Rumos


# Reading Python Programming

## OOP

13 de julho de 2024

# Agenda da Sessão

01. Programação Orientada a Objetos

02. Classe e Objeto | Atributos e Métodos

03. Acesso e Encapsulamento

# 01 Programação Orientada a Objetos

# Programação Orientada a Objetos

## Contextualização

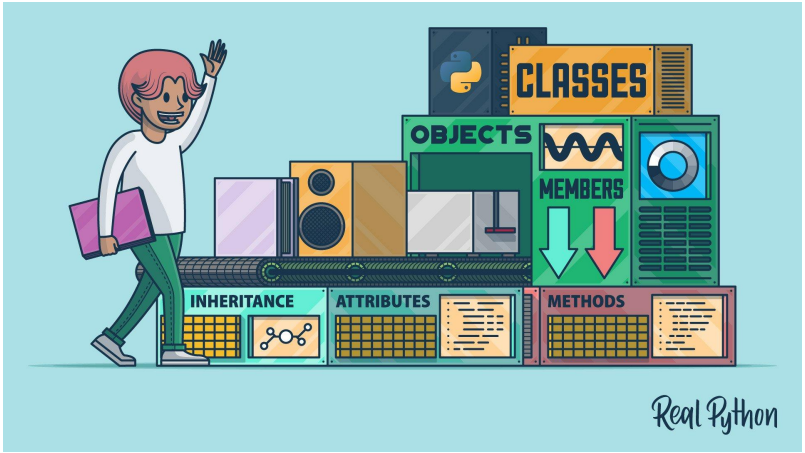
- Existem diversos paradigmas de programação, mas o mais relevante em linguagens de alto nível é a programação orientada a objetos - como em Python e Java
- Este paradigma permite criar programas de maneira mais organizada e eficiente, reutilizando código e modelando objetos do mundo real
- Em Python é possível utilizar outros paradigmas como a programação procedimental e a programação funcional

# Programação Orientada a Objetos

## Contextualização

Elementos essenciais em programação orientada a objetos:

- Classes "Blueprints" que definem a estrutura e o comportamento dos objetos
- Objetos Instâncias de uma classe e representam entidades do mundo real.
- Atributos Características dos objetos, como nome, idade, cor, etc.
- Métodos Ações que os objetos podem executar, como funções que pertencem a objetos específicos.



# 02

# Classe e Objeto | Atributos e Métodos

# Classe e Objeto

Classes são os moldes que definem a estrutura e o comportamento de objetos.

Sintaxe

- class Keyword dedicada
- def __init__() Método Construtor que é invocada sempre que se pretende criar uma instância/objeto da classe
- {Classe}({argumentos}) Forma de invocar o método construtor e instanciar um objeto da classe
- self Variável/apontador para o objeto instanciado dentro da classe

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age


person_maria = Person("Maria", 30)
```

# Atributos e Métodos

## Atributos

- Características que definem um objeto
- Armazenam informações sobre o objeto
- Atributos podem ser acessados usando a notação de ponto, como objeto.atributo.

## Métodos

- Ações que um objeto pode executar.
- Métodos podem ser acessados usando a notação de ponto, como objeto.método().
- self é um parâmetro especial que deve ser o primeiro em todos os métodos da classe

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def calc_area(self):
        return 3.14 * self.radius ** 2


circle_one = Circle(5)
print("Radius: " + circle_one.radius)
print("Area: " + circle_one.calc_area())
```

# Joalharia Alface

## Objetos com classe!

### Objetivo

Na classe Jewel acrescentar os atributos – kind, material, price – e completar os três métodos incompletos e instanciar um objeto deste género.

### Extra

Criar lista com 3 jóias diferentes e depois fazer um loop de "vendedor", em que apresenta cada uma das jóias.

# 03 Acesso e Encapsulamento

# Acesso e Encapsulamento

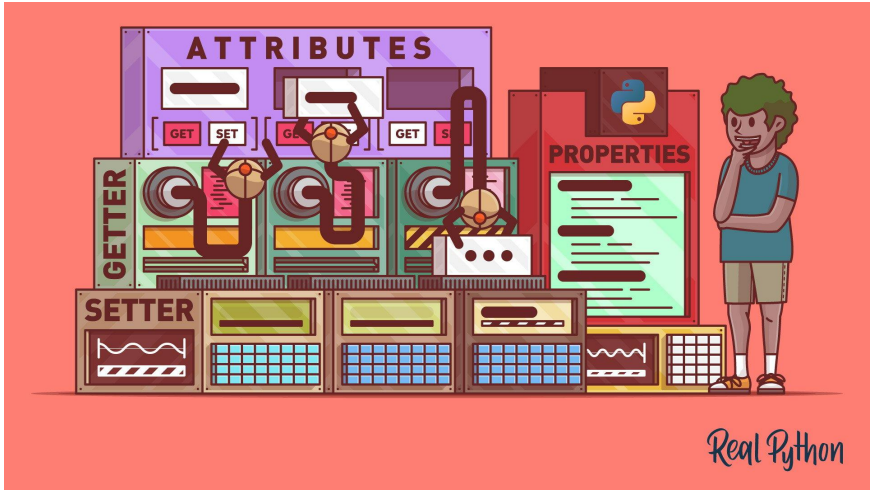
## Contexto

### Geral

- Visa proteger os atributos e métodos de uma classe
- Controlo como podem ser acessados e modificados.

### Python

- Encapsulamento é uma convenção e não é rigoroso/real
- Possível aumentar controlo através de decoradores



# Acesso e Encapsulamento

## Acesso Público

- Sem restrições de acesso
- self.{atributo/método}

## Acesso Privado

- Convenção que indica que não deve ser acedido
- self._{atributo/método} Declarado com um underscore

## Acesso Privado por Name Mangling

- Dificulta acesso “direto”
- self.__{atributo/método} Declarado com dois underscore
- {objecto}._{classe}{atributo/método} Aceder fora da classe

```
class Person:
    def __init__(self, name, age, city):
        self.name = name
        self._age = age
        self.__city = city


person_john = Person("John", 22, “Mira”)

# public access
print(person_john.name)

# private access
print(person_john._age)

# private access by name mangling
print(person_john._Person__city)
```

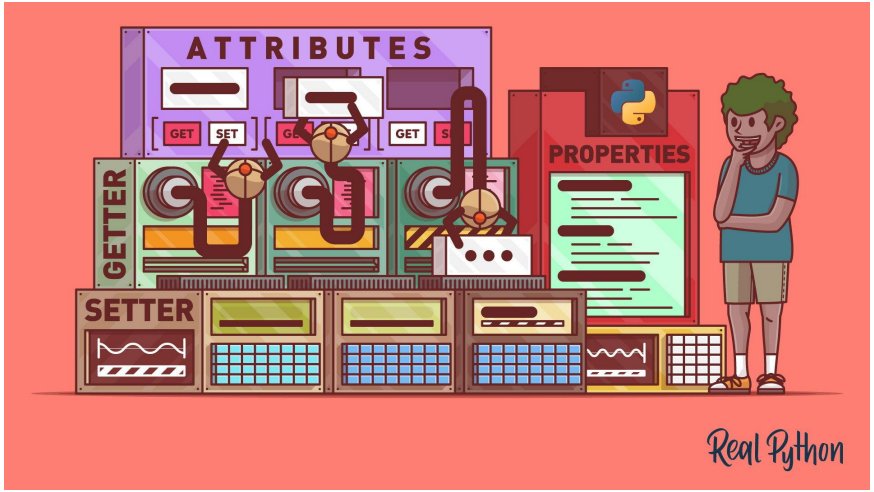
# Getters e Setters

## Geral

- Métodos acessórios para manipular atributos privados na própria classe
- Garantem maior segurança e escalabilidade do código

## Python

- Getters e Setters são feitos por convenção, dada não existir real restrição de acesso
- Podem ser identificados/criados por decoradores



# Getters e Setters

## Com decoradores

Decoradores são identificados por @ e colocados antes do método em si.

@property

- Getter
- Retorna valor de forma protegida através da invocação do próprio atributo

@{atributo}.setter

- Setter
- Manipula valor de forma protegida atribuindo pela via direta

```
class Person:
    def __init__(self, name):
        self.__name = name

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, new_name):
        self.__name = new_name


person = Person("Rui")
print(person.name)
person.name = "Ivo"
print(person.name)
```

# Getters e Setters

## Por convenção

get_{atributo} | set_{atributo}

- Torna explícito a privacidade dos atributos
- Podem ser métodos com “acesso restringido”

```
class Person:
    def __init__(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

    def set_name(self, new_name):
        self.__name = new_name


person = Person("Rui")
print(person.get_name())
person.set_name("Ivo")
```

# A zeros

## Acesso a Conta Bancária

### Objetivo

1 - Criar classe Bank Account com os atributos - account_owner, money_available, utilizando o tipo de acesso mais apropriado;

2 - Criar Getters e Setters para cada um dos atributos;

3 - Criar os métodos withdraw(value), deposit(value);

Rumos

Formação e Certificação Técnica que potenciam **Profissionais e Organizações**

## Lisboa

Edifício Mirage – Entrecampos
Rua Dr. Eduardo Neves, 3
1050-077 Lisboa

ver google maps ↗

**Tel** +351 217 824 100
**Email** info@training.rumos.pt

## Porto

Edifício Mirage – Entrecampos
Rua Dr. Eduardo Neves, 3
1050-077 Lisboa

ver google maps ↗

**Tel** +351 222 006 551
**Email** info@training.rumos.pt

Siga-nos

Rumos