Урок 9. Дескрипторы и метаклассы

Введение в тестирование. Оператор assert. Модульное тестирование и модуль unittest.

[Краткий экскурс в ООП-1](#_2et92p0)

[Дескрипторы атрибутов](#_tyjcwt)

[Протокол дескриптора](#_1t3h5sf)

[Типы дескрипторов](#_4d34og8)

[Хранение значений атрибутов](#_v74jltjqup73)

[Поиск атрибутов](#_17dp8vu)

[Чтение атрибута](#_3rdcrjn)

[Запись атрибута](#_26in1rg)

[Удаление атрибута](#_lnxbz9)

[Доступ к атрибутам](#_35nkun2)

[Метаклассы](#_xtkdteriecjp)

[Знакомство с метаклассами](#_2jxsxqh)

[Методы \_\_new\_\_, \_\_init\_\_, \_\_call\_\_](#_z337ya)

[Метод \_\_prepare\_\_](#_3j2qqm3)

[Примеры использования метаклассов](#_jqggmbv9pffu)

[Django](#_2xcytpi)

[SQLAlchemy](#_1ci93xb)

[Scapy](#_gc4vogqmjt4)

[Kivy](#_2bn6wsx)

[Практическое задание](#_3as4poj)

[Дополнительные материалы](#_1pxezwc)

[Используемая литература](#_49x2ik5)

[Приложение](#_k06v4uz041a8)

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, например, **@property;**
* 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»).

# 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.

## 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) |

1. **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 |

1. **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\_\_**.

1. 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.

1. 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\_\_**.

1. 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\_\_**.

1. 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\_\_**.

1. 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 |

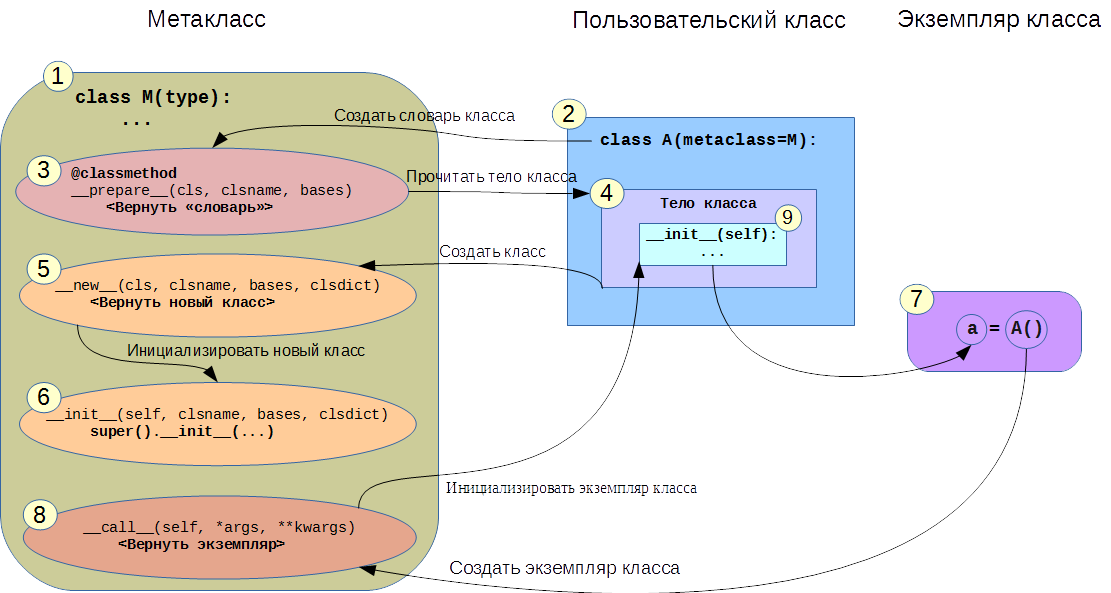
После простых модификаций, показанных в примере, можно обойти поля типа **TypedProperty\_v2** любого подкласса **Entity**, воспользовавшись методом класса **field\_names**:

|  |
| --- |
| 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**.



## 