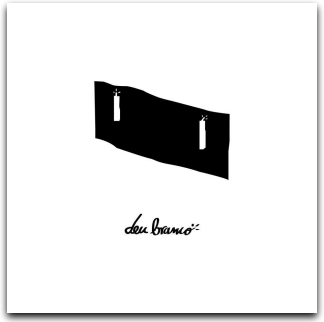


Raça - Deu branco (2014)

Live de Python #64

Orientação a objetos #3

PyLadies na Python Brasil[14]

por PyLadies Brasil

R$ 1.300

apoiados por **2 pessoas**

6% **17** dias restantes

flex **Meta R$ 21.244**

Campanha Flexível

Apoiar este projeto

Você pode apoiar este projeto até o dia 20/09/2018 às 23h59m59s

Natal, RN Ciência e Tecnologia

https://www.catarse.me/pyladies_na_python_brasil_14_2018

Ajude a Live de Python

apoia.se/livedepython

picPay: @livedepython

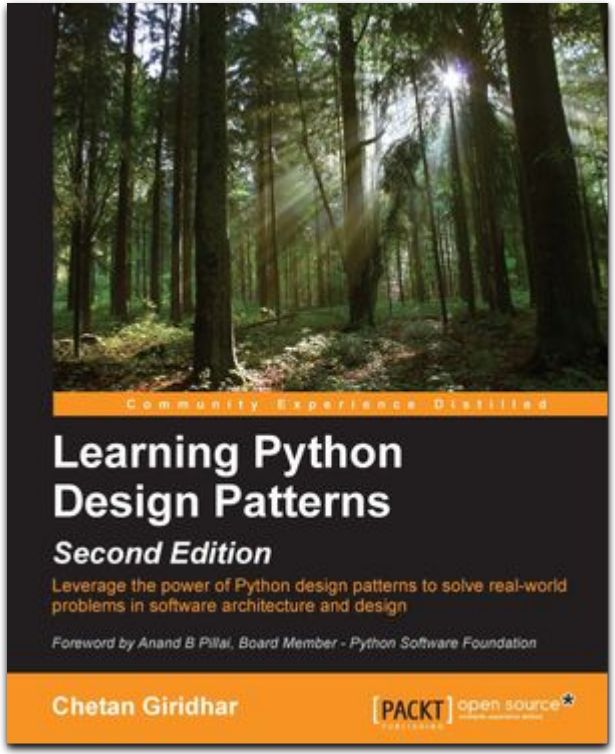
Roteiro

- Encapsulamento
- Composição
- Primeiros __dunders__ (Métodos mágicos)
  - __new__
  - __init__
  - __del__
  - __str__
- Super

Encapsulamento (v1)

Pra Giridhar, encapsulamento é:

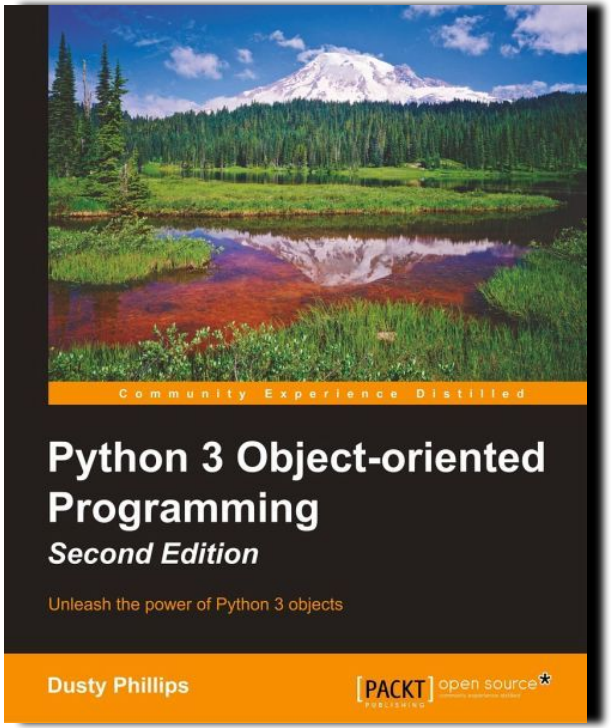
- Atributo oculto do mundo, estado privado
- Clientes da classe não podem modificar o Atributo
- Por convenção devemos colocar ‘_atributo’



Encapsulamento (v2)

Pra Philips, encapsulamento é diferente de “ocultar” informações, é mais abrangente do que isso.

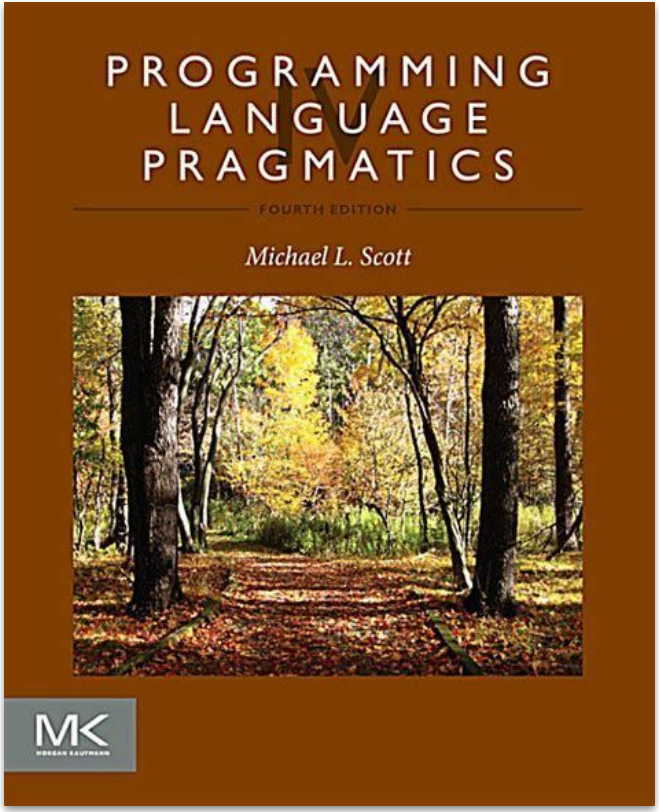
Imagine que você coloque uma carta em uma cápsula do tempo. Se você enterrar ela, a mensagem vai estar oculta, já se a garrafa não for enterrada e for transparente. Você vai conseguir visualizar



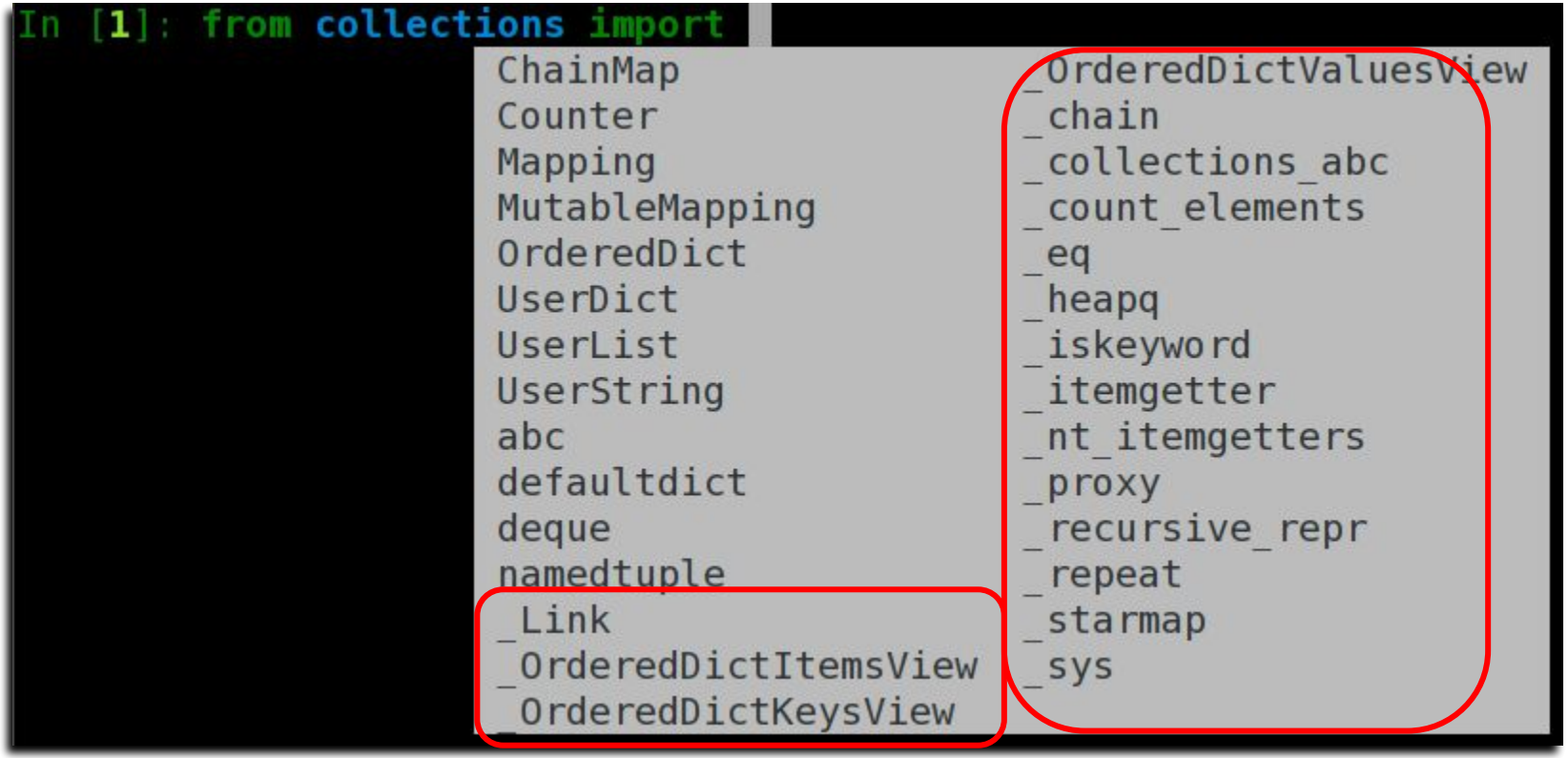
Encapsulamento (v3)

A resposta de Scott é:

Mecanismos de encapsulamento permitem ao programador agrupar dados e as sub-rotinas que operam neles juntos em um único lugar, e ocultar detalhes irrelevantes dos usuários de uma abstração.



Encapsulamento (Módulo)



Encapsulamento (Objeto)

```
In [1]: from collections import namedtuple

In [2]: nt = namedtuple('Type', 'field')

In [3]: dir(nt)[-10:]
Out[3]:
['__str__',
 '__subclasshook__',
 '_asdict',
 '_fields',
 '_fields_defaults',
 '_make',
 '_replace',
 'count',
 'field',
 'index']
```

Composição

```
class Pizzaria:
    def __init__(self):
        self.forno = Forno()

    def pedido(self, pizza):
        self.forno.assar(pizza)
```

Composição é uma maneira de “ligar” objetos sem herança.

Ou seja, se os tipos não “são filhos” (estendem o mesmo tipo) essa é a maneira usada para ligar classes.

A Pizzaria faz uso do forno, mas a pizzaria não é um tipo de forno

Composição

Esse exemplo foi extremamente “radical” porém, qualquer tipo de objeto que fosse definido dentro da classe, que não fosse passado por parâmetros. Seria denominado composição.

```
class Pizzaria:
    def __init__(self):
        self.forno = Forno()

    def pedido(self, pizza):
        self.forno.assar(pizza)
```

```
def Forno:
    def __init__(self):
        self.pizzas = []
        self.lenha = None
```

Fica a questão

Quem interage com a pizzaria quer saber ou tem necessidade de visualizar o forno?

__dunders__ - Métodos especiais

Bom, viemos até aqui empurrando com a barriga certas coisas como __init__.

Vamos discutir alguns métodos especiais para customização do nosso objeto

https://docs.python.org/3/reference/datamodel.html

__init__ Vs. __new__

Muitos chamam o __init__ de construtor, quando na verdade ele é o inicializador. Já o __new__ é construtor.

Geralmente, quase nunca, o __new__ deve ser alterado. Mas, vamos olhar como eles funcionam.

```
class Singleton:
    def __new__(cls):
        if not hasattr(cls, 'criado'):
            cls.criado = super().__new__(cls)
        return cls.criado
```

__new__

Chamado para criar uma nova instância de classes cls. __new __ () é um método estático (mas é um caso especial, não precisa de @staticmethod) que usa a classe da qual uma instância foi solicitada como seu primeiro argumento.

(...)

O valor de retorno de __new __ () deve ser a nova instância de objeto (geralmente uma instância de cls).

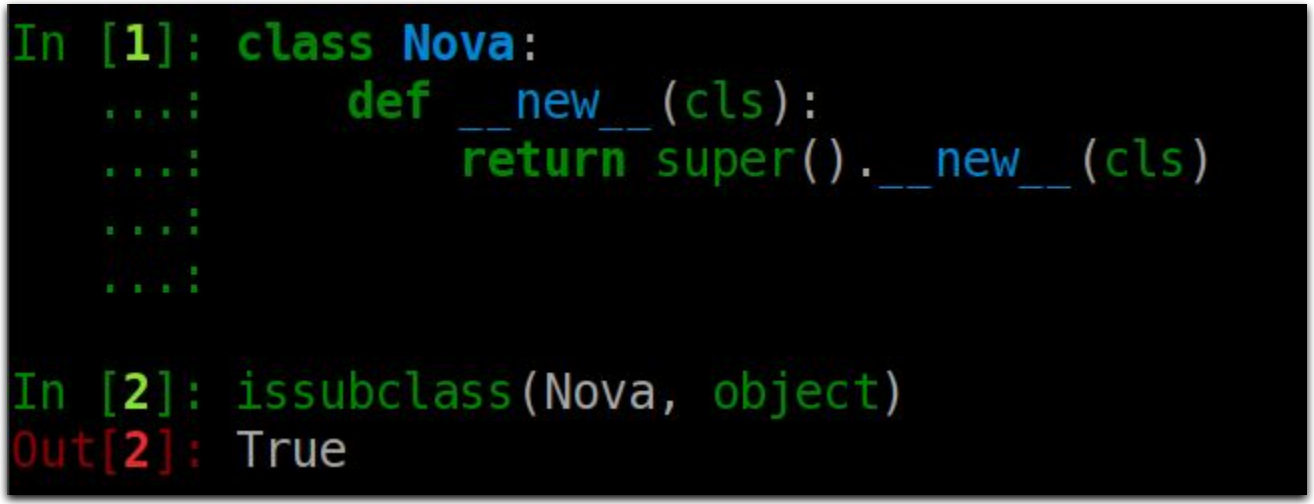
__new__

Implementações típicas criam uma nova instância da classe invocando o método __new __ () da superclasse usando super() .__ new __ (cls [, ...]) com argumentos apropriados e modificando a instância recém-criada conforme necessário antes de retorná-la.

```
class Nova:
    def __new__(cls):
        return super().__new__(cls)
```

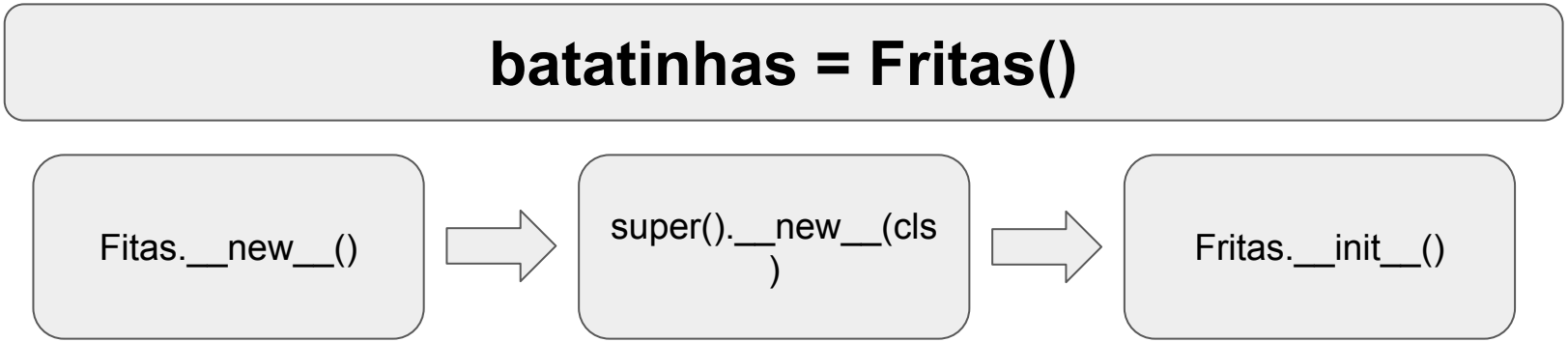
__new__

Implementações típicas criam uma nova instância da classe invocando o método __new __ () da superclasse usando super() .__ new __ (cls [, ...]) com argumentos apropriados e modificando a instância recém-criada conforme necessário antes de retorná-la.



__init__

Chamado depois que a instância foi criada por __new__, mas antes de ser retornada a quem a invocou.



super()

Contudo, a sintaxe do **super** vem a tona agora. Quando estamos falando de herança, a sub-classe deve chamar o __init__ da super classe, ou de sua classe Pai.

Ou seja, o **super**, tem a função de chamar o método definido na super classe

```
class Pássaro:
    def __init__(self, nome):
        self.nome = nome

class CanarinhoPistola(Pássaro):
    def __init__(self, nome, camisa):
        super().__init__(nome)
        self.camisa = camisa
```

__init__

__new __ () e __init __ () trabalham juntos na construção de objetos (__new __ () para criá-lo, e __init __ () para customizá-lo)

OBS:

nenhum valor não-None pode ser retornado por __init __ (); isso fará com que um TypeError seja aumentado em tempo de execução.

__del__ - O finalizador

Chamado quando a instância está prestes a ser destruída. Isso também é chamado de finalizador ou (indevidamente) um destruidor. Se uma classe base tiver um método __del __ (), o método __del __ () da classe derivada, se houver, deverá chamá-lo explicitamente para garantir a exclusão adequada da parte da classe base da instância.

Ou seja:

O que vai acontecer antes do objeto ser de fato destruído

__str__ - A representação em string

Chamado por str (objeto) e as funções internas format () e print () para calcular a representação de string “informal” ou muito bem imprimível de um objeto. O valor de retorno deve ser um objeto de string.

Sério, vamos implementar a fila outra vez


ATÉ A PRÓXIMA, MINHA GENTE.