
cake

Python

Eliane Maciel

eliane.maciel@cakeerp.com

https://github.com/elianemaciel

https://www.linkedin.com/in/elianemaciel/

- Estudante de Ciência da Computação - UCS
- Desenvolvedora de sistemas - Cake ERP
- Desenvolvedora Python desde 2013.
- Organizadora do PyLadies Caxias do Sul e Django Girls
- Organizadora PyCaxias e Python Brasil 2020
- Palestrante nos eventos Tchelinux, PyCaxias e Python Sul.

http://cakeerp.com/

	Aula 1	Aula 2	Aula 3	Aula 4
Manhã	Apresentação	Laços, (for, while)	Orientação a Objeto	Django Framework
	Sintaxe, Virtualenv	Funções	Classes	Instalação, criando um projeto.
	Operadores, Variáveis, Strings	Datetime e Random	Métodos	Interface de Administração
	Git	Exercícios	Herança	Models
Tarde	Listas, Tuplas	Buit in, Expressions		Avaliação
	Dicionários	Exceções	Exercícios	
	Condicionais (if, else)	Decorators, RegeX		
	Exercícios	Exercícios		

Avaliação

- Exercícios
  - GitHub
  - https://github.com/elianemaciel/programa_talentos_nl_2019
- Desafios - HackerRank
  - www.hackerrank.com/programa-de-talentos-nl

- Avaliação

O que é Python?

- É uma linguagem de programação
- Código aberto
- Alto nível
- Comunidade livre, forte, diversificada, alegre e acolhedora
- Tudo é um objeto
- Legibilidade do código fonte

O que é Python?

- Foi criada em 1989 por Guido Van Rossum
- O nome Python foi inspirado no seriado britânico Monty Python
- Python Software Foundation: A missão da Python Software Foundation é promover, proteger e promover a linguagem de programação Python, além de apoiar e facilitar o crescimento de uma comunidade diversificada e internacional de programadores Python.

Por que Python?

"Simples é melhor que complexo" (Tim Peters)

“ Python me ajuda a focar nos meus conceitos em vez de ficar brigando com a linguagem”.

”Python tem sido uma parte importante do Google desde o início, e permanece

assim conforme o sistema cresce e evolui... estamos procurando por mais

pessoas com conhecimento nessa linguagem“. Peter Norvig, diretor de qualidade de busca do Google In.

“Python é Legal!”

Sintaxe

1. Indentações
2. Boa Legibilidade
3. Uso de caracteres reduzido

4. PEPs - Python Enhancement Proposals (propostas de aprimoramento)
5. Padrão de Desenvolvimento:
    1. https://www.python.org/dev/peps/pep-0008/

- https://www.python.org/dev/peps/

Bibliotecas

- PIP - sistema de gerenciamento dos pacotes
- pip freeze
- pip install
- pip – help (mostra os comandos)
- easy_install
- IPython

Variáveis

- Python utiliza tipagem dinâmica, os tipos das variáveis são inferido pelo interpretador em tempo de execução.
- A variável é criada através da atribuição.
- A tipagem é forte, ou seja, o interpretador verifica se as operações são válidas
- Podem ser usados algarismos, letras ou _
- Nunca devem começar com um algarismo

Variáveis

- Atribuição Aumentada
  - -=, *=, /=, //=, %= e **=.
- Não podemos usar palavras-chave naturais : **def, if, while for etc.**
- Python é uma Linguagem de tipagem dinâmica e forte.

Variáveis

Variável é um nome que associamos a um valor ou expressão.

<nome da variável> = <valor que quero armazenar>

- Exemplo:

```
>>>  numero = 20
>>> mensagem = "E aí, Doutor?"
>>> decimal = 20.00
```

Variáveis - Tipos

Númericos: int, float

```
>>> n = 2
>>> f = 2.0
```

Literais: strings (aspas simples ou duplas)

```
>>> telefone = '2145-8970'
```

Lógicos (boolean)

```
>>> aprovada = True
>>> aprovada = False
```

Operadores

+	adição
-	subtração
*	multiplicação
/	divisão
//	divisão inteira
**	exponenciação
%	resto de divisão

or	lógico ou
and	lógico e
not	lógico de negação

<	estritamente menos que
<=	menos ou igual que
>	estritamente maior que
>=	maior ou igual que
==	igual
!=	diferente
is	identidade do objeto
is not	não identidade do objeto

Operadores

Exemplos:

```
>>> a = 2
>>> b = 3
>>> a > b
>>> b > a
```

Strings

Pode se usar aspas simples, duplas ou triplas.

```
>>> string = 'Programando em Python'
>>> string.upper()
>>> string.lower()
>>> string.capitalize()
>>> string.title()
>>> string.swapcase()
```

Strings

```
>>> string = 'Programando em Python'
```

Acessar uma posição específica:

```
>>> string[0]
>>> string[2]
```

De uma posição até outra:

```
>>> string[1:12]
```

Strings

Replace:

<variável>.replace('string que quero mudar', 'nova string')

```
>>> spock = 'Fascinante, capitão Kirk'
>>> spock.replace('Fascinante' , 'Incrível')
```

Strings

São imutáveis.

```
>>> texto = 'Alo Mundo'
>>> texto[0] =  '@'
```

TypeError: 'str' object does not support item assignment

Strings

- # Estilo de formatação ruim
  print('hello ' + name + '!')

  # Estilo de formatação antiga
  print('Hello %s!' % name)

  # Estilo de formatação nova
  print('Hello {}!'.format(name))
- https://docs.python.org/2/library/string.html
- https://docs.python.org/2.7/library/stdtypes.html#string-methods

Strings

- Formatação de Strings:
- >>> frase = "Selecione periodo menor que {numero} meses"
- >>> frase.**format**(numero=5)
- >>> frase2 = "Data Inicial {data_ini} e Data Final {data_fim}"
- >>> frase2.**format**(data_ini='01/02/2018', data_fim='20/02/2018')

Git - Configuração

- GIT - sistema de controle de versão de arquivos

1. git config --global user.name “Thais Vergani”
   - git config user.name
2. git config --global user.email “thais.vergani1@gmail.com”

3. git init: inicializa o repositório no diretório atual

GIT HUB

- GitHub é uma plataforma de hospedagem de código-fonte com controle de versão usando o Git.
- criar repositório
  - git add remote.
- git clone

- https://github.com/elianemaciel/programa_talentos_nl_2019

GIT

- git status
- git add (nome_do_arquivo)
- git commit -m “mensagem do seu commit”
- git log
- gif diff
  - git diff --name-only

Exercícios

Devem ser postados no Github

Estruturas

Listas

<variável> = [info1, info2, info3]

```
>>> exemplo = ['Gato', 9, True]
>>> exemplo[0]
>>> exemplo2 = ['Gato', 9, True, ['azul', 'verde']]
>>> exemplo2[3]
```

Listas

Uma lista é uma coleção ordenada

- append(novo_elemento)
- extend(segunda_lista)
- insert(posicao, novo_elemento)
- remove(valor_do_elemento)
- pop(posicao)
- index(valor_do_elemento)
- count(valor_do_elemento)
- sort(cmp=None, key=None, reverse=False)
- reverse()
- len(lista)

- https://pythonhelp.wordpress.com/2013/06/26/brincando-com-listas/
- https://docs.python.org/2/library/string.html

Slicing

- s[1]
- Índice negativo: s[-1]
- Fatias:
  - Sintaxe básica é s[inicio:final]: s[1:4]
  - Fatiando no inicio da string: s[:5]
  - Fatiando até o final da string: s[3:]
  - Índice negativo também pode ser usado em fatias: s[-3:-1]

Dicionários

O dicionário em si consiste em relacionar uma chave a um valor específico.

<variável> = {'<chave>': <valor>}

```
>>> dicionario = {'c': "Caroline"}
>>> dicionario['a'] = "Aline" # Adiciona um elemento ou altera
>>> dicionario2 = {1: "Beatriz", 2: "Pedro", 3: "Fernanda"}
```

Dicionários

```
>>> dicionario['aa']
```

```
KeyError: 'aa'
```

```
>>> dicionario.get('aa')
```

Dicionários

- copy()
- get()
- del()
- keys()
- clear()
- has_key()
- items()
- dict compreension
  - d = dict((key, value) for (key, value) in iterable)
  - d = {key: value for (key, value) in iterable}

Tuplas

São similares às listas, mas imutáveis. Não podemos adicionar ou modificar nenhum de seus elementos.

Exemplo:

```
>>> a = (3,5,8)
>>> aa =(['a'], 5)
>>> aa[0]
```

Tuplas

- Métodos
  - index()
  - count()
- *Sequence Unpacking*

Desvios Condicionais

Condicionais

Dois pontos e indentação.

- Sintaxe:

```
if   <condição dada por operador booleano>:
    <o que tenho que fazer, caso a condição seja satisfeita>
else:
    <o que tenho que fazer, caso a condição não seja satisfeita>
```

Condicionais - Exemplo

```
p = int(input('Primeiro Valor'))
s = int(input('Segundo Valor'))

if p > s :
    print ('O primeiro é maior  a = %d' % p)
else:
    print ('O segundo é maior  b = %d' % s)
```

Condicionais Aninhados

Sintaxe:

if <condição dada por operador booleano>:

<o que tenho que fazer, caso a condição seja satisfeita>

(caso a condição anterior não seja satisfeita, tenho mais uma condição para verificar)

elif <condição dada por operador booleano>:

<o que tenho que fazer se a segunda condição for satisfeita>

else:

<o que tenho que fazer, caso nenhuma das condições acima sejam satisfeitas>

Condicionais

```
n = int(raw_input())
if n % 2 != 0:
    print 'Weird'
elif n >= 2 and n <= 5:
    print 'Not Weird'
elif n >= 6 and n<=20:
    print 'Weird'
elif n > 20:
    print 'Not Weird'
```

Exercícios

Aula 2

Laços de Repetição

Laços de Repetição - For

Sintaxe:

```
for <variavel> in <variavel iteração>:
    <bloco de execução>
```

```
>>> for i in range(10): # Range é uma função retorna uma lista de números
...         print(i)
```

Laços de Repetição - While

Sintaxe:

while <condição dada por operador booleano>**:** # enquanto for verdadeiro

<bloco de execução> # faça

```
i = 0
while i < 10:
    print(i)
    i += 1
```

Laços de Repetição - break, continue, pass

- break - interrompe o loop ou while.
- continue - continua com a próxima iteração do loop
- pass - Passa para a próxima instrução

Exercícios

Funções

Funções

Funções são definidas usando a palavra chave def:

```
def <nome_da_função>(<parâmetros>):
    <bloco_de comandos>
    return <retorno>
```

```
>>>  def hello():
        print('Hello Word!!')
>>> hello()
```

Funções

```
b  = 8 # Variável global

def  imprime():
        b  = 9  # Variável local
        print (b)
        return

>>> imprime()
```

Funções

Assim a função que pode ser chamada com menos argumentos do que é definida para permitir.

```
def ask_ok(retries=4, complaint='Mensagem padrao!'):
    print(retries)
    print(complaint)
```

```
>>> ask_ok()
>>>  ask_ok(6, "Mudei a mensagem")
```

Random

Random

```
>>> from random import randint
>>> print(random.randint(0, 5))
```

- https://docs.python.org/3.6/library/random.html?highlight=random#module-random

Datetime

```
1  from datetime import date
2  now = date.today()
3  now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
4
5  birthday = date(1996, 9, 23)
6  age = now - birthday
7  print age.days
8
```

- dt = datetime.strptime("21/11/06 16:30", "%dd/%mm/%YYYY %H:%M")
- https://docs.python.org/3.6/library/datetime.html?highlight=datetime#module-datetime

Compreension – lists e dict

- Sintaxe

  [element-expression **for** element **in** sequence **]**

- Ex:

  [x**2 **for** x **in** **range**(10)]

- Sintaxe

  { key: value **for** (key, value) **in** sequence **}**

- Ex:

  {key: value **for** (key, value) **in** enumerate(range(10))}

Expressions – lambda()

Operador pode ter qualquer número de argumentos, mas apenas uma expressão.

- Sintaxe:

  lambda <parameters> **:** <expression>

  add = **lambda** x, y **:** x + y

  print add(2, 3) *# Output: 5*

https://docs.python.org/3.6/reference/expressions.html?highlight=lambda#lambda

Built in Functions

- Built in functions são implementadas em C dentro do interpretador do Python.
- Demais trechos de código devem ser interpretados.

- len() - Retorna o tamanho de um objeto

- Max() e Min()

- https://docs.python.org/3.6/library/functions.html

Built in Functions – map()

- Sintaxe:

map(function_object, iterable1, iterable2,...)

Espera um objeto de função e qualquer número de iteráveis como lista, dicionário, etc. Ele executa o **function_object** para cada elemento na seqüência e retorna uma lista dos elementos modificados pelo objeto de função.

Ex:

```
lista = list(map(lambda x: x*2, [1,2,3,4]))
```

Built in Functions - filter

- Sintaxe

  filter(function_object, iterable)

A função espera dois argumentos, function_object e iterable. function_object retorna um valor booleano.

Ex:

```
a = [1, 2, 3, 4, 5, 6]
filter(lambda x : x % 2 == 0, a) # Output: [2, 4, 6]
```

Tratamento de excessões

SyntaxError, TypeError, NameError

```
>>> print( 0 / 0 ))
  File "<stdin>", line 1
    print( 0 / 0 ))
                  ^
SyntaxError: invalid syntax
```

```
try:
   <bloco de código>
except:
    <bloco de código, caso tenha excessão>
```

Decorators

- São funções que são aplicadas em outras funções e retornam funções modificadas.
- Podem ser usados para criar ou alterar características das funções quanto para “envolver” as funções, acrescentando uma camada em torno delas com novas funcionalidades.

```
def decorator(f):
  f.decorated = True
  return f

@decorator
def func(arg):
  return arg
```

Exercícios

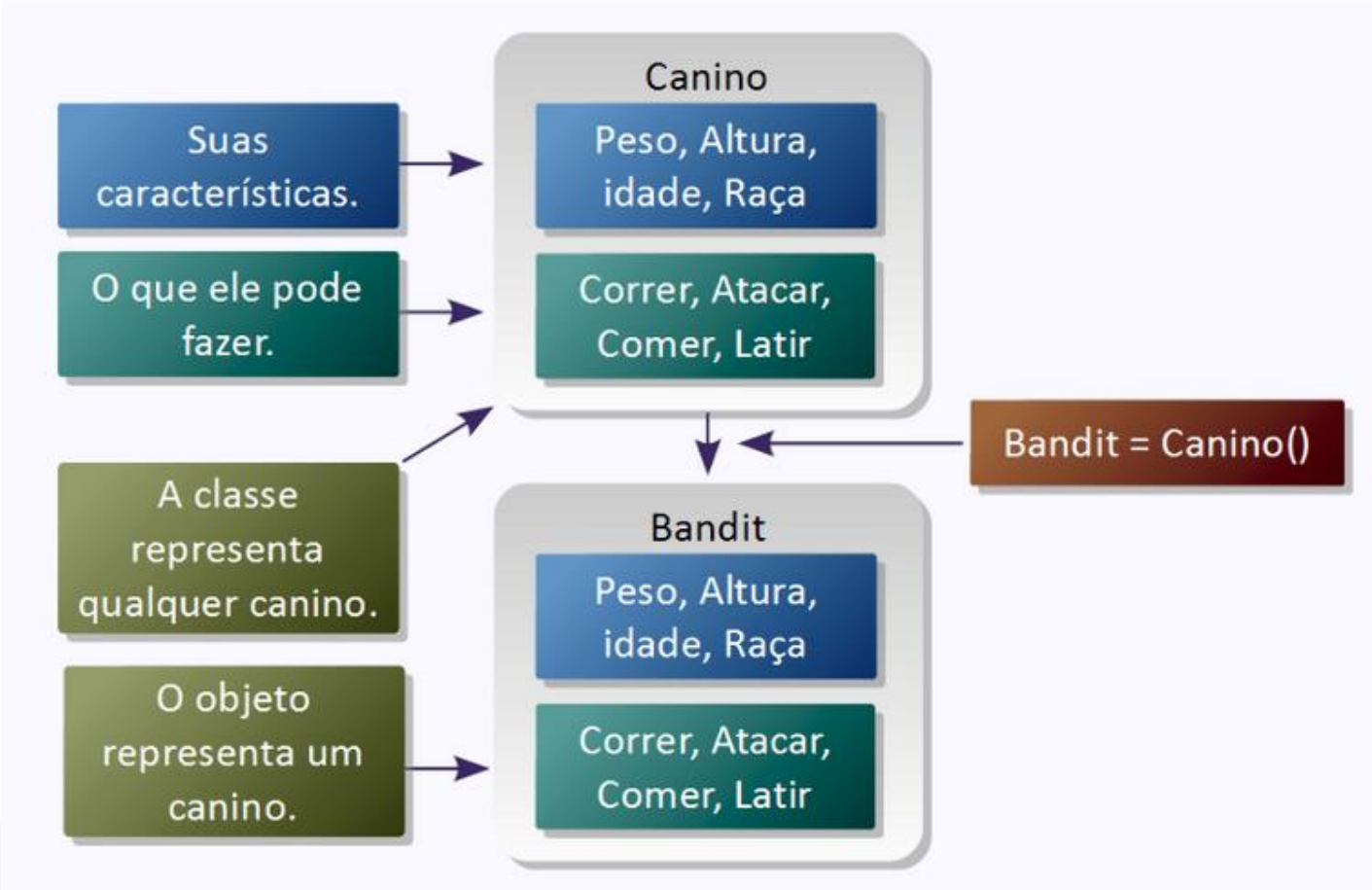
Aula 3

Orientação a Objetos

Classes

- Objetos são abstrações computacionais que representam entidades, com suas qualidades (atributos) e ações (métodos) que estas podem realizar. A classe é a estrutura básica do paradigma de orientação a objetos, que representa o tipo do objeto, um modelo a partir do qual os objetos serão criados.

Classes


Canino
Suas características.
Peso, Altura, idade, Raça
O que ele pode fazer.
Correr, Atacar, Comer, Latir
Bandit = Canino()
A classe representa qualquer canino.
Bandit
Peso, Altura, idade, Raça
O objeto representa um canino.
Correr, Atacar, Comer, Latir

Classes

- Construtor método **__init__()**
- Métodos especiais são identificados por nomes no padrão **__metodo__()** (dois sublinhados no início e no final do nome).
- A variável **self**, representa o objeto.
- O argumento **cls** representa a classe em si
- O método **__repr__()** é usado internamente pelo comando print para obter uma representação do objeto em forma de texto.

Classes - Exemplo

```
class Classe:
    def __init__(self):
        pass


class Cell:
    def __init__(self, formula='', format='%s'):
        self.formula = formula
        self.format = format
```

Classes

```
class Cell(object):
    """
    Classe para células de planilha
    """

    def __init__(self, formula='"', format='%s'):
        """
        Inicializa a célula
        """

        self.formula = formula
        self.format = format

    def __repr__(self):
        """
        Retorna a representação em string da célula
        """

        return self.format % eval(self.formula)


print Cell('123**2')
print Cell('23*2+2')
print Cell('abs(-1.45 / 0.3)', '%2.3f')
```

Classes

- Não existem variáveis e métodos privados (que só podem ser acessados a partir do próprio objeto).
- É usada uma convenção, usar um nome que comece com sublinhado (_), deve ser considerado parte da implementação interna do objeto e sujeito a mudanças sem aviso prévio. Além disso, a linguagem oferece uma funcionalidade chamada *Name Mangling*, que acrescenta na frente de nomes que iniciam com dois sublinhados (__), um sublinhado e o nome da classe.

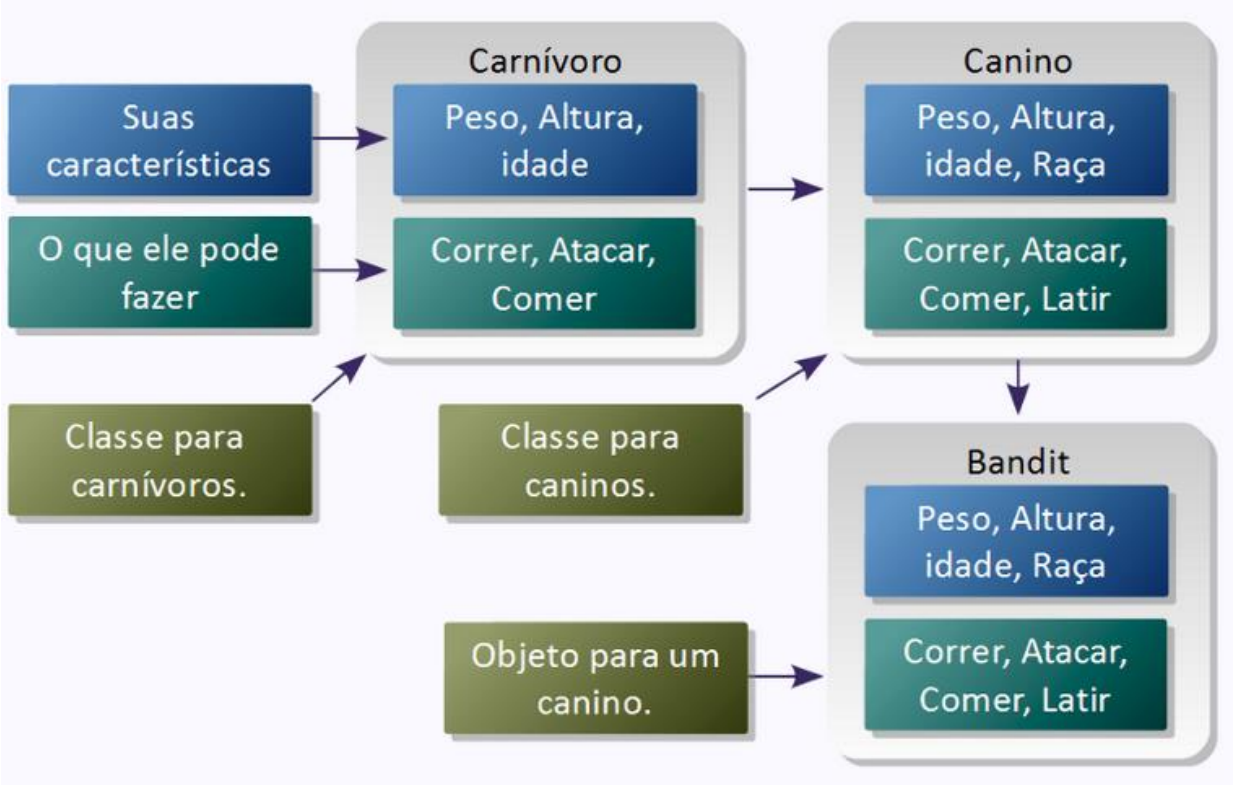
Classes abertas

- Em Python, as classes podem ser alteradas em tempo de execução, devido a natureza dinâmica da linguagem.

```
class User(object):
    """Uma classe bem simples.
    """
    def __init__(self, name):
        """Inicializa a classe, atribuindo um nome
        """
        self.name = name


# Um novo método para a classe
def set_password(self, password):
    """Troca a senha
    """
    self.password = password

print 'Classe original:', dir(User)

# O novo método é inserido na classe
User.set_password = set_password
```

Herança Simples

- Herança é um mecanismo que a orientação a objeto provê, com objetivo de facilitar o reaproveitamento de código.



Herança - Simples

```
class Pendrive(object):

    def __init__(self, tamanho, interface='2.0'):

        self.tamanho = tamanho
        self.interface = interface


class MP3Player(Pendrive):

    def __init__(self, tamanho, interface='2.0', turner=False):

        self.turner = turner
        Pendrive.__init__(self, tamanho, interface)
```

A classe *MP3Player* é derivada da classe *Pendrive*.

Herança múltipla

```
class Classe1:
    nome = ''
    def __init__(self):
            pass
class Classe2:
    def __init__(self):
            pass
class Classe3(Classe1, Classe2):
    def __init__(self):
            pass
```

https://nbviewer.jupyter.org/github/ricardoduarte/python-para-desenvolvedores/blob/master/Capitulo20/Capitulo20_Heranca_multipla.ipynb

Classes - Métodos

```
class Gerente(object):

    def __init__(self, nome, cpf, salario, senha):
        self.nome = nome
        self.cpf = cpf
        self.salario = salario
        self.senha = senha
        self.numeroDeFuncionariosGerenciados = 0

    def autentica(self, senha):
        if self.senha == senha:
            print "Acesso Permitido!"
            return True
        else:
            print "Acesso Negado!"
            return False
```

Propriedades

- As propriedades são criadas através da função / decorador *property*.
  - Validar a entrada do atributo.
  - Criar atributos apenas de leitura.
  - Simplificar o uso da classe.
  - Mudar de um atributo convencional para uma propriedade sem a necessidade de alterar as aplicações que utilizam a classe.

Propriedades

```python
# Exemplo de property de leitura


class Projetil(object):

    def __init__(self, alcance, tempo):

        self.alcance = alcance
        self.tempo = tempo

    @property
    def velocidade(self):

        return self.alcance / self.tempo


moab = Projetil(alcance=10000, tempo=60)

# A velocidade é calculada
print moab.velocidade
```

Abstract Method

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    def __init__(self, value):
        self.value = value
        super().__init__()

    @abstractmethod
    def do_something(self):
        pass
```

```
class DoAdd42(AbstractClassExample):
    pass
x = DoAdd42(4)
```

Referências

- https://ark4n.wordpress.com/python/

cake

www.cakeerp.com
pensegrande@cakeerp.com
(54) 3290-2100

Rua Angelo Michielin, 31
95041-050,
Caxias do Sul, RS,