

Programação Orientada a Objetos

Metaclasses

Metaclasses

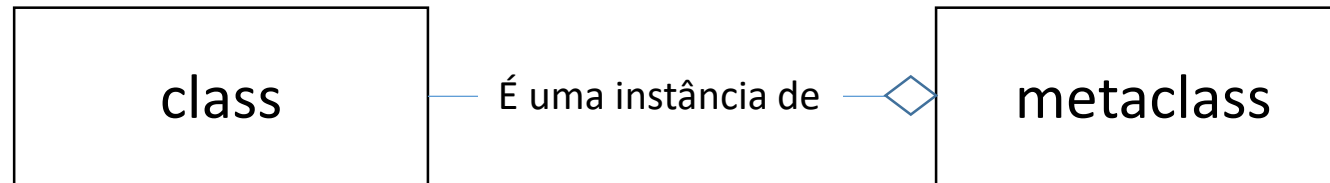
Metaclasses é um recurso do Python que é considerado por muitos como uma das coisas mais difíceis da linguagem, sendo desta forma, evitada por um grande número de desenvolvedores.

Na realidade, não é tão complicado quanto parece, uma vez que você compreenda alguns conceitos básicos.

Metaclasses permite fazer coisas que não são possíveis usando outras abordagens.

Metaclasses

Uma metaclasses é uma classe de outra classe, o que significa que a classe é uma instância de sua metaclasses.



Metaclasses

Algumas definições

- Metaclasses é um tipo (classe) que define outros tipos (classes). A coisa mais importante a saber para entender como uma metaclasses funciona é que as classes que definem instâncias de objetos também são objetos. Então, se eles são objetos, eles possuem uma classe associada.
- Uma classe é um objeto, e assim como qualquer outro objeto, é uma instância de algo. Este algo é uma metaclasses.
- Metaclasses é uma classe que pode ser instanciada por outras classes.
- Uma metaclasses é uma fábrica de classes, sendo utilizada para criar instâncias de classes.
- A metaclasses define o comportamento das classes que derivam dela.
- Como uma classe comum define o comportamento de suas instâncias, uma metaclasses define o comportamento das classes e suas instâncias.

Metaclasses

Em *Python*, a classe *type* é uma metaclasses e pode ser usada para criar novas metaclasses, ou seja, quando criamos uma classe sem definir sua metaclasses, será utilizada a metaclasses *type*.

Como em *Python* tudo é um objeto. Se dissermos que `a=5`, então `type(a)` devolve `<type 'int'>`, que significa que “a” é do tipo *int*. No entanto `type(int)` devolve `<type 'type'>`, que sugere a presença de uma metaclasses, pois *int* é uma classe do tipo *type*.

Metaclasses

A definição de classe é determinada por sua metaclasses, portanto, quando criamos uma classe com ***class A***, *Python* a cria com ***A = type(name, bases, dict)***, onde:

name é nome da classe.

base é a classe-base.

dict é a variável de atributos.

Se definirmos uma metaclasses na criação da classe como ***class A(metaclass=MinhaMetaClasse)***, *Python* criará a classe com ***A = MinhaMetaClasse(name, bases, dict)***.

Metaclasses

Todos objetos, por padrão, são do tipo *type*.

```
class MinhaClasse(object):  
    pass
```

```
print(type(MinhaClasse))  
print(type(int))  
print(type(str))
```

```
<class 'type'>  
<class 'type'>  
<class 'type'>
```

A classe criada *MinhaClasse* é do tipo *type*.
int e *str* também são do tipo *type*.

```
class MinhaClasse(object):  
    pass
```

```
print(isinstance(MinhaClasse, type))
```

```
True
```

Conferindo se *MinhaClasse* é uma instância de *type*.

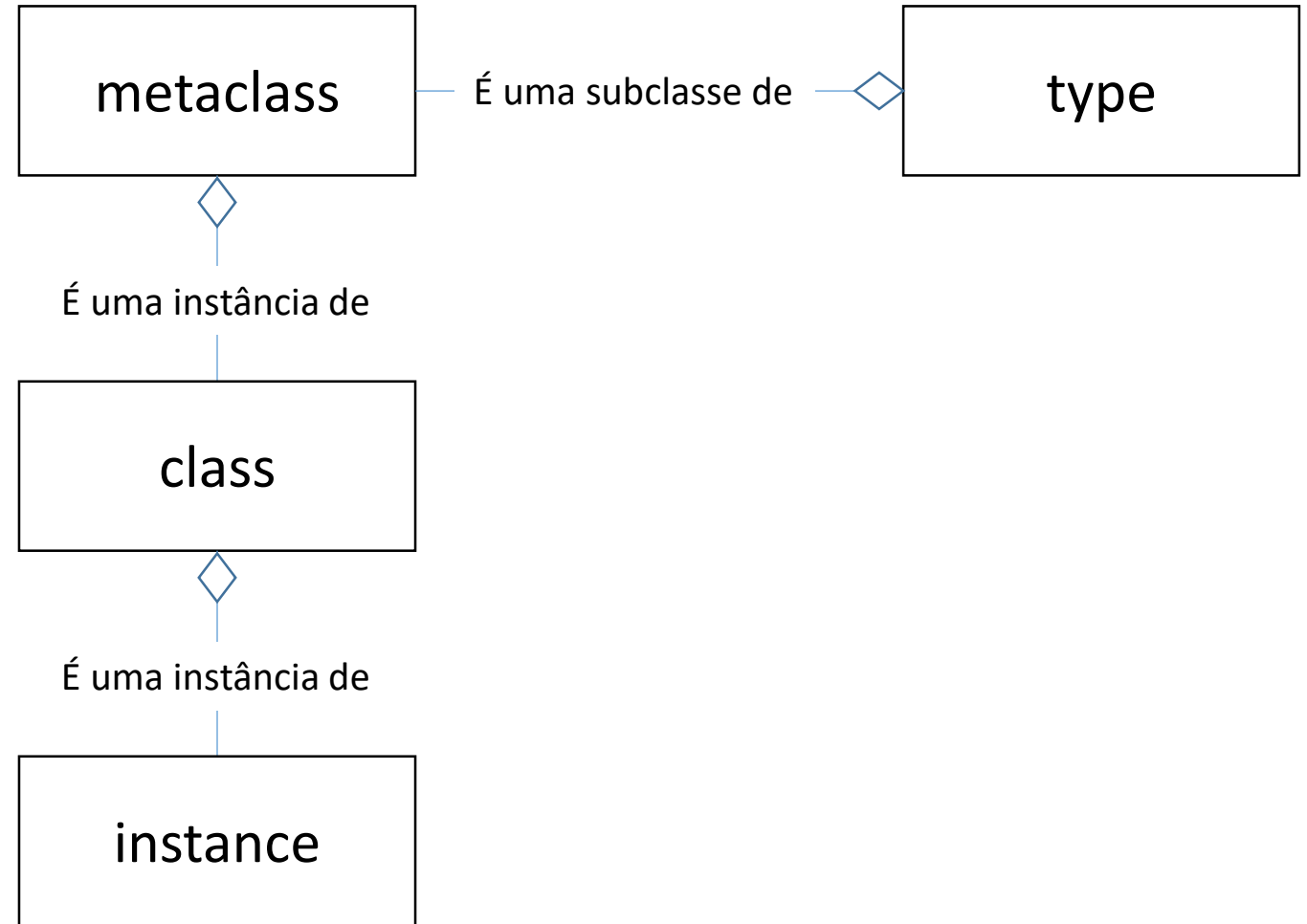
Metaclasses

Diagrama de instâncias



Metaclasses

Em Python é possível substituir a metaclasses por um objeto de classe com nosso próprio tipo. Normalmente, a nova metaclasses ainda é uma subclasse da classe type porque não fazê-lo tornaria as classes resultantes altamente incompatíveis com outras classes em termos de herança.



Metaclasses

Utilizando *type* para criar uma classe

O Python também tem a capacidade de gerar uma classe “programaticamente” usando a função *type()*.

Sintaxe:

`type(name, bases, namespace) => type(nome, classes_base, atributos)`

`name` = Nome da classe que se tornará o atributo `__name__`

`bases` = Uma tupla que detalha a classe base, torna-se o atributo `__bases__`

`namespace` = Um dicionário que é o *namespace* contendo definições para o corpo da classe, torna-se o atributo `__dict__`

Então, a função *type()* serve para dois propósitos:

- 1) Determinar qual o tipo de um objeto.
- 2) Criar um novo tipo.

Metaclasses

Utilizando *type* para criar uma classe. Exemplo1:

```
def meu_metodo(self):  
    return 1  
  
MinhaClasse = type('MinhaClasse', (object,), {'metodo': meu_metodo})  
  
instancia = MinhaClasse()  
x = instancia.metodo()  
print(x)
```

Equivale a

```
class MinhaClasse:  
    def metodo(self):  
        return 1  
  
instancia = MinhaClasse()  
x = instancia.metodo()  
print(x)
```

Metaclasses

Utilizando *type* para criar uma classe. Exemplo2:

```
def inicializador_pessoa(self, nome, sobrenome):
    self.nome = nome
    self.sobrenome = sobrenome

Pessoa = type('Pessoa', (object,), {'__init__': inicializador_pessoa, 'a': 'Teste', 'b': 10})

print("type(Pessoa) :", type(Pessoa))
print("vars(Pessoa) :", vars(Pessoa))

pes = Pessoa('Evaldo', 'Wolkers')

print("type(pes) :", type(pes))

print(pes.nome)
print(pes.sobrenome)
print(pes.a)
print(pes.b)
```

Metaclasses

Criando uma metaclasses e usando como se fosse *type()*.

```
# Para criar nossa metaclasses, vamos declarar a classe herdando da classe type
class MinhaMetaClasse(type):
    pass

# A metaclasses (MinhaMetaClasse) pode ser usada como se fosse type
MinhaClasse = MinhaMetaClasse('MinhaClasse', (), {})

minhaInstancia = MinhaClasse()
print(type(minhaInstancia))
```

```
<class '__main__.MinhaClasse'>
```

Metaclasses

Todas as classes criadas usam *type* como metaclasses. Podemos modificar este comportamento padrão informando a palavra-chave *metaclass* como argumento da declaração da classe.

```
# Para criar nossa metaclasses, vamos declarar a classe herdando da classe type
class MinhaMetaClasse(type):
    pass

# Definindo a metaclasses na declaração da classe
class MinhaClasse(metaclass=MinhaMetaClasse):
    pass

minhaInstancia = MinhaClasse()
print(type(minhaInstancia))
```

```
<class '__main__.MinhaClasse'>
```

Metaclasses

Agora, quando a classe `MinhaClasse` é construída, ela usará `MinhaMetaClasse` para construí-la.

Isso não tem muita utilidade, a menos que a metaclasses estenda a classe *type* de alguma forma.

O valor fornecido como argumento da metaclasses geralmente é outro objeto de classe, mas pode ser qualquer outro objeto chamado que aceita os mesmos argumentos que a classe `type` e espera retornar outro objeto de classe. Como já vimos, a assinatura de chamada de `type` é:

`type(name, bases, namespace).`

Metaclasses

Estendendo a classe `type`

Se for declarado um método `__new__` em uma metaclasses, ele controla como a classe é construída.

A assinatura de `__new__` é:

`__new__(mcs, nome, bases, atributos)`, onde `mcs` é a instância da metaclasses e os três parâmetros restantes são os mesmos da função *type*.

Um uso para substituir o método `__new__` é gerar dinamicamente métodos para a classe.

Metaclasses

Criando métodos dinamicamente substituindo o método `__new__`

```
def fabrica_de_funcoes(nome_metodo, retorno):
    def metodo_criado(self):
        return retorno

    metodo_criado.__name__ = nome_metodo
    return metodo_criado

class MinhaMetaClasse(type):
    def __new__(mcs, nome, bases, atributos):
        for nome, retorno in (('metodo1', 'Olá'), ('metodo2', 'Como vai você?'), ('metodo3', 30 + 40), ('metodo4', str(70 * 7))):
            atributos[nome] = fabrica_de_funcoes(nome, retorno)
        return super().__new__(mcs, nome, bases, atributos)

class MinhaClasse(metaclass=MinhaMetaClasse):
    pass

print("vars (MinhaClasse) :", vars(MinhaClasse))

mc = MinhaClasse()
print(mc.metodo1())
print(mc.metodo2())
print(mc.metodo3())
print(mc.metodo4())
```

Metaclasses

As metaclasses podem ser evitadas a favor de soluções mais simples como propriedades, descritores ou decoradores de classe.

Também é verdade que muitas vezes as metaclasses podem ser substituídas por outras abordagens mais simples, mas existem situações em que as coisas não podem ser feitas facilmente sem elas. Por exemplo, é difícil imaginar a implementação de ORM (Object Relational Mapping – Mapeamento objeto-relacional) do *Django* construída sem uso extensivo de metaclasses.

Metaclasses

Exemplo de Uso com o padrão Singleton

O Singleton proporciona uma forma de ter um e somente um objeto de determinado tipo, além de disponibilizar um ponto de acesso global. Por isso, os Singletons são geralmente usados em casos como logging ou operações de banco de dados, spoolers de impressão e muitos outros cenários em que seja necessário que haja apenas uma instância disponível para toda a aplicação. Evitando assim, requisições conflitantes para o mesmo recurso.

Com o padrão Singleton, as intenções são:

- Garantir que um e somente um objeto da classe seja criado.
- Oferecer um ponto de acesso para um objeto que seja global no programa.
- Controlar o acesso concorrente a recursos compartilhados.

Metaclasses

```
class MetaSingleton(type):
    _instancias = {}

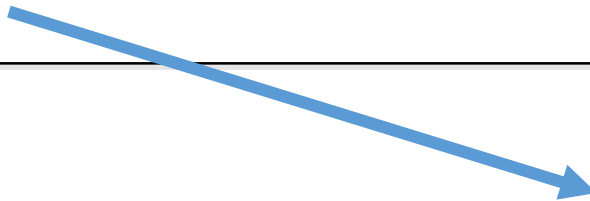
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instancias:
            cls._instancias[cls] = super(MetaSingleton, cls).__call__(*args, **kwargs)
        return cls._instancias[cls]

class Logger(metaclass=MetaSingleton):
    pass

logger1 = Logger()
logger2 = Logger()
print(logger1)
print(logger2)
```

Método `__call__`

O método especial `__call__` de Python é chamado quando um objeto precisa ser criado para uma classe já existente. Nesse código, quando instanciamos a classe `Logger` com `Logger()`, o método `__call__` da metaclasses `MetaSingleton` é chamado, o que significa que a metaclasses agora controla a instanciação do objeto.



```
<__main__.Logger object at 0x000001B3A63A0C50>
<__main__.Logger object at 0x000001B3A63A0C50>
```

Metaclasses


```
class PessoaMeta(type):
    def __init__(cls, name, bases, dict):
        super().__init__(name, bases, dict)
        cls._instance_map = {}

    def __call__(cls, first_name, last_name):
        key = (first_name, last_name)
        if key not in cls._instance_map:
            new_instance = super().__call__(first_name, last_name)
            cls._instance_map[key] = new_instance
        return cls._instance_map[key]

class Pessoa(metaclass=PessoaMeta):
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

pessoa1 = Pessoa("Evaldo", "Wolkers")
pessoa2 = Pessoa("Evaldo", "Wolkers")
pessoa3 = Pessoa("Seu", "Madruga")

print(pessoa1)
print(pessoa2)
print(pessoa3)
print(pessoa1 is pessoa2)
print((pessoa3 is pessoa1) or (pessoa3 is pessoa2))
```



```
<__main__.Pessoa object at 0x000001E97A28A400>
<__main__.Pessoa object at 0x000001E97A28A400>
<__main__.Pessoa object at 0x000001E97A28A438>
True
False
```

Metaclasses

Alguns exemplos de onde usar metaclasses:

- Registro de log e criação de perfil.
- Verificação de interface.
- Registro de classes no momento da criação.
- Adicionar novos métodos automaticamente.
- Criação de propriedades automáticas.
- Bloqueio e sincronização automática de recursos.

CONTINUA...