# Tecnólogo em Análise e Desenvolvimento de Sistemas

## Desenvolvimento Web I

UNIVERSIDADE CANDIDO MENDES

Prof. Ricardo Tavares

ricardo.tavares@ucam-campos.br

# Classes e objetos

- Python não requer a utilização de classes e objetos;
- A utilização de POO auxilia na organização e estrutura de programas mais complexos
  - Agrupa dados e comportamentos em um único local;
  - Promove a modularização do software, ou seja, partes individuais do programa não precisam saber como outras partes trabalham internamente
  - Se uma parte do software necessitar de modificações, atualizações ou até substituição é possível ser realizada com um mínimo de alterações em outras partes do software
  - Permite o isolamento de diferentes partes do software entre elas.

# Classes e objetos

## Classes e objetos - exemplo - Conta bancária

- Em nosso primeiro exemplo, teremos como informação de uma conta apenas *saldo* e *numero da conta*

```
1 class Conta:
2     def __init__(self, numero, saldo=0):
3         self.saldo =saldo
4         self.numero = numero
```

# Classes e objetos

## Classes e objetos - exemplo - Conta bancária

```
1 class Conta:
2     def __init__(self, numero, saldo=0):
3         self.saldo =saldo
4         self.numero = numero
```

- Usamos a palavra **class** para indicar a definição de uma classe.
- A classe **Conta** tem dois **atributos** (*saldo* e *numero*)
- _init_ é um método especial, denominado **construtor** por ser chamado sempre que um objeto da classe é criado (instanciado).
- o método _init_ recebe um parâmetro chamado *self*, que indica o objeto que está sendo criado.

# Classes e objetos

## Classes e objetos - exemplo - Conta bancária

```
1 class Conta:
2     def __init__(self, numero, saldo=0):
3         self.saldo =saldo
4         self.numero = numero
```

- *self.saldo* indica o atributo *saldo* da conta que está sendo criada. Se não colocássemos *self.* na frente, saldo seria uma variável local e teria seu valor jogado fora quando o método acabasse de ser executado.

- no interpretador, podemos criar um objeto **conta** e atribuí-lo a uma variável. Para isso, usamos o nome da classe e os parâmetros indicados no método construtor __init__.

```
1 In [1]:  c1=Conta(1234, 100)
```

# Classes e objetos

## Classes e objetos - exemplo - Conta bancária

- Adicionaremos a nosso exemplo o comportamento da conta ao reagir às operações de *saque* e *deposito*, além da operação *resumo* para ver o saldo.

```
1  class Conta:
2      def __init__(self, numero, saldo=0):
3          self.saldo =saldo
4          self.numero = numero
5
6      def resumo(self):
7          print("CC numero: %s Saldo: %10.2f" % (self.numero, self.saldo))
8
9      def saque(self, valor):
10         if self.saldo>=valor:
11             self.saldo -= valor
12
13     def deposito(self, valor):
14         self.saldo += valor
```

- A classe **Conta** tem agora dois **atributos** (*saldo* e *numero*) e quatro **métodos** (__init__, *resumo*, *saque* e *deposito*)

# Classes e objetos

## Classes e objetos - exemplo - Conta bancária

- o primeiro parâmetro de todos os métodos de uma classe em Python tem que ser o *self*. Ele representa a instância sobre a qual o método atuará.

- os atribudos de um objeto tem seu valor preservado durante todo o tempo de vida do objeto. O tempo de vida de um objeto é o tempo em que alguma variável do seu programa o referencia.

```
1  class Conta:
2      def __init__(self, numero, saldo=0):
3          self.saldo =saldo
4          self.numero = numero
5
6      def resumo(self):
7          print("CC numero: %s Saldo: %10.2f" % (self.numero, self.saldo))
8
9      def saque(self, valor):
10         if self.saldo>=valor:
11             self.saldo -= valor
12
13     def deposito(self, valor):
14         self.saldo += valor
```

# Classes e objetos

## Classes e objetos - exemplo - Conta bancária

- Apesar de imprescindível na **definição** de cada método, não é necessário passar o *self* como parâmetro na hora de **chamar** um método, isso é feito automaticamente no interpretador Python.

```
In [1]:  c1=Conta(1234, 100)

In [2]: c1.resumo
Out[2]: <bound method Conta.resumo of <__main__.Conta object at 0x00000294A612DA58
     >>

In [3]: c1.resumo()
Out[3]: CC numero: 1234 Saldo:      100.00

In [4]: c1.saque(50)

In [5]: c1.resumo()
Out[5]: CC numero: 1234 Saldo:       50.00

In [6]: c1.deposito(200)

In [7]: c1.resumo()
Out[7]:CC numero: 1234 Saldo:      250.00
```

# Classes e objetos

- Encapsulamento:
  - O underscore _ alerta que aquele é um atributo privado e ninguém deve modificar, nem mesmo ler, o atributo em questão fora do domínio da classe;
  - Crie métodos getters e setters para realizar a interface com a classe;

```
class Conta:
    def __init__(self, numero, saldo=0):
        self.saldo =saldo
        self.numero = numero
```

```
class Conta:
    def __init__(self, numero, saldo=0):
        self._saldo = saldo
        self._numero = numero
    def get_saldo(self):
        return self._saldo
    def set_saldo(self, saldo):
        if(saldo < 0):
            print("saldo não pode ser negativo")
        else:
            self._saldo = saldo
```

# Classes e objetos

- Encapsulamen
  - O underscore mesmo ler, o atributo em
  - Crie métodos om a classe;


```
class Conta:
    def __init__(sel
        self.saldo =
        self.numero
```

```
, numero, saldo=0):
 saldo
= numero
f):
saldo
f, saldo):
:
ldo não pode ser negativo")

do = saldo
```

# Classes e objetos

## Classes e objetos - aplicação

- Usamos classes e objetos para facilitar a construção de programas mais complexos.
- Repare que nos métodos de *saque* e *deposito* não foi necessário passar o saldo como parâmetro, apenas o valor a ser sacado ou depositado.
- Esse efeito de memória ou permanência dos atributos é proposital para evitar passagens de muitos parâmetros ao lidarmos com informações complexas em nossos programas.
- A ideia é imitar o comportamento de objetos do mundo real: ao fazer um depósito ou saque em uma conta, você não precisa informar ao banco qual o saldo dessa conta.

# Classes e objetos

- Classes e Objetos:
  - Na prática!

# Classes e objetos

## Herança

- A orientação a objetos permite utilizar classes já definidas como base para a construção de outras classes.
- Dizemos que a nova classe *herda* da classe antiga propriedades e / ou comportamentos.
- Esse é um recurso para reutilizar código já desenvolvido.

## Herança - Exemplo

Em nosso sistema de controle de contas bancárias, queremos agora oferecer a modalidade de conta especial, que permite aos correntistas sacar mais dinheiro do que a quantia disponível na conta, até um certo limite.

# Classes e objetos

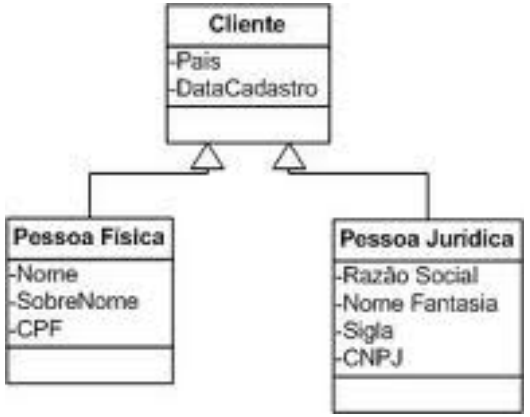
## Herança - Exemplo

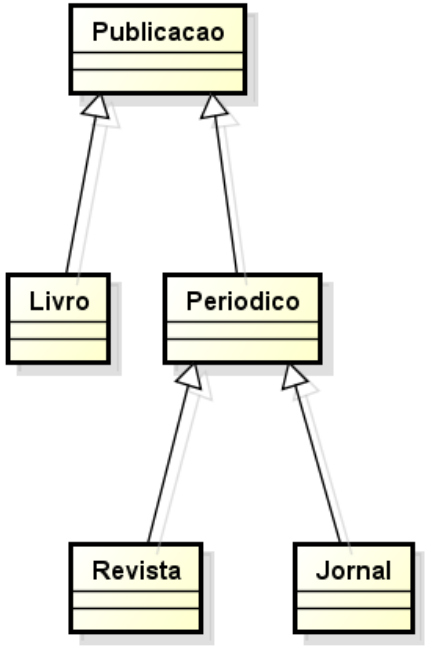
- Na definição da classe, ao lado do nome da classe colocamos dentro dos parântese o nome da classe de qual ela **herda** (caso haja).
- A classe *ContaEspecial* **herda** todos os atributos e métodos da classe *Conta*.
- Dizemos então que *ContaEspecial* é uma **subclasse** da classe *Conta*, e que *Conta* é uma **superclasse** de *ContaEspecial*.

```
1 class ContaEspecial(Conta):
2     def __init__(self, correntistas, numero, saldo=0, limite=0):
3         Conta.__init__(self, correntistas, numero, saldo)
4         self.limite = limite
5
6     def saque(self, valor):
7         if self.saldo + self.limite >= valor:
8             self.saldo -= valor
9             self.operacoes+= [("SAQUE", valor)]
```

# Classes e objetos

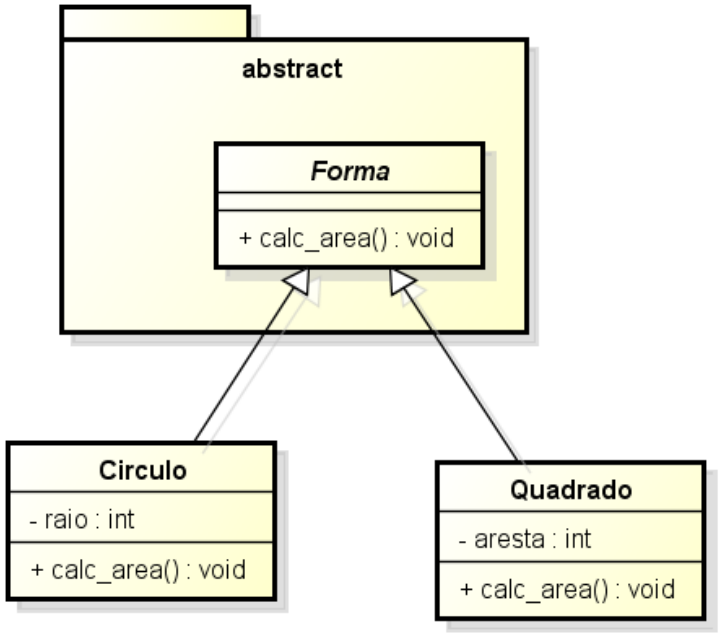
- Herança:
  - Na prática!



# Classes e objetos

- Modularização:
  - Na prática!



# Classes e objetos

- Exercício da Biblioteca

UNIVERSIDADE
CANDIDO
MENDES

EAD