Что такое метаклассы в Python?

**ОПУБЛИКОВАНО**[**18.03.2021**](https://pythonist.ru/chto-takoe-metaklassy-v-python/)[**ВОПРОСЫ С СОБЕСЕДОВАНИЯ**](https://pythonist.ru/category/voprosy-s-sobesedovaniya-python/)

Как обычно проходит собеседования на позицию разработчика Python? Обычно одним из первых вопросов будет просьба рассказать о типа данных (или составных типах данных) в Python. Потом через несколько других общих вопросов разговор обязательно перейдет к теме дескрипторов и метаклассов в Python. И хотя это такие вещи, которые в реальной практике редко когда приходится использовать, каждый разработчик должен иметь хотя бы общее представление о них, — [пишет](https://webdevblog.ru/chto-takoe-metaklassy-v-python/) сайт webdevblog.ru.

В большинстве языков классы — это просто фрагменты кода, описывающие, как создать объект. В общем случае это верно и для Python:

>>> **class** ObjectCreator(object):

... **pass**

...

>>> my\_object = ObjectCreator()

>>> print(my\_object)

<\_\_main\_\_.ObjectCreator object at 0x8974f2c>

Но в Python классы — это так же еще и нечто большее. В Python все является объектами в том числе и классы. Как только вы используете ключевое слово class, Python выполняет его и создает OBJECT.

Давайте сейчас создадим в памяти объект с именем «ObjectCreator».

>>> **class** ObjectCreator(object):

... **pass**

...

Этот объект (то есть класс) сам по себе так же может создавать объекты (точнее его экземпляры), и поэтому он является классом.

Но все же это объект, а значит:

* вы можете присвоить его переменной
* вы можете скопировать его
* вы можете добавить к нему атрибуты
* вы можете передать его как параметр функции

Например:

>>> print(ObjectCreator) # вы можете отобразить класс, потому что это объект

<**class** '\_\_main\_\_.ObjectCreator'>

>>> **def** echo(o):

... print(o)

...

>>> echo(ObjectCreator) # вы можете передать класс в качестве параметра

<**class** '\_\_main\_\_.ObjectCreator'>

>>> print(hasattr(ObjectCreator, 'new\_attribute'))

**False**

>>> ObjectCreator.new\_attribute = 'foo' # вы можете добавлять атрибуты в класс

>>> print(hasattr(ObjectCreator, 'new\_attribute'))

**True**

>>> print(ObjectCreator.new\_attribute)

foo

>>> ObjectCreatorMirror = ObjectCreator # вы можете присвоить класс переменной

>>> print(ObjectCreatorMirror.new\_attribute)

foo

>>> print(ObjectCreatorMirror())

<\_\_main\_\_.ObjectCreator object at 0x8997b4c>

В общем в Python вы можете делать с классом все, что можно делать с объектом. В том числе создавать класс на лету.

**Динамическое создание классов**

Поскольку классы являются объектами, вы можете создавать их на лету, как и любой объект.

Во-первых, вы можете создать класс в функции, используя ключевое слово class:

>>> **def** choose\_class(name):

... **if** name == 'foo':

... **class** Foo(object):

... **pass**

... **return** Foo # возвращает class, а не его экземпляр

... **else**:

... **class** Bar(object):

... **pass**

... **return** Bar

...

>>> MyClass = choose\_class('foo')

>>> print(MyClass) # функция вернула класс, а не его экземпляр

<**class** '\_\_main\_\_.Foo'>

>>> print(MyClass()) # you can create an object from this class

<\_\_main\_\_.Foo object at 0x89c6d4c>

Но это не так динамично, так как вам все равно придется писать весь класс самостоятельно. Поскольку классы являются объектами, они должны быть чем-то порождены. Когда вы используете ключевое слово class, Python создает этот объект автоматически. Но, как и в большинстве случаев в Python, он позволяет делать это и вручную. Помните функцию type? Старая добрая функция, которая позволяет [узнать, к какому типу относится объект](https://pythonist.ru/proverka-tipov-v-python/):

>>> print(type(1))

<type 'int'>

>>> print(type("1"))

<type 'str'>

>>> print(type(ObjectCreator))

<type 'type'>

>>> print(type(ObjectCreator()))

<**class** '\_\_main\_\_.ObjectCreator'>

Что ж, у type есть и совершенно другие способности, он также может создавать классы на лету. type может принимать описание класса как параметры и возвращать класс. (Я знаю, это звучит глупо, что одна и та же функция может иметь два совершенно разных использования в зависимости от параметров, которые вы ей передаете. Это проблема из-за обратной совместимости в Python).

Итак, type работает следующим образом:

type(name, bases, attrs)

Где:

* name: имя класса
* bases: кортеж родительского класса (для экземпляра, может быть пустым)
* attrs: словарь, содержащий имена и значения атрибутов

Таким образом

>>> **class** MyShinyClass(object):

... **pass**

можно создать вручную:

>>> MyShinyClass = type('MyShinyClass', (), {}) # returns a class object

>>> print(MyShinyClass)

<**class** '\_\_main\_\_.MyShinyClass'>

>>> print(MyShinyClass()) # создаем экземпляр класса

<\_\_main\_\_.MyShinyClass object at 0x8997cec>

Вы заметите, что мы используем «MyShinyClass» как имя класса и как переменную для хранения ссылки на класс. Они могут быть разными, но нет причин усложнять ситуацию. type принимает словарь для определения атрибутов класса.

Так:

>>> **class** Foo(object):

... bar = **True**

Может быть переведен на:

>>> Foo = type('Foo', (), {'bar':**True**})

И используется как обычный класс:

>>> print(Foo)

<**class** '\_\_main\_\_.Foo'>

>>> print(Foo.bar)

**True**

>>> f = Foo()

>>> print(f)

<\_\_main\_\_.Foo object at 0x8a9b84c>

>>> print(f.bar)

**True**

И, конечно, вы можете унаследоваться от него. То есть такое:

>>> **class** FooChild(Foo):

... **pass**

можно переделать в такое:

>>> FooChild = type('FooChild', (Foo,), {})

>>> print(FooChild)

<**class** '\_\_main\_\_.FooChild'>

>>> print(FooChild.bar) # bar унаследован от Foo

**True**

В конце концов, вы захотите добавить методы в свой класс. Просто определите функцию с соответствующей подписью и назначьте ее как атрибут.

>>> **def** echo\_bar(self):

... print(self.bar)

...

>>> FooChild = type('FooChild', (Foo,), {'echo\_bar': echo\_bar})

>>> hasattr(Foo, 'echo\_bar')

**False**

>>> hasattr(FooChild, 'echo\_bar')

**True**

>>> my\_foo = FooChild()

>>> my\_foo.echo\_bar()

**True**

И вы можете добавить еще больше методов после динамического создания класса, точно так же, как можно добавить методы к обычно создаваемому объекту класса.

>>> **def** echo\_bar\_more(self):

... print('yet another method')

...

>>> FooChild.echo\_bar\_more = echo\_bar\_more

>>> hasattr(FooChild, 'echo\_bar\_more')

**True**

Вы видите, к чему мы идем: **в Python классы — это объекты, и вы можете создавать классы на лету, динамически**. Это то, что Python делает, когда вы используете ключевое слово class, и делает это с помощью метакласса.

**Что такое метаклассы**

Метаклассы — это «материал», который создает классы.

Вы определяете классы для создания объектов, верно? Но мы узнали, что классы Python — это объекты. Что ж, **метаклассы создают эти объекты**. Это классы классов, и их можно изобразить так:

MyClass = MetaClass()

my\_object = MyClass()

Мы уже обсудили, что этот type позволяет делать что-то вроде этого:

MyClass = type('MyClass', (), {})

Это потому, что функции type на самом деле является метаклассом. type — это метакласс, который Python использует для создания всех классов за кулисами.

Теперь вы можете спросить, почему type пишется с маленькой буквы?

Что ж, я думаю, это вопрос согласованности со str — классом, который создает строковые объекты, и int — классом, который создает целочисленные объекты. type — это просто класс, который создает объекты class.

Вы можете убедиться в этом, проверив атрибут \_\_class\_\_.

Все является объектом в Python. Сюда входят целые числа, строки, функции и классы. Все они объекты. И все они созданы из класса:

>>> age = 35

>>> age.\_\_class\_\_

<type 'int'>

>>> name = 'bob'

>>> name.\_\_class\_\_

<type 'str'>

>>> **def** foo(): **pass**

>>> foo.\_\_class\_\_

<type 'function'>

>>> **class** Bar(object): **pass**

>>> b = Bar()

>>> b.\_\_class\_\_

<**class** '\_\_main\_\_.Bar'>

Теперь, что такое \_\_class\_\_ любого \_\_class\_\_?

>>> age.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> name.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> foo.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> b.\_\_class\_\_.\_\_class\_\_

<type 'type'>

Итак, метакласс — это просто материал, который создает объекты class. Если хотите, можете назвать это «фабрикой классов». type — это встроенный метакласс, который использует Python, но, конечно, и вы можете создать свой собственный метакласс.

**Атрибут**[**\_\_metaclass\_\_**](http://docs.python.org/2/reference/datamodel.html?highlight=__metaclass__#__metaclass__)

В Python 2 вы можете добавить атрибут \_\_metaclass\_\_ при описании класса:

**class** Foo(object):

\_\_metaclass\_\_ = something...

[...]

Если вы это сделаете, Python будет использовать этот метакласс для создания класса Foo.

Сначала вы пишете class Foo(object), но объект класса Foo еще не создан в памяти. Python будет искать \_\_metaclass\_\_ в определении класса. Если он его найдет, он будет использовать его для создания класса объекта Foo. Если не найдет, он будет использовать type для создания класса. Прочтите это несколько раз.

Когда вы делаете это:

**class** Foo(Bar):

**pass**

Python делает следующее:

Есть ли атрибут \_\_metaclass\_\_ в Foo ?

Если да, то будет создан в памяти объект класса с именем Foo, используя то, что находится в \_\_metaclass\_\_.

Если Python не может найти \_\_metaclass\_\_, он будет искать \_\_metaclass\_\_ на уровне MODULE и пытаться сделать то же самое (но только для классов, которые ничего не наследуют, в основном классов старого стиля).

Затем, если он вообще не может найти какой-либо \_\_metaclass\_\_, он будет использовать собственный метакласс Bar (первого родителя) (который может быть type по умолчанию) для создания объекта класса.

Примечание: атрибут \_\_metaclass\_\_ не будет унаследован, а метакласс родительского элемента (Bar .\_\_ class\_\_) — будет. Если Bar использовал атрибут \_\_metaclass\_\_, который создал Bar с помощью type() (а не type .\_\_ new \_\_ ()), подклассы не унаследуют это поведение.

Теперь главный вопрос: что можно добавить в \_\_metaclass\_\_?

**Ответ — то, что может создать класс.**

А что можно создать класс? type или что-либо, что его подклассы используют для этого.

**Метаклассы в Python 3**

В Python 3 был изменен синтаксис для установки метакласса:

**class** Foo(object, metaclass=something):

...

то есть атрибут \_\_metaclass\_\_ больше не используется в качестве аргумента ключевого слова в списке базовых классов.

Однако поведение метаклассов в основном осталось неизменным. Одна вещь, добавленная к метаклассам в Python 3, заключается в том, что вы также можете передавать атрибуты как ключевые слова-аргументы в метакласс, например:

**class** Foo(object, metaclass=something, kwarg1=value1, kwarg2=value2):

**Пользовательские метаклассы**

**Основная цель метакласса — автоматически изменять класс при его создании.**

Обычно вы делаете это для API, когда хотите создать классы, соответствующие текущему контексту.

Представьте себе глупый пример, в котором вы решили, что атрибуты всех классов в вашем модуле должны быть написаны в верхнем регистре. Есть несколько способов сделать это, но один из них — установить \_\_metaclass\_\_ на уровне модуля.

Таким образом, все классы этого модуля будут созданы с использованием этого метакласса, и нам просто нужно указать метаклассу трансформировать все атрибуты в верхний регистр.

К счастью, \_\_metaclass\_\_ на самом деле может быть любым вызываемым, он не обязательно должен быть формальным классом (то есть что-то с ‘class’ в имени не обязательно должно быть классом).

Итак, мы начнем с простого примера, используя функцию.

# the metaclass will automatically get passed the same argument

# that you usually pass to `type`

**def** upper\_attr(future\_class\_name, future\_class\_parents, future\_class\_attrs):

**"""**

**Return a class object, with the list of its attribute turned**

**into uppercase.**

**"""**

# pick up any attribute that doesn't start with '\_\_' and uppercase it

uppercase\_attrs = {

attr **if** attr.startswith("\_\_") **else** attr.upper(): v

**for** attr, v **in** future\_class\_attrs.items()

}

# let `type` do the class creation

**return** type(future\_class\_name, future\_class\_parents, uppercase\_attrs)

\_\_metaclass\_\_ = upper\_attr # this will affect all classes in the module

**class** Foo(): # global \_\_metaclass\_\_ won't work with "object" though

# but we can define \_\_metaclass\_\_ here instead to affect only this class

# and this will work with "object" children

bar = 'bip'

Давайте проверим:

>>> hasattr(Foo, 'bar')

**False**

>>> hasattr(Foo, 'BAR')

**True**

>>> Foo.BAR

'bip

Теперь сделаем то же самое, но с использованием реального класса для метакласса:

# remember that `type` is actually a class like `str` and `int`

# so you can inherit from it

**class** UpperAttrMetaclass(type):

# \_\_new\_\_ is the method called before \_\_init\_\_

# it's the method that creates the object and returns it

# while \_\_init\_\_ just initializes the object passed as parameter

# you rarely use \_\_new\_\_, except when you want to control how the object

# is created.

# here the created object is the class, and we want to customize it

# so we override \_\_new\_\_

# you can do some stuff in \_\_init\_\_ too if you wish

# some advanced use involves overriding \_\_call\_\_ as well, but we won't

# see this

**def** \_\_new\_\_(upperattr\_metaclass, future\_class\_name,

future\_class\_parents, future\_class\_attrs):

uppercase\_attrs = {

attr **if** attr.startswith("\_\_") **else** attr.upper(): v

**for** attr, v **in** future\_class\_attrs.items()

}

**return** type(future\_class\_name, future\_class\_parents, uppercase\_attrs)

Давайте перепишем приведенное выше, но с более короткими и более реалистичными именами переменных, теперь, когда мы знаем, что они означают:

**class** UpperAttrMetaclass(type):

**def** \_\_new\_\_(cls, clsname, bases, attrs):

uppercase\_attrs = {

attr **if** attr.startswith("\_\_") **else** attr.upper(): v

**for** attr, v **in** attrs.items()

}

**return** type(clsname, bases, uppercase\_attrs)

Возможно, вы заметили дополнительный аргумент cls. В этом нет ничего особенного: \_\_new\_\_ всегда получает класс, в котором он определен, в качестве первого параметра. Точно так же, как у вас есть self для обычных методов, которые получают экземпляр в качестве первого параметра, или определяющий класс для методов класса.

Но это неправильное ООП. Мы вызываем type напрямую и не переопределяем и не вызываем родительский \_\_new\_\_. Давайте сделаем это вместо этого:

**class** UpperAttrMetaclass(type):

**def** \_\_new\_\_(cls, clsname, bases, attrs):

uppercase\_attrs = {

attr **if** attr.startswith("\_\_") **else** attr.upper(): v

**for** attr, v **in** attrs.items()

}

**return** type.\_\_new\_\_(cls, clsname, bases, uppercase\_attrs)

Мы можем сделать его еще чище, используя super, который упростит наследование (потому что да, у вас могут быть метаклассы, унаследованые от метаклассов, унаследованые от type):

**class** UpperAttrMetaclass(type):

**def** \_\_new\_\_(cls, clsname, bases, attrs):

uppercase\_attrs = {

attr **if** attr.startswith("\_\_") **else** attr.upper(): v

**for** attr, v **in** attrs.items()

}

**return** super(UpperAttrMetaclass, cls).\_\_new\_\_(

cls, clsname, bases, uppercase\_attrs)

О, и в python 3, если вы выполните этот вызов с аргументами, например:

**class** Foo(object, metaclass=MyMetaclass, kwarg1=value1):

Это переводится в метаклассе для его использования:

**class** MyMetaclass(type):

**def** \_\_new\_\_(cls, clsname, bases, dct, kwargs1=default):

...

Ну собственно вот и все. Больше нечего сказать о метаклассах.

Причина сложности кода с использованием метаклассов заключается не в метаклассах, а в том, что вы обычно используете метаклассы для выполнения извращенных вещей, полагаясь на самоанализ, манипулирование наследованием, переменные, такие как \_\_dict\_\_ и т. д.

На самом деле, метаклассы особенно полезны для черной магии и, следовательно, для сложных вещей. Но сами по себе они просты. Все что они делают:

* перехватывают создание класса
* изменяют класс
* возвращают измененный класс

**Зачем использовать классы метаклассов вместо функций?**

Поскольку \_\_metaclass\_\_ может принимать любые вызываемые объекты, зачем использовать класс, если он явно более сложен?

Для этого есть несколько причин:

* Ваши намерения в этом случае будут более ясно. Когда вы читаете UpperAttrMetaclass (type), вам проще понять, что будет дальше
* Вы можете использовать ООП. Метакласс может наследоваться от метакласса, переопределять родительские методы. Метаклассы могут даже использовать другие метаклассы.
* Подклассы класса будут экземплярами его метакласса, если вы указали класс метакласса, не с помощью функции метакласса.
* Вы можете лучше структурировать свой код. Вы никогда не используете метаклассы для чего-то столь же тривиального, как приведенный выше пример. Обычно они используются для чего-то сложного. Возможность создавать несколько методов и группировать их в один класс очень полезна для облегчения чтения кода.
* Вы можете подключиться к\_\_new\_\_, \_\_init\_\_ и \_\_call\_\_. Это позволит вам делать разные вещи. Даже если обычно вы можете делать все это в \_\_new\_\_, некоторым людям просто удобнее использовать \_\_init\_\_.
* Они называются метаклассами, черт возьми! Это должно что-то значить!

**Зачем вам использовать метаклассы?**

А теперь большой вопрос. Зачем вам использовать какую-то непонятную функцию, подверженную ошибкам?

Ну, обычно вам это не нужно:

*Метаклассы — это более глубокая магия, и 99% пользователей не должны использовать ее. Если вы задаетесь вопросом, нужны ли это вам, то скорее всего это вам не нужно (люди, которым это действительно нужно, точно знают, что это такое, и не нуждаются в объяснении почему).*

*Python Guru Tim Peters*

Основной вариант использования метакласса — создание API. Типичным примером этого является Django ORM. Это позволяет вам определить что-то вроде

**class** Person(models.Model):

name = models.CharField(max\_length=30)

age = models.IntegerField()

Но если вы используете это так:

person = Person(name='bob', age='35')

print(person.age)

Этот код не вернет объект IntegerField. Он вернет int и даже может взять его прямо из базы данных.

Это возможно, потому что models.Model определяет \_\_metaclass\_\_ и использует некоторую магию, которая превратит Person, которую вы только что определили с помощью простых операторов, в сложный крючок для поля базы данных.

Django делает что-то сложное простым, предоставляя простое API и используя метаклассы, воссоздавая код из этого API, чтобы выполнять реальную работу за кулисами.

**Заключение**

Во первых, вы знаете, что классы — это объекты, которые могут создавать экземпляры. Но и сами классы по себе являются экземплярами. Метаклассов.

>>> **class** Foo(object): **pass**

>>> id(Foo)

142630324

В Python все является объектом, и все они являются экземплярами классов или экземплярами метаклассов.

За исключением type.

type на самом деле является отдельным метаклассом. И это не то, что можно было бы воспроизвести на чистом Python, это делается путем небольшого мошенничества на уровне реализации.

Во-вторых, метаклассы сложны. Возможно, вы не стоит их использовать для очень простых изменений класса. Лучше изменять классы, используя два других метода:

* monkey patching
* декораторы классов

В 99% случаев, когда вам нужно изменить класс, вам лучше использовать их.

Но в 98% случаев вам вообще не нужно менять класс.

Статья написана на основе следующего топика на StackOverflow: https://stackoverflow.com/questions/100003/what-are-metaclasses-in-python