



Python orientado a objetos

Prof. Marcelo Nascimento Costa

Prof. Kleber de Aguiar

Apresentação

Você estudará como implementar os conceitos de orientação a objetos em Python. O paradigma de programação orientada a objetos é largamente utilizado para o desenvolvimento de software. Essa proximidade facilita a manutenção dos softwares orientados a objetos pois a sua implementação se aproxima dos conceitos do mundo real.

Propósito

Preparação

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.



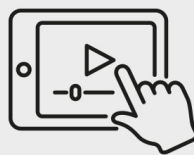
Introdução

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.

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





Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.



Download material



1 - Orientação a objetos

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

Conceitos e pilares de programação orientada a objetos

A orientação a objetos (OO) é um paradigma essencial na programação moderna, centrado na organização de software em torno de objetos, que são instâncias de classes. Esse método oferece uma abordagem mais intuitiva e eficiente para resolver problemas complexos. Neste conteúdo, exploraremos os pilares fundamentais da orientação a objetos: objetos, atributos e operações.

No vídeo a seguir, apresentaremos os conceitos e pilares fundamentais da orientação a objetos (POO): objetos, atributos e operações. Acompanhe!

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



Conceitos de programação orientada a objetos (POO)

Pilares da orientação a objetos

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 mundo 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.

A abordagem baseada em objetos 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).

Observe um diferencial entre a programação orientada a objetos (POO) e a programação convencional.

POO

Programação
convencional

O software é organizado como uma coleção de objetos separados que incorpora a estrutura e o comportamento dos dados.



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.

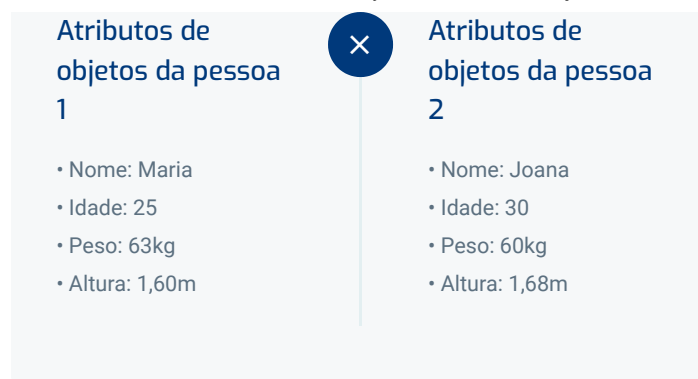
As características de uma pessoa para um sistema acadêmico podem ser a formação ou o nome do pai e o da mãe, enquanto as de um indivíduo, para o sistema de controle de uma academia, são a altura e o peso.

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 diferença entre os atributos de duas mulheres.





Operações

Uma operação é uma função ou transformação que pode ser aplicada a objetos ou dados pertencentes a um objeto.

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 dos atributos das duas mulheres, 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.

Veja agora outros exemplos de operações. Arraste as palavras que correspondem às classes apresentadas.



Repare que as operações Contratar_Funcionario, Despedir_Funcionario e Ferias_Funcionario são referentes à Classe Empresa, assim como Abrir, Fechar e Ocultar são referentes à Classe Janela.

O desenvolvimento de um sistema orientado a objetos consiste em realizar um mapeamento da seguinte maneira.



- Características
- Comportamento



- Atributos
- Operações

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.

Atividade 1

A programação orientada a objetos (OO) é um paradigma que organiza o código a partir de alguns pilares básicos: objetos, atributos (dados) e métodos (operações), que definem seu estado e comportamento. Em programação orientada a objetos (OO), qual das seguintes afirmações é verdadeira sobre objetos, atributos e operações?

A

Um objeto é uma definição de classe que especifica atributos e métodos.

B

Atributos são funções que definem o comportamento de um objeto.

C

Operações, ou métodos, são os dados armazenados em um objeto.

D

Um objeto é uma instância de uma classe, contendo atributos e métodos.

E

Atributos e métodos são sinônimos em programação orientada a objetos.

Parabéns! A alternativa D está correta.

Em programação orientada a objetos, uma classe é como um molde que define atributos (dados) e métodos (operações ou funções). Um objeto, por outro lado, é uma instância de uma classe, significando que ele é um exemplar específico dessa classe com valores concretos para os atributos e a capacidade de executar os métodos definidos na classe. As outras opções apresentam conceitos incorretos ou confusos: letra A descreve uma classe e não um objeto; letra B

descreve métodos como se fossem atributos; letra C confunde métodos com atributos; letra E iguala atributos e métodos, que são distintos.

O conceito de classe

A classe é o alicerce da programação orientada a objetos, funcionando como um molde para criar objetos. Ela define atributos (dados) e métodos (comportamentos) que os objetos derivados dela possuirão. Compreender o conceito de classe é essencial para aproveitar todo o potencial da orientação a objetos em seus projetos de software.

Neste vídeo, falaremos sobre o conceito de classe, que descreve as características e os comportamentos de um conjunto de objetos. Confira!

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



A classe descreve as características e os comportamentos 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).

Comentário

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**, 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. Observe na imagem a comunicação entre **objetos diferentes**.

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, 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.

Atividade 2

Uma classe é um componente central na programação orientada a objetos (OO).

Avalie sua compreensão desse conceito fundamental respondendo à questão a seguir.

Qual das seguintes afirmações é verdadeira sobre classes na programação orientada a objetos?

A

Uma classe é uma instância de um objeto que contém atributos e métodos.

B

Classes não podem conter métodos, apenas atributos.

C

Uma classe define atributos e métodos, servindo como um molde para criar objetos.

D

A classe é usada apenas para agrupar funções sem nenhum dado associado.

E

Objetos e classes são exatamente a mesma coisa em programação orientada a objetos.

Parabéns! A alternativa C está correta.

Na programação orientada a objetos, uma classe é um blueprint que define atributos (dados) e métodos (comportamentos) que seus

objetos (instâncias) terão. As outras opções são incorretas. Letra A descreve um objeto e não uma classe; letra b está errada porque classes podem conter métodos; letra D está errada porque classes agrupam tanto dados quanto funções; letra E confunde objetos e classes, que são conceitos distintos.

O conceito de encapsulamento

O encapsulamento refere-se à ideia de ocultar a implementação interna de uma classe ou objeto e fornecer apenas uma interface externa para interagir com ele. Em Python, o encapsulamento é alcançado por meio do uso de modificadores de acesso, como `public`, `protected` e `private`. Compreender esse conceito é crucial para desenvolver sistemas robustos e escaláveis, além de facilitar a manutenção e a reutilização de código.

Neste vídeo, apresentaremos o conceito de encapsulamento, que consiste na separação dos aspectos externos de um objeto acessíveis a outros objetos, além de seus detalhes internos de implementação, que ficam ocultos dos demais objetos. Acompanhe!

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



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, 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.

Observe o exemplo a seguir.

Imagine que 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 essa distância. Para resolver o problema, ele precisa colaborar com um objeto `Mapa` que “saiba” calcular e, portanto, possua essa responsabilidade.

Pessoa acessando aplicativo de entrega pelo celular.

O objeto Mapa implementa essa responsabilidade por meio da operação **CalculaMelhorCaminho**, 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.

A imagem a seguir ilustra o encapsulamento da classe Mapa.

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.

Atividade 3

Encapsulamento é um conceito fundamental na programação orientada a objetos (OO), promovendo a segurança, a modularidade e a manutenibilidade do código. O que o encapsulamento representa na programação orientada a objetos?

A

Um mecanismo para definir métodos privados em uma classe.

B

A exposição pública de todos os atributos de uma classe.

C

O acesso direto e ilimitado aos dados de uma classe por qualquer parte do programa.

D

Uma prática para proteger os dados internos de uma classe e permitir acesso controlado por meio de métodos.

E

A restrição completa do acesso aos dados de uma classe, tornando-os inacessíveis de fora da classe.

Parabéns! A alternativa D está correta.

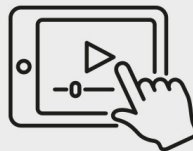
Encapsulamento em OO é o princípio de proteger os dados de uma classe, tornando-os acessíveis apenas por meio de métodos específicos (métodos de acesso). Isso ajuda a controlar como os dados são manipulados e evita que sejam alterados de forma inesperada, promovendo a segurança e a consistência do código. As outras opções estão incorretas. Letra A: o encapsulamento não se refere apenas a métodos privados, mas sim a todo o conceito de proteção de dados; letra B: encapsulamento não implica necessariamente a exposição pública de todos os atributos; letra C: encapsulamento restringe o acesso aos dados, não permitindo acesso direto e ilimitado; letra E: embora o encapsulamento restrinja o acesso aos dados, não os torna completamente inacessíveis; ainda é possível acessá-los por meio de métodos específicos.

O conceito de herança e polimorfismo

Herança permite que classes derivadas adquiram atributos e métodos de uma classe base, promovendo reutilização e organização de código. Polimorfismo possibilita que objetos de diferentes classes sejam tratados de forma uniforme, adaptando comportamentos específicos. Juntos, esses conceitos fortalecem a flexibilidade e a extensibilidade na programação orientada a objetos.

Neste vídeo, apresentaremos o conceito de herança, que permite a uma classe derivar características de outra, facilitando a reutilização de código e do polimorfismo, que permite métodos com o mesmo nome funcionarem de maneira diferente, dependendo do contexto, aumentando a flexibilidade. Confira!

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



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.

Comentário

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. Observe!

Exemplo de herança.

No esquema, 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).

Comentário

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.

A seguir, detalharemos os tipos de herança.

Herança simples

A herança é considerada simples quando uma classe herda as características existentes **apenas de uma superclasse**. A imagem a seguir 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. Observe!

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). Veja!

Comportamento mover() – método polimórfico.

Como exemplificado na imagem, 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.

Atividade 4

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.

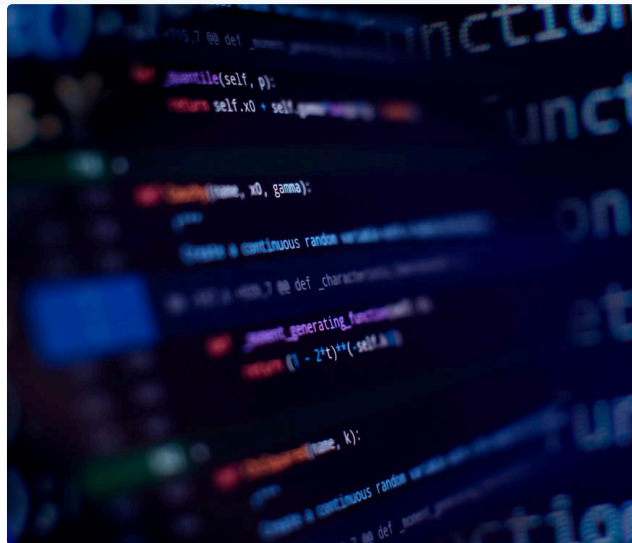
C Apenas I, II e III.

D Apenas II, III e IV.

E Apenas I e II.

Parabéns! A alternativa C está correta.

A questão aborda três conceitos importantes da orientação a objetos: herança, polimorfismo e encapsulamento. O item IV está errado devido à troca com o conceito de encapsulamento visto na seção “Encapsulamento”.



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.

Classes e objetos

Em Python, classes e objetos são fundamentais para a programação orientada a objetos. Classes definem a estrutura e o comportamento dos objetos, enquanto objetos são instâncias dessas classes. Este conteúdo abordará como criar e utilizar classes e objetos em Python, proporcionando uma base sólida para desenvolver aplicações robustas e eficientes. Prepare-se para explorar esses conceitos essenciais!

Neste vídeo, apresentaremos os conceitos de classes e objetos, abordando construtores e self, métodos, métodos com retorno e referências entre objetos na memória. Confira!

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



Definição de classe

Uma classe é uma declaração de tipo que encapsula constantes, variáveis e métodos que realizam a manipulação dos valores dessas variáveis.

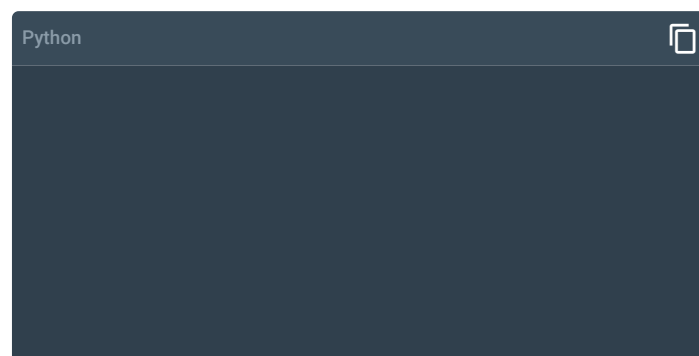
Atenção!

Cada classe deve ser única em um sistema orientado a objetos.

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.

Veja o arquivo **Conta.py**.



Exemplo de criação de classe.

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.

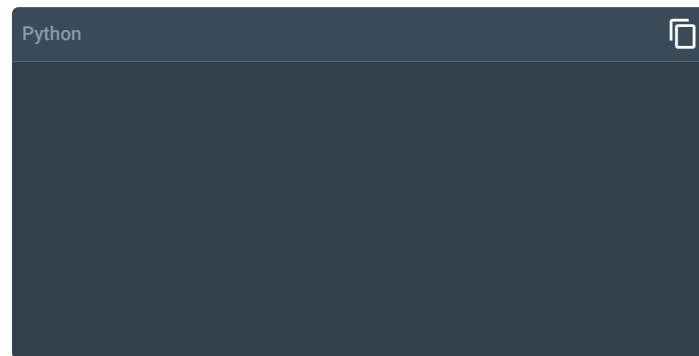
Você deve reproduzir e executar todos os exemplos que serão apresentados a seguir.

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 a seguir:



Classe Conta

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__`, 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 com seus valores iniciais (Menezes, 2019).

Comentário

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 exemplo, 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`, 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 `numero` 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`.

Agora, vamos instanciar o nosso objeto com o método `__init__` no emulador (clique em Executar e verifique a saída do código no console do emulador).





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.

Veja a seguir um exemplo de uma classe sem um método construtor.

Exercício

TUTORIAL

COPIAR

Python3

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. Veja!

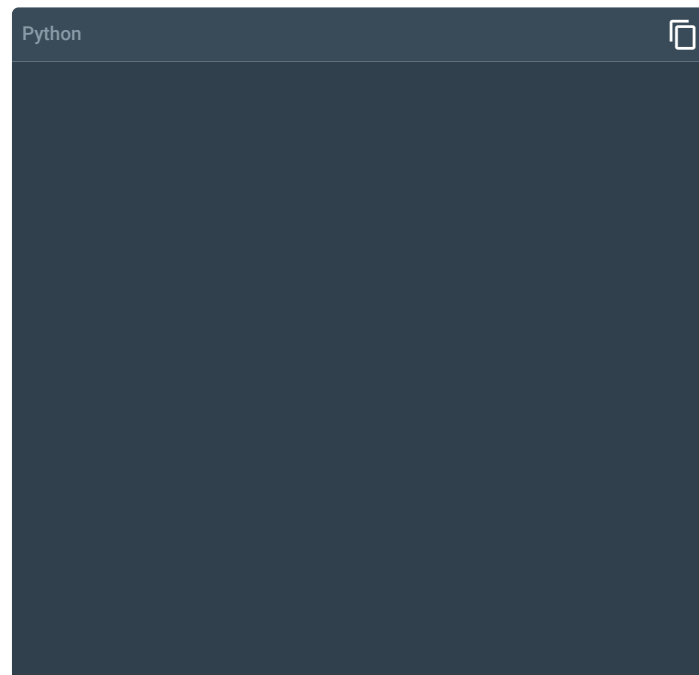
Python

Método depositar.

No código, 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.

A seguir, observe o novo código.

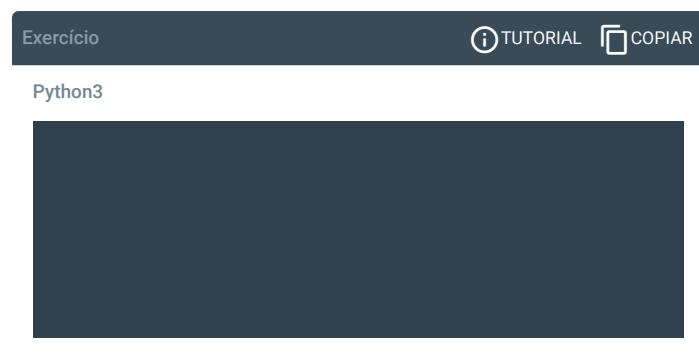


Classe Conta com métodos depositar e sacar.

No exemplo, 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 gerar_extrato() for executado, o valor impresso será 200.

Clique em Executar no emulador e verifique este resultado.



null



null

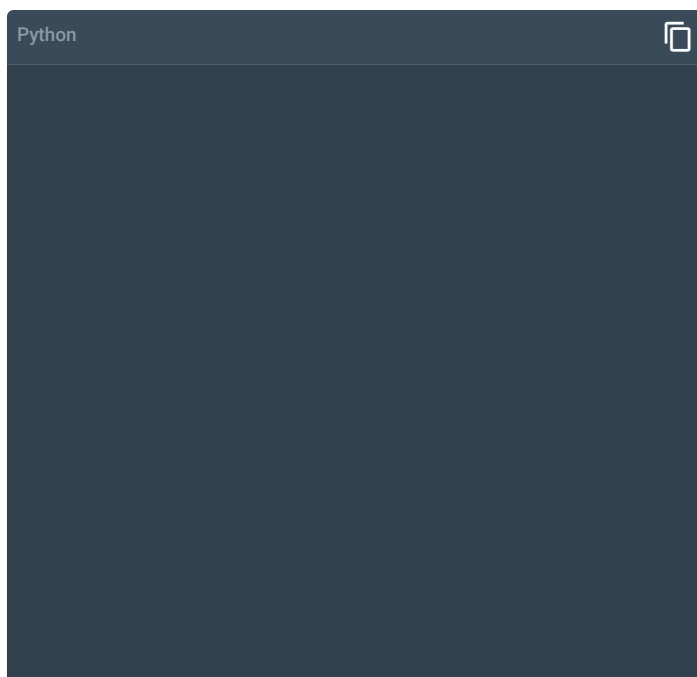


Métodos com retorno

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 a seguir, 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 código a seguir, apresentaremos como ficará o método e um exemplo de uso. Acompanhe!



Método sacar com retorno.

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

O retorno do comando `c1.sacar(400)`, seria



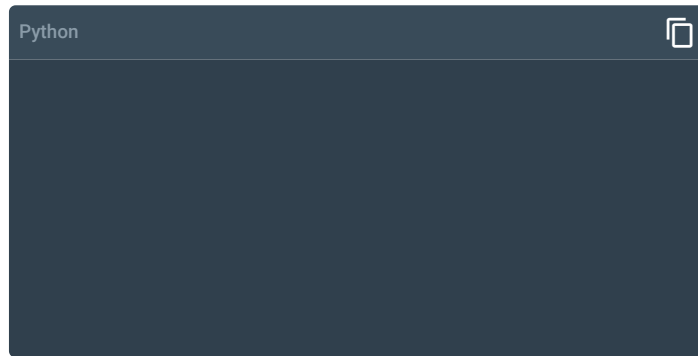
Clique em um dos botões para responder.

TRUE

FALSE

Referências entre objetos na memória

Uma classe pode ter mais de uma ou várias instâncias de objetos na memória, como demonstra o código a seguir.



Criação de dois objetos.

O resultado na **memória** após a execução é apresentado na imagem.

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 imagem, 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")
```

Observe na imagem.

Estado da memória `conta1` e `conta2` no mesmo endereço.

Se executarmos os comandos referenciando o CPF da conta, verificaremos que eles 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 deve ser apresentado como mostraremos a seguir.





Classe Conta com transferência.

Observe!

```
>>>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.

Atividade 1

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





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

A

A criação de objetos chama primeiramente o método `__ini__()` e, em seguida, o método `__new__()`.

B

A palavra "self" deve ser fornecida como argumento em todos os métodos da classe.

C

A variável `num_contrato` é encapsulada e individual para cada instância de classe.

D

Os dois sinais de underscore no método `__del__()` indica que ele se trata de um método sem retorno.

E

O construtor `__init__()` é um método obrigatório na classe.

Parabéns! A alternativa B está correta.

A palavra "self" necessita ser declarada como primeiro argumento em todos os métodos de instâncias de uma classe, pois indica uma referência a um objeto da classe. Quando um método da classe é chamado, a referência do objeto que o invocou é passada ao argumento `self` do método em questão e então o interpretador do Python é capaz de identificar exatamente qual dos objetos instanciados da classe está executando o método que foi chamado/invocado.

Classes e objetos na prática

Agora que vimos como funciona a criação de classe e a instanciação de objeto em Python, vamos fazer um exercício.

Crie uma classe chamada televisão que siga os requisitos a seguir.

- Receba como parâmetros em seu construtor o canal inicial, o maior canal e o menor canal.
- Possua como atributos o canal sintonizado, o número do maior canal e o número do menor canal.
- Possua dois métodos, um para diminuir o canal atual e outro para aumentar o canal sintonizado.
- Instancie uma tv e teste as trocas de canal.

O vídeo a seguir apresenta um exercício prático de criação de classes e instanciação de objetos em Python. Acompanhe!

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



Roteiro de prática

Execute as seguintes etapas para resolução do pedido.

- Crie a classe Televisão.
- Crie o método construtor.
- Estabeleça os parâmetros do método construtor.
- Crie os atributos canal, canal mínimo e canal máximo.
- Defina os métodos para ir para o canal superior e para o canal inferior.
- Crie dois objetos de televisão e teste a subida e a descida dos canais.

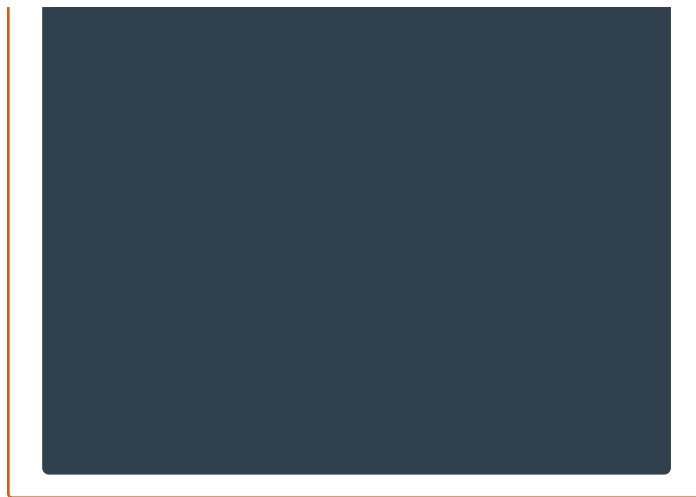
Confira, agora, o código fonte gerado.

Código fonte



Python





Atividade 2

Modifique a classe Televisão criada a partir do roteiro de prática de forma que, se pedirmos para mudar o canal para baixo, além do mínimo, ela vá para o canal máximo. Se mudarmos para cima, além do canal máximo, que volte ao canal mínimo.

Exercício

TUTORIAL

COPIAR

Python3

null

null



Chave de resposta ▾

Veja a seguir o código da solução da atividade.

Python



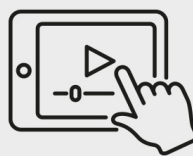


Tipos de associação entre objetos

A associação entre objetos em Python permite que instâncias de diferentes classes colaborem entre si. Esse conceito é essencial para modelar relações complexas no mundo real, como um aluno e suas disciplinas. Neste conteúdo, vamos explorar como implementar associações, tornando seu código mais modular e interconectado, refletindo de maneira precisa as interações entre diversos objetos.

Neste vídeo, falaremos sobre agregação e composição. Primeiramente, explicaremos a agregação, uma relação em que uma classe contém outra sem posse total, permitindo independência. Em seguida, abordaremos a composição, uma relação mais forte em que uma classe possui outra, havendo uma classe contenedora. Acompanhe!

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



Agregação

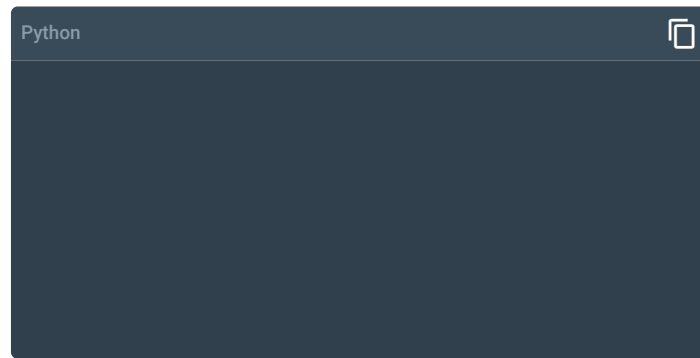
Para atender a novas necessidades do sistema de conta corrente do banco, agora é necessário adicionar uma funcionalidade para o gerenciamento de conta conjunta, ou seja, uma conta corrente pode ter um conjunto de clientes associados. Isso pode ser representado como uma agregação, conforme aponta o esquema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

Observe que o losango vazado na imagem tem a semântica da agregação.

Classe agrega 1 ou vários clientes.

Observe o código da classe Cliente.

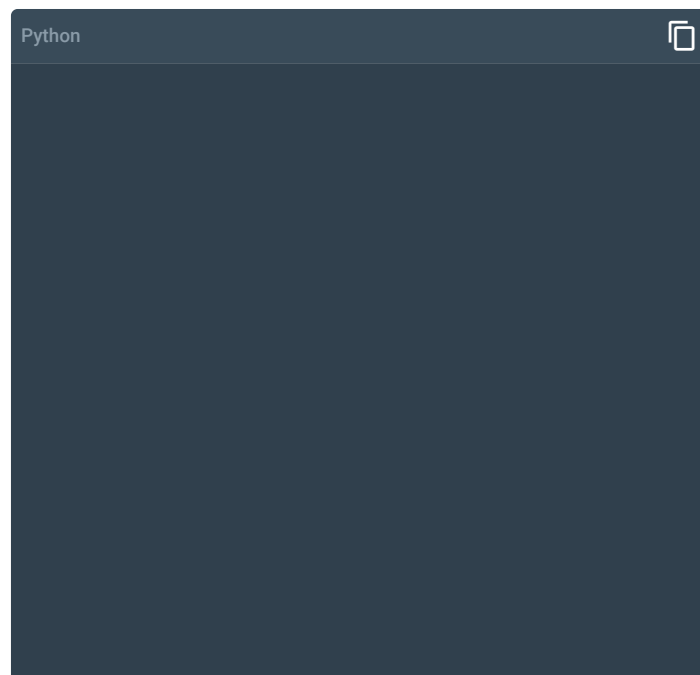
```
Python
```



Classe cliente.

Observe o código da classe Conta.

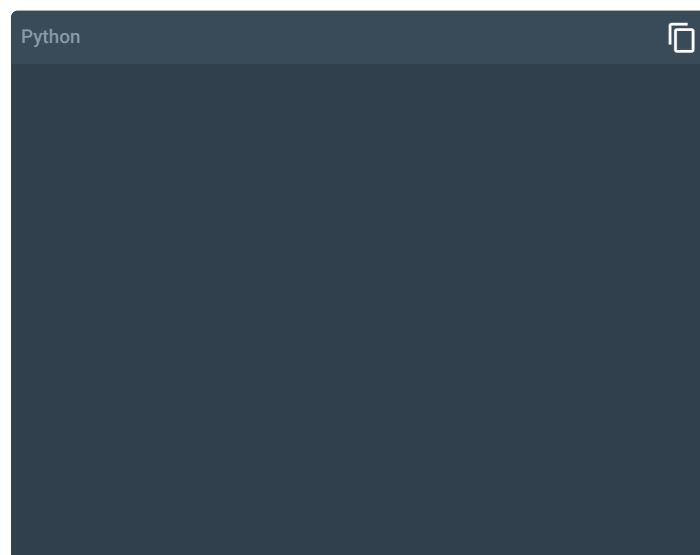
```
Python
```



Classe cliente.py.

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

```
Python
```





Programa testecontas.py.

Na linha número 7, 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?

Altere o programa do código 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

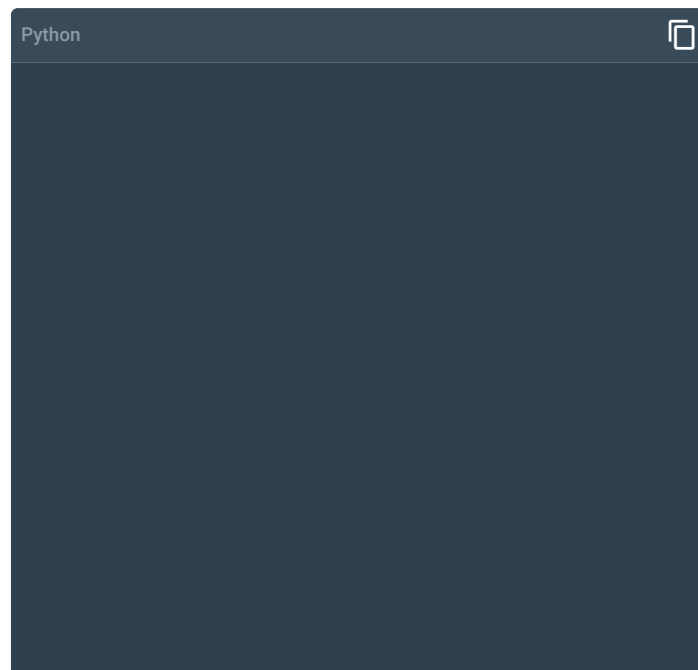
A classe Conta ainda não está completa de acordo com as necessidades do sistema de conta corrente do banco. Isso ocorre porque o banco precisa gerar extratos contendo o histórico de todas as operações realizadas para conta corrente.

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.

Observe que 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.

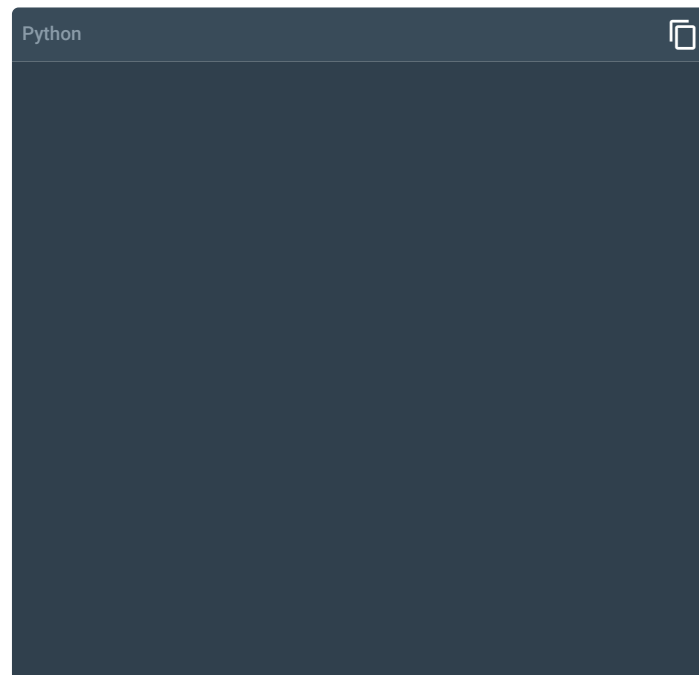


Classe extrato.

A classe Conta possui todas as transações, como sacar, depositar e transfereValor. 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 exemplificou a imagem. 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 maneira apresentada a seguir.



Classe Conta.

Observe o que foi realizado.

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.

Vamos criar um outro programa testecontas2.py e colocar os seguintes comandos.





Observação: você também pode executar esses comandos no terminal.

As duas primeiras linhas importam as definições das classes necessárias para criar clientes e contas.

As próximas duas linhas criam instâncias da classe Cliente, cada uma com um CPF, nome e endereço específicos. Dois clientes são criados: o cliente1 Joao com CPF "123" e o cliente2 Maria com CPF "456".

A linha seguinte cria uma instância da classe Conta, que associa os clientes criados anteriormente a uma conta bancária específica com um saldo inicial. Uma conta (conta1) é criada com o número 1, associada aos clientes Joao e Maria, com um saldo inicial de 2000.

As linhas seguintes realizam um depósito e um saque na conta criada, atualizando o saldo e registrando as transações no extrato. Um depósito de 1000 é realizado na conta, aumentando o saldo para 3000. A transação de depósito é registrada no extrato.

Depois, um saque de 1500 é realizado na conta, diminuindo o saldo para 1500. A transação de saque é registrada no extrato.

A última linha chama o método para exibir o extrato da conta, mostrando todas as transações realizadas até o momento.

O programa terá a seguinte saída

DEPOSITO 1000.00 Data 08/Aug/24

SAQUE 1500.00 Data 08/Aug/24

Atividade 3

Em Python, os conceitos de agregação e composição são fundamentais para a modelagem de relacionamentos entre objetos. Compreender as diferenças entre esses dois tipos de relacionamentos é imprescindível para o desenvolvimento de sistemas robustos e flexíveis. Qual das seguintes afirmações melhor descreve a diferença entre agregação e composição em Python?

A

Na agregação em Python, um objeto é composto por outros objetos e não pode existir independentemente deles, enquanto na composição, os objetos são independentes entre si.

B

Agregação e composição são conceitos idênticos em Python e podem ser usados de forma intercambiável.

C

Na composição em Python, um objeto é parte integrante de outro objeto, enquanto na agregação, os objetos são independentes entre si.

D

Agregação em Python é um relacionamento mais forte do que composição, no qual um objeto pode existir independentemente de outro objeto.

E

Na agregação em Python, um objeto tem outro objeto como parte de si mesmo, enquanto, na composição, os objetos são independentes entre si.

Parabéns! A alternativa C está correta.

Em Python, os conceitos de agregação e composição são fundamentais para a modelagem de relacionamentos entre objetos, sendo crucial entender suas diferenças para o desenvolvimento de sistemas robustos e flexíveis. Na composição, um objeto é parte integrante de outro objeto, estabelecendo uma relação forte onde o ciclo de vida do objeto componente depende do objeto composto. Se o objeto composto for destruído, os objetos componentes também serão destruídos, como no caso de um carro e seu motor, onde o motor não existe sem o carro. Em contraste, a agregação estabelece uma relação mais fraca, onde um objeto contém outros objetos que podem existir independentemente do objeto agregador. Se o objeto agregador for destruído, os objetos agregados continuam a existir, exemplificado por uma sala de aula e seus alunos, que podem existir fora da sala e serem associados a diferentes salas. Assim, na agregação, os objetos são independentes entre si, enquanto na composição, os objetos têm uma relação de dependência.

Integridade dos objetos: encapsulamento

Encapsulamento é uma prática fundamental na programação orientada a objetos, que visa proteger os dados internos de uma classe e expor apenas o necessário. Este conteúdo abordará como implementar encapsulamento usando modificadores de acesso, como públicos, protegidos e privados, ajudando a criar código mais seguro e modular. Explore as técnicas para manter seus dados seguros e bem organizados.

Neste vídeo, detalharemos a implementação do encapsulamento, que protege a integridade dos objetos ocultando seus dados internos e permitindo acesso controlado através de métodos públicos. Acompanhe!

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



Encapsulamento

Conforme aprendemos, 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.

Seguindo em nosso 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).

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

Atributos públicos e privados

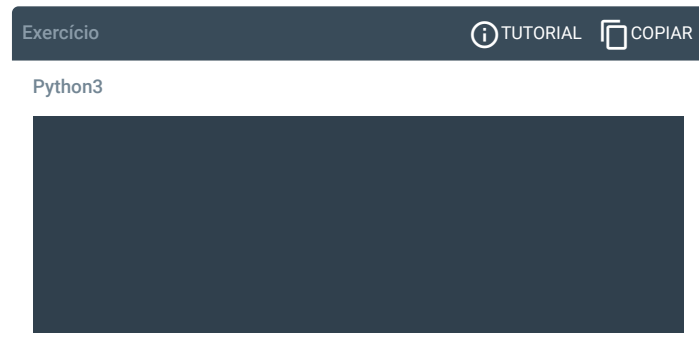
Para seguir o encapsulamento e proibir alterações indevidas dos atributos, deve-se definir atributos privados para a classe.

Comentário

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, no qual existe uma classe chamada Conta com todos os atributos privados, e verifique.



O erro gerado indica que não existe o atributo saldo. Mas como não existe se ele está na definição da classe? Ele realmente existe, mas não está visível para o programa porque o atributo é privado.

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. Veja.

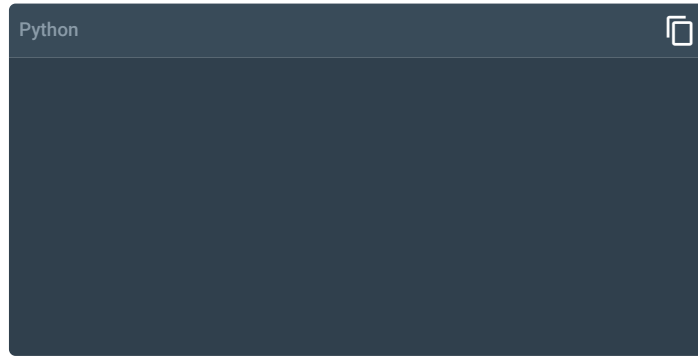
```
>>> conta._Conta__saldo
1000
```

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

Decorator @property

As **properties** são uma estratégia importante disponibilizada pelo Python para acessar e modificar atributos de forma controlada. Ao usar o decorador `@property` em métodos, você mantém os atributos como protegidos, permitindo que sejam acessados apenas por meio desses métodos decorados. Isso garante que os atributos sejam manipulados de maneira segura e conforme as regras definidas na classe.

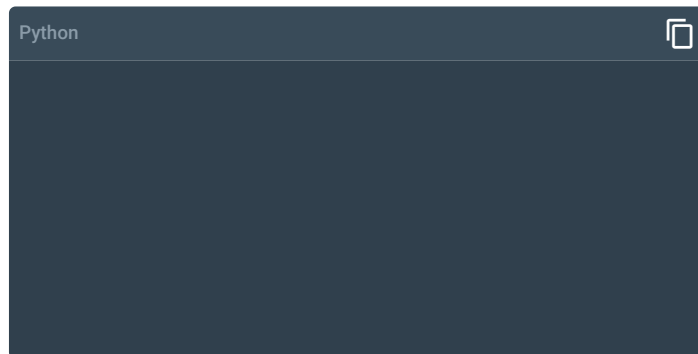
Por exemplo, na classe **Conta**, o atributo privado **saldo** não pode ser acessado diretamente para leitura. Em vez disso, ele é acessado por meio de um método decorado com `@property`, veja!



Definição de uma propriedade.

Esse método permite acessar o valor do atributo **saldo** de forma controlada. Quando você deseja permitir que o atributo **saldo** seja modificado, pode utilizar o decorador `@<nomedometodo>.setter`.

Isso força que qualquer alteração no valor do atributo privado passe por esse método, onde é possível incluir validações ou outras regras de negócios.



Definição de um método setter.

Os *properties* ajudam a garantir o encapsulamento no Python.

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.

Nesse caso, ao tentar modificar o **saldo**, o método verifica se o valor é negativo e, se for, impede a alteração e exibe uma mensagem de erro. Caso contrário, o **saldo** é atualizado.

O uso de *properties* ajuda a garantir o encapsulamento em Python, permitindo que você controle como os atributos são acessados e modificados. No entanto, é uma boa prática definir esses métodos somente se houver uma regra de negócios específica associada ao atributo. Se não houver, o acesso aos atributos pode ser feito diretamente, conforme definido na classe.

No código a seguir, demonstraremos como acessar e modificar os atributos usando os métodos decorados com `@property` e `@<nomedometodo>.setter`. Clique em Executar para ver como funciona na prática. Depois, altere o valor do saldo para um valor negativo e veja o que acontece.

Exercício

TUTORIAL

COPIAR

Python3

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 como mostra o código a seguir.

Python

Classe `Círculo`.

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

Python

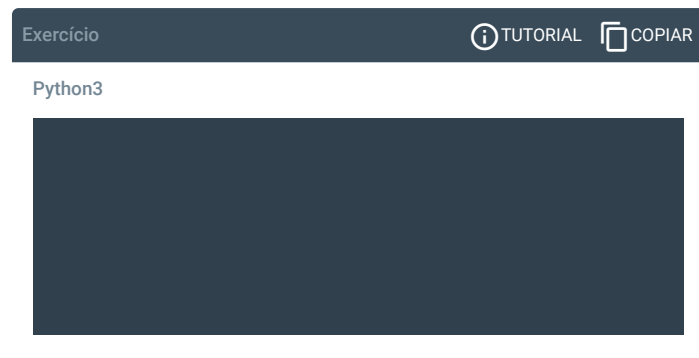


Classe Círculo com atributo de classe.

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.

Veja o programa abaixo que irá criar dois círculos. Execute e analise o resultado.



null



null



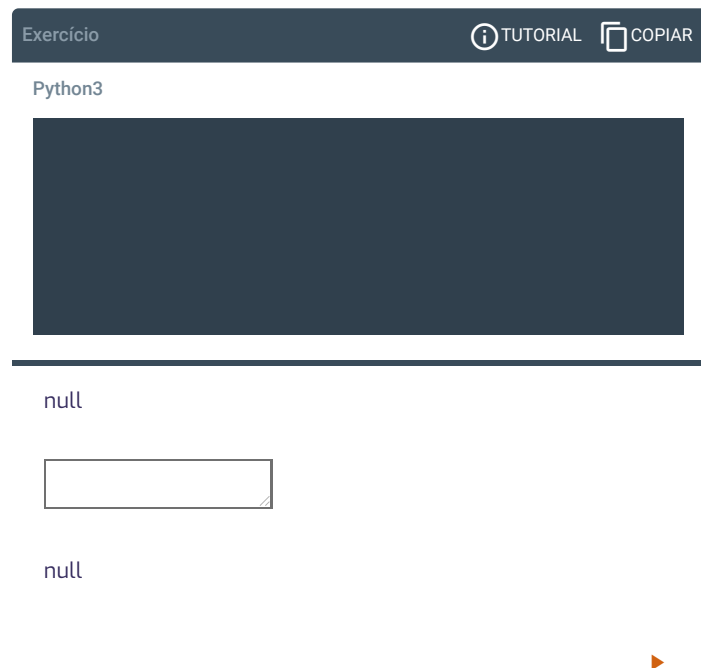
Cada vez que uma nova instância de **Circulo** é criada, o método `__init__` é chamado. Dentro deste método, `Circulo.total_circulos += 1` é utilizado para incrementar o valor de `total_circulos` diretamente na classe `Circulo`. Esse incremento é feito de forma que cada nova instância da classe contribui para aumentar o contador, refletindo o número total de círculos criados.

Quando acessamos `circ1.total_circulos` ou `circ2.total_circulos`, estamos na verdade acessando o valor de `Circulo.total_circulos`, porque `total_circulos` é um atributo de classe, e todas as instâncias compartilham o mesmo valor. Isso significa que, independentemente de qual instância é usada para acessar o atributo, o valor será o mesmo, representando o total acumulado de instâncias criadas.

No final do programa, ao acessar `Circulo.total_circulos` diretamente, o valor mostrado também será 2, confirmando que o atributo de classe é compartilhado e reflete o número total de instâncias criadas até o momento. O uso de `total_circulos` como um atributo de classe permite que o programa mantenha um registro centralizado e atualizado do número de objetos `Circulo` em qualquer ponto da execução.

No entanto, o acesso direto ao valor da variável de classe não é ideal em boas práticas de programação. Para seguir as convenções de encapsulamento, a variável de classe deveria ser marcada como privada, usando um underscore (`_`) antes do seu nome, tornando-se `_total_circulos`. Isso protege o atributo de acessos não intencionais ou modificações diretas, encorajando o uso de métodos controlados para acessar ou modificar seu valor.

Como isso fica agora? O resultado será apresentado a seguir.



O comando `print(Circulo.total_circulos)` tenta acessar diretamente o atributo `total_circulos` da classe `Circulo`. No entanto, como o atributo foi definido como `_total_circulos` (com underscore), tornou um atributo privado e não visível fora da classe. Não existe um atributo `total_circulos` na classe.

Isso resultará em um erro de atributo (`AttributeError`), pois o Python não encontrará `Circulo.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.

Memória estática.

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

Exercício

TUTORIAL

COPIAR

Python3

null

null



No código, o método `type(self)._total_circulos += 1` é utilizado para incrementar `_total_circulos` a cada vez que uma nova instância é criada.

Usar `type(self)._total_circulos` é equivalente a usar `Circulo._total_circulos`, mas permite que o código funcione corretamente mesmo se for usado em subclasses, garantindo que o incremento seja realizado na classe apropriada.

O método `get_total_circulos` é marcado com o decorator `@classmethod`, o que significa que ele pode ser chamado diretamente na classe `Circulo` ou em qualquer uma de suas instâncias.

Esse método retorna o valor atual do atributo `_total_circulos`, permitindo acessar quantas instâncias de `Circulo` foram criadas.

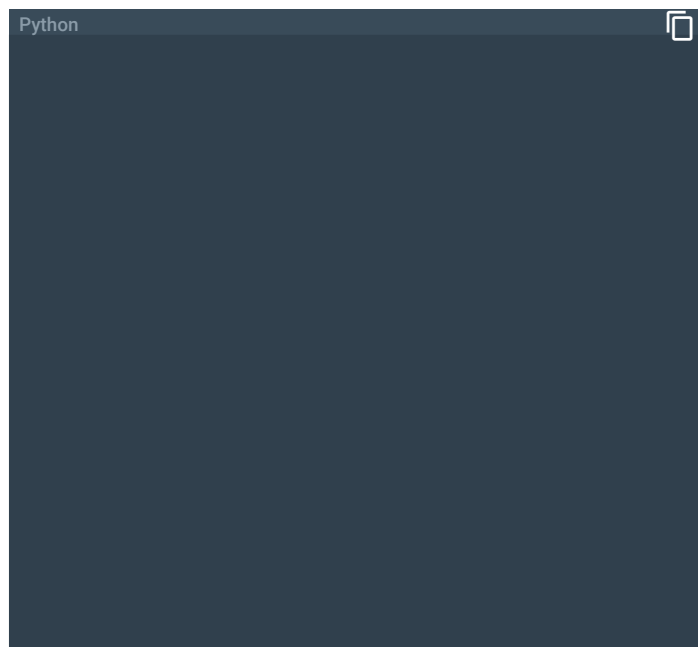
No exemplo de uso, duas instâncias de `Circulo` (`circ1` e `circ2`) são criadas, cada uma chamando o método `__init__`, que incrementa `_total_circulos`.

- Após a criação de `circ1`, `_total_circulos` é 1.
- Após a criação de `circ2`, `_total_circulos` é 2.
- O método `Circulo.get_total_circulos()` retorna 2, que é o número total de instâncias criadas até o momento.

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. Veja!



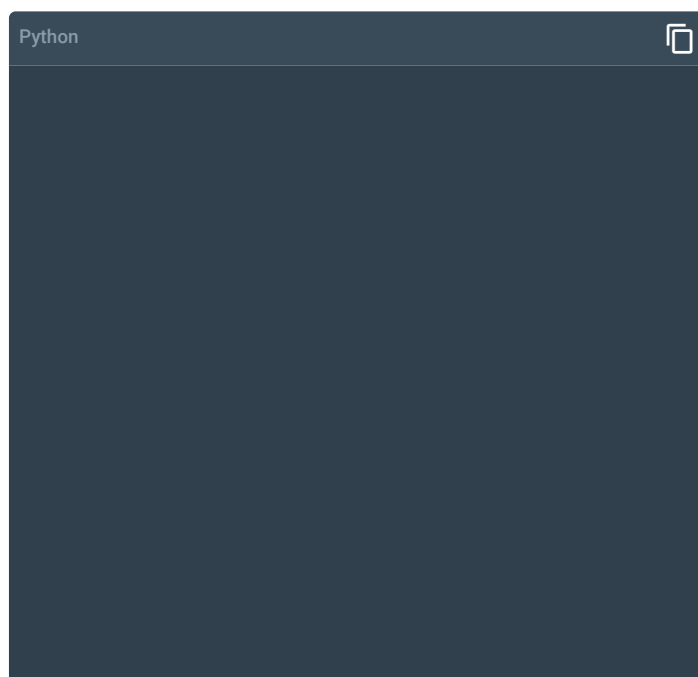
Método privado.

No código, 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. Observe!



Método estático.

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

Aqui, o método `sqrt` da classe `Math` foi chamado sem que um objeto 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.

Atividade 4

Decoradores em Python são funções que modificam o comportamento de outras funções ou métodos, adicionando funcionalidades sem alterar seu código.

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.

`@classmethod` recebe a classe como primeiro argumento, permitindo acessar ou modificar o estado da classe. `@staticmethod` não recebe

argumentos especiais e não pode alterar o estado da classe ou da instância, funcionando como uma função comum dentro da classe.

Associação entre objetos e encapsulamento na prática

Vamos realizar agora dois laboratórios: o primeiro explorando agregação e o segundo os métodos de classe e estáticos. Confira!

Cenário 1

Criar um sistema simples de gerenciamento de uma biblioteca que contém vários livros usando o conceito de agregação. Você deve criar duas classes: Livro e Biblioteca.

Siga as orientações a seguir.

1. A classe Livro deve ter os seguintes atributos:
 - título (str);
 - autor (str);
 - isbn (str).
2. A classe Biblioteca deve ter:
 - Um nome (str);
 - Uma lista de livros.
3. A classe Biblioteca deve ter métodos para:
 - Adicionar um livro à biblioteca;
 - Remover um livro da biblioteca;
 - Listar todos os livros na biblioteca.

Cenário 2

Criar uma classe Pessoa que recebe parâmetro para o construtor.

1. A classe Pessoa deve ter
 - Nome;
 - Idade;
 - Método de classe para receber o ano de nascimento;
 - Nome da pessoa e calcular sua idade;
 - Método estático que informe se a pessoa é maior ou menor de idade.

Neste vídeo, apresentaremos dois exercícios práticos abordando associação de objetos e encapsulamento em Python. Acompanhe!

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



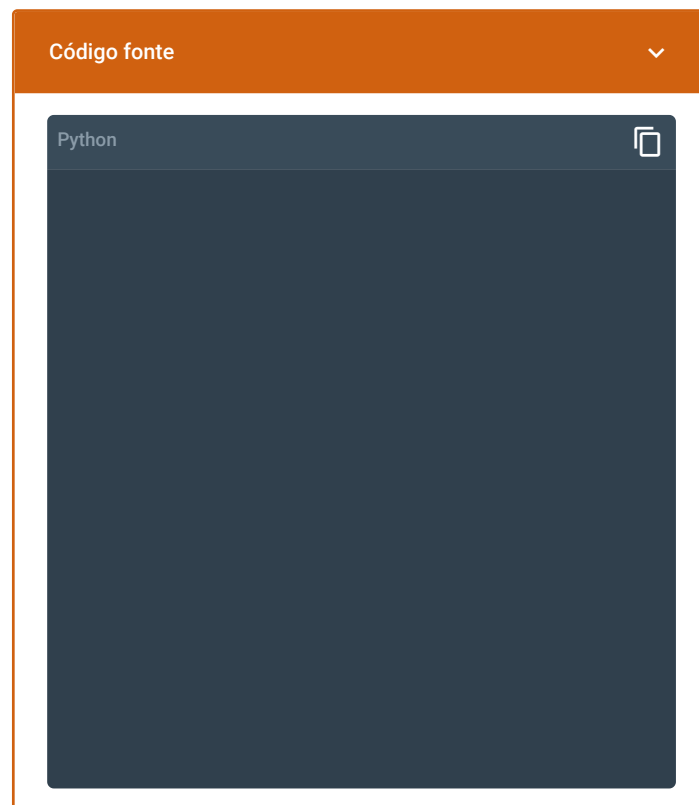
Roteiro de prática

Roteiro para o cenário 1

Acompanhe o passo a passo.

1. Defina a classe Livro:
 - Construtor que inicializa os atributos título, autor e isbn.
2. Defina a classe Biblioteca:
 - Construtor que inicializa o nome da biblioteca e uma lista vazia de livros;
 - Método para adicionar um livro (adicionar_livro);
 - Método para remover um livro (remover_livro);
 - Método para listar todos os livros (listar_livros).
3. Testar o funcionamento.

Confira, agora, o código fonte gerado.

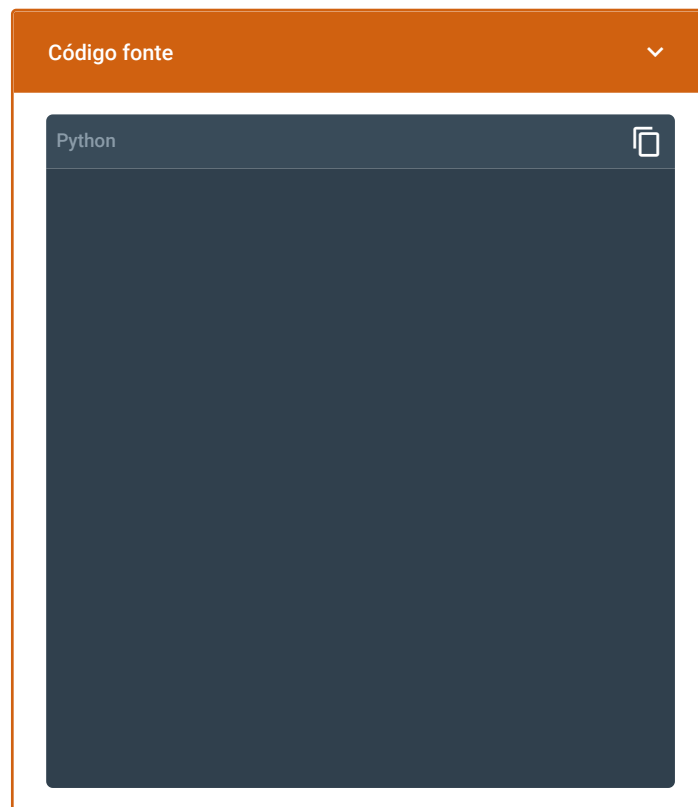


Roteiro para o cenário 2

Acompanhe o passo a passo.

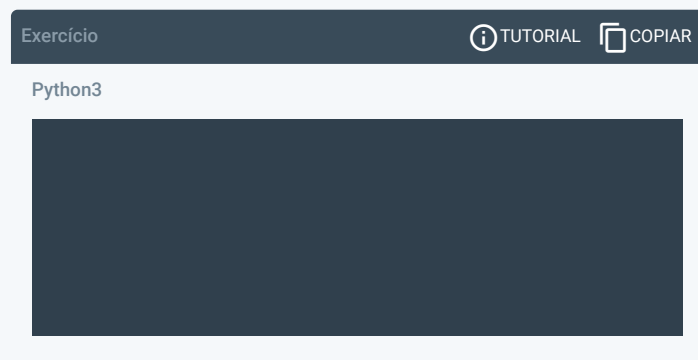
1. Faça o import para o projeto o datetime.
2. Defina a classe Pessoa:
 - Construtor que inicializa os atributos nome e idade;
 - Crie o método de classe para instanciar um objeto pessoa a partir do ano de seu nascimento;
 - Crie o método estático para definir se a pessoa instanciada é maior ou menor de idade.
3. Crie instâncias de pessoas e teste o funcionamento.

Confira, agora, o código fonte gerado.



Atividade 5

A partir da solução disponibilizada para o cenário 1, adicione um método na classe Biblioteca para buscar um livro pelo título e retornar as informações do livro.



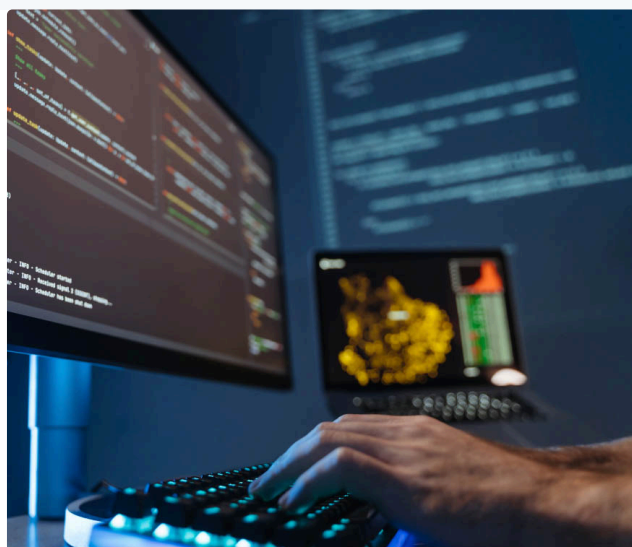
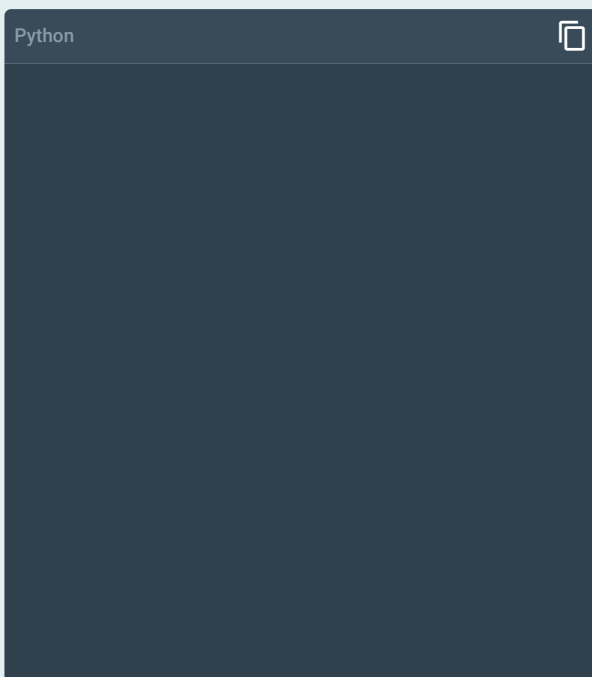
null

null



[Chave de resposta ▾](#)

Veja a seguir a solução da atividade.



3 - Orientação a objetos com herança e polimorfismo

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

Herança

Herança em Python permite que uma classe derive características e comportamentos de outra, promovendo reutilização e organização de

código. Neste conteúdo, exploraremos como implementar herança, criando hierarquias de classes que facilitam a manutenção e a extensão de aplicações. Descubra como herança pode tornar seu código mais modular e eficiente.

Neste vídeo, falaremos sobre a implementação da herança na orientação a objetos, que envolve criar uma classe derivada que herda atributos e métodos de uma classe base, permitindo reutilização de código e extensão de funcionalidades existentes. Confira!

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



Implementando 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.

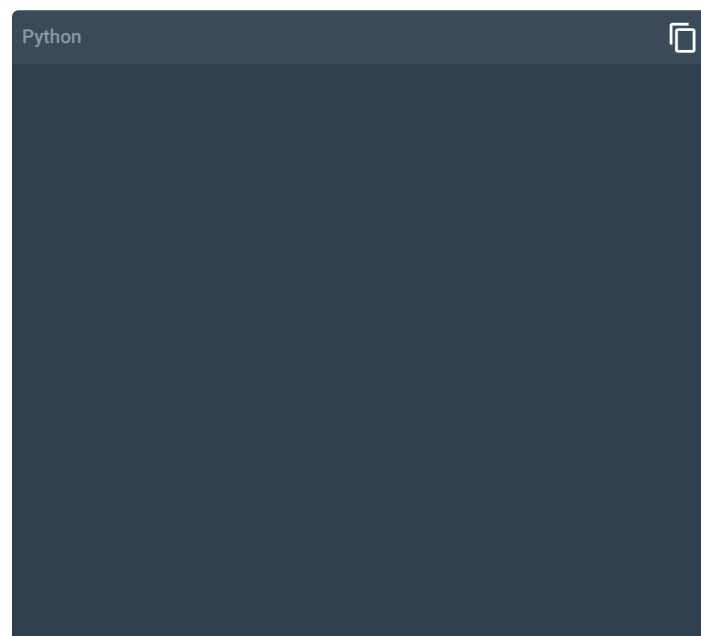
Imagine que a área de produtos de um banco define quais clientes podem ter acesso a um produto chamado Conta especial, que visa atender quem possui conta na instituição há mais de um ano. Na visão de negócios, a conta especial possui a funcionalidade de permitir o saque até um certo limite, que é determinado na criação da conta. Nesse ponto, há um dos grandes ganhos da orientação a objetos.

Pessoa utilizando caixa eletrônico.

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 imagem.

Herança Conta -> ContaEspecial.

Agora, veja o código da implementação da nova classe Conta Especial.



Classe ContaEspecial.

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

1

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 Python, 2020). A orientação a objetos permite inclusive a reutilização do construtor da classe pai (linha 6).

2

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).

3

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).

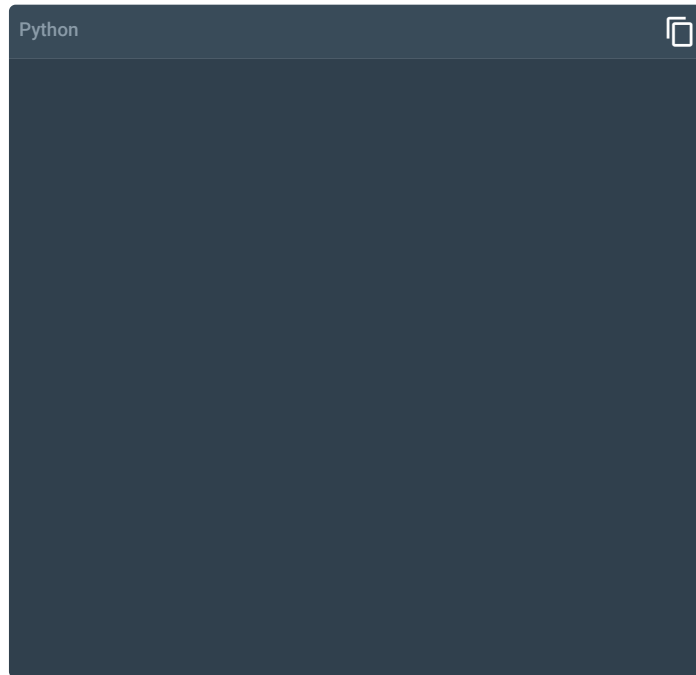
Vamos analisar o seguinte trecho de código que instancia as classes `Conta` e `ContaEspecial`.

Python





A saída do programa será:



Vamos entender a execução do programa.

O programa começa com a criação de três clientes: João (cliente1), Maria(cliente2) e Joana(cliente3). Cada cliente é identificado por seu CPF, nome e endereço.

Em seguida, o programa cria três contas bancárias:

- **Conta1:** Uma conta comum para João, com um saldo inicial de R\$ 2000.
- **Conta2:** Outra conta comum, desta vez para Maria, também com um saldo inicial de R\$ 2000.
- **Conta3:** Uma conta especial para Joana, que tem um saldo inicial de R\$ 1000 e um limite de crédito adicional de R\$ 2000.

Exibição dos Saldos Iniciais

Antes de realizar qualquer transação, o programa imprime os saldos iniciais das três contas. Essa ação permite que o usuário visualize o estado inicial de cada conta. Em particular, a conta especial de Joana é destacada por seu limite de crédito, além do saldo inicial.

Operações Bancárias

A primeira operação realizada no programa é um depósito de R\$ 500 na conta de Maria. Este depósito é simples e direto, aumentando o saldo de

Maria para R\$ 2500. Logo após, o programa tenta realizar um saque de R\$ 3000 da mesma conta.

No entanto, como o saldo de Maria não é suficiente para cobrir esse saque, a operação falha. O saldo permanece inalterado em R\$ 2500, e o programa informa ao usuário que o saque não foi possível devido ao saldo insuficiente.

Na sequência, o programa foca na conta especial de Joana. Primeiro, R\$ 100 são depositados, elevando o saldo de Joana para R\$ 1100. Em seguida, o programa realiza um saque de R\$ 2000. Como esse valor excede o saldo disponível, a conta entra no limite de crédito, resultando em um saldo negativo de R\$ -900 e reduzindo o limite de crédito de Joana para R\$ 1100. O sistema permite essa transação e atualiza os valores adequadamente, refletindo o uso do limite de crédito.

Finalmente, o programa tenta realizar um segundo saque de R\$ 2000 na conta de Joana, que já está utilizando parte de seu limite. Desta vez, o saque falha, pois o valor solicitado ultrapassa o limite de crédito restante. O programa então exibe uma mensagem informando que a operação não pode ser concluída e mantém os valores do saldo e do limite inalterados.

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.

Comentário

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

É um mecanismo que possibilita a uma classe herdar o código de duas ou mais superclasses.

A herança múltipla é implementada 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.

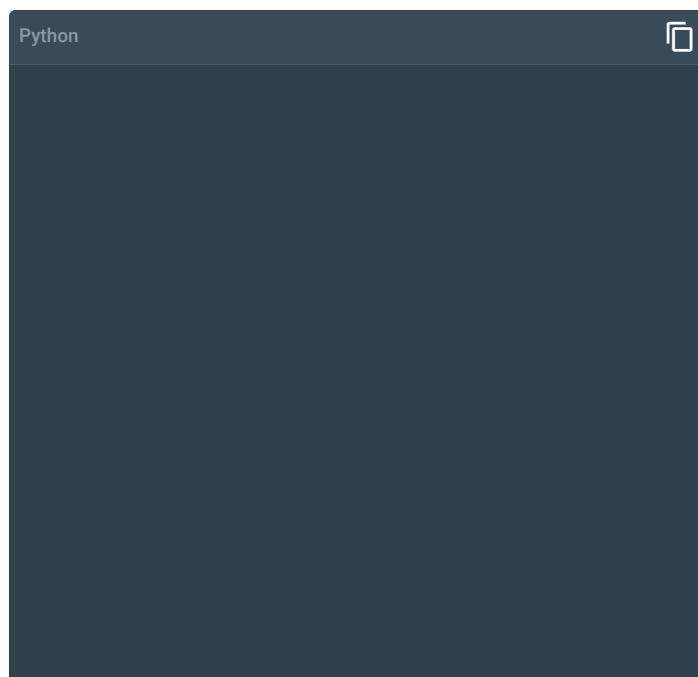
Terá de ser cobrada uma taxa de manutenção mensal, mesmo se o rendimento for de apenas um dia.

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

Este será o diagrama com as novas classes criadas no sistema de conta corrente, observe!

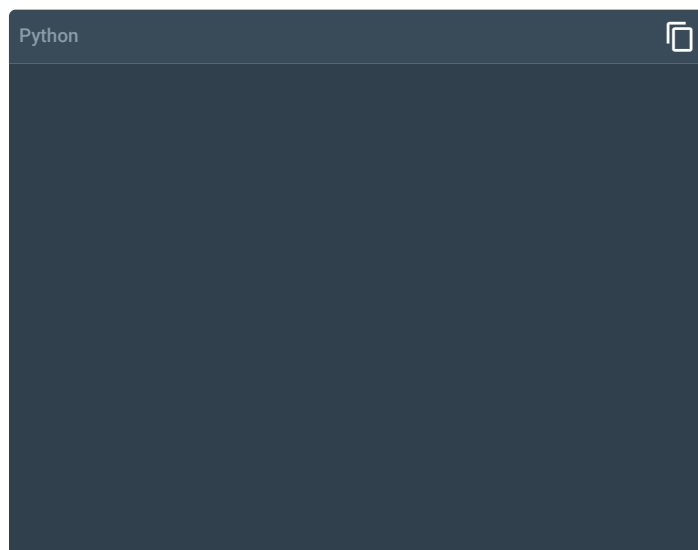
Herança múltipla.

A implementação da classe ContaPoupanca está detalhada a seguir.



Conta-poupança.

Veja agora a implementação da classe ContaRemuneradaPoupanca.



Conta remunerada.

Observe 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.

Vamos analisar o programa abaixo.

```
Python
```

A saída do programa será:

```
Python
```

O programa cria dois clientes, João(**cliente1**) e Maria(**cliente2**), que são representados por objetos da classe Cliente.

A seguir, o programa cria três tipos distintos de contas bancárias. A primeira conta, associada a João e Maria, é uma conta bancária tradicional (**conta1**) com um saldo inicial de R\$ 2000. Essa conta é criada utilizando a classe Conta.

Paralelamente, uma segunda conta é criada usando a classe **Poupanca (contapoupanca1)**. Esta conta é especial, pois é definida com uma taxa de remuneração mensal de 10%. Embora não seja diretamente associada a um cliente ou saldo inicial neste cenário, a criação da conta poupança serve para ilustrar como a remuneração pode ser aplicada a um saldo ao longo do tempo.

A terceira conta, no entanto, é a mais interessante. Criada para João (**contaremunerada1**), a **ContaRemuneradaPoupanca** combina as características de uma conta tradicional com os benefícios de uma conta poupança.

Com um saldo inicial de R\$ 1000 e uma taxa de remuneração de 10% ao mês, essa conta não só permite operações bancárias comuns, como também aplica uma remuneração diária ao saldo, destacando a flexibilidade e o poder da herança múltipla utilizada na definição da classe ContaRemuneradaPoupanca.

Após a criação das contas, o programa aplica a remuneração à conta remunerada de poupança de João. O método remuneraConta é chamado, e o saldo de R\$ 1000 é ajustado com a remuneração diária baseada na taxa mensal.

Especificamente, a remuneração é calculada dividindo a taxa mensal por 30 dias, resultando em um acréscimo de aproximadamente R\$ 3,33 ao saldo original.

Finalmente, o programa exibe o saldo atualizado da conta remunerada de João. Após a aplicação da remuneração, o saldo da conta é apresentado como R\$ 1003,33, confirmando que a remuneração foi corretamente aplicada.

Atividade 1

Herança é um conceito fundamental na programação orientada a objetos (OO), permitindo que uma classe adquira atributos e métodos de outra classe. Em Python, herança é uma técnica poderosa para reutilizar código e criar hierarquias de classes.

Qual das seguintes afirmações descreve corretamente o conceito de herança em Python?

A	Herança em Python permite que uma classe adquira apenas métodos de outra classe, não atributos.
B	Uma classe em Python pode herdar de várias classes pai, mas não pode sobrescrever métodos herdados.
C	Herança em Python é uma técnica usada exclusivamente para criar atributos em uma classe.
D	Ao herdar de uma classe pai em Python, a classe filha pode substituir os métodos da classe pai com implementações diferentes.
E	Em Python, herança é uma técnica usada para ocultar métodos e atributos de uma classe pai.

Parabéns! A alternativa D está correta.

A herança em Python permite que uma classe filha herde atributos e métodos de uma classe pai. A alternativa d) destaca corretamente que, ao herdar de uma classe pai, a classe filha pode substituir os métodos da classe pai com suas próprias implementações, se necessário. Isso é conhecido como sobrescrita de método, uma prática comum em programação orientada a objetos. As outras alternativas estão incorretas porque: a. a herança permite herdar tanto métodos quanto atributos; b. uma classe em Python pode, sim, sobrescrever métodos herdados; c. a herança não é usada exclusivamente para criar novos atributos, mas para herdar e reutilizar código; e. herança não é usada para ocultar métodos e atributos, mas para torná-los acessíveis à classe filha.

Polimorfismo

Polimorfismo em Python permite que objetos de diferentes classes sejam tratados através de uma interface comum, adaptando comportamentos conforme necessário. Este conteúdo explicará como implementar polimorfismo, utilizando métodos e interfaces para criar código flexível e dinâmico. Aprenda a aproveitar este poderoso conceito para escrever programas mais adaptáveis e elegantes.

Neste vídeo, falaremos sobre a utilização e implementação do polimorfismo na linguagem de programação Python. Confira!

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



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.

Pessoa programando em computador.

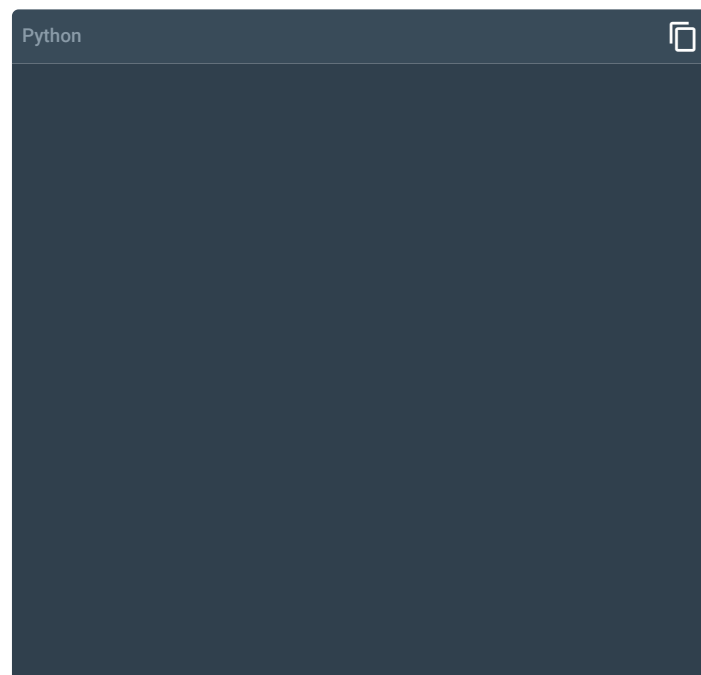
Vamos imaginar que agora tenhamos uma entidade Banco que controla todas as contas criadas no sistema de conta corrente. As contas podem ser do tipo conta, contacomum ou contarenumerada. O cálculo do rendimento da conta Cliente desconta IOF e IR; a conta Renumerada, apenas o IOF. Já a conta Cliente não 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.

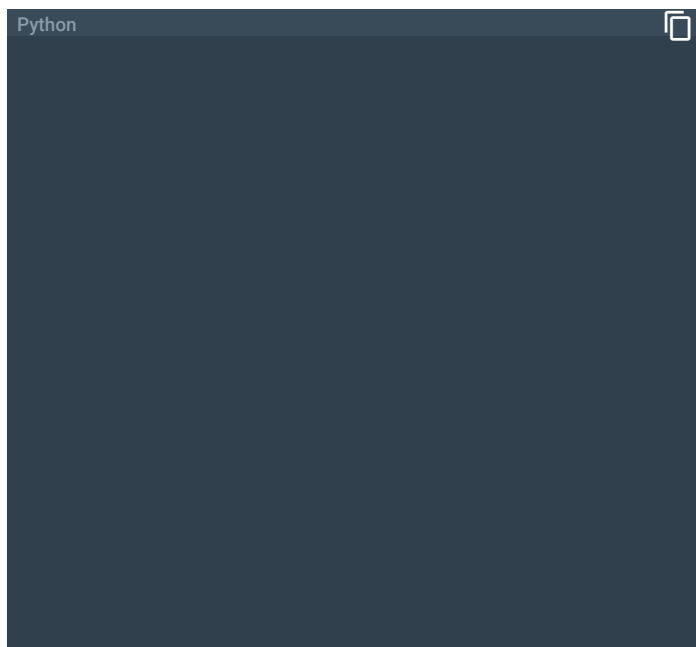
Diagrama Banco.

Vamos analisar agora a implementação da classe ContaCliente. Acompanhe!



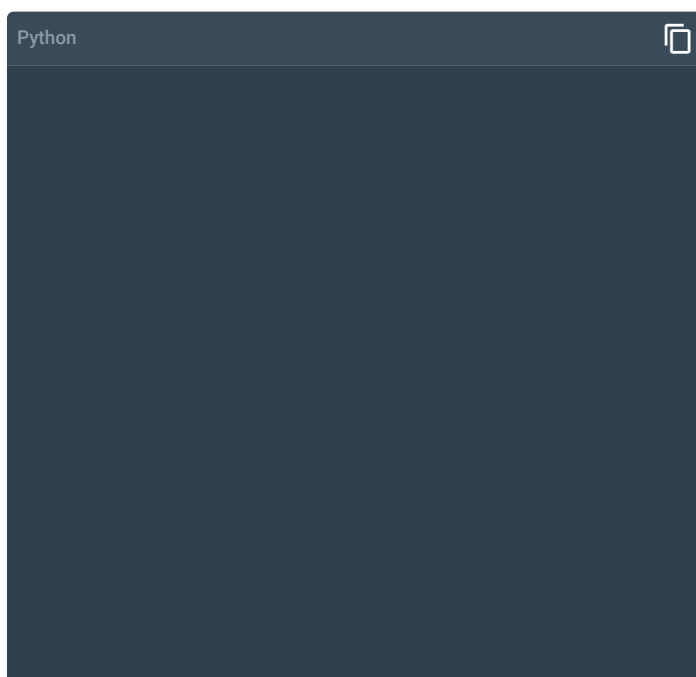
Classe ContaCliente.

Agora, vamos analisar agora a implementação da classe ContaComum.



Classe ContaComum.

E, por fim, veja agora a criação da classe ContaRemunerada.



Classe ContaCliente.

Veja algumas observações.

- 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.

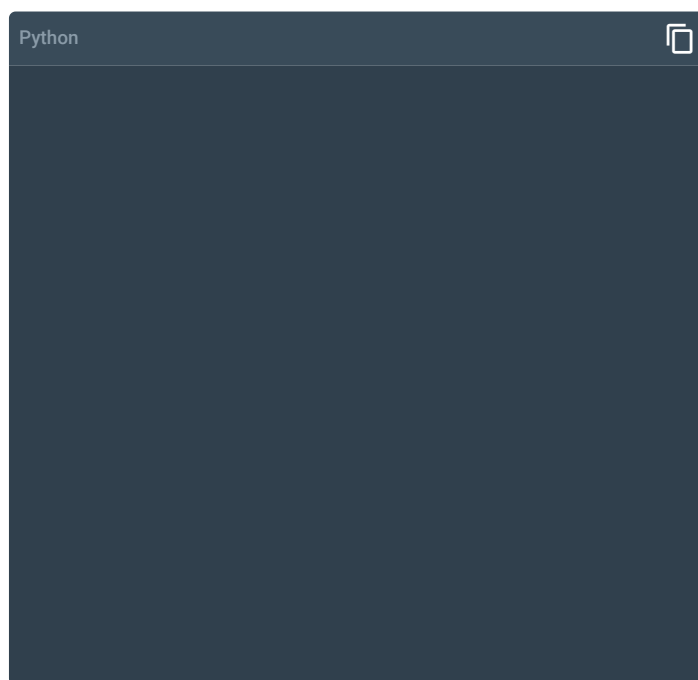
Vamos analisar a implementação da classe Banco e do programa que a utiliza.





Classe Banco.

Vamos analisar o programa que utiliza as classes.



Programa que utiliza as classes.

Observe os itens a seguir.

- 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 ContaCliente. Em (5), o objeto contacomum1 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 `ContaCliente` 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.

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.

Atividade 2

Em relação ao conceito de polimorfismo em Python, podemos afirmar que:

A

O polimorfismo permite que métodos com o mesmo nome, mas com diferentes assinaturas, sejam definidos em uma mesma classe.

B

Em Python, o polimorfismo se refere à capacidade de objetos de diferentes classes responderem ao mesmo método de maneira específica à sua classe.

C

Polimorfismo em Python é implementado exclusivamente através de herança múltipla.

D

Python não suporta polimorfismo, pois as classes não podem compartilhar métodos com o mesmo nome.

E

O polimorfismo em Python permite que uma classe redefina métodos de outra classe, mas não permite que objetos diferentes compartilhem um método comum.

Parabéns! A alternativa B está correta.

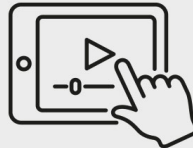
O polimorfismo em Python refere-se à capacidade de diferentes classes implementarem métodos com o mesmo nome, mas com comportamentos específicos a cada classe. Isso permite que funções e métodos utilizem objetos de diferentes classes de forma intercambiável, contanto que esses objetos implementem o método requerido. Esse conceito é fundamental para a flexibilidade e reutilização de código em programação orientada a objetos.

Herança e polimorfismo na prática

Neste conteúdo, você aprenderá a implementar polimorfismo e herança em Python, criando um sistema de classes que modela diferentes tipos de animais e suas habilidades de movimento e fala. Você vai definir uma classe base, criar subclasses específicas para adicionar capacidades de movimento.

Neste vídeo, apresentaremos um exercício prático de criação de classes base e subclasses mostrando os aspectos de herança e polimorfismo em Python. Acompanhe!

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



Roteiro de prática

Siga o passo a passo para criação da Classe Base Animal.

1. Defina uma classe base chamada Animal com dois métodos (o construtor da classe deve receber como parâmetro o nome do animal):
 - falar(self): método abstrato para ser implementado por subclasses.
 - mover(self): método abstrato para ser implementado por subclasses.
2. Crie três subclasses (Cachorro, Gato, Vaca) que herdam de Animal e implementam os métodos falar e mover.
3. Crie as classes (Voador e Nadador) para adicionar habilidades de movimento específicas.
4. Crie uma classe Pato que herda de Animal, Voador e Nadador. Implemente os métodos falar e mover, além de um método andar.

5. Implemente duas funções (fazer som e fazer movimento) que aceitam um objeto Animal e chamam os métodos falar e mover.
6. Crie instâncias das classes Cachorro, Gato, Vaca e Pato. Use as funções fazer som e fazer movimento para verificar os comportamentos.

Exercício

TUTORIAL

COPIAR

Python3

null

null



Atividade 3

Altere a solução apresentada no vídeo acrescentando uma classe Jacaré que anda e nada.

Crie um objeto da classe e teste o funcionamento.

Exercício

TUTORIAL

COPIAR

Python3

null

null



Chave de resposta ▾

Veja a seguir o código da solução da atividade.

Python

Classe abstrata e classe para tratamento de exceções

Classes abstratas em Python fornecem uma estrutura para outras classes, definindo métodos que devem ser implementados pelas subclasses. Já as classes para tratamento de exceções facilitam a criação de exceções personalizadas, melhorando o manejo de erros. Neste conteúdo, abordaremos como usar classes abstratas e criar classes de exceção, fortalecendo a robustez e a clareza do seu código.

No vídeo a seguir, falaremos sobre a implementação de classes abstratas e tratamento de exceções na linguagem de programação Python. Acompanhe!

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



Classes abstratas

Definir uma classe abstrata é uma característica bastante utilizada pela programação orientada a objetos. Falaremos sobre esse tipo de classe a seguir.

Uma classe abstrata não pode ser instanciada durante a execução do programa orientado a objetos, ou seja, não pode haver objetos dessa classe executados na memória.

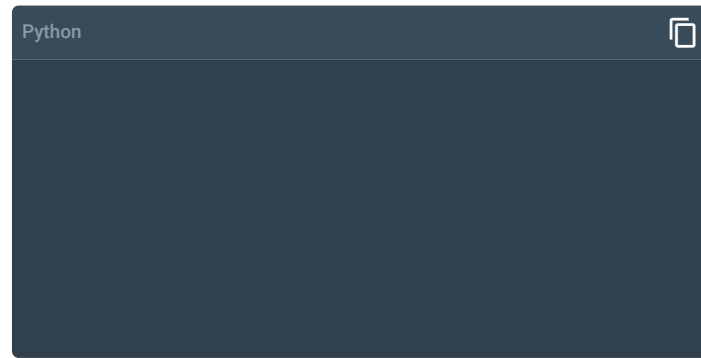
O termo “abstrato” remete a um conceito do mundo real, considerando a necessidade apenas de objetos concretos no programa. A classe abstrata se encaixa perfeitamente no problema do sistema de conta corrente do banco. Nesse sistema, o banco não quer tratar de clientes do tipo ContaCliente, e sim apenas dos objetos do tipo ContaComum e Conta VIP.

Observe a imagem.

Diagrama de classes abstratas.

Houve apenas a adição de estereótipo <> 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).

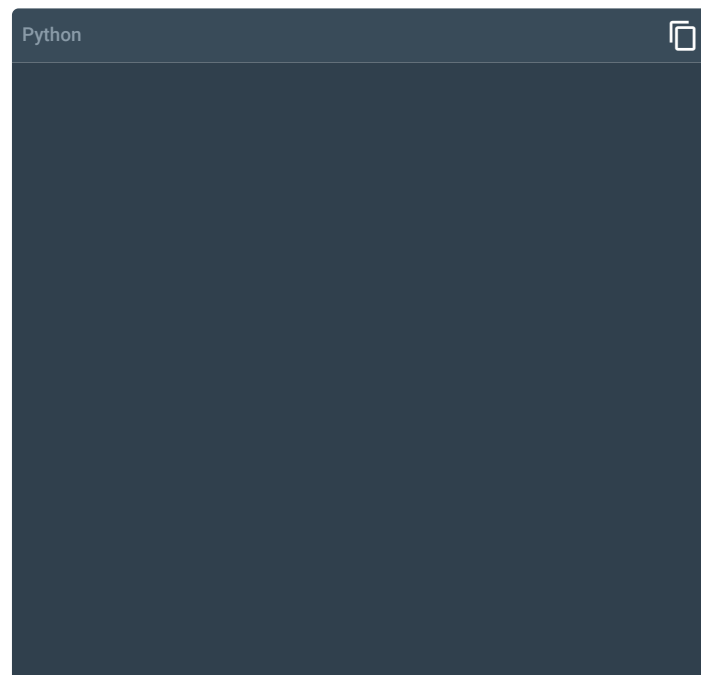
Toda classe abstrata é uma subclasse da classe ABC (Caelum, 2020).
Para tornar a classe Conta Cliente abstrata, muda-se sua definição.
Observe no código a seguir.



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 a seguir.



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, 1000, 0.1)
```





null



null



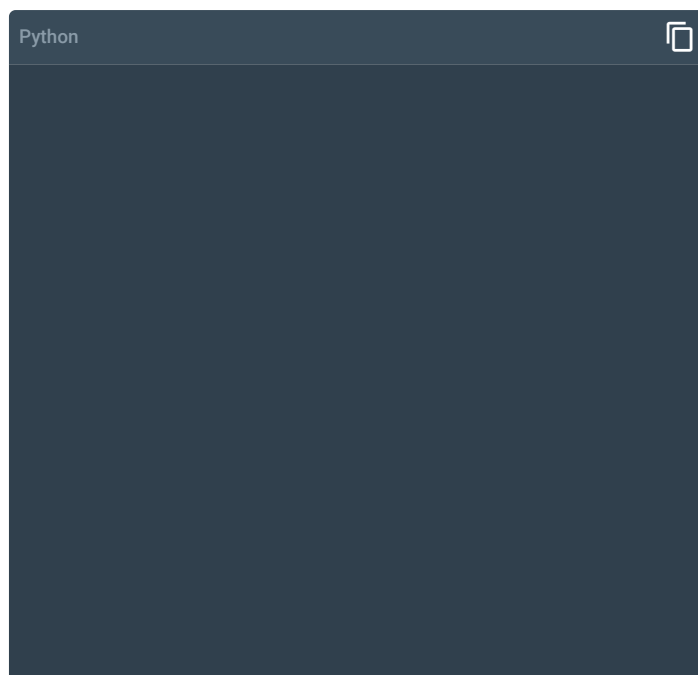
Ao tentar criar uma instância de `ContaCliente` com `cc1 = ContaCliente(1, 0.1, 0.25, 1000, 0.1)`, o Python irá lançar um erro do tipo `TypeError`. Esse erro ocorre porque `ContaCliente` é uma classe abstrata e possui métodos abstratos que não foram implementados.

Para que o código funcione, você deve criar uma subclasse concreta de `ContaCliente` que implemente o método `calcula_rendimento`. Somente depois disso, você poderá instanciar essa subclasse.

Considerando a modelagem que estamos utilizando, apenas as subclasses `ContaComum` e `ContaVIP` podem ser instanciadas.

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.

Uma solução seria:



Recomendação

Como exercício, crie as classes concretas `ContaComum` e `ContaVIP` sem implementar o método abstrato do superclasse `ContaCliente`.

Tratamento de Exceções em Python: Conceitos e Práticas

No desenvolvimento de software, lidar com situações imprevistas é uma necessidade constante. Em Python, essas situações são tratadas por meio de exceções, que representam eventos inesperados capazes de interromper o fluxo normal de um programa.

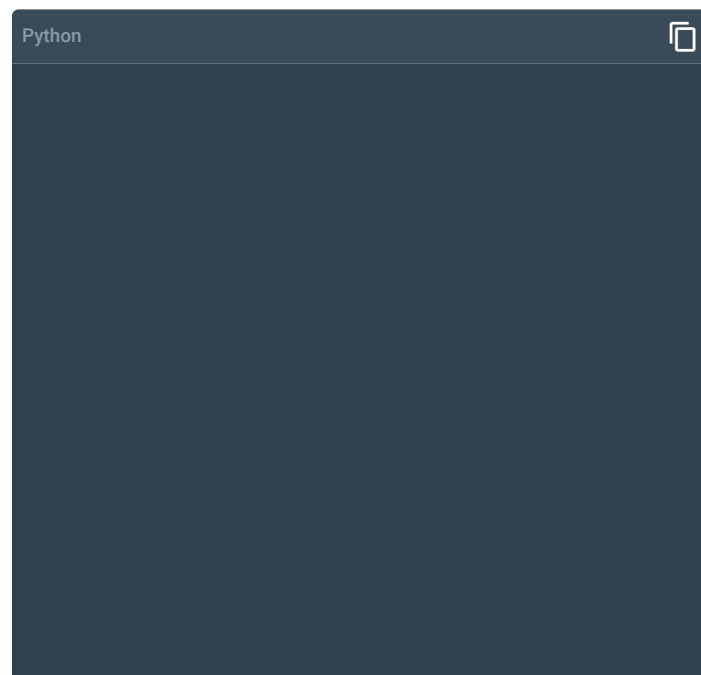
O que São Exceções?

Exceções são eventos que ocorrem durante a execução de um programa, interrompendo seu fluxo normal. Em Python, quando uma exceção é lançada e não é tratada, o programa encerra sua execução abruptamente, exibindo uma mensagem de erro. Para evitar esse comportamento indesejado, é fundamental entender como capturar e tratar exceções, garantindo que o programa continue funcionando de forma controlada.

Além disso, Python permite a criação de exceções personalizadas, o que possibilita a diferenciação entre os erros inerentes à linguagem e aqueles específicos da lógica da aplicação. Isso contribui para um código mais claro e fácil de manter.

Tratamento de Exceções: Divisão por Zero

Um exemplo clássico de exceção ocorre ao tentar realizar uma divisão por zero, o que em Python resulta na exceção `ZeroDivisionError`. Para ilustrar o tratamento desse tipo de erro, considere o seguinte exemplo:

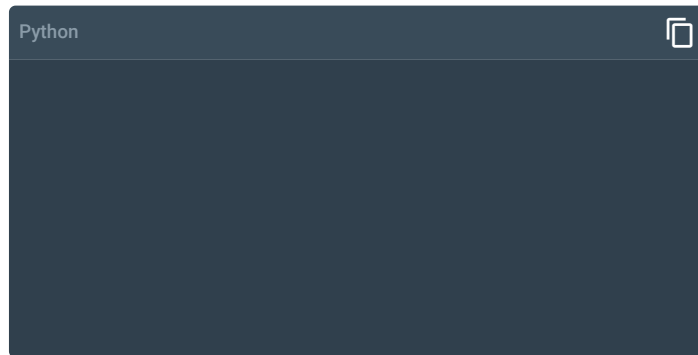


Neste código, ao tentar dividir 10 por 0, Python lança uma exceção `ZeroDivisionError`. O bloco `try...except` captura essa exceção, permitindo que o programa lide com o erro de forma controlada, imprimindo uma mensagem de erro personalizada em vez de interromper a execução do programa.

Criando Exceções Personalizadas

Python também permite a criação de exceções personalizadas, que são úteis quando se deseja capturar e tratar erros específicos de um determinado contexto da aplicação. Para criar uma exceção

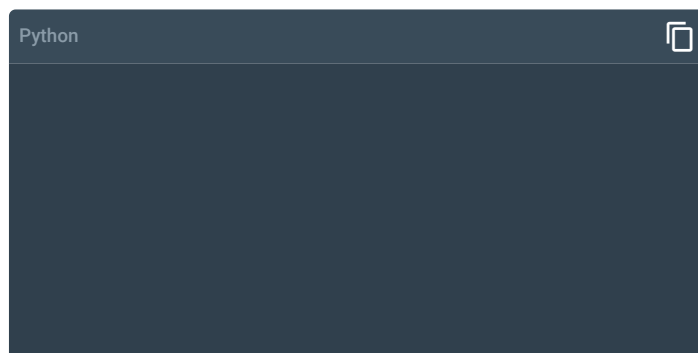
personalizada, utilizamos a herança da classe `Exception`, conforme o exemplo a seguir:



A classe `ExcecaoCustomizada` define uma nova exceção que pode ser lançada e tratada separadamente das exceções padrão de Python. O uso do comando `pass` indica que não há necessidade de adicionar comportamento específico à exceção; a distinção pelo nome é o suficiente.

Lançando Exceções

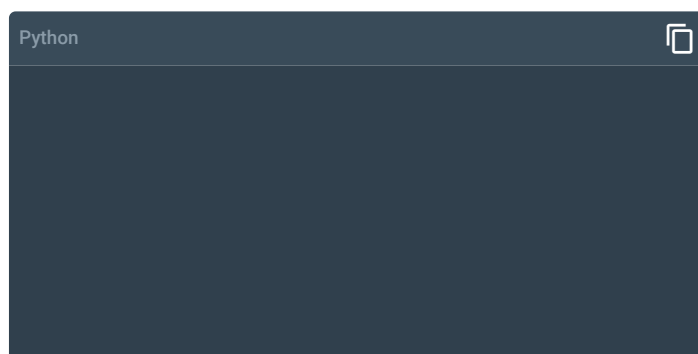
Uma vez que uma exceção personalizada tenha sido criada, ela pode ser lançada em situações específicas, usando a instrução `raise`. Considere o exemplo abaixo, onde um método lança a exceção `ExcecaoCustomizada` caso o valor passado seja negativo:



Neste exemplo, ao detectar um valor negativo, a função `checa_valor` interrompe sua execução normal e lança a exceção `ExcecaoCustomizada`, transmitindo uma mensagem explicativa. Esse mecanismo é essencial para garantir que condições indesejadas sejam tratadas adequadamente.

Tratando Exceções com `try...except`

Uma vez que uma exceção tenha sido lançada, é importante saber como tratá-la. O bloco `try...except` é a ferramenta adequada para esse propósito:



Neste código, a função `checa_valor` é chamada com um valor negativo, o que resulta na exceção `ExcecaoCustomizada` sendo lançada. O bloco `try...except` captura essa exceção, e o programa trata o erro de forma controlada, exibindo a mensagem associada à exceção.

Vamos a um exemplo!

Neste exemplo, definimos uma exceção personalizada chamada `ExcecaoCustomizada`, criamos funções que podem lançar essa exceção e outras exceções comuns, e, em seguida, tratamos essas exceções usando blocos `try...except`.

Execute o código a seguir e veja o resultado!

Exercício

TUTORIAL

COPIAR

Python3

null

null

Neste código, temos três componentes principais:

1. Definição da Exceção Personalizada:

- 1.1. A classe `ExcecaoCustomizada` é criada para representar uma exceção específica que pode ser lançada em nosso código. Ela herda de `Exception`, a classe base para todas as exceções em Python. Isso permite que `ExcecaoCustomizada` seja usada e tratada como qualquer outra exceção no Python.

2. Funções que Podem Lançar Exceções:

- 2.1. A função `checa_valor(valor)` verifica se um valor é negativo. Se for, ela lança uma `ExcecaoCustomizada` com uma mensagem explicativa.
- 2.2. A função `divide(a, b)` realiza a divisão de dois números. Se `b` for zero, o Python automaticamente lança uma `ZeroDivisionError`.

3. Tratamento de Exceções:

- 3.1. O bloco `try...except` é utilizado para tentar executar as funções e capturar qualquer exceção que seja lançada. No primeiro bloco, tentamos dividir 10 por 0, o que gera

uma `ZeroDivisionError`. Essa exceção é capturada e uma mensagem de erro é exibida.

- 3.2. No segundo bloco `try...except`, chamamos `checa_valor(-10)`, o que resulta na exceção personalizada `ExcecaoCustomizada` sendo lançada. O bloco `except` captura essa exceção e exibe a mensagem associada.

O tratamento de exceções é uma prática fundamental na programação em Python, permitindo que os programas sejam mais robustos e resilientes a erros. Esse exemplo destaca a flexibilidade de Python em permitir a criação de exceções personalizadas, que podem ser utilizadas para capturar e tratar condições específicas da lógica do seu programa. Isso é especialmente útil em projetos maiores, onde a clareza e a especificidade dos erros são cruciais para a manutenção e depuração do código.

Atividade 4

Classes abstratas são uma parte importante da programação orientada a objetos em Python, fornecendo uma estrutura para outras classes e definindo métodos que devem ser implementados por suas subclasses. Qual das seguintes afirmações melhor descreve uma classe abstrata em Python?

A

Uma classe abstrata em Python é uma classe que não pode ser instanciada diretamente e pode conter métodos concretos e métodos abstratos.

B

Em Python, uma classe abstrata é uma classe que só pode conter métodos abstratos e não pode ser herdada por outras classes.

C

Uma classe abstrata em Python é uma classe que só pode conter métodos concretos e não pode ser herdada por outras classes.

D

Em Python, uma classe abstrata é uma classe que pode ser instanciada diretamente e não pode conter métodos concretos.

E

Em Python, classes abstratas não são suportadas; todas as classes devem ser instanciáveis diretamente.

Parabéns! A alternativa A está correta.

Em Python, uma classe abstrata (A) é uma classe que não pode ser instanciada diretamente e pode conter métodos concretos (com implementações) e métodos abstratos (sem implementações). Métodos abstratos são métodos que devem ser implementados por subclasses concretas, garantindo uma estrutura consistente na hierarquia de classes. As outras opções estão incorretas. Letra B: classes abstratas podem ser herdadas por outras classes e podem conter métodos concretos; Letra C: uma classe abstrata em Python pode conter métodos abstratos; letra D: uma classe abstrata em Python não pode ser instanciada diretamente; letra E: Python suporta classes abstratas usando o módulo abc.

Classes abstratas e tratamento de exceções na prática

Vamos praticar conceitos de programação orientada a objetos em Python, com foco na utilização de classes abstratas e no tratamento de exceções; para isso, abordaremos dois cenários. Veja!

Cenário 1

Acompanhe o passo a passo.

1. Criar um sistema de classes para modelar diferentes tipos de veículos, utilizando uma classe abstrata como base para definir a interface comum.
2. Chamar os métodos mover() e ligar() em cada instância.

Cenário 2

Acompanhe o passo a passo.

1. Criar uma classe chamada Calculadora que tenha métodos para realizar operações matemáticas básicas: adição, subtração, multiplicação e divisão.
2. Implementar tratamento de exceções para evitar tipos de dados incorretos.

Neste vídeo, apresentaremos dois exercícios práticos ilustrando o uso de classes abstratas e o tratamento de exceções em Python. Confira!

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



Roteiro de prática

Roteiro para o cenário 1

Acompanhe o passo a passo.

1. Importe abc para o projeto (abstract base classes).
2. Defina a classe abstrata Veículo:
 - Crie uma classe abstrata chamada Veículo com os métodos abstratos mover() e ligar().
3. Crie subclasses de Veículo:
 - Implemente duas subclasses de Veículo: Carro e Bicicleta;
 - Cada subclasse deve fornecer implementações concretas para os métodos abstratos de acordo com o comportamento de cada veículo.
4. Teste as implementações:
 - Crie instâncias de Carro e Bicicleta;
 - Chame os métodos mover() e ligar() em cada instância e verifique as saídas para garantir que funcionem conforme o esperado.

Confira, agora, o código fonte gerado.

Código fonte

Python

Roteiro para o cenário 2

Acompanhe o passo a passo.

1. Defina a classe Calculadora:

- Crie métodos para adição, subtração, multiplicação e divisão;
- Tratamento a exceções de tipos de dados incorretos.

2. Teste as implementações:

- Crie instâncias da Classe;
- Chame os métodos e realizar operações aritméticas;
- Passe valores inválidos para os métodos.

Confira, agora, o código fonte gerado.

Código fonte

Python

Atividade 5

Primeiro, faça o seguinte exercício:

A partir da solução do cenário 1, crie uma classe chamada Aviao que herda de Veículo e implemente os métodos mover() e ligar() para representar o comportamento de um avião.

Exercício

TUTORIAL

COPIAR

Python3

```
null
```

```
null
```



Chave de resposta ▾

Python



Em seguida, realize o exercício proposto a seguir.

A partir da solução do cenário 2, implemente a correção do erro de divisão por zero

Exercício

TUTORIAL

COPIAR

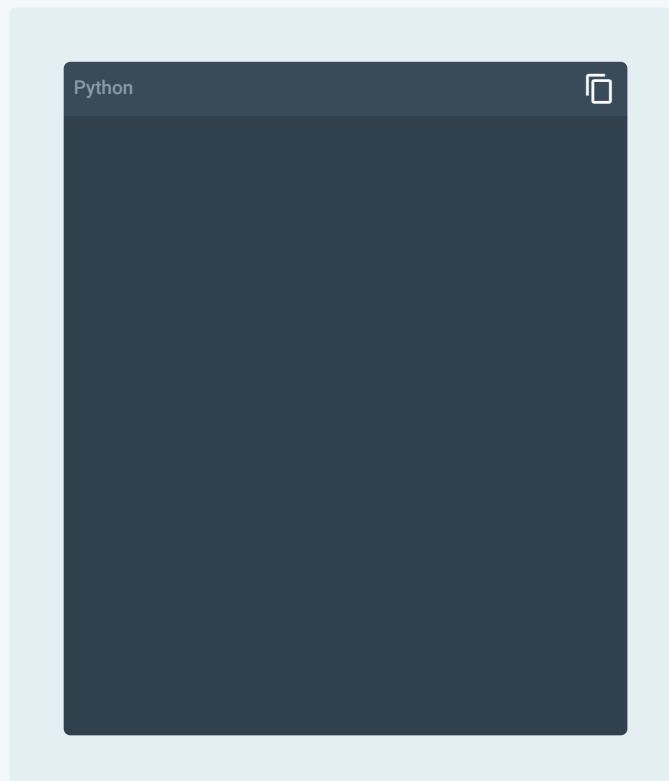
Python3

null

null



Chave de resposta ▾



O que você aprendeu neste conteúdo?

- Conceitos e pilares de programação orientada a objetos.
- Implementação de encapsulamento.
- Aplicação dos mecanismos de herança e polimorfismo.
- Tipos de associação entre objetos.
- Emprego de classes abstratas.
- Criação de tipos de exceções.



Fala, mestre!

O vídeo apresenta Gustavo Guanabara introduzindo a disciplina "Paradigmas de Linguagem de Programação em Python", com foco na programação orientada a objetos (POO). Ele explica a importância da POO para organização, reaproveitamento e extensibilidade do código, abordando seus quatro pilares: abstração, encapsulamento, herança e polimorfismo. O curso é dividido em três módulos: conceitos básicos de classe, objeto, métodos e atributos; aplicação da POO em Python; e conceitos avançados como herança e polimorfismo. Gustavo enfatiza a relevância da POO para o desenvolvimento de programas escaláveis e recomenda acompanhar a sequência de vídeos para um aprendizado eficaz.

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



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 pela 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.

Referências

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

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

CORREIA, C.; TAFNER, M. **Análise orientada a objetos**. Gurarulhos: Visual Books, 2018.

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

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

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. Nova York: Apress, 2017.

PYTHON COURSE. **Blog Python**. Consultado na internet em: 14 jun. 2020.

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

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

REAL PYTHON. **Blog Python**. Consultado na internet em: 14 jun. 2020.



Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.



Download material

O que você achou do conteúdo?



Relatar problema