

Python

Лекция 5

Преподаватель: Дмитрий Косицин
BSU FAMCS (Fall'18)

Contents

- Работа с атрибутами
- Механизм создания классов
- Метаклассы
- Дескрипторы
- Интерпретатор, байткод и интроспекция
- GIL и GC
- Прочие возможности и замечания

Работа с атрибутами ...

© Dzmitryi Kasitsyn

BSU FAMCS (Fall'18) • 4

Классы. Финальные замечания ...

© Dzmitryi Kasitsyn

BSU FAMCS (Fall'18) • 11

Python. Финальные замечания ...

© Dzmitryi Kasitsyn

BSU FAMCS (Fall'18) • 27

Работа с атрибутами

...

Динамические атрибуты

Python позволяет *создавать, изменять и удалять* атрибуты run-time. За это отвечают следующие магические методы и глобальные функции:

- Получение атрибута по имени: `__getattr__`, `__getattribute__` | `getattr`
- Присваивание атрибута по имени: `__setattr__` | `setattr`
- Удаление атрибута по имени: `__delattr__` | `delattr`
- Проверка наличия атрибута: `hasattr` (функция-обертка над `getattr`)

Замечание. Некоторые объекты являются *readonly*, например, добавлять или удалять атрибуты **object** нельзя.

Замечание. Изменять атрибуты можно (*не рекомендуется*), модифицируя `__dict__`.

Пример работы с атрибутами

```
class A(object):  
    def f(self):  
        pass
```

```
a = A()
```

```
assert hasattr(a, 'f') and hasattr(A, 'f')  # 'f' is a class attribute  
assert getattr(a, 'g', None) is None      # no such attribute -> default
```

```
setattr(a, 'x', 2)  # a.x = 2  
assert getattr(a, 'x') == 2  # assert a.x == 2  
assert not hasattr(A, 'x')  # attribute has been set for instance only
```

```
delattr(a, 'x')  # del a.x  
assert not (hasattr(a, 'x') or hasattr(A, 'x'))
```

Магические методы работы с атрибутами

Метод	Вызывается	Примечание
<code>__getattr__(self, name)</code> # x.name	При обращении к атрибуту, которого нет	<ul style="list-style-type: none">• Возвращает значение или бросает <code>AttributeError</code>
<code>__getattribute__(self, name)</code> # x.name	При обращении к любому атрибуту	<ul style="list-style-type: none">• Возвращает значение или бросает <code>AttributeError</code>• Обращение к другим атрибутам: <code>super().__getattribute__(name)</code>
<code>__setattr__(self, name, value)</code> # x.name = value	При установке любого атрибута	<ul style="list-style-type: none">• Может изменять другие атрибуты вызовом <code>super().__setattr__(name, value)</code>

Пример реализации методов работы с атрибутами

```
class Proxy(object):  
    def __init__(self, inner_object):  
        self._inner_object = inner_object  
  
    def __setattr__(self, name, value):  
        if name != '_inner_object':  
            setattr(self._inner_object, name, value)  
        else:  
            super().__setattr__(name, value)  
  
    def __getattr__(self, name):  
        if name == '_inner_object':  
            return super().__getattr__(name)  
        return getattr(self._inner_object, name)
```


Пример реализации методов работы с атрибутами

```
p = Proxy([1])    # p._inner_object = [1]
p.append(2)      # p._inner_object = [1, 2]
```

Напоминание. Обратите внимание на вызовы методов базовых классов:

- `super(Proxy, self).__getattr__(name, value)`
- `object.__getattr__(self, name)`

Такие вызовы (в случае одинаковых методов, разумеется) эквиваленты, как через **super (...)**, так и через `<base class>.method_name`.

Замечания по работе с атрибутами

Если *Proxy* не нужно добавлять атрибуты, то можно было ограничиться переопределением метода `__getattr__` и реализацией конструктора:

```
class Proxy(object):  
    ...  
    def __getattr__(self, name):  
        return getattr(self._inner_object, name)
```

Важно! Интерпретатор оптимизирует вызов **всех** *магических* методов: явный вызов `x.__len__` обратится к `__getattribute__`, неявный `len(x)` – нет.

Замечание. Создавать новые методы, присваивая функции объекту или классу, не корректно. Присваивать нужно объекты типа `types.MethodType`.

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](https://www.python.org/dev/peps/pep-0520/)).

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 `__getattr__` 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>>
```

Python. Финальные замечания

...

Интерпретатор. Байткод

Для Python кода в стандартной библиотеке есть AST-парсер.

Код можно либо преобразовать в Abstract Syntax Tree ([ast module](#)), либо скомпилировать – преобразовать в байткод, который далее интерпретировать ([dis module](#)).

Каждой инструкции байткода соответствует функция в интерпретаторе, которая ее выполняет.

Замечание. Подробнее о байткоде тут: [article](#), [article](#).

Замечание. Статья об интерпретаторе тут: [article](#), [slides](#).

Пример байткода

```
import dis

global_a = 0

def f(closure_b):
    enclosing_c = 1

    def g(param_d):
        local_e = 2
        return (global_a +
                closure_b + enclosing_c +
                param_d + local_e)

    dis.dis(g)

f(-1)
```

8	0	LOAD_CONST	1	(2)
	3	STORE_FAST	1	(local_e)
9	6	LOAD_GLOBAL	0	(global_a)
	9	LOAD_DEREF	0	(closure_b)
	12	BINARY_ADD		
	13	LOAD_DEREF	1	(enclosing_c)
	16	BINARY_ADD		
	17	LOAD_FAST	0	(param_d)
	20	BINARY_ADD		
	21	LOAD_FAST	1	(local_e)
	24	BINARY_ADD		
	25	RETURN_VALUE		

Интерпретаторы

Есть множество реализаций интерпретаторов, которые написаны на разных языках программирования:

- CPython, Jython, IronPython – интерпретаторы на C, Java и .Net
- PyPy, Numba – Just-in-Time compilers
- Cython, Nuitka – использование типов и оптимизации (на базе CPython)
- Stackless Python, Julia – прочие реализации и расширения языка

Естественно, что если интерпретатор написан на C, то можно из Python напрямую взаимодействовать с C-кодом – реализовывать C/C++ extensions ([standard library](#), [boost](#), [Pybind11](#)).

Интерпретатор CPython

Все объекты в CPython описываются структурами.

Типу **object** соответствует структура **PyObject**, с указателем на которую он повсеместно работает.

Список в Python – аналог **vector** в C++ STL, расширяется в 9/8 раз (плюс константа; см. *listobject.c*).

Словарь в Python – хэш-таблица с открытой адресацией (см. *dictobject.c*):

- минимальный размер по умолчанию 8
- смещение считается как $j = ((5*j) + 1) \bmod 2^{*i}$
- расширяется при наполненности от 1/2 до 2/3 в 2-3 раза от количества хранимых элементов

Интроспекция

Интерпретируемый язык позволяет легко создавать код «на лету», а также контролировать выполнение программы: проверить тип объекта, узнать сигнатуру функции, последний фрейм и все переменные в нем, стек вызовов.

Для работы со стеком и фреймами используется модуль [sys](#), для работы с объектами – модуль [inspect](#).

Замечание. В Python стек рекурсии ограничен (по умолчанию 1000, может быть изменена). Оптимизация хвостовой рекурсии отсутствует.

В Python есть собственный интерактивный отладчик – [pdb](#) (зачастую используется многими IDE).

GIL and GC

В Python есть **Global Interpreter Lock** – механизм, который гарантирует одновременное выполнение только одного потока. Переключение **GIL** в последней версии Python происходит по таймеру.

Для всех объектов в Python ведется счетчик ссылок, а все объекты классифицируются в три поколения. **GC** также умеет разрешать циклические зависимости, если у объектов не переопределен метод `__del__`.

Сериализация данных

- Для текста и простых объектов: json, yaml; для структур: struct.
- Для Python объектов (кроме потоков, lambda-функций, frame'ов и некоторых др.) и их передачи между процессами: [pickle](#)
- Другие библиотеки: [bson](#), [dill](#); [protobuf](#)

Подробнее о нерасказанном

- Global interpreter lock ([presentation](#))
- Garbage collection and weak references ([docs](#), [docs](#), [article](#))
- [Multiprocessing](#) and [multithreading](#) (incl. [concurrent](#), [subprocess](#) and [signal](#))
- Coroutines and asynchronous coroutines ([docs](#); [article](#), [article](#), [article](#); [PEP-342](#), [PEP-479](#), [PEP-492](#), [PEP-525](#), [PEP-530](#), [PEP-3148](#), [PEP-3156](#))
- Type annotations and type hints (e.g. [PyCharm IDE docs](#) + Python 3 features: [PEP-318](#), [PEP-481](#), [PEP-526](#), [PEP-3107](#))
- Packaging ([setuptools](#), [distutils](#))

Возможности Python

Python позволяет:

- Писать утилиты (работа с файловой системой, простая обработка данных) и прототипы программ
- Работать с данными и производить вычисления
- Реализовывать высокоуровневые интерфейсы (web frameworks)

Python является удобной оберткой над многими низкоуровневыми штуками: позволяет их единообразно использовать и не думать о работе с памятью.

Прочие возможности

- Основная библиотека для работы с изображениями – [Pillow](#)
- Статья о том, как писать приложение на Kivy: [article](#), еще есть на Habr
- Ссылка на telegram-группу минского Python сообщества: [link](#)

Python Zen

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Readability counts.

Special cases aren't special enough to break the rules.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

...

Spider-Man rule: With Great Power Comes Great Responsibility!

The End

Thank you!