Métodos mágicos

1. Métodos "convencionais"

Considere uma classe para representar frações, contendo dois atributos que significam, respectivamente, seu numerador e denominador. Vamos considerar um pedacinho dela aqui: sua inicialização e um método para somar dois objetos fração. Iremos manter as coisas simples para focar em um novo conceito, mas não se esqueça de encapsular adequadamente suas classes!

class Fracao:

def \_\_init\_\_(self, num, den):

self.num = num

self.den = den

def soma(self, fracao2):

# primeiro multiplicamos os denominadores:

denominador = self.den \* fracao2.den

# agora o produto cruzado entre numeradores e denominadores:

numerador = self.num \* fracao2.den + self.den \* fracao2.num

# montamos a nova fração:

resultado = Fracao(numerador, denominador)

return resultado

Note que o método soma será chamado a partir de um objeto Fracao e deverá receber como parâmetro outro objeto Fracao.

meio = Fracao(1, 2)

terco = Fracao(1, 3)

meio\_mais\_terco = meio.soma(terco)

print(f'{meio\_mais\_terco.num}/{meio\_mais\_terco.den}')

Resultado na tela:

5/6

Note que, em comparação com outras classes que já utilizamos, como list ou str, nossa classe Fracao é bastante inconveniente de trabalhar.

Por exemplo, nós tivemos que formatar a nossa fração para poder imprimi-la. Simplesmente utilizar um print não resolve muita coisa:

print(meio\_mais\_terco)

Resultado:

<\_\_main\_\_.Fracao object at 0x000001D39ED3E5E0>

Compare com imprimir uma lista, onde o colchete e as vírgulas já vêm automaticamente:

lista = [1, 'dois', 3.0]

print(lista)

# resultado: [1, 'dois', 3.0]

A soma também é bastante inconveniente. Você deve se lembrar que podemos "somar" duas strings utilizando o operador + sem dificuldade alguma:

str1 = 'Métodos'

str2 = 'Mágicos'

print(str1 + str2)

# resultado: MétodosMágicos

Já nos nossos objetos Fracao, que nós sabemos matematicamente como somar, não conseguimos utilizar o operador +:

print(meio + terco)

Resultado:

TypeError: unsupported operand type(s) for +: 'Fracao' and 'Fracao'

Você deve se lembrar que um dos quatro princípios da programação orientada a objeto era a abstração. Esse princípio dita que a classe deve ocultar ao máximo a sua complexidade e fornecer uma interface simples e fácil de ser utilizada.

O que é mais fácil e intuitivo: f1 + f2 ou f1.soma(f2)? print(f1) ou um print com uma f-string enorme referenciando diversos atributos diretamente?

O que list, str, dict e várias outras classes prontas do Python tem de especial que a nossa classe não tem? Mágica.

Aliás, métodos mágicos.

2. Métodos Mágicos

2.1. Sintaxe "dunder"

Nos nossos exemplos, frequentemente incluímos um método em nossas classes cujo nome não escolhemos. Nós sempre nos adequamos a um padrão que nos foi imposto: o \_\_init\_\_. Coincidentemente (ou não), nós também nunca chamamos esse método explicitamente: ele sempre foi chamado de maneira automática pelo Python em uma situação específica (a criação de um objeto).

Existe uma família de métodos com um comportamento semelhante ao \_\_init\_\_. Eles também possuem nomes padrão, e seus nomes são sempre precedidos e sucedidos por dois underscores, o que lhes rendeu o apelido dunder methods, abreviação de "double underscore". Esses métodos também não foram feitos para serem chamados pelo nome. Ao invés disso, eles são chamados automaticamente pelo Python em situações específicas.

Esses métodos também são chamados de métodos mágicos, e eles são responsáveis por todas essas situações em que conseguimos utilizar "recursos padrão" do Python, como operadores ou funções comuns sobre os nossos objetos. Vejamos alguns exemplos.

Quando dissemos no passado que o método \_\_init\_\_ representa o construtor, essa foi uma simplificação. O processo de instanciação de objetos envolve dois métodos mágicos diferentes: o \_\_new\_\_ e o \_\_init\_\_. O método \_\_new\_\_ é executado primeiro, e ele cria um objeto vazio vinculado à classe desejada. Em seguida, esse objeto é passado para o \_\_init\_\_ junto com os parâmetros utilizados na instanciação do objeto para que esse método faça a inicialização dos atributos.

Na grande maioria dos casos, você não precisa implementar sua própria versão do \_\_new\_\_, pois geralmente não temos interesse em alterar a lógica de reservar um objeto na memória, apenas de definir seus atributos. Mas há alguns poucos casos bastante interessantes onde convém interferir nesse fluxo. . Caso você tenha curiosidade, você pode encontrar alguns exemplos ilustrando o fluxo de execução de construção de um objeto aqui: https://realpython.com/python-class-constructor/#pythons-class-constructors-and-the-instantiation-process

2.2. Método de representação

Um dos inconvenientes que citamos ali em cima foi o print. Ao passarmos nossos próprios objetos para o print, o resultado é um número gigante e que faz pouco sentido para seres humanos em situações normais.

Para permitir a impressão direta de nossos objetos, sem que o programador utilizando nossa classe precise manualmente acessar os atributos e formatá-los, podemos criar um método mágico de representação em string: o \_\_str\_\_.

O \_\_str\_\_ é o método que será chamado quando tentarmos converter nossos objetos para string:

str(objeto)

Ao passarmos um objeto para o print, o Python irá procurar por seu método \_\_str\_\_. Caso ele exista, ele será chamado e o seu retorno será escrito na tela.

Portanto, iremos criar um método chamado de \_\_str\_\_ que irá retornar uma string já no formato desejado.

class Fracao:

def \_\_init\_\_(self, num, den):

self.num = num

self.den = den

def soma(self, fracao2):

# primeiro multiplicamos os denominadores:

denominador = self.den \* fracao2.den

# agora o produto cruzado entre numeradores e denominadores:

numerador = self.num \* fracao2.den + self.den \* fracao2.num

# montamos a nova fração:

resultado = Fracao(numerador, denominador)

return resultado

def \_\_str\_\_(self):

return f'{self.num}/{self.den}'

###

meio = Fracao(1, 2)

print(meio)

terco = Fracao(1, 3)

print(terco)

meio\_mais\_terco = meio.soma(terco)

print(f'{meio} + {terco} = {meio\_mais\_terco}')

Olha só! Agora conseguimos printar diretamente objetos de nossa classe. Inclusive ficou bem mais fácil de montar strings complexas com diversos objetos de nossa classe no meio.

Tecnicamente o \_\_str\_\_ não é um método de representação, mas um método de coerção de tipos: ele ensina o Python como gerar uma string a partir do seu objeto.

Também é possível criar métodos de coerção para outros tipos, como \_\_int\_\_, \_\_bool\_\_ e \_\_float\_\_. Baseado nos seus conhecimentos sobre valores truthy e falsy e em seus conhecimentos de matemática, como ficariam esses três métodos em uma classe representando fração?

2.3. Métodos aritméticos

O próximo inconveniente que deveríamos endereçar é a soma. Nós, seres humanos, temos bastante facilidade para compreender o que o código abaixo faria:

f1 = Fracao(1,2)

f2 = Fracao(1,3)

resultado = f1 + f2

O Python, porém, ainda não. Afinal, nossos objetos possuem diversos atributos, e não está claro para o Python quais deles entram na conta, e nem mesmo qual exatamente é a conta a ser realizada.

A soma - bem como qualquer outra operação aritmética padrão do Python - pode ser implementada através de um método mágico seguindo o seguinte padrão:

O nome do método será entre dois pares de underscores (sintaxe dunder).

O método receberá 2 parâmetros: self (representando o objeto à esquerda do operador) e other (representando o objeto à direita).

O método irá retornar o resultado da operação.

A expressão:

f1 + f2

é "traduzida" pelo Python para:

f1.\_\_add\_\_(f2)

\_\_add\_\_ é o método mágico de soma, f1 será o self e f2 será o other.

Os principais métodos aritméticos são:

add: soma (+)

sub: subtração (-)

mul: multiplicação (\*)

truediv: divisão real (/)

floordiv: divisão inteira (//)

mod: resto da divisão (%)

pow: potência (\*\*)

Vejamos nossa classe Fracao atualizada para aceitar o operador de soma:

class Fracao:

def \_\_init\_\_(self, num, den):

self.num = num

self.den = den

def \_\_add\_\_(self, other):

# primeiro multiplicamos os denominadores:

denominador = self.den \* other.den

# agora o produto cruzado entre numeradores e denominadores:

numerador = self.num \* other.den + self.den \* other.num

# montamos a nova fração:

resultado = Fracao(numerador, denominador)

return resultado

def \_\_str\_\_(self):

return f'{self.num}/{self.den}'

###

meio = Fracao(1, 2)

terco = Fracao(1, 3)

meio\_mais\_terco = meio + terco

print(meio\_mais\_terco)

a = 10

b = 20

c = a + b

print(c)

Note como a sintaxe agora se parece muito mais com a forma como nós escreveríamos "no papel" e menos com um encadeamento de funções. Não precisamos mais memorizar o nome dos métodos da classe e podemos utilizar de maneira intuitiva os operadores que já estamos familiarizados desde a infância.

2.4. Métodos de comparação

Os operadores aritméticos não são os únicos que utilizamos com frequência no Python. Frequentemente utilizamos alguns operadores lógicos, que deverão nos retornar True ou False. É bem comum utilizarmos esses operadores com números, para verificar se um deles é maior do que o outro, menor, igual ou diferente.

Porém, eles podem ser utilizados para objetos diversos, como strings. Veja o resultado do código abaixo:

str1 = 'Banana'

str2 = 'Abacate'

str3 = 'Abacaxi'

print(str1 < str2) # Banana < Abacate

print(str1 < str3) # Banana < Abacaxi

print(str2 < str3) # Abacate < Abacaxi

Os criadores da classe string parecem ter utilizado métodos mágicos para criar alguma regrinha permitindo a comparação de strings. De maneira simplificada, é ordem alfabética (mas dê uma olhadinha na observação ao final deste material).

Podemos criar nossas próprias regras para comparar nossos objetos Fracao também. Os métodos mágicos de comparação também seguirão a lógica do self/other dos métodos aritméticos. O que muda é o retorno, que idealmente deve ser um booleano.

Os métodos são:

gt - greater than/maior que (>)

ge - greater or equal/maior ou igual (>=)

lt - less than/menor que (<)

le - less or equal/menor ou igual (<=)

eq - equal/igual (==)

ne - not equal/diferente (!=)

Vamos implementar o > em nossa classe Fracao. Execute o código abaixo e veja seu resultado:

class Fracao:

def \_\_init\_\_(self, num, den):

self.num = num

self.den = den

def \_\_add\_\_(self, other):

# primeiro multiplicamos os denominadores:

denominador = self.den \* other.den

# agora o produto cruzado entre numeradores e denominadores:

numerador = self.num \* other.den + self.den \* other.num

# montamos a nova fração:

resultado = Fracao(numerador, denominador)

return resultado

def \_\_gt\_\_(self, other):

# uma lógica possível para comparar frações:

# realizar a divisão delas e comparar o resultado

div1 = self.num/self.den

div2 = other.num/other.den

return div1 > div2

def \_\_str\_\_(self):

return f'{self.num}/{self.den}'

meio = Fracao(1, 2)

terco = Fracao(1, 3)

meio\_mais\_terco = meio + terco

print(meio\_mais\_terco)

print(f'{meio} > {terco}: {meio > terco}')

print(f'{terco} > {meio}: {terco > meio}')

print(f'{meio} > {meio\_mais\_terco}: {meio > meio\_mais\_terco}')

print(f'{terco} > {meio\_mais\_terco}: {terco > meio\_mais\_terco}')

print(f'{meio\_mais\_terco} > {terco}: {meio\_mais\_terco > terco}')

print(f'{meio\_mais\_terco} > {meio}: {meio\_mais\_terco > meio}')

if meio\_mais\_terco > meio and meio\_mais\_terco > terco:

print('A soma das frações é maior do que as frações')

else:

print('Quebramos a matemática?')

Note que ao implementarmos os métodos de comparação, nós podemos até mesmo criar expressões lógicas completas para utilizar em condicionais e loops.

2.5. Outros métodos mágicos

Nós abordamos aqui alguns dos métodos mágicos mais usados, mas praticamente todos os operadores e comportamentos padrão do Python podem ser redefinidos para objetos de nossa classe, inclusive métodos para iterar nosso objeto (usar um for para percorrê-lo), acessar elementos utilizando colchetes, realizar algumas conversões, dentre várias outras coisas interessantes.

Este link (em inglês) é bastante abrangente e ilustra tudo o que podemos personalizar em uma classe.

Uma boa referência em português, apesar de um pouco menos didática, é a documentação oficial.

Aqui também temos um curso bastante didático sobre métodos mágicos.

3. Observação sobre comparação de strings

Vamos fazer um comentário um pouco off-topic aqui, mas como mencionamos comparação entre strings, seria bom esclarecer o que se passa debaixo dos panos.

Observação sobre comparação de str

Por não ser o foco da aula, falamos de maneira simplificada que comparação entre strings considera ordem alfabética. É um pouco mais complicado do que isso.

Internamente, cada caractere é armazenado como um número. Quando utilizamos qualquer tipo de aplicação que irá exibir um texto (um editor de textos, navegador de internet, ou mesmo os nossos programinhas em Python rodando no terminal), a aplicação usa esses números como índices em uma tabela de codificação de caracteres.

Temos diversos esquemas diferentes de codificação de caracteres em uso pelo mundo, e quando você está usando um programa ou navegando por um site e você nota símbolos estranhos no texto (frequentemente onde teríamos caracteres especiais, como letras com acento), é provável que o autor do texto tenha utilizado uma tabela e o seu computador esteja usando outra.

Vários programas permitem a conversão entre essas tabelas, e você já deve ter visto essa "sopa de letrinhas" em alguma aba ou janela de configuração em algum editor de textos: utf-8, utf-16, windows-1252 (ou cp-2152), e até mesmo alguns padrões ISO.

Para ilustrar a ideia, vamos colocar aqui uma das tabelas mais simples, a tabela ASCII:

Fonte: https://upload.wikimedia.org/wikipedia/commons/thumb/1/1b/ASCII-Table-wide.svg/1200px-ASCII-Table-wide.svg.png

Note que o caractere 'A' está no índice 65, o 'B' está no índice 66, e assim sucessivamente. Por isso, 'Abacate' < 'Banana' é verdadeiro: a primeira string começa com uma letra no índice 65 da tabela, a segunda com uma letra no índice 66.

No caso de 'Abacate' e 'Abacaxi', temos um "empate" das 5 primeiras letras. Então é a sexta letra que vai mandar: 'x' é maior do que 't', por estar em uma posição superior na tabela.

Note que a ordem não é exatamente alfabética: entre os caracteres maiúsculos, seguimos ordem alfabética. Entre os minúsculos, idem. E entre os dígitos numéricos, também temos ordem crescente correspondente aos valores. Mas todos os minúsculos são "maiores" do que qualquer maiúsculo, que por sua vez são "maiores" do que qualquer dígito numérico. Símbolos, operadores e sinais de pontuação estão em posições diversas.