Python

Лекция 3

Преподаватель: *Дмитрий Косицин*
BSU FAMCS (Fall'19)

Декораторы

Декораторы

Декораторы (PEP-318):

- Выполняют некоторое дополнительное действие при вызове или создании функции
- Модифицируют функцию после создания
- Могут принимать аргументы
- Упрощают написание кода

Рассмотрим пример декоратора – функции, которая при вызове декорированной функции проверяет, возвращенное ей значение имеет тип *float*. Функцию-декоратор назовем *check_return_type_float*.

Пример реализации

Пример использования (проверяет, что возвращаемое значение типа *float*):

```
>>> @check_return_type_float
>>> def g():
>>>     return 'not a float'
```

Эквивалентной записью будет следующая:

```
>>> def g():
>>>     return 'not a float'
>>>
>>> g = check_return_type_float(g)
```

Пример реализации

Декоратор реализован как функция, которая возвращает другую функцию – *wrapper*:

```
>>> def check_return_type_float(f):
>>>     def wrapper(*args, **kwargs):
>>>         result = f(*args, **kwargs)
>>>         assert isinstance(result, float)
>>>         return result
>>>     return wrapper
```

Нюансы декорирования

Поскольку декоратор возвращает другую функцию, в примере *check_return_type_float* у переменной **f** будет имя '*wrapper*'.

Для того, чтобы метаданные (имя, документация) были корректными, внутренней функции (wrapper'у) добавляют декоратор **functools.wraps**:

```
>>> def check_return_type_float(f):
>>>     @functools.wraps(f)
>>>     def wrapper(*args, **kwargs):
>>>         # some code
>>>     return wrapper
```

Замечание. В Python 3 декорировать можно не только функции, но и классы (PEP-3129).

Реализация декоратора с помощью класса

```
>>> class FloatTypeChecker(object):
>>>     def __init__(self, f):
>>>         self.f = f
>>>
>>>     def __call__(self, *args, **kwargs):
>>>         result = self.f(*args, **kwargs)
>>>         assert isinstance(result, float)
>>>         return result
>>>
>>> check_return_type_float = FloatTypeChecker
```

Вопрос: Как применить здесь **functools.wraps**?

Замечание. Добавлять данный декоратор желательно везде. Это и хороший стиль кода, и так остается возможность узнать имя вызванной функции run-time.

Декораторы с параметрами

Для создания более общего декоратора, логично ему добавить возможность принимать параметры.

```
>>> def check_return_type(type_):
>>>     def wrapper(f):
>>>         @functools.wraps(f)
>>>         def wrapped(*args, **kwargs):
>>>             result = f(*args, **kwargs)
>>>             assert isinstance(result, type_)
>>>             return result
>>>         return wrapped
>>>     return wrapper
>>>
>>> @check_return_type(float)
>>> def g():
>>>    return 'not a float'
```

Несколько декораторов

Эквивалентной записью будет следующая:

```
>>> def g():
>>>     return 'not a float'
>>>
>>> g = check_return_type(float)(g)
```

Допустимо применять несколько декораторов – один над другим.

```
>>> @decorator2
>>> @decorator1
>>> def f():
>>>     pass
```

Эквивалентная запись применения декораторов к функции **f** имеет вид:

```
>>> f = decorator2(decorator1(f))
```

Параметризованный декоратор-класс

Декораторы с параметрами можно реализовать как класс. В таком случае параметры будут сохраняться в методе ***__init__***, а декорированную функцию следует возвращать в ***__call__***.

```
>>> class FloatTypeChecker(object):
>>>     def __init__(self, result_type):
>>>         self._result_type = result_type
>>>
>>>     def __call__(self, f):
>>>         @functools.wraps(f)
>>>         def wrapper(*args, **kwargs):
>>>             result = f(*args, **kwargs)
>>>             assert isinstance(result, self._result_type)
>>>             return result
>>>         return wrapper
```

Свойства (пример)

```
class Animal(object):
    def __init__(self, age=0):
        self._age = age

    @property
    def age(self):
        """age of animal"""
        return self._age

    @age.setter
    def age(self, age):
        assert age >= self._age
        self._age = age
```

Свойства

Свойства – это *дескрипторы*, которые можно создать, декорируя методы с помощью **property** (docstring свойства получается из getter'а):

- getter – @property
- setter – @<name>.setter
- deleter – @<name>.delete

Полный синтаксис декоратора-дескриптора **property** имеет вид:

```
age = property(fget, fset, fdelete, doc)
```

Замечание. Создать *write-only* свойство можно только явно вызвав **property** с параметром *fget* равным **None**.

Менеджеры контекста

...

Менеджеры контекста

В процессе работы с файлами важно корректно работать с исключениями: файл необходимо закрыть в любом случае.

Данный синтаксис позволяет закрыть файл по выходе из блока **with**:

```
with open(file_name) as f:
    # some actions
```

Функция **open** возвращает специальный объект – *context manager*. Менеджер контекста последовательно *инициализирует* контекст, *входит* в него и корректно обрабатывает *выход*.

Пример менеджера контекста

```
class ContextManager(object):
    def __init__(self):
        print('__init__()')

    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))


with ContextManager() as c:
    print('inside context "%s"' % c)
```

Менеджер контекста

Менеджер контекста работает следующим образом:

- создается и инициализируется (метод **__init__**)
- организуется вход в контекст (метод **__enter__**) и возвращается объект контекста (в примере с файлом – объект типа **file**)
- выполняются действия внутри контекста (внутри блока **with**)
- организуется выход из контекста с возможной обработкой исключений (метод **__exit__**)

В примере будет выведено следующее:

```
__init__()
__enter__()
inside context "some data"
__exit__(None, None)
```

Менеджер контекста

Замечание. Если исключения не произошло, то параметры, передаваемые в функцию **__exit__** – тип, значение исключения и ***traceback*** – имеют значения **None**.

Замечание. Менеджер контекста, реализуемый функцией **open**, по выходе из контекста просто вызывает метод ***close*** (см. декоратор *contextlib.closing*).

Менеджеры контекста используются:

- для корректной, более простой и переносимой обработки исключений в некотором блоке кода
- Для управления ресурсами

Декоратор *contextlib.contextmanager* позволяет создать менеджер контекста из функции-генератора, что значительно упрощает синтаксис.

Менеджер контекста из генератора

```
>>> @contextlib.contextmanager
>>> def get_context():
>>>     print('__enter__()’)
>>>     try:
>>>         yield 'some data’
>>>     finally:
>>>         print('__exit__()’)
>>>
>>> with get_context() as c:
>>>     print('inside context "%s"' % c)
__enter__()
inside context "some data"
__exit__()
```

Итераторы и генераторы

Последовательности. Итерируемые объекты. Итераторы. Генераторы. Дополнительные способы итерирования.

Sequence and iterable

Последовательность (*sequence*) – упорядоченный *индексируемый* набор объектов, например, **list**, **tuple** и **str**.

У этих объектов переопределены «магические методы» ***__len__*** (длина последовательности) и ***__getitem__*** (отвечает за индексацию).

Итерируемое (*iterable*) – упорядоченный набор объектов, элементы которого можно получать по одному.

У таких объектов реализован метод ***__iter__*** – возвращает итератор, который позволяет обойти итерируемый объект.

Итераторы

Итератор (iterator) представляет собой «поток данных» – он позволяет обойти все элементы *итерируемого* объекта, возвращая их в некоторой последовательности.

В итераторе переопределен метод ***__next__*** (***next*** в Python 2), вызов которого либо возвращает следующий объект, либо бросает исключение ***StopIteration***, если все объекты закончились.

Для явного получения итератора и взятия следующего элемента используются *built-in* методы **iter** и **next**.

```
for item in sequence:
    action(item)
```

```
def for_sequence(sequence, action):  # "for" for sequence
    i, length = 0, len(sequence)
    while i < length:
        item = sequence[i]
        action(item)
        i += 1
```

```
def for_iterable(iterable, action):  # "for" for iterator
    iterator = iter(iterable)
    try:
        while True:
            item = next(iterator)
            action(item)
    except StopIteration:
        pass
```

Итераторы

Итераторы представляют собой классы, содержащие информацию о текущем состоянии итерирования по объекту (например, индекс).

После обхода всех элементов итератор «истощается» (exhausted), бросая исключение ***StopIteration*** при каждом следующем вызове ***__next__***.

Замечание. Функция ***next*** имеет второй параметр – значение по умолчанию, которое будет возвращено, когда итератор исчерпается.

Замечание. У функции ***iter*** также есть второй аргумент – значение, до *получения* которого будет продолжаться итерирование.

Пример реализации итератора

```
class RangeIterator(collections.Iterator):
    def __init__(self, start, stop=None, step=1):
        self._start = start if stop is not None else 0
        self._stop = stop if stop is not None else start
        self._step = step  # positive only

        self._current = self._start

    def __next__(self):
        if self._current >= self._stop:
            raise StopIteration()

        result = self._current
        self._current += self._step
        return result
```

Пример использования итератора

Поскольку итераторы хранят информацию о состоянии, их можно прервать и впоследствии продолжить итерироваться. *Вопрос: что выведет следующий код?*

```
>>> odd_indices_iterator = RangeIterator(1, 10, 2)
>>>
>>> for idx in odd_indices_iterator:
>>>     if idx > 5:
>>>         break
>>>     print(idx)
>>>
>>> for idx in odd_indices_iterator:
>>>     print(idx)
```

Итерируемые и истощаемые

Последовательности *итерируемы* и *не истощаемы* (можно много раз итерироваться по ним).

Итерируемые объекты (не последовательности) могут как не истощаться (**range** в Py3/**xrange** в Py2), так и истощаться (генераторы).

Итераторы *итерируемы* (возвращают сами себя) и *истощаемы* (можно только один раз обойти).

Замечание. Зачастую в классах не реализуют отдельный класс-итератор. В таком случае метод ***__iter__*** возвращает *генератор*.

Замечание. В Python 2 **range** возвращает список, а **xrange** – генератор.

Пример итерируемого объекта

```
class SomeSequence(collections.Iterable):
    def __init__(self, *items):
        self._items = items

    def __iter__(self):
        for item in self._items:
            yield item

    def __iter__(self):
        yield from self._items  # только в Python 3.3+

    def __iter__(self):  # простой и менее гибкий вариант
        return iter(self._items)
```

Напоминание. В модуле **collections** есть и другие базовые классы, например **Sequence**. Эти классы реализуют множество полезных методов, требуя переопределить лишь несколько.

Генератор

Генератор – итератор, с которым можно взаимодействовать (PEP-255).

Каждый следующий объект возвращается с помощью выражения **yield**. Это выражение *приостанавливает* работу генератора и передает значение в вызывающую функцию. При повторном вызове исполнение продолжается с *текущей* позиции либо до следующего **yield**, либо до конца функции.

Генераторы удобно использовать, когда вся последовательность сразу не нужна, а нужно лишь по ней итерироваться.

```
>>> assert all(x % 2 for x in range(1, 10, 2))
```

Замечание. Выражения-генераторы имеют вид comprehensions с круглыми скобками. При передаче в функцию дополнительные круглые скобки не нужны.

Замечания по генераторам

Конструкция **yield from** делегирует, по сути, исполнение некоторому другому итератору (Python 3.3+, PEP-380).

В Python 3 появилась возможность у генераторов (например, **range**) узнать длину генерируемой ими последовательности (метод ***__len__***) и проверить, генерируют ли они определенный элемент (метод ***__contains__***).

В Python 2 ввиду реализации **xrange** не принимает числа типа **long**.

Также в Python 3 есть специальный класс – **collections.ChainMap**, который представляет собой обертку над несколькими mapping'ами.

Дополнительные способы итерирования

В стандартной библиотеке есть модуль **itertools**, в котором реализовано множество итераторов:

- **cycle** – зацикливает некоторый iterable
- **count** – бесконечный счетчик с заданным начальным значением и шагом
- **repeat** – возвращает некоторое значение заданное число раз

Также есть комбинаторные итераторы:

- **product** – итератор по декартову произведению последовательностей (по сути, генерирует кортежи, если бы был реализован вложенный *for*)
- **combinations** – итератор по упорядоченным сочетаниям элементов
- **permutations** – итератор по перестановкам переданных элементов

Дополнительные способы итерирования

- **chain** – итерируется последовательно по нескольким iterable
- **zip_longest** – аналог zip, только прекращает итерироваться, когда исчерпывается не первый, а последний итератор
- **takewhile/dropwhile/filterfalse/compress** – отбирает элементы последовательности в соответствии с предикатом
- **islice** – итераторный аналог slice (не создает списка элементов)
- **groupby** – группирует последовательные элементы
- **starmap** – аналог **map**, только распаковывает аргумент при передаче
- **tee** – создает ***n*** копий итератора

Замечание. В Python 2 доступны **ifilter** и **izip** – итераторные аналоги **filter** и **zip**.

Замечание. В Python 3.2 появилась функция **accumulate**, которая возвращает итератор по кумулятивному массиву.

Классы. Финальные замечания

...

Механизм создания классов

Определение класса приводит к следующим действиям:

1. Определяется подходящий ***метакласс*** (класс, который создает другие классы)
2. Подготавливается namespace класса
3. Выполняется тело класса
4. Создается объект класса и присваивается переменной

```
class X(object):
    a = 0

# equivalent: type(name, bases, namespace)
X = type('X', (object, ), {'a': 0})
```

Замечания по созданию классов

Метаклассом по умолчанию является **type**.

Выполнение тела класса приводит к созданию ***словаря*** всех его атрибутов, который передается в **type**. Далее этот словарь доступен через **__dict__** или с помощью built-in функции **vars**.

Замечание. Изменять, добавлять и удалять атрибуты *можно*, модифицируя **__dict__**. Данный способ **менее явный**, нежели использование **getattr** и пр.

Важно! Атрибуты классов при наследовании не перезаписываются, а поиск их происходит последовательно в словарях базовых классов.

Замечания по созданию классов

Вопрос: есть ли разница между реализацией синонима (alias) для функции (функции g и h в примере)?

```
class X(object):
    def f(self):
        return 0

    def g(self):
        return self.f()

    h = f
```

Обычно реализация синонимов необходима при реализации операторов.

Произвольный код в теле класса

Код в модуле выполняется подобно коду телу класса. Неудивительно, ведь модуль – тоже класс! Значит, в теле класса можно писать любые синтаксически корректные конструкции!

```
class C(object):
    if sys.version_info.major == 3:
        def f(self):
            return 1
    else:
        def g(self):
            return 2
```

Замечание. В Python 3 порядок объявления атрибутов сохраняется (PEP-520).

Abstract base classes

В Python есть возможность создавать условные интерфейсы и абстрактные классы. Для этого используется метакласс **ABCMeta** (в Python 3.4 – базовый класс **ABC**) из модуля **abc** (PEP-3119).

Для объявления абстрактного метода используется декоратор **abstractmethod**, абстрактного свойства – **abstractproperty**.

В Python иерархия типов введена для чисел – модуль **numbers** PEP-3141, а также коллекций и функционалов – модуль **collections.abc**.

Создание экземпляра класса

Создание экземпляра класса заключается в вызове метода __ **new** __ для получения объекта класса и метода **__init__** для его инициализации.

```
class C(object):
    def __new__(cls, name):
        return super().__new__(cls)   # make a new class

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

c = C('class')
```

Создание экземпляра класса

Сигнатура метода **__new__** совпадает с сигнатурой **__init__**.

В методе **__new__** можно возвращать объект ***другого*** класса, модифицировать и присваивать атрибуты!

Метод **__init__** не вызывается автоматически, если **__new__** возвращает объект другого класса.

Метаклассы

```
class Meta(type):
    def __new__(mcs, name, bases, attrs, **kwargs):
        # invoked to create class C itself
        return super().__new__(mcs, name, bases, attrs)

    def __init__(cls, name, bases, attrs, **kwargs):
        # invoked to init class C itself
        return super().__init__(name, bases, attrs)

    def __call__(cls):
        # invoked to create an instance of C
        # -> call __new__ and __init__ inside
        # Note: __call__ must share the signature
        #   with class' __new__ and __init__ method signatures
        return super().__call__()
```

Метаклассы

```
1   class C(metaclass=Meta):
2       def __new__(cls):
3           return super().__new__(cls)
4
5       def __init__(self):
6           pass
7
8   c = C()
```

Строка 1: вызываются методы **__new__** и **__init__** метакласса *Meta* (создается объект – класс).

Строка 8: вызывается метод **__call__** метакласса *Meta,* который вызывает методы **__new__** и **__init__** класса *C*.

Метаклассы

Методы **__new__** и **__init__** метакласса принимают **kwargs – ключевые аргументы. Они используются для настройки класса – вызова метода **__prepare__**, который возвращает *mapping* для сохранения атрибутов класса (см. PEP-3115).

Примером метакласса в стандартной библиотеке является Enum (Py 3.4+).

Замечание. В Python 3.6 появился метод **__init_subclass__** (PEP-487), позволяющий изменить создание классов наследников (например, добавить атрибуты).

В классе присутствуют специальный атрибут **__bases__** (кортеж базовых классов) и функция **__subclasses__**, возвращающая список подклассов.

Замечания о классах

В Python 3.6. можно переопределить метод **__set_name__**(self, owner, name) у дескрипторов для получения имени *name*, под которым дескриптор сохраняется в классе *owner*.

Замечание. При реализации **__getattribute__** в некоторых случаях требуется принимать во внимание дескрипторы.

В классах допустимы некоторые атрибуты, характеризующие класс:

- **__slots__** (используется вместо **__dict__**)
- **__annotations__** (аннотации типов: PEP-318, PEP-481, PEP-3107)
- **__weakref__** («слабые ссылки», docs, PEP-205).

Дескрипторы

Свойства (**property**) и декораторы **staticmethod** и **classmethod** являются *дескрипторами* – специальными объектами, реализованными как атрибуты класса (непосредственного или одного из родителей).

В классах в зависимости от типа дескриптора реализуются методы:

- **__get__**(self, instance, owner) # *owner* – instance class / type
- **__set__** (self, instance, value) # *self* – объект дескриптора
- **__delete__**(self, instance) # *instance* – объект, в котором вызывается дескриптор

Стандартное поведение дескрипторов заключается в работе со словарями объекта, класса или базовых классов.

Пример. Вызов **a.x** приводит к вызову **a.__dict__['x']**, потом **type(a).__dict__['x']** и далее по цепочке наследования.

Пример дескриптора

```
class Descriptor(object):
    def __init__(self, label):
        self.label = label

    def __get__(self, instance, owner):
        return instance.__dict__.get(self.label)

    def __set__(self, instance, value):
        instance.__dict__[self.label] = value

class C(object):
    x = Descriptor('x')

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

Дескрипторы

Методы и свойства в классе являются дескрипторами. По сути, в каждой функции (неявно) есть метод **__get__**:

```
class Function(object):
    def __get__(self, obj, objtype=None):
        "Simulate func_descr_get() in Objects/funcobject.c"
        return types.MethodType(self, obj, objtype)
```

Декораторы **classmethod** и **staticmethod** модифицируют аргументы вызова:

Transformation	Called from an Object	Called from a Class
function	f(obj, *args)	f(*args)
staticmethod	f(*args)	f(*args)
classmethod	f(type(obj), *args)	f(klass, *args)

Дескрипторы

Метод – объект-функция, который хранится в словаре атрибутов класса. Доступ же обеспечивается с помощью механизма дескрипторов (см. пример, пример).

```
>>> class D(object):
...     def f(self, x):
...         return x
...
>>> d = D()
>>> D.__dict__['f']  # Stored internally as a function
<function f at 0x00C45070>
>>> D.f  # Get from a class becomes an unbound method
<unbound method D.f>
>>> d.f  # Get from an instance becomes a bound method
<bound method D.f of <__main__.D object at 0x00B18C90>>
```

Ответы на вопросы

...

Ответы на вопросы

- При создании декораторов-классов для применения **functools.wraps** обычно переопределяют метод ***__new__***. Применить напрямую к классу его не удастся. Второй вариант – применить **functools.update_wrapper** в методе ***__init__*** ко вновь созданному объекту класса.