**Introdução da unidade**



Objetivos da Unidade

Ao longo desta Unidade, você irá:

* identificar a classe em Python;
* ilustrar os conceitos de biblioteca request;
* demonstrar a linguagem de consulta estruturada.

Olá estudantes, sejam bem-vindos!

Nesta unidade, abordaremos conceitos importantes sobre Python, linguagens de programação, banco de dados e linguagens SQL.

Você terá a oportunidade de treinar nos simuladores os códigos apresentados nas aulas.

Desejo a você muito sucesso!

**Introdução da aula**



Qual é o foco da aula?

Nesta aula, vamos compreender alguns conceitos básicos do paradigma de programação orientado a objetos.

Objetivos gerais de aprendizagem

Ao longo desta aula, você irá:

* identificar a classe em Python;
* examinar o método *construtor*na herança e sobrescrita;
* interpretar a herança múltipla.

**Situação-problema**

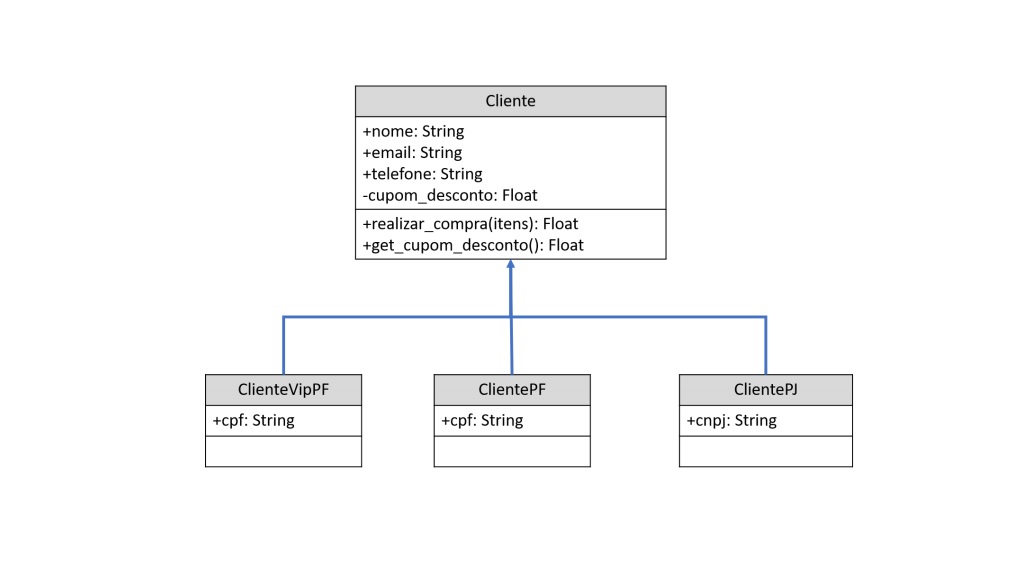
Um diagrama de classes permite ilustrar o reúso de código através do mecanismo de herança, onde uma classe herda os atributos e métodos de outra classe.

Como desenvolvedor em uma empresa de consultoria de software, você foi alocado para iniciar a implementação de um sistema de vendas. Nesta primeira etapa, os engenheiros de software fizeram um levantamento de requisitos e identificaram que o sistema vai atender a três tipos distintos de clientes: pessoas físicas (PF) que compram com assiduidade, pessoas físicas que compram esporadicamente e pessoas jurídicas (PJ).

Os clientes VIPs podem comprar um número ilimitado de itens. Por sua vez, os clientes PF que não são VIPs, só podem comprar até 20 itens, e os clientes PJ podem comprar até 50 itens. No levantamento de requisitos, também foi determinado que cada tipo de cliente terá um cupom de desconto que será usado para conceder benefícios nas compras. Os clientes VIPs terão um desconto de 20% a cada compra.

Os clientes esporádicos (PF) terão um desconto de 5%. Por sua vez, os clientes com CNPJ terão um desconto de 10%. O cupom de desconto deverá ser encapsulado em um método.

Você recebeu a missão de implementar esse primeiro esboço do sistema, tarefa para cuja execução os engenheiros de software lhe entregaram o diagrama de classes representado na figura abaixo. Agora é com você! Implemente a solução e apresente-a a equipe!

Esboço inicial do diagrama de classes. Fonte: elaborada pela autora.

**O que são objetos e o que as classes têm a ver com eles?**



Uma classe é uma abstração que descreve entidades do mundo real e quando instanciadas dão origem a objetos com características similares. Portanto, a classe é o modelo e o objeto é uma instância.

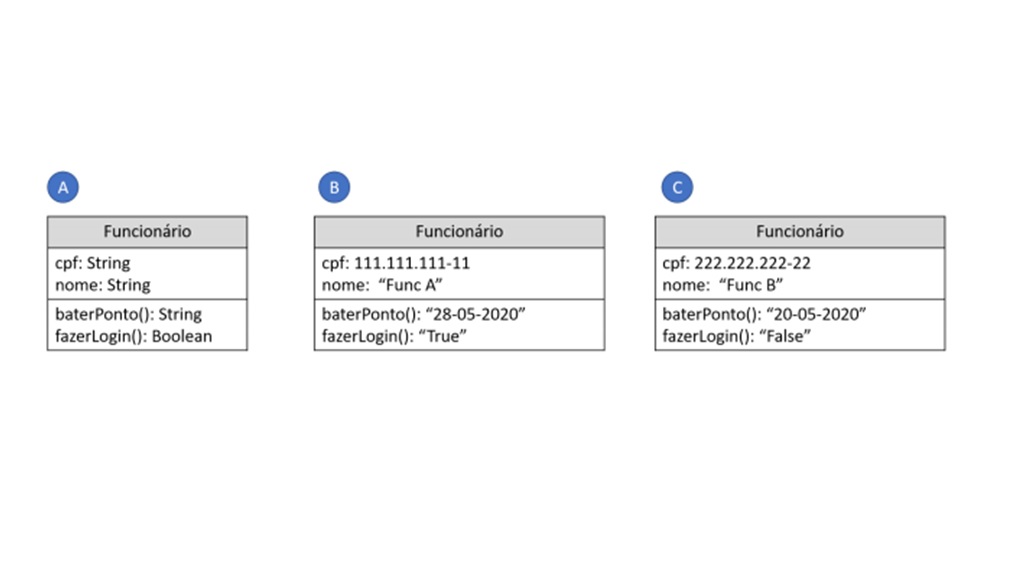
Vamos começar esta aula com alguns conceitos básicos do paradigma de programação orientado a objetos. Como nos lembra Weisfeld (2013), as tecnologias mudam muito rapidamente na indústria de software, ao passo que os conceitos evoluem. Portanto, é preciso conhecer os conceitos para então implementá-los na tecnologia adotada pela empresa (e essa tecnologia pode mudar rapidamente).

O desenvolvimento de software orientado a objetos (OO) existe desde o início dos anos 1960, mas, segundo Weisfeld (2013), foi somente em meados da década de 1990 que o paradigma orientado a objetos começou a ganhar impulso. De acordo com o mesmo autor, uma linguagem é entendida como orientada a objetos se ela aplica o conceito de abstração e suporta a implementação do encapsulamento, da herança e do polimorfismo. Mas, afinal, o que são objetos e o que as classes têm a ver com eles?

**Abstração - Classes e objetos**: **Objetos**são os componentes de um programa OO. Um programa que usa a tecnologia OO é basicamente uma coleção de objetos. Uma **classe**é um modelo para um objeto. Podemos considerar uma classe uma forma de organizar os dados (de um objeto) e seus comportamentos (PSF, 2020a). Vamos pensar na construção de uma casa: antes do "objeto casa" existir, um arquiteto fez a planta, determinando tudo que deveria fazer parte daquele objeto. Portanto, a **classe**é o modelo e o objeto é uma instância. Entende-se por instância a existência física, em memória, do objeto.

Para compreendermos a diferença entre classe e objeto, observe a figura abaixo. Usamos a notação gráfica oferecida pela UML (*Unified Modeling Language*) para criarmos um diagrama de classe. Cada diagrama de classes é definido por três seções separadas: o próprio nome da classe, os dados e os comportamentos.

Na figura (A), em que temos a classe *funcionário*, são especificados o que um funcionário deve ter. No nosso caso, como dados, ele deve ter um CPF e um nome e, como comportamento, ele deve bater ponto e fazer login. Agora veja a figura (B) e a figura (C) – esses dados estão "preenchidos", ou seja, foram instanciados e, portanto, são objetos.

Diagrama de classe: classe versus objeto. Fonte: elaborada pela autora.

Com base na figura acima, conseguimos definir mais sobre as classes e objetos: eles são compostos por dados e comportamento.

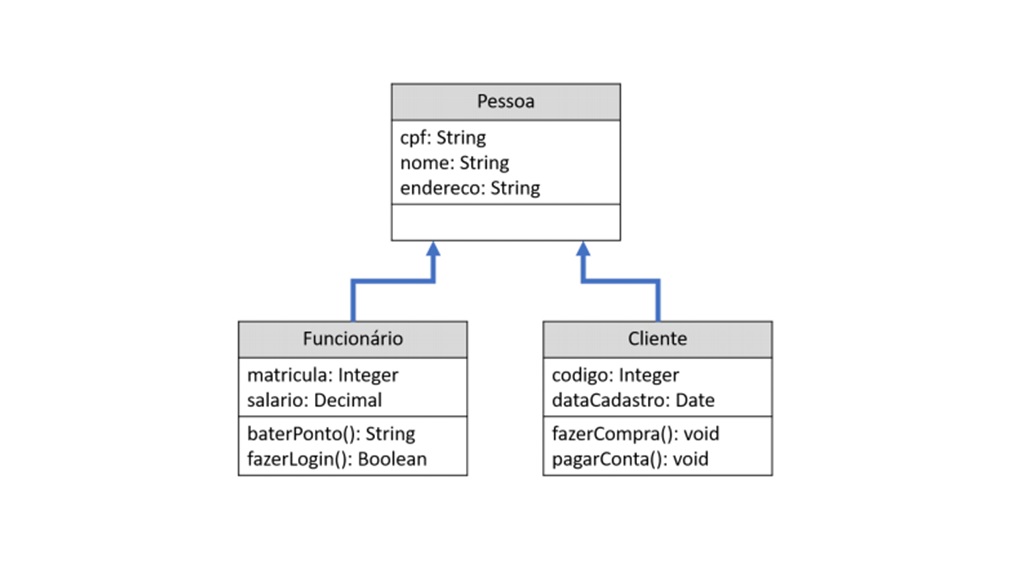
**Atributos**- Os dados armazenados em um objeto representam o estado do objeto. Na terminologia de programação OO, esses dados são chamados de **atributos**. Os atributos contêm as informações que diferenciam os vários objetos – os funcionários, neste caso.

**Métodos**- O comportamento de um objeto representa o que este pode fazer. Nas linguagens procedurais, o comportamento é definido por procedimentos, funções e sub-rotinas. Na terminologia de programação OO, esses comportamentos estão contidos nos **métodos**, aos quais você envia uma mensagem para invocá-los.

**Encapsulamento**- O ato de combinar os atributos e métodos na mesma entidade é, na linguagem OO, chamado de **encapsulamento**(Weisfeld, 2013), termo que também aparece na prática de tornar atributos privados, quando estes são encapsulados em métodos para guardar e acessar seus valores.

**Herança**- A figura abaixo utiliza um diagrama de classes para ilustrar outro importante conceito da OO, a herança. Por meio desse mecanismo, é possível fazer o reúso de código, criando soluções mais organizadas. A herança permite que uma classe herde os atributos e métodos de outra classe.

Observe, na figura, que as classes *funcionário*e *cliente*herdam os atributos da classe *pessoa*. A classe pessoa pode ser chamada de classe-pai, classe-base, superclasse, ancestral; por sua vez, as classes derivadas são as classes-filhas, subclasses.

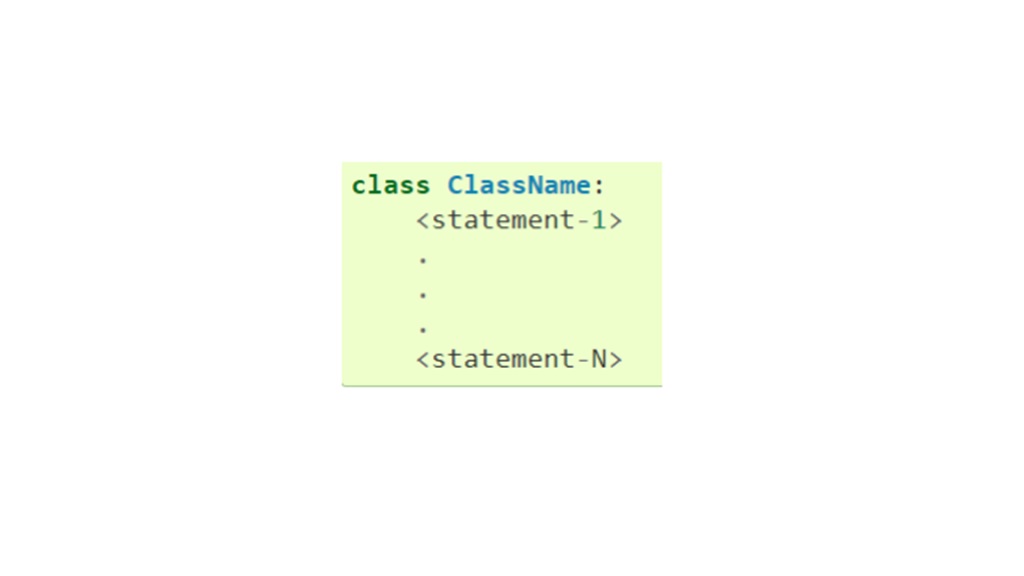
Diagrama de classe: herança. Fonte: elaborada pela autora.

**Poliformismo**- Para finalizar nossa introdução ao paradigma OO, vamos falar do *polimorfismo*. Polimorfismo é uma palavra grega que, literalmente, significa *muitas formas*. Embora o polimorfismo esteja fortemente associado à herança, é frequentemente citado separadamente como uma das vantagens mais poderosas das tecnologias orientadas a objetos (WEISFELD, 2013). Quando uma mensagem é enviada para um objeto, este deve ter um método definido para responder a essa mensagem. Em uma hierarquia de herança, todas as subclasses herdam as interfaces de sua superclasse. No entanto, como toda subclasse é uma entidade separada, cada uma delas pode exigir uma resposta separada para a mesma mensagem.

**A classe em Python**



Uma vez que suporta a implementação do encapsulamento, da herança e do polimorfismo, Python é uma linguagem com suporte ao paradigma orientado a objetos. A figura abaixo ilustra a sintaxe básica necessária para criar uma classe em Python. Utiliza-se a palavra reservada *class*para indicar a criação de uma classe, seguida do nome e dois pontos. No bloco indentado devem ser implementados os atributos e métodos da classe.

Sintaxe de classe em Python. Fonte: PSF (2020a, [s.p.]).

Veja a seguir a implementação da nossa primeira classe em Python.

In [1]: class PrimeiraClasse:

def imprimir\_mensagem(self, nome): # Criando um método print(f"Olá {nome}, seja bem vindo!")

In [2]: objeto1 = PrimeiraClasse() # Instanciando um objeto do tipo PrimeiraClasse objeto1.imprimir\_mensagem('João') # Invocando o método

Olá João, seja bem vindo!

Na entrada 1, criamos nossa primeira classe. Na linha 1 temos a declaração, e na linha 3 criamos um método para imprimir uma mensagem. **Todo método em uma classe deve receber como primeiro parâmetro uma variável que indica a referência à classe; por convenção, adota-se o parâmetro *self***. Conforme veremos, o parâmetro self será usado para acessar os atributos e métodos dentro da própria classe. Além do parâmetro obrigatório para métodos, estes devem receber um parâmetro que será usado na impressão da mensagem. Assim como acontece nas funções, um parâmetro no método é tratado como uma variável local.

Na entrada 2, criamos uma instância da classe na linha 1 (criamos nosso primeiro objeto!). A variável "objeto1" agora é um objeto do tipo *PrimeiraClasse*, razão pela qual pode acessar seus atributos e métodos. Na linha 2, invocamos o método *imprimir\_mensagem()*, passando como parâmetro o nome *João*. Por que passamos somente um parâmetro se o método escrito na entrada 1 espera dois parâmetros? O parâmetro *self*é a própria instância da classe e é passado de forma implícita pelo objeto. Ou seja, só precisamos passar explicitamente os demais parâmetros de um método.

Para acessarmos os recursos (atributos e métodos) de um objeto, após instanciá-lo, precisamos usar a seguinte sintaxe: objeto.recurso. Vale ressaltar que os atributos e métodos estáticos não precisam ser instanciados, mas esse é um assunto para outra hora.

\_\_\_\_\_\_\_

**📝 Exemplificando**

Vamos construir uma classe Calculadora, que implementa como métodos as quatro operações básicas matemáticas, além da operação de obter o resto da divisão.

In [3]: class Calculadora:

 def somar(self, n1, n2):

 return n1 + n2

 def subtrair(self, n1, n2):

 return n1 - n2

 def multiplicar(self, n1, n2):

 return n1 \* n2

 def dividir(self, n1, n2):

 return n1 / n2

 def dividir\_resto(self, n1, n2):

 return n1 % n2

In [4]: calc = Calculadora()

print('Soma:', calc.somar(4, 3))

print('Subtração:', calc.subtrair(13, 7))

print('Multiplicação:', calc.multiplicar(2, 4))

print('Divisão:', calc.dividir(16, 5))

print('Resto da divisão:', calc.dividir\_resto(7, 3))

Soma: 7

Subtração: 6

Multiplicação: 8

Divisão: 3.2

Resto da divisão: 1

Na entrada 3, em que implementamos os cinco métodos, veja que todos possuem o parâmetro self, pois são métodos da instância da classe. Cada um deles ainda recebe dois parâmetros que são duas variáveis locais nos métodos. Na entrada 4, instanciamos um objeto do tipo Calculadora e, nas linhas 3 a 7, acessamos seus métodos.

**Construtor da classe \_init\_()**



Até o momento criamos classes com métodos, os quais utilizam variáveis locais. E os atributos das classes? Nesta aula, vamos aprender a criar e utilizar **atributos de instância**, também chamadas de variáveis de instâncias. Esse tipo de atributo é capaz de receber um valor diferente para cada objeto. Um atributo de instância é uma variável precedida com o parâmetro self, ou seja, a sintaxe para criar e utilizar é self.nome\_atributo.

Ao instanciar um novo objeto, é possível determinar um estado inicial para variáveis de instâncias por meio do método construtor da classe. Em Python, o método construtor é chamado de \_\_*init\_\_()*. Veja o código a seguir.

In [5]: class Televisao:

 def \_\_init\_\_(self):

 self.volume = 10

 def aumentar\_volume(self):

 self.volume += 1

 def diminuir\_volume(self):

 self.volume -= 1

In [6]: tv = Televisao()

print("Volume ao ligar a tv = ", tv.volume)

tv.aumentar\_volume()

print("Volume atual = ", tv.volume)

Volume ao ligar a tv = 10

Volume atual = 11

Na entrada 5, criamos a classe *Televisao*, que possui um atributo de instância e três métodos, o primeiro dos quais é (\_*\_init*\_\_), aquele que é invocado quando o objeto é instanciado. Nesse método construtor, instanciamos o atributo volume com o valor 10, ou seja, todo objeto do tipo *Televisao*será criado com volume = 10. Veja que o atributo recebe o prefixo *self*., que o identifica como variável de instância. Esse tipo de atributo pode ser usado em qualquer método da classe, uma vez que é um atributo do objeto, eliminando a necessidade de passar parâmetros para os métodos. Nos métodos *aumentar\_volume* e *diminuir\_volume* alteramos esse atributo sem precisar passá-lo como parâmetro, já que é uma variável do objeto, e não uma variável local.

**Variáveis e métodos privados** - Em linguagens de programação OO, como Java e C#, as classes, os atributos e os métodos são acompanhados de modificadores de acesso, que podem ser: public, private e protected. Em Python, não existem modificadores de acesso e todos os recursos são públicos. Para simbolizar que um atributo ou método é privado, por convenção, usa-se um sublinhado "\_"  antes do nome; por exemplo, \_cpf, \_calcular\_desconto() (PSF, 2020a).

Conceitualmente, dado que um atributo é privado, ele só pode ser acessado por membros da própria classe. Portanto, ao declarar um atributo privado, precisamos de métodos que acessem e recuperam os valores ali guardados. Em Python, além de métodos para este fim, um atributo privado pode ser acessado por decorators. Embora a explicação sobre estes fuja ao escopo desta aula, recomendamos a leitura do portal Python Course (2020). Observe o código a seguir, em cuja entrada 7, o atributo \_*saldo* é acessado pelos métodos *depositar()* e *consultar\_saldo()*.

In [7]: class ContaCorrente:

 def \_\_init\_\_(self):

 self.\_saldo = None

 def depositar(self, valor):

 self.\_saldo += valor

 def consultar\_saldo(self):

 return self.\_sald

Na entrada 7, implementamos a classe *ContaCorrente*, que possui dois atributos privados: \_cpf e \_saldo. Para guardar um valor no atributo *cpf*, deve-se chamar o método *set\_cpf*, e, para recuperar seu valor, usa-se *get\_cpf*. Para guardar um valor no atributo *saldo*, é preciso invocar o método *depositar()*, e, para recuperar seu valor, deve-se usar o método *get\_saldo*. Lembre-se: em Python, atributos e métodos privados são apenas uma convenção, pois, na prática, os recursos podem ser acessados de qualquer forma.

**Herança em Python**



Como vimos, um dos pilares da OO é a reutilização de código por meio da herança, que permite que uma classe-filha herde os recursos da classe-pai. Em Python, uma classe aceita múltiplas heranças, ou seja, herda recursos de diversas classes. A sintaxe para criar a herança é feita com parênteses após o nome da classe: class NomeClasseFilha(NomeClassePai). Se for uma herança múltipla, cada superclasse deve ser separada por vírgula.

Para nosso primeiro exemplo sobre herança em Python, vamos fazer a implementação do diagrama de classes da figura (Sintaxe de classe em Python). Observe o código a seguir. Na entrada 8, criamos a classe *pessoa*com três atributos que são comuns a todas pessoas da nossa solução. Na entrada 9, criamos a classe *funcionario*, que herda todos os recursos da classe *pessoa*, razão pela qual dizemos que "**um funcionário é uma pessoa**". Na entrada 10, criamos a classe *cliente*, que também herda os recursos da classe *pessoa*, logo, "**um cliente é uma pessoa**".

In [8]: class Pessoa:

 def \_\_init\_\_(self):

 self.cpf = None

 self.nome = None

 self.endereco = None

In [9]: class Funcionario(Pessoa):

 def \_\_init\_\_(self):

 self.matricula = None

 self.salacio = None

 def bater\_ponto(self):

 # código aqui

 pass

 def fazer\_login(self):

 # código aqui

 pass

In [10]: class Cliente(Pessoa):

 def \_\_init\_\_(self):

 self.codigo = None

 self.dataCadastro = None

 def fazer\_compra(self):

 # código aqui

 pass

 def pagar\_conta(self):

 # código aqui

 pass

In [11]: f1 = Funcionario()

f1.nome = "Funcionário A"

print(f1.nome)

c1 = Cliente()

c1.cpf = "111.111.111-11"

print(c1.cpf)

Funcionário A

111.111.111-11

Na entrada 11, podemos ver o resultado da herança. Na linha 1, instanciamos um objeto do tipo *funcionário*, atribuindo à variável *nome*o valor "Funcionário A". O atributo *nome*foi herdado da classe-pai. Veja, no entanto, que o utilizamos normalmente, pois, na verdade, a partir de então esse atributo faz parte da classe mais especializada. O mesmo acontece para o atributo *cpf*, usado no objeto c1 que é do tipo *cliente*.

**Métodos mágicos em Python** - Estamos usando herança desde a criação da nossa primeira classe nesta aula. Não sabíamos disso porque a herança estava sendo feita de modo implícito. Quando uma classe é criada em Python, ela herda, mesmo que não declarado explicitamente, todos os recursos de uma classe-base chamada *object*. Veja o resultado da função *dir()*, que retorna uma lista com os recursos de um objeto. Como é possível observar na saída 12 (Out[12]), a classe *Pessoa*, que explicitamente não tem nenhuma herança, possui uma série de recursos nos quais os nomes estão com *underline*(sublinhado). Todos eles são chamados de métodos mágicos e, com a herança, podem ser sobrescritos, conforme veremos a seguir.

In [12]: dir(Pessoa())

Out[12] ['\_\_class\_\_',

 '\_\_delattr\_\_',

 '\_\_dict\_\_',

 '\_\_dir\_\_',

 '\_\_doc\_\_',

 '\_\_eq\_\_',

 '\_\_format\_\_',

 '\_\_ge\_\_',

 '\_\_getattribute\_\_',

 '\_\_gt\_\_',

 '\_\_hash\_\_',

 '\_\_init\_\_',

 '\_\_init\_subclass\_\_',

 '\_\_le\_\_',

 '\_\_lt\_\_',

 '\_\_module\_\_',

 '\_\_ne\_\_',

 '\_\_new\_\_',

 '\_\_reduce\_\_',

 '\_\_reduce\_ex\_\_',

 '\_\_repr\_\_',

 '\_\_setattr\_\_',

 '\_\_sizeof\_\_',

 '\_\_str\_\_',

 '\_\_subclasshook\_\_',

 '\_\_weakref\_\_',

 'cpf',

 'endereco',

 'nome']

**Método construtor na herança e sobrescrita**



Na herança, quando adicionamos a função \_\_*init\_\_(),* a classe-filho não herdará o construtor dos pais. Ou seja, o construtor da classe-filho sobrescreve (override) o da classe-pai. Para utilizar o construtor da classe-base, é necessário invocá-lo explicitamente, dentro do construtor-filho, da seguinte forma: ClassePai.\_\_init\_\_(). Para entendermos como funciona a sobrescrita do método construtor e o uso de métodos mágicos, vamos implementar nossa própria versão da classe *'int*', que é usada para instanciar objetos inteiros. Observe o código a seguir.

In [13] class int42(int):

 def \_\_init\_\_(self, n):

 int.\_\_init\_\_(n)

 def \_\_add\_\_(a, b):

 return 42

 def \_\_str\_\_(n):

 return '42'

In [14] a = int42(7)

b = int42(13)

print(a + b)

print(a)

print(b)

42

42

42

In [15] c = int(7)

d = int(13)

print(c + d)

print(c)

print(d)

20

7

13

Na entrada 13, temos a classe-filho *int42*, que tem como superclasse a classe int. Veja que, na linha 3, definimos o construtor da nossa classe, mas, na linha 4, fazemos com que nosso construtor seja sobrescrito pelo construtor da classe-base, o qual espera receber um valor. O método *construtor*é um método mágico, assim como o \_\_add\_\_ e o \_\_str\_\_. O primeiro retorna a soma de dois valores, mas, na nossa classe int42, ele foi sobrescrito para sempre retornar o mesmo valor. O segundo método, \_\_str\_\_, retorna uma representação de string do objeto, e, na nossa classe, sobrescrevemos para sempre imprimir 42 como a string de representação do objeto, que será exibida sempre que a função *print()* for invocada para o objeto.

Na entrada 14, podemos conferir o efeito de substituir os métodos mágicos. Nossa classe *int42* recebe como parâmetro um valor, uma vez que estamos usando o construtor da classe-base *int*. O método \_*\_add*\_\_, usado na linha 3, ao invés de somar os valores, simplesmente retorna o valor 42, pois o construímos assim. Nas linhas 4 e 5 podemos ver o efeito do método \_\_str\_\_, que também foi sobrescrito para sempre imprimir 42.

Na entrada 15, usamos a classe original *int* para você poder perceber a diferença no comportamento quando sobrescrevemos os métodos mágicos.

Ao sobrescrever os métodos mágicos, utilizamos outra importante técnica da OO, o **polimorfismo**. Essa técnica, vale dizer, pode ser utilizada em qualquer método, não somente nos mágicos. Construir métodos com diferentes comportamentos pode ser feito sobrescrevendo (**override**) ou sobrecarregando (**overload**) métodos. No primeiro caso, a classe-filho sobrescreve um método da classe-base, por exemplo, o construtor, ou qualquer outro método. No segundo caso, da sobrecarga, um método é escrito com diferentes assinaturas para suportar diferentes comportamentos.

Em Python, graças à sua natureza de interpretação e tipagem dinâmica, a operação de sobrecarga não é suportada de forma direta, o que significa que não conseguimos escrever métodos com os mesmos nomes, mas diferentes parâmetros (RAMALHO, 2014). Para contornar essa característica, podemos escrever o método com parâmetros default. Assim, a depender dos que forem passados, mediante estruturas condicionais, o método pode ter diferentes comportamentos ou fazer o overload com a utilização do decorator*functools.singledispatch*, cuja explicação foge ao escopo e propósito desta seção [no entanto, recomendamos, para aprofundamento sobre o assunto, a leitura das páginas 202 a 205 da obra de Ramalho (2014), além da documentação oficial do Python (que você pode encontrar [aqui](https://docs.python.org/pt-br/3.7/library/functools.html" \t "_blank)).

**Herança múltipla**



Python permite que uma classe-filha herde recursos de mais de uma superclasse. Para isso, basta declarar cada classe a ser herdada separada por vírgula. A seguir temos uma série de classes. A classe *PCIEthernet* herda recursos das classes *PCI*e *Ethernet*; logo, PCIEthernet é uma PCI e também uma Ethernet. A classe *USBWireless* herda de USB e Wireless, mas Wireless herda de Ethernet; logo, USBWireless é uma USB, uma Wireless e também uma Ethernet, o que pode ser confirmado pelos resultados da função built-in isinstance(objeto, classe), que checa se um objeto é uma instância de uma determinada classe.

In [16] class Ethernet():

 def \_\_init\_\_(self, name, mac\_address):

 self.name = name

 self.mac\_address = mac\_address

class PCI():

 def \_\_init\_\_(self, bus, vendor):

 self.bus = bus

 self.vendor = vendor

class USB():

 def \_\_init\_\_(self, device):

 self.device = device

class Wireless(Ethernet):

 def \_\_init\_\_(self, name, mac\_address):

 Ethernet.\_\_init\_\_(self, name, mac\_address)

class PCIEthernet(PCI, Ethernet):

 def \_\_init\_\_(self, bus, vendor, name, mac\_address):

 PCI.\_\_init\_\_(self, bus, vendor)

 Ethernet.\_\_init\_\_(self, name, mac\_address)

class USBWireless(USB, Wireless):

 def \_\_init\_\_(self, device, name, mac\_address):

 USB.\_\_init\_\_(self, device)

 Wireless.\_\_init\_\_(self, name, mac\_address)

eth0 = PCIEthernet('pci :0:0:1', 'realtek', 'eth0', '00:11:22:33:44')

wlan0 = USBWireless('usb0', 'wlan0', '00:33:44:55:66')

print('PCIEthernet é uma PCI?', isinstance(eth0, PCI))

print('PCIEthernet é uma Ethernet?', isinstance(eth0, Ethernet))

print('PCIEthernet é uma USB?', isinstance(eth0, USB))

print('\nUSBWireless é uma USB?', isinstance(wlan0, USB))

print('USBWireless é uma Wireless?', isinstance(wlan0, Wireless))

print('USBWireless é uma Ethernet?', isinstance(wlan0, Ethernet))

print('USBWireless é uma PCI?', isinstance(wlan0, PCI))

PCIEthernet é uma PCI? True

PCIEthernet é uma Ethernet? True

PCIEthernet é uma USB? False

USBWireless é uma USB? True

USBWireless é uma Wireless? True

USBWireless é uma Ethernet? True

USBWireless é uma PCI? False

\_\_\_\_\_\_\_

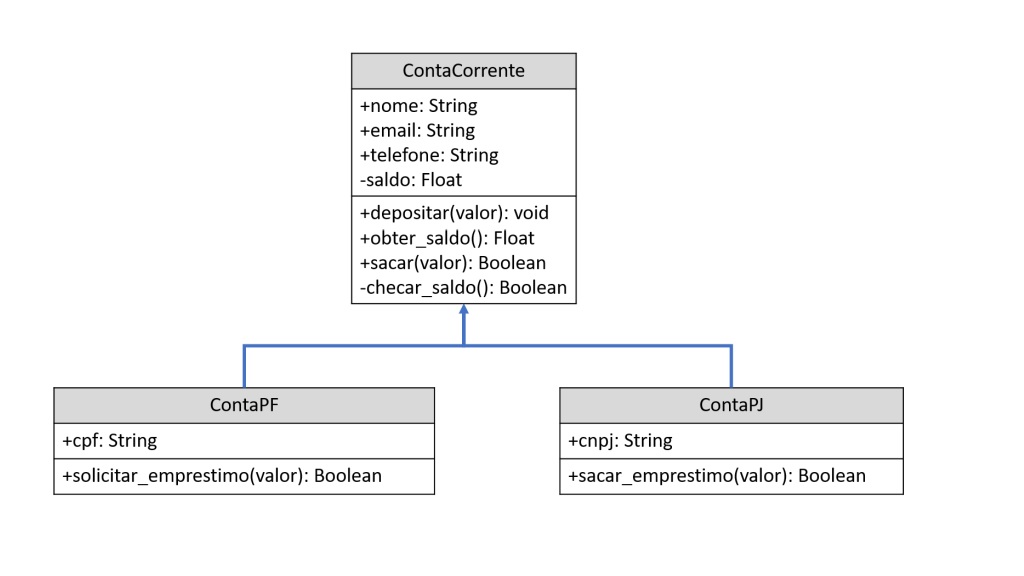
**📝 Exemplificando**

Para treinar tudo o que aprendemos nessa aula, vamos implementar a solução representada no diagrama de classes da figura abaixo, na qual temos uma classe-base chamada *ContaCorrente* com os seguintes campos:

* Nome, que é do tipo *string*e deve ser público.
* Email, que é do tipo *string*e deve ser público.
* Telefone, que é do tipo *string*e deve ser público.
* Saldo, que é do tipo ponto *flutuante*e deve ser público.

A classe-base conta ainda com os métodos:

* *Depositar*, que recebe como parâmetro um valor, não retorna nada e deve ser público.
* *Obter saldo*, que não recebe nenhum parâmetro, retorna um ponto flutuante e deve ser público.
* *Sacar*, que recebe como parâmetro um valor, retorna se foi possível sacar (booleano) e deve ser público.
* *Checar saldo*, que não recebe nenhum parâmetro, retorna um booleano e deve ser privado, ou seja, será usado internamente pela própria classe.

Diagrama de classe ContaCorrente. Fonte: elaborada pela autora.

Na entrada 17, implementamos a classe *ContaCorrente*. Na linha 2, especificamos que a classe deve ser construída recebendo como parâmetro um nome (se o nome não for passado, o objeto não é instanciado). Dentro do construtor, criamos os atributos e os inicializamos. Veja que a variável *\_saldo* recebe um sublinhado (underline) como prefixo para indicar que é uma variável privada e deve ser acessada somente por membros da classe. O valor desse atributo deve ser alterado pelos métodos depositar e sacar, mas veja que o atributo é usado para *checar o saldo* (linha 9) e também no método de consulta ao saldo (linha 25).

O método *\_checar\_saldo()* é privado e, por isso, recebe o prefixo de sublinhado (underline) "\_". Tal método é usado como um dos passos para a realização do saque, pois esta só pode acontecer se houver saldo suficiente. Veja que, na linha 18, antes de fazermos o saque, invocamos o método *self.\_checar\_saldo(valor)*– o *self*indica que o método pertence à classe.

In [17] class ContaCorrente:

 def \_\_init\_\_(self, nome):

 self.nome = nome

 self.email = None

 self.telefone = None

 self.\_saldo = 0

 def \_checar\_saldo(self, valor):

 return self.\_saldo >= valor

 def depositar(self, valor):

self.\_saldo += valor

 def sacar(self, valor):

 if self.\_checar\_saldo(valor):

 self.\_saldo -= valor

 return True

 else:

 return False

 def obter\_saldo(self):

 return self.\_saldo

Na entrada 18, criamos a classe *ContaPF*, que herda todas as funcionalidades da classe-pai *ContaCorrente*, inclusive seu construtor (linha 3). Veja, na linha 2, que o objeto deve ser instanciado passando o nome e o cpf como parâmetros – o nome faz parte do construtor-base. Além dos recursos herdados, criamos o atributo *cpf*e o método *solicitar\_emprestimo*, que consulta o saldo para aprovar ou não o empréstimo.

In [18] class ContaPF(ContaCorrente):

 def \_\_init\_\_(self, nome, cpf):

 super().\_\_init\_\_(nome)

 self.cpf = cpf

 def solicitar\_emprestimo(self, valor):

 return self.obter\_saldo() >= 500

Na entrada 19, criamos a classe *ContaPJ*, que herda todas as funcionalidades da classe-pai *ContaCorrente*, inclusive seu construtor (linha 3). Veja, na linha 2, que o objeto deve ser instanciado passando o nome e o cnpj como parâmetros – o nome faz parte do construtor-base. Além dos recursos herdados, criamos o atributo *cnpj*e o método *sacar\_emprestimo*, que verifica se o valor solicitado é inferior a 5000. Observe que, para esse valor, não usamos o parâmetro *self*, pois se trata de uma variável local, não um atributo de classe ou instância.

In [19] class ContaPJ(ContaCorrente):

 def \_\_init\_\_(self, nome, cnpj):

 super().\_\_init\_\_(nome)

 self.cnpj = cnpj

 def sacar\_emprestimo(self, valor):

 return valor <= 5000

Na entrada 20, criamos um objeto do tipo *ContaPF* para testar as funcionalidades implementadas. Veja que instanciamos com os valores necessários. Na linha 2, fazemos um depósito, na linha 3, consultamos o saldo e, na linha 4, solicitamos um empréstimo. Como há saldo suficiente, o empréstimo é aprovado. Na segunda parte dos testes, realizamos um saque, pedimos para imprimir o restante e solicitamos novamente um empréstimo, o qual, neste momento, é negado, pois não há saldo suficiente.

In [20] conta\_pf1 = ContaPF("João", '111.111.111-11')

conta\_pf1.depositar(1000)

print('Saldo atual é', conta\_pf1.obter\_saldo())

print('Receberá empréstimo = ', conta\_pf1.solicitar\_emprestimo(2000))

conta\_pf1.sacar(800)

print('Saldo atual é', conta\_pf1.obter\_saldo())

print('Receberá empréstimo = ', conta\_pf1.solicitar\_emprestimo(2000))

Saldo atual é 1000

Receberá empréstimo = True

Saldo atual é 200

Receberá empréstimo = False

Na entrada 21, criamos um objeto do tipo *ContaPJ* para testar as funcionalidades implementadas. Veja que instanciamos com os valores necessários. Na linha 2, consultamos o saldo e na linha 3 solicitamos o saque de um empréstimo. Mesmo não havendo saldo para o cliente, uma vez que a regra do empréstimo não depende desse atributo, o saque é permitido.

In [21] conta\_pj1 = ContaPJ("Empresa A", "11.111.111/1111-11")

print('Saldo atual é', conta\_pj1.obter\_saldo())

print('Receberá empréstimo = ', conta\_pj1.sacar\_emprestimo(3000))

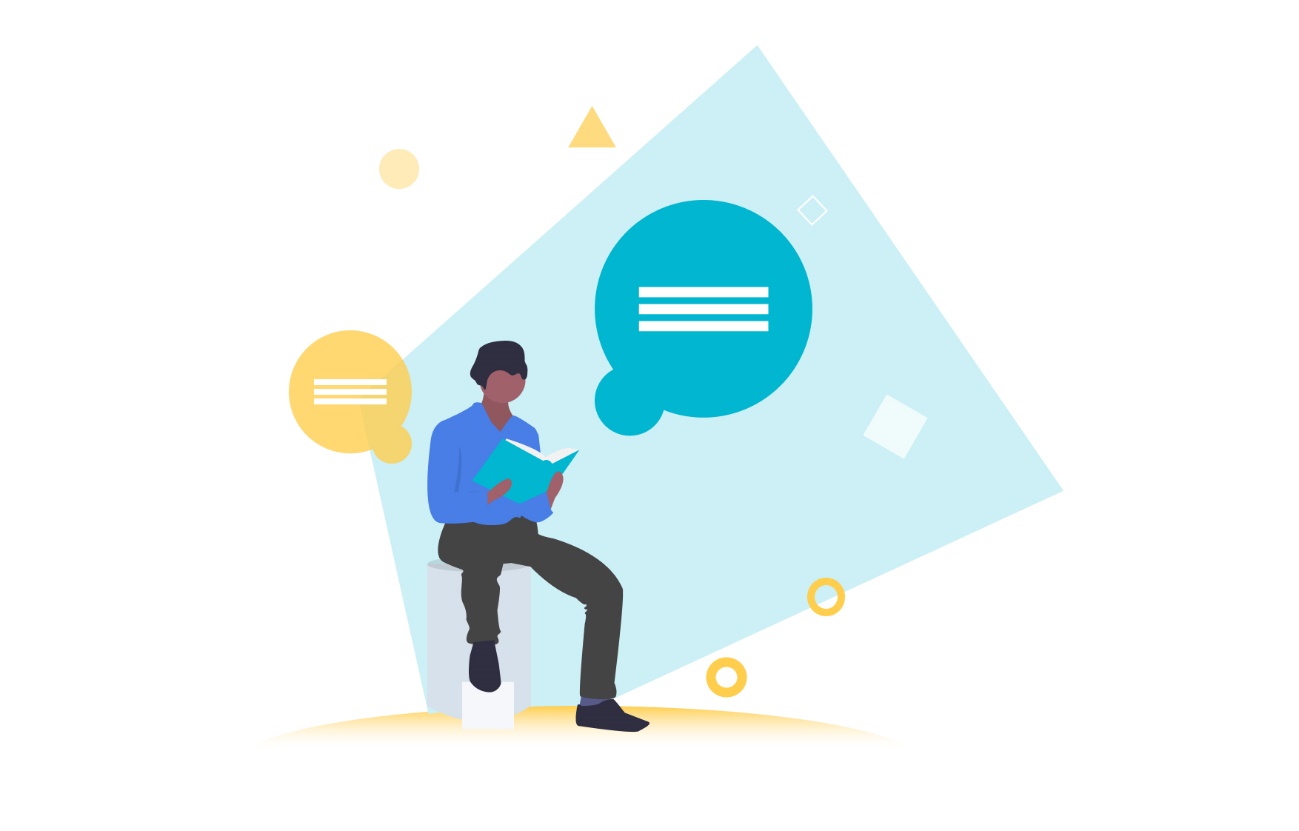
Saldo atual é 0

Receberá empréstimo = True

\_\_\_\_\_\_\_

Que tal testar todas essas implementações e funcionalidades na ferramenta [Python Tutor](http://pythontutor.com/)? Aproveite a ferramenta e explore novas possibilidades.

**Conclusão**



Para este desafio, será necessário implementar quatro classes: uma base e três subclasses, que herdam os recursos. Todos os atributos que são comuns às classes devem ficar na classe-base. Por sua vez, os atributos específicos devem ser implementados nas classes-filhas.

A quantidade de itens que cada cliente pode comprar é diferente, razão pela qual o método *realizar\_compra*precisa ser sobrescrito em cada subclasse para atender a essa especificidade. O cupom de desconto também varia de acordo com o tipo de cliente. Observe, a seguir, uma possível implementação para a solução.

In [22]: class Cliente:

 def \_\_init\_\_(self):

 self.nome = None

 self.emil = None

 self.telefone = None

 self.\_cupom\_desconto = 0

 def get\_cupom\_desconto(self):

 return self.\_cupom\_desconto

 def realizar\_compras(self, lista\_itens):

 pass

In [23] class ClienteVipPF(Cliente):

 def \_\_init\_\_(self):

 super().\_\_init\_\_()

 self.\_cupom\_desconto = 0.2

 def realizar\_compras(self, lista\_itens):

 return f"Quantidade total de itens comprados = {len(lista\_itens)}"

In [24] class ClientePF(Cliente):

 def \_\_init\_\_(self):

 super().\_\_init\_\_()

 self.\_cupom\_desconto = 0.05

 def realizar\_compras(self, lista\_itens):

 if len(lista\_itens) <= 20:

 return f"Quantidade total de itens comprados = {len(lista\_itens)}"

 else:

 return "Quantidade de itens superior ao limite permitido."

In [25] class ClientePJ(Cliente):

 def \_\_init\_\_(self):

 super().\_\_init\_\_()

 self.\_cupom\_desconto = 0.1

 def realizar\_compras(self, lista\_itens):

 if len(lista\_itens) <= 50:

 return f"Quantidade total de itens comprados = {len(lista\_itens)}"

 else:

 return "Quantidade de itens superior ao limite permitido."

In [26] cli1 = ClienteVipPF()

cli1.nome = "Maria"

print(cli1.get\_cupom\_desconto())

cli1.realizar\_compras(['item1', 'item2', 'item3'])

0.2

Out[26]:'Quantidade total de itens comprados = 3'

Implementamos as quatro classes necessárias. Veja que em cada classe-filha, determinados o valor do cupom de desconto logo após invocar o construtor da classe-base. Em cada subclasse também sobrescrevemos o método *realizar\_compras* para atender aos requisitos específicos.

Conforme você pode notar, a orientação a objetos permite entregar uma solução mais organizada, com reaproveitamento de código, o que certamente pode facilitar a manutenção e a inclusão de novas funcionalidades. Além disso, utilizar o diagrama de classes para se comunicar com outros membros da equipe tem grande valor no projeto de desenvolvimento de um software.

Utilize o emulador [Python Tutor](http://pythontutor.com/) para testar a implementação e fazer novos testes!