# Introdução à linguagem Python

Thiago Martins

# Orientação a objetos

- Idéia central: Representar a computação como um conjunto de entidades (“agentes” ou “objetos”) que interagem entre si para realizar tarefas (objetos).
- Objetos podem representar entidades concretas ou abstratas.
- O funcionamento de objetos (o seu estado e como ele evolui) é *encapsulado.*
- Cada objeto expõe *interfaces*. As interfaces governam a interação entre objetos.
- A interface é de certa forma um “contrato”, um objeto que expõe uma dada interface está comprometido com um certo comportamento.

# Orientação a objetos

- 1950-1960: Foco em algoritmos e no fluxo de controle, voltados a resolução de problemas de cálculo e manipulação simples de dados (Fortran, Algol).
- 1970-1980: Foco no fluxo de dados e programaçãoestruturada (Cobol, Pascal, C).
- 1990-2000: Orientação a objetos: sistemas decompostos em estrutura formada por classes e objetos (C++, Java).
- Hoje?

# Orientação a objetos

- 1950-1960: Foco em algoritmos e no fluxo de controle, voltados a resolução de problemas de cálculo e manipulação simples de dados (Fortran, Algol).
- 1970-1980: Foco no fluxo de dados e programaçãoestruturada (Cobol, Pascal, C).
- 1990-2000: Orientação a objetos: sistemas decompostos em estrutura formada por classes e objetos (C++, Java).
- Hoje?

# Orientação a objetos em Python

- Python é uma linguagem orientada à objetos?
- SIM:
  - *Todos* os dados primitivos são objetos!
  - Permite a criação de novas *classes* de objetos (embora seja possível criar objetos sem classes)
- NÃO:
  - A sintaxe linguagem é na prática independente de conceitos de OO.
  - Python não dispõe de mecanismos de controle de encapsulamento.

# Orientação a objetos em Python

- Todos os dados podem ser tratados como objetos (e de fato o são?).

```
a = 1
dir(a)
```

- As interfaces são implementadas em Python por *métodos*.
- Um método se assemelha a uma *função* aplicada a um objeto.

```
a = [5,4,3,2,1]
a.sort()
```

- Métodos também podem receber parâmetros e retornar valores

```
a = [1,2,2,1,3]
a.count(1)
```

# Orientação a objetos em Python

- Criação de objetos via classe “vazia” (não usual!!).

```
class ObjetoVazio():
    pass
a = ObjetoVazio()
a.um_valor = 1
a.outro_valor = “dois”
def ola():
    print(“olá”)
a.cumprimento = ola
print(a.um_valor)
print(a.outro_valor)
a.cumprimento()
b = a
b.cumprimento()
```

**Mas não é assim que se faz usualmente!**

# Orientação a objetos em Python

- Classes em Python
  - No mundo de OO, uma Classe serve para declarar uma categoria de objetos, com comportamento uniforme.

```
class Valor():
    def __init__(self, x=0):
        self._x = x

    def incrementa(self):
        self._x += 1

    def __str__(self):
        return(str(self._x))
```

# Orientação a objetos em Python

- O método __init__ é invocado para criar uma *instância* ou seja, um objeto da classe declarada.

```
a = Valor(1)
print(a)
a.incrementa()
print(a)
print(a._x)
```

O mesmo que `Valor.incrementa(a)`

- Em geral, métodos e campos com nomes iniciados por _ não devem ser acessados por fora da classe. – Mais sobre isso adiante –
- Métodos iniciados e terminados por __ são *especiais* e têm um significado especial na linguagem. __str__ por exemplo é invocado para se obter a representação em string do objeto.

# Classes em Python

```
class Valor():
    def __init__(self, x=0):
        self._x = x

    def incrementa(self):
        self._x += 1

    def __str__(self):
        return(str(self._x))
```

Construtor

Atributo de objeto

Métodos

Corpo da classe

# Uma classe *também* é um objeto

```
class Classe():
    mensagem = “olá!”
    def muda_mensagem(nova):
        Classe.mensagem = nova
    def mostra_mensagem(self):
        print(Classe.mensagem)
```

```
a = Classe()
a.mostra_mensagem()
Classe.muda_mensagem(“adeus”)
a.mostra_mensagem()
```

# Orientação a objetos em Python

- Herança: É possível criar classes que *extendem* uma classe preexistente.

```
class ValorSomavel(Valor):
    def __init__(self, x=0):
        super().__init__(x)

    def __add__(self, outro):
        return ValorSomavel(self._x + outro._x)
```

# Orientação a objetos em Python

- Herança: Métodos podem ser *sobrepostos*

```
class Pessoa():
    def __init__(self, telefone):
        self._telefone = telefone
    def mostra_contato(self):
        print("Telefone: " + str(self._telefone))


  class Funcionario(Pessoa):
      def __init__(self, telefone, ramal):
          self._ramal = ramal
          super().__init__(telefone)
      def mostra_contato(self):
          print("Telefone: " + str(self._telefone))
          print("Ramal: " + str(self._ramal))
```

# Orientação a objetos em Python

- Herança: Métodos podem ser *sobrepostos.* Super também funciona aqui

```
class Pessoa():
    def __init__(self, telefone):
        self._telefone = telefone
    def mostra_dados(self):
        print("Telefone: " + str(self._telefone))


  class Funcionario(Pessoa):
      def __init__(self, telefone, ramal):
          self._ramal = ramal
          super().__init__(telefone)
      def mostra_contato(self):
          super().mostra_dados()
          print("Ramal: " + str(self._ramal))
```

# Orientação a objetos em Python

- Herança múltipla...


```
class Pessoa():
    def __init__(self, **args):
        self._nome = args['nome']

    def mostra_dados(self):
        print(f"Nome: {self._nome}")

class Funcionario(Pessoa):
    def __init__(self, **args):
        self._cargo = args['cargo']
        super().__init__(**args)

    def mostra_dados(self):
        super().mostra_dados()
        print(f'Cargo: {self._cargo}')
```

```
class Aluno(Pessoa):
    def __init__(self, **args):
        self._curso = args['curso']
        super().__init__(**args)

    def mostra_dados(self):
        super().mostra_dados()
        print(f'Curso: {self._curso}')

class Monitor(Aluno, Funcionario):
    def __init__(self, **args):
        super().__init__(**args)

    def mostra_dados(self):
         super().mostra_dados()
```

# Orientação a objetos em Python

- Herança múltipla...

  A ordem de chamada de métodos NÃO é trivial!


```
class O():
  def msg(self):
    print("O")

class F(O):
  def msg(self):
    print("F")
    super().msg()

class E(O):
  def msg(self):
    print("E")
    super().msg()

class D(O):
  def msg(self):
    print("D")
    super().msg()
```

```
class C(D, F):
  def msg(self):
    print("C")
    super().msg()

class B(D, E):
  def msg(self):
    print("B")
    super().msg()

class A(B, C):
  def msg(self):
    print("A")
    super().msg()
```