Python orientado a objeto

Prof. Marcelo Nascimento Costa

Prof. Kleber de Aguiar

Descrição

Introdução aos conceitos de: 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 orientado a objetos (OO).

Propósito

Compreender o desenvolvimento de software orientado a objetos utilizando uma linguagem de programação, como a do Python, que tem grande aceitação no meio comercial e acadêmico. Entender os conceitos e os pilares da orientação a objetos. 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 a modularização e a utilização de bibliotecas em Python. Antes de iniciar a leitura deste conteúdo, é necessário possuir uma versão do interpretador Python e o ambiente de desenvolvimento PyCharm (ou outro ambiente que suporte o desenvolvimento na linguagem Python).

Objetivos

Módulo 1

Orientação a objetos

Definir os conceitos gerais da orientação a objetos.

Módulo 2

Orientação a objetos na linguagem Python

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

Módulo 3

Orientação a objetos como herança e polimorfismo

Descrever os conceitos da orientação a objetos como herança e polimorfismo.

Módulo 4

Orientação a objetos aplicados a Python e outras existentes no mercado

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 se aproximar 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. Devido à sua sintaxe simples e robusta, ela é uma ferramenta poderosa para a implementação de sistemas orientados a objetos.

Neste conteúdo, apresentaremos os conceitos da orientação a objetos, demonstrando como implementá-los em Python. Também faremos uma comparação dele com as linguagens Java e C++ - em relação às principais características da orientação a objetos.

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

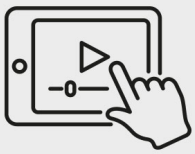
1 - Orientação a objetos

Ao final deste módulo, você será capaz de definir os conceitos gerais da orientação a objetos.

Vamos começar!

Aplicações do paradigma orientado a objetos

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



Conceitos e pilares de programação orientada a objetos

Conceitos de programação orientada a objetos (POO)

A POO 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, em seguida, para a implementação dos respectivos modelos nas linguagens de programação orientadas a objetos.

Conforme apresentam Rumbaugh e demais autores (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 mudo real pensando no projeto, que, por sua vez, é influenciado por ideias sobre codificação. Tais ideias também 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 daqueles relativos à implementação dos projetos.

Essa abordagem procura refletir os problemas do mundo real por meio da 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).

O software é organizado como uma coleção de objetos separados que incorpora tanto a estrutura quanto o comportamento dos dados. Isso 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 disso, eles são mais adaptáveis às modificações e às 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 altera 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).

> **Durante a etapa de levantamento dos objetos, deve-se analisar apenas os objetos relevantes (abstrair) com as respectivas características mais importantes para o problema a ser resolvido.**

Exemplo

As características de uma pessoa para um sistema acadêmico podem ser a formação ou o nome do pai e o da mãe, enquanto as de um indivíduo, para o sistema de controle de uma academia, são a altura e o peso.

Atributos

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

Observe a seguir a diferença entre os atributos de duas mulheres:

Atributos de objetos da pessoa 1.

Atributos e valores armazenados

Nome - Maria

Idade - 35

Peso - 63kg

Altura - 1,70m

Atributos e valores armazenados

Nome - Joana

Idade - 30

Peso - 60kg

Altura - 1,65m

Atributos de objetos da pessoa 2.

Operações

Uma operação é uma função ou transformação que pode ser aplicada a objetos ou dados pertencentes a um objeto. É importante dizer que todo objeto possui um conjunto de operações, as quais, aliás, podem ser chamadas por outros objetos com o propósito de colaborarem entre si.

> ### Esse conjunto de operações é conhecido como interface.

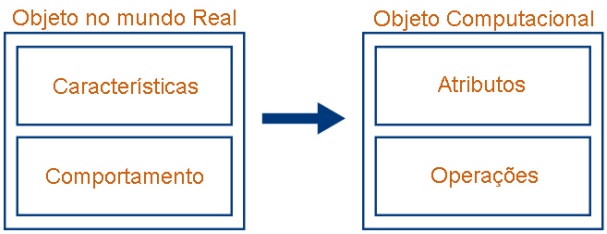
A única forma de colaboração entre os objetos é por meio das suas respectivas interfaces (FARINELLI, 2020). Utilizando o exemplo acima, podemos alterar o nome, a idade e o peso da pessoa graças a um conjunto de operações. Desse modo, essas operações normalmente alteram o estado do objeto.

Vejamos agora outros exemplos de operações:

Exemplo

Classe empresa = Contratar_Funcionario, Despedir_Funcionario;
Classe janela = Abrir, fechar, ocultar.

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



Mapeamento de objetos do mundo real.

Basicamente, deve-se analisar o mundo real e identificar quais objetos precisam 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 tais objetos.

O conceito de classe

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 possui os atributos e as operações definidos na classe.

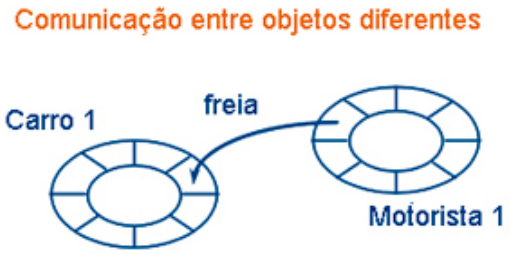
Durante a execução de um programa orientado a objetos, são instanciados os objetos a partir da classe. Assim, **um objeto é chamado de instância de sua classe**.

A classe é o bloco básico para a construção de programas orientados a objetos (OO), aponta Costa (2015).

Atenção!

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 nas informações da imagem *Atributos de objetos da pessoa 1*, que vimos anteriormente, uma classe Pessoa deve ser definida 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.



Colaboração entre motorista e carro.

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 significa a chamada de uma operação feita pelo objeto receptor da mensagem (COSTA, 2015).

De acordo com a imagem anterior, 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.

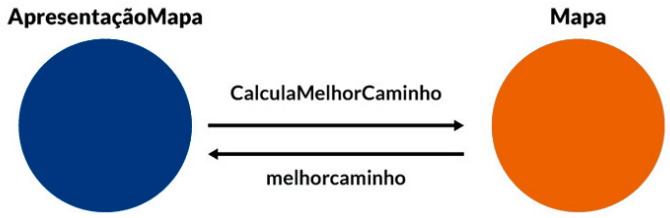
O conceito de encapsulamento

Seu conceito 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, o encapsulamento 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 a fim de 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. No entanto, o objeto não "sabe" como calcular a distância entre os dois pontos. Para resolver esse problema, ele precisa colaborar com um objeto Mapa que "saiba" calcular e, portanto, possua 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.



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. Contudo, essa mudança não afetará o objeto ApresentaçãoMapa, pois a implementação foi realizada isoladamente (encapsulamento) no objeto Mapa sem causar impacto em outros objetos no sistema.

Resumindo

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

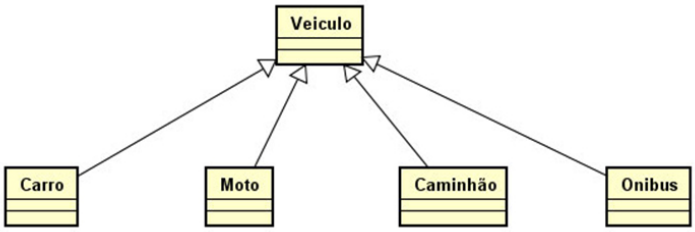
Os mecanismos de herança e polimorfismo

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 especializá-las 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 o esquema a seguir:



Exemplo de herança.

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

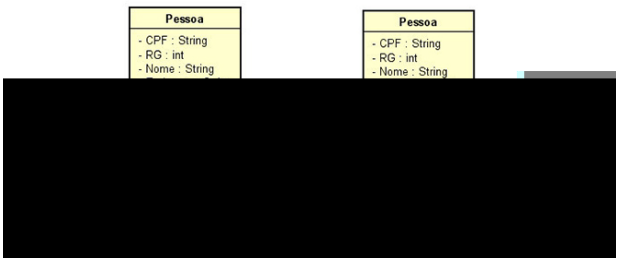
Uma classe pode ser definida genericamente como uma superclasse e, em seguida, especializada em classes mais específicas (subclasses). A herança permite a reutilização de código em larga escala, pois possibilita que se herde todo o código já implementado na classe pai e se adicione apenas o código específico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

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

Conheça a seguir os tipos de herança:

Herança simples

A herança é considerada simples quando uma classe herda as características existentes apenas de uma superclasse. A figura adiante apresenta uma superclasse Pessoa, que possui os atributos CPF, RG, Nome e Endereço. Em seguida, a classe Professor precisa herdar os atributos dessa superclasse, além de adicionar atributos específicos do contexto da classe Professor, como titularidade e salário.



Exemplo de herança Pessoa -> Professor./ Exemplo de herança Pessoa -> Professor e Pessoa -> Aluno.

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

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 outro curso de graduação na mesma instituição em que trabalha.

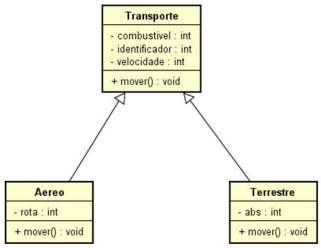
Ele, portanto, possuirá os atributos da classe Professor e os da classe Aluno. Além disso, também haverá um atributo DescontoProfessor, que será obtido apenas quando houver a associação de professor e aluno com a universidade.

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.

Exemplo de herança Pessoa -> Professor e Pessoa -> Aluno e Professor/Aluno ->ProfessorAluno.

Polimorfismo

O polimorfismo é a capacidade de haver o 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).



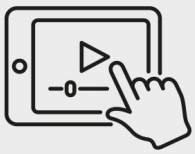
mover() – método polimórfico.

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

Conceitos de paradigma orientados a objetos

No video a seguir, apresentaremos alguns exemplos dos conceitos de orientação a objeto.

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



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Na programação orientada a objetos, há conceitos, como, por exemplo, 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 a reutilização de código por meio de uma técnica conhecida como herança.

II. A herança permite projetar classes genéricas que podem ser especializadas em classes mais particulares, etapa na qual as classes especializadas reutilizam o código das mais genéricas.

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

E Apenas I e II.

Parabéns! A alternativa C está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3EA%20quest%C3%A3o%20aborda%20tr%C3%AAs%20conceitos%20importantes%20da%20orienta%C3%A7%C3%A3o%20a%20objetos%3

Questão 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 uma:

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.
E	herança múltipla de CC em relação a CB e PP.

Parabéns! A alternativa D está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3EA%20quest%C3%A3o%20aborda%20uma%20implementa%C3%A7%C3%A3o%20pr%C3%A1tica%20de%20heran%C3%A7a%20m%C3%B

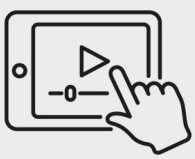
2 - Orientação a objetos na linguagem Python

Ao final deste módulo, você será capaz de descrever os conceitos básicos da programação orientada a objetos na linguagem Python.

Vamos começar!

Agregação, classes e métodos estáticos (parte 1)

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



Classes e objetos

Uma classe é uma declaração de tipo que encapsula constantes, variáveis e métodos que realizam a 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, como, por exemplo, definir no ambiente de desenvolvimento Python a classe Conta. O nome final do arquivo tem de ser Conta.py.

Devemos ressaltar que todos esses exemplos podem ser executados no terminal do Python.

Arquivo **Conta.py**

Python

```python
class Conta:
    pass
```

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

Recomendação

Você deve reproduzir e executar todos os exemplos que serão apresentados a seguir.

Construtores e self

A classe Conta foi criada, porém não foram definidos atributos e instanciados objetos para ela. Ambos são uma característica básica dos programas orientados a objetos.

Nas linguagens orientadas a objetos, para se instanciar objetos, é preciso criar os construtores da classe. Em Python, a palavra reservada **__init__()** serve para a inicialização de classes, como é possível verificar abaixo:

Python

```python
class Conta:
    def __init__(self, numero, cpf, nomeTitular, saldo):
        self.numero = numero
        self.cpf = cpf
        self.nomeTitular = nomeTitular
        self.saldo = saldo
```

Classe Conta Inicial.

Diferentemente de outras linguagens de programação orientadas a objetos, o Pyhton constrói os objetos em duas etapas. A primeira etapa é utilizada com a palavra reservada **__new__**, a qual, em seguida, executa o método **__init__**.

O **__new__** cria a instância e é utilizado para alterar as classes dinamicamente nos casos de sistemas que envolvam metaclasses e frameworks. Após **__new__** ser executado, esse método chama o **__init__** para a inicialização da classe como 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, o método **__init__** será chamado .

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 Classe Conta Inicial vista anteriormente, notamos que o código possui diversos comandos self, o que indica uma referência ao próprio objeto. Por exemplo, o comando self.numero é uma indicação de que o número é um atributo do objeto, ao qual, por sua vez, é atribuído um valor. O restante dos comandos self indica a criação dos atributos cpf, nomeTitular e saldo referentes à classe Conta.

Vamos instanciar agora o nosso objeto com o método **__init__** no emulador (clique em Executar e verifique a saída do código no console do emulador):

Exercício TUTORIAL COPIAR

Python3

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

null

null

É importante ressaltar que, em Python, não é obrigatório ter um método construtor na classe. Isso ocorrerá apenas se for necessária alguma ação na construção do objeto, como a inicialização e/ou a atribuição de valores.

Segue um exemplo de uma classe sem um método construtor:

Exercício TUTORIAL COPIAR

Python3

```
class A():
    def f(self):
        print("foo")
```

null

null

Métodos

Toda classe precisa 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:

Python

```python
def __depositar__(self,valor)
    self.saldo += valor
```

Método depositar.

No código acima, foi definido um método __depositar__, que recebe a própria instância do objeto por meio do self e de um parâmetro valor. O número passado por intermédio 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, o estado da conta foi alterado com o novo saldo de 300 reais graças à referência self.

Segue o novo código:

Python

```python
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
```

Classe com métodos depositar e sacar.

No exemplo anterior, adicionamos mais um método sacar(self, valor), do qual 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.

A conta tinha, por exemplo, um saldo de 300 reais após o primeiro depósito. Após a chamada de sacar (100), o saldo da conta será de 200 reais. Desse modo, se o método gerarextrato() for executado, o valor impresso será 200.

Clique em Executar no emulador a seguir e verifique este resultado:

Exercício
Python3

TUTORIAL COPIAR

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

null

null

Métodos com retorno

Em Python, não é obrigatório haver um comando para indicar quando o método deve ser finalizado. Porém, na orientação a objetos, é bastante comum, como é o caso da programação procedural, retornar um valor a partir da análise do estado do objeto.

Conforme exemplificaremos adiante, o saque de um valor maior do que o saldo atual do cliente não é permitido; portanto, retorna a resposta “False” para o objeto que está executando o saque.

No emulador seguinte, apresentaremos como o método ficará e um exemplo de uso desse método:

Python

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

def main():
    c1 = Conta(1,1,"Joao",0)
    c1.depositar(300)
```

```
24      saque = c1.sacar(400)
25      c1.gerar_extrato()
```

Método sacar com retorno.

> Agora questione-se: ao executar o comando c1.sacar(400), qual será o retorno?

Referências entre objetos na memória

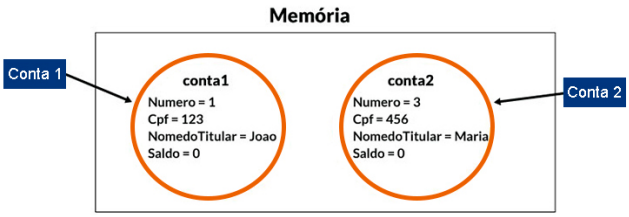
Uma classe pode ter mais de uma ou várias instâncias de objetos na memória, como demonstra esta imagem:

Python

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

Método sacar com retorno.

O resultado na memória após a execução é apresentado na figura seguinte:



Estado da memória conta1 e conta2.

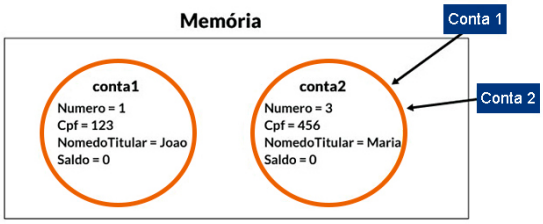
Na memória, foram criados dois objetos diferentes referenciados pelas variáveis conta1 e conta2 do tipo conta. Os operadores “==” e “!=” comparam se as duas variáveis de referência apontam para o mesmo endereço de memória (CAELUM, 2020).

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

Pelo esquema da memória apresentado na figura anterior, 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 este resultado:

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



Estado da memória conta1 e conta2 no mesmo endereço.

Se executarmos os comandos referenciando o CPF da conta, verificaremos que eles, conforme mostramos acima, possuem os mesmos valores.

>>>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 precisa ser passada como parâmetro do método para a transferência ser executada.

O método precisa ficar da seguinte maneira:

Python

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

Classe Conta com transferência.

>>>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, 1.000 reais foram depositados na conta1, enquanto foi realizada uma transferência no valor de 500 reais para a conta2. No final, o saldo ficou 500 para conta1 e 500 para conta2.

Devemos 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.

Tipos de associação entre objetos

Agregação

Para atender a novas necessidades do sistema de conta corrente do banco, agora é necessário adicionar uma funcionalidade para o gerenciamento de conta conjunta, ou seja, uma conta corrente pode ter um conjunto de clientes associados. Isso pode ser representado como uma agregação, conforme aponta o esquema a seguir. Uma observação: o losango na imagem tem a semântica da agregação.

Classe agrega 1 ou vários clientes.

Python

```python
class Cliente
    def__init__(self,cpf,nome,endereco):
        self.cpf = cpf
        self.nome = nome
        self.endereco = endereco
```

Classe cliente.py.

Python

```python
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}")
```

Classe contas.py

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

Python

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

Classe testecontas.py

Atenção!

Na linha número 5, é instanciado um objeto conta1 com dois clientes agregados:
cliente1 e cliente2. Esses dois objetos são passados como parâmetros.

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

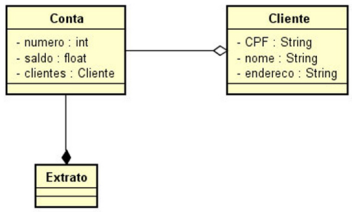
Sugestão: altere o programa do código anterior a fim de criar mais uma conta para dois clientes diferentes. Como desafio, tente, por meio do objeto conta, imprimir o nome e o endereço dos clientes associados às contas.

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 precisa 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. Essa composição representa que uma conta pode ser composta por vários extratos.

Uma observação: o losango preenchido tem a semântica da composição.



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.

Python

```python
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)}")
```

Classe extrato.py.

A classe Conta possui todas as transações, como sacar, depositar e transferir_valor. Cada transação realizada deve adicionar uma linha ao extrato da conta.

A composição Conta->Extrato inclusive precisa ser inicializada no construtor da classe Conta, conforme exemplificava a figura anterior. 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:

Python

```python
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.satetime.today()
        self.extrato = Extrato ()

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

```
14          self.extrato.trasacoes.append(["DEPOSITO", valor, "Data",datetime.datetime.today ()])
15
16      def sacar(self, valor):
17          if self.saldo < valor:
18              return False
19          else:
20              self.saldo -= valor
21              self.extrato.transacoes.append(["SAQUE", valor, "Data", datetime.datetime.today()])
22              return True
23
24      def transfereValor(self, contadestino, valor):
25          if self.saldo < valor:
26              return "Não existe saldo suficiente"
27          else:
28              contadestino.depositar(valor)
29              self.saldo -= valor
30              self.extrato.transacoes.append(["TRANSFERENCIA", valor, "Data", datetime.datetime.today()])
31              return "Transferencia Realizada"
32
33
```

Classe conta.py composta.

Adição da linha nº 10 – criação de um atributo extrato, fazendo referência a um objeto Extrato.

Adição das linhas nºs 14, 21 e 30– adição de linhas ao array de transações do objeto Extrato por meio 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
DEPÓSITO 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.

Integridade dos objetos: encapsulamento

Encapsulamento

Conforme vimos no módulo anterior, o encapsulamento é fundamental para a manutenção da integridade dos objetos e a proibição de qualquer alteração indevida nos valores dos atributos (estado) do objeto (CAELUM, 2020). Esse 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 a fim de não haver saque maior que o valor e de deixar a conta no negativo (estado inválido para o sistema).

Python

```python
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
```

Método sacar no estado inválido.

Como proibir alterações indevidas dos atributos em Python? É o que veremos a seguir.

Atributos públicos e privados

Para seguir o encapsulamento e proibir alterações indevidas dos atributos, deve-se 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 por meio de métodos do objeto.

Para tornar um atributo privado, é preciso iniciá-lo com dois underscores ('__'). E qual seria o retorno do interpretador ao se acessar um atributo privado para classe Conta? Um erro seria gerado.

Clique em Executar no emulador abaixo - no qual existe uma classe chamada Conta com todos os atributos privados - e verifique:

Exercício TUTORIAL COPIAR

Python3

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

def
```

null

null

É importante ressaltar que, em Python, não há realmente atributos privados. O interpretador renomeia o atributo privado para _nomedaClasse__nomedoatributo.

O atributo, portanto, ainda pode ser acessado. Embora ele funcione, isso é considerado uma prática que viola o princípio de encapsulamento da orientação a objetos.

>>> conta._Conta__saldo
1000

> **Na prática, deve haver uma disciplina para que os atributos como __ ou _ definido nas classes não sejam acessados diretamente.**

Decorator @property

Uma estratégia importante disponibilizada pelo Python são as properties. Utilizando o decorator property nos métodos, mantém-se os atributos como protegidos, os quais, por sua vez, são acessados apenas por meio 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:

Python

```
@property
def saldo(self):
    return self._saldo
```

Definição de uma propriedade.

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

Notação:

@< nomedometodo >.setter

Python

```
@saldo.setter
def saldo(self, saldo):
  if (self.saldo < 0):
    print("saldo inválido")
  else:
    self._saldo = saldo
```

Definição de um método setter.

Os properties ajudam a garantir o encapsulamento no Python.

Dica

Uma boa prática implementada em todas as linguagens orientadas a objetos será a de 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.

No código a seguir, demonstraremos como acessar os métodos decorados com **@property** e **@< nomedometodo >.setter** (clique em Executar):

Exercício TUTORIAL COPIAR

Python3

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

```

null

null

Atributos de classe

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

Inicialmente, a classe Círculo vai ser criada:

Python

```
class Circulo():
  def__init__(self,pontox,pontoy,raio):
    self.pontox = pontox
    self.pontoy = pontoy
    self.raio = raio
```

Classe Círculo.

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

Python

```
1  class Circulo():
2      total_circulos = 0
3
4      def__init__(self,pontox, pontoy, raio):
5          self.pontox = pontox
6          self.pontoy = pontoy
7          self.raio = raio
8          self.total_circulos +=1
```

Classe Círculo com atributo de classe.

Na linha 2, 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, que terá um valor único para todos objetos da classe.

Na linha 8, o valor da variável de classe a cada instanciação de um objeto da classe Círculo é atualizado .

Python

```
1  >> from Circulo import Circulo
2  >>>circ1 = Circulo(1,1,10)
3  >>>circ1.total_circulos
4  1
5  >>>circ2 = Circulo(2,2,20)
6  >>>circ2.total_circulos
7  2
8  >>>Circulo.total_circulos
9  2
```

Classe Círculo.

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

Como isso fica agora? O resultado será apresentado adiante.

Python

```
1  class Circulo:
2
3    _total_circulos = 0
4
5    def__init__(self,pontox, pontoy, raio):
6      self.pontox = pontox
7      self.pontoy = pontoy
8      self.raio = raio
9      circulo._total_circulos += 1
```

Classe Círculo com atributo de classe privado.

Repetindo o mesmo código:

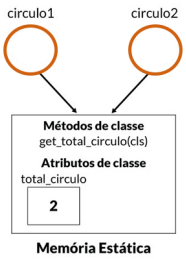
```python
>>>from Circulo import Circulo
>>>circ1 = Circulo(1,1,10)
>>>circ1._total_circulos
1
>>>Circulo.total_circulos
Traceback (most recent call last):
File '<input>', line 1, in <module>
AttributeError: type object 'Circulo' has no attribute 'total_circulos'
```

Classe Círculo com atributo de classe privado 2.

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 nesta imagem:



Memória estática.

Para definir um método como estático, deve-se usar o decorator @classmethod. Observe agora a classe Círculo alterada:

```python
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):
        return cls._total_circulos
```

Método de classe.

Na linha 11, é criado o parâmetro cls como referência para classe. Na linha 12, 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. Desse modo, o método pode ser declarado como privado, mesmo que ainda possa ser chamado diretamente como se fosse um método público.

Os dois underscores antes do método indicam que ele é privado:

Python

```
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}")
```

Método privado.

No código acima, foi definido o método **__gerarsaldo** como privado. Portanto, ele 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 haver uma referência para um objeto da classe, ou seja, não existe a obrigatoriedade da instanciação de um objeto da classe. O método pode ser chamado diretamente:

Python

```
import math
class Math:

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

Método estático.

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

No caso acima, o método sqrt da classe Math foi chamado sem que um objet da classe Math fosse instanciado.

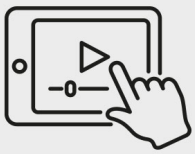
Atenção!

Os métodos estáticos não são uma boa prática na programação orientada a objetos. Eles devem ser utilizados apenas em casos especiais, como o de classes de log em sistemas.

Agregação, classes e métodos estáticos (parte 2)

No vídeo a seguir, entenderemos um pouco mais sobre agregação, classes e métodos estáticos.

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



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

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

Python

```python
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):
```

Aponte a opção **incorreta** sobre a classe acima e sobre as regras da programação orientada a objetos em Python.

A — A criação de objetos chama primeiramente 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.

E — O método __new__() é implicitamente chamado dentro do método __init__().

Parabéns! A alternativa B está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3EA%20inst%C3%A2ncia%20do%20objeto%20(self)%20deve%20ser%20passada%20implicitamente%20em%20cada%20chamada%20do%

Questão 2

Qual é a diferença na utilização dos decorators @staticmethod e @classmethod?

A — O decorator @staticmethod define métodos de classe que manipulam atributos de classe.

B — O decorator @classmethod define métodos estáticos que permitem o acesso a métodos sem instanciação da classe.

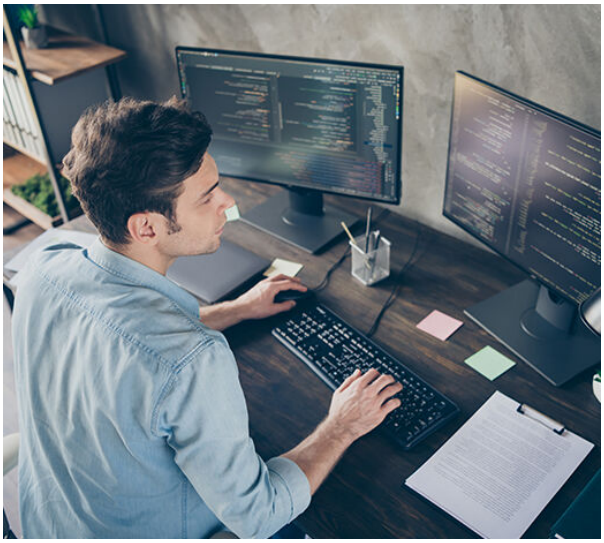
C — O decorator @classmethod permite que os atributos de classe sejam alterados na área de memória.

D — Os decorators @staticmethod e @classmethod podem ser usados de formas intercambiáveis.

E — O decorator @staticmethod permite que um método privado possa ser chamado diretamente como se fosse um método público.

Parabéns! A alternativa C está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3EO%20decorator%20%40classmethod%20fica%20armazenado%20na%20mesma%20%C3%A1rea%20de%20mem%C3%B3ria%20dos%20



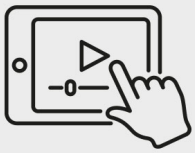
3 - Orientação a objetos como herança e polimorfismo

Ao final deste módulo, você será capaz de descrever os conceitos da orientação a objetos como herança e polimorfismo.

Vamos começar!

Herança, polimorfismo e exceções (parte 1)

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



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, o polimorfismo e o tratamento das exceções.

A herança permite uma redução da repetição de código, autorizando uma classe filho a herdar métodos e atributos da classe pai; o polimorfismo, 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.

Implementando herança

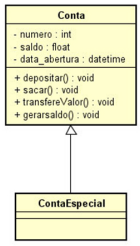
É um dos princípios mais importantes da programação orientada a objetos, pois permite a reutilização de código com a possiblidade de extensão para se ajustar às regras de negócio dos sistemas.

Exemplo

A área de produtos do banco define quais clientes podem ter acesso a um produto chamado Conta especial, o qual busca atender a 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, que é 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, a qual, por sua vez, herdará todo código de conta, conforme mostra a figura seguinte:



Herança Conta -> ContaEspecial.

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

Python

```
1  from ContasClientesExtrato import Conta
2  import datetime
3
4  class ContaEspecial(Conta):
5      def__init__(self,clientes,numero,saldo,limite):
6          Conta.__init__(self,clientes,numero,saldo)
7          self.limite = limite
8
9      def sacar(self, valor):
10         if (self.saldo + self.limite) < valor:
11             return False
12         else:
13             self.saldo -= valor
14             self.extrato.transacoes.append(["SAQUE", valor "Data", datetime.datetime.today()])
15             return True
```

Classe ContaEspecial.

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

Método construtor __init__-

A classe tem de ser instanciada com a passagem do limite como um parâmetro da construção do objeto. O método __Init__ foi sobrescrito da superclasse Conta. Já o método super(), que foi utilizado para chamar um método da superclasse, pode ser usado em qualquer método da subclasse (REAL PYTHONON, 2020). A orientação a objetos permite inclusive a reutilização do construtor da classe pai (linha 6).

Atributo limite

Adicionado apenas na subclasse ContaEspecial, em que ele será utilizado para implementar a regra de saques além do valor do saldo (linha 7).

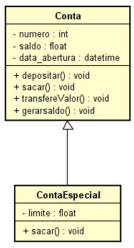
Método sacar()

O método precisa verificar se o valor a ser sacado, passado como parâmetro, é menor 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 (linha 9).

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
DEPÓSITO 100.00 Data 14/Jun/2020
```

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



Herança Conta -> ContaEspecial.

> **A 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 casos com código parecido, o reuso era bastante baixo. O programador tinha de criar um programa Conta Corrente Especial repetindo todo o código já definido no programa Conta Corrente. Com a duplicação do código, era necessário realizar a manutenção de dois códigos parecidos em dois programas diferentes.

Herança múltipla

A herança múltipla é um mecanismo que possibilita a uma classe herdar o 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.

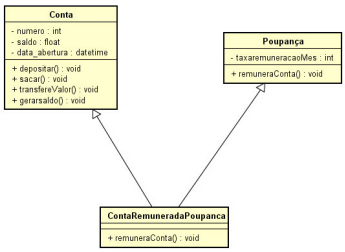
Para nosso sistema, vamos considerar a necessidade de um novo produto, que consiste em uma conta corrente similar àquela definida anteriormente no sistema, com as seguintes características:

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

Terá de 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:



Herança múltipla.

A implementação está detalhada nestes códigos:

Python

```python
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
```

Conta-poupança.

LINGUAGEM DE PROGRAMAÇÃO UTILIZADA

```
from ContasClientesExtrato import Conta
from ContaPoupanca import Poupanca

class ContaRemuneradaPoupanca(Conta, Poupanca):
    def__init__(self, taxaremuneracao, clientes, numero, saldo, taxaremuneracao):
        Conta.__init__(self,clientes,numero,saldo)
        Poupanca.__init__(self,taxaremuneracao)
        self.taxaremuneracao = taxaremuneracao

    def remuneraConta(Self):
        self.saldo += self.saldo * (self.taxaremuneracao/30)
        self.saldo -= self.taxaremuneracao
```

Eis alguns pontos importantes sobre a implementação:

Declaração de herança múltipla

A linha 4 indica que a classe é herdeira de Conta e de Poupança (nessa ordem). Tal 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). Isso pode ser visto nas linhas 6 e 7.

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
```

Implementando polimorfismo

Polimorfismo é o mecanismo que permite a um método com o mesmo nome ser executado de modo diferente a depender 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 imaginar que agora tenhamos 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, apenas o IOF. Já a conta Cliente não tem desconto nenhum.

Todos os descontos são realizados em cima do valor bruto após o rendimento mensal. Uma vez por mês, o banco executa o cálculo do rendimento de todos os tipos de contas.

O diagrama será apresentado na imagem adiante:

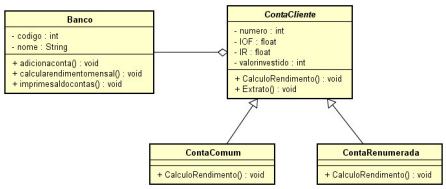


Diagrama Banco.

Vamos analisar agora a implementação das classes:

Python

```python
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



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)
```

Classe ContaCliente.

Python

```python
from ContaCliente import ContaCliente
class ContaComum(ContaCliente)
    def__init__(self,numero,IOF,IR,valorinvestido,taxarendimento):
        super().__init__(numeroIOFIR,valorinvestido,taxarendimento)

    def CalculoRendimento(Self): #(2)
        sef.valorinvestido += (self.valorinvestido * self.taxarendimento)
```

Classe ContaComum.

```python
from ContaCliente import ContaCliente
class ContaRemunerada(ContaCliente):
    def__init__(self,numero,IOF,IR,valorinvestido,taxarendimento):
        super().__init__(numeroIOFIR,valorinvestido,taxarendimento)

    def CalculoRendimento(Self): #(3)
        sef.valorinvestido += (self.valorinvestido * self.taxarendimento)
        sef.valorinvestido -= self.valorinvestido * self.IOF
```

Classe ContaCliente.

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

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

```python
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:
```

Classe Banco.

```python
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)
```

```
8  banco1.adicionaconta(contaremunerada1) #(6)
9  banco1.calcularendimentomensal#(7)
10 banco1.imprimesaldocontas() #(8)
```

Classe BancoContas.

Em (4), (5) e (6), o 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 contacomoum1 passado é uma ContaComum, que passa no teste “É-UM”, pois uma ContaComum também é uma ContaCliente.

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 de 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 verificará o teste “É-UM”, porém esse método não foi sobrescrito pelas subclasses da hierarquia ContaCliente. Portanto, será chamado o método Extrato da superclasse.

Saiba mais

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 é a instância do objeto em execução para chamar o método correto. Com base nele, 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 pode haver objetos dessa classe executados na memória.

O termo “abstrato” remete a um conceito do mundo real, considerando a necessidade 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 de clientes do tipo ContaCliente, e sim apenas dos objetos do tipo ContaComum e Conta VIP.

Diagrama de classes abstratas.

Houve apenas a adição de esteréotipo <> 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 sua definição para:

Python

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

Definição de classe abstrata.

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

O decorator @abstractmethod indica para a linguagem que o método é abstrato (STANDARD LIBRARY, 2020), o que ocorre no código adiante:

Python

```python
from abc import ABC, abstractmethod

class ContaCliente(ABC):
    def __init__(self,sumero,IOF,IR,valorinvestido,taxarendimento):
        self.numero = numero
        self.IOF =   IOF
        self.IR = IR
        self.valorinvestido = valorinvestido
        self.taxarendimento = taxarendimento

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

Definição de método abstrato.

Quando se tentar instanciar um objeto da classe, será obtido um erro indicando que essa classe não pode ser instanciada. Faça o teste! No emulador , insira a linha de código abaixo e clique em Executar.

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

Exercício TUTORIAL COPIAR

Python3

```
from abc import ABC, abstractmethod

class ContaCliente(ABC):
    def __init__(self,sumero,IOF,IR,valorinvestido,taxarendimento):
        self.numero = numero
```

null

null

Apenas as subclasses ContaComum e ContaVIP podem ser instanciadas.

Atenção!

As classes mencionadas devem obrigatoriamente implementar os métodos. Caso contrário, elas 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.

Tratamento de exceções

Como diversas linguagens orientadas a objetos, o Python permite a criação de tipos de exceções para diferenciar os erros gerados pelas bibliotecas da linguagem daqueles gerados pelas aplicações desenvolvidas. Deve-se, para isso, usar a característica da herança para herdar novas exceções a partir da classe Exception do Python (MENEZES, 2020).

Python

```
class ExcecaoCustomizada(exception):
    pass

    def throws(): (2)
        raise ExcecaoCustomizada
        try:
            throws()
        except ExcecaoCustomizada as ex:
            print ("Excecao lançada")
```

Exceção customizada.

No código acima:

Na linha 1, define-se a exceção customizada ExcecaoCustomizada com o método pass, pois ele não executa nada relevante.

Na linha 4, é definido um método que, se for chamado, criará a exceção na memória ExcecaoCustomizada.

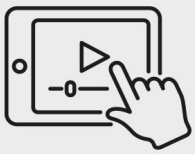
Na linha 6, é utilizado o try...except, que indica para o interpretador que a área do código localizada entre o try e o except poderá lançar exceções que deverão ser tratadas nas linhas de código após o except.

Ao final da execução, será impressa a "Exceção lançada" pela captura da exceção, a qual, por sua vez, é lançada pelo método throws().

Herança, polimorfismo e exceções (parte 2)

No vídeo a seguir, estabeleceremos uma compreensão maior sobre herança, polimorfismo e exceções.

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



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 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 por meio do polimorfismo dos métodos.

D — A separação de blocos de código em Phyton é feita utilizando a endentação de código.

E	Python não suporta o conceito de herança múltipla.

Parabéns! A alternativa C está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3EA%20quest%C3%A3o%20abrange%20uma%20vis%C3%A3o%20geral%20sobre%20os%20conceitos%20de%20orienta%C3%A7%C3%A3

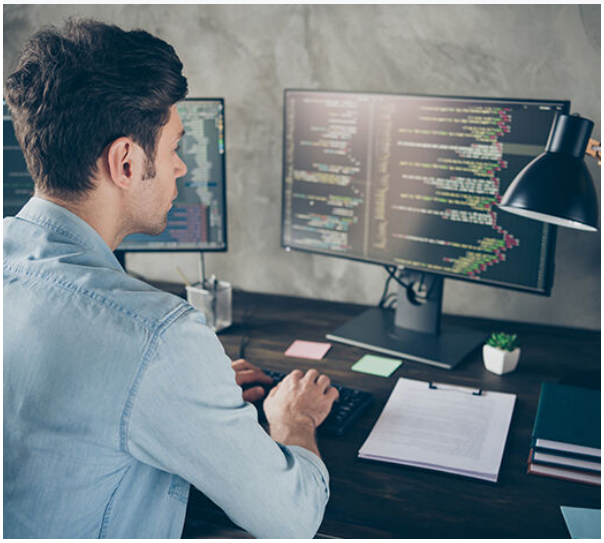
Questão 2

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 a herança múltipla por meio do poliformismo.
E	classes abstratas não podem ser instanciadas durante a execução do programa orientado a objetos.

Parabéns! A alternativa C está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3EA%20quest%C3%A3o%20abrange%20uma%20vis%C3%A3o%20geral%20a%20implementa%C3%A7%C3%A3o%20dos%20conceitos%20



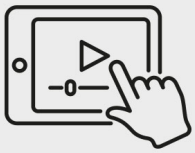
4 - Orientação a objetos aplicados a Python e outras existentes no mercado

Ao final deste módulo, você será capaz de comparar a implementação dos conceitos orientados a objetos aplicados a Python com outras linguagens orientadas a objetos existentes no mercado.

Vamos começar!

Exercícios de orientação a objetos

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



Orientação a objetos e suas linguagens de programação

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

Comparação com C++ e JAVA

Instanciação de objetos

As linguagens Java e C++ nos obrigam a utilizar a palavra reservada **new** e a indicar o tipo do objeto. No entanto, a linguagem C++ referencia todos os objetos explicitamente por intermédio 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()

Tabela: Instanciação de Objetos em Java/C++/Python.
Marcelo Nascimento Costa

Construtores de objetos

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

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)

Tabela: Construtores de Objetos em Java/C++/Python.
Marcelo Nascimento Costa

Modificadores de acesso a atributos

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

Java	C++	Python
Possui os seguintes modificadores: public private protected default	Possui os seguintes modificadores: public private	Possui os seguintes modificadores: público privado – iniciado com "_" Modificador privado pode ser burlado e acessado diretamente

Tabela: Modificadores de acesso em atributos em Java/C++/Python.
Marcelo Nascimento Costa

Herança múltipla de classes

As linguagens C++ e Python implementam a herança múltipla diretamente por meio de classes, enquanto Java implementa graças à herança múltipla de interfaces.

Java	C++	Python
Não implementa. Utiliza herança múltipla de interfaces para substituir essa característica.	Implementa com a referência das classes herdadas na declaração da classe: Class ContaRemunerada: public Conta, public Poupanca)	Implementa com a referência das classes herdadas na declaração da classe: class ContaRemuneradaPoupanca(Conta, Poupanca):

Tabela: Herança Múltipla em Java/C++/Python.
Marcelo Nascimento Costa

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

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

Tabela: Classes informais em Java/C++/Python.
Marcelo Nascimento Costa

Tipos primitivos

As linguagens Java e C++ possuem um conjunto parecido de tipos primitivos, enquanto, em Python, todas 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

Tabela: Tipos Primitivos em Java/C++/Python.
Marcelo Nascimento Costa

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

Tabela: Interfaces em Java/C++/Python.
Marcelo Nascimento Costa

Sobrecarga de métodos

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

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

Tabela: Sobrecarga de métodos em Java/C++/Python.
Marcelo Nascimento Costa

Tabela comparativa

A próxima tabela 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 e baseados em design patterns.

Portanto, Java e C++ levam uma vantagem considerável na construção de sistemas puramente orientados a objetos.

Por sua simplicidade, o Python vem crescendo como a primeira linguagem de programação ensinada na graduação. Devido à quantidade de bibliotecas estatísticas e de frameworks web existentes, ela 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(*)

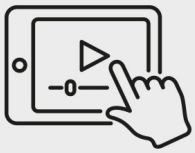
Características	Linguagens		
	Java	C++	Python
Modificadores de acesso métodos	X	X	X
Herança múltipla		X	X
Classes informais		X	X
Tipos primitivos	X	X	
Interfaces	X	X	
Sobrecarga de métodos	X	X	

Tabela: Comparativa - (*) Pode ser burlado e ser acessado diretamente
Marcelo Nascimento Costa

Comparação entre as linguagens C++, Java e Python

No vídeo a seguir, faremos uma breve contextualização sobre o assunto abordado.

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



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Em Python, como se pode trabalhar sempre com tipos objetos?

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

B obj(8).__.

C 5.__add__(3).

D sub(5).__add__(3).

E add_(5,3).

Parabéns! A alternativa C está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3ENa%20alternativa%20C%2C%20%C3%A9%20realizada%20a%20soma%20seguindo%20a%20nota%20Python.%20A%20quest%C3%A3o%

Questão 2

Analisando as características da orientada a objetos de C++, Java e Python, considere as seguintes afirmações e, em seguida, escolha a(s) alternativa(s) correta(s):

I. Em Java, é permitida a criação da herança múltipla de classes por meio das classes denominadas interfaces.

II. Em Python, assim 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.

B II e III.

C I e II.

D III.

E I.

Parabéns! A alternativa D está correta.

%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3Cp%20class%3D'c-paragraph'%3EPython%20n%C3%A3o%20possui%20tipos%20privados.%20Os%20tipos%20privados%20que%20garantem%20o%20encapsulamento%

Considerações finais

Vimos neste conteúdo que o paradigma orientado a objetos é largamente utilizado no mercado devido à 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. Desse modo, ele vem ganhando mercado tanto na área acadêmica quanto na comercial para o desenvolvimento de sistemas orientado a objetos.

Outro ponto importante que destacamos é que, devido à sua simplicidade, o Python vem sendo utilizada como a primeira linguagem a ser aprendida nos primeiros períodos dos cursos de Tecnologia de Engenharia - até mesmo para fixar os conceitos da orientação a objetos. A quantidade de bibliotecas opensource disponíveis para a implementação de algoritmos de Data Science tornou essa linguagem importante para a implementação de sistemas de aprendizado e visualização de dados.

Faremos neste podcast um resumo geral do conteúdo:

Para ouvir o *áudio*, acesse a versão online deste conteúdo.

Referências

ANAYA, M. **Clean code in Python:** refactor your legacy code base. Packt Publishing. Publicado em: 29 ago. 2018.

CAELUM. **Python e orientação a objetos.** Consultado na internet em: 14 jun. 2020.

COSTA, M. **Análise orientada a objetos.** Notas de aula. mar. 2017.

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

GIRIDHAR, C. **Learning Python design patterns.** Packit Publishing, 2018.

JAVA. **Java tutorial.** Consultado na internet em: 14 jun. 2020.

MENEZES, N. C. **Introdução à 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 na internet em: 14 jun. 2020.

QCONCURSOS. **Questões de concursos.** Consultado na internet em: 18 jun. 2020.

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

RAMALHO, L. **Introdução à orientação a objetos em Python (sem sotaque).** Consultado na internet em: 14 jun. 2020.

REAL PYTHON. **Blog Python.** Consultado na internet 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 na internet em: 24 jun. 2020.

TIO-BE. **TIO-BE Index.** Consultado ena internet em: 14 jun. 2020.

TUTORIALSPOINT. **C++ tutorial.** Consultado na internet em: 14 jun. 2020.

Explore +

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, leia 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, consulte a obra *Practical techniques for Python developers and testers*, de Ashwin Pajankar, lançada em 2017 pela Apress.