Singleton / Borg

Live de Python # 273

Roteiro

1. Um passo pra trás

Uma base para entende ro padrão

2. Singleton

O padrão GOF

3. Os problemas

Linguagens dinâmicas vs identidade

4. Borg

Uma solução idiomática alternativa

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto <3

Ademar Peixoto, Adriana Cavalcanti, Adriano Casimiro, Alexandre Girardello, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alfredo Braga, Allan Kleitson, Alysson Oliveira, Andre, Andre Azevedo, Andre Makoski, Andre Paula, Apc 16, Aslay Clevisson, Aurelio Costa, Bernarducs, Brisa Nascimento, Bruno Barcellos, Bruno Batista, Bruno Freitas, Bruno Lira, Bruno Lopes, Bruno Ramos, Bruno Santos, Caio Nascimento, Carlos Gonçalves, Carlos Ramos, Cecilia Oliveira, Celio Araujo, Christian Fischer, Daniel Aguiar, Daniel Bianchi, Daniel Brito, Daniel Real, Daniel Souza, Daniel Wojcickoski, Danillo Ferreira, Danilo Boas, Danilo Silva, David Couto, David Kwast, Denis Bernardo, Dgeison, Diego Guimarães, Dino, Edgar, Eduardo Oliveira, Elton Guilherme, Emerson Rafael, Ennio Ferreira, Érico Andrei, Everton Silva, Fabio Barros, Fábio Barros, Fabio Faria, Fabiokleis, Fabricio Biazzotto, Felipe Augusto, Felipe Corrêa, Fernanda Prado, Filipe Oliveira, Francisco Aclima, Frederico Damian, Fulvio Murenu, Gabriel Lira, Gabriel Mizuno, Gabriel Paiva, Gabriel Ramos, Gabriel Simonetto, Geilton Cruz, Geisler Dias, Giovanna Teodoro, Giuliano Silva, Guilherme Felitti, Guilherme Ostrock, Guilherme Piccioni, Guilherme Silva, Gustavo Suto, Haelmo Almeida, Harold Gautschi, Heitor Fernandes, Helton, Helvio Rezende, Henri Alves, Henrique Andrade, Henrique Machado, Henriquesebastiao, Henrique Sebastião, Hiago Couto, Igor Cruz, Igor Taconi, Ivan Santiago, Jairo Lenfers, Janael Pinheiro, Jean Victor, Jeferson Vitalino, Jefferson Antunes, Jlx, Joao Rocha, Jonas Araujo, Jonatas Leon, Jônatas Silva, Jorge Silva, Jornada Filho, Jose Barroso, Jose Edmario, José Gomes, Joseíto Júnior, Jose Mazolini, Jose Terra, Josir Gomes, Jrborba, Juan Felipe, Juliana Machado, Julio Batista-silva, Julio Franco, Júlio Sanchez, Kael, Kaio Peixoto, Leandro O., Leandro Pina, Leandro Vieira, Leonan Ferreira, Leonardo Alves, Leonardo Mello, Leonardo Nazareth, Letícia Sampaio, Logan Merazzi, Lucas Carderelli, Lucas Castro, Lucas Lattari, Lucas Mello, Lucas Mendes, Lucas Moura, Lucas Nascimento, Lucas Oliveira, Lucas Schneider, Lucas Simon, Luciano Ratamero, Luciano Teixeira, Luiz Carlos, Luiz Duarte, Luiz Martins, Luiz Paula, Luiz Perciliano, Mackilem Laan, Marcelo Araujo, Marcelo Fonseca, Marcio Freitas, Marcio Junior, Marco Mello, Marcos Almeida, Marcos Gomes, Marcos Oliveira, Marcos Rodrigues, Marina Passos, Marlon Rocha, Mateusamorim96, Mateus Lisboa, Matheus Vian, Mírian Batista, Murilo Carvalho, Murilo Viana, Ocimar Zolin, Otávio Carneiro, Patrick Felipe, Pedro Henrique, Peterson Santos, Philipe Vasconcellos, Phmmdev, Prof Santana, Pydocs Pro, Pytonyc, Rafael Araújo, Rafael Costa, Rafael Faccio, Rafael Paranhos, Rafael Romão, Raimundo Ramos, Ramayana Menezes, Renan, Renan Sebastião, Renã Pedroso, Renato José, Rene Pessoto, Renne Rocha, Ricardo Silva, Ricardo Viana, Richard Costa, Rinaldo Magalhaes, Rodrigo Barretos, Rodrigo Oliveira, Rodrigo Vaccari, Rodrigo Vieira, Rogério Nogueira, Rui Jr, Samanta Cicilia, Santhiago Cristiano, Selmison Miranda, Sergio Nascimento, Shinodinho, Téo Calvo, Tharles Andrade, Thedarkwithin, Thiago Araujo, Thiago Borges, Thiago Lucca, Thiago Paiva, Tiago, Tiago Henrique, Tomás Tamantini, Tony Dias, Tyrone Damasceno, Valdir, Victor Reis, Vinicios Lopes, Vinicius Silva, Vinicius Souza, Vinicius Stein, Vitor Silva, Vladimir Lemos, Wagner Gabriel, Williamslews, Willian Lopes, Zeca Figueiredo

Obrigado você

Alguns conceitos antes

Um passo para trás

Forma de bolo

```python
class Aluno:
    def __init__(self, nome, ra, curso):
        self.nome = nome
        self.ra = ra
        self.curso = curso

    def nota(
        self,
        avaliação: Literal['p1', 'p2', 'p3', 'final'],
        diciplina
    ):
        portal.checa_avaliação(self.ra, avaliação, diciplina) # abstração!
```

Forma de bolo

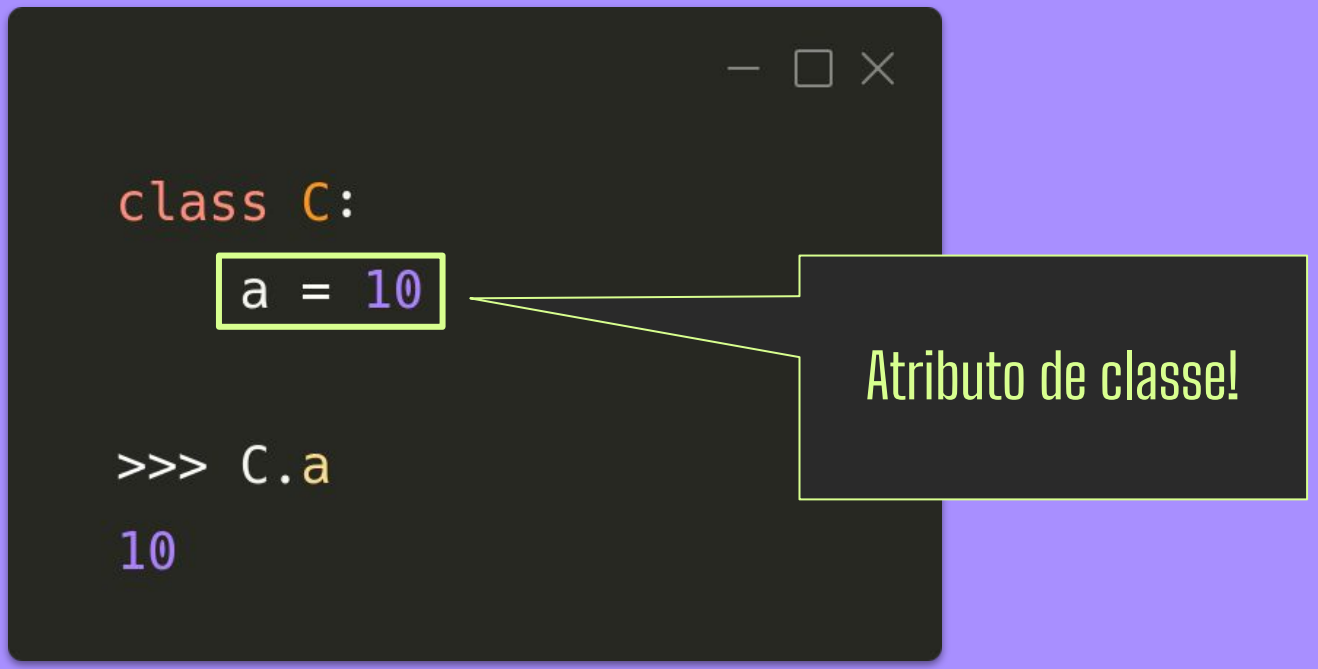
```python
class Aluno:
    def __init__(self, nome, ra, curso):
        self.nome = nome
        self.ra = ra
        self.curso = curso

    def nota(
        self,
        avaliação: Literal['p1', 'p2', 'p3', 'final'],
        diciplina
```

```python
fausto = Aluno('Fausto', '0816192313', 'Tecnologia em magias')
fausto.nota('final', 'futurologia 1')
```

Classe vs Objeto vs Instância

Uma coisa que necessariamente precisa ficar clara antes de discutirmos o padrão é que *__tudo__* em python são **objetos**.

Classes são objetos!



Classe vs Objeto vs Instância

Classe são instâncias de **type**.

```
class C:
    a = 10

>>> C.a
10
```

```
>>> isinstance(C, type)
True

>>> C.__class__
type
```

Métodos de classe

Não precisa da instância pra ser chamado

```
class C:
    @classmethod
    def metodo_de_classe(cls):
        return 'Fui chamado!'

>>> C.metodo_de_classe()
'Fui chamado!'
```

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 qual forma isso está acontecer. 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.

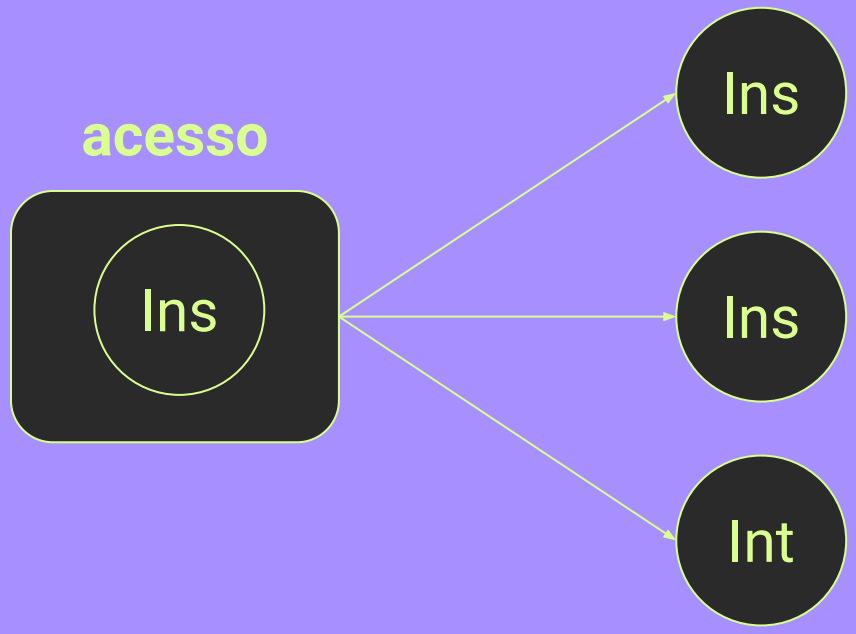
A alteração da criação nesses padrões é feita por atributos e métodos de classe.

Singleton

O padrão GOF

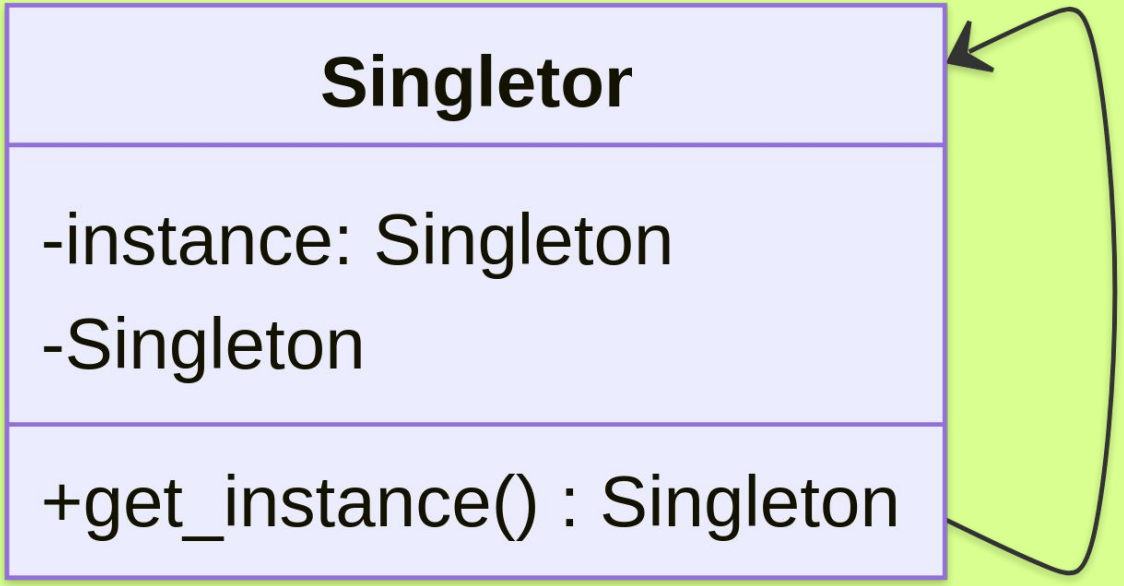
Singleton

"Garantir que a **classe tenha somente uma instância** e fornecer **um ponto global de acesso** para a mesma".



GOF

A instância deve ser obtida sem chamar o construtor. Devemos pedir ela por um método.



A implementação original [gof.py]

```python
class Singleton:
    instance: Singleton

    @classmethod
    def get_instance(cls) -> Singleton:
        if not hasattr(cls, 'instance'):
            cls.instance = Singleton()
        return cls.instance
```

A implementação original

```python
class Singleton:
    instance: Singleton

    @classmethod
    def get_instance(cls) -> Singleton:
        if not hasattr(cls, 'instance'):
            cls.instance = Singleton()
        return cls.instance
```

Ponto único de acesso

A implementação original

```python
class Singleton:
    instance: Singleton

    @classmethod
    def get_instance(cls) -> Singleton:
        if not hasattr(cls, 'instance'):
            cls.instance = Singleton()
        return cls.instance
```

Ponto único de acesso

Atributo de classe

A implementação original

```python
class Singleton:
    instance: Singleton

    @classmethod
    def get_instance(cls) -> Singleton:
        if not hasattr(cls, 'instance'):
            cls.instance = Singleton()
        return cls.instance
```

Criação da instância!

Exemplos de uso [0]

Na biblioteca padrão temos uma forma de uso clara para o uso de singletons e elas usam uma interface padrão. A biblioteca de **logging**:

```python
import logging

logger = logging.getLogger()

logger.warning('Tá certo isso Arnaldo?')
```

Exemplo de uso [1]

Outro caso bem comum, são os acessos compartilhados.

Imagine toda vez que precisar se comunicar com algo, ter que abrir uma nova conexão.

Isso é realmente necessário?

```python
class Database:
    instance: Database

    @classmethod
    def get_instance(cls) -> Database:
        if not hasattr(cls, 'instance'):
            cls.instance = Database()
        return cls.instance

    def __init__(self):
        self.conn = self._get_conn()

    def _get_conn(self):
        sleep(3)
        return 'Con'
```

Exemplo de uso [1]

Outro caso bem comum, são os acessos compartilhados.

Imagine toda vez que precisar se comunicar com algo, ter que abrir uma nova conexão.

Isso é realmente necessário?

```python
class Database:
    instance: Database

    @classmethod
    def get_instance(cls) -> Database:
        if not hasattr(cls, 'instance'):
            cls.instance = Database()
        return cls.instance
```

```
>>> d0 = Database()  # demora 3 segundos!
>>> d1 = Database()  # usa o "cache"
```

Exemplos de uso [2]

Outro caso interessante de uso do singleton é o carregamento de um estado global.

Onde capsularmos diversas variáveis em uma classe. Vamos imaginar o carregamento de variáveis de ambiente. Para isso vou usar o Pydantic:

```python
from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
    model_config = SettingsConfigDict(env_file='.env')
    API_KEY: str
```

Exemplos de uso [2]

Dessa forma, garantimos que as variáveis globais serão lidas uma única vez.

```python
from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
    @classmethod
    def get_instance(cls) -> Settings:
        if not hasattr(cls, 'instance'):
            cls.instance = Settings()
        return cls.instance

    model_config = SettingsConfigDict(env_file='.env')
    API_KEY: str
```

Problemas

Não vivemos só de GOF!

Um problema linguístico

Basicamente, o padrão GOF leva em conta linguagens estáticas, que constroem objetos usando o operador **new**:

```
Batata name = new Batata()
```

Por conta disso, a opção mais certeira era evitar o uso do método construtor, o deixando privado.

O construtor em python, um sigleton diferente [construtor.py]

Em python não existem métodos privados, não é possível privar o construtor. Logo, a forma mais simples de resolver o problema é usar o ponto de acesso no construtor:

```python
class Singleton:
    instance: Singleton

    def __new__(cls) -> Singleton:
        if not hasattr(cls, 'instance'):
            cls.instance = super().__new__(cls)
        return cls.instance
```

Dessa forma, temos a mesma instância

```python
class Singleton:
    instance: Singleton

    def __new__(cls) -> Singleton:
        if not hasattr(cls, 'instance'):
            cls.instance = super().__new__(cls)
        return cls.instance
```

```
>>> s0 = Singleton()
>>> s1 = Singleton()
>>> s2 = Singleton()

>>> s0 is s1, s1 is s2, s2 is s0
(True, True, True)
```

Dessa forma, temos a mesma instância

Toda vez que o construtor for chamado, o objeto retornado será **exatamente** o mesmo. Pois a instância está armazenada na própria classe.

```
>>> s0 = Singleton()
>>> s1 = Singleton()
>>> s2 = Singleton()

>>> s0 is s1, s1 is s2, s2 is s0
(True, True, True)
```

Padrões de projeto?

Are Design Patterns Missing Language Features

On various places, it has been claimed that use of DesignPatterns, especially complex ones like VisitorPattern, are actually indicators that the language being used isn't powerful enough. Many DesignPatterns are by convention rather than encapsulable in a library or component, and as such contain repetition and thus violate OnceAndOnlyOnce. If it didn't contain at least *some* repetition, or something that could be Refactored out, then it wouldn't be a pattern.

https://wiki.c2.com/?AreDesignPatternsMissingLanguageFeatures

Singleton podem ser substituídos por:

A list of DesignPatterns and a language feature that (largely) replaces them:

```
VisitorPattern  ..............  GenericFunctions (MultipleDispatch)
FactoryPattern  ..............  MetaClasses, closures
SingletonPattern .............  MetaClasses
IteratorPattern...............  AnonymousFunctions
             (used with HigherOrderFunctions,
              MapFunction, FilterFunction, etc.)
InterpreterPattern............  Macros (extending the language)
              EvalFunction, MetaCircularInterpreter
              Support for parser generation (for differing syntax)
CommandPattern ...............  Closures, LexicalScope,
              AnonymousFunctions, FirstClassFunctions
HandleBodyPattern.............  Delegation, Macros, MetaClasses
RunAndReturnSuccessor.........  TailCallOptimization
```

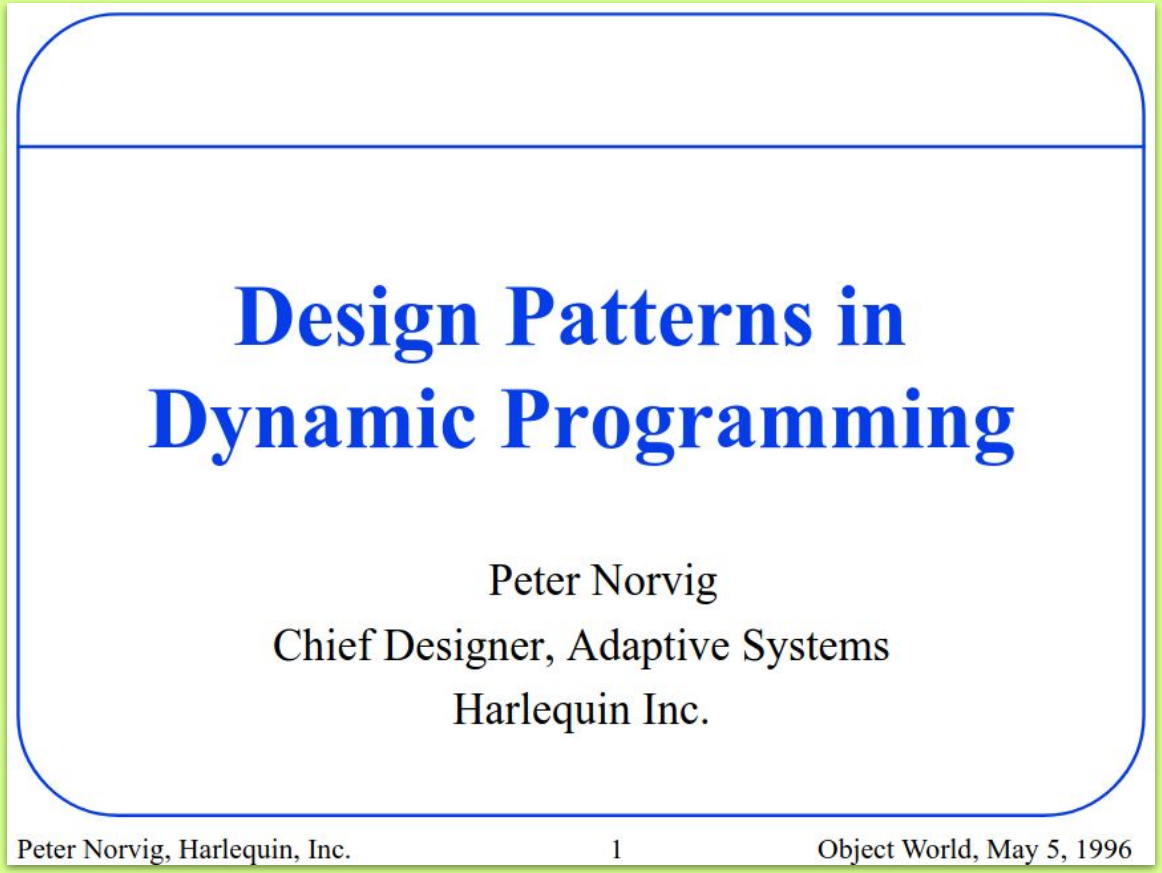
https://wiki.c2.com/?AreDesignPatternsMissingLanguageFeatures

MetaSingleton [metasingleton.py]

A ideia por trás da metaclasse é mudar a forma de como instâncias são construídas. Podendo não só serem usados no próprio objeto, mas em qualquer singleton em potencial.

```python
class MetaSingleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

class Singleton(metaclass=MetaSingleton):
    ...
```

Python não é dinâmico???



http://www.norvig.com/design-patterns/design-patterns.pdf

Singletos não deveriam ser problema

Pattern: Singleton as Memoization

- Can use memoization to implement Singleton pattern
- **Implementation**:

```
(defmethod-memo make ((class SINGLETON))
  ...)
```

- **Invisible Implementation**: Don't need singletons if you can dispatch on constants:

```
define constant s1 = make(<class>, n: 1);
define method m (x == s1) ... end

define constant s2 = make(<class>, n: 2);
define method m (x == s2) ... end
```

Memoize [norvig.py]

O módulo `functools` define as seguintes funções:

`@functools.`**`cache`**`(`*`user_function`*`)`

Cache simples e leve de funções sem vínculo. Às vezes chamado de "memoizar".

Returns the same as `lru_cache(maxsize=None)`, creating a thin wrapper around a dictionary lookup for the function arguments. Because it never needs to evict old values, this is smaller and faster than `lru_cache()` with a size limit.

```python
from functools import cache

@cache
class Singleton:
    ...
```

Funções / Módulos

Embora existam diversas alternativas. O uso de funções, módulos e namespaces podem resolver isso sem a necessidade de nada!

```
class _Singleton: ...

sing = _Singleton()

def Singleton():
    return sing
```

PolySingleton [polysingleton.py]

Um problema comum de singletons é a necessidade de uma classe única em cada contexto da aplicação. Para isso temos **polysingletons**:

```python
from functools import cache

@cache
class PolytSingleton:
    def __init__(self, context=None):
        self.context = context
```

PolySingleton - Como Usar!

Dessa forma, cada parâmetro gera um singleton diferente:

```
>>> s0 = Singleton()
>>> s1 = Singleton()
>>> s2 = Singleton('context2')
>>> s0 is s1, s0 is s2
True, False
```

PolySingleton na stdlib

Um exemplo clássico nesse sentido é a própria biblioteca de logging, que mantém singletons baseados em contextos:

```
import logging

logger_context1 = logging.getLogger('context1')
logger_context2 = logging.getLogger('context2')

logger_context1.warning('Tá certo isso Arnaldo?')
logger_context2.warning('Tá certo isso Arnaldo?')
```

BORG

Solução idiomática!

Borg! [borg_classico.py]

Em "*Five Easy Pieces: Simple Python Non-Patterns*", Martelli vai nos mostrar que talvez o problema que o singleton tenta resolver não precisa ser necessariamente envolver a criação de uma instância única, mas sim o compartilhamento de estado entre diversas instâncias.

```python
class Borg:
    _shared_state = {}
    def __init__(self):
        self.__dict__ = self._shared_state
```

MetaBorg [meta_borg.py]

Da mesma forma que o singleton pode ser metaclasse, o borg também.

```python
class MetaBorg(type):
    _state = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._state:
            cls._state[cls] = cls.__new__(cls)
        return cls._state[cls]

class C(metaclass=MetaBorg): ...
```

RegisBorg [regis_borg.py]

A ideia do RegisBorg é ser um borg guiado por contexto, assim como o polysingleton.

```
class RegisBorg:
    _instances = {}
    def __init__(self, context):
        if not context in self._instances:
            self._instances[context] = self.__dict__
        else:
            self.__dict__ = self._instances[context]
```

Piadas! [regis_meta_borg.py]

Embora não exista uma necessidade real de um meta regis borg. Eu fiz, pq **SIM**

```
class MetaRegisBorg(type):
    _state = {}
    def __call__(cls, context, *args, **kwargs):
        if (cls, context) not in cls._state:
            cls._state[(cls, context)] = cls.__new__(cls)
        return cls._state[(cls, context)]
```

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto <3

Referências

- Livro padrões de projeto (GOF): https://archive.org/details/designpatternsel00gamm/page/126/mode/2up
- wiki c2 - Singleton: https://wiki.c2.com/?SingletonPattern
- Wiki c2 - Padrões são funcionalidades faltantes?: https://wiki.c2.com/?AreDesignPatternsMissingLanguageFeatures
- Peter Norvig - Padrões de projetos em linguagens dinâmicas: http://www.norvig.com/design-patterns/design-patterns.pdf
- Apresentação do Borg por Martelli: https://code.activestate.com/recipes/66531-singleton-we-dont-need-no-stinkin-singleton-the-bo/
- Alex Marterlli - Five Easy Pieces: http://www.aleax.it/5ep.html
- Referência para o nome Borg: https://en.wikipedia.org/wiki/Borg
- Refactoring Guru - Padrão singleton: https://refactoring.guru/pt-br/design-patterns/singleton
- Victor E. Bazterra - Python Borg, and the new MetaBorg?: https://baites.github.io/computer-science/patterns/singleton-series/2018/06/11/python-borg-and-the-new-metaborg.html