Python e os objetos

Eduardo Mendes (z4r4tu5tr4)

Nome:
Eduardo Mendes
Instituição:
Fatec Americana / GAS
Uptime:
725824800s
Email:
mendesxeduardo@gmail.com
git:

github.com/z4r4tu5tr4

O que vamos ver hoje?

- Noções básicas de objetos pythônicos
- Variáveis não são caixas
- Tipos de dados
- Introspecção de objetos pythônicos
- Funções, closures e decoradores
- Classes
- __Dunders__

- Um vetor simpático
- Um baralho quase embaralhável

Como isso vai funcionar?

- Toda opinião é bem vinda, sempre
- Se não entendeu, toda hora é hora de interromper
- Live code, pq sim!

Noções básicas de objetos pythônicos

Tudo são objetos [0]

- Tipagem dinâmica

```
In [1]: a = 7

In [2]: b = "string"
```

Tudo são objetos [1]

- Porém, tipagem forte

```
----> 1 a + b

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Tudo são objetos [2]

```
In [1]: a = 5

In [2]: b = "Eduardo"

In [3]: c = 2.0

In [4]: d = [1,2,3,4,5]

In [5]: e = lambda x: x**2

In [6]: def f():
   ...:     pass
```

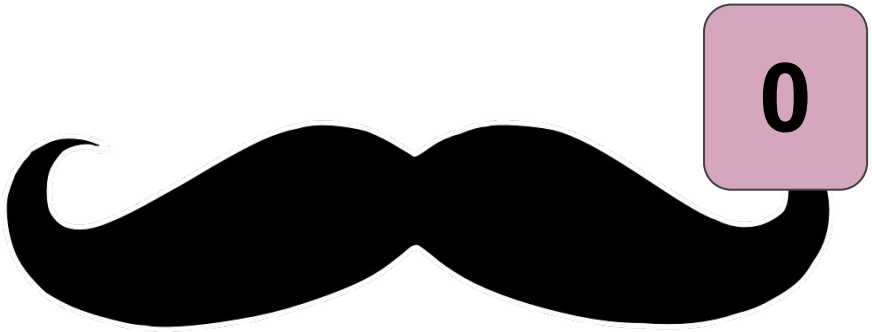
Tudo são objetos [3]

```
>>> [type(e) for e in [a,b,c,d,e,f]]
[<class 'int'>, <class 'str'>, <class 'float'>, <class 'list'>, <class 'function
'>, <class 'function'>]
```

Variáveis não são caixas

(a triste história do bigode)

Variaveis -> Alias

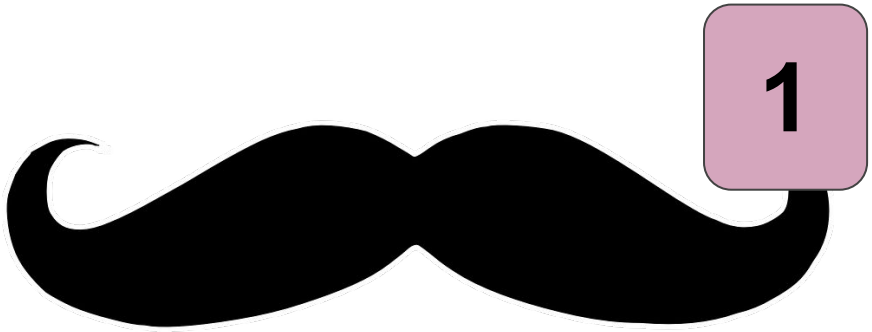


Referências

Objeto

Nome do objeto

Variaveis -> Alias



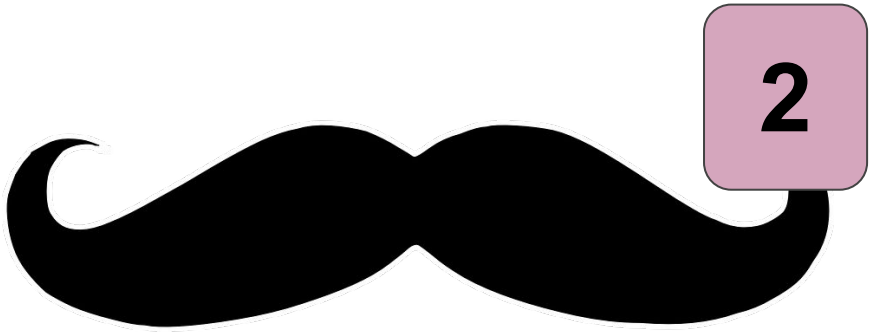

```
bigodinho = mustache()
```

Mustache

bigodinho

Apelido carinhoso

Variaveis -> Alias



Mustache

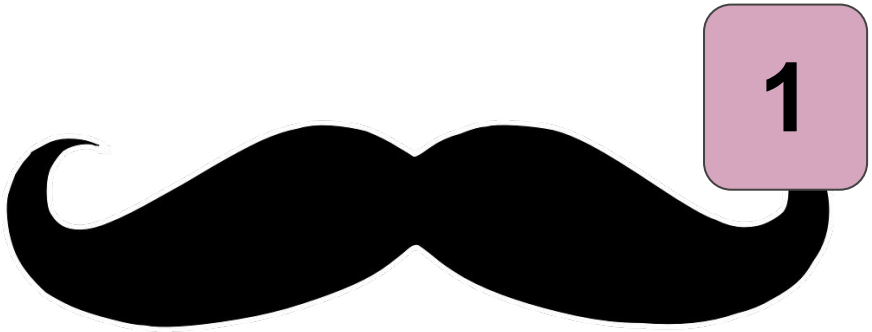
bigodinho

bigode

```
bigodinho = mustache()
```

```
bigode = bigodinho
```

Variaveis -> Alias



Mustache

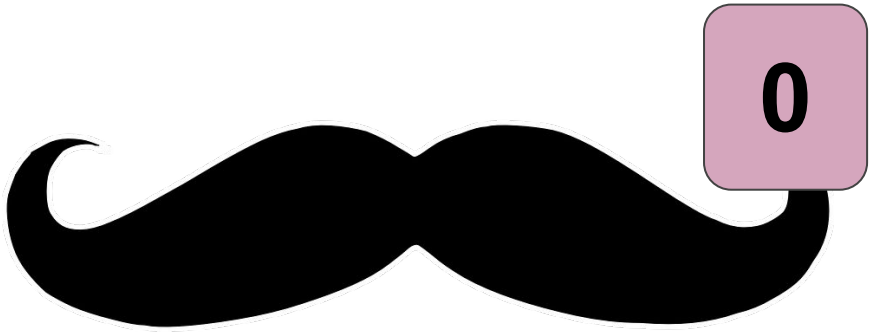
bigodinho

```
bigodinho = mustache()
```

```
bigode = bigodinho
```

```
del bigode
```

Variaveis -> Alias




```
bigodinho = mustache()
```

```
bigode = bigodinho
```

```
del bigode
```

```
del bigodinho
```

Variaveis -> Alias

0

Objeto destruído

Mustache

bigodinho

bigode

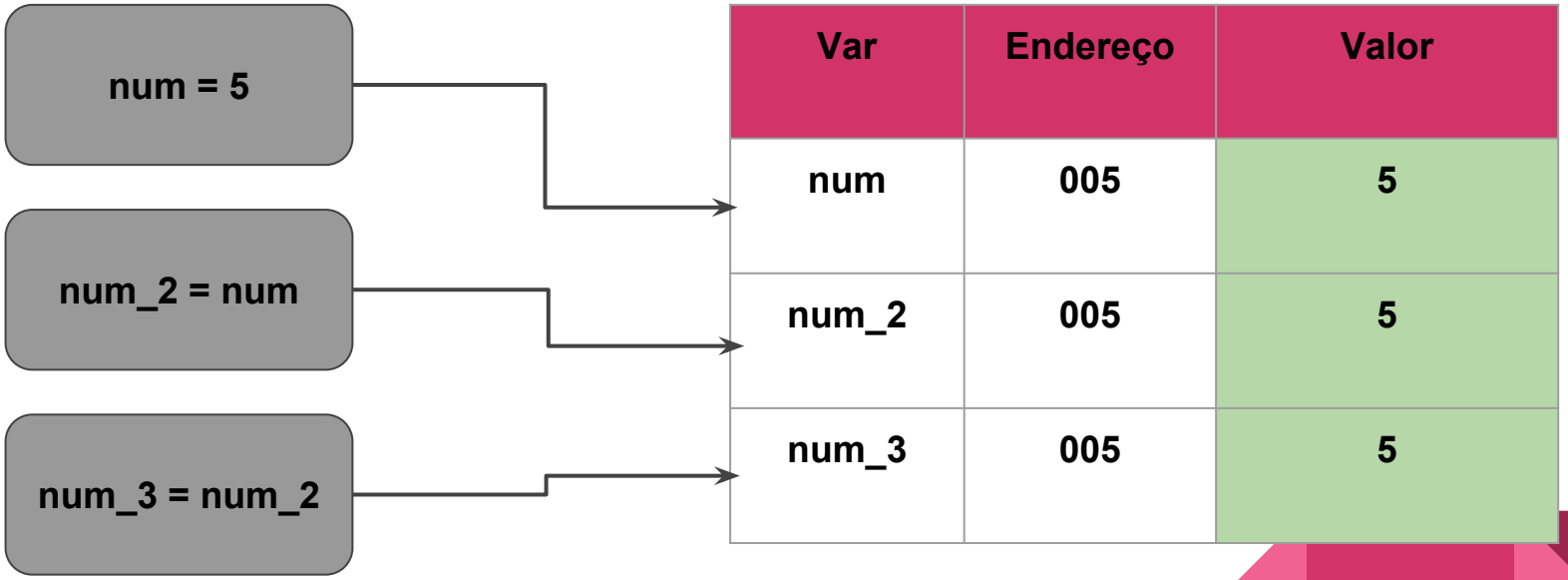
tache()

= bigodinho

del bigode

del bigodinho

Variáveis -> Alias



Variáveis -> Alias

num = 5

num_2 = num

num_3 = num_2

num_3 = 10

Var	Endereço	Valor
num	005	10
num_2	005	10
num_3	005	10

Tipos de dados

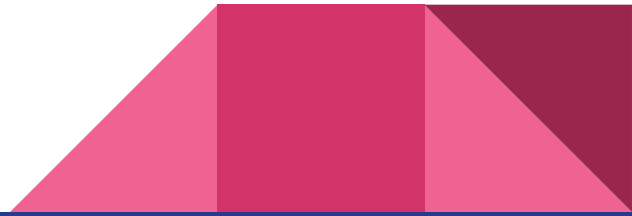
Mutáveis

- Listas
- Dicionários
- Conjuntos
- __setitem__

x

Imutáveis

- Tuplas
- Strings
- Conjuntos “let it go”
- Números



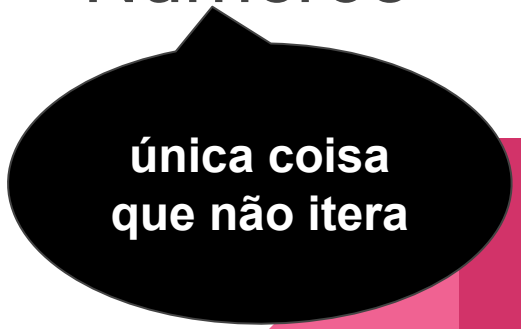
Mutáveis

- Listas
- Dicionários
- Conjuntos
- __setitem__

x

Imutáveis

- Tuplas
- Strings
- Conjuntos “let it go”
- Números



Mutáveis

- Lis
-
-
-

Aqueles que mudam

x

Imutáveis

- Tu
-
- o”
-

Aqueles que NÃO mudam

Tudo que não é um número é iterável

```
In [1]: [x for x in "string"]
Out[1]: ['s', 't', 'r', 'i', 'n', 'g']

In [2]: [x for x in ["l","i","s","t","a"]]
Out[2]: ['l', 'i', 's', 't', 'a']

In [3]: [x for x in {"c","o","n","j","u","n","t","o"}]
Out[3]: ['n', 'o', 'u', 'j', 'c', 't']

In [4]: [x for x in {"d":"i", "c":"i", "o":"n", "a":"r", "i":"o"}]
Out[4]: ['d', 'i', 'o', 'c', 'a']

In [5]: [x for x in ("t","u","p","l","a")]
Out[5]: ['t', 'u', 'p', 'l', 'a']

In [6]: [x for x in 21]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-282901a6ec0d> in <module>()
----> 1 [x for x in 21]

TypeError: 'int' object is not iterable
```

introspecção básica

Type e Dir

Retorna a lista de métodos do objeto

```
>>> type(‘a’)
    <class 'str'>
>>> dir(‘a’)
>>> ['count', 'index','islower', 'lower', 'replace', 'upper']
```

id

Retorna o espaço de memória em que o objeto foi alocado

```
>>> type(‘a’)
    <class 'str'>
>>> id(‘a’)
>>> 725824800
```

Hash(); is

```
>>> hash(1) == hash(1.0)
>>> True
>>> 1.0 is 1
>>> False
```

dis.dis()

```
In [13]: dis("list([1,2,3])")
  1           0 LOAD_NAME                0 (list)
              3 LOAD_CONST               0 (1)
              6 LOAD_CONST               1 (2)
              9 LOAD_CONST               2 (3)
             12 BUILD_LIST               3
             15 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             18 RETURN_VALUE

In [14]: dis("[1,2,3]")
  1           0 LOAD_CONST               0 (1)
              3 LOAD_CONST               1 (2)
              6 LOAD_CONST               2 (3)
              9 BUILD_LIST               3
             12 RETURN_VALUE
```

dis.dis()

```
In [13]: dis("list([1,2,3])")
  1           0 LOAD_NAME                (
              3 LOAD_CONST               (
              6 LOAD_CONST               1 (2)
              9 LOAD_CONST               2 (3)
             12 BUILD_LIST               3
             15 CALL_FUNCTION            (1 positional, 0 keyword pair)
             18 RETURN_VALUE

In [14]: dis("[1,2,3]")
  1           0 LOAD_CONST               0 (1)
              3 LOAD_CONST               1 (2)
              6 LOAD_CONST               2 (3)
              9 BUILD_LIST               3
             12 RETURN_VALUE
```

18 VS 12

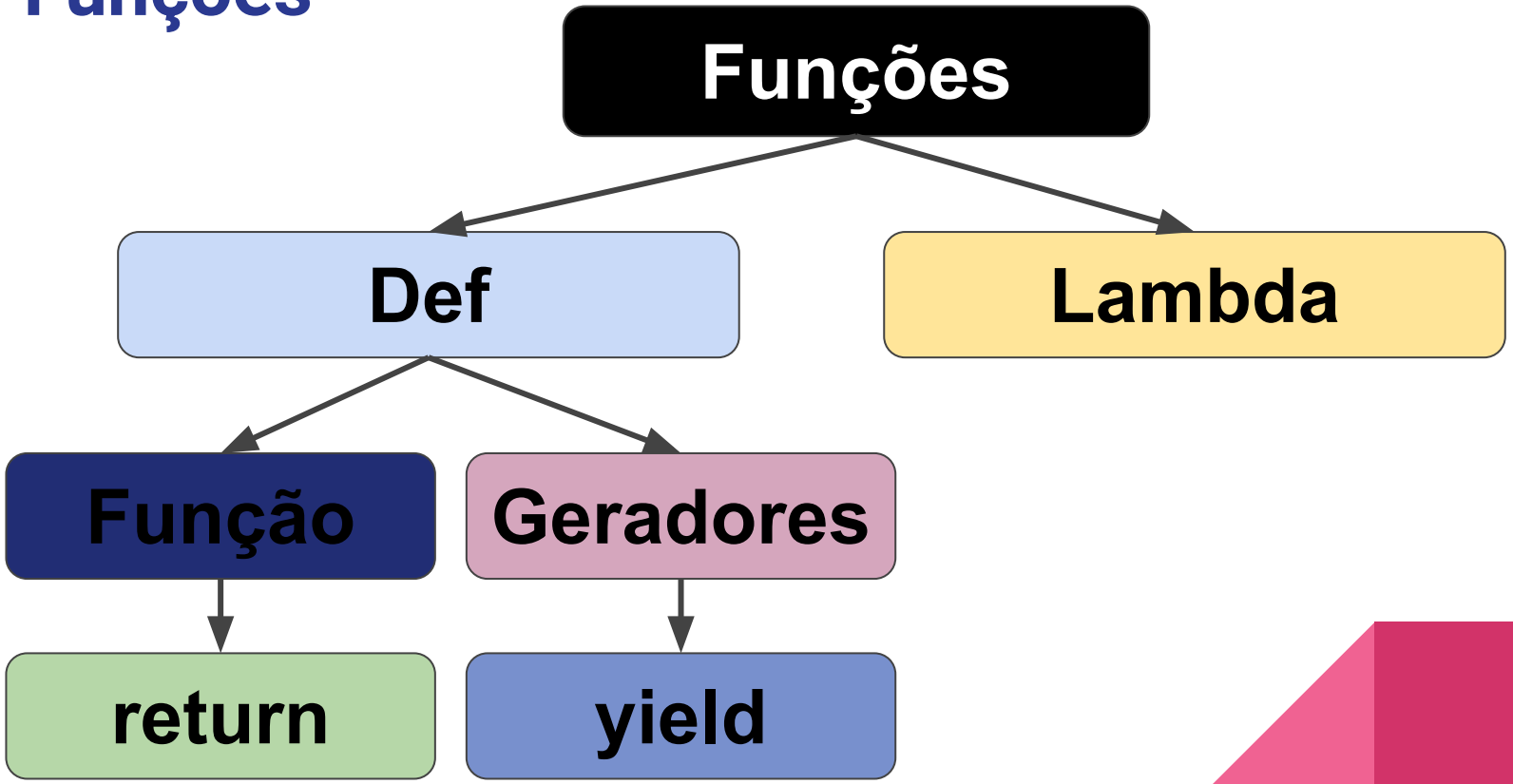
Funções, Closures e decoradores

Funções do escopo built-in

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

https://docs.python.org/3/library/functions.html

Funções



Padrões[0] - Funções

Função nomeada

```
def nome (args):
    return args
```

Função anônima

```
lambda args: op(args)
```

Função geradora

```
def nome (args):
    yield args
```

Padrões[1] - Funções

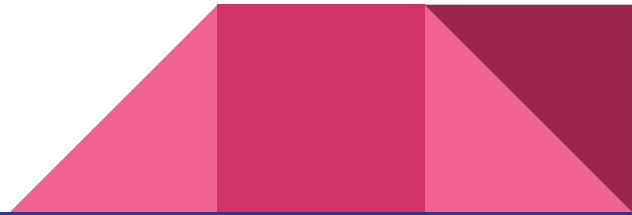
```
def func(a, b=1, *c, **d):
        return a, b, c, d

func( * [1, 2, 3, 4, 5], x=7 )
        # 1 2 (3, 4, 5) {'x': 7}
```

Padrões[2] - Funções

```
def func(a, b=1, *c, **d):
        return a, b, c, d

func( * [1, 2, 3, 4, 5], x=7 )
        # 1 2 (3, 4, 5) {'x': 7}
```

Closures e decoradores [0] - “Teoria” funcional

- Uma pitada leve de programação funcional
  - Como tudo são objetos, qualquer parâmetro passado como argumento é um objeto;
  - Como não temos tipo, uma função pode ser passada para uma função;
  - N funções podem ser passadas N para funções;
  - Podemos fazer aninhamento de funções;
  - Uma função que executa N funções é uma closure



Closures e decoradores[1] - Funções de funções

```
In [16]: func = lambda x: x**2

In [17]: list(map(func, [1,2,3,4,5]))
Out[17]: [1, 4, 9, 16, 25]
```

Closures e decoradores[2] - Finalmente closure

```
In [15]: def closure(func):
    ...:     def func_wrapper():
    ...:         print("Entramos no wrapper")
    ...:         func()
    ...:         print("Saimos do wrapper")
    ...:     return func_wrapper
    ...:
```

Closures e decoradores[3] - Decoradores

```
In [18]: @closure
    ...: def x():
    ...:     print("oi")
    ...:

In [19]: x()
Entramos no wrapper
oi
Saimos do wrapper
```

Classes

O básico

```
In [1]: class oi:
   ...:     pass
   ...:

In [2]: teste = oi()
```

```
In [3]: class oi:
   ...:     def __init__(self, a, b):
   ...:         self.a = a
   ...:         self.b = b
   ...:
   ...:

In [4]: teste = oi()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-8734963fe985> in <module>()
----> 1 teste = oi()

TypeError: __init__() missing 2 required positional arguments: 'a' and 'b'
```

O básico

Sem inicialização

```
In [1]: class oi:
   ...:     pass
   ...:

In [2]: teste = oi()
```

Método inicializador

```
In [3]: class oi:
   ...:     def __init__(self, a, b):
   ...:         self.a = a
   ...:         self.b = b
   ...:
   ...:

In [4]:
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-...> in <module>()
----> 1

TypeError: __init__() missing 2 required positional arguments: 'a' and 'b'
```

O básico

Sem inicialização

```
In [1]: class oi:
   ...:     pass
   ...:

In [2]: teste = oi()
```

```
In [3]: class oi:
   ...:     def __init__(self, a, b):
   ...:         self.a = a
   ...:         self.b = b
   ...:
   ...:

In [4]: teste = oi()
---------------------------------------------------------------------------
TypeError                                 (most recent call last)
<ipython-input-4-8734963fe985> :
----> 1 teste = oi()

TypeError: __init__() missing 2 required positional arguments: 'a' and 'b'
```

O atributo “eu mesmo”

O básico - decoradores de classe

```python
class oi:
    def __init__(self, x):
        self.x = x

    def mul(self, y):
        return self.x * y

    @classmethod
    def mul_class(cls, y):
        return cls.x * y

    @staticmethod
    def mul_static(x, y):
        return x * y
```

Método de instancia

O básico - decoradores de classe

```python
class oi:
    def __init__(self, x):
        self.x = x

    def mul(self, y):
        return self.x * y

    @classmethod
    def mul_class(cls, y):
        return cls.x * y

    @staticmethod
    def mul_static(x, y):
        return x * y
```

Método de classe

O básico - decoradores de classe

```python
class oi:
    def __init__(self, x):
        self.x = x

    def mul(self, y):
        return self.x * y

    @classmethod
    def mul_class(cls, y):
        return cls.x * y

    @staticmethod
    def mul_static(x, y):
        return x * y
```

Método de estático

__dunders__

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

__dunders__ - dois underlines no começo e final

- Os métodos dunders, são métodos que implementam coisas “mágicas” na linguagem
  - Ou seja, você pode usar as funções nativas do python em seu objeto

Alguns bem básicos - iterável

```
class oi:
    def __init__(self, x):
        self.x = x

    def __repr__(self): #print(self)
        return "Classe Oi: {x}".format(x=self.x)

    def __len__(self): #len(self)
        return len(self.x)

    def __getitem__(self, position): #iterador
        return self.x[position]

    def __reversed__(self): #iterador reverso
        return list(reversed(self.x))
```

Inicializador

Alguns bem básicos - iterável

```
class oi:
    def __init__(self, x):
        self.x = x

    def __repr__(self): #print(self)
        return "Classe Oi: {x}".format(x=self.x)

    def __len__(self): #len(self)
        return len(self.x)

    def __getitem__(self, position): #iterador
        return self.x[position]

    def __reversed__(self): #iterador reverso
        return list(reversed(self.x))
```

Representação

Alguns bem básicos - iterável

```
class oi:
    def __init__(self, x):
        self.x = x

    def __repr__(self): #print(self)
        return "Classe Oi: {x}".format(x=self.x)

    def __len__(self): #len(self)
        return len(self.x)

    def __getitem__(self, position): #iterador
        return self.x[position]

    def __reversed__(self): #iterador reverso
        return list(reversed(self.x))
```

Contagem

Alguns bem básicos - iterável

```
class oi:
    def __init__(self, x):
        self.x = x

    def __repr__(self): #print(self)
        return "Classe Oi: {x}".format(x=self.x)

    def __len__(self): #len(self)
        return len(self.x)

    def __getitem__(self, position): #iterador
        return self.x[position]

    def __reversed__(self): #iterador reverso
        return list(reversed(self.x))
```

Iterador simples

Alguns bem básicos - iterável

```
class oi:
    def __init__(self, x):
        self.x = x

    def __repr__(self): #print(self)
        return "Classe Oi: {x}".format(x=self.x)

    def __len__(self): #len(self)
        return len(self.x)

    def __getitem__(self, position): #iterador
        return self.x[position]

    def __reversed__(self): #iterador reverso
        return list(reversed(self.x))
```

Iterador reverso

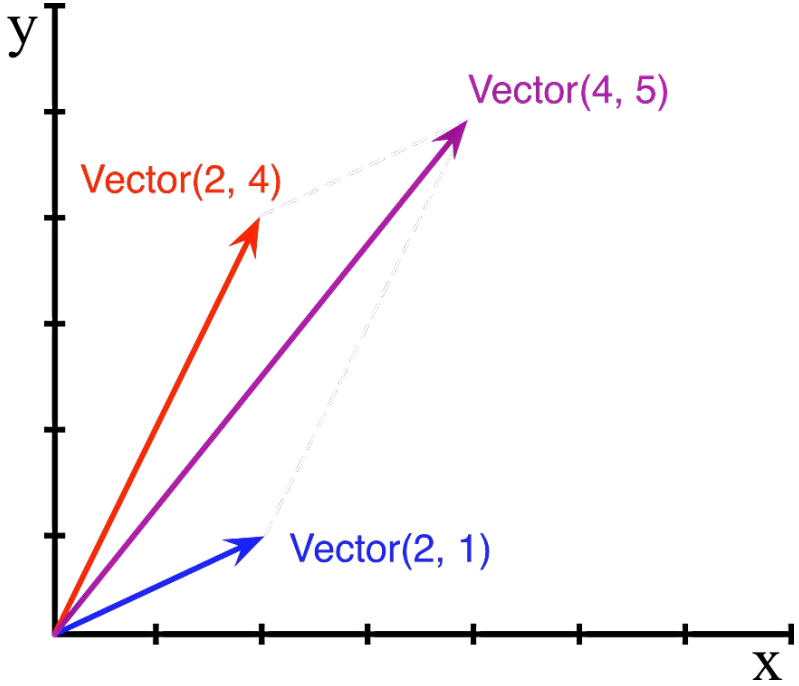
Alguns bem básicos - Sobrecarga de método

```
class oi:
    def __init__(self, x):
        self.x = x

    def __repr__(self): #print(self)
        return "Classe Oi: {x}".format(x=self.x)

    def __add__(self, x): # <class> + val
        return self.x + x

    def __radd__(self, x): # val + <class>
        return  x + self.x

    def __mul__(self, x): # <class> * val
        return self.x * x
```

Pratica 1: Vetores

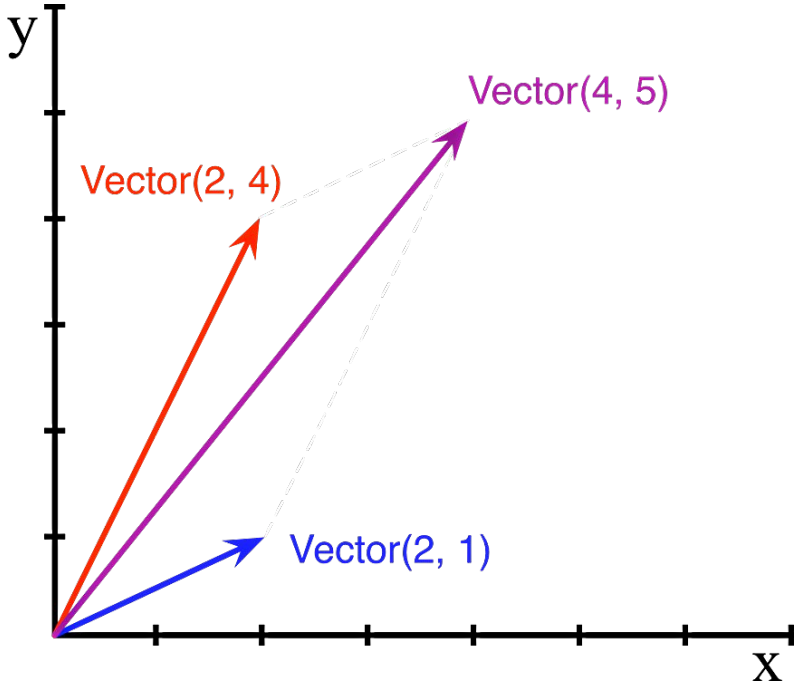
Pratica 1 - Vetor

- Criar uma classe de vetor euclidiano
- Receber x e y
- Ter uma representação #print
- Ser bolleano #bool(<vector>)
- Poder ser somado a outro obj. vetor
- Poder ser multiplicado por um número escalar



Pratica 1 - o que é preciso?

- from math import hypot
- __init__
- __repr__
- __abs__
- __bool__
- __add__
- __mul__



Pratica 1 - Vetor

```
class Vector:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Vector(%r, %r)' % (self.x, self.y)
```

Pratica 1 - Vetor

```
def __abs__(self):
    return hypot(self.x, self.y)

def __bool__(self):
    return bool(abs(self))

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

def __mul__(self, scalar):
    return Vector(self.x * scalar, self.y * scalar)
```

Pratica 2:
Baralho quase embaralhável

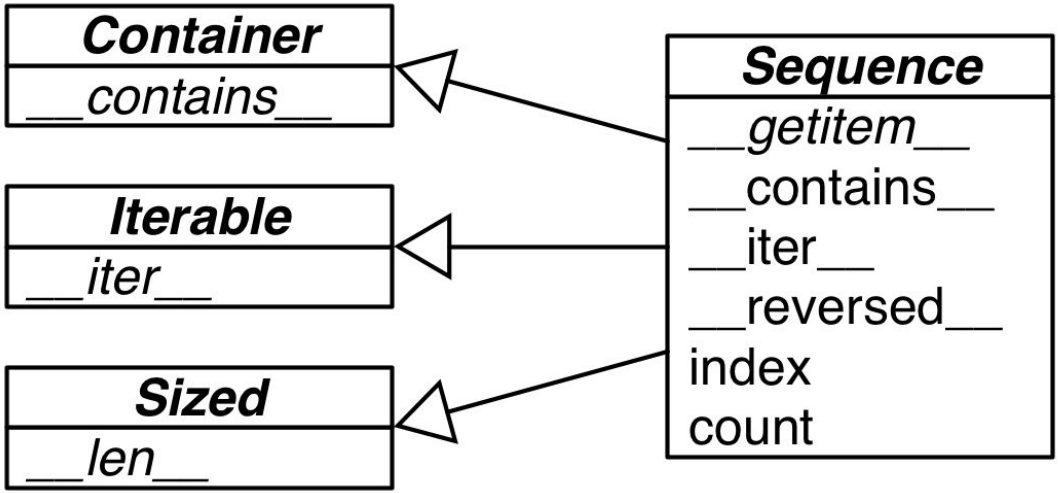
Pratica 2 - Baralho

- Montar uma classe baralho
- Preciso saber quantas cartas
- Quais cartas
- Iterar
- Embaralhar “VALENDO PREMIO”

from collections import namedtuple

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

In [2]: v = namedtuple('Ponto', ['x','y'])

In [3]: v(7,8)
Out[3]: Ponto(x=7, y=8)

In [4]: b = namedtuple('Carta', ['val','naipe'])

In [5]: b
Out[5]: __main__.Carta

In [6]: b('A', 'P')
Out[6]: Carta(val='A', naipe='P')
```

from collections.abc import Sequence



Por fim, o desafio

```
import collections

Carta = collections.namedtuple('Card', ['val', 'naipe'])

class Baralho(collections.abc.Sequence):
    valores = [str(n) for n in range(2, 11)] + list('JQKA')
    naipes = 'Espada Copas Ouro Paus'.split()

    def __init__(self):
        self.cards = [Carta(valor, naipe) for naipe in self.naipes
                                          for valor in self.valores]

    def __len__(self):
        return len(self.cards)

    def __getitem__(self, pos):
        return self.cards[pos]
```

```python
import collections

Carta = collections.namedtuple('Card', ['val', 'naipe'])

class Baralho(collections.abc.Sequence):
    valores = [str(n) for n in range(2, 11)] + list('JQKA')
    naipes = 'Espada Copas Ouro Paus'.split()

    def __init__(self):
        self.cards = [Carta(valor, naipe) for naipe in self.naipes
                                          for valor in self.valores]

    def __len__(self):
        return len(self.cards)

    def __getitem__(self, pos):
        return self.cards[pos]

    def __setitem__(self, pos, oi):
        self.cards[pos] = oi
```

XOXO

Dúvidas?

mendesxeduardo@gmail.com

Algumas referências

- Fluent Python - Ramalho
- Documentação Python 3
- Functional programming Python - Mertz
- Python Cookbook