получить доступ, если в классе-наследнике есть такое же имя?

Декорирование имен

```
class C(object):
    def public_method(self):
        pass
    def __private_method(self):
        pass
```

Декорирование имен

```
class C(object):
    def public_method(self):
        pass
    def __private_method(self):
        pass

print dir(C)
```

```
['_C__private_method',
 ...,
 '__doc__', '__init__',
 ...,
 'public_method']
```

Закрытые атрибуты

```
class CLASSNAME(object):
    __ATTRIBUTENAME = 0
```

Доступ изнутри

- `__ATTRIBUTENAME`

Доступ извне

- `_CLASSNAME__ATTRIBUTENAME`
- Легко заметен в коде
- Обнаруживается редакторами и чекерами

Обзор обозначений

Обозначение	**Значение**
`name`	Открытый атрибут
`_name`	Защищенный атрибут
`__name`	Закрытый атрибут (внутри)
`_class__name`	Закрытый атрибут (снаружи)
`__name__`	Зарезервированный атрибут

Оформление атрибутов

Простое оформление атрибутов

```
class Parrot(object):
    def __init__(self):
        self.voltage = 1000
```

Простое оформление атрибутов

```
class Parrot(object):
    def __init__(self):
        self.voltage = 1000

p = Parrot()
p.voltage = 'qwerty'
```

Простое оформление атрибутов

```
class Parrot(object):
    def __init__(self):
        self.voltage = 1000

p = Parrot()
p.voltage = 'qwerty'
```

(–) Класс не узнает, что его атрибут изменили.

(–) Пользователь не узнает, что ошибся.

Простое оформление атрибутов

```
class Parrot(object):
    def __init__(self):
        self.voltage = 1000

p = Parrot()
p.voltage = 'qwerty'
```

(−) Класс не узнает, что его атрибут изменили.

(−) Пользователь не узнает, что ошибся.

(+) Быстро писать.

Сложное оформление атрибутов

```
class Parrot(object):
    def __init__(self):
        self.__voltage = 1000

    def voltage(self):
        return self.__voltage
```

Сложное оформление атрибутов

```
class Parrot(object):
    def __init__(self):
        self.__voltage = 1000

    def voltage(self):
        return self.__voltage
```

(+) Атрибут защищен

(−) Нельзя изменять

(−) `voltage()` вместо `voltage`

Переопределение оператора «.»

```
class Parrot(object):
    def __init__(self):
        self.__voltage = 1000
```

Переопределение оператора «.»

```
class Parrot(object):
    def __init__(self):
        self.__voltage = 1000

    def __getattr__(self, name):
        if name == 'voltage':
            return self.__voltage
        raise AttributeError(
            name + ' not found')
```

Переопределение оператора «.»

```
class Parrot(object):
    def __init__(self):
        self.__voltage = 1000

    def __getattr__(self, name):
        if name == 'voltage':
            return self.__voltage
        raise AttributeError(
            name + ' not found')

    def __setattr__(self, name, value):
        if name == 'voltage':
            assert isinstance(value, int)
            self.__voltage = value
        else:
            super(Parrot, self).__setattr__(
                name, value)
```

Декоратор property

```
class Parrot(object):
    def __init__(self):
        self.__voltage = 1000
```

Декоратор property

```
class Parrot(object):
    def __init__(self):
        self.__voltage = 1000

    def __get_voltage(self):
        return self.__voltage


    def __set_voltage(self, value):
        assert isinstance(value, int)
        self.__voltage = value
```

Декоратор property

```
class Parrot(object):
    def __init__(self):
        self.__voltage = 1000

    def __get_voltage(self):
        return self.__voltage


    def __set_voltage(self, value):
        assert isinstance(value, int)
        self.__voltage = value

    voltage = property(
        __get_voltage, __set_voltage)
```

Декоратор property

```
class Parrot(object):
    def __init__(self):
        self.__voltage = 1000

    def __get_voltage(self):
        return self.__voltage


    def __set_voltage(self, value):
        assert isinstance(value, int)
        self.__voltage = value

    voltage = property(
        __get_voltage, __set_voltage)
```

Также можно передать: функцию удаления, документацию.

Декоратор property

```
class Parrot(object):
    def __init__(self):
        self.__voltage = 1000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self.__voltage
```

Декоратор property

```
class Parrot(object):
    def __init__(self):
        self.__voltage = 1000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self.__voltage

    @voltage.setter
    def voltage(self, value):
        assert isinstance(value, int)
        self.__voltage = value
```

Декоратор property

```
class Parrot(object):
    def __init__(self):
        self.__voltage = 1000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self.__voltage

    @voltage.setter
    def voltage(self, value):
        assert isinstance(value, int)
        self.__voltage = value
```

(Только new-style классы)

Порядок разрешения имен

```
c = C()
c.foo
```

- Поискать атрибут через дескрипторы в `C`
- Поискать атрибут через дескрипторы в родителях `C`
- Поискать атрибут в `c.__dict__`
- Поискать атрибут в `C.__dict__`
- Поискать атрибут в классах-родителях `C`
- `raise AttributeError`

(дескрипторы создаются декоратором `property`)