NÃO PODE FALTAR

CLASSES E MÉTODOS EM PYTHON

Vanessa Cadan Scheffer

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.

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

INTRODUÇÃO A ORIENTAÇÃO A OBJETOS

Vamos começar esta seção 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 do 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?

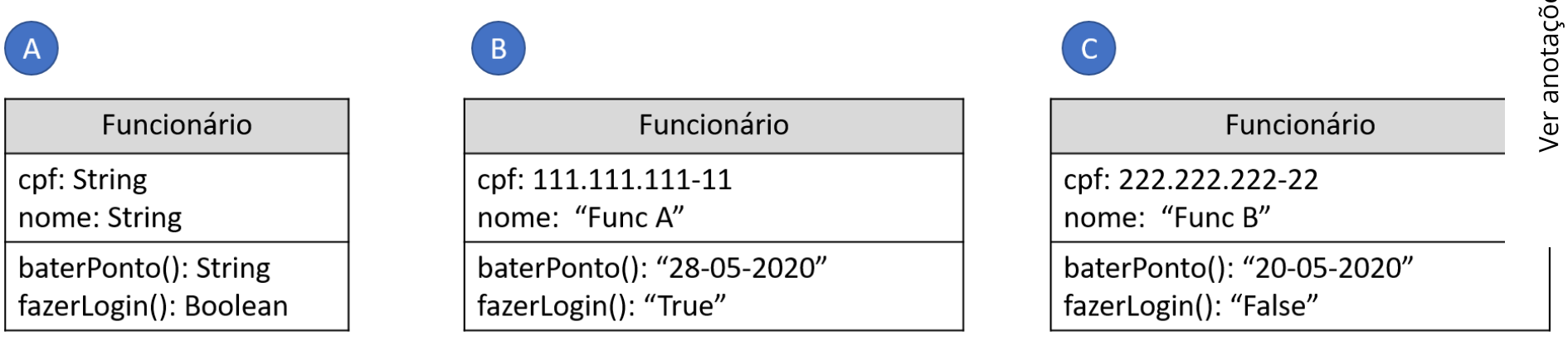
ABSTRAÇÃO - CLASSES E OBJETOS

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 3.1. 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 3.1(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 3.1(B) e a Figura 3.1(C) – esses dados estão "preenchidos", ou seja, foram instanciados e, portanto, são objetos.

Figura 3.1 - Diagrama de classe: classe *versus* objeto



Fonte: elaborada pela autora.

Com base na Figura 3.1, 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

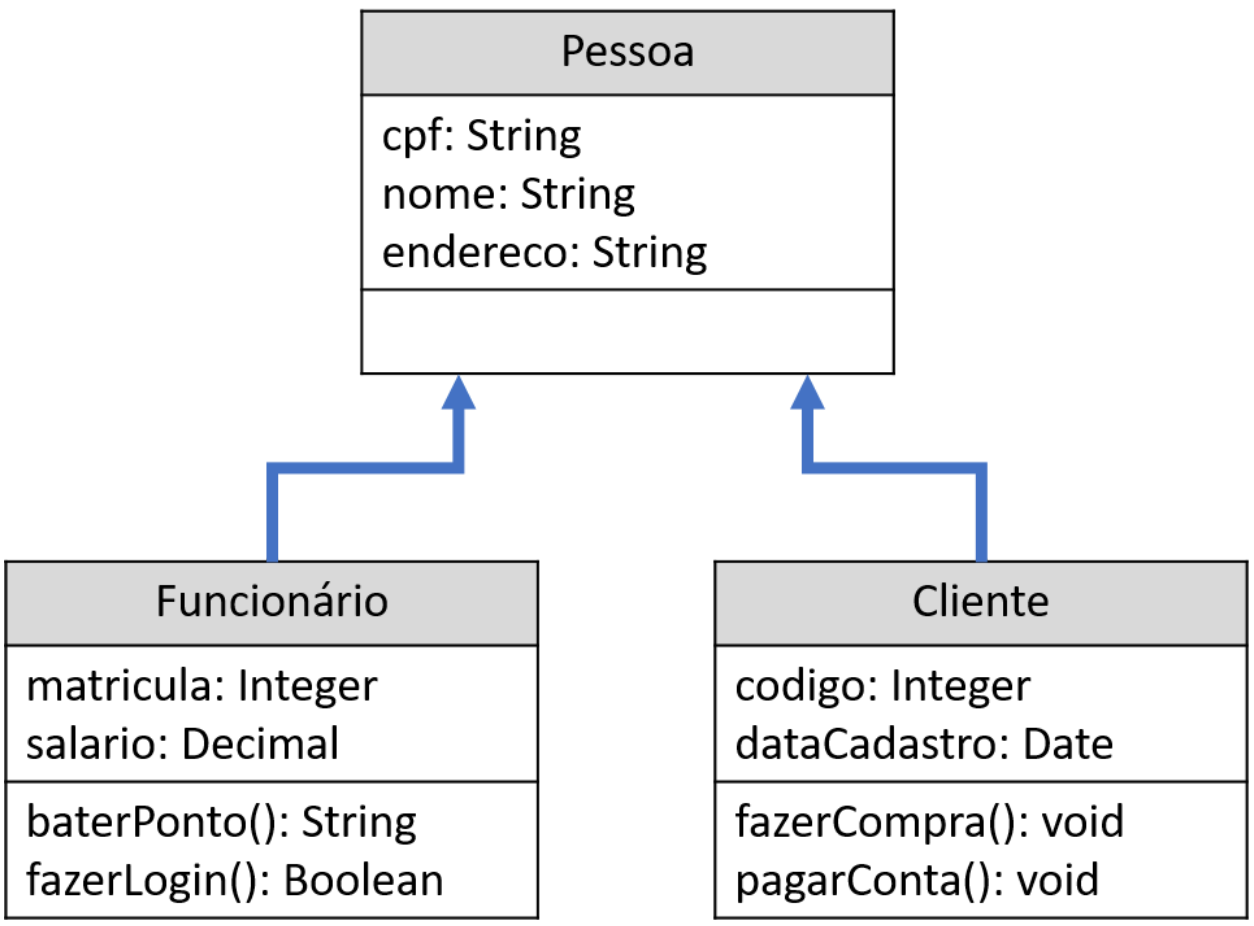
O ato de combinar os atributos e métodos na mesma entidade é, na linguagem OO, chamado de **encapsulamento** (Weisfeld, 2013), termo que também aparece na prática de tornar atributos privados, quando estes são encapsulados em

métodos para guardar e acessar seus valores.

HERANÇA

A Figura 3.2 utiliza um diagrama de classes para ilustrar outro importante conceit 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 3.2, 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.

Figura 3.2 | 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 ur delas pode exigir uma resposta separada para a mesma mensagem.

A CLASSES 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 3.3 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 o atributos e métodos da classe.

Figura 3.3 | 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 seção, 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 = 1 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 seção, 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 C*ontaCorrente*, 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 pode 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 3.2. 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. Nã 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']
```

Ver anotações 0

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 versã 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
```

Ver anotações 0

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 encontra em: https://docs.python.org/pt-br/3.7/library/functools.html)**.**

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

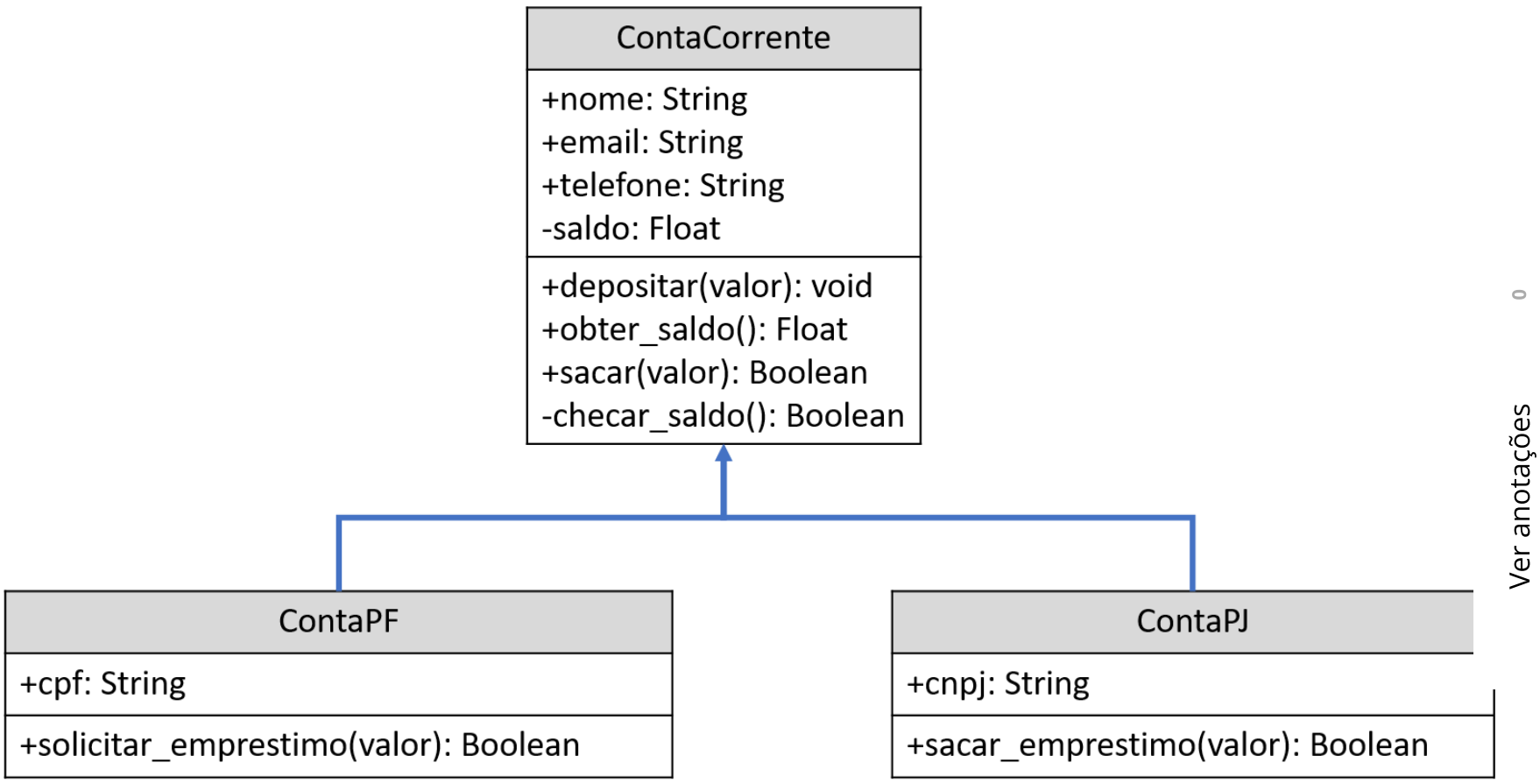
Para treinar tudo o que aprendemos nessa seção, vamos implementar a solução representada no diagrama de classes da Figura 3.4, na qual temos uma classe-base chamada *ContaCorrente* com os seguintes campos:

- 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.

Figura 3.4 | 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
```

Ver anotações 0

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
```

Ver anotações 0

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? Aproveite a ferramenta e explore novas possibilidades.

REFERÊNCIAS E LINKS ÚTEIS

LJUBOMIR, P. **Introdução à computação usando Python**: um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

PSF. Python Software Fundation. **Classes**. 2020a. Disponível em: https://bit.ly/3fSURVx. Acesso em: 10 maio 2020.

PYTHON COURSE. Properties vs. Getters and Setters. Python 3 Tutorial. 2020. Disponível em: https://bit.ly/2XWY9Rn. Acesso em: 14 jun. 2020

RAMALHO, L. **Fluent Python**. Gravenstein: O'Reilly Media, 2014.

WEISFELD, M. A. **The Object-Oriented:** Thought Process. 4. ed. [*S.l.*]: Addison-Wesley Professional, 2013.