



## DEFINIÇÃO

Introdução aos conceitos da orientação a objetos; Classes e Encapsulamento; Herança e Polimorfismo; Construtores; Atributos e Métodos; Implementação de Herança; Implementação de Polimorfismo; Classes Abstratas; Tratamento de Exceções; Python e linguagens OO.

## PROPÓSITO

Compreender o desenvolvimento de software orientado a objetos, utilizando uma linguagem de programação como Python com grande aceitação no meio comercial e acadêmico. Entender os conceitos e pilares da orientação a objetos e saber contextualizar o Python entre as outras linguagens tradicionais orientadas a objetos como Java e C++.

# PREPARAÇÃO

Para este módulo, é necessário conhecimentos de programação em linguagem Python, incluindo modularização e utilização de bibliotecas em Python. Antes de iniciar o conteúdo deste tema é necessário que tenha o interpretador Python na versão 3.7.7 e o ambiente de desenvolvimento PyCharm ou outro ambiente que suporte o desenvolvimento na linguagem Python.

## OBJETIVOS

### MÓDULO 1

Definir os conceitos gerais da orientação a objetos

### MÓDULO 2

Descrever os conceitos básicos da programação orientada a objetos na linguagem Python

## MÓDULO 3

Descrever os conceitos da orientação a objetos como Herança e Polimorfismo

## MÓDULO 4

Comparar a implementação dos conceitos orientados a objetos aplicados a Python com outras linguagens orientadas a objetos existentes no mercado

# INTRODUÇÃO

O paradigma de programação orientado a objetos é largamente utilizado para o desenvolvimento de software devido à sua implementação ser próxima dos conceitos do mundo real. Essa proximidade facilita a manutenção dos softwares orientados a objetos. A linguagem Python implementa os conceitos do paradigma orientado a objetos e, devido à sua sintaxe simples e robusta, torna-se uma ferramenta poderosa para a implementação de sistemas orientados a objetos.

Na apostila, serão apresentados os conceitos da orientação a objetos, como implementá-los em Python e uma comparação com as linguagens Java e C++ - em relação às principais características da orientação a objetos.

# MÓDULO 1

---

- ⦿ Definir os conceitos gerais da orientação a objetos

## CONCEITOS DE POO - PROGRAMAÇÃO ORIENTADA A OBJETOS

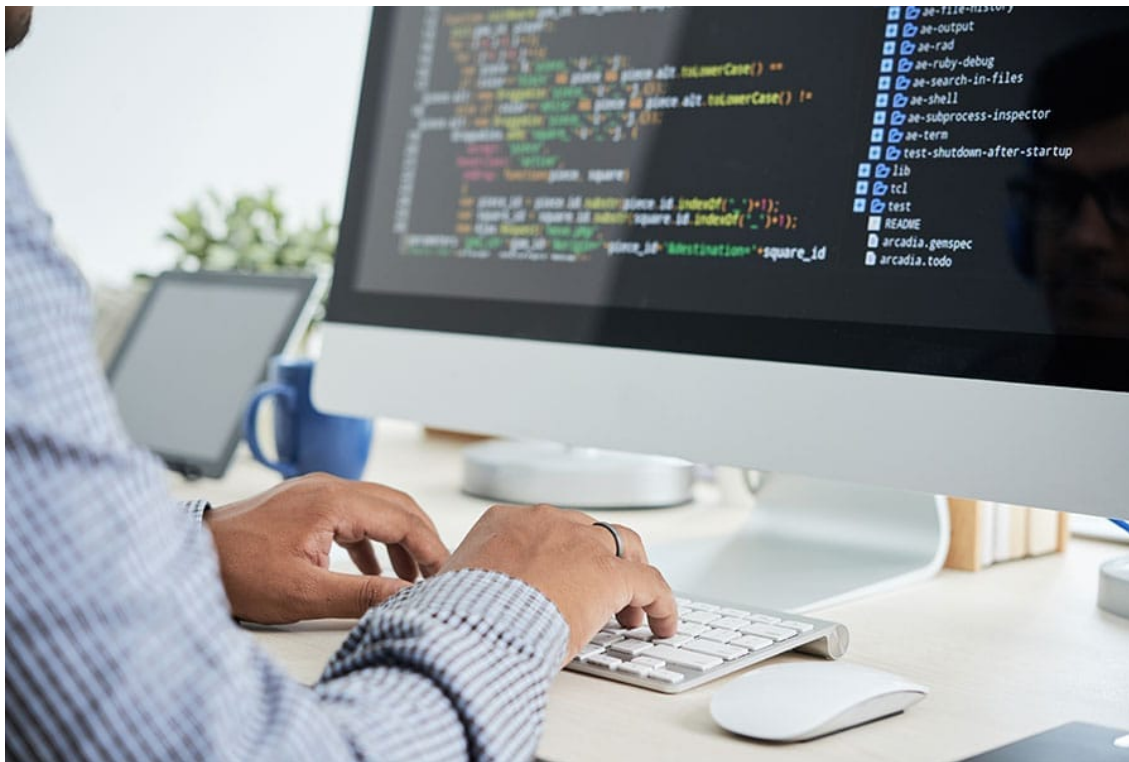
A programação orientada a objetos foi considerada uma revolução na programação, pois mudou completamente a estruturação dos programas de computador. Essa mudança se estendeu, inclusive, para os modelos de análise do mundo real e depois para a implementação dos respectivos modelos nas linguagens de programação orientadas a objetos.

Conforme apresentado por Rumbaugh *et al* (1994):

**“A TECNOLOGIA BASEADA EM OBJETOS É MAIS DO QUE APENAS UMA FORMA DE PROGRAMAR. ELA É MAIS IMPORTANTE COMO UM MODO DE PENSAR EM UM PROBLEMA DE FORMA ABSTRATA, UTILIZANDO CONCEITOS DO MUNDO REAL E NÃO IDEIAS COMPUTACIONAIS”.**

Jaeger, 1995.

Muitas vezes, analisamos um problema do mundo real pensando no projeto, que, por sua vez, é influenciado por ideias sobre codificação, que são fortemente influenciadas pelas linguagens de programação disponíveis. A abordagem baseada em objetos permite que os mesmos conceitos e a mesma notação sejam usados durante todo o processo de desenvolvimento de software, ou seja, não existem conceitos de análise de projetos diferentes dos conceitos de implementação dos projetos. A abordagem procura refletir os problemas do mundo real através de interação de objetos modelados computacionalmente. Portanto, o desenvolvedor do software não necessita realizar traduções para outra notação em cada etapa do desenvolvimento de um projeto de software (COSTA, 2015).



Fonte: Freepik

O software é organizado como uma coleção de objetos separados que incorporam, tanto a estrutura, quanto o comportamento dos dados. Contrasta com a programação convencional, segundo a qual a estrutura e o comportamento dos dados têm pouca vinculação entre si. Os

modelos baseados em objetos correspondem mais aproximadamente ao mundo real. Em consequência, são mais adaptáveis às modificações e evoluções dos sistemas.

# PILARES DA ORIENTAÇÃO A OBJETOS

## OBJETOS

Um objeto é a representação computacional de um elemento ou processo do mundo real. Cada objeto possui suas características (informações) e uma série de operações (comportamento) que alteram as suas características (estado do objeto). Todo o processamento das linguagens de programação orientadas a objetos se baseia no armazenamento e na manipulação das informações (estados). São exemplos de objetos do mundo real e computacional: Aluno, Professor, Livro, Empréstimo e Locação. (Costa, 2015)

É importante, durante a etapa de levantamento dos objetos, analisar apenas os objetos relevantes (abstrair) com as respectivas características mais importantes para o problema a ser resolvido. Por exemplo, as características de uma pessoa para um sistema acadêmico podem ser formação, nome do pai e da mãe, enquanto as características de um indivíduo para o sistema de controle de uma academia são altura e peso.

## ATRIBUTOS

São propriedades do mundo real que descrevem um objeto. Cada objeto possui suas respectivas propriedades do mundo real, os quais possuem valores. A orientação a objetos define as propriedades como atributos. O conjunto de valores dos atributos de um objeto definem o seu estado naquele momento (RUMBAUGH, 1994).

Veja nas tabelas a seguir a diferença entre os atributos de duas mulheres:



Fonte: drobotdean / Freepik

Atributos	Valores armazenados pelos Atributos
Nome	Maria

Idade	35
Peso	63kg
Altura	1,70m

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal



Fonte: katemangostar / Freepik



Atributos	Valores armazenados pelos Atributos
Nome	Joana
Idade	30
Peso	60kg
Altura	1,65m

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

# OPERAÇÕES

Uma operação é uma função ou transformação que pode ser aplicada a objetos ou dados pertencentes a um objeto. Importante dizer que todo objeto possui um conjunto de operações, que podem ser chamadas por outros objetos de modo a colaborarem entre si. **Esse conjunto de operações é conhecido como interface**. A única forma de colaboração entre os objetos é por meio das suas respectivas interfaces (FARINELLI, 2020).

Utilizando o exemplo acima, podemos alterar nome, idade e peso da pessoa por meio de um conjunto de operações. Portanto, normalmente, essas operações alteram o estado do objeto. Vamos ver a seguir outros exemplos de operações:

## ★ EXEMPLO

Classe empresa = Contratar\_Funcionario, Despedir\_Fucionario;

Classe janela = Abrir, fechar, ocultar.

O desenvolvimento de um sistema orientado a objetos consiste em realizar um mapeamento, conforme apresentado na imagem a seguir.

Fonte: Autor

📷 Figura 1 - Mapeamento de objetos do mundo real.

Basicamente, deve-se analisar o mundo real e identificar quais objetos devem fazer parte da solução do problema. Para cada objeto identificado, levantam-se os **atributos** que descrevem as propriedades dos objetos e as **operações** que podem ser executadas sobre esses objetos.



# CLASSES

A classe descreve as características e os comportamento de um conjunto de objetos. De acordo com a estratégia de classificação, cada objeto pertence a uma única classe e possuirá os atributos e as operações definidos na classe. Durante a execução de um programa orientado a objetos, são instanciados os objetos a partir da classe, portanto, **um objeto é chamado de instância de sua classe**.

A classe é o bloco básico para a construção de programas OO – Orientados a Objetos (COSTA, 2015).

## ATENÇÃO

Importante ressaltar que cada nome de atributo é único dentro de uma classe, no entanto, essa premissa não é verdadeira quando se consideram todas as classes. Por exemplo, as classes Pessoa e Empresa podem ter um atributo comum chamado de Endereço.

Com base na Tabela 1, vista anteriormente, deve ser definida uma classe Pessoa com os atributos Nome, Idade, Peso e Altura. A partir da classe Pessoa, pode-se instanciar uma quantidade ilimitada de objetos contendo os mesmos atributos. Os objetos de uma classe sempre compartilham o conjunto de operações que atuam sobre seus dados, alterando o estado do objeto.

Um programa orientado a objetos consiste basicamente em um conjunto de objetos que colaboram entre si, por meio de uma troca de mensagens, para a solução de um problema computacional. Cada troca de mensagens significa a chamada de uma operação feita pelo objeto receptor da mensagem. (COSTA, 2015)

## COMUNICAÇÃO ENTRE OBJETOS DIFERENTES

Fonte: Autor

 Figura 2 – Colaboração motorista e carro.

De acordo com a Figura 2, vista anteriormente, um objeto motorista 1, instanciado a partir da classe Motorista, envia a mensagem “Freia” para um objeto carro 1, instanciado a partir da classe Carro. O objeto carro 1, ao receber a mensagem, executa uma operação para acionar os freios do automóvel. Essa operação também diminuirá o valor do atributo velocidade do objeto carro 1.

## ENCAPSULAMENTO

O conceito de encapsulamento consiste na separação dos aspectos externos (operações) de um objeto acessíveis a outros objetos, além de seus detalhes internos de implementação, que ficam ocultos dos demais objetos (RUMBAUGH, 1994). Algumas vezes, é conhecido como o

princípio do ocultamento de informação, pois permite que uma classe encapsule atributos e comportamentos, ocultando os detalhes da implementação. Partindo desse princípio, a interface de comunicação de um objeto deve ser definida de modo a revelar o menos possível sobre o seu funcionamento interno.

## ★ EXEMPLO

Foi desenvolvido um objeto `ApresentaçãoMapa`, pertencente a um aplicativo de entrega móvel, que possui a responsabilidade de apresentar um mapa com o menor caminho entre dois pontos. Porém, o objeto não “sabe” como calcular a distância entre os dois pontos.

Para resolver esse problema, ele precisa colaborar com um objeto `Mapa` que “sabe” calcular e, portanto, possui essa responsabilidade. O objeto `Mapa` implementa essa responsabilidade por meio da operação `Calcula Melhor Caminho`, cujo resultado é a menor rota entre duas coordenadas geográficas.

Utilizando o encapsulamento, o objeto `Mapa` calcula e retorna o melhor caminho para o objeto `ApresentaçãoMapa` de maneira transparente, escondendo a complexidade da execução dessa tarefa.

Fonte: Autor

📷 Figura 3 – Encapsulamento da Classe `Mapa`.

Uma característica importante do encapsulamento é que pode surgir um modo diferente de se calcular o melhor caminho entre dois pontos. Por conta disso, o objeto `Mapa` deverá mudar o seu comportamento interno para implementar esse novo cálculo. Porém, essa mudança não afetará o objeto `ApresentaçãoMapa`, pois a implementação foi realizada isoladamente (encapsulamento) no objeto `Mapa`, sem impacto para outros objetos no sistema.

## RESUMINDO

Os objetos clientes têm conhecimento apenas das operações que podem ser requisitadas e precisam estar cientes apenas do que as operações realizam, e não de como elas estão implementadas.

## HERANÇA

Na orientação a objetos, a herança é um mecanismo por meio do qual classes compartilham atributos e comportamentos, formando uma hierarquia. Uma classe herdeira recebe as características de outra classe para reimplementá-las ou especializar de uma maneira diferente da classe pai. A herança permite capturar similaridades entre classes, dispondo-as em hierarquias. As similaridades incluem atributos e operações sobre as classes (FARINELLI, 2020).

Essa estrutura reflete um mapeamento entre classes, e não entre objetos, conforme esquema a seguir:

Fonte: Autor

 Figura 4 – Exemplo de herança.

No esquema anterior, as classes Carro, Moto, Caminhão e Ônibus herdam características em comum da classe Veículo, como os atributos chassi, ano, cor e modelo.

Uma classe pode ser definida genericamente como uma superclasse e, depois, especializada em classes mais específicas (subclasses). A herança permite a reutilização de código em larga escala, pois possibilita que se herde todo o código já implementado na classe pai e se adicione apenas o código específico para as funcionalidades novas implementadas pela classe filha.

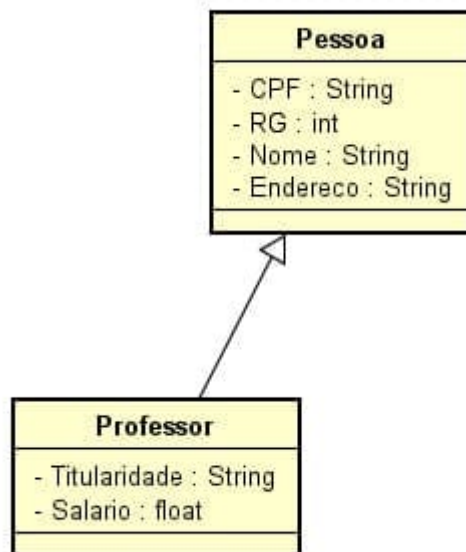
A evolução dos sistemas orientados a objetos também é facilitada, pois, caso surja uma classe nova com atributos e/ou operações comuns a outra, basta inseri-la na hierarquia, acelerando a implementação.

Veja a seguir os tipos de herança:

## HERANÇA SIMPLES

A herança é considerada simples quando uma classe herda as características existentes apenas de uma superclasse.

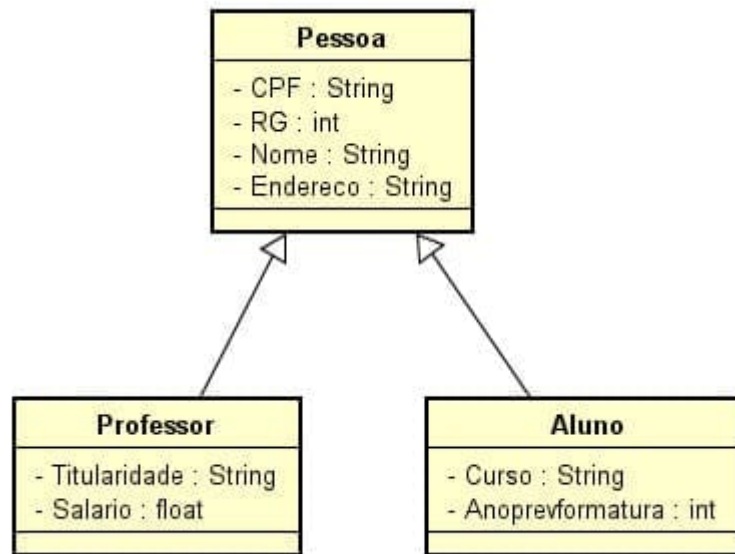
Na figura a seguir, temos uma superclasse Pessoa que possui os atributos CPF, RG, Nome e Endereço. Em seguida, a classe Professor precisa herdar os atributos da superclasse Pessoa, além de adicionar atributos específicos do contexto da classe Professor, como titularidade e salário.



Fonte: Autor

📷 Figura 5 – Exemplo de herança Pessoa -> Professor.

Considerando um sistema acadêmico, a classe Aluno também se encaixaria na hierarquia acima, tornando-se uma subclasse de Pessoa. Porém, precisaria de outros atributos associados a seu contexto, como curso e Anoprevformatura.



Fonte: Autor

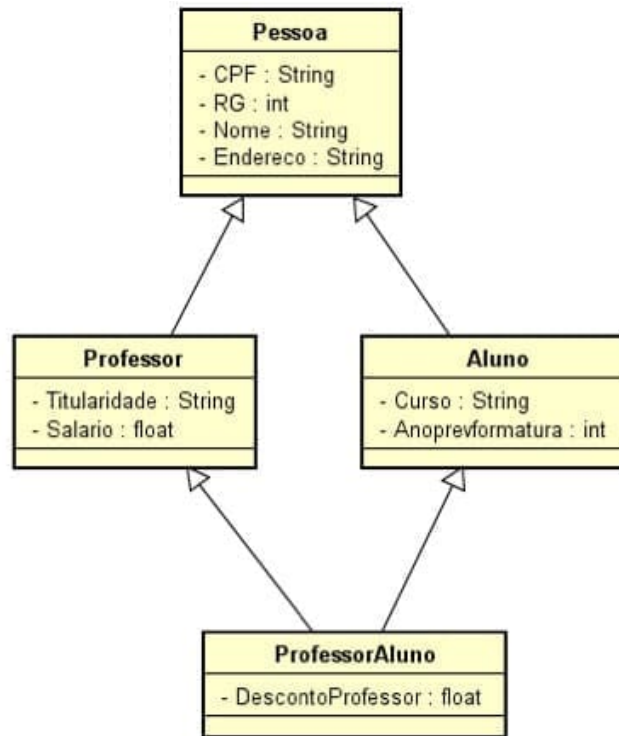
📷 Figura 6 – Exemplo de herança Pessoa -> Professor e Pessoa -> Aluno.

## HERANÇA MÚLTIPLA

A herança é considerada múltipla quando uma classe herda características de duas ou mais superclasses. Por exemplo, no caso do sistema acadêmico, o docente também pode desejar realizar um outro curso de graduação na mesma instituição em que trabalha. Portanto, ele possuirá os atributos da classe Professor e os da classe Aluno. Além disso, haverá também um atributo DescontoProfessor, o qual apenas quando houver a associação professor e aluno com a universidade.

Para adaptar essa situação no mundo real, deve ser criada uma modelagem de classes. Uma nova subclasse ProfessorAluno precisa ser adicionada, herdando atributos e operações das classes Professor e Aluno. Isso configura uma herança múltipla. Essa nova subclasse deverá ter o atributo DescontoProfessor, que faz sentido apenas para essa classe.





Fonte: Autor

📷 Figura 7 – Exemplo de herança Pessoa -> Professor e Pessoa -> Aluno e Professor/Aluno ->ProfessorAluno.

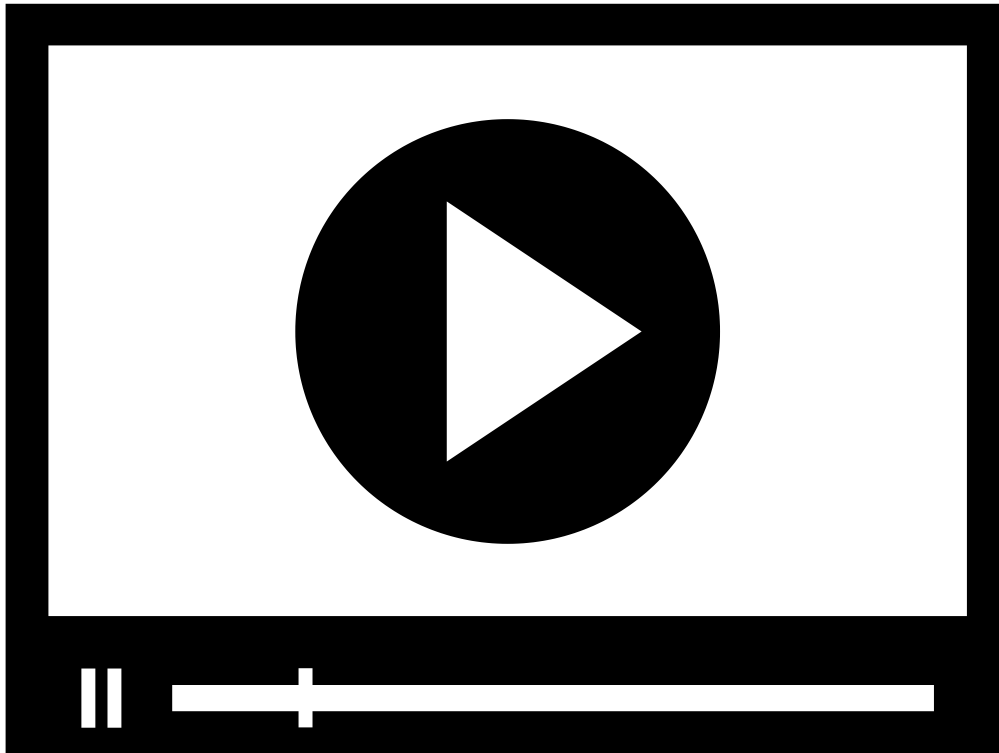
## POLIMORFISMO

O Polimorfismo é a capacidade de um mesmo comportamento diferente em classes diferentes. Uma mesma mensagem será executada de maneira diversa, dependendo do objeto receptor. O polimorfismo acontece quando reimplementamos um método nas subclasses de uma herança (FARINELLI, 2020).

Fonte: Autor

📷 Figura 8 – mover() – método polimórfico.

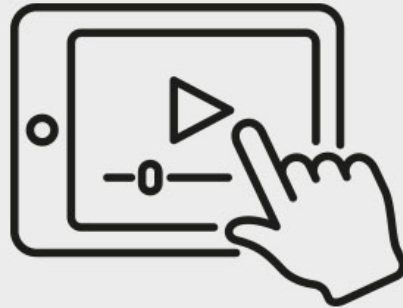
Como exemplificado na figura 1.8, o comportamento mover() em um objeto instanciado pela classe Aereo será diferente do mover() em um objeto da classe Terrestre. Um objeto poderá enviar uma mensagem para se mover e o objeto receptor decidirá como isso será feito.



## CONCEITOS GERAIS DE ORIENTAÇÃO OBJETO

Agora o Prof Marcelo Costa irá apresentar alguns exemplos dos conceitos de orientação objeto:

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



## VERIFICANDO O APRENDIZADO

- 1. NA PROGRAMAÇÃO ORIENTADA A OBJETOS, TEMOS CONCEITOS COMO HERANÇA E POLIMORFISMO. SOBRE ESSES CONCEITOS ANALISE AS ASSERTIVAS E ASSINALE A ALTERNATIVA QUE APONTA A(S) CORRETA(S).**
  - I. PARA EVITAR CÓDIGO REDUNDANTE, O PARADIGMA DE ORIENTAÇÃO A OBJETOS OFERECE UMA ESTRUTURA HIERÁRQUICA E MODULAR PARA REUTILIZAÇÃO DE CÓDIGO ATRAVÉS DE UMA TÉCNICA CONHECIDA COMO HERANÇA.**
  - II. HERANÇA PERMITE PROJETAR CLASSES GENÉRICAS QUE PODEM SER ESPECIALIZADAS EM CLASSES MAIS PARTICULARES, ONDE AS CLASSES ESPECIALIZADAS REUTILIZAM O CÓDIGO DAS MAIS GENÉRICAS.**

**III. LITERALMENTE, “POLIMORFISMO” SIGNIFICA “MUITAS FORMAS”. NO CONTEXTO E PROJETO ORIENTADO A OBJETOS, ENTRETANTO, REFERE-SE À HABILIDADE DE UMA VARIÁVEL DE OBJETO DE ASSUMIR FORMAS DIFERENTES.**

**IV. POLIMORFISMO PERMITE QUE OS ATRIBUTOS DE UMA CLASSE NÃO TENHAM ACESSO DIRETAMENTE.**

**A) Apenas I.**

**B) Apenas I e III.**

**C) Apenas I, II e III.**

**D) Apenas II, III e IV.**

**2. OS BANCOS SÃO INSTITUIÇÕES QUE INVESTEM FORTEMENTE NA ÁREA DE TECNOLOGIA DA INFORMAÇÃO, INCLUSIVE COM A CONTRATAÇÃO DE MILHARES DE PROFISSIONAIS E A CONSTRUÇÃO DE GRANDES AMBIENTES DE DATACENTER. POR ISSO, É UM DOMÍNIO DE CONHECIMENTO BASTANTE IMPORTANTE E QUE DEVE SER UTILIZADO COMO EXEMPLO DURANTE UM CURSO DE GRADUAÇÃO. SABENDO DISSO, ANALISE A SEGUINTE SITUAÇÃO EM UM SISTEMA BANCÁRIO: A CONTABANCARIA(CB) ESPECIALIZA AS CLASSES ITEMSUPORTADO (IS) E ITEMSUJEITOAJUROS (ISJ) E GENERALIZA AS CLASSES CONTACORRENTE (CC) E POUPANÇA (PP). NESSE SENTIDO, É CORRETO AFIRMAR QUE OCORRE (0,5)**

**A) Relação de dependência entre IS e ISJ.**

**B) Relação de dependência entre CC e PP.**

**C) Herança múltipla de CB em relação a CC e PP.**

D) Herança múltipla de CB em relação a IS e ISJ.

---

## GABARITO

1. Na programação orientada a objetos, temos conceitos como Herança e Polimorfismo. Sobre esses conceitos analise as assertivas e assinale a alternativa que aponta a(s) correta(s).

I. Para evitar código redundante, o paradigma de orientação a objetos oferece uma estrutura hierárquica e modular para reutilização de código através de uma técnica conhecida como herança.

II. Herança permite projetar classes genéricas que podem ser especializadas em classes mais particulares, onde as classes especializadas reutilizam o código das mais genéricas.

III. Literalmente, “polimorfismo” significa “muitas formas”. No contexto e projeto orientado a objetos, entretanto, refere-se à habilidade de uma variável de objeto de assumir formas diferentes.

IV. Polimorfismo permite que os atributos de uma classe não tenham acesso diretamente.

A alternativa "C " está correta.

A questão aborda 3 conceitos importantes da orientação a objetos: Herança, Polimorfismo e Encapsulamento. O item IV acima está errado devido a troca com o conceito de encapsulamento visto na seção “Encapsulamento”.

2. Os bancos são instituições que investem fortemente na área de Tecnologia da Informação, inclusive com a contratação de milhares de profissionais e a construção de grandes ambientes de datacenter. Por isso, é um domínio de conhecimento bastante importante e que deve ser utilizado como exemplo durante um curso de graduação. Sabendo disso, analise a seguinte situação em um sistema bancário: A ContaBancaria(CB) especializa as classes ItemSuportado (IS) e ItemSujeitoAJuros (ISJ) e generaliza as classes ContaCorrente (CC) e Poupança (PP). Nesse sentido, é correto afirmar que ocorre (0,5)

A alternativa "D " está correta.

A questão aborda uma implementação prática de herança múltipla, onde a Classe ContaBancaria recebe herança de ItemSuportado e ItemSujeitoAJuros.

## MÓDULO 2

---

- ⦿ Descrever os conceitos básicos da programação orientada a objetos na linguagem Python

## CLASSES E OBJETOS EM PYTHON

Uma classe é uma declaração de tipo que encapsula constantes, variáveis e métodos que realizam manipulação dos valores dessas variáveis. Cada classe deve ser única em um sistema orientado a objetos.

## DEFINIÇÃO DE CLASSE

Como boa prática, cada classe deve fazer parte de um único arquivo.py para ajudar na estruturação do sistema orientado a objetos em Python. Outra boa prática consiste no nome da classe semelhante ao do arquivo. Por exemplo, definir no ambiente de desenvolvimento Python a classe Conta. O nome final do arquivo deve ser Conta.py.

Deve-se ressaltar que todos esses exemplos podem ser executados no terminal do Python.

```
class Conta:
```

```
pass
```

Uma classe é definida utilizando a instrução **class** e **:** para indicar o início do bloco de declaração da classe. A palavra reservada **pass** indica que a classe será definida posteriormente, servindo apenas para permitir que o interpretador execute a classe até o final sem erros.

## RECOMENDAÇÃO

É recomendável que você reproduza e execute todos os exemplos que serão apresentados a seguir.

## CONSTRUTORES E SELF

A classe conta foi criada, porém não lhe foram definidos atributos e instanciados objetos, característica básica dos programas orientados a objetos. Nas linguagens orientadas a objetos, para se instanciar objetos, devemos criar os construtores da classe.

Em Python, a palavra reservada **\_\_init\_\_()** serve para inicialização de classes, como abaixo:

```
class Conta:
```

```
def __init__(self, numero, cpf, nomeTitular, saldo):
```

```
    self.numero = numero
```

```
    self.cpf = cpf
```

```
    self.nomeTitular = nomeTitular
```

```
    self.saldo = saldo
```

Diferentemente de outras linguagens de programação orientadas a objetos, o Python constrói os objetos em duas etapas. A primeira etapa é utilizada com a palavra reservada **\_\_new\_\_**, que, em seguida, executa o método **\_\_init\_\_**.

O `__new__` cria a instância e é utilizado para alterar as classes dinamicamente para casos de sistemas que envolvam *metaclasses* e *frameworks*. Após `__new__` ser executado, este método chama o `__init__` para inicialização da classe com seus valores iniciais (MENEZES, 2019).

Para efeitos de comparação com outras linguagens de programação, e por questões de simplificação, consideraremos o `__init__` como o construtor da classe. Portanto, toda vez que instanciarmos objetos da classe *Conta*, será chamado o método `__init__`.

Analisando o nosso exemplo anterior, vimos a utilização da palavra *self* como parâmetro no construtor. Como o objeto já foi instanciado implicitamente pelo `__new__`, o método `__init__` recebe uma referência do objeto instanciado como *self*.

Analisando o restante do construtor da Figura 9, o código possui diversos comandos *self*, o qual indica referência ao próprio objeto. Por exemplo, o comando `self.numero` indica que o número é um atributo do objeto, ao qual é atribuído um valor. O restante dos comandos *self* indicam que foram criados os atributos `cpf`, `nomeTitular` e `saldo` referentes a classe *Conta*.

Vamos instanciar o nosso objeto com o método `__init__`:

```
>>>from Conta import Conta
>>>c1 = Conta(1,1,"Joao",1000)
```

Importante ressaltar que, em Python, não é obrigatório ter um método construtor na classe, apenas se for necessária alguma ação na construção do objeto, como inicialização e/ou atribuição de valores. Segue um exemplo de uma classe sem um método construtor:

```
class A():
    def f():
        print 'foo'
```

## MÉTODOS

Toda classe deve possuir um conjunto de métodos para manipular os atributos e, por consequência, o estado do objeto. Por exemplo, precisamos depositar dinheiro na conta para aumentar o valor da conta corrente:



```
def __depositar__(self,valor)
```

```
    self.saldo += valor
```

No código acima, foi definido um método `__depositar__` que recebe a própria instância do objeto através do `self` e de um parâmetro `valor`. O número passado por meio do parâmetro será adicionado ao saldo da conta do cliente. Vamos supor que o estado anterior do objeto representasse o saldo com o valor zero da conta. Após a chamada desse método passando como parâmetro o valor 300, através da referência `self`, o estado da conta foi alterado com o novo saldo de 300 reais.

Segue o novo código:

```
class Conta:
```

```
    def __init__(self, numero, cpf, nomeTitular, saldo):
```

```
        self.numero = numero
```

```
        self.cpf = cpf
```

```
        self.nomeTitular = nomeTitular
```

```
        self.saldo = saldo
```

```
    def depositar(self, valor):
```

```
        self.saldo += valor
```

```
    def sacar(self, valor):
```

```
        self.saldo -= valor
```

No exemplo anterior, adicionamos mais um método `sacar(self, valor)`, onde subtraímos o valor, passado como parâmetro, do saldo do cliente. Pode ser adicionado um método `extrato` para avaliar os valores atuais da conta corrente, ou seja, o estado atual do objeto. Por exemplo, a conta tinha saldo de 300 reais após o primeiro depósito. Após a chamada de `sacar (100)`, o saldo da conta será 200 reais.

```
>>>from Conta import Conta
```

```
>>>c1 = Conta(1,1,"Joao",0)
```

.....c1.depositar(300)

.....c1.sacar(100)

class Conta:

def \_\_init\_\_(self, numero, cpf, nomeTitular, saldo):

self.numero = numero

self.cpf = cpf

self.nomeTitular = nomeTitular

self.saldo = saldo

def depositar(self, valor):

self.saldo += valor

def sacar(self, valor):

self.saldo -= valor

def gerarextrato(self):

print(f"numero: {self.numero} \n cpf: {self.cpf}\nsaldo: {self.saldo}")

Se o método gerarextrato() for executado, o valor impresso será:

....c1.gerarextrato()

200

## MÉTODOS COM RETORNO

Em Python, não é obrigatório um comando para indicar quando o método deve ser finalizado. Porém, na orientação a objetos, como na programação procedural, é bastante comum retornar um valor a partir da análise do estado do objeto. Conforme exemplificado a seguir, não é permitido o saque de um valor maior do que o saldo atual do cliente, portanto, retorna a resposta “False” para o objeto que está executando o saque. O método deve ficar da seguinte forma:

```
def sacar(self,valor):  
    if self.saldo < valor:  
        return False  
    else:  
        self.saldo -= valor  
        return True
```

```
....c1.sacar(100)
```

```
True
```

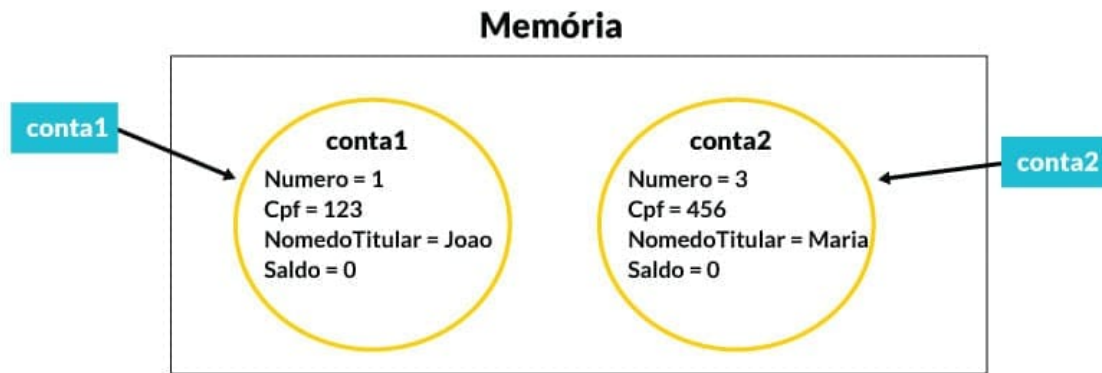
Agora, questione-se: ao executar o comando `c1.sacar(300)`, qual será o retorno?

## REFERÊNCIAS ENTRE OBJETOS NA MEMÓRIA

Uma classe pode ter mais de uma ou várias instâncias de objetos na memória, como exemplificado na imagem a seguir:

```
from Conta import Conta  
conta1 = Conta(1, 123, 'Joao',0)  
conta2 = Conta(3, 456, 'Maria',0)
```

O Resultado na memória após a execução é apresentado na Figura 15:



Fonte: Autor

📷 Figura 15 – Estado da memória conta1 e conta2.

Na memória, foram criados dois objetos diferentes referenciados pelas variáveis `conta1` e `conta2` do tipo `conta`. Os operadores `=="` e `!="` comparam se as duas variáveis de referência apontam para o mesmo endereço de memória (CAELUM, 2020).

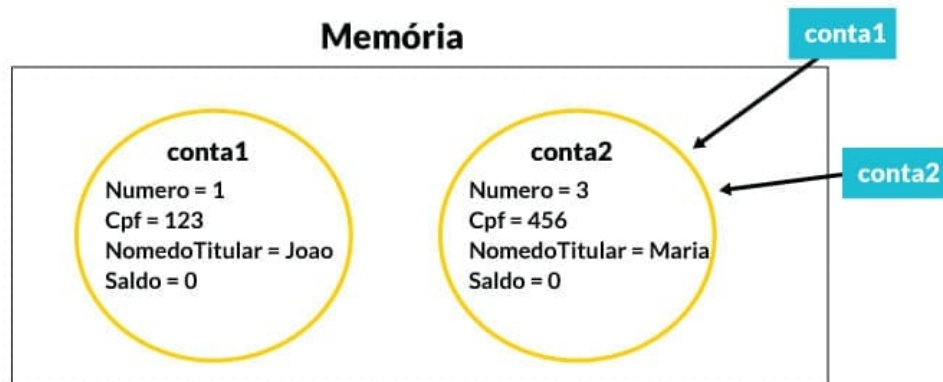
```
>>> if (conta1 != conta2):  
... print("Endereços diferentes de memória")
```

Pelo esquema da memória, apresentado na Figura 15, realmente `conta1` e `conta2` apontam para endereços diferentes de memória.

O comando `"="` realiza o trabalho de igualar a posição de duas referências na memória.

Fazendo `conta1 = conta2`, podemos ver o resultado:

```
conta1 = conta2  
if (conta1 == conta2):  
... print("enderecos iguais de memoria")
```



Fonte: Autor

📷 Figura 16 – Estado da memória conta1 e conta2 no mesmo endereço.

Se executarmos os comandos referenciando o CPF da conta, verificamos que possuem os mesmos valores, conforme mostrado acima.

```
>>>conta1.cpf
```

```
456
```

```
>>> conta2.cpf
```

```
456
```

Esse exemplo pode ser estendido para realizar uma transferência de valores de uma conta para outra. A segunda conta deve ser passada como parâmetro do método para ser executada a transferência. O método deve ficar da seguinte maneira:

```
class Conta:
```

```
def __init__(self, numero, cpf, nomeTitular, saldo):
```

```
    self.numero = numero
```

```
    self.cpf = cpf
```

```
    self.nomeTitular = nomeTitular
```

```
    self.saldo = saldo
```

```

def depositar(self, valor):
    self.saldo += valor

def sacar(self, valor):
    if self.saldo < valor:
        return False
    else:
        self.saldo -= valor
    return True

def gerarextrato(self):
    print(f"numero:{self.numero}\n cpf:{self.cpf}\nsaldo:{self.saldo}")

def transfereValor(self, contaDestino, valor):
    if self.saldo < valor:
        return ("Não existe saldo suficiente")
    else:
        contaDestino.depositar(valor)
        self.saldo -= valor
    return("Transferencia Realizada")

```

```

>>>from Conta import Conta
... conta1 = Conta(1, 123,'Joao',0)
... conta2 = Conta(3, 456,'Maria',0)
... conta1.depositar(1000)
... conta1.transfereValor(conta2,500)
... print(conta1.saldo)
... print(conta2.saldo)

```

500

500

Em resumo, foi depositado 1000 na conta1 e realizada uma transferência de valor de 500 para conta2. No final, o saldo ficou 500 para conta1 e 500 para conta2.

Importante ressaltar que, no comando `conta1.transfereValor(conta2,500)`, é passada uma referência da conta2 para o objeto contaDestino por meio de um operador “=”. O comando `contaDestino = conta2` é executado internamente no Python.

## AGREGAÇÃO

Para atender novas necessidades do sistema de conta corrente do banco, agora se faz necessário adicionar uma funcionalidade para gerenciamento de conta conjunta, ou seja, uma conta corrente pode ter um conjunto de clientes associados. Isto pode ser representado como uma agregação, conforme apresentado no esquema a seguir. Uma observação: o losango na imagem tem a semântica da agregação.

Fonte: Autor

 Figura 18 – Classe agrega 1 ou vários clientes

Para agregação, devemos criar dois arquivos `contas.py` e `cliente.py`:

```
class Conta:
```

```
def __init__(self,cpf,nome,endereco):
```

```
self.cpf = cpf
```

```
self.nome = nome
```

```
self.endereco = endereco
```

```
class Conta:
    def __init__(self, clientes, numero, saldo):
        self.clientes = clientes
        self.numero = numero
        self.saldo = saldo
    def depositar(self, valor):
        self.saldo += valor
    def sacar(self, valor):
        if self.saldo < valor:
            return False
        else:
            self.saldo -= valor
            return True
    def transfereValor(self, contaDestino, valor):
        if self.saldo < valor:
            return ("Não existe saldo suficiente")
        else:
            contadestino.depositar(valor)
            self.saldo -= valor
            return("Transferencia Realizada")
    def gerarsaldo(self):
        print(f"numero:{self.numero}\n saldo: {self.saldo}")
```

Um programa testecontas.py deve ser criado para o usarmos na instanciação dos objetos das duas classes e gerarmos as transações realizadas nas contas dos clientes.



```
from contas import Conta
from clientes import Cliente

cliente1 = Cliente(123, "Joao", "Rua 1")
cliente2 = Cliente(345, "Maria", "Rua 2")
conta1 = Conta([cliente1, cliente2], 1, 0) (1)

conta1.gerarsaldo()
conta1.depositar(1500)
conta1.sacar(500)
conta1.gerarsaldo()
```

## ATENÇÃO

Em (1) foi instanciado um objeto conta1 com dois clientes agregados: cliente1 e cliente2. Esses dois objetos são passados como parâmetros em (1).

Qual o resultado dessa execução? Qual valor final na conta?

Sugestão: altere o programa do código anterior para criar mais uma conta para dois clientes diferentes. Como desafio, tente, através do objeto conta, imprimir o nome e o endereço dos clientes associados às contas.

## COMPOSIÇÃO

A classe conta ainda não está completa de acordo com as necessidades do sistema de conta corrente do banco. Isso ocorre porque o banco precisa gerar extratos contendo o histórico de todas as operações realizadas para conta corrente. Para isso, o sistema deve ser atualizado para adicionar uma composição de cada conta com o histórico de operações realizadas. O diagrama a seguir representa a composição entre as classes Conta e Extrato. Esta composição representa que uma conta pode ser composta por vários extratos. Uma observação: o losango preenchido tem a semântica da composição.

Fonte: Autor

📷 Figura 22 – Classe Conta composta de 1 ou mais extratos

A classe Extrato tem as responsabilidades de armazenar todas as transações realizadas na conta e de conseguir imprimir um extrato com a lista dessas transações.

```
class Extrato:
    def __init__(self):
        self.transacoes = []

    def extrato(self, numeroconta):
        print(f"Extrato : {numeroconta} \n")
        for p in self.transacoes:
            print(f'{p[0]:15s} {p[1]:10.2f} {p[2]:10s} {p[3].strftime("%d/%b/%y"))}')
```

A classe conta possui todas as transações como sacar, depositar e transferir\_valor. Cada transação realizada deve adicionar uma linha ao extrato da conta. Inclusive, a composição Conta->Extrato deve ser inicializada no construtor da classe Conta, conforme exemplificado na Figura 23. No construtor de Extrato, foi adicionado um atributo transações, o qual foi inicializado para receber um **array de valores** – transações da conta.

A classe Conta alterada deve ficar da seguinte maneira:

---

# ARRAY DE VALORES

Um array ou vetor é uma estrutura que permite que sejam armazenados um conjunto de valores do mesmo tipo.

```
import datetime

from Extrato import Extrato

class Conta:

    def __init__(self, clientes, numero, saldo):

        self.clientes = clientes

        self.numero = numero

        self.saldo = saldo

        self.sata abertura = datetime.datetime.today()

        self.extrato = Extrato () (1)

    def depositar(self, valor):

        self.saldo += valor

        self.extrato.trasacoes.append(["DEPOSITO", valor, "Data", datetime.datetime.today ()])(2)

    def sacar(self, valor):

        if self.saldo < valor:

            return False

        else:

            self.saldo -= valor
```

```

self.extrato.transacoes.append(["SAQUE", valor, "Data", datetime.datetime.today()]) (3)

return True

def transfereValor(self, contadestino, valor):

if self.saldo < valor:

return "Não existe saldo suficiente"

else:

contadestino.depositar(valor)

self.saldo -= valor

self.extrato.transacoes.append(["TRANSFERENCIA", valor, "Data", datetime.datetime.today()]) (4)

return "Transferencia Realizada"

def gerarsaldo(self):

print(f"numero: {self.unmero}\n saldo:{self.saldo}")

```

Alterações da classe conta:

Adição da linha (1) – criação de um atributo extrato, fazendo referência a um objeto Extrato

Adição das linhas (2), (3) e (4) – adição de linhas ao array de transações do objeto Extrato através do atributo extrato.

Execução no terminal:

```

>>>from clientes import Cliente

... from ContasClientesExtrato import Conta

... cliente1 = Cliente("123","Joao","Rua X")

... cliente2 = Cliente ("456","Maria","Rua W")

... conta1 = Conta([cliente1,cliente2],1,2000)

... conta1.depositar(1000)

... conta1.sacar(1500)

```

**conta1.extrato.extrato(conta1.numero)**

**Extrato : 1**

**DEPOSITO 1000.00 Data 14/Jun/2020**

**SAQUE 1500.00 Data 14/Jun/2020**

## **DICA**

Experimente adicionar mais uma conta e realize uma transferência de valores entre ambas. Depois, crie uma classe Banco para armazenar todos os clientes e todas as contas do banco.

# **ENCAPSULAMENTO**

Conforme apresentado no módulo anterior, o encapsulamento é fundamental para a manutenção da integridade dos objetos e proibir qualquer alteração indevida nos valores dos atributos (estado) do objeto (CAELUM, 2020). Este ponto foi fundamental para a popularização da orientação aos objetos: reunir dados e funções em uma única entidade e proibir a alteração indevida dos atributos.

## **EXEMPLO**

No caso da classe conta, imagine que algum programa tente realizar a seguinte alteração direta no valor do saldo:

conta1.saldo = -200

Esse comando viola a regra de negócio do método sacar(), que indica para não haver saque maior do que o valor e deixar a conta no negativo (estado inválido para o sistema).

```
def sacar(self, valor):  
    if self.saldo < valor:  
        return False  
  
    else:  
        self.saldo -= valor  
        self.extrato.transacoes.append(["SAQUE", valor, "Data", datetime.datetime.today()])  
        return True
```

Como proibir alterações indevidas dos atributos em Python?

## ATRIBUTOS PÚBLICOS E PRIVADOS

Para seguir o encapsulamento e proibir alterações indevidas dos atributos, devemos definir atributos privados para a classe. Por *default*, em Python, os atributos são definidos como público, ou seja, podem ser acessados diretamente sem respeitar o encapsulamento - acesso feito apenas através de métodos do objeto.

Para tornar um atributo privado, devemos iniciá-lo com dois *underscores* ('\_\_'). A classe conta possui todos os atributos privados:

```
>>>class Conta:  
... def __init__(self,numero,saldo):  
...     self.numero = __numero  
...     self.saldo = __saldo:
```

Qual o retorno do interpretador ao se acessar um atributo privado para classe Conta?

```
>>> conta = Conta(1,1000)
```

```
>>> conta.saldo
```

*Traceback (most recent call last):*

*File "<input>", line 1, in <module>*

*AttributeError: 'Conta' object has no attribute 'saldo'*

É importante ressaltar que, em Python, não existe realmente atributos privados. O interpretador renomeia o atributo privado para `__nomedaClasse__nomedoatributo`. Portanto, o atributo ainda pode ser acessado. Embora funcione, é considerado uma prática que viola o princípio de encapsulamento da orientação a objetos.

```
>>> conta._Conta__saldo
```

```
1000
```

Na prática, deve existir uma disciplina para não serem acessados diretamente os atributos como `__` ou `_` definido nas classes.

## DECORATOR @PROPERTY

Uma estratégia importante disponibilizada pelo Python são as *properties*. Utilizando o *decorator property* nos métodos, mantém-se os atributos como protegidos, e estes são acessados apenas através dos métodos “decorados” com *property* (CAELUM, 2020).

No caso da classe `conta`, não se pode acessar o atributo `saldo` (privado) para leitura. Com o código, ele será acessado pelo método *decorator* `@property`:

```
@property
```

```
def saldo(self):
```

```
    return self._saldo
```

Os métodos decorados com a *property* `@setter` forçam que todas alterações de valor dos atributos privados devem passar por esses métodos.

Notação:

```
@<nomedometodo>.setter
```

```
@saldo.setter
```

```
def saldo(self, saldo):
```

```
if (self.saldo < 0):
```

```
print("saldo inválido")
```

```
else:
```

```
self._saldo = saldo
```

Os properties ajudam a garantir o encapsulamento no Python. Uma boa prática implementada em todas as linguagens orientadas a objetos é definir esses métodos apenas se realmente houver regra de negócios diretamente associada ao atributo. Caso não haja, deve-se deixar o acesso aos atributos conforme definido na classe.

## ATRIBUTOS DE CLASSE

Existem algumas situações que os sistemas precisam controlar valores associados à classe, e não aos objetos (instâncias) das classes; por exemplo, ao desenvolver um aplicativo de desenho, como o Paint, que precisa contar o número de círculos criados na tela.

Inicialmente, a classe Circulo vai ser criada:

```
class Circulo():
```

```
def __init__(self, pontox, pontoy, raio):
```

```
self.pontox = pontox
```

```
self.pontoy = pontoy
```

```
self.raio = raio
```

No entanto, conforme mencionado, é necessário controlar a quantidade de círculos criados.

```
class Circulo():
```



```
total_circulos = 0 (1)
```

```
def __init__(self, pontox, pontoy, raio):  
    self.pontox = pontox  
    self.pontoy = pontoy  
    self.raio = raio  
    self.total_circulos += 1 (2)
```

Em (1), indicamos para o interpretador que seja criada uma variável `total_circulos`. Como a declaração está localizada antes do `init`, o interpretador “entende” que se trata de uma variável de classe, ou seja, terá um valor único para todos objetos da classe. Em (2), é atualizado o valor da variável de classe a cada instanciação de um objeto da classe `Circulo`.

Como acessar os valores armazenados em uma variável de classes?

```
>> from Circulo import Circulo
```

```
>>> circ1 = Circulo(1,1,10)
```

```
>>> circ1.total_circulos
```

```
1
```

```
>>> circ2 = Circulo(2,2,20)
```

```
>>> circ2.total_circulos
```

```
2
```

```
>>> Circulo.total_circulos
```

```
2
```

Esse acesso direto ao valor da variável de classe não é desejado. Deve-se colocar a variável com atributo privado com o *underscore* ‘`_`’.

Como fica agora? O resultado é apresentado a seguir.

```
class Circulo:
```

```
    _total_circulos = 0
```

```
    def __init__(self, pontox, pontoy, raio):
```

```
        self.pontox = pontox
```

```
        self.pontoy = pontoy
```

```
        self.raio = raio
```

```
        circulo._total_circulos += 1
```

**Repetindo o mesmo código:**

```
>>>from Circulo import Circulo
```

```
>>>circ1 = Circulo(1,1,10)
```

```
>>>circ1._total_circulos
```

```
1
```

```
>>>Circulo.total_circulos
```

**Traceback (most recent call last):**

**File "<input>", line 1, in <module>**

**AttributeError: type object 'Circulo' has no attribute 'total\_circulos'**

## MÉTODOS DE CLASSE

Os métodos de classe são a maneira indicada para se acessar os atributos de classe. Eles têm acesso diretamente à área de memória que contém os atributos de classe. O esquema é apresentado na imagem a seguir.

## Figura 32 – Memória Estática

Para definir um método como estático, deve-se usar o decorator `@classmethod`. Assim, segue a classe `Circulo` alterada:

```
class Circulo:
```

```
    _total_circulos = 0
```

```
    def __init__(self, pontox, pontoy, raio):
```

```
        self.pontox = pontox
```

```
        self.pontoy = pontoy
```

```
        self.raio = raio
```

```
        type(self)._total_circulos += 1
```

```
    @classmethod
```

```
    def get_total_circulos(cls): (1)
```

```
        return cls._total_circulos (2)
```

Em (1), é criado o parâmetro `cls` como referência para classe. Em (2), é retornado o valor do atributo de classe `_total_circulos`.

## MÉTODOS PÚBLICOS E PRIVADOS

As mesmas regras definidas para atributos são válidas para os métodos das classes. Assim, o método pode ser declarado como privado, mas ainda pode ser chamado diretamente, como se fosse um método público. Os dois *underscores* antes do método indicam que ele é privado:

```
class Circulo:
```

```
    def __init__(self, clientes, numero, saldo):
```

```
self.clientes = clientes
self.numero = numero
self.saldo = saldo
def __gerarsaldo(self):
print(f"numero: {self.numero}\n saldo:{self.saldo}")
```

No código acima, foi definido o método `gerarsaldo` como privado. Portanto, pode ser acessado apenas internamente pela classe `Conta`. Um dos principais padrões da orientação a objetos consiste nos métodos públicos e nos atributos privados. Desse modo, respeita-se o encapsulamento.

## MÉTODOS ESTÁTICOS

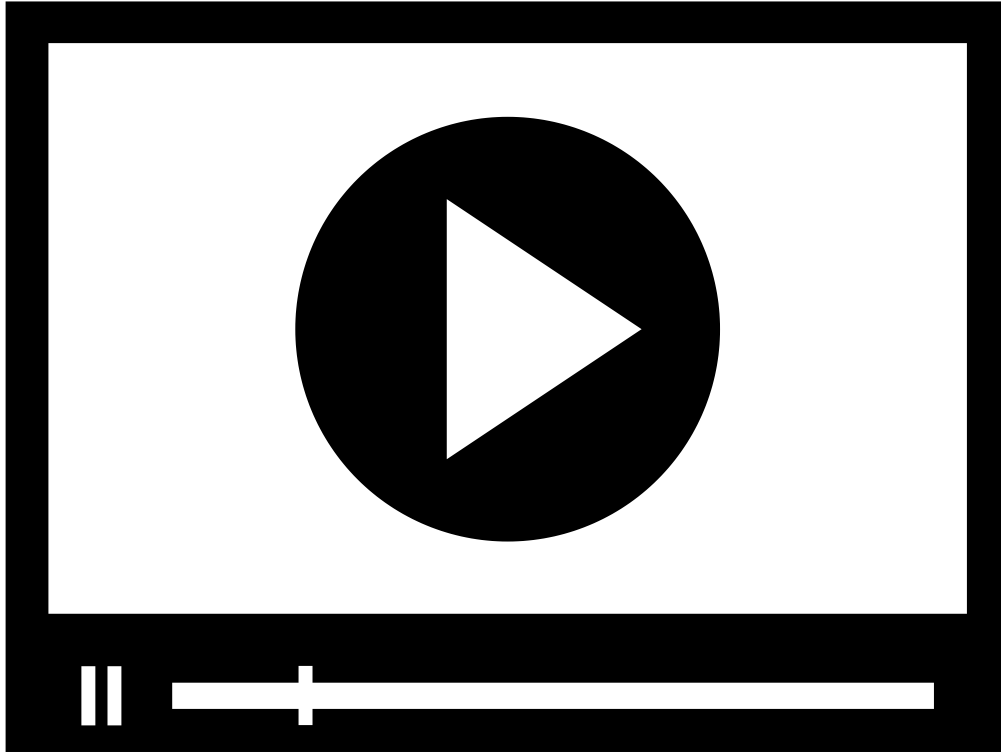
São métodos que podem ser chamados sem ter uma referência para um objeto da classe, ou seja, não existe obrigatoriedade da instanciação de um objeto da classe. O método pode ser chamado diretamente:

```
import math
class Math:

    @staticmethod
    def sqrt(x):
        return math.sqrt(x)

>>> Math.sqrt(20)
4.47213595499958
```

No caso acima, foi chamado o método `sqrt` da classe `Math` sem haver instanciado um objeto da classe `Math`. Os métodos estáticos não são uma boa prática na programação orientada a objetos. Devem ser utilizados apenas em casos especiais, como classes de log em sistemas.



## **ENTENDA MAIS SOBRE AGREGAÇÃO, CLASSES E MÉTODOS ESTÁTICOS**

No vídeo a seguir, faremos uma breve contextualização do tema.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



## VERIFICANDO O APRENDIZADO

**1. ANALISE O SEGUINTE CÓDIGO ESCRITO EM PYTHON, QUE DEFINE A ESTRUTURA DA CLASSE CONTABANCARIA:**

**CLASS CONTABANCARIA:**

**NUM\_CONTAS = 0**

**DEF \_\_INIT\_\_(SELF, CLIENTES, NUMERO, SALDO):**

**SELF.AGENCIA = AGENCIA**

**SELF.NUMERO = NUMERO**

**SELF.SALDO = SALDO**

**CONTABANCARIA.NUM\_CONTAS += 1**

```
DEF _DEL_(SELF):  
CONTABANCARIA.NUM_CONTAS -= 1  
DEF DEPOSITAR(SELF,VALOR):  
SELF.SALDO = SELF.VALOR + VALOR  
DEF SACAR(SELF, VALOR):  
SELF.SALDO = SELF.VALOR - VALOR  
DEF CONSULTARSALDO(SELF):  
RETURN SELF.SALDO
```

**SOBRE A CLASSE ACIMA E AS REGRAS DE PROGRAMAÇÃO ORIENTADA A OBJETOS EM PYTHON, A OPÇÃO INCORRETA:**

- A) A criação de objetos chama primeiro o método `__new__()` e, em seguida, o `__init__()`.
- B) A palavra `self` deve ser fornecida como argumento em todos os métodos públicos de instâncias.
- C) A variável `num_contas` é encapsulada e individual para cada instância da classe.
- D) O método `__del__` é privado.

**2. QUAL A DIFERENÇA NA UTILIZAÇÃO DOS DECORATORS `@staticmethod` E `@classmethod`:**

- A) O decorator `@staticmethod` definem métodos de classe que manipulam atributos de classe.
- B) O decorator `@classmethod` definem métodos estáticos que permite acesso a métodos sem instanciação da classe.

**C)** O decorator `@classmethod` permite que os atributos de classe sejam alterados na área de memória.

**D)** os decorators `@staticmethod` e `@classmethod` podem ser usados de forma intercambiáveis.

---

## **GABARITO**

**1. Analise o seguinte código escrito em Python, que define a estrutura da classe `ContaBancaria`:**

```
class ContaBancaria:
    num_contas = 0
    def __init__(self, clientes, numero, saldo):
        self.agencia = agencia
        self.numero = numero
        self.saldo = saldo
    ContaBancaria.num_contas += 1
    def _del_(self):
        ContaBancaria.num_contas -= 1
    def depositar(self, valor):
        self.saldo = self.valor + valor
    def sacar(self, valor):
        self.saldo = self.valor - valor
    def consultarSaldo(Self):
        return self.saldo
```

**Sobre a classe acima e as regras de programação orientada a objetos em Python, a opção INCORRETA:**



A alternativa **"B "** está correta.

A instância do objeto (self) deve ser passada implicitamente em cada chamada do método.

## 2. Qual a diferença na utilização dos decorators `@staticmethod` e `@classmethod`:

A alternativa **"C "** está correta.

O decorator `@classmethod` fica armazenado na mesma área de memória dos atributos de classe. Portanto, pode alterar os valores dos atributos de classe.

# MÓDULO 3

---

🕒 Descrever os conceitos da orientação a objetos como Herança e Polimorfismo

## HERANÇA E POLIMORFISMO

As linguagens orientadas a objeto oferecem recursos que permitem o desenvolvimento de sistemas de uma forma mais ágil e eficiente. Esses recursos são a herança, polimorfismo e o tratamento das exceções. A herança permite uma redução da repetição de código, permitindo que uma classe filho herde métodos e atributos da classe pai. O polimorfismo permite que, em determinadas situações, os objetos possam ter

comportamentos específicos. Caso haja algum problema durante a execução do código, o tratamento de exceções facilita a manipulação do erro.

# HERANÇA

É um dos princípios mais importantes da programação orientada a objetos, pois permite a reutilização de código com a possibilidade de extensão para se ajustar às regras de negócio dos sistemas.

## ★ EXEMPLO

A área de produtos do banco define quais clientes podem ter acesso a um produto chamado Conta especial, o qual busca atender quem possui conta na instituição há mais de um ano.

Na visão de negócios, a conta especial possui a funcionalidade de permitir o saque da conta corrente até um certo limite, determinado durante a criação da conta. Nesse ponto, há um dos grandes ganhos da orientação a objetos. O objeto do mundo real Conta Especial será mapeado para uma classe ContaEspecial que herdará todo código de Conta, conforme a figura 36:

Fonte: Autor

📷 Figura 36 - Herança Conta -> ContaEspecial.

Este é o código da implementação da nova classe Conta Especial:

```
from ContasClientesExtrato import Conta
import datetime

class ContaEspecial(Conta):
    def __init__(self, clientes, numero, saldo, limite):
        Conta.__init__(self, clientes, numero, saldo) (1)
        self.limite = limite (2)
    def sacar(self, valor): (3)
        if (self.saldo + self.limite) < valor:
            return False
        else:
            self.saldo -= valor
            self.extrato.transacoes.append(["SAQUE", valor, "Data", datetime.datetime.today()])
            return True
```

Analisando o código ContaEspecial acima, tem-se as seguintes modificações:

## MÉTODO CONSTRUTOR `__INIT__` -

A classe tem que ser instanciada com passagem do limite como um parâmetro da construção do objeto. O método `__init__` foi sobrescrito da superclasse Conta. O método `super()` foi utilizado para chamar um método da superclasse e pode ser utilizado em qualquer método da subclasse (REAL PYTHONON, 2020). A orientação a objetos permite, inclusive, a reutilização do construtor da classe pai (1).

## ATRIBUTO LIMITE

Foi adicionado apenas na subclasse ContaEspecial, onde será utilizado para implementar a regra de saques além do valor do saldo (2).

# MÉTODO SACAR()

O método precisa verificar se o valor a ser sacado, passado como parâmetro, é menor do que a soma do saldo atual mais o limite da conta especial. Nesse caso, a classe Conta Especial reescreveu o método sacar da superclasse Conta. Essa característica é conhecida como sobrescrita (override) dos métodos (3).

A execução no terminal gera o seguinte:

```
>>>from clientes import Cliente
```

```
... from ContasClientesExtrato import Conta
```

```
... from ContaEspecial import ContaEspecial
```

```
... cliente1 = Cliente("123","Joao","Rua X")
```

```
... cliente2 = Cliente ("456","Maria","Rua W")
```

```
... cliente3 = Cliente ("789","Joana", "Rua H")
```

```
... conta1 = Conta([cliente1,cliente2],1,2000)
```

```
... conta2 = ContaEspecial([cliente3],3,1000,2000)
```

```
... conta2.depositar(100)
```

```
... conta2.sacar(3200)
```

```
Valor do Saque    3200.00 maior que o valor do saldo mais o limite    3100.00
```

```
>>> conta2.extrato.extrato(conta2.numero)
```

```
Extrato : 3
```

```
DEPOSITO    100.00 Data    14/Jun/2020
```

Veja o diagrama atualizado com a implementação dos métodos na subclasse:

📷 Figura 38. Herança Conta -> ContaEspecial.

Importante ressaltar que ContaEspecial é uma classe comum e pode ser instanciada como todas as outras classes independentes da instanciação de objetos da classe Conta.

Uma análise comparativa com a programação procedural indica que, mesmo em caso com código parecido, o reuso era bastante baixo. O programador tinha que criar um programa Conta Corrente Especial repetindo todo o código já definido no programa Conta Corrente. Com a duplicação do código, era necessário realizar manutenção de dois códigos parecidos em dois programas diferentes.

## HERANÇA MÚLTIPLA

A herança múltipla é um mecanismo que possibilita uma classe herdar código de duas ou mais superclasses. Esse mecanismo é implementado por poucas linguagens orientadas a objetos e insere uma complexidade adicional na arquitetura das linguagens.

Para nosso sistema, vamos considerar a necessidade de um novo produto que consiste em uma conta corrente, similar a definida anteriormente no sistema, com as seguintes características:

Deverá ter um rendimento diário com base no rendimento da conta poupança.

Deverá ser cobrada uma taxa de manutenção mensal, mesmo se o rendimento for de apenas um dia.

A Classe Poupança também será criada para armazenar a taxa de renumeração e o cálculo do rendimento mensal da poupança.

Este será o diagrama com as novas classes criadas no sistema de conta corrente:

Fonte: Autor

📷 Figura 39. Herança Múltipla.

A implementação ficou detalhada nos códigos a seguir:

```
import datetime

class ContaPoupanca:

    def __init__(self, taxaremuneracao):

        self.taxaremuneracao = taxaremuneracao

        self.data_abertura = datetime.datetime.today()


    def remuneracaoConta(self)

        self.saldo += self.saldo * self.taxaremuneracao


from ContasClientesExtrato import Conta
from ContaPoupanca import Poupanca


class ContaRemuneradaPoupanca(Conta, Poupanca): (1)

    def __init__(self, taxaremuneracao, clientes, numero, saldo, taxaremuneracao):

        Conta.__init__(self, clientes, numero, saldo) (2)

        Poupanca.__init__(self, taxaremuneracao) (2)

        self.taxaremuneracao = taxaremuneracao


    def remuneraConta(Self):

        self.saldo += self.saldo * (self.taxaremuneracao/30)

        self.saldo -= self.taxaremuneracao
```

Alguns pontos importantes sobre a implementação:

Declaração de herança múltipla: em (1) indica que a classe é herdeira de Conta e Poupança nessa ordem. Essa ordem tem importância, pois existem dois métodos no pai com o mesmo nome. O Python dá prioridade para a primeira classe que implementa esse método na ordem da

declaração – (PYTHON COURSE, 2020)

Construtor da classe: Deve ser chamado o construtor explicitamente das superclasses com o seguinte formato:

nomeclasse.\_\_init\_\_(construtores)

Execução no terminal:

```
>>>from clientes import Cliente
... from ContasClientesExtrato import Conta
... from ContaPoupanca import Poupanca
... from ContaRemuneradaPoupanca import ContaRemuneradaPoupanca
... cliente1 = Cliente("123","Joao","Rua X")
... cliente2 = Cliente ("456","Maria","Rua W")
... conta1 = Conta([cliente1,cliente2],1,2000)
... contapoupanca1 = Poupanca(0.1)
... contarenumerada1 = ContaRemuneradaPoupanca(0.1,cliente1,5,1000,5)
... contarenumerada1.remuneraConta()
... contarenumerada1.geralsaldo()
numero: 5
saldo: 998.3333333333334
```

## POLIMORFISMO

Polimorfismo é o mecanismo que permite que um método com o mesmo nome seja executado de modo diferente, dependendo do objeto que está chamando o método. A linguagem define em tempo de execução (late binding) qual método deve ser chamado. Essa característica é

bastante comum em herança de classes devido à redefinição da implementação dos métodos nas subclasses.

Vamos assumir que agora teremos uma entidade Banco que controla todas as contas criadas no sistema de conta corrente. As contas podem ser do tipo conta, contacomum ou contarenumerada. O cálculo do rendimento da conta Cliente desconta IOF e IR; a conta Renumerada desconta apenas o IOF; a conta Cliente não tem desconto nenhum. Todos os descontos são realizados em cima do valor bruto após o rendimento mensal. Uma vez por vez o Banco executa o cálculo do rendimento de todos os tipos de contas. O diagrama é apresentado na Figura 42.

Fonte: Autor

📷 Figura 42 - Diagrama Banco.

Vamos analisar agora a implementação das classes:

```
class ContaCliente:
```

```
    def __init__(self, numero, IOF, IR, valorinvestido, taxarendimento):
```

```
        self.numero = numero
```

```
        self.IOF = IOF
```

```
        self.IR = IR
```

```
        self.valorinvestido = valorinvestido
```

```
        self.taxarendimento = taxarendimento
```

```
    def CalculoRendimento(self):
```

```
        self.valorinvestido += (self.valorinvestido * self.taxarendimento)
```

```
        self.valorinvestido = (self.valorinvestido - (self.taxarendimento * self.IOF * self.IR))
```



```
def Extrato(Self): (1)

print (f'O saldo atual da conta {self.numero} é {self.valorinvestido:10.2f}')
```

```
from ContaCliente import ContaCliente

class ContaComum(ContaCliente)

def __init__(self,numero,IOF,IR,valorinvestido,taxarendimento):
    super().__init__(numero,IOF,IR,valorinvestido,taxarendimento)
```

```
def CalculoRendimento(Self): (2)

self.valorinvestido += (self.valorinvestido * self.taxarendimento)
```

```
from ContaCliente import ContaCliente

class ContaRemunerada(ContaCliente):

def __init__(self,numero,IOF,IR,valorinvestido,taxarendimento):
    super().__init__(numero,IOF,IR,valorinvestido,taxarendimento)
```

```
def CalculoRendimento(Self): (3)

self.valorinvestido += (self.valorinvestido * self.taxarendimento)

self.valorinvestido -= self.valorinvestido * self.IOF
```

Em (1), foi definido um método Extrato que é igual para as 3 classes, ou seja, as subclasses herdarão o código completo desse método. Em (2) e (3), as subclasses possuem regras de negócios diferentes, portanto sobrescreveram o método CalculoRendimento para atender às suas necessidades.

Vamos analisar a implementação da classe Banco e do programa que a utiliza:

```
class Banco():

def __init__(self, codigo,nome):
```

```
self.codigo = codigo
```

```
self.nome = nome
```

```
self.contas = []
```

```
def adicionaconta(self, contacliente):
```

```
self.contas.append(contacliente)
```

```
def calcularendimentomensal(self): (7)
```

```
for c in self.contas:
```

```
    c.CalculoRendimento()
```

```
def imprimesaldocontas(self):
```

```
for c in self.contas:
```

```
    banco1 = Banco(999,"teste")
```

```
    contacliente1 = ContaCliente (1,0.01,0.1,1000,0.05)
```

```
    contacomum1 = ContaComum(2,0.01,0.1,2000,0.05)
```

```
    contaremunerada1 = ContaRemunerada(3,0.01,0.1,2000,0.05)
```

```
    banco1.adicionaconta(contacliente1) (4)
```

```
    banco1.adicionaconta(contacomum1) (5)
```

```
    banco1.adicionaconta(contaremunerada1) (6)
```

```
    banco1.calcularendimentomensal(7)
```

```
    banco1.imprimesaldocontas() (8)
```



Em (4), (5) e (6), banco adiciona todas as contas da hierarquia em um único método devido ao teste “É-UM” das linguagens orientadas a objetos. No método, isso é definido para receber um objeto do tipo ContaCliente. Toda vez que o método é chamado, a linguagem testa se o objeto passado “É-UM” objeto do tipo ContaCliente.

---

Em (4), o objeto é da própria classe Conta Cliente. Em (5), o objeto contacomum1 passado é uma ContaComum, que passa no teste “É-UM”, pois uma ContaComum é uma ContaCliente também.



Em (6), o objeto contarenumerada1 “É-UM” objeto ContaComum. Essas ações são feitas de forma transparente para o programador.

---

Em (7), acontece a “mágica” do Polimorfismo. Pois, em (4), (5) e (6) são adicionadas contas de diferentes tipos para o array conta da classe Banco. Assim, no momento da execução do método c.calculorendimentomensal(), o valor de c recebe, em cada loop, um tipo de objeto diferente da hierarquia da classe ContaCliente. Portanto, na instrução c.CalculoRendimento(), o interpretador tem que identificar dinamicamente de qual objeto da hierarquia Conta Cliente deve ser chamado o método CalculoRendimento.



Em (8), acontece uma característica que vale ser ressaltada. Pelo polimorfismo, o interpretador irá verificar o teste “É-UM”, porém esse método não foi sobrescrito pelas subclasses da hierarquia ContaCliente. Portanto, será chamado o método Extrato da superclasse.

Se executarmos a classe o programa banco\_contarenumerada, o resultado será:

O saldo atual da conta 1 é: 1048.95

O saldo atual da conta 2 é: 2100.00

O saldo atual da conta 3 é: 2079.00

O polimorfismo é bastante interessante em sistemas com dezenas de subclasses herdeiras de uma única classe, assim todas as subclasses redefinem esse método. Sem o polimorfismo, haveria a necessidade de “perguntar” para a linguagem qual a instância do objeto em execução para chamar o método correto. Com base no polimorfismo, essa checagem é feita internamente pela linguagem de maneira transparente.


## CLASSES ABSTRATAS

Definir uma classe abstrata é uma característica bastante utilizada pela programação orientada a objetos. Esse é um tipo de classe que não pode ser instanciado durante a execução do programa orientado a objetos, ou seja, não podem existir objetos dessa classe sendo

executados na memória. O termo abstrato remete a um conceito do mundo real, considerando que preciso apenas de objetos concretos no programa.

A classe abstrata se encaixa perfeitamente no problema do sistema de conta corrente do banco. Nesse sistema, o banco não quer tratar clientes do tipo ContaCliente, e sim apenas os objetos do tipo ContaComum e Conta VIP.

Fonte: Autor

 Figura 48 - Diagrama de Classes abstratas.

Houve apenas adição de estereótipo <<abstract>> para indicar que Conta Cliente é uma classe abstrata.

O Python utiliza um módulo chamado abc para definir uma classe como abstrata, a qual será herdeira da superclasse ABC (Abstract Base Classes). Toda classe abstrata é uma subclasse da classe ABC (CAELUM, 2020). Para tornar a classe Conta Cliente abstrata, muda-se a definição para:

```
from abc import ABC
class ContaCliente(ABC):
```

Para uma classe ser considerada abstrata, tem de ter pelo menos um método abstrato. Esse método abstrato pode ter implementação, embora não faça sentido, pois ele deverá, obrigatoriamente, ser implementado pelas subclasses. Em nosso exemplo, como não teremos o ContaCliente, esta conta não terá Calculo do rendimento.

O decorator @abstractmethod indica para a linguagem que o método é abstrato (STANDARD LIBRARY, 2020), conforme o código apresentado na Figura 3.15:

```
from abc import ABC, abstractmethod
class ContaCliente(ABC)
def __init__(self, numero, IOF, IR, valorinvestido, taxarendimento):
self.numero = numero
```

```
self.IOF = IOF
self.IR = IR
self.valorinvestido = valorinvestido
self.taxarendimento = taxarendimento
@abstractmethod
def CalculoRendimento(self):
    pass
```

Quando se tentar instanciar um objeto da classe, obtém-se um erro indicando que essa classe não pode ser instanciada..

```
>>> from ContaClienteAbstrata import ContaCliente
```

```
cc1 = ContaCliente(1,0.1,0.25,0.1)
```

*Traceback (most recent call last):*

*File "<input>", line 2, in <module>*

***TypeError: Can't instantiate abstract class ContaCliente with abstract methods CalculoRendimento***

Apenas as subclasses ContaComum e ContaVIP podem ser instanciadas.

## ATENÇÃO

Um alerta importante é que as classes mencionadas devem, obrigatoriamente, implementar os métodos. Caso contrário, serão classes abstratas e delegarão para as suas subclasses a implementação concreta do método abstrato.

**Fica como sugestão** criar as classes concretas ContaComum e ContaVIP sem implementar o método abstrato do superclasse ContaCliente.

# EXCEÇÕES

Como diversas linguagens orientadas a objetos, o Python permite criar novos tipos de exceções para diferenciar os erros gerados pelas bibliotecas da linguagem dos erros gerados pelas aplicações desenvolvidas. Devemos usar a característica da herança para herdar novas exceções a partir classe Exception do Python (MENEZES, 2020).

```
class ExcecaoCustomizada(exception): (1)
    pass
    def throws(): (2)
        raise ExcecaoCustomizada
    try: (3)
        throws()
    except ExcecaoCustomizada as ex:
        print ("Excecao lançada")
```

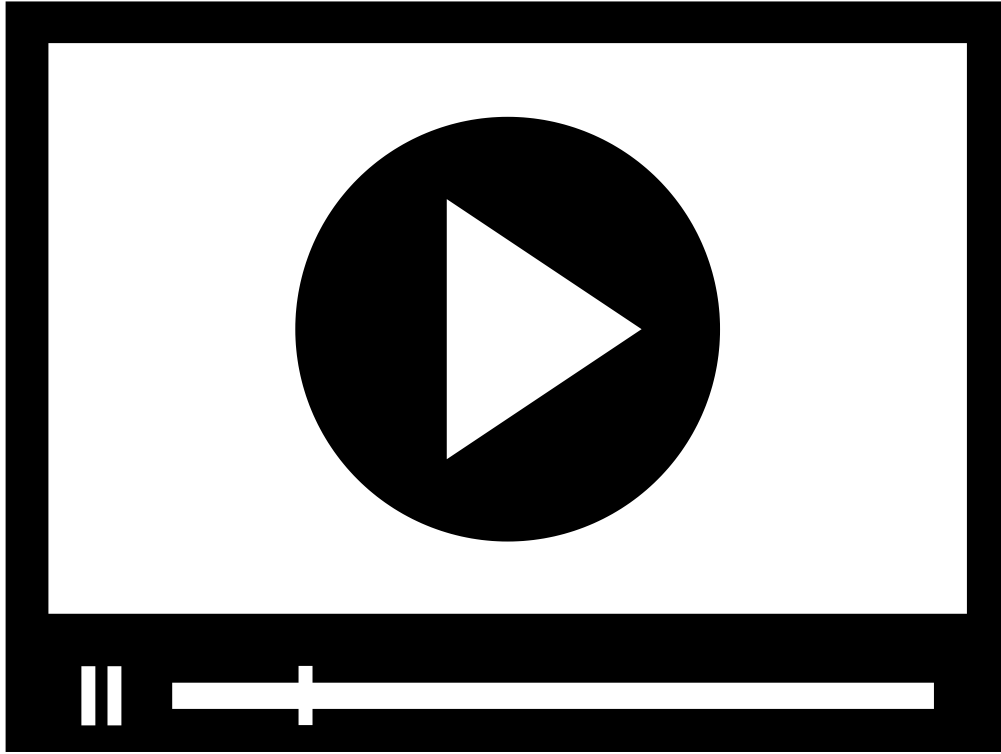
No código acima:

Em (1), definimos a exceção customizada ExcecaoCustomizada com o método pass, pois não executa nada relevante.

Em (2), é definido um método que, se for chamado, cria a exceção na memória ExcecaoCustomizada.

Em (3), é utilizado o try...except, que indica para o interpretador que a área do código localizada entre o try e o except poderá lançar exceções que deverão ser tratadas nas linhas de código após o except.

Ao final da execução, será impresso “Exceção lançada” pela captura da exceção lançada pelo método throws().



## **ENTENDA MAIS SOBRE HERANÇA, POLIMORFISMO E EXCEÇÕES**

No vídeo a seguir, faremos uma breve contextualização do tema.



Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



## VERIFICANDO O APRENDIZADO

### 1. SOBRE A LINGUAGEM DE PROGRAMAÇÃO PYTHON, MARQUE A ALTERNATIVA INCORRETA.

- A)** Python suporta a maioria das técnicas da programação orientada a objetos.
- B)** Python suporta e faz uso constante de tratamento de exceções como uma forma de testar condições de erro e outros eventos inesperados no programa.
- C)** A linguagem Python permite a utilização do conceito de sobrecarga de métodos através do polimorfismo dos métodos.
- D)** A separação de blocos de código em Python é feita utilizando a endentação de código.

## **2. EM RELAÇÃO AOS CONCEITOS DA ORIENTAÇÃO A OBJETOS E À IMPLEMENTAÇÃO EM PYTHON, PODEMOS AFIRMAR QUE:**

- A)** O Poliformismo permite a sobrecarga de métodos em uma classe Python.
- B)** Objetos diferentes podem ser agrupados na mesma classe, sempre que tenham o mesmo número de bytes.
- C)** Na linguagem Python existe o conceito de classes abstratas, que fornecem uma capacidade de reutilização para programas orientados a objetos.
- D)** A linguagem Python implementa parcialmente herança múltipla através do poliformismo.

---

## **GABARITO**

### **1. Sobre a linguagem de programação PYTHON, marque a alternativa incorreta.**

A alternativa **"C "** está correta.

A questão abrange uma visão geral sobre os conceitos de Orientação a Objetos implementados em Python. O Python não permite a sobrecarga de métodos nativamente.

### **2. Em relação aos conceitos da orientação a objetos e à implementação em Python, podemos afirmar que:**

A alternativa **"C "** está correta.

A questão abrange uma visão geral a implementação dos conceitos orientados a objetos implementados em Python. Uma classe em Python pode representar um objeto abstrato do mundo real, pois esse possui propriedades e responsabilidades que serão utilizados por outros

objetos da hierarquia no programa orientado a objetos.

## MÓDULO 4

---

- 🕒 Comparar a implementação dos conceitos orientados a objetos aplicados a Python com outras linguagens orientadas a objetos existentes no mercado

# LINGUAGEM DE PROGRAMAÇÃO ORIENTADAS A OBJETOS

As linguagens orientadas a objetos C++ e Java são bastante utilizadas assim como Python. Inclusive, estão em posições subseqüentes no ranking das linguagens mais utilizadas de todos os paradigmas (TIO-BE, 2020). Portanto, é interessante fazer um paralelo das principais características de cada linguagem em relação a orientada a objetos: Python (RAMALHO, 2020), C++ (C++, 2020) e Java (JAVA, 2020)

## COMPARAÇÃO COM C++ E JAVA

### INSTANCIAMENTO DE OBJETOS

As linguagens Java e C++ obrigam utilizar a palavra reservada **new** e indicar o tipo do objeto. Porém, a linguagem C++ referencia todos os objetos explicitamente através de ponteiros. Na tabela a seguir, o objeto Conta instanciado será referenciado pelo ponteiro \*c1. A linguagem

Python tem uma forma simplificada de instanciação de objetos.

Java	C++	Python
Conta c1 = new Conta()	Conta *c1 = new Conta()	c1 = Conta()

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

## CONSTRUTORES DE OBJETOS

As linguagens Java e C++ exigem um método definido como público e com o mesmo nome da classe. O Python obriga ser um método privado `__init__`, conforme apresentado na tabela 4.

Java	C++	Python
Utilizado um método público com o mesmo nome da classe sem tipo de retorno: Public Conta()	Utilizado um método com o mesmo nome da classe sem tipo de retorno: Conta::Conta(void)	Utilizando um método público com a obrigatoriedade da passagem do objeto com self: def __init__(self)

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

## MODIFICADORES DE ACESSO ATRIBUTOS

As linguagens C++ e Python possuem apenas os modificadores público e privado para atributos. No entanto, ao contrário de Python, C++ e Java, garantem que um atributo definido como privado seja acessado estritamente pela própria classe. O Python permite burlar um atributo

privado, conforme detalhado na Tabela 5.

Java	C++	Python
<p>Possui os seguintes modificadores:</p> <p>public</p> <p>private</p> <p>protected</p> <p>default</p>	<p>Possui os seguintes modificadores:</p> <p>public</p> <p>private</p>	<p>Possui os seguintes modificadores:</p> <p>público</p> <p>privado – iniciado com “_”</p> <p>Modificador privado pode ser burlado e acessado diretamente</p>

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

## HERANÇA MÚLTIPLA DE CLASSES

As linguagens C++ e Python implementam herança múltipla diretamente através de classes, enquanto Java implementa através de herança múltipla de interfaces.

Java	C++	Python
<p>Não implementa. Utiliza herança múltipla de interfaces para substituir essa característica.</p>	<p>Implementa com a referência das classes herdadas na declaração da classe:</p>	<p>Implementa com a referência das classes herdadas na declaração da classe:</p>

	Class ContaRemunerada: public Conta, public Poupanca)	class ContaRemuneradaPoupanca(Conta, Poupanca):
--	--	---

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

# CLASSES SEM DEFINIÇÃO FORMAL

A linguagem Java implementa o conceito como classes anônimas. Todas as linguagens implementam o conceito de classes sem definição formal.

Java	C++	Python
<pre>protected void Calcular(){     class Calculo{         private int soma;         public void setSoma(int soma) {             this.soma = soma;         }         public int getSoma() {             return soma;         }     } }</pre>	<p>Existe o conceito de classes locais – internos a funções:</p> <pre>void func() {     class LocalClass {     } }</pre>	<p>Existe o conceito de classes locais:</p> <pre>def scope_test():     def do_local():         spam = "local spam"</pre>

}		
}		
}		

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

# TIPOS PRIMITIVOS

As linguagens Java e C++ possuem um conjunto parecido de tipos primitivos, enquanto, em Python, todos as variáveis são definidas como objetos.

Java	C++	Python
Possui vários tipos primitivos:  int  byte  short  long  float  double  boolean  char	Possui vários tipos primitivos:  bool  char  int  float  double  void  wchar_t	Todas as variáveis são consideradas objetos:  >>>5.__add__(3)  8

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

## INTERFACES

As linguagens Java e C++ implementam interfaces simples e múltiplas, enquanto, em Python, não existe o conceito de interfaces.

Java	C++	Python
Implementa interfaces simples e múltiplas:  Simples - Class Funcionario implements Autenticavel  Múltipla - Class Funcionario implements Autenticavel, Descontavel	Implementa interfaces simples e múltiplas:  Simples - Class Funcionario: public Autenticavel  Múltipla - Class Funcionario: public Autenticavel, public Descontavel	Não implementa

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

## SOBRECARGA DE MÉTODOS

As linguagens Java e C++ implementam sobrecarga de métodos nativamente, enquanto Python não implementa nativamente.

Java	C++	Python
Implementa sobrecarga de métodos:  calculaimposto(salario)	Implementa sobrecarga de métodos:  calculaimposto(salario)	Não implementa



calculaImposto(salario,IR)	calculaImposto(salario,IR)	
----------------------------	----------------------------	--

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

# TABELA COMPARATIVA

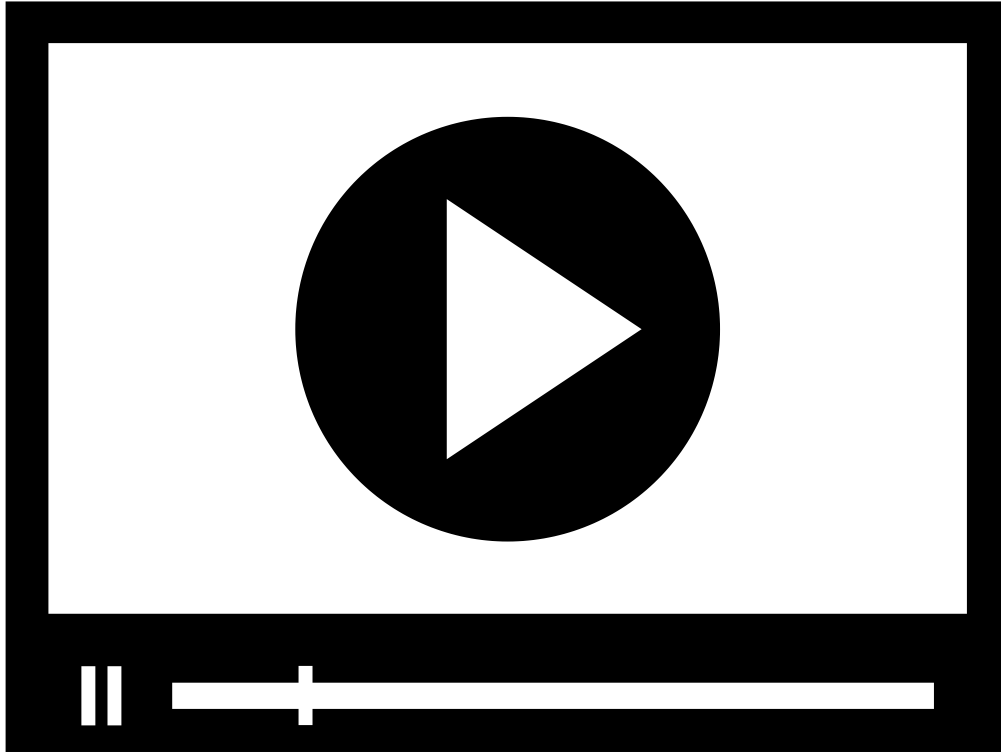
A Tabela 11 apresenta um resumo das características das linguagens Java, C++ e Python. A linguagem Python não possui atributos privados, interfaces e sobrecarga de métodos, que são necessidades fundamentais para a construção de sistemas orientados a objetos robustos baseado em *design patterns*. Portanto, Java e C++ levam uma vantagem considerável para construção de sistemas puramente orientados a objetos.

Por sua simplicidade, o Python vem crescendo como a primeira linguagem de programação ensinada na graduação e, devido a quantidade de bibliotecas estatísticas e frameworks web existentes, é utilizada para o desenvolvimento de algoritmos de Data Science e sistemas Web.

Características	Linguagens		
	Java	C++	Python
Instanciação de objetos	X	X	X
Construtores de objetos	X	X	X

Modificadores de acesso atributos	X	X	X(*)
Modificadores de acesso métodos	X	X	X
Herança múltipla		X	X
Classes informais		X	X
Tipos primitivos	X	X	
Interfaces	X	X	
Sobrecarga de métodos	X	X	

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal



## COMPARAÇÃO ENTRE AS LINGUAGENS C++, JAVA E PYTHON

No vídeo a seguir, faremos uma breve contextualização do tema.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



## VERIFICANDO O APRENDIZADO

### 1. EM PYTHON, COMO TRABALHAR SEMPRE COM TIPOS OBJETOS:

**A)** `x = c + ADD(a + b).`

**B)** `obj(8).__.`

**C)** `5.__add__(3).`

**D)** `sub(5).__add__(3).`

**2. ANALISANDO AS CARACTERÍSTICAS ORIENTADA A OBJETOS DE C++, JAVA E PYTHON, CONSIDERAMOS AS SEGUINTE AFIRMAÇÕES:**

**I. EM JAVA, É PERMITIDA A CRIAÇÃO DE HERANÇA MÚLTIPLA DE CLASSES ATRAVÉS DAS CLASSES DENOMINADAS INTERFACES**

**II. EM PYTHON, COMO EM C++ E JAVA, EXISTEM OS TIPOS PRIMITIVOS E OS OBJETOS PARA SEREM UTILIZADOS PELOS PROGRAMAS**

**III. EM PYTHON, O ENCAPSULAMENTO NÃO SEGUE OS PRINCÍPIOS DA ORIENTAÇÃO A OBJETOS**

**A) II, apenas.**

**B) II e III, apenas.**

**C) I e II, apenas.**

**D) III, apenas.**

---

## **GABARITO**

**1. Em Python, como trabalhar sempre com tipos objetos:**

A alternativa **"C "** está correta.

Na alternativa C, é realizada a soma seguindo a nota Python. A questão abrange o tratamento do Python em relação a todos os tipos serem considerados objetos.

**2. Analisando as características Orientada a Objetos de C++, Java e Python, consideramos as seguintes afirmações:**

**I. Em Java, é permitida a criação de herança múltipla de classes através das classes denominadas interfaces**

**II. Em Python, como em C++ e Java, existem os tipos primitivos e os objetos para serem utilizados pelos programas**

**III. Em Python, o encapsulamento não segue os princípios da orientação a objetos**

A alternativa **"D "** está correta.

Python não possui tipos privados. Os tipos privados que garantem o encapsulamento são em Java e C++.

## CONCLUSÃO

## CONSIDERAÇÕES FINAIS

O paradigma orientado a objetos é largamente utilizado no mercado devido a facilidade na transformação de conceitos do mundo real para objetos implementados em uma linguagem de software. A linguagem Python implementa o paradigma utilizando uma sintaxe simples e robusta. Portanto, vem ganhando mercado, tanto na área acadêmica, quanto na área comercial, para o desenvolvimento de sistemas orientado a objetos.

Outro ponto importante é que, devido a sua simplicidade, Python vem sendo utilizada como a primeira linguagem a ser aprendida nos primeiros períodos dos cursos de tecnologia e engenharia, inclusive para fixar os conceitos da orientação a objetos.

A quantidade de bibliotecas open-source disponíveis para a implementação de algoritmos de Data Science tornou a linguagem importante para a implementação de sistemas de aprendizado e visualização de dados.

Para ouvir um *podcast* sobre o assunto, acesse a versão online deste conteúdo.



Podcast disponível em:

## REFERÊNCIAS

ANAYA, M. **Clean Code in Python: Refactor Your Legacy Code Base**. Packt Publishing Ltd, 29 ago. 2018.

CAELUM. **Python e Orientação a Objetos**. Consultado em meio eletrônico em: 14 jun. 2020.

COSTA, M. **Análise Orientada a Objetos**, março de 2017, 100 f. Notas de Aula.

FARINELLI, F. **Conceitos básicos de programação orientada a objetos**. Consultado em meio eletrônico em: 14 jun. 2020.

GIRIDHAR, C. **Learning Python Design Patterns**. Packit Publishing, 2018.

JAVA. **Java Tutorial**. Consultado em meio eletrônico em: 14 jun. 2020.

MENEZES, N. C. **Introdução a programação com Python**. 3. ed. São Paulo: Novatec, 2019.

PAJANKAR, A. **Python Unit Test Automation**: Practical Techniques for Python Developers and Testers. Apress, 2017.

PYTHON COURSE, **Blog Python**. Consultado em meio eletrônico em: 14 jun. 2020.

QCONCURSOS, **Questões de Concursos**. Consultado em meio eletrônico em: 18 jun. 2020.

RAMALHO, L. **Fluent Python**. Sebastopol, CA: O'Reilly Media, 2015.

RAMALHO, L. **Introdução à Orientação a Objetos em Python** (sem sotaque). Consultado em meio eletrônico em: 14 jun. 2020.

REAL PYTHON, **Blog Python**, Consultado em meio eletrônico em: 14 jun. 2020.

RUMBAUGH, J. *et al.* **Modelagem e projetos baseados em objetos**, Rio de Janeiro: Campus, 1994.

STANDARD LIBRARY, **The Python Standard Library**. Consultado em meio eletrônico em: 24 jun. 2020.

TIO-BE. **TIO-BE Index**. Consultado em meio eletrônico em: 14 jun. 2020.

TUTORIALSPPOINT. **C++ Tutorial**. Consultado em meio eletrônico em: 14 jun. 2020.

---

## EXPLORE+

Para saber mais sobre os assuntos tratados neste tema:

O desenvolvimento de frameworks com metaclasses é um assunto importante e bastante utilizado no mercado. Para estudar um pouco mais esse tema, indicamos o livro *Fluent Python*, de Luciano Ramalho, publicado em 2015 pela O'Reilly Media.

Os designs patterns devem ser estudados e aplicados em sistemas orientados a objetos de qualquer porte. Para se aprofundar nesse assunto, indicamos a obra *Clean Clean Code in Python: Refactor Your Legacy Code Base*, de Mariano Anaya, publicado em 2018 pela



Packit Publishing e o livro *Learning Python Design Patterns*, de Giridhar publicado em 2018 Packit Publishing.

Os testes unitários buscam garantir a qualidade da implementação de acordo com as regras de negócios. Para se aprofundar nesse assunto, indicamos a obra *Python Unit Test Automation: Practical Techniques for Python Developers and Testers* de Ashwin Pajankar publicado em 2017 pela Apress.

---

## CONTEUDISTA

Marcelo Nascimento Costa

 **CURRÍCULO LATTES**