Prototype

Live de Python #283

Roteiro

1. Prototype

Uma visão geral sobre o padrão

2. Implementação clássica

O GOF

3. Copy

A biblioteca padrão

4. Implementação Pythonica

Como fazer as coisas funcionarem

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto <3

Adriana Cavalcanti, Alan Costa, Alexandre Girardello, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alfredo Braga, Allan Kleitson, Alysson Oliveira, Andre Azevedo, Andre Makoski, Andre Paula, Antonio Filho, Apc 16, Arthur Santiago, Aslay Clevisson, Aurelio Costa, Belisa Arnhold, Bernarducs, Biancarosa, Brisa Nascimento, Bruno Barcellos, Bruno Batista, Bruno Freitas, Bruno Ramos, Bruno Russian, Brunu, Carlos Gonçalves, Celio Araujo, Christian Fischer, Cleiton Fonseca, Controlado, Curtos Treino, Daniel Aguiar, Daniel Bianchi, Daniel Brito, Daniel Souza, Daniel Wojcickoski, Danilo Boas, Danilo Silva, David Couto, David Kwast, Denis Bernardo, Dgeison, Diego Guimarães, Dino, Diogo Faria, Edgar, Eduardo Pizorno, Emerson Rafael, Érico Andrei, Everton Silva, Fabio Barros, Fabio Faria, Fabiokleis, Felipe Adeildo, Felipe Augusto, Felipe Corrêa, Fernanda Prado, Fernandocelmer, Fichele Marias, Francisco Aclima, Frederico Damian, Fulvio Murenu, Gabriel Lira, Gabriel Mizuno, Gabriel Paiva, Gabriel Simonetto, Geilton Cruz, Geisler Dias, Giovanna Teodoro, Giuliano Silva, Guibeira, Guilherme Felitti, Guilherme Ostrock, Guilherme Piccioni, Gustavo Suto, Haelmo Almeida, Harold Gautschi, Heitor Fernandes, Hellyson Ferreira, Helton, Helvio Rezende, Henri Alves, Henrique Andrade, Henrique Machado, Henriquesebastiao, Herian Cavalcante, Hiago Couto, Hideki, Igor Taconi, Ivan Santiago, Janael Pinheiro, Jean Melo, Jean Victor, Jeferson Vitalino, Jefferson Antunes, Jefferson Silva, Jerry Ubiratan, Jhonata Medeiros, Jlx, Joao Rocha, John Peace, Jonas Araujo, Jonatas Leon, Joney Sousa, Jorge Silva, Jose Barroso, Jose Edmario, Joseíto Júnior, Jose Mazolini, José Predo), Josir Gomes, Jrborba, Juan Felipe, Juliana Machado, Julio Franco, Julio Silva, Kaio Engineer, Kaio Peixoto, Leandro O., Leandro Pina, Leandro Vieira, Leonan Ferreira, Leonardo Mello, Leonardo Nazareth, Lisandro Pires, Lucas Carderelli, Lucas Castro, Lucas Mello, Lucas Mendes, Lucas Nascimento, Lucas Schneider, Luciano Ratamero, Luis Ottoni, Luiz Duarte, Luiz Martins, Luiz Paula, Luiz Perciliano, Mackilem Laan, Marcelo Araujo, Marcelo Fonseca, Marcelo Grimberg, Marcio Freitas, Marcos Almeida, Marcos Oliveira, Maria Santos, Marina Passos, Marlon Rocha, Mateusamorim96, Mateus Lisboa, Matheus Vian, Mírian Batista, Mlevi Lsantos, Murilo Carvalho, Ocimar Zolin, Otávio Carneiro, Patrick Felipe, Pedro Henrique, Peterson Santos, Phmmdev, Prof Santana, Pytonyc, Rafael Faccio, Rafael Ferreira, Rafael Fontenelle, Rafael Lopes, Rafael Romão, Raimundo Ramos, Ramayana Menezes, Renan, Renan Sebastião, Rene Pessoto, Renne Rocha, Ricardo Silva, Ricardo Viana, Richard Sousa, Rinaldo Magalhaes, Rodrigo Barretos, Rogério Nogueira, Rui Jr, Samanta Cicilia, Santhiago Cristiano, Sergio Nascimento, Sherlock Holmes, Shirakawa, Tenorio, Téo Calvo, Tharles Andrade, Thiago Araujo, Thiago Lucca, Thiago Paiva, Tiago, Tiago Emanuel, Tomás Tamantini, Valdir, Varlei Menconi, Vinicius Meneses, Vinicius Silva, Vinicius Souza, Vinicius Stein, Vladimir Lemos, Williamslews, Willian Lopes, Zeca Figueiredo, Zero! Studio

Obrigado você

O padrão

Proto type

Padrões criacionais

Quando nos referimos a padrões de projeto criacionais, estamos nos referindo especificamente a formas de criação de "instâncias".

A preocupação desses padrões está em como um exemplar de uma classe específica será criado, de que forma isso está acontecendo. Com qual mecanismo a instância será criada.

Padrões criacionais

Existem diversos padrões como criacionais, como:

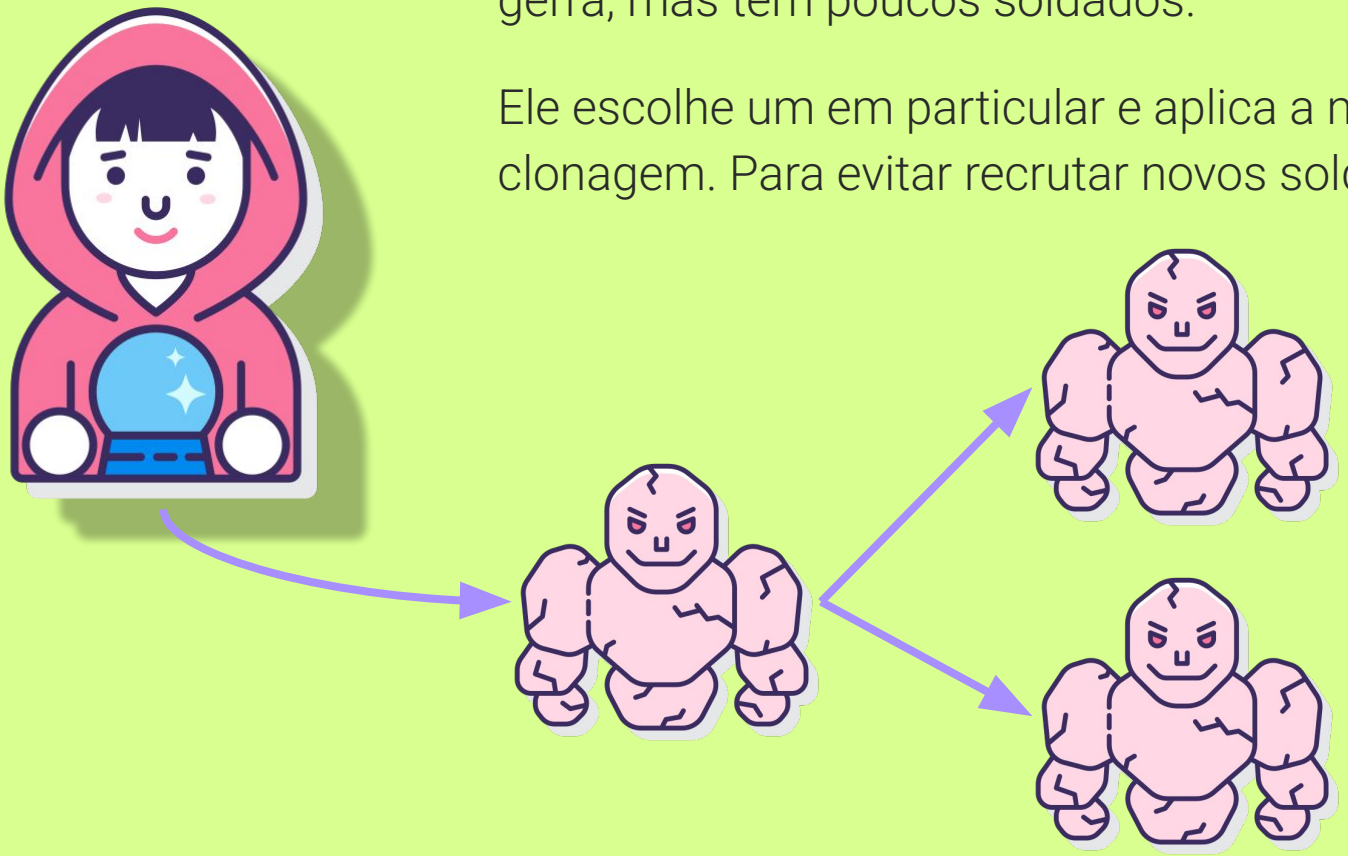
- **Builder**: Criação de instâncias em etapas
- **Factory**: Interfaces padrões para criação de objetos distintos
- **Prototype**: Criação de uma nova instância partindo de um clone de uma existente
- **Singleton**: Garantir que uma classe crie apenas uma instância e forneça um ponto global de acesso.

A alteração da criação nesses padrões é feita por atributos e métodos de classe.

Uma explicação lúdica

Vamos imaginar que Fausto vai lutar em uma gerra, mas tem poucos soldados.

Ele escolhe um em particular e aplica a magia de clonagem. Para evitar recrutar novos soldados



A intenção

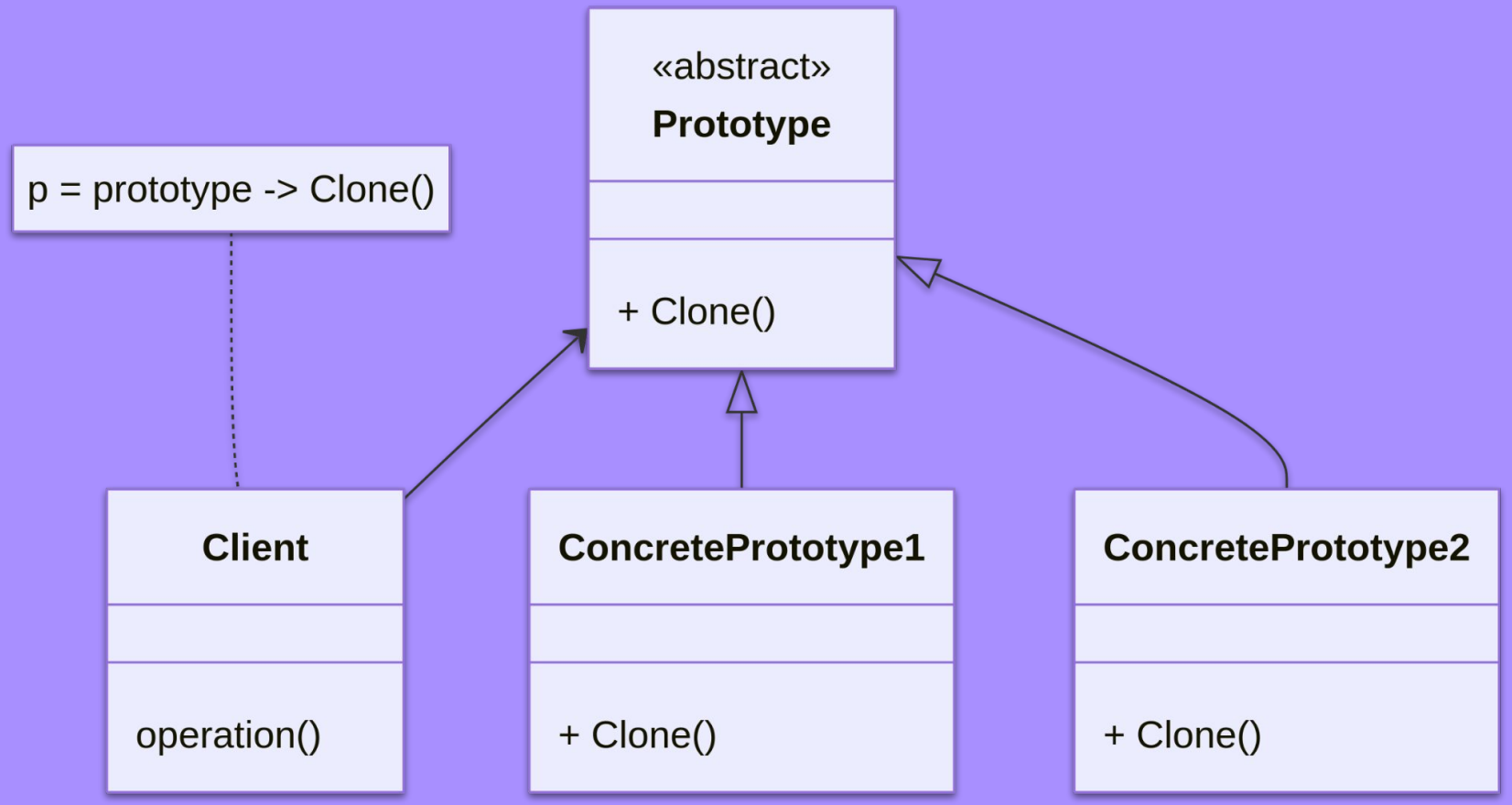
Criar **novos objetos pela cópia de uma instância**. Onde a responsabilidade de "clonar" é do próprio objeto!

Um exemplo concreto

Em python temos um tipo que implementa essa propriedade de se copiar, o dicionário:

```
>>> d = {'chave': 'valor'}
>>> d2 = d.copy()
```

Diagrama


«abstract»
Prototype
+ Clone()
p = prototype -> Clone()
Client
operation()
ConcretePrototype1
+ Clone()
ConcretePrototype2
+ Clone()

Motivos para fazer isso

Algumas justificativas do uso do padrão:

- **Overhead de criação**: Alguns objetos, durante sua inicialização, podem demorar, fazer chamadas externas, ...
- **Acoplamento**: Outros objetos precisam saber como copiar o estado de outro objeto
- **Dependências**: Você pode criar novas classe usando as já existentes
- **...**

Padrões alinhados com esse

- **Registry**: Geralmente criamos várias cópias com variações do objeto e deixamos acessíveis ao sistema todo
- **Factory method**: Uma forma de criar novos objetos partindo de um método, mas usando a criação real
- **Memento**: Copia de estado interno, mas sem criar uma nova instância.

Implementação

Clássica

Implementação clássica

```python
from  abc import ABC, abstractmethod

class Prototype(ABC):
    @abstractmethod
    def clone(self):
        ...
```

Implementação clássica

```python
from  abc import ABC, abstractmethod

class Prototype(ABC):
    @abstractmethod
    def clone(self):
        ...
```

```python
class Guerreiro(Prototype):
    def __init__(self, classe, ataque, defesa):
        self.classe = classe
        self.ataque = ataque
        self.defesa = defesa

    def clone(self) -> Prototype:
        construtor = self.__new__
        estado = {
            'classe': self.classe,
            'ataque': self.ataque,
            'defesa': self.defesa,
        }

        clone = construtor(self.__class__)
        clone.__dict__ = estado
        return clone
```

Implementação clássica

```python
class Guerreiro(Prototype):
    def __init__(self, classe, ataque, defesa):
        self.classe = classe
        self.ataque = ataque
        self.defesa = defesa

    def clone(self) -> Prototype:
        construtor = self.__new__
        estado = {
            'classe': self.classe,
            'ataque': self.ataque,
            'defesa': self.defesa,
        }

        clone = construtor(self.__class__)
        clone.__dict__ = estado
        return clone
```

O construtor do objeto

A classe do objeto

Implementação clássica

```python
class Guerreiro(Prototype):
    def __init__(self, classe, ataque, defesa):
        self.classe = classe
        self.ataque = ataque
        self.defesa = defesa

    def clone(self) -> Prototype:
        construtor = self.__new__
        estado = {
            'classe': self.classe,
            'ataque': self.ataque,
            'defesa': self.defesa,
        }

        clone = construtor(self.__class__)
        clone.__dict__ = estado
        return clone
```

Estado dos atributtos

Repassa para o clone

Implementação clássica

```
g = Guerreiro('Pedra', 7, 31)
new_g = g.clone()
print(new_g.classe)  # Pedra
```

Evitando algo como [clone_1.py]

O cliente viola o encapsulamento, além de acoplar a solução de outro objeto. Para cada tipo novo de "clone" um novo método terá que ser implementado.

```
class Cliente:
    def clone(self, guerreiro):
        construtor = guerreiro.__new__
        estado = {
            'classe': guerreiro.classe,
            'ataque': guerreiro.ataque,
            'defesa': guerreiro.defesa,
        }
        clone = construtor(guerreiro.__class__)
        clone.__dict__ = estado
        return clone
```

Usando dunders a nosso favor [clone_2.py]

Sabendo que todo o estado fica em **__dict__**, podemos simplificar a nossa implementação:

```
def clone(self) -> 'Guerreiro':
    construtor = self.__new__
    clone = construtor(self.__class__)
    clone.__dict__ = self.__dict__.copy()

    return clone
```

A biblioteca nativa

Copy

Copy

Python tem uma biblioteca padrão para fazer cópias, a **copy**.

A biblioteca tem o objetivo exclusivo de copiar objetos usando um protocolo (dunders) implementado em objetos.

As funções disponíveis no módulo são:

- copy.**copy**(): Faz uma cópia rasa de um objeto
- copy.**deepcopy**(): Faz uma cópia profunda do objeto
- copy.**replace**(): Faz uma cópia substituindo atributos

https://docs.python.org/pt-br/3.13/library/copy.html

Exemplo de cópia

De uma certa forma, qualquer objeto em python pode ser "clonado" usando a biblioteca **copy**:

```
>>> from copy import copy

>>> c = C('a', 'b')
>>> c2 = copy(c)
>>> c2.name
'a'
```

Protocolo de cópia

Para que os objetos possam ser clonados por **copy**, eles precisam implementar os protocolos:

- **__copy__**: Para cópia rasa (*shallow*)
- **__deepcopy__**: Para cópia profunda (*deep*)
- **__replace__**: Para cópias com substituições

Você implementa o método da forma como achar que deve ser feito.

___copy___

A implementação que fizemos no método "clone"

```python
class User:
  def __init__(self, name: str, email: str) -> None:
      self.name = name
      self.email = email

  def __copy__(self) -> User:
      print('Estou em __copy__')
      new = self.__class__.__new__(self.__class__)
      new.__dict__ = self.__dict__.copy()
      return new
```

O uso

Dessa forma, quando a função copy for chamada, ela executará o método `__copy__` do objeto:

```
>>> copy.copy(User('a', 'b'))
'Estou em __copy__'
<__main__.User at 0x7ef176c0b610>
```

O protocolo de pickle

Por padrão, qualquer objeto em python pode ser "clonado", pois os objetos `type` e `object` (a base de todos os objetos) implementam os métodos:

- **__reduce__**: Protocolo que gera o "valor de redução"
  - Uma tupla com informações de como construir o objeto
- **__reduce_ex__**: Protocolo de redução versionado

___reduce___

```
class C:
  def __init__(self, a='a', b='b'):
      self.val_0 = a
      self.val_1 = b

>>> C().__reduce__()
(<function copyreg._reconstructor(cls, base, state)>,
 (__main__.C, object, None),
 {'val_0': 'a', 'val_1': 'b'})
```

__reduce__

```
class C:
  def __init__(self, a='a', b='b
      self.val_0 = a
      self.val_1 = b

>>> C().__reduce__()
(<function copyreg._reconstructor(cls, base, state)>,
 (__main__.C, object, None),
 {'val_0': 'a', 'val_1': 'b'})
```

Uma forma de reconstruir o objeto

__reduce__

```
class C:
  def __init__(self, a='a', b=
      self.val_0 = a
      self.val_1 = b

>>> C().__reduce__()
(<function copyreg._reconstructor(cls, base, state)>,
 (__main__.C, object, None),
 {'val_0': 'a', 'val_1': 'b'})
```

A classe e de quem ela herda

__reduce__

```
class C:
  def __init__(self, a='a', b=
      self.val_0 = a
      self.val_1 = b

>>> C().__reduce__()
(<function copyreg._reconstructo
 (__main__.C, object, None),
 {'val_0': 'a', 'val_1': 'b'})
```

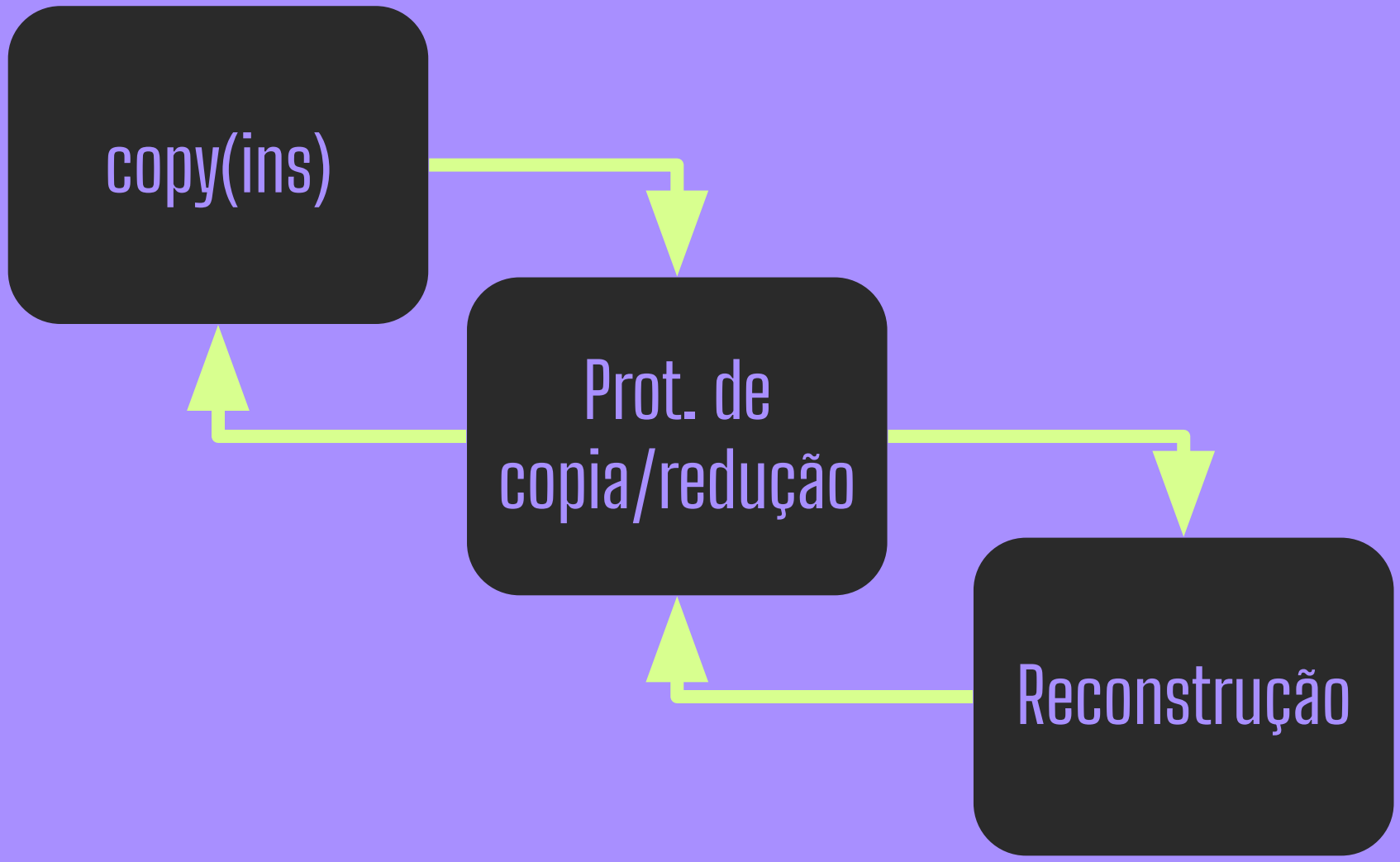
O estado da instância

___reduce_ex___

Tem o mesmo objetivo de reduce, mas conta com versionamento no protocolo de redução

```
>>> c.__reduce_ex__(4)
(<function copyreg.__newobj__(cls, *args)>,
 (__main__.C,),
 {'name': 'a', 'email': 'b'},
 None,
 None)
```

O que copy faz de fato?



Pseudo código da reconstrução

Após gerar a redução algo como isso é feito pelo copy (copy._reconstruct):

```
>>> func, args, state, *_ = c.__reduce__()
>>> copia = func(*args)
>>> copia.__dict__ |= state
>>> vars(copia)
{'name': 'a', 'email': 'b'}
```

Cópias rasas

Cópias rasas mantém a referência para o objeto original. Isso pode ser um problema em alguns casos:

```
class C:
    def __init__(self):
        self.state = [1, 2, 3]

>>> c0 = C()
>>> c1 = copy(c0)
>>> c0.state.append(4)
>>> c1.state
[1, 2, 3, 4]
```

Cópia profunda

Diferente da cópia rasa, que pega o estado do objeto e recria em uma nova construção, a cópia profunda faz uma cópia de todos os objetos internos do estado.

```
from copy import deepcopy

>> deepcopy(c)
<__main__.C at 0x7851646d9450>
```

Cópias profundas

A diferença nesse caso é que os objetos internos não mantém mais a mesma referência, eles foram de fatos clonados também:

```
class C:
    def __init__(self):
        self.state = [1, 2, 3]

>>> c0 = C()
>>> c1 = deepcopy(c0)
>>> c0.state.append(4)
>>> c1.state
[1, 2, 3]
```

___replace___

O protocolo de replace é bastante recente, inserido no python 3.13, e permite que façamos cópias com substituição de valores. Algo como:

```
from copy import replace
from datetime import date

>>> nascimento = date(1993, 3, 6)
>>> replace(nascimento, day=10)
datetime.date(1993, 3, 10)
```

Isso é padrão?

Infelizmente não, o protocolo de pickle não suporta copy.replace, ele deve ser implementado no próprio objeto:

```
class C:
    def __init__(self, a='a', b='b'):
        self.a = a
        self.b = b

    def __replace__(self, **vals):
        new = deepcopy(self)
        new.__dict__ = copy | {
            key: value for key, value in attrs.items() if key in copy
        }
```

Objetos que implementam replace

Alguns objetos da biblioteca padrão implementam o protocolo:

- collections.namedtuple
- dataclasses.dataclass
- datetime
  - datetime
  - date
  - time

```
from copy import replace
from datetime import date

>>> nascimento = date(1993, 3, 6)
>>> replace(nascimento, day=10)
datetime.date(1993, 3, 10)
```

Implementação

Pytho
nica

Copiando `self`

Uma estratégia que pode ser usada para cópias é usar o **copy.copy** e copiar `**self**`. Resolvendo o padrão:

```python
class User:
    def __init__(self, name: str, email: str):
        self.name = name
        self.email = email

    def clone(self) -> User:
        return copy(self)
```

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto <3

Referências

Código fonte:

- copyreg: https://github.com/python/cpython/blob/3.13/Lib/copyreg.py
- copy: https://github.com/python/cpython/blob/3.13/Lib/copy.py

Documentação:

- copy: https://docs.python.org/pt-br/3.13/library/copy.html
- pickle: https://docs.python.org/3.13/library/pickle.html

Livro:

- Design Patterns: Elements of Reusable Object-Oriented Software; Gamma at al.