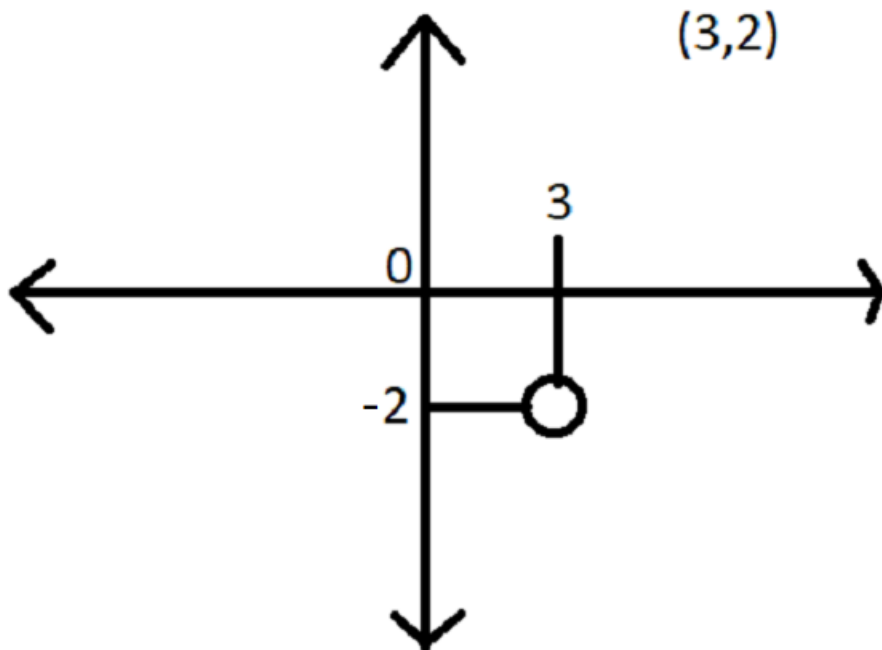


Termos a serem explorados:

Point: é um tipo não primitivo do python que é destinado especificamente para poo, em sua teoria o point é um tipo primitivo que organiza seus valores em uma tupla de modo a representar um plano vetorial exemplo $\rightarrow (x, y)$, ele não é nada mais do que uma tupla que representa o quão longe um valor está do ponto 0 de um vetor, mas na pratica o point é apenas uma tupla que guarda dois valores numericos. exemplo da teoria:



Metodos especiais do POO

Metodos especiais são metodos que alteram o comportamento de uma classes e como

seu objeto interage com o resto da programação.

Metodos especiais de retorno: são metodos que ditam como uma classe obviamente invocada como obj deve interagir, e oque obj deve retornar caso uma operador logico ou aritimetico seja usado entre ele e alguma outra classe.

para fazer essas preconfigurações podemos chamar os seguintes metodos dentro da classe:

O parametro obj2 é apenas um nome aleatorio que foi dado para especificar qualquer outro obj que estiver interagindo com o obj principal

Aritimeticos:

`__add__(self, obj2)` ---> definir o comportamento de um obj no caso de (+)

`__sub__(self, obj2)` ---> definir o comportamento de um obj no caso de (-)

`__mul__(self, obj2)` ---> definir o comportamento de um obj no caso de (*)

Logicos:

`__gt__(self, obj2)` ---> definindo o comportamento do obj no caso de (>)

`__ge__(self, obj2)` ---> definindo o comportamento do obj no caso de (>=)

`__lt__(self, obj2)` ---> definindo o comportamento do obj no caso de (<)

`__le__(self, obj2)` ---> definindo o comportamento do obj no caso de (<=)

`__eq__(self, obj2)` ---> definindo o comportamento do obj no caso de (==)

Outros:

`__len__(self)` ---> definindo comportamento do obj no caso de uma func `len()`

exemplo (no exemplo a seguir as cls apresenta os atb ---> x e y):

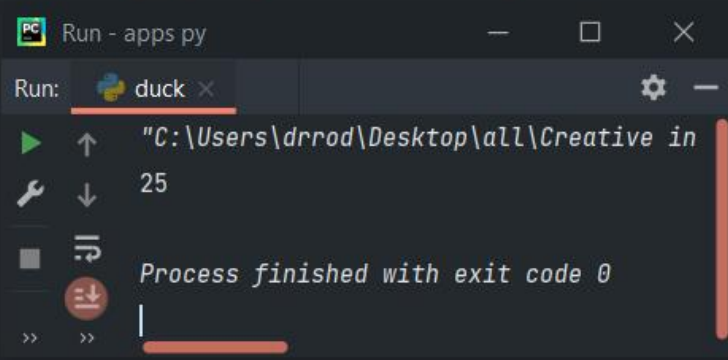
```
class pato:
    def __init__(self, a, b):
        self.x = a
        self.y = b

    def __add__(self, other):
        return (self.x + self.y) + (other.x + other.y)

    def __sub__(self, other):
        return (self.x + self.y) - (other.x + other.y)

    def __mul__(self, other):
        return (self.x + self.y) - (other.x + other.y)

abc = pato(10, 5)
dce = pato(5, 5)
print(abc+dce)
```



The screenshot shows a Python IDE with a dark theme. The code defines a class `pato` with methods `__init__`, `__add__`, `__sub__`, and `__mul__`. Below the code, the following lines are executed: `abc = pato(10, 5)`, `dce = pato(5, 5)`, and `print(abc+dce)`. A terminal window titled "Run - apps.py" is open, showing the output of the code: `"C:\Users\drrod\Desktop\all\Creative in 25"`. The terminal also shows the message "Process finished with exit code 0".

Podemos também definir qual sera o retorno de um objeto caso ao pedirmos para printar algum elemento da nossa classe e ele printe o endereço do elemento dentro do obj exemplo:

```
class pato:
    def __init__(self, a, b):
        self.x = a
        self.y = b

    def __add__(self, other):
        return (self.x + self.y) + (other.x + other.y)

    def __sub__(self, other):
        return (self.x + self.y) - (other.x + other.y)

    def __mul__(self, other):
        return (self.x + self.y) - (other.x + other.y)

print(pato(1,2))
```

Run - apps.py

Run: duck ×

"C:\Users\drrod\Desktop\all\Creative in Python\..."

<__main__.pato object at 0x00000143BC7A6FD0>

Process finished with exit code 0

para isso é necessário usar o metodo especial `__str__(self)`, no qual você pode configurar qual sera o retorno que ocorrera neste caso

exemplo:

```
class pato:
    def __init__(self, a, b):
        self.x = a
        self.y = b

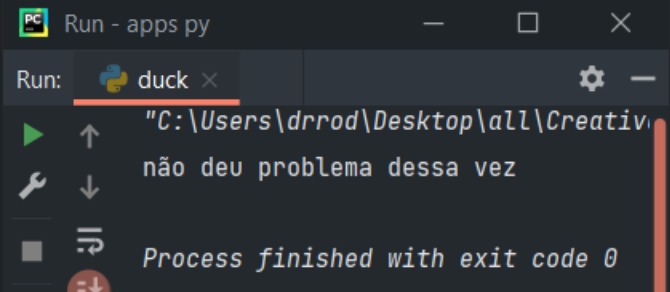
    def __add__(self, other):
        return (self.x + self.y) + (other.x + other.y)

    def __sub__(self, other):
        return (self.x + self.y) - (other.x + other.y)

    def __mul__(self, other):
        return (self.x + self.y) - (other.x + other.y)

    def __str__(self):
        return 'não deu problema dessa vez'

print(pato(1, 2))
```



Atributos de classe:

Até agora no Poo todos os atributos que utilizamos faziam referencia ao objeto quando ele fosse invocado e não a classe em si.

Para podermos provar a afirmação anterior darei um exemplo:

Envocamos duas variaveis distintas "b1" e "b2", em cada uma delas chamaremos a classe pato() que contem o atributo: self.lasanha = "garfield",

agora se mudarmos o valor de self.lasanha em "b1" o valor de lasanha não ia ser alterado em "b2" pois um atributo ".self" sempre estara ligado ao seu objeto e não a sua

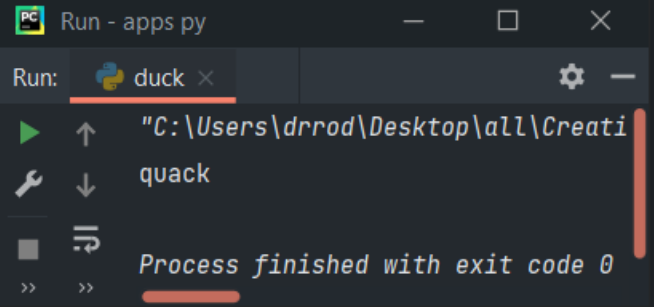
classe.

Para criar um atributo universal para uma classe (ou seja um atributo que se alterado causa mudança em todos os objetos relacionados a ela mesma e que possa ser invocado fora da classe chamando diretamente a classe, sem a necessidade de um objeto) é muito simples basta declarar esse atributo no inicio da classe, como se fosse uma variavel comum. E quando quisermos nos referir a este atributo dentro de algum metodo dessa classe usamos o ".self" normalmente

exemplo:

```
class pato:
    animal = "quack"
    def barulhodepato(self):
        print(self.animal)

print(pato.animal)
```



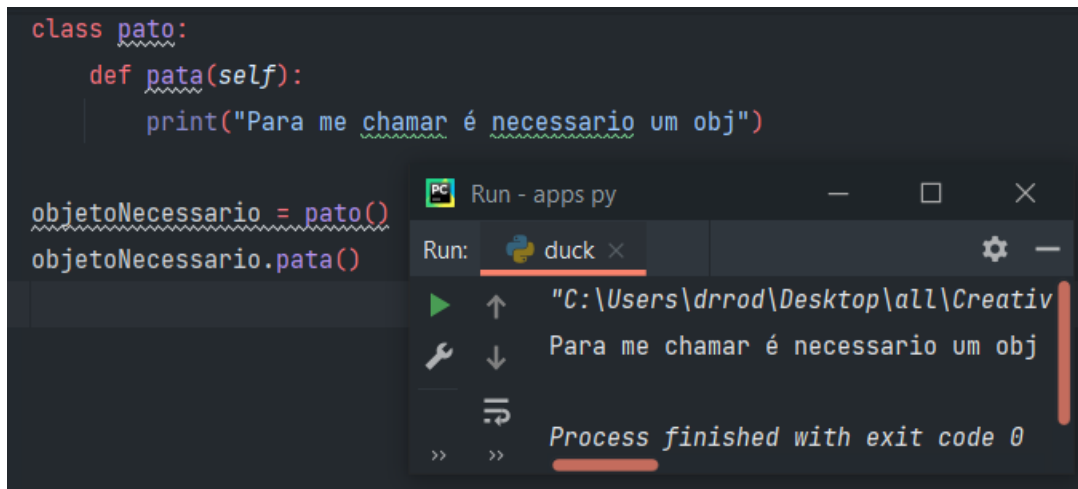
Metodos Decorators padrões (M.D.P):

Para declarar um metodo decorator é necessario criar um metodo e colocar em cima dele um "@" seguido do nome do decorator que você quer declarar para aquele metodo:

exemplo:

```
class pato:
    @classmethod
    def pata(cls):
        print("Declarei um decorator")
```

M.D.P @classmethod ---> @classmethod não faz nada mais do que especificar que o método que está sendo referido pelo decorator não precisa de um objeto para ser chamado como no exemplo abaixo:



```
class pato:
    def pata(self):
        print("Para me chamar é necessario um obj")

objetoNecessario = pato()
objetoNecessario.pata()
```

The screenshot shows a code editor with the following code:

```
class pato:
    def pata(self):
        print("Para me chamar é necessario um obj")

objetoNecessario = pato()
objetoNecessario.pata()
```

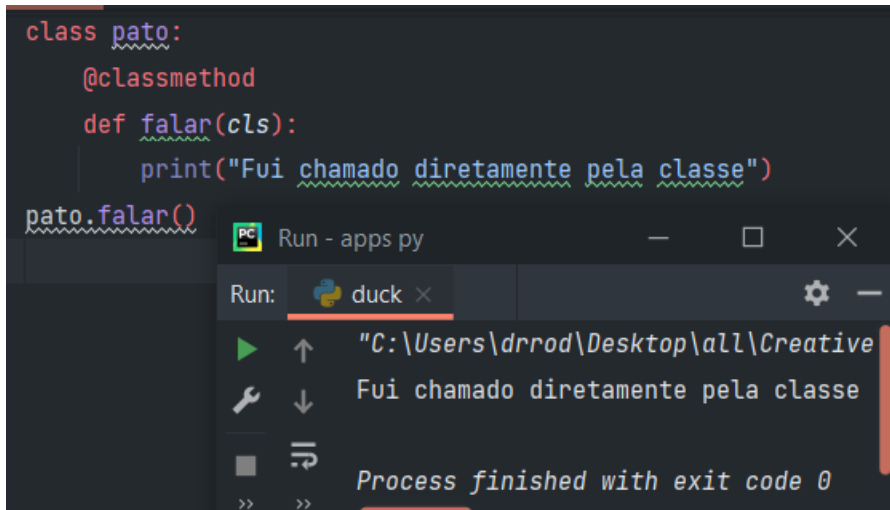
Below the code, a terminal window titled "Run - apps py" shows the output of the program:

```
Run: duck x
"C:\Users\drrod\Desktop\all\Creativ
Para me chamar é necessario um obj
Process finished with exit code 0
```

O método que está sendo referido pelo decorator pode ser chamado diretamente pela sua classe, e como parametro ele não passa o **self**, ele passa o **cls**, pois **para conseguir ter acesso a todos os atributos da classe mesmo sem declarar um objeto é necessario especificar em seu parametro qual é a classe na qual as informações dos atributos estão sendo puxadas** .

exemplo:

```
class pato:
    @classmethod
    def falar(cls):
        print("Fui chamado diretamente pela classe")
pato.falar()
```



M.D.P @staticmethod ---> @staticmethod consegue usar metodos de uma classe sem envoca-la ou se referir a ela em seu parametro, em base o staticmethod permite com que execute um metodo apenas expecificando sua localização dentro da classe, mas como consequencia o staticmethod não pode se referir a nenhum atributo localizado dentro da classe pois ele não está associada a mesma nem com o **self** e nem com o **cls**

exemplo do erro que dará se um atributo de uma classe for usado em um metodo @staticmethod:


```
class pato:
    animal = "quack"

    @staticmethod
    def barulhodepato():
        print(self.animal)

pato.barulhodepato()
```

apps py

duck ×

Traceback (most recent call last):

File "C:/Users/drrrod/Desktop/all/Creative in Python/PyPy/apps py/stud/d", line 1, in <module>

pato.barulhodepato()

TypeError: barulhodepato() missing 1 required positional argument: 'self'

finished with exit code 1

```
class pato:
    animal = "quack"

    @staticmethod
    def barulhodepato(self):
        print(self.animal)

pato.barulhodepato()
```

s py

ck ×

Traceback (most recent call last):

File "C:/Users/drrrod/Desktop/all/Creative in Python/PyPy/apps py/stud/d", line 1, in <module>

pato.barulhodepato()

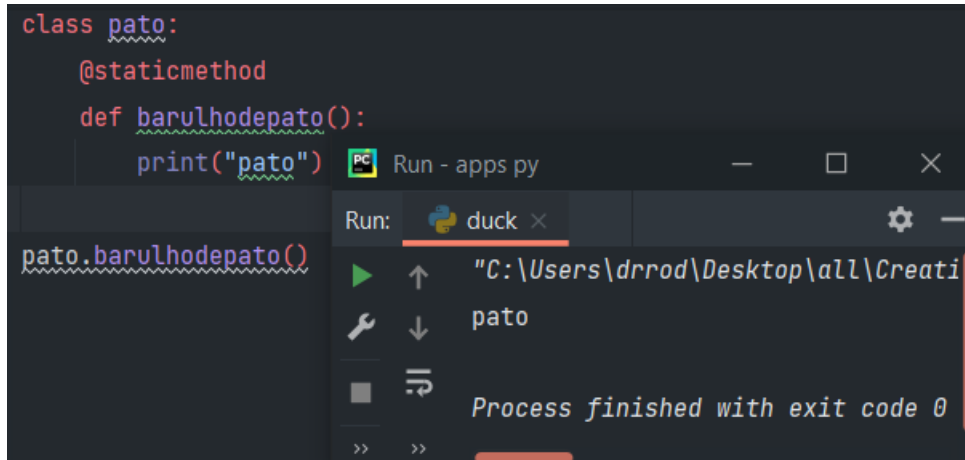
TypeError: barulhodepato() missing 1 required positional argument: 'self'

process finished with exit code 1

Exemplo uso correto do @staticmethod:

```
class pato:
    @staticmethod
    def barulhodepato():
        print("pato")

pato.barulhodepato()
```



Encapsulamento

Strictly speaking

_x is called a “protected” attribute (one underscore)

__x is called a “private” attribute (double underscore)

Here I use the word “private” for internal attributes with only one leading underscore as well

Metodos decorators property(M.D.P)

Os M.D.P são metodos criados para serem integrados em conjunto com o

encapsulamento, sendo que eles foram criados para serem a única conexão possível de se ter entre um atributo/metodo privado ou protegido e o plano global do programa.

Os métodos property existentes são os:

Setters ---> os setters são os métodos que irão tratar o atributo quando ele for referido fora da classe e depois os enviarão para o getter, ou seja ele vão setar o valor que vai ser retornado

Getter ---> A função do getter é puxar um valor antes que ele possa ser atribuído a um atributo, mandar este valor para um setter e retornar este valor já modificado devolta ao atributo (exemplo):

```
class Mostrar:
    def __init__(self, alfabeto):
        self.abc = alfabeto
```

A função do getter acima seria puxar e mandar o parametro "**alfabeto**" para um setter antes que ele seja atribuído ao "**self.abc**", após isso ele tem a função de retornar o "**alfabeto**" já modificado de volta para o "**self.abc**"

Deleters ---> O deleter vai definir o que aconteceu dentro da classe se houver uma tentativa de deletar algum atributo ou metodo de dentro da classe

Para explicar o uso do M.D.P darei um exemplo: criaremos uma classe que cadastre nome para patos, porem não poderão ser cadastrados nomes repetidos .

Para isso criarei a classe pato:

```

class Pato:
    name = []

    def __init__(self, nome):
        self.__duck = nome
        self.name.append(self.__duck)
        print(self.name)

    @property
    def __duck(self):
        return self.newduck

    @__duck.setter
    def __duck(self, namaa):
        if namaa in self.name:
            print("alreaddy exist")
            self.newduck = ""
        else:
            self.newduck = namaa

while True:
    Pato(input("ducks name ---> "))

```

Atraves dessa classe explicarei exatamente o uso dos getters e setters.

Se não houvesse o getter nem o setter a classe acima simplesmente daria o valor do parametro nome para o atributo "**__duck**" e em seguida o adicionaria na lista "name" porem temos um metodo property, e explicarei como utiliza-lo.

Vamos começar com o getter, o decorator "**@property**" é obrigatorio para a criação de um metodo getter, para se criar este metodo também é obrigatorio que seu nome seja exatamente o nome do atributo de origem, ou seja o atributo de origem é o **self.__duck**, então o nome do metodo getter tera que ser "**__duck**".

Dentro do metodo getter sera definido um atributo (**conhecido como atributo getter**) que tera seu valor retornado futuramente para o atributo de origem. (Lembrando que este atributo getter não pode ter o mesmo nome que o atributo de origem).

Setter: A função do setter é receber um valor vindo do getter, trata-lo e retorna-lo novamente ao getter quando for tratado.

Como já vimos uma das funções do getter é enviar o valor que sera tratado para o setter, e para o getter saber para onde ele deve enviar este valor criamos o decorator setter usando o nome do metodo getter e fazemos isso colocando: **@ + "o nome do metodo getter" .setter**.

Para receber o valor que o getter puxou do atributo de origem precisamos criar um parametro (conhecido como parametro setter), no exemplo acima criamos **namaa** e o tratamos dentro do nosso metodo setter, lembrando que o nome do metodo setter deve ser o mesmo que o do metodo getter. Por final após o tratamento feito temos que associar o valor do nosso parametro setter ao atributo getter que no exemplo acima é o **self.newduck**.

Resumindo oque foi feito nos criamos um metodo setter (**__duck**) para tratar o valor que o getter puxou do atributo de origem e um decorator (**@__duck.setter**) que diz para o getter onde ele deve enviar seu valor. O valor enviado pelo getter é recebido por um parametro setter (**namaa**), após recebido o parametro setter tem seu valor tratado e depois associado ao atributo getter, quando o getter recebe o valor retornado pelo parametro setter ele retorna seu atributo getter de volta ao atributo de origem.