Reflexão computacional em Python

Álvaro Alvin Oesterreich Santos

Metadados

Metadados são dados sobre dados, informação sobre a informação, em Python por tudo ser um objeto desde as coisas que conhecemos como tipos básicos como inteiros, caracteres e tipos boolianos até instâncias de classes criadas que são o que 'normalmente' se é chamado de objeto em outras linguagens de programação mais tradicionais. Os metadados desses objetos são as informações sobre eles próprios, e o que define as informações de um objeto? A classe dele, essa é a grande chave da questão em Python, quais informações a classe de cada objeto possui sobre ele.

Tendo em vista que em Python tudo é um objeto, tudo também possui uma classe "mãe" de certa forma. Para descobrimos a classe de cada objeto em Python podemos utilizar a função **type()**, que recebe como argumento um objeto, e retorna qual o seu tipo, ou em outras palavras qual a sua classe.

Então, utilizado a função type() em uma variável que armazena um inteiro ela nos retornara o seu tipo, *int*, e se utilizarmos esse retorno como argumento de outra função type() para qual o tipo do tipo teremos um retorno interessante, o tipo do tipo é um *type*, que significa tipo. Logo, o tipo do tipo *int* é um tipo, e aplicando a mesma lógica podemos descobrir qual o tipo de *type*, e teremos como retorno novamente um *type* então podemos chegar a conclusão de que tudo no Phython deriva de uma classe mãe type, logo type é a metaclasse de todos os elemtnos como podemos observar no exemplo abaixo:

-x é uma instancia da classe Foo (pode ser qualquer outra classe).

-Foo é uma instancia da metaclasse type.

-type também é uma instancia da classe type, logo é uma instancia de si mesmo.

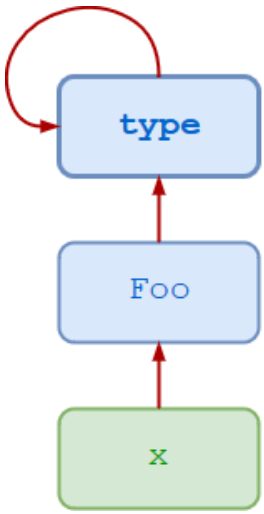

Figura 1 – Demostração instancias

(disponível em: https://realpython.com/python-metaclasses/)

Autorrepresentação

Em Python o que nos ajuda a acessar a capacidade do Python de se autorrepresentar, ou seja mostrar seu próprio programa é a biblioteca **inspect**, com ela é possível ter acesso a diversas informações do próprio programa como diz sua própria descrição “inspect live objects” (“Inspecionar objetos vivos” em português). Através dessa bibloteca é possível ter acesso a metadados e até ao próprio código de diversos objetos do programa em tempo de execução.

Código modificável em tempo de execução

O objetivo do código é de se modificar em tempo de execução e para isso foi utilizada a biblioteca inspect previamente apresentada.

Começamos com uma classe simples:

```
class Pessoa(metaclass=PessoaMeta):

    def __init__(self, nome, idade, profissao) ->
None:
        self.nome = nome
        self.idade = idade
        self.profissao = profissao
        pass

    def get_nome(self):
        """return the name """
        return self.nome

    def get_idade(self):
        """return the age"""
        return self.idade

    def get_profissao(self):
        """return the job """
        return self.profissao

    def print(self):
        print("Nome:", self.nome, "idade:",
 self.idade, "profissao:", self.profissao, '\n')
        pass
```

Figura 2: Classe inicial

Uma classe Pessoa que possui os atributos nome, idade e profissão e os seus respectivos métodos getters(por enquanto deve se ignorar o “metaclass=PessoaMeta”). Poderia ser criado manualmente um método para retornar o nome desses atributos, mas é possível fazer isso com métodos que utilizam metadados para determinar esses atributos, uma forma é através da funcionalidade da autorrepresentação obter o nome das funções da classe a a partir delas determinar o nome dos elementos que possuem função getter.

Para fazer isso vamos então criar manualmente a metaclasse na nossa classe pessoa, vamos criar uma metaclasse manualmente, que nós já sabemos que é uma classe do tipo type que ao ser chamada recebe o nome, uma tupla da classe base, e um dicionario que guarda a definição da classe base, seus métodos. Agora nossa metaclasse possui todas as informações da classe base que é Pessoa, porque na classe pessoa foi adicionado “metaclass=PessoaMeta” como argumento e chamando a si mesmo com sua classe base e armazena o a metaclasse que retorna em uma variavel.

Essa variável sera retornada com a metaclasse da classe base, então podemos adicionar atributos e métodos a essa metaclasse para em tempo de execução os dados da classe base podem ser modificados. Então podemos criar um atributo que sera adicionado à metaclasse, percorremos os métodos da classe contidos no dicionário e verificamos seus nomes utilizando o inspect, e caso sejam getters podemos extrair o nome do elemento do getter diretamente do nome do método por

haver um padrão. E depois de extrairmos essa informação basta adicioná-la ao novo atributo criado. Para fazer tudo isso temos a seguinte metaclasse:

```
class PessoaMeta(type):                                 #cls = self; name = nome da classe
    def __new__(cls, name, bases, dct):                 #bases = a tupla de classes base
        x = super().__new__(cls, name, bases, dct)      #dict = dicionario de metodos e seus   implementacoes
        aux = "atributos:"
        #passa pelos item da classe referida e para toda funçao identificada como getter obtem o seu atributo
        for key, val in dct.items():
            if(type(val) == FunctionType):
                if(isGetter(val)):
                    aux = aux + " " + GetterToPropiety(val)
        x.proprieties = aux
        return x
```

Figura 3: metaclasse

Para verificar se a função é um getter foi criada uma função isGetter, que retorna se a implementação da função inicia com "   def get_":

```
def isGetter(f:FunctionType):
    return str(inspect.signature(f)) == '(self)' and str(inspect.getsource(f)).startswith("    def get_")
```

Figura 4: função isGetter

Para extrair o nome do elemento da função getter foi criada a função GetterToPropiety que recorta a string da implementação retornada pelo inspect para obter o nome da variável do getter:

```
def GetterToPropiety(f:FunctionType):
    aux = ""
    #vai até o '(' e armazena em uma variave auxiliar
    for x in str(inspect.getsource(f)):
        if(x == "("):
            break
        aux = aux + x
    #corta fora o 'def get_' para obter o nome do atributo
    aux = aux[12:]
    return aux
```

Figura 5: função GetterToPropiety

Assim, ao retornar a metaclasse com o novo atributo adicionado a ela, o objeto recebe um novo atributo em tempo de execução. Então podemos instanciar nossa classe em um objeto no programa principal e como podemos acessar o novo atributo adicionado com a metaclasse:

```
pessoa1 = Pessoa("Alvaro", 19, "estudante")
print("Atributos obtidos por meio de reflexao e metaprogramacao com as funcoes getters:")
print(pessoa1.proprieties)
```

Figura 6: Programa principal

```
Atributos obtidos por meio de reflexao e metaprogramacao com as funcoes getters:
atributos: nome idade profissao
```

Figura 7: Saída do programa

Referencias

Malik, Farhad.**Advanced Python:Metaprogramming**. Disponível em: https://medium.com/fintechexplained/advanced-python-metaprogramming-980da1be0c7d. Acesso em 23/05/2021

Sturtz, John. **Python Metaclasses**. Disponível em: https://realpython.com/python-metaclasses/. Acesso em 25/05/2021

Python documentation, **inspect — Inspect live object**. https://docs.python.org/3/library/inspect.html. Acesso em 25/05/2021