Урок 2

Дескрипторы и метаклассы

Дескрипторы атрибутов, доступ к атрибутам. Метаклассы.

На этом уроке рассмотрим дескрипторы атрибутов и метаклассы. Потребуются базовые знания о возможностях работы с классами в Python. Будем использовать понятия «класс», «атрибут», «метод», «наследование», «инкапсуляция», «интерфейс класса», «родительский/дочерний класс» — их надо помнить. Затронем более сложные и интересные возможности классов в Python.

Краткий экскурс в ООП-1

Вспомним, что изучали по теме ООП в курсе Python 1:

- класс — «абстракция», экземпляр — конкретный «представитель» абстракции;
- атрибут — (как правило) данные класса/экземпляра, метод — функция класса;
- интерфейс — способ «общения» с данным классом (например, «интерфейс итератора»);
- классы бывают классические и «нового стиля». В Python 3 все классы — «нового стиля»;
- тело класса выполняется при первом чтении файла интерпретатором;
- для методов класса существуют специальные декораторы: **@property, @classmethod, @staticmethod;**
- свойство — вычисляемый атрибут (метод, обернутый декоратором **@property**);
- функции **getattr**, **setattr**, **hasattr**;
- в ООП нередко используют устоявшиеся подходы к решению задач проектирования — шаблоны (паттерны) проектирования (в курсе Python 1 познакомились с шаблонами «Строитель», «Делегирование», «Фабрика»).

У свойств могут быть свои атрибуты — **setter, getter, deleter**:

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

То же самое может быть записано по-другому:

```
class C:
    def _get_x(self):
        """I'm the 'x' property."""
        return self._x

    def _set_x(self, value):
        self._x = value

    def _del_x(self):
        del self._x

    x = property(_get_x, _set_x, _del_x)
```

Дескрипторы атрибутов

При использовании свойств (**@property**) доступ к атрибутам управляется серией пользовательских функций **get**, **set** и **delete**. Такой способ не вполне универсален, так как для каждого однотипного атрибута должен быть свой набор **get/set/delete**-методов. Более универсально использование объекта дескриптора. Это обычный объект, представляющий значение атрибута. За счет реализации одного или более специальных методов **__get__()**, **__set__()** и **__delete__()** он может подменять механизмы доступа к атрибутам и влиять на выполнение этих операций.

Рассмотрим пример дескриптора, который контролирует тип значения для атрибута и препятствует удалению атрибута из экземпляра объекта (файл **листинг 1**):

```
class TypedProperty:
    def __init__(self, name, type_name, default=None):
        self.name = "_" + name
        self.type = type_name
        self.default = default if default else type_name()

    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default)

    def __set__(self, instance, value):
        if not isinstance(value, self.type):
            raise TypeError("Значение должно быть типа %s" % self.type)
        setattr(instance, self.name, value)

    def __delete__(self, instance):
        raise AttributeError("Невозможно удалить атрибут")

class Foo:
    name = TypedProperty("name", str)
    num = TypedProperty("num", int, 42)
```

В этом примере класс **TypedProperty** определяет дескриптор, который выполняет проверку типа при присваивании значения атрибуту и вызывает исключение при попытке удалить атрибут:

```
f = Foo()
a = f.name          # Неявно вызовет Foo.name.__get__(f, Foo)
f.name = "Гвидо"    # Вызовет Foo.name.__set__(f, "Guido")
del f.name          # Вызовет Foo.name.__delete__(f)
```

Протокол дескриптора

Чтобы создать дескриптор, нужно реализовать класс не менее чем с одним методом:

- **__get__(self, obj, type=None)** — должен вернуть значение **value**;
- **__set__(self, obj, value)** — возвращает **None**;
- **__delete__(self, obj)** — возвращает **None**.

Типы дескрипторов

Дескрипторы делят на два типа:

1. **Дескриптор данных (data-descriptor)** — реализует метод **__set__** (может также иметь методы **__get__** и **__delete__**). Всегда перегружает словарь экземпляра.
2. Простой дескриптор (**non-data-descriptor**) — не имеет метода **__set__** (реализует методы **__get__** и/или **__delete__**). Может быть перегружен через словарь экземпляра.

Примеры, демонстрирующие особенности использования дескрипторов различных типов, приведены в файле **листинг 2**.

Хранение значений атрибутов

При работе с дескрипторами атрибутов возникает вопрос — как хранить значения атрибутов. Варианты (файл **листинг 3**):

1. **Хранить данные в атрибуте объекта дескриптора** — при этом они будут общими для всех экземпляров классов, использующих этот дескриптор:

```
# Первый способ сохранить данные — просто в атрибуте объекта дескриптора.
class Grade:
    def __init__(self):
        self._value = 0

    def __get__(self, instance, instance_type):
        return self._value

    def __set__(self, instance, value):
        if not (1 <= value <= 5):
            raise ValueError("Оценка должна быть от 1 до 5")
        self._value = value

class Exam():
    ''' Класс Экзамен.
        Для простоты хранит только оценку за экзамен.
    '''
    grade = Grade()

# Но не стоит забывать, что при таком подходе
# данные будут сохранены на уровне атрибута класса Экзамен!!!
# Т.е. будут общими для всех экземпляров класса Экзамен.

# Для демонстрации создадим два Экзамена:
math_exam = Exam()
math_exam.grade = 3
language_exam = Exam()
language_exam.grade = 5

print("  Проверим результаты: ")
print("Первый экзамен ", math_exam.grade, " — верно?")
print("Второй экзамен ", language_exam.grade, " — верно?")
print('Потому что... ')
print('math_exam.grade is language_exam.grade =', math_exam.grade is
language_exam.grade)
```

2. **Хранить данные в отдельном словаре объекта дескриптора** — ключом будет служить сам объект внешнего класса:

```
class Grade:
    def __init__(self):
        self._values = {}

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return self._values.get(instance, 0)

    def __set__(self, instance, value):
        if not (1 <= value <= 5):
            raise ValueError("Оценка должна быть от 1 до 5")
        self._values[instance] = value
```

Хотя это решение простое и полноценно работает, оно будет приводить к утечкам памяти. Словарь **_values** будет хранить ссылку на каждый внешний экземпляр класса, который когда-либо передавался в метод **__set__**. Это приведет к тому, что счетчик ссылок у внешних экземпляров никогда не будет равен нулю, и сборщик мусора не выполнит свою работу.

Для данного решения вместо обычного **dict** нужно использовать класс **weakref.WeakKeyDictionary**:

```
from weakref import WeakKeyDictionary
class Grade:
    def __init__(self):
        self._values = WeakKeyDictionary()
        ...
```

Модуль **weakref** поддерживает слабые ссылки. В обычном случае сохранение ссылки на объект приводит к увеличению их счетчика. Это препятствует уничтожению объекта, пока значение счетчика не достигнет нуля. Слабая ссылка позволяет обращаться к объекту, не увеличивая счетчик.

Класс **WeakKeyDictionary([dict])** создает словарь, в котором ключи представлены слабыми ссылками. Когда обычных ссылок на объект ключа не остается, соответствующий элемент словаря автоматически удаляется. В необязательном аргументе **dict** передается словарь, элементы которого добавляются в возвращаемый объект типа **WeakKeyDictionary**. Слабые ссылки могут создаваться только для объектов определенных типов, поэтому существует много ограничений на допустимые типы объектов ключей. Встроенные строки нельзя использовать в качестве ключей со слабыми ссылками. Однако экземпляры пользовательских классов, объявляющих метод **__hash__()**, могут выступать ключами. Экземпляры класса **WeakKeyDictionary** имеют два дополнительных метода: **iterkeyrefs()** и **keyrefs()**, — которые возвращают слабые ссылки на ключи.

Данное решение содержит незначительное ограничение — в одном внешнем классе нельзя сохранять данные дескрипторов одного типа (например, «Экзамен» с несколькими «Оценками»).

3. **Хранить данные в отдельном атрибуте внешнего класса** — требуется только определить способ именования атрибута. Такой подход позволяет во внешнем классе создавать несколько объектов-дескрипторов одного класса:

```
class Grade:
    def __init__(self, name):
        # Для данного подхода необходимо сформировать отдельное имя атрибута
        self.name = '_' + name

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return "*{}*".format(getattr(instance, self.name))

    def __set__(self, instance, value):
        if not (1 <= value <= 100):
            raise ValueError("Балл ЕГЭ должен быть от 1 до 100")
        setattr(instance, self.name, value)

class ExamEGE():
    ''' Комплексный экзамен, на котором оцениваются разные критерии. '''
    # Для обновленного Grade нужно добавить строковые имена
    math_grade = Grade('math_grade')
    writing_grade = Grade('writing_grade')
    science_grade = Grade('science')
```

Поиск атрибутов

Действия чтения, записи и удаления атрибута Python будет выполнять по-разному.

Чтение атрибута

При попытке получить значение атрибута (**print(obj.attr)**) выполняются следующие действия:

1. Если **attr** — это специальный атрибут (на уровне Python), вернуть его.
2. Существует ли **attr** в **obj.__class__.__dict__** (т.е. **obj.__class__.__dict__[“attr”]**)?
    - Если да и это дескриптор данных, вернуть результат работы дескриптора (результат метода **__get__** дескриптора);
    - Выполнить аналогичную проверку во всех базовых классах **obj.__class__**.
3. Существует ли **attr** в **obj.__dict__**?
    - Если да, вернуть это значение;
    - Если **obj** — это класс, выполнить проверку во всех его базовых классах:
        - если в этом классе или его базовых классах существует дескриптор, вернуть его результат.
4. Существует ли **attr** в **obj.__class__.__dict__**?
    - Если существует и это не дескриптор данных (**non-data**), вернуть результат дескриптора;

- Если существует и это не дескриптор, просто вернуть его;
- Выполнить аналогичную проверку для всех базовых классов **obj.__class__**.

5. Создать исключение **AttributeError**.

Запись атрибута

Установка значения для атрибутов (**obj.attr = data**) выполняется проще чтения:

1. Существует ли **attr** в **obj.__class__.__dict__**?
    - Если да и это дескриптор данных, использовать дескриптор для установки значения (метод **__set__** дескриптора);
    - Выполнить такую же проверку во всех базовых классах **obj.__class__**.
2. Добавить значение **data** для ключа **attr** в словарь **obj.__dict__** (то есть **obj.__dict__[“attr”] = data**).

Удаление атрибута

Удаление атрибута (**del obj.attr**) выполняется аналогично записи:

1. Существует ли **attr** в **obj.__class__.__dict__**?
    - Если да и это дескриптор данных, использовать дескриптор для удаления атрибута (метод **__delete__** дескриптора);
    - Выполнить такую же проверку во всех базовых классах **obj.__class__**.
2. Удалить значение **data** для ключа **attr** из словаря **obj.__dict__** (т.е. **obj.__dict__.pop([“attr”])**).

Доступ к атрибутам

Рассмотрим методы, обеспечивающие доступ к атрибутам:

- **__getattr__** — вызывается, когда атрибут не найден в словаре **__dict__** экземпляра класса;
- **__getattribute__** — вызывается при каждом доступе к атрибуту объекта, даже если атрибут не существует в словаре **__dict__** экземпляра класса; а также при обращении к функциям **getattr**, **hasattr**;
- **__setattr__** — вызывается всякий раз, когда атрибут назначается экземпляру класса (в том числе при обращении к функции **setattr**).

Приведем примеры, демонстрирующие работу данных методов (**листинг 4**):

```
# ——— __getattr__ + __getattribute__
class ValidatingDB:
    def __init__(self):
        self.exists = 5

    def __getattr__(self, name):
        print(' ValidatingDB.__getattr__(%s)' % name)
        value = 'Super %s' % name
        setattr(self, name, value)
        return value

    def __getattribute__(self, name):
        print(' ValidatingDB.__getattribute__(%s)' % name)
        return super().__getattribute__(name)

data = ValidatingDB()
print('Атрибут exists:', data.exists)
print('Атрибут foo: ', data.foo)
print('Снова атрибут foo: ', data.foo)
print('Есть ли атрибут zoom в объекте:', hasattr(data, 'zoom'))
print('Атрибут face в объекте, доступ через getattr:', getattr(data, 'face'))

# Использование метода __setattr__
class SavingDB:
    def __setattr__(self, name, value):
        print(' SavingDB.__setattr__(%s, %r)' % (name, value))
        # Сохранение данных в БД
        # ...
        super().__setattr__(name, value)

data = SavingDB()
print('data.__dict__ до установки атрибута: ', data.__dict__)
data.foo = 5
print('data.__dict__ после установки атрибута: ', data.__dict__)
data.foo = 7
print('data.__dict__ в итоге:', data.__dict__)
```

При реализации методов **__getattribute__** и **__setattr__** можно столкнуться с ситуацией рекурсии, когда методы вызываются при каждом обращении к атрибуту объекта. В итоге Python исчерпывает стек вызовов и прерывает работу:

```
class BrokenDictionaryDB(object):
    def __init__(self, data):
        self._data = data

    def __getattribute__(self, name):
        print('Called __getattribute__(%s)' % name)
        return self._data[name]

data = BrokenDictionaryDB({'foo': 3})
print(data.foo)
```

Загвоздка в том, что метод **__getattribute__** обращается к **self._data**, что снова приводит к вызову **__getattribute__**, а он вновь — к **self._data**. И так пока не остановится работа интерпретатора.

Для решения этой проблемы внутри методов **__getattribute__** и **__setattr__** необходимо обращаться к атрибуту объекта через объект **super**: **super().__getattribute__ или super().__setattr__**.

```
class DictionaryDB(object):
    def __init__(self, data):
        self._data = data

    def __getattribute__(self, name):
        data_dict = super().__getattribute__('_data')
        return data_dict[name]

data = BrokenDictionaryDB({'foo': 'This is the right way!'})
print(data.foo)
```

- Экземпляры дескрипторов создаются только на уровне класса (не для экземпляра в отдельности, не внутри метода **__init__()** или других);
- Имя атрибута-дескриптора в классе имеет более высокий приоритет перед другими атрибутами на уровне экземпляров;
- Следует понимать разницу между дескриптором данных и простым дескриптором;
- Сначала вызываются методы **__getattribute__()/ __setattr__()**, потом уже **__get__()/__set__** дескриптора;
- Следует избегать бесконечной рекурсии при реализации методов **__getattribute__** и **__setattr__**.

Метаклассы

[Metaclasses] are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).
— Tim Peters
inventor of the timsort algorithm and prolific Python contributor

Многим известно это высказывание Тима Петерса, и под влиянием этой фразы некоторые программисты Python решили, что на изучение метаклассов не стоит тратить время. Несмотря на это, рекомендуем уделить внимание этой теме — как минимум будете лучше разбираться в технологиях, которыми пользуетесь. А вполне вероятно, что и сможете применить полученные знания для создания собственного фреймворка.

Знакомство с метаклассами

Когда программа на Python объявляет класс, само определение его становится объектом:

```
class Foo(object):
    pass

isinstance(Foo, object)          # Вернет True
```

Что-то должно было создать объект **Foo**. Этим процессом управляет специальный объект — метакласс. Он знает, как создавать классы и управлять ими.

В предыдущем примере метаклассом, под управлением которого создается объект **Foo**, является **type()**. Если попытаться вывести тип объекта **Foo**, можно увидеть, что это **type**:

```
>>> type(Foo)
<class 'type'>
```

Когда с помощью инструкции **class** определяется новый класс, происходит последовательность действий. Тело класса выполняется интерпретатором, как набор инструкций, с использованием отдельного словаря. Инструкции выполняются так же, как в обычном программном коде, но дополнительно изменяются имена частных членов класса (начинающихся с префикса __). В заключение имя класса, список базовых классов и словарь передаются конструктору метакласса, который создает соответствующий объект класса:

```
class_name = "Foo"          # Имя класса
class_parents = (object, ) # Базовые классы
class_body = """            # Тело класса
    def __init__(self, x):
        self.x = x
    def blah(self):
        print("Hello World")
"""
class_dict = {}
# Выполнить тело класса с использованием локального словаря class_dict
exec(class_body, globals(), class_dict)

# Создать объект класса Foo
Foo = type(class_name, class_parents, class_dict)
```

Заключительный этап создания класса, когда вызывается метакласс **type()**, можно изменить. В классе можно явно указать его метакласс, добавив именованный аргумент **metaclass** в кортеж с именами базовых классов (в Python 3) или установив переменную класса **__metaclass__** (в Python 2):

```
class Foo(metaclass=type):
    pass
    ...
```

Если метакласс явно не указан, инструкция **class** проверит первый элемент в кортеже базовых классов (если таковой имеется). В этом случае метаклассом будет тип первого базового класса. То есть инструкция **class Foo(object): pass** создаст объект **Foo** того же типа, которому принадлежит класс **object**.

Если базовые классы не указаны, инструкция **class** проверит наличие аргумента с именем **metaclass**. Если такой аргумент присутствует, он будет использоваться при создании классов. С его помощью можно управлять этим процессом.

Если аргумент **metaclass** не обнаружится, интерпретатор будет использовать метакласс по умолчанию. В Python 3 метаклассом по умолчанию является **type**.

Методы __new__, __init__, __call__

В основном метаклассы используются во фреймворках, когда требуется более полный контроль над определениями пользовательских объектов. Когда определяется нестандартный метакласс, он обычно наследует класс **type** и переопределяет методы **__init__()** или **__new__()**. Рассмотрим пример метакласса, который требует, чтобы все методы снабжались строками документирования (**листинг 5**):

```
class DocMeta(type):
    def __init__(self, clsname, bases, clsdict):
        for key, value in clsdict.items():
            # Пропустить специальные и частные методы
            if key.startswith("__"):
                continue

            # Пропустить любые невызываемые объекты
            if not hasattr(value, "__call__"):
                continue

            # Проверить наличие строки документирования
            if not getattr(value, "__doc__"):
                raise TypeError("%s must have a docstring" % key)

        type.__init__(self, clsname, bases, clsdict)
```

В этом метаклассе реализован метод **__init__()**, который проверяет содержимое словаря класса. Он отыскивает методы и проверяет, имеют ли они строки документирования. Если в каком-либо методе строка документирования отсутствует, возникает исключение **TypeError**. В противном случае для инициализации класса вызывается реализация метода **type.__init__()**.

Чтобы воспользоваться этим метаклассом, класс должен явно выбрать его. Обычно для этого сначала определяется базовый класс:

```
class Documented(metaclass=DocMeta):
    pass
```

А затем он используется как родоначальник всех объектов, которые должны включать описание:

```
class Foo(Documented):

    def spam(self, a, b):
        ''' Метод spam делает кое-что '''
        pass

    def boo(self):
        print('A little problem')
```

Этот пример иллюстрирует одно из основных применений метаклассов: проверку и сбор информации об определениях классов. Метакласс ничего не изменяет в создаваемом классе — просто выполняет дополнительные проверки.

В более сложных случаях перед тем, как создать класс, метакласс может не только проверять, но и изменять содержимое его определения. Если предполагается вносить изменения, необходимо переопределить метод **__new__()**, который выполняется перед созданием класса. Этот прием часто объединяется с обертыванием атрибутов дескрипторами или свойствами, потому что это единственный способ получить имена, использованные в классе. В качестве примера рассмотрим модифицированную версию дескриптора **TypedProperty_v2**, который был реализован в теме

«Дескрипторы». Обе версии дескриптора расположены в файле **листинг 6**, а соответствующий пример с метаклассом — в файле **листинг 7**:

```
class TypedProperty_v2:
    ''' Дескриптор атрибутов, контролирующий принадлежность указанному типу '''
    def __init__(self, type_name, default=None):
        self.name = None
        self.type = type_name
        if default:
            self.default = default
        else:
            self.default = type_name()

    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default)

    def __set__(self, instance, value):
        if not isinstance(value, self.type):
            raise TypeError("Значение должно быть типа %s" % self.type)
        setattr(instance, self.name, value)

    def __delete__(self, instance):
        raise AttributeError("Невозможно удалить атрибут")
```

В данном примере атрибуту **name** дескриптора просто присваивается значение **None**. Заполнение этого атрибута будет поручено метаклассу:

```
class TypedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        slots = []
        for key, value in clsdict.items():
            if isinstance(value, TypedProperty_v2):
                value.name = "_" + key
            slots.append(value.name)
        clsdict['__slots__'] = slots
        return type.__new__(cls, clsname, bases, clsdict)

class Typed(metaclass=TypedMeta):
    ''' Базовый класс для объектов, определяемых пользователем '''
    pass
```

В этом примере метакласс просматривает словарь класса, чтобы отыскать экземпляры класса **TypedProperty_v2**. Если находит, устанавливает значение атрибута **name** и добавляет его в список имен **slots**. После этого в словарь класса добавляется атрибут **__slots__** и вызывается метод **__new__()** метакласса **type**, который создает объект класса. Пример использования нового метакласса:

```
class Foo(Typed):
    name = TypedProperty_v2(str)
    num = TypedProperty_v2(int, 42)
```

Реализация метода **__call__()** в метаклассе позволяет управлять классом, когда создается экземпляр (по аналогии обращения к имени класса как к функции). Результатом работы метода **__call__** метакласса должен быть экземпляр пользовательского класса.

Используя возможности метаклассов с использованием метода **__call__**, можно интересно реализовать шаблон «Одиночка» (Singleton — для класса может быть добавлен только один экземпляр, все новые вызовы конструктора объекта будут возвращать созданный ранее экземпляр) (файл **листинг 8**):

```
class Singleton(type):
    def __init__(self, *args, **kwargs):
        # У каждого подконтрольного класса будет атрибут __instance,
        # который будет хранить ссылку на созданный экземпляр класса
        self.__instance = None
        super().__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        if self.__instance is None:
            # Если еще не создан ни один экземпляр класса, создаем его
            self.__instance = super().__call__(*args, **kwargs)
            return self.__instance
        else:
            # Если уже есть экземпляр класса, возвращаем его
            return self.__instance

class A(metaclass=Singleton):
    def __init__(self):
        print('Class A')

class B(metaclass=Singleton):
    def __init__(self):
        print('Class B')

# Создадим несколько экземпляров каждого класса и проверим их на идентичность
a_1 = A()
a_2 = A()
b_1 = B()
b_2 = B()

print('a_1 is a_2 — ', a_1 is a_2)
print('b_1 is b_2 — ', b_1 is b_2)
print('a_1 is b_1 — ', a_1 is b_1)
print('a_2 is b_2 — ', a_2 is b_2)
```

Метод __prepare__

В дополнение к методам **__new__**, **__init__** и **__call__** в Python 3 для метаклассов был добавлен специальный метод **__prepare__**. Он относится только к метаклассам и обязан быть методом класса (должен быть снабжен декоратором **@classmethod**). Интерпретатор вызывает метод **__prepare__** до **__new__**, чтобы тот создал отображение (словарь), которое будет заполнено атрибутами из тела класса. Первым аргументом **__prepare__** получает сам метакласс, а за ним — имя конструируемого класса и кортеж его базовых классов. Вернуть он должен отображение, которое будет передано в

последнем аргументе методу **__new__** и далее методу **__init__**, когда метакласс примется за построение нового класса.

Применяя метод **__prepare__**, можно использовать упорядоченный словарь (**collections.OrderedDict**) вместо обычного для отображения атрибутов класса (полный код примера содержится в файле **листинг 9**):

```
import collections

class EntityMeta(type):
    """ Метакласс для прикладных классов с контролируемыми полями """
    @classmethod
    def __prepare__(cls, name, bases):
        # Атрибуты класса будут теперь храниться в экземпляре OrderedDict
        return collections.OrderedDict()

    def __init__(cls, name, bases, attr_dict):
        super().__init__(name, bases, attr_dict)
        cls._field_names = []    # Атрибут _field_names создается в
конструируемом классе
        for key, attr in attr_dict.items():
            if isinstance(attr, TypedProperty_v2):
                type_name = type(attr).__name__
                attr.name = '_{}_{}'.format(type_name, key)
                cls._field_names.append((key, attr.name))

class Entity(metaclass=EntityMeta):
    """ Прикладной класс с контролируемыми полями """
    @classmethod
    def field_names(cls):
        ''' Просто возвращает поля в порядке добавления '''
        for name in cls._field_names:
            yield name
```

После простых модификаций, показанных в примере, можно обойти поля типа **TypedProperty_v2** любого подкласса **Entity**, воспользовавшись методом класса **field_names**:

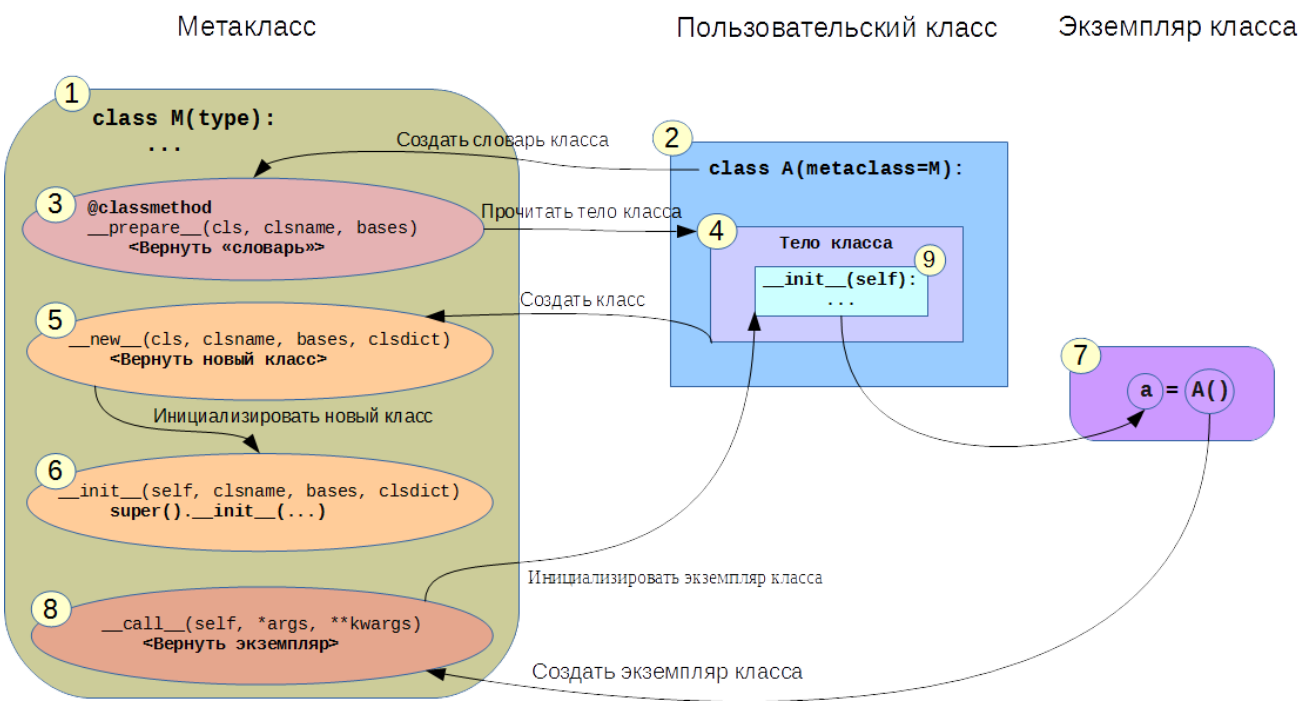
```
class LineItem(Entity):
    description = TypedProperty(str, 'Simple Line')
    weight = TypedProperty(int, 13)
    price = TypedProperty(float, 19.99)

for name in LineItem.field_names():
    print(name)
```

Начиная с Python 3.6, метод **__prepare__** по умолчанию возвращает **OrderedDict**.

Метаклассы способны коренным образом изменять поведение и семантику пользовательских классов. Но не следует злоупотреблять этой возможностью и изменять поведение классов так, чтобы оно существенно отличалось от описанного в стандартной документации — чтобы не путать пользователей.

Подводя итоги знакомства с метаклассами, рассмотрим диаграмму взаимодействия метакласса, класса и экземпляра класса. Пример, отражающий очередность вызова методов метакласса, приведен в **листинге 11**.



Примеры использования метаклассов

Примеры из реальных проектов.

Django

Django — многим уже знакомый свободный фреймворк для веб-приложений на Python, использующий шаблон проектирования MVC. Рассмотрим часть метода **__new__** метакласса **ModelBase** из Django 1.8 (данный метакласс объемен, при желании самостоятельно ознакомьтесь с его кодом):

```
class ModelBase(type):
    """ Metaclass for all models. """
    def __new__(cls, name, bases, attrs):
        super_new = super(ModelBase, cls).__new__

        # Also ensure initialization is only performed for subclasses of Model
        # (excluding Model class itself).
        parents = [b for b in bases if isinstance(b, ModelBase)]
        if not parents:
            return super_new(cls, name, bases, attrs)

        # Create the class.
        module = attrs.pop('__module__')
        new_class = super_new(cls, name, bases, {'__module__': module})
        attr_meta = attrs.pop('Meta', None)
        abstract = getattr(attr_meta, 'abstract', False)
        if not attr_meta:
            meta = getattr(new_class, 'Meta', None)
        else:
            meta = attr_meta
        base_meta = getattr(new_class, '_meta', None)

        # Look for an application configuration to attach the model to.
        app_config = apps.get_containing_app_config(module)
        ...
    ...
```

SQLAlchemy

SQLAlchemy — это программная библиотека на Python для работы с реляционными СУБД с применением технологии ORM. Служит для синхронизации объектов Python и записей реляционной базы данных. **SQLAlchemy** позволяет описывать структуры баз данных и способы взаимодействия с ними на Python без использования **SQL**. Рассмотрим **SQLAlchemy** в теме «Взаимодействие с БД» в данном курсе.

1. Метакласс **DeclarativeMeta** (создает классы, относящиеся к **sqlalchemy.schema.Table**):

```
class DeclarativeMeta(type):
    def __init__(cls, classname, bases, dict_):
        if '_decl_class_registry' not in cls.__dict__:
            _as_declarative(cls, classname, cls.__dict__)
        type.__init__(cls, classname, bases, dict_)

    def __setattr__(cls, key, value):
        _add_attribute(cls, key, value)
```

2. Функция **declarative_base()** создает метакласс (фабрика метаклассов), который в дальнейшем будет добавлять классы, связанные с таблицами БД:

```
def declarative_base(bind=None, metadata=None, mapper=None, cls=object,
                     name='Base', constructor=_declarative_constructor,
                     class_registry=None,
                     metaclass=DeclarativeMeta):
    lcl_metadata = metadata or MetaData()
    if bind:
        lcl_metadata.bind = bind

    if class_registry is None:
        class_registry = weakref.WeakValueDictionary()

    bases = not isinstance(cls, tuple) and (cls,) or cls
    class_dict = dict(_decl_class_registry=class_registry,
                      metadata=lcl_metadata)
    if isinstance(cls, type):
        class_dict['__doc__'] = cls.__doc__

    if constructor:
        class_dict['__init__'] = constructor
    if mapper:
        class_dict['__mapper_cls__'] = mapper

    return metaclass(name, bases, class_dict)
```

Scapy

Scapy — интерактивная оболочка и программная библиотека для манипулирования сетевыми пакетами на Python 2. Класс **Packet_metaclass** помогает создавать классы различных сетевых пакетов:

```
class Packet_metaclass(type):
    def __new__(cls, name, bases, dct):
        if "fields_desc" in dct: # perform resolution of references to other
packets
            current_fld = dct["fields_desc"]
            resolved_fld = []
            for f in current_fld:
                if isinstance(f, Packet_metaclass): # reference to another
fields_desc
                    for f2 in f.fields_desc:
                        resolved_fld.append(f2)
                else:
                    resolved_fld.append(f)
        else: # look for a field_desc in parent classes
            resolved_fld = None
            for b in bases:
                if hasattr(b,"fields_desc"):
                    resolved_fld = b.fields_desc
                    break
       ...

    def __getattr__(self, attr):
        for k in self.fields_desc:
            if k.name == attr:
                return k
        raise AttributeError(attr)

    def __call__(cls, *args, **kargs):
        if "dispatch_hook" in cls.__dict__:
            cls =  cls.dispatch_hook(*args, **kargs)
        i = cls.__new__(cls, cls.__name__, cls.__bases__, cls.__dict__)
        i.__init__(*args, **kargs)
        return i
```

Kivy

Kivy — кроссплатформенный графический фреймворк на Python, направленный на создание новейших пользовательских интерфейсов даже для приложений, работающих с сенсорными экранами. Приложения, написанные на **Kivy**, могут работать не только на традиционных платформах Linux, OS X и Windows, но и на Android, iOS и Raspberry Pi.

Метакласс **WidgetMetaclass** служит для регистрации новых виджетов:

```
class WidgetMetaclass(type):
    '''Metaclass to automatically register new widgets for the
    :class:`~kivy.factory.Factory`.
    .. warning::
        This metaclass is used by the Widget. Do not use it directly!
    '''
    def __init__(mcs, name, bases, attrs):
        super(WidgetMetaclass, mcs).__init__(name, bases, attrs)
        Factory.register(name, cls=mcs)

#: Base class used for Widget, that inherits from :class:`EventDispatcher`
WidgetBase = WidgetMetaclass('WidgetBase', (EventDispatcher, ), {})
```

Тема метапрограммирования — нетривиальная, и «с наскока» в нее не погрузиться. Но знания о работе метаклассов позволяют лаконично управлять созданием и модификацией обычных классов. При этом важно осознавать, где метакласс необходим, а где без него можно обойтись. Разобравшись в этой теме, легче понимать устройство крупных библиотек.

Чтобы закрепить материал, рекомендуем еще раз самостоятельно разобраться в примерах кода к данному уроку.

Практическое задание

Продолжение работы с проектом «Мессенджер»:

1. Реализовать метакласс **ClientVerifier**, выполняющий базовую проверку класса «Клиент» (для некоторых проверок уместно использовать модуль **dis**):
    - отсутствие вызовов **accept** и **listen** для сокетов;
    - использование сокетов для работы по **TCP**;
    - отсутствие создания сокетов на уровне классов, то есть отсутствие конструкций такого вида:

```
class Client:
    s = socket()
    ...
```

2. Реализовать метакласс **ServerVerifier**, выполняющий базовую проверку класса «Сервер»:
    - отсутствие вызовов **connect** для сокетов;
    - использование сокетов для работы по **TCP**.
3. Реализовать дескриптор для класса серверного сокета, а в нем — проверку номера порта. Это должно быть целое число (>=0). Значение порта по умолчанию равняется 7777. Дескриптор надо создать в отдельном классе. Его экземпляр добавить в пределах класса серверного сокета. Номер порта передается в экземпляр дескриптора при запуске сервера.

Дополнительные материалы

1. Пользовательские атрибуты в Python.
2. Metaprogramming with Metaclass in Python.
3. Understanding Python metaclasses.
4. PyCon 2013. David Beazley. Python 3 Metaprogramming.
5. Порядок разрешения методов в Python.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. David Beazley, Brian K. Jones. Python Cookbook. Third Edition (каталог «Дополнительные материалы»).
2. Brandon Rhodes, John Goerzen. Foundations of Python Network Programming (каталог «Дополнительные материалы»).
3. Бизли Дэвид. Python. Подробный справочник. 4-е издание (каталог «Дополнительные материалы»).
4. Лучано Ромальо.Python. К вершинам мастерства (каталог «Дополнительные материалы»).
5. Лутц Марк. Изучаем Python. 4-е издание (каталог «Дополнительные материалы»).

Приложение

Листинг 1

```
# ---------------------- Дескрипторы атрибутов -------------------------------

print(' ========== Базовый пример работы с дескриптором атрибутов ==========')

class TypedProperty:
    def __init__(self, name, type_name, default=None):
        self.name = "_" + name
        self.type = type_name
        self.default = default if default else type_name()

    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default)

    def __set__(self, instance, value):
        if not isinstance(value, self.type):
            raise TypeError("Значение должно быть типа %s" % self.type)
        setattr(instance, self.name, value)

    def __delete__(self, instance):
        raise AttributeError("Невозможно удалить атрибут")
```

```
class Foo:
    name = TypedProperty("name", str)
    num = TypedProperty("num", int, 42)


if __name__ == '__main__':
    f = Foo()
    a = f.name           # Неявно вызовет Foo.name.__get__(f, Foo)
    f.name = "Гвидо"     # Вызовет Foo.name.__set__(f, "Guido")
    del f.name           # Вызовет Foo.name.__delete__(f)
```

Листинг 2

```
# --------------------- Дескрипторы атрибутов -------------------------------

print(' ============ Примеры работы с дескрипторами атрибутов ===============')

print(' ------------ Data-дескриптор ---------------')

class DataDesc:
    ''' Data-дескриптор
    '''
    def __get__(self, obj, cls=None):
        print('  DataDesc.__get__')
        print('  ', self, obj, cls)
        return '**magic-descriptor**'

    def __set__(self, obj, value):
        print('  DataDesc.__set__')
        print('  ', self, obj, value)
        pass

    def __delete__(self, obj):
        print('  DataDesc.__delete__')
        print('  ', self, obj)
        pass


class D:
    ''' Класс с дескриптором данных
    '''
    d = DataDesc()


d_obj = D()

print('0. Содержимое d_obj.__dict__ в самом начале:', d_obj.__dict__)

print('1. Получить значение атрибута...')
# При доступе к атрибуту будет вызван метод __get__ дескриптора
x = d_obj.d
```

```
print('1. Значение атрибута (доступ через дескриптор):', x)

# Создание атрибута в словаре экземпляра класса (дескриптор)
print('2. Установить значение атрибута...')
d_obj.d = "полезное значение"
print('3. Содержимое d_obj.__dict__ после установки атрибута:', d_obj.__dict__)

x = d_obj.d
print('4. Значение атрибута (доступ через дескриптор):', x)

# Удаление атрибута из словаря экземпляра класса
print('5. Удалить атрибут...')
del d_obj.d
print('6. Содержимое d_obj.__dict__ удаления атрибута:', d_obj.__dict__)


print('7. Получить атрибут на уровне класса...')
x = D.d
print('8. Значение атрибута D.d:', x)

# Дескриптор будет заменён обычной строкой на уровне класса
print('9. Установить D.d ...')
D.d = "=A value in class="            # <<-- здесь не вызывается метод __set__

print(' == \/ Обратите внимание  \/ ==')
print('10. Значение атрибута D.d:', D.d)
print('11. Значение атрибута d_obj.d:', d_obj.d)

print()
print(' ------------ Non-data-дескриптор ---------------')

class GetonlyDesc:
    ''' Non-data дескриптор
    '''
    def __get__(self, obj, cls=None):
        return '**magic-descriptor**'


class C:
    ''' Класс с одним дескриптором
    '''
    d = GetonlyDesc()


cobj = C()

# При доступе к атрибуту будет вызван метод __get__ дескриптора
x = cobj.d
print('0. Содержимое объекта в самом начале:', cobj.__dict__)
print('1. Значение атрибута (доступ через дескриптор):', x)

# Создание атрибута в словаре экземпляра класса (дескриптор)
```

```
cobj.d = "setting a value"
x = cobj.d
print('2. Значение атрибута (доступ через __dict__):', x)
print('3. Содержимое объекта после  установки атрибута:', cobj.__dict__)

# Удаление атрибута из словаря экземпляра класса
del cobj.d
print('4. Содержимое объекта после удаления атрибута:', cobj.__dict__)

x = C.d
print('5. Значение атрибута C.d:', x)

# Дескриптор будет заменён обычной строкой на уровне класса
C.d = "setting a value on class"
print('6. Значение атрибута C.d:', C.d)
```

Листинг 3

```
---------------------- Дескрипторы атрибутов -------------------------------
#
print(' ====== Способы хранения значений при работе с дескрипторами =======')
print(' ========== 1. Хранение в атрибуте дескриптора ==============')

# Первый способ сохранить данные - просто в атрибуте объекта дескриптора.

class Grade:
    def __init__(self):
        self._value = 0

    def __get__(self, instance, instance_type):
        return self._value

    def __set__(self, instance, value):
        if not (1 <= value <= 5):
            raise ValueError("Оценка должна быть от 1 до 5")
        self._value = value


class Exam():
    ''' Класс Экзамен.
        Для простоты хранит только оценку за экзамен.
    '''
    grade = Grade()

# Но не стоит забывать, что при таком подходе
# данные будут сохранены на уровне атрибута класса Экзамен!!!
# Т.е. будут общими для всех экземпляров класса Экзамен.

# Для демонстрации создадим два Экзамена:
math_exam = Exam()
math_exam.grade = 3
```

```
language_exam = Exam()
language_exam.grade = 5

print("  Проверим результаты: ")
print("Первый экзамен ", math_exam.grade, " - верно?")
print("Второй экзамен ", language_exam.grade, " - верно?")

print('Потому что... ')
print('math_exam.grade is language_exam.grade =', math_exam.grade is
language_exam.grade)

print()

#
================================================================================
print()
print(' ====== 2. Хранение данных в отдельном словаре дескриптора =======')
print('* Внимание! Хранение данных в обычном dict будет приводить к утечкам
памяти! *')

class Grade:
    def __init__(self):
        self._values = {}

    def __get__(self, instance, instance_type):
        if instance is None: return self
        return self._values.get(instance, 0)

    def __set__(self, instance, value):
        if not (1 <= value <= 5):
            raise ValueError("Оценка должна быть от 1 до 5")
        self._values[instance] = value


# Хотя данное решение достаточно простое и полноценно работает,
# оно будет приводить к утечкам памяти!
# Словарь _values будет хранить ссылку на каждый внешний экземпляр класса,
# который когда-либо передавался в метод __set__.
# Это приведет к тому, что счётчик ссылок у внешних экземпляров никогда не
будет равен нулю,
# и сборщик мусора никогда не выполнит свою работу.

print()
print(' Вместо обычного dict нужно использовать класс
weakref.WeakKeyDictionary')

from weakref import WeakKeyDictionary

# ------------------------------------------------------------
# Модуль weakref обеспечивает поддержку слабых ссылок.
# В обычном случае сохранение ссылки на объект приводит к увеличению счетчика
```

```
ссылок,
# что препятствует уничтожению объекта, пока значение счетчика не достигнет
нуля.
# Слабая ссылка позволяет обращаться к объекту, не увеличивая его счетчик
ссылок.
#
--------------------------------------------------------------------------------
--------------------
# Класс WeakKeyDictionary([dict]) создает словарь, в котором ключи
представлены слабыми ссылками.
# Когда количество обычных ссылок на объект ключа становится равным нулю,
# соответствующий элемент словаря автоматически удаляется.
# В необязательном аргументе dict передается словарь, элементы которого
добавляются
# в возвращаемый объект типа WeakKeyDictionary.
# Слабые ссылки могут создаваться только для объектов определенных типов,
поэтому
# существует большое число ограничений на допустимые типы объектов ключей.
# В частности, встроенные строки НЕ МОГУТ использоваться в качестве ключей со
слабыми ссылками.
# Однако экземпляры пользовательских классов, объявляющих метод __hash__(),
могут играть роль ключей.
# Экземпляры класса WeakKeyDictionary имеют два дополнительных метода,
iterkeyrefs() и keyrefs(),
# которые возвращают слабые ссылки на ключи.

class Grade:
    def __init__(self):
        self._values = WeakKeyDictionary()

    def __get__(self, instance, instance_type):
        if instance is None: return self
        return self._values.get(instance, 0)

    def __set__(self, instance, value):
        if not (1 <= value <= 5):
            raise ValueError("Оценка должна быть от 1 до 5")
        self._values[instance] = value


class Exam():
    ''' Класс Экзамен.
        Для простоты хранит только оценку за экзамен
    '''
    grade = Grade()


# Для демонстрации создадим два Экзамена:
math_exam = Exam()
math_exam.grade = 3

language_exam = Exam()
```

```
language_exam.grade = 5

print("  Проверим результаты: ")
print("Первый экзамен ", math_exam.grade, " - верно?")
print("Второй экзамен ", language_exam.grade, " - верно?")

print()

# Недостаток конкретно данного решения - в одном классе нельзя сохранять
данные дескрипторов одного типа.
# Т.е., например, сделать экзамен с несколькими оценками.


print(' ====== Хранение в __dict__ экземпляра внешнего класса =======')

# Такой подход, помимо прочего, позволяет в одном внешнем классе
# создавать несколько объектов-дескрипторов одного класса.

class Grade:
    def __init__(self, name):
        # Для данного подхода необходимо сформировать отдельное имя атрибута,
        # иначе при совпадении имени name и имени дескриптора
        # создаваемый атрибут перезапишет объект дескриптора в данном
экземпляре
        self.name = '_' + name

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return "*{}*".format(getattr(instance, self.name))

    def __set__(self, instance, value):
        if not (1 <= value <= 100):
            raise ValueError("Балл ЕГЭ должен быть от 1 до 100")
        setattr(instance, self.name, value)


class ExamEGE():
    ''' Комплексный экзамен, на котором оцениваются разные критерии.
    '''
    # Для обновлённого Grade нужно обновить и создание атрибутов, добавив
строковые имена.
    # Строковые имена могут не совпадать с именами атрибутов.
    math_grade = Grade('math_grade')
    writing_grade = Grade('writing_grade')
    science_grade = Grade('science')


# Проверим обновлённый дескриптор Оценку и объекты Экзамены.
first_exam = ExamEGE()
first_exam.writing_grade = 3
first_exam.math_grade = 4
```

```
print("Содержимое first_exam.__dict__:")
print(' ', first_exam.__dict__)

second_exam = ExamEGE()
second_exam.writing_grade = 2
second_exam.science_grade = 5

print()
print("  Проверим результаты: ")
print("Первый экзамен ", first_exam.writing_grade, first_exam.math_grade, " -
верно?")
print("Второй экзамен ", second_exam.writing_grade, second_exam.science_grade, "
- верно?")
```

Листинг 4

```
# -- Доступ к атрибутам (методы __getattr__, __getattribute__, __setattr__) --

print(' ================= Контроль доступа к атрибутам ======================')

print(' ------------ Порядок обращения к специальным методам  ------------')
print(' ----- 1. __getattr__ ')

# Метод __getattr__ вызывается, когда атрибут не найден в __dict__ экземпляра
класса
class LazyDB(object):
    def __init__(self):
        self.exists = 5

    def __getattr__(self, name):
        print(' LazyDB.__getattr__(%s)' % name)
        value = 'Super %s' % name
        setattr(self, name, value)
        return value


data = LazyDB()
print('data.__dict__ до обращения к атрибутам:', data.__dict__)
print('Атрибут exists:', data.exists)
print('Атрибут foo: ', data.foo)
print('data.__dict__ после обращения к атрибутам: ', data.__dict__)
print()

print(' ----- 2. __getattribute__ ')
# Метод __getattribute__ вызывается при обращении к атрибуту объекта,
# даже если атрибут не существует в словаре __dict__ экземпляра класса;
# вызывается при обращении к функциям getattr(), hasattr().

class ValidatingDB:
    def __init__(self):
```

```
        self.exists = 5

    def __getattribute__(self, name):
        print(' ValidatingDB.__getattribute__(%s)' % name)
        try:
            return super().__getattribute__(name)
        except AttributeError:
            value = 'Puper %s' % name
            setattr(self, name, value)
        return value


data = ValidatingDB()
print('Атрибут exists:', data.exists)
print('Атрибут foo: ', data.foo)
print('Атрибут foo: ', data.foo)
print('Атрибут zero через getattr: ', getattr(data, 'zero'))
print()


print(' ----- 3. __getattr__ + __getattribute__ ')

class ValidatingDB:
    def __init__(self):
        self.exists = 5

    def __getattr__(self, name):
        print(' ValidatingDB.__getattr__(%s)' % name)
        value = 'Super %s' % name
        setattr(self, name, value)
        return value

    def __getattribute__(self, name):
        print(' ValidatingDB.__getattribute__(%s)' % name)
        return super().__getattribute__(name)


data = ValidatingDB()
print('Атрибут exists:', data.exists)
print('Атрибут foo: ', data.foo)
print('Снова атрибут foo: ', data.foo)
print('Есть ли атрибут zoom в объекте:', hasattr(data, 'zoom'))
print('Атрибут face в объекте, доступ через getattr:', getattr(data, 'face'))
print()


print(' ----- 4. __setattr__ ')
# Метод __setattr__ вызывается, когда атрибут назначается экземпляру класса
# (в т.ч. при обращении к функции `setattr`).

class SavingDB:
    def __setattr__(self, name, value):
```

```
        print(' SavingDB.__setattr__(%s, %r)' % (name, value))
        # Сохранение данных в БД
        # ...
        super().__setattr__(name, value)


data = SavingDB()
print('data.__dict__ до установки атрибута: ', data.__dict__)
data.foo = 5
print('data.__dict__ после установки атрибута: ', data.__dict__)
data.foo = 7
print('data.__dict__ в итоге:', data.__dict__)
print()


print(' -------- Проблема рекурсивного обращения к атрибутам ----------------')
print('       (раскомментируйте строки кода для создания экземпляра)')

class BrokenDictionaryDB:
    def __init__(self, data):
        self._data = {}

    def __getattribute__(self, name):
        print('BrokenDictionaryDB.__getattribute__(%s)' % name)
        return self._data[name]

# Раскомментируйте строки ниже, чтобы увидеть проблему рекурсии:
# data = BrokenDictionaryDB({'foo': 3})
# print(data.foo)
print()


print(' -------- Чтобы избежать рекурсии, используйте объект super -----------')
# Для решения проблемы рекурсивного обращения к атрибутам используйте объект
super:

class DictionaryDB:
    def __init__(self, data):
        self._data = data

    def __getattribute__(self, name):
        print('DictionaryDB.__getattribute__(%s)' % name)
        data_dict = super().__getattribute__('_data')
        return data_dict[name]


data = DictionaryDB({'foo': 'This is the right way!'})
print(data.foo)
print()

print(' ---------------- __getattribute__ и дескриптор ---------------- ')
# В дополнение рассмотрим совместное использование
```

```
# методов контроля доступа к атрибутам и дескриптора атрибутов

class Grade:
    def __init__(self, name):
        self.name = '_' + name

    def __get__(self, instance, instance_type):
        print(' Grade.__get__')
        if instance is None:
            return self
        return "*{}*".format(getattr(instance, self.name))

    def __set__(self, instance, value):
        print(' Grade.__set__')
        if not (1 <= value <= 100):
            raise ValueError("Балл ЕГЭ должен быть от 1 до 100")
        setattr(instance, self.name, value)


class Exam:
    grade = Grade('grade')

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

    def __getattribute__(self, name):
        print(' Exam.__getattribute__(%s)' % name)
        return super().__getattribute__(name)


data = Exam('Математика')
print('Задаём количество баллов за экзамен...')
data.grade = 95

print('Выводим это количество баллов...')
print(data.grade)

print('А что такое Exam.grade?')
print(Exam.grade)
```

Листинг 5

```
# -------------------------- Метаклассы -------------------------------------

print(' ----- Демонстрация работы с методом __init__ метакласса -----')

class DocMeta(type):
    ''' Метакласс, проверяющий наличие строк документации в подконтрольном
классе
    '''
    def __init__(self, clsname, bases, clsdict):
```

```
        # К моменту начала работы метода __init__ метакласса
        # словарь атрибутов контролируемого класса уже сформирован.
        for key, value in clsdict.items():
            # Пропустить специальные и частные методы
            if key.startswith("__"): continue

            # Пропустить любые невызываемые объекты
            if not hasattr(value, "__call__"): continue

            # Проверить наличие строки документирования
            if not getattr(value, "__doc__"):
                raise TypeError("Метод %s должен иметь строку документации" %
key)

        type.__init__(self, clsname, bases, clsdict)


class Documented(metaclass=DocMeta):
    ''' Базовый класс для документированных классов. Можно оставить пустым.
    '''
    pass


# Дочерний класс получает метакласс "в нагрузку" от родительского класса
class Foo(Documented):
    ''' Прикладной пользовательский класс.
    '''
    def spam(self, a, b):
        ''' Метод spam делает кое-что '''
        pass

    def boo(self):
        print('A little problem')
```

Листинг 6

```
class TypedProperty_v1:
    def __init__(self, name, type_name, default=None):
        self.name = "_" + name
        self.type = type_name
        self.default = default if default else type_name()

    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default)

    def __set__(self, instance, value):
        if not isinstance(value, self.type):
            raise TypeError("Значение должно быть типа %s" % self.type)
        setattr(instance, self.name, value)

    def __delete__(self, instance):
```

```
        raise AttributeError("Невозможно удалить атрибут")


class Foo:
    name = TypedProperty_v1("name", str)
    num = TypedProperty_v1("num", int, 42)


if __name__ == '__main__':
    f = Foo()
    a = f.name           # Неявно вызовет Foo.name.__get__(f,Foo)
    f.name = "Гвидо"     # Вызовет Foo.name.__set__(f,"Guido")
    del f.name           # Вызовет Foo.name.__delete__(f)


class TypedProperty_v2:
    ''' Дескриптор атрибутов, контролирующий принадлежность указанному типу
    '''
    def __init__(self, type_name, default=None):
        self.name = None
        self.type = type_name
        if default:
            self.default = default
        else:
            self.default = type_name()

    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default)

    def __set__(self, instance, value):
        if not isinstance(value, self.type):
            raise TypeError("Значение должно быть типа %s" % self.type)
        setattr(instance, self.name, value)

    def __delete__(self, instance):
        raise AttributeError("Невозможно удалить атрибут")
```

Листинг 7

```
# -------------------------- Метаклассы ----------------------------

from descriptor import TypedProperty_v2

print(' ------- Демонстрация работы с методом __new__ метакласса ----------')

class TypedMeta(type):
    ''' Метакласс, создающий список __slots__,
        который будет содержать только атрибуты типа TypedProperty
    '''
    def __new__(cls, clsname, bases, clsdict):
        slots = [ ]
```

```
        for key, value in clsdict.items():
            if isinstance(value, TypedProperty_v1):
                value.name = "_" + key
                slots.append(value.name)
        clsdict['__slots__'] = slots
        return type.__new__(cls, clsname, bases, clsdict)


class Typed(metaclass=TypedMeta):
    ''' Базовый класс для объектов, определяемых пользователем.
        Можно просто оставить пустым. Вся "магия" делается метаклассом.
    '''
    pass


# Дочерний класс получает в "наследство" также и метакласс
class Foo(Typed):
    ''' Пользовательский класс с контролируемыми атрибутами
    '''
    name = TypedProperty_v1('name', str)
    num = TypedProperty_v1('num', int, 42)
    zzz = 15


foo = Foo()

# Попытка добавить новый атрибут объекту приведёт к исключению:
# foo.xxx = 13          # <- раскомментируйте строку, чтобы увидеть исключение
# print(foo.xxx)

# Атрибут, который отсутствует в __slots__ становится read-only атрибутом
print(foo.zzz)
# foo.zzz = 77          # <- раскомментируйте строку, чтобы увидеть исключение

# При этом "легитимные" атрибуты типа TypedProperty_v1
# ведут себя обычным для атрибутов образом...
foo.num = 99
foo.name = 'Bigno!'
print(foo.num, foo.name)

# ... А также имеют дополнительные преимущества:
foo.num = 'str'
foo.name = 17
```

Листинг 8

```
# ------------------------- Метаклассы ----------------------------

print(' ---- Шаблон Одиночка с использованием __call__ метакласса ---- ')

# Объявляем метакласс, который будет контролировать создание нового класса
```

```
class Singleton(type):

    def __init__(self, *args, **kwargs):
        print('__init__ in Metaclass. ', self, args, kwargs)
        self.__instance = None
        super().__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        print('__call__ in Metaclass')
        print(' ', self, args, kwargs)
        if self.__instance is None:
            self.__instance = super().__call__(*args, **kwargs)
            return self.__instance
        else:
            return self.__instance


class BaseA(metaclass=Singleton):
    def __init__(self):
        print('Class BaseA')


class BaseB(metaclass=Singleton):
    def __init__(self):
        print('Class BaseB')


a_1 = BaseA()
a_2 = BaseA()

b_1 = BaseB()
b_2 = BaseB()

print('a_1 is a_2 - ', a_1 is a_2)
print('b_1 is b_2 - ', b_1 is b_2)
print('a_1 is b_1 - ', a_1 is b_1)
print('a_2 is b_2 - ', a_2 is b_2)
print('a_1 is b_2 - ', a_1 is b_2)
print('a_2 is b_1 - ', a_2 is b_1)
```

Листинг 9

```
# -------------------------- Метаклассы ----------------------------

# Пример использования метода метакласса __prepare__
# Пример актуален для Python до версии 3.6 -
# в Python 3.6 __prepare__ по умолчанию возвращает OrderedDict

import collections
from descriptor import TypedProperty_v2
```

```
print(' ------ Демонстрация работы с методом __prepare__ метакласса --------')


class EntityMeta(type):
    """ Метакласс для прикладных классов с контролируемыми полями
    """
    # Метод __prepare__ вызывается до чтения тела пользовательского класса,
    # возвращает объект-отображение (dict-like) для хранения атрибутов класса
    @classmethod
    def __prepare__(cls, name, bases):
        # Атрибуты класса будут теперь храниться в экземпляре OrderedDict
        return collections.OrderedDict()

    def __init__(cls, name, bases, clsdict):
        super().__init__(name, bases, clsdict)

        # Атрибут _fieid_names создаётся в конструируемом классе
        cls._field_names = []
        for key, attr in clsdict.items():
            if isinstance(attr, TypedProperty_v2):
                # Заполняем список только атрибутами типа TypedProperty
                type_name = type(attr).__name__
                attr.name = '_{}_{}'.format(type_name, key)
                cls._field_names.append((key, attr.name))


class Entity(metaclass=EntityMeta):
    """ Прикладной класс с контролируемыми полями
    """
    @classmethod
    def field_names(cls):
        ''' Просто возвращает поля в порядке добавления '''
        for name in cls._field_names:
            yield name


class LineItem(Entity):
    ''' Класс-пример со множеством атрибутов
    '''
    reading_short = TypedProperty_v2(int, 13)
    description_very_long = TypedProperty_v2(str, 'Simple Line')
    here_are_numerous_simple = TypedProperty_v2(int, 1)
    price_ho_ho = TypedProperty_v2(float, 19.99)
    after_the_introduction = TypedProperty_v2(int, 73)
    await_it_is_not_a_weight = TypedProperty_v2(int, 3)


print('Атрибуты пользовательского класса: ')
# Получим все имена атрибутов класса
for field in LineItem.field_names():
    print(field)
```

Листинг 10

```
# ------------------------- Метаклассы ----------------------------

# Классы в Python - это тоже объекты. Созданием классов заведуют метаклассы.
# В обычном случае созданием классов занимается функция type

print(' ------- Создание класса функцией type ----------')

# Используя функцию type можно вот так создать новый класс:
Spam = type("Spam", (object,), {"name":'Python', "age":25})
print('Новый класс, созданный функцией type:', Spam)
print('Содержимое класса:', dir(Spam))
print('Атрибуты класса:', Spam.__dict__)
print()

#
# --------------------------------------------------------------------------------

print(' ------- Демонстрация очерёдности вызова методов метакласса ----------')

class Meta(type):
    @classmethod
    def __prepare__(cls, clsname, bases):
        print('>> Meta. __prepare__', cls, clsname, bases)
        return dict()

    def __new__(cls, clsname, bases, clsdict):
        print('>> Meta.__new__', cls, clsname, bases, clsdict)
        return type.__new__(cls, clsname, bases, clsdict)

    def __init__(self, *args, **kwargs):
        print('>> Meta.__init__', args, kwargs)
        super().__init__(*args, **kwargs)

    def __call__(cls, *args, **kwargs):
        print('>> Meta.__call__', args, kwargs)
        return super().__call__(*args, **kwargs)

print('Перед созданием пользовательского класса Z')

class Z(metaclass=Meta):
    print('> Class Z. Начало тела класса')

    def __init__(self, x):
        print('> Z.__init__', x)
        self.x = x

    print('> Class Z. Конец тела класса')


print('\nПеред созданием экземпляра класса Z')
```

```
zorro = Z(13)
print(zorro)
```