

ЭТОТ classный ПИТОН

Артур Чеканов, DataArt

О себе

- Senior Python Developer в ДатаАрт
- Background:
 - Web/Game/Desktop development
 - C/C++
 - .NET
 - Javascript
 - Python
- Собеседую людей по Python и Frontend разработке

ООП

- Парадигма программирования
- Объект - сущность, которой можно посылать сообщения и которая может на них реагировать используя свои данные (wikipedia)
- Наследование, инкапсуляция, полиморфизм
- Объект содержит данные и функции работающие с ними

Классы

- Old style class

```
class SomeClass:  
    pass
```

- New style class

```
class SomeClass(object):  
    pass
```

New-style

- Descriptors
- Другое MRO
- type
- Python 3

Классы в Python

- Наследование: множественное, `mixin`
- Полиморфизм: `duck-typing`
- Инкапсуляция: `name-mangling`

Классы декларация

- Наследуется от object
- Инициализация переменных в `__init__`
- Каждый метод должен иметь `self` в качестве первого аргумента
 - Ну почти каждый

```
class SomeClass(object):  
  
    def __init__(self):  
        self.var = 1  
  
    def func(self):  
        print self.var
```

Наследование

- Не забываем super

```
class Vector2D(object):  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
class Vector3D(Vector2D):  
    def __init__(self, x, y, z):  
        super(Vector3D, self).__init__(x, y)  
        self.z = z
```


Наследование. Mixin

- Класс, но со смысловыми нюансами
- Не имеет смысла сам по себе
- Может быть наследован любым классом

```
class ConsoleMixin(object):  
  
    def print_to_console(self, message):  
        print message  
  
    def read_from_console(self, message=''):  
        return raw_input(message)  
  
    def confirm(self, message):  
        self.print_to_console(message)  
        while True:  
            answer = self.read_from_console('yes/no: ').lower()  
            if answer == 'yes':  
                return True  
            if answer == 'no':  
                return False
```

Наследование. Mixin

- Программа вычисления числа пи

```
class ProgramPi(ConsoleMixin):

    def run(self):
        self.print_to_console('Calculating PI')

        pi = 0
        denominator = 1

        step = 0
        sign = 1
        while True:
            pi += sign * (4.0 / denominator)
            denominator += 2
            sign *= -1

            step += 1
            if step % 10000 == 0:
                self.print_to_console('PI is %s' % pi)
                self.print_to_console('On iteration %s' % step)
                if not self.confirm('Would you like to continue?'):
                    break
```

Полиморфизм. Duck-typing

- Если нечто выглядит как утка, плавает как утка и крякает как утка, то это, наверное, и есть утка

```
def run_program(program):  
    try:  
        program.run()  
    except AttributeError:  
        print 'Not a program'  
  
def run_program(program):  
    if hasattr(program, 'run'):  
        program.run()  
    else:  
        print 'Not a program'
```

Инкапсуляция. Name mangling

```
class Node3D(object):
    def __init__(self, x, y):
        self.transformation = [x, y]

    def update(self):
        print 'Object is in %s' % self.transformation

class Player(Node3D):
    pass

class Werewolf(Player):
    def __init__(self, x, y):
        super(Werewolf, self).__init__(x, y)
        self.transformation = False

    def begin_transform(self):
        self.transformation = True

w = Werewolf(1, 2)
w.update() # Object is in False
```

Инкапсуляция. Name mangling

```
class Node3D(object):
    def __init__(self, x, y):
        self.__transformation = [x, y]

    def update(self):
        print 'Object is in %s' % self.__transformation

class Player(Node3D):
    pass

class Werewolf(Player):
    def __init__(self, x, y):
        super(Werewolf, self).__init__(x, y)
        self.transformation = False

    def begin_transform(self):
        self.transformation = True

w = Werewolf(1, 2)
w.update() # Object is in [1, 2]
```

Инкапсуляция. Name mangling

- Не так то просто обратиться извне
- Однако возможно

```
w = Werewolf(1, 2)
w.__transformation # AttributeError
w._Node3D__transformation # [1, 2]
```

Декораторы

Декораторы. Зачем

- Предположим нужно измерить производительность
- А если функций много?

```
from datetime import datetime

def some_hard_calculations():
    before = datetime.now()
    pass
    dt = datetime.now() - before
    logger.info('Cool calculations completed in %s second(s)', dt.total_seconds())
```


Декораторы. Зачем

- Предположим нужно измерить производительность
- А если функций много?
- Проще написать один раз и использовать где нужно

```
@measure  
def some_hard_calculations():  
    pass
```

Декораторы. Как?

- Декоратор - синтаксический сахар

```
@measure  
def some_hard_calculations():  
    pass
```

```
def some_hard_calculations():  
    pass  
some_hard_calculations = measure(some_hard_calculations)
```

Декораторы. Как?

- Декоратор должен принимать функцию в качестве аргумента
- Декоратор должен отдавать функцию
- IDE постарается поставить скобки в return, не поддавайтесь!

```
def measure(func):  
    def wrapper(*args, **kwargs):  
        before = datetime.now()  
        func(*args, **kwargs)  
        dt = datetime.now() - before  
        logger.info('Cool calculations completed in %s second(s)', dt.total_seconds())  
    return wrapper  
  
@measure  
def some_hard_calculations():  
    pass
```

Декораторы. Как?

- А что если мы хотим немного персонализировать декоратор?
- Нет проблем: создадим функцию, которая вернет декорирующую функцию

```
def measure(name):
    def create_wrapper(func):
        def wrapper(*args, **kwargs):
            before = datetime.now()
            func(*args, **kwargs)
            dt = datetime.now() - before
            logger.info('%s: Cool calculations completed in %s second(s)', name, dt.total_seconds())
        return wrapper
    return create_wrapper

@measure('Very Hard Calculations')
def some_hard_calculations():
    pass
```

Декораторы. Как?

- А вы знали что класс может вести себя как функция?

```
class MyFunc(object):  
    def __call__(self):  
        print 'Hello World!'  
  
my_func = MyFunc()  
my_func() # Hello World!
```

Декораторы. Как?

- Декоратором может быть класс

```
class measure(object):

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

    def __call__(self, func):
        def wrapper(*args, **kwargs):
            before = datetime.now()
            func(*args, **kwargs)
            self.post_measure(datetime.now() - before)
        return wrapper

    def post_measure(self, dt):
        logger.info('%s: Cool calculations completed in %s second(s)', self.name, dt.total_seconds())

@measure('Very Hard Calculations')
def some_hard_calculations():
    pass
```

Декораторы. Как?

- Однако есть проблемы

```
def some_hard_calculations():
    """
    This is extremely long running procedure, use carefully!
    """
    pass

print some_hard_calculations.__doc__ # as expected

@measure('profile')
def some_hard_calculations():
    """
    This is extremely long running procedure, use carefully!
    """
    pass

print some_hard_calculations.__doc__ # None (!!!)
```

Декораторы. Как?

- Их легко исправить

```
from functools import wraps

class measure(object):

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

    def __call__(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            before = datetime.now()
            func(*args, **kwargs)
            self.post_measure(datetime.now() - before)
        return wrapper

    def post_measure(self, dt):
        logger.info('%s: Cool calculations completed in %s second(s)', self.name, dt.total_seconds())

@measure('profile')
def some_hard_calculations():
    """
    This is extremely long running procedure, use carefully!
    """
    pass

print some_hard_calculations.__doc__ # as expected
```


Декораторы.

- Их может быть несколько

```
def log_execution(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        logger.debug('Executing function %s', func.__name__)
        func(*args, **kwargs)
    return wrapper

@measure('profile')
@log_execution
def some_hard_calculations():
    """
    This is extremely long running procedure, use carefully!
    """
    pass
```

Метаклассы

Метаклассы

- Класс - это объект, его можно присвоить, передать

```
class Message(object):  
    def message(self):  
        return 'Hello World'  
  
AnotherMessage = Message  
print AnotherMessage().message()
```

- Класс тоже можно создать

```
Message = type('Message', (), {  
    'message': lambda self: 'Hello World'  
})  
  
print Message().message()
```

Метаклассы.

- Класс можно создать
- Класс можно поменять при создании
- Для этого нужно указать метакласс

Метаклассы. Для чего?

- Они редко используются
- Можно убрать методы
- Можно их добавить
- Можно валидировать
- Можно программировать декларативно
 - Django ORM
 - Django Forms

Метаклассы. Пример

- Что если мы хотим валидировать наличие методов?
- Вспомним duck-typing и mixin
- Попробуем сделать так чтобы у класса Vector были обязательные

```
class Structure(object):  
    def __init__(self, **kwargs):  
        for field, value in kwargs.items():  
            setattr(self, field, value)
```

```
class Vector(Structure):  
    pass
```

Метаклассы. Пример

- Нужен базовый класс валидатора и сам валидатор

```
class ClassMethodsValidator(object):  
    REQUIRED_METHODS = ()  
  
class VectorValidator(ClassMethodsValidator):  
    REQUIRED_METHODS = ('dot', 'length')
```

Метаклассы. Пример

- Нужен валидирующий метакласс

```
class ClassMethodsValidatorMetaclass(type):
    def __new__(cls, name, bases, args):
        for base in bases:
            if not isinstance(base, ClassMethodsValidator):
                continue
            for required in base.REQUIRED_METHODS:
                if required not in args:
                    raise TypeError('The method "%s" is required' % required)

        return type.__new__(cls, name, bases, args)
```


Метаклассы. Пример

- А теперь все вместе

```
class Structure(object):
    def __init__(self, **kwargs):
        for field, value in kwargs.items():
            setattr(self, field, value)

    __metaclass__ = ClassMethodsValidatorMetaclass

class VectorValidator(ClassMethodsValidator):
    REQUIRED_METHODS = ('dot', 'length')

class Vector(Structure, VectorValidator):
    def dot(self, v):
        return self.x * v.x + self.y * v.y

    def length(self):
        return math.sqrt(self.dot(self))
```

Дескрипторы

Дескрипторы

- Объект, определяющий доступ к атрибуту
- Любой объект определяющий хотя бы один из методов
 - `__get__`
 - `__set__`
 - `__delete__`
- Что происходит при вызове `obj.attr`
 - `obj.__dict__['attr']`
 - `type(obj).__dict__['attr']`
 - Поиск по базовым классам
- Что будет если в цепочке вверху попадет дескриптор?
 - Будет вызван один из его методов
- Только `new-style` классы

Дескрипторы

- Напишем простой дескриптор который будет делать ничего

```
class value(object):  
  
    def __init__(self, default):  
        self.value = default  
  
    def __get__(self, instance, owner):  
        return self.value  
  
    def __set__(self, instance, value):  
        self.value = value  
  
class Vector2Di(object):  
    x = value(1)  
    y = value(1)
```

Дескрипторы

- И добавим немного type-check

```
class field(object):

    def __init__(self, default, type):
        self.value = default
        self.type = type

    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        if not isinstance(value, self.type):
            raise AttributeError('%s is not of type %s' % (value, self.type.__name__))
        self.value = value

class Vector2Di(object):
    x = field(1, int)
    y = field(1, int)

a = Vector2Di()

a.x = 0.2 # AttributeError
```

Вопросы?