

CAPÍTULO 10

Classes e objetos

A programação orientada a objetos facilita a escrita e manutenção de nossos programas, utilizando classes e objetos. Classes são a definição de um novo tipo de dados que associa dados e operações em uma só estrutura. Um objeto pode ser entendido como uma variável cujo tipo é uma classe, ou seja, um objeto é uma instância de uma classe.

A programação orientada a objetos é uma técnica de programação que organiza nossos programas em classes e objetos em vez de apenas funções, como vimos até agora. É um assunto muito importante e extenso, merecendo vários livros e muita prática para ser completamente entendido. O objetivo deste capítulo é apresentar o básico da orientação a objetos de forma a introduzir o conceito e estimular o aprendizado dessa técnica.

10.1 Objetos como representação do mundo real

Podemos entender um objeto em Python como a representação de um objeto do mundo real, escrita em uma linguagem de programação. Essa representação é limitada pela quantidade de detalhes que podemos ou queremos representar, uma abstração.

Vejamos, por exemplo, um aparelho de televisão. Podemos dizer que uma televisão tem uma marca e um tamanho de tela. Podemos também pensar no que podemos fazer com esse aparelho, por exemplo mudar de canal, ligá-lo ou desligá-lo. Vejamos como escrever isso em Python na listagem 10.1.

► Listagem 10.1 – Modelagem de uma televisão

```
>>> class Televisão: ❶
    def __init__(self): ❷
        self.ligada=False ❸
        self.canal=2 ❹
    >>> tv = Televisão() ❺
    >>> tv.ligada ❻
False
    >>> tv.canal
2
    >>> tv_sala=Televisão() ❻
    >>> tv_sala.ligada=True ❾
    >>> tv_sala.canal=4 ❿
    >>> tv.sala
2
    >>> tv_sala.canal
4
```

Em ❶, criamos uma nova classe chamada `Televisão`. Utilizamos a instrução `class` para indicar a declaração de uma nova classe e `:` para iniciar seu bloco. Quando declaramos uma classe, estamos criando um novo tipo de dados. Esse novo tipo define seus próprios métodos e atributos. Lembre-se dos tipos `string` e `list`. Esses dois tipos predefinidos do Python são classes. Quando criamos uma lista ou uma `string`, estamos instanciando ou criando uma instância dessas classes, ou seja, um objeto. Quando definimos nossas próprias classes, podemos criar nossos próprios métodos e atributos.

Em ❷, definimos um método especial chamado `__init__`. Métodos nada mais são que funções associadas a uma classe. O método `__init__` será chamado sempre que criarmos objetos da classe `Televisão`, sendo por isso chamado de construtor (*constructor*). Um método construtor é chamado sempre que um objeto da classe é instanciado. É o construtor que inicializa nosso novo objeto com seus valores-padrão. O método `__init__` recebe um parâmetro chamado `self`. Por enquanto, entenda `self` como o objeto televisão em si, o que ficará mais claro adiante.

Em ❸, dizemos que `self.ligada` é um valor de `self`, ou seja, do objeto televisão. Todo método em Python tem `self` como primeiro parâmetro. Dizemos que `self.ligada` é um atributo do objeto. Como `self` representa o objeto em si, escreveremos `self.ligada`. Sempre que quisermos especificar atributos de objetos,

devemos associá-los a `self`. Caso contrário, se escrevéssemos `ligada=False`, ligada seria apenas uma variável local do método `_init_`, e não um atributo do objeto. Em ④, dizemos que `canal` também é um valor ou característica de nossa televisão. Observe também que escrevemos `self.canal` para criar um atributo, e não uma simples variável local.

Em ⑤ criamos um objeto `tv` utilizando a classe `Televisão`. Dizemos que `tv` é agora um objeto da classe `Televisão` ou que `tv` é uma instância de `Televisão`. Quando solicitamos a criação de um objeto, o método construtor de sua classe é chamado, em Python, `_init_`, como declaramos em ②. Em ⑥, exibimos o valor do atributo `ligada` e `canal` do objeto `tv`.

Já em ⑦ criamos outra instância da classe `Televisão` chamada `tv_sala`. Em ⑧, mudamos o valor de `ligada` para `True` e o canal para 4 em ⑨.

Observe que ao imprimirmos o canal de cada televisão temos valores independentes, pois `tv` e `tv_sala` são dois objetos independentes, podendo cada um ter seus próprios valores, como duas televisões no mundo real. Quando criamos um objeto de uma classe, ele tem todos os atributos e métodos que especificamos ao declarar a classe e que foram inicializados em seu construtor. Essa característica simplifica o desenvolvimento dos programas, pois podemos definir o comportamento de todos os objetos de uma classe (métodos), preservando os valores individuais de cada um (atributos).

Exercício 10.1 Adicione os atributos `tamanho` e `marca` à classe `Televisão`. Crie dois objetos `Televisão` e atribua tamanhos e marcas diferentes. Depois, imprima o valor desses atributos de forma a confirmar a independência dos valores de cada instância (objeto).

Vejamos agora como associar um comportamento à classe `Televisão`, definindo dois métodos `muda_canal_para_cima` e `muda_canal_para_baixo` (Listagem 10.2).

► Listagem 10.2 – Adição de métodos para mudar o canal

```
>>> class Televisão:
...     def __init__(self):
...         self.ligada=False
...         self.canal=2
...
...     def muda_canal_para_baixo(self): ①
...         self.canal-=1
```

```
...     def muda_canal_para_cima(self): ②
...         self.canal+=1
...
>>> tv = Televisão()
>>> tv.muda_canal_para_cima() ③
>>> tv.muda_canal_para_cima()
>>> tv.canal
4
>>> tv.muda_canal_para_baixo() ④
>>> tv.canal
3
```

Em ①, definimos o método `muda_canal_para_baixo`. Observe que não utilizamos `_` antes do nome do método, pois esse nome não é um nome especial do Python, mas apenas um nome escolhido por nós. Veja que passamos também um parâmetro `self`, que representa o objeto em si. Observe que escrevemos diretamente `self.canal-=1`, utilizando o atributo `canal` da televisão. Isso é possível porque criamos o atributo `canal` no construtor (`_init_`). É usando atributos que podemos armazenar valores entre as chamadas dos métodos.

Em ②, fizemos a mesma coisa, mas, dessa vez, com `muda_canal_para_cima`.

Em ③, chamamos o método. Observe que escrevemos o nome do método após o nome do objeto, separando-os com um ponto, bem como que o método foi chamado da mesma forma que uma função, mas que na chamada não passamos nenhum parâmetro. Na realidade, o interpretador Python adiciona o objeto `tv` à chamada, utilizando-o como o `self` do método em ②. É assim que o interpretador consegue trabalhar com vários objetos de uma mesma classe.

Depois, em ④, fazemos a chamada do método `muda_canal_para_baixo`. Veja o valor retornado por `tv.canal`, antes e depois de chamarmos o método.

A grande vantagem de usar classes e objetos é facilitar a construção dos programas. Embora simples, você pode observar que não precisamos enviar o canal atual da televisão ao método `muda_canal_para_cima`, simplificando a chamada do método. Esse efeito “memória” facilita a configuração de objetos complexos, pois armazenamos as características importantes em seus atributos, evitando repassar esses valores a cada chamada.

Na verdade, esse tipo de construção imita o comportamento do objeto no mundo real. Quando mudamos o canal da tv para cima ou para baixo, não informamos o canal atual para televisão!

10.2 Passagem de parâmetros

Um problema com a classe `Televisão` é que não controlamos os limites de nossos canais. Na realidade, podemos até obter canais negativos ou números muito grandes, como 35790. Vamos modificar o construtor de forma a receber o canal mínimo e máximo suportado por tv (Listagem 10.3).

Execute o programa da listagem 10.3 e verifique se as modificações deram resultado. Observe que mudamos o comportamento da classe `Televisão` sem mudar quase nada no programa que a utiliza. Isso porque isolamos os detalhes do funcionamento da classe do resto do programa. Esse efeito é chamado de encapsulamento. Dizemos que uma classe deve encapsular ou ocultar detalhes de seu funcionamento o máximo possível. No caso dos métodos para mudar o canal, incluímos a verificação sem alterar o resto do programa. Simplesmente solicitamos que o método realize seu trabalho, sem se preocupar com os detalhes internos de como ele realizará essa operação.

► Listagem 10.3 – Verificação da faixa de canais de tv

```
class Televisão:
    def __init__(self, min, max):
        self.ligada = False
        self.canal = 2
        self.cmin = min
        self.cmax = max
    def muda_canal_para_baixo(self):
        if(self.canal-1>=self.cmin):
            self.canal-=1
    def muda_canal_para_cima(self):
        if(self.canal+1<=self.cmax):
            self.canal+=1
tv=Televisão(1,99)
for x in range(0,120):
    tv.muda_canal_para_cima()
print(tv.canal)
for x in range(0,120):
    tv.muda_canal_para_baixo()
print(tv.canal)
```

Exercício 10.2 Atualmente, a classe `Televisão` inicializa o canal com 2. Modifique a classe `Televisão` de forma a receber o canal inicial em seu construtor.

Exercício 10.3 Modifique a classe `Televisão` 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. Exemplo:

```
> > > tv=Televisão(2,10)
> > > tv.muda_canal_para_baixo()
> > > tv.canal
10
> > > tv.muda_canal_para_cima()
> > > tv.canal
2
```

Ao trabalharmos com classes e objetos, assim como fizemos ao estudar funções, precisamos representar em Python uma abstração do problema. Quando realizamos uma abstração reduzimos os detalhes do problema ao necessário para solucioná-lo. Estamos construindo um modelo, ou seja, modelando nossas classes e objetos.

Antes de continuarmos, você deve entender que o modelo pode variar de uma pessoa para outra, como todas as partes do programa. Um dos detalhes mais difíceis é decidir o quanto representar e onde limitar os modelos. No exemplo da classe `Televisão` não escrevemos nada sobre a tomada da TV, se esta tem controle remoto, em que parte da casa está localizada ou mesmo se tem controle de volume.

Como regra simples, modele apenas as informações que você precisa, adicionando detalhes à medida que for necessário. Com o tempo, a experiência ensinará quando parar de detalhar seu modelo, como também apontará erros comuns.

Tudo que aprendemos com funções é também válido para métodos. A principal diferença é que um método é associado a uma classe e atua sobre um objeto. O primeiro parâmetro do método é chamado `self` e representa a instância sobre a qual o método atuará. É por meio de `self` que teremos acesso aos outros métodos de uma classe, preservando todos os atributos de nossos objetos. Você não precisa passar o objeto como primeiro parâmetro ao invocar (chamar) um método: o interpretador Python faz isso automaticamente para você. Entretanto, não se esqueça de declarar `self` como o primeiro parâmetro de seus métodos.

Exercício 10.4 Utilizando o que aprendemos com funções, modifique o construtor da classe `Televisão` de forma que `min` e `max` sejam parâmetros opcionais, onde `min` vale 2 e `max` vale 14, caso outro valor não seja passado.

Exercício 10.5 Utilizando a classe `Televisão` modificada no exercício anterior, crie duas instâncias (objetos), especificando o valor de `min` e `max` por nome.

10.3 Exemplo de um banco

Vejamos o exemplo de classes, mas, dessa vez, vamos modelar contas correntes de um banco. Imagine o Banco Tatu, moderno e eficiente, mas precisando de um novo programa para controlar o saldo de seus correntistas. Cada conta corrente pode ter um ou mais clientes como titular. O banco controla apenas o nome e telefone de cada cliente. A conta corrente apresenta um saldo e uma lista de operações de saques e depósitos. Quando o cliente fizer um saque, diminuiremos o saldo da conta corrente. Quando ele fizer um depósito, aumentaremos o saldo. Por enquanto, o Banco Tatu não oferece contas especiais, ou seja, o cliente não pode sacar mais dinheiro que seu saldo permite.

Vamos resolver esse problema por partes. A classe `Cliente` é simples, tendo apenas dois atributos: nome e telefone. Digite o programa da listagem 10.4 e salve-o em um arquivo chamado `cliente.py`.

► Listagem 10.4 – Classe Clientes (clientes.py)

```
class Cliente:
    def __init__(self, nome, telefone):
        self.nome = nome
        self.telefone = telefone
```

Abra o arquivo `clientes.py` no IDLE e execute-o (Run – F5). No interpretador, experimente criar um objeto da classe `Cliente`.

```
joão=Cliente("João da Silva", "777-1234")
maria=Cliente("Maria Silva", "555-4321")
joão.nome
```

```
joão.telefone
maria.nome
maria.telefone
```

Como nos outros programas Python, podemos executar um definição de classe dentro de um arquivo `.py` e utilizar o interpretador para experimentar nossas classes e objetos. Essa operação é muito importante para testarmos as classes, modificar alguns valores e repetir o teste.

Como o programa do Banco Tatu vai ficar maior que os programas que já trabalhamos até aqui, vamos gravar cada classe em um arquivo `.py` separado. Em Python essa organização não é obrigatória, pois a linguagem permite que tenhamos todas as nossas classes em um só arquivo, se assim quisermos.

Vamos agora criar o arquivo `teste.py`, que vai simplesmente importar `clientes.py` e criar dois objetos.

► Listagem 10.5 – Programa teste.py que importa a classe Cliente (clientes.py)

```
from clientes import Cliente
joão=Cliente("João da Silva", "777-1234")
maria=Cliente("Maria da Silva", "555-4321")
```

Observe que com poucas linhas de código conseguimos reutilizar a classe `Cliente`. Isso é possível porque `clientes.py` e `teste.py` estão no mesmo diretório. Essa separação permite que passemos a experimentar as novas classes no interpretador, armazenando o `import` e a criação dos objetos em um arquivo à parte. Assim poderemos definir nossas classes separadamente dos experimentos ou testes.

Para resolver o problema do Banco Tatu, precisamos de outra classe, `Conta`, para representar uma conta do banco com seus clientes e seu saldo. Vejamos o programa da listagem 10.6. A classe `Conta` é definida recebendo clientes, número e saldo em seu construtor (`__init__`), onde em clientes esperamos uma lista de objetos da classe `Cliente`. `número` é uma string com o número da conta, e `saldo` é um parâmetro opcional, tendo zero (0) como padrão. A listagem também apresenta os métodos `resumo`, `saque` e `depósito`. O método `resumo` exibe na tela o número da conta corrente e seu saldo. `saque` permite retirar dinheiro da conta corrente, verificando se essa operação é possível (`self.saldo > valor`). `depósito` simplesmente adiciona o valor solicitado ao saldo da conta corrente.

► Listagem 10.6 – Classe Conta (contas.py)

```
class Conta:
    def __init__(self, clientes, número, saldo = 0):
        self.saldo = saldo
        self.clientes = clientes
        self.número = número
    def resumo(self):
        print("CC Número: %s Saldo: %10.2f" %
              (self.número, self.saldo))
    def saque(self, valor):
        if self.saldo >= valor:
            self.saldo -= valor
    def depósito(self, valor):
        self.saldo += valor
```

Altere o programa *teste.py* de forma a importar a classe *Conta*. Crie uma conta corrente para os clientes João e Maria.

Faça alguns testes no interpretador:

```
conta.resumo()
conta.saque(1000)
conta.resumo()
conta.saque(50)
conta.resumo()
conta.depósito(200)
conta.resumo()
```

Embora nossa conta corrente comece a funcionar, ainda não temos a lista de operações de cada elemento. Essa lista é, na realidade, um extrato de conta. Vamos alterar a classe *Conta* de forma a adicionar um atributo que é a lista de operações realizadas (Listagem 10.7). Considere o saldo inicial como um depósito. Vamos adicionar também um método *extrato* para imprimir todas as operações realizadas.

► Listagem 10.7 – Conta com registro de operações e extrato (contas.py)

```
class Conta:
    def __init__(self, clientes, número, saldo = 0):
        self.saldo = 0
        self.clientes = clientes
```

```
    self.número = número
    self.operações = []
    self.depósito(saldo)
    def resumo(self):
        print("CC N°%s Saldo: %10.2f" %
              (self.número, self.saldo))
    def saque(self, valor):
        if self.saldo >= valor:
            self.saldo -= valor
            self.operações.append(["SAQUE", valor])
    def depósito(self, valor):
        self.saldo += valor
        self.operações.append(["DEPÓSITO", valor])
    def extrato(self):
        print("Extrato CC N° %s\n" % self.número)
        for o in self.operações:
            print("%10s %10.2f" % (o[0], o[1]))
        print("\n      Saldo: %10.2f\n" % self.saldo)
```

Modifique também o programa de testes para imprimir o extrato de cada conta (Listagem 10.8).

► Listagem 10.8 – Testando Cliente e Contas

```
from clientes import Cliente
from contas import Conta

joão=Cliente("João da Silva", "777-1234")
maria=Cliente("Maria da Silva", "555-4321")
conta1=Conta([joão], 1, 1000)
conta2=Conta([maria, joão], 2, 500)
conta1.saque(50)
conta2.depósito(300)
conta1.saque(190)
conta2.depósito(95.15)
conta2.saque(250)
conta1.extrato()
conta2.extrato()
```

Exercício 10.6 Altere o programa de forma que a mensagem saldo insuficiente seja exibida caso haja tentativa de sacar mais dinheiro que o saldo disponível.

Exercício 10.7 Modifique o método resumo da classe Conta para exibir o nome e o telefone de cada cliente.

Exercício 10.8 Crie uma nova conta, agora tendo João e José como clientes e saldo igual a 500.

Para resolver o problema do Banco Tatú precisamos de uma classe para armazenar todas as nossas contas. Como atributos do banco teríamos seu nome e a lista de contas. Como operações, considere a abertura de uma conta corrente e a listagem de todas as contas do banco.

► Listagem 10.9 – Classe Banco (bancos.py)

```
class Banco:
    def __init__(self, nome):
        self.nome=nome
        self.clientes=[]
        self.contas=[]
    def abre_conta(self, conta):
        self.contas.append(conta)
    def lista_contas(self):
        for c in self.contas:
            c.resumo()
```

Agora, vamos criar os objetos (Listagem 10.10).

► Listagem 10.10 – Criando os objetos

```
from clientes import Cliente
from bancos import Banco
from contas import Conta
joão = Cliente("João da Silva", "3241-5599")
```

```
maria = Cliente("Maria Silva", "7231-9955")
jósé = Cliente("José Vargas", "9721-3040")
contaJM = Conta( [joão, maria], 100)
contaJ = Conta( [jósé], 10)
tatú = Banco("Tatú")
tatú.abre_conta(contaJM)
tatú.abre_conta(contaJ)
tatú.lista_contas()
```

Exercício 10.9 Crie classes para representar estados e cidades. Cada estado tem um nome, sigla e cidades. Cada cidade tem nome e população. Escreva um programa de testes que crie três estados com algumas cidades em cada um. Exiba a população de cada estado como a soma da população de suas cidades.

10.4 Herança

A orientação a objetos permite modificar nossas classes, adicionando ou modificando atributos e métodos, tendo como base outra classe. Vejamos o exemplo do Banco Tatú, em que o novo sistema foi um sucesso. Para atrair novos clientes, o Banco Tatú começou a oferecer contas especiais aos clientes. Uma conta especial permite que possamos sacar mais dinheiro que atualmente disponível no saldo da conta, até um determinado limite. As operações de depósito, extrato e resumo são as mesmas de uma conta normal. Vamos criar a classe ContaEspecial herdando o comportamento da classe Conta. Digite a listagem 10.11 no mesmo arquivo onde a classe Conta foi definida (contas.py).

► Listagem 10.11 – Uso de herança para definir ContaEspecial

```
class ContaEspecial(Conta):
    def __init__(self, clientes, número, saldo = 0, limite=0):
        Conta.__init__(self, clientes, número, saldo) ❷
        self.limite = limite ❸
    def saque(self, valor):
        if self.saldo + self.limite >= valor:
            self.saldo -= valor
            self.operações.append(["SAQUE", valor])
```

Em ❶ definimos a classe `ContaEspecial`, mas observe que escrevemos `Conta` entre parênteses. Esse é o formato de declaração de classe usando herança, ou seja, é assim que declaramos a herança de uma classe em Python. Essa linha diz: crie uma nova classe chamada `ContaEspecial` herdando todos os métodos e atributos da classe `Conta`. A partir daqui, `ContaEspecial` é uma subclasse de `Conta`. Também dizemos que `Conta` é a superclasse de `ContaEspecial`.

Em ❷, chamamos o método `__init__` de `Conta`, escrevendo `Conta.__init__` seguido dos parâmetros que normalmente passaríamos. Toda vez que você utilizar herança, o método construtor da superclasse, no caso `Conta`, deve ser chamado. Observe que, nesse caso, passamos `self` para `Conta.__init__`. É assim que reutilizamos as definições já realizadas na superclasse, evitando ter que reescrever as atribuições de clientes, número e saldo. Chamar a inicialização da superclasse também tem outras vantagens, como garantir que modificações no construtor da superclasse não tenham que ser duplicadas em todas as subclasses.

Em ❸ criamos o atributo `self.limite`. Esse atributo será criado apenas para classes do tipo `ContaEspecial`. Observe que criamos o novo atributo depois de chamarmos `Conta.__init__`.

Observe também que não chamamos `Conta.saque` no método `saque` de `ContaEspecial`. Quando isso ocorre, estamos substituindo completamente a implementação do método por uma nova. Uma das grandes vantagens de utilizar herança de classes é justamente poder substituir ou complementar métodos já definidos.

Para testar essa modificação, escreva o trecho de programa da listagem 10.11 no mesmo arquivo em que você definiu a classe `Conta`. Modifique seu programa de teste para que se pareça com o programa da listagem 10.12.

Listagem 10.12 – Criação e uso de uma `ContaEspecial`

```
from clientes import Cliente
from contas import Conta, ContaEspecial ❶
joão=Cliente("João da Silva", "777-1234")
maria=Cliente("Maria da Silva", "555-4321")
conta1=Conta([joão], 1, 1000)
conta2=ContaEspecial([maria, joão], 2, 500, 1000) ❷
conta1.saque(50)
conta2.deposito(300)
conta1.saque(190)
conta2.deposito(95.15)
```

```
conta2.saque(1500)
conta1.extrato()
conta2.extrato()
```

Vejamos o que mudou no programa de testes. Em ❶ adicionamos o nome da classe `ContaEspecial` ao `import`. Dessa forma, poderemos utilizar a nova classe em nossos testes. Já em ❷ criamos um objeto `ContaEspecial`. Veja que, praticamente, não mudamos nada, exceto o nome da classe, que agora é `ContaEspecial`, e não `Conta`. Um detalhe importante para o teste é que adicionamos um parâmetro ao construtor, no caso 1000, como o valor de `limite`. Execute esse programa de teste e observe que, para a `conta2`, obtivemos um saldo negativo.

Utilizando herança, modificamos muito pouco nosso programa, mantendo a funcionalidade anterior e adicionando novos recursos. O interessante de tudo isso é que foi possível reutilizar os métodos que já havíamos definido na classe `Conta`. Isso permitiu que a definição da classe `ContaEspecial` fosse bem menor, pois lá especificamos apenas o comportamento que é diferente.

Quando você utilizar herança, tente criar classes nas quais o comportamento e características comuns fiquem na superclasse. Dessa forma, você poderá definir subclasses enxutas. Outra vantagem de utilizar herança é que, se mudarmos algo na superclasse, essas mudanças serão também usadas pelas subclasses. Um exemplo seria modificarmos o método de extrato. Como em `ContaEspecial` não especificamos um novo método de extrato, ao modificarmos o método `Conta.extrato`, estaremos também modificando o extrato de `ContaEspecial`, pois as duas classes compartilham o mesmo método.

É importante notar que, ao utilizarmos herança, as subclasses devem poder substituir suas superclasses, sem perda de funcionalidade e sem gerar erros nos programas. O importante é que você conheça esse novo recurso e comece a utilizá-lo em seus programas. Lembre-se de que você não é obrigado a definir uma hierarquia de classes em todos os seus programas. Com o tempo, a necessidade de utilizar herança ficará mais clara.

Exercício 10.10 Modifique as classes `Conta` e `ContaEspecial` para que a operação de saque retorne verdadeiro se o saque foi efetuado e falso em caso contrário.

Exercício 10.11 Altere a classe `ContaEspecial` de forma que seu extrato exiba o limite e o total disponível para saque.

Exercício 10.12 Observe o método saque das classes `Conta` e `ContaEspecial`. Modifique o método saque da classe `Conta` de forma que a verificação da possibilidade de saque seja feita por um novo método, substituindo a condição atual. Esse novo método retornará verdadeiro se o saque puder ser efetuado, e falso em caso contrário. Modifique a classe `ContaEspecial` de forma a trabalhar com esse novo método. Verifique se você ainda precisa trocar o método saque de `ContaEspecial` ou apenas o novo método criado para verificar a possibilidade de saque.

10.5 Desenvolvendo uma classe para controlar listas

Agora que já temos uma ideia de como utilizar classes em Python, vamos criar uma classe que controle uma lista de objetos. Nos exemplos anteriores, ao usarmos a lista de clientes como uma lista simples, não fizemos nenhuma verificação quanto à duplicidade de valores. Por exemplo, uma lista de clientes onde os dois clientes são a mesma pessoa seria aceita sem problemas. Outro problema de nossa lista simples é que ela não verifica se os elementos são objetos da classe `Cliente`. Vamos modificar isso construindo uma nova classe, chamada `ListaÚnica`.

► Listagem 10.13 – Classe ListaÚnica (`listaunica.py`)

```
class ListaÚnica:
    def __init__(self, elem_class):
        self.lista = []
        self.elem_class = elem_class
    def __len__(self):
        return len(self.lista)
    def __iter__(self):
        return iter(self.lista)
    def __getitem__(self, p):
        return self.lista[p]
    def indiceVálido(self, i):
        return i >= 0 and i < len(self.lista)
    def adiciona(self, elem):
        if self.pesquisa(elem) == -1:
            self.lista.append(elem)
    def remove(self, elem):
        self.lista.remove(elem)
```

```
def pesquisa(self, elem):
    self.verifica_tipo(elem)
    try:
        return self.lista.index(elem)
    except ValueError:
        return -1
def verifica_tipo(self, elem):
    if type(elem) != self.elem_class:
        raise TypeError("Tipo inválido")
def ordena(self, chave = None):
    self.lista.sort(key=chave)
```

Vejamos o que podemos fazer com essa classe, realizando alguns testes no interpretador:

```
>>> from listaunica import *
>>> lu = ListaÚnica(int) ❶
>>> lu.adiciona(5) ❷
>>> lu.adiciona(3)
>>> lu.adiciona(2.5) ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "listaunica.py", line 19, in adiciona
    if self.pesquisa(elem) == -1:
  File "listaunica.py", line 26, in pesquisa
    self.verifica_tipo(elem)
  File "listaunica.py", line 34, in verifica_tipo
    raise TypeError("Tipo inválido")
TypeError: Tipo inválido
```

Em ❶ criamos um objeto com a classe `ListaÚnica` que acabamos de importar. Veja que passamos `int` como parâmetro no construtor, indicando que essa lista deve apenas conter valores do tipo `int` (inteiros). Em ❷, adicionamos o número inteiro `5` como elemento de nossa lista, assim como `3` na linha seguinte. Veja que não tivemos algum problema, mas que em ❸, ao tentarmos adicionar `2.5` (um número do tipo `float`), obtivemos uma exceção com mensagem de erro tipo inválido (`TypeError`). Isso acontece porque, em nosso construtor, o parâmetro `elem` é na realidade a classe que desejamos para nossos elementos. Sempre que

um elemento é adicionado (método `adiciona`), uma verificação do tipo do novo elemento é feita pelo método `verifica_tipo`, que gera uma exceção, caso o tipo não seja igual ao tipo passado ao construtor. Veja que o método `adiciona` também realiza uma pesquisa para verificar se um elemento igual já não faz parte da lista e só realiza a adição caso um elemento igual não tenha sido encontrado. Dessa forma, podemos garantir que nossa lista conterá apenas elementos do mesmo tipo e sem repetições.

Vamos continuar a testar a nossa nova classe no interpretador:

```
>>> len(lu) ❸
2
>>> for e in lu: ❹
...     print(e)
...
5
3
>>> lu.adiciona(5) ❺
>>> len(lu) ❻
2
>>> lu[0] ❼
5
>>> lu[1]
3
```

Em ❸, utilizamos a função `len` com nosso objeto e obtivemos o resultado esperado. Veja que não tivemos que escrever `len(lu.lista)`, mas apenas `len(lu)`. Isso é possível, pois implementamos o método `_len_`, que é responsável por retornar o número de elementos a partir de `self.lista`.

Em ❹, utilizamos nosso objeto `lu` em um `for`. Isso foi possível com a implementação do método `_iter_`, que é chamado quando utilizamos um objeto com `for`. O método `_iter_` simplesmente chama a função `iter` do Python com nossa lista interna `self.lista`. A intenção de utilizarmos esses métodos é de esconder alguns detalhes de nossa classe `ListaÚnica` e evitar o acesso direto à `self.lista`.

Em ❺, testamos a inclusão de um elemento repetido para, na linha seguinte, em ❼, confirmarmos que a quantidade de elementos da lista não foi alterada. Isso acontece porque, ao realizar a pesquisa no método `adiciona`, o valor é encontrado, e a adição é ignorada.

Em ❻, utilizamos índices com nossa classe, da mesma forma que fazemos com uma lista normal do Python. Isso é possível, pois implementamos o método `_getitem_`, que recebe o valor do índice (`p`) e retorna o elemento correspondente de nossa lista.

Python possui vários métodos mágicos, métodos especiais que têm o formato `__nome__`. O método `_init_`, usado em nossos construtores é um método mágico, `_len_`, `_getitem_` e `_iter_` também. Esses métodos permitem dar outro comportamento a nossas classes e usá-las quase que como classes da própria linguagem. A utilização desses métodos mágicos não é obrigatória, mas possibilita uma grande flexibilidade para nossas classes.

Vejamos outro exemplo de classe com métodos mágicos (especiais). A classe `Nome`, definida na listagem 10.14, é utilizada para guardar nomes de pessoas e manter uma chave de pesquisa.

► Listagem 10.14 – Classe Nome (nome.py)

```
class Nome:
    def __init__(self, nome):
        if nome == None or not nome.strip():
            raise ValueError("Nome não pode ser nulo nem em branco")
        self.nome = nome
        self.chave = nome.strip().lower()
    def __str__(self):
        return self.nome
    def __repr__(self):
        return "<Classe {} em 0x{:02x} Nome: {} Chave: {}>".format(id(self),
                                                               self.nome, self.chave, type(self).__name__)
    def __eq__(self, outro):
        print("__eq__ Chamado")
        return self.nome == outro.nome
    def __lt__(self, outro):
        print("__lt__ Chamado")
        return self.nome < outro.nome
```

Na classe `Nome`, definimos o método `_init_` de forma a gerar uma exceção caso a string passada como nome seja nula (`None`) ou em branco. Veja que implementamos também o método mágico `_str_`, que é chamado ao imprimirmos um

objeto da classe `Nome`. Dessa forma, podemos configurar a saída de nossas classes com mais liberdade e sem mudar o comportamento esperado de uma classe normal em Python.

Vejamos o resultado desse programa no interpretador:

```
>>> from nome import *
>>> A=Nome("Nilo") ❶
>>> print(A) ❷
Nome
>>> B=Nome(" ") ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "nome.py", line 4, in __init__
      raise ValueError("Nome não pode ser nulo nem em branco")
ValueError: Nome não pode ser nulo nem em branco
>>> C=None ❹
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "nome.py", line 4, in __init__
      raise ValueError("Nome não pode ser nulo nem em branco")
ValueError: Nome não pode ser nulo nem em branco
```

Em ❶, criamos um objeto da classe `Nome`, passando a string "Nilo". Em ❷, imprimimos o objeto `A` diretamente, veja que o resultado foi o nome Nilo, que é o valor retornado pelo nosso método `__str__`. Em ❸ e ❹, testamos a inicialização de um objeto com valores inválidos. Veja que uma exceção do tipo `ValueError` foi gerada, impedindo a criação do objeto com o valor inválido. O objetivo desse tipo de validação é garantir que o valor usado na classe seja válido. Podemos implementar regras mais complexas dependendo do objetivo de nosso programa, mas, para esse exemplo, verificar se a string está em branco ou se é nula (`None`) é suficiente.

Vamos continuar a descobrir os outros métodos especiais no interpretador:

```
>>> A=Nome("Nilo")
>>> A ❶
<Classe Nome em 0x26ee8f0 Nome: Nilo Chave: nilo>
>>> A == Nome("Nilo") ❷
__eq__ Chamado
True
```

```
>>> A != Nome("Nilo") ❸
__eq__ Chamado
False
>>> A < Nome("Nilo") ❹
__lt__ Chamado
False
>>> A > Nome("Nilo") ❺
__lt__ Chamado
False
```

Em ❶, temos a representação do objeto `A`, ou seja, a forma que é usada pelo interpretador para mostrar o objeto fora da função `print` ou quando usamos a função `repr`. Veja que a saída foi gerada pelo método `__repr__` da classe `Nome`.

Em ❷, utilizamos o operador `==` para verificar se o objeto `A` é igual a outro objeto. Em Python, o comportamento padrão é que `==` retorne `True` se os dois objetos forem o mesmo objeto, ou `False` caso contrário. No entanto, o objeto `A` e `Nome("Nilo")` são claramente duas instâncias diferentes da classe `Nome`. Veja que o resultado é `True` e que nosso método `__eq__` foi chamado. Isso acontece porque `__eq__` é o método especial utilizado para comparações de igualdade (`==`) de nossos objetos. Em nossa implementação, retornamos `True` se o conteúdo de `self.nome` for igual nos dois objetos, independentemente de serem o mesmo objeto ou não. Veja também que em ❸, o método `__eq__` também foi chamado, embora não tenhamos definido o método `__neq__(!=)`. Esse comportamento vem da implementação padrão desses métodos, onde `__eq__` foi negado para implementar o `__neq__` inexistente. Varemos depois uma forma de declarar todos os operadores de comparação apenas implementando o `__eq__` e `__lt__`.

Em ❹, usamos o operador `<` (menor que) para comparar os dois objetos. Veja que o método especial `__lt__` foi chamado nesse caso. Nossa implementação do método `__lt__` utiliza a comparação de `self.nome` para decidir a ordem dos objetos. Em ❺, o operador `>` (maior que) retorna o valor correto, e também chama o método `__lt__`, uma vez que o método `__gt__` não foi implementado, de forma análoga ao que aconteceu entre `__eq__` e `__neq__`.

Agora veja o que acontece quando tentamos utilizar os operadores `>=` (maior ou igual) e `<=` (menor ou igual):

```
>>> A >= Nome("Nilo")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```

TypeError: unorderable types: Nome() >= Nome()
>>> A <= Nome("Nilo")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Nome() <= Nome()

```

Isso acontece porque os operadores `>=` e `<=` chamam os métodos `__ge__` e `__le__`, respectivamente, e ambos não foram implementados.

Como algumas dessas relações não funcionam como o esperado, existe uma forma mais simples de implementar esses métodos especiais, apenas com a implementação de `__eq__` e de `__lt__`. Na listagem 10.15, vamos usar um recurso chamado *decorators* (adornos ou decoradores).

► Listagem 10.15 – Usando anotações (nome.py)

```

from functools import total_ordering

@total_ordering
class Nome:
    def __init__(self, nome):
        if nome == None or not nome.strip():
            raise ValueError("Nome não pode ser nulo nem em branco")
        self.nome = nome
        self.chave = Nome.CriaChave(nome)
    def __str__(self):
        return self.nome
    def __repr__(self):
        return "<Classe {} em 0x{:x} Nome: {} Chave: {}>".format(id(self),
            self.nome, self.chave, type(self).__name__)
    def __eq__(self, outro):
        print("__eq__ Chamado")
        return self.nome == outro.nome
    def __lt__(self, outro):
        print("__lt__ Chamado")
        return self.nome < outro.nome
    @staticmethod
    def CriaChave(nome):
        return nome.strip().lower()

```

O primeiro decorador `@total_ordering` é definido no módulo `functools`, por isso tivemos que importá-lo no início de nosso programa. Ele é responsável por implementar, ou seja, gerar o código responsável pela implementação de todos os métodos de comparação especiais, a partir de `__eq__` e de `__lt__`. Dessa forma, `__neq__` será a negação de `__eq__`; `__gt__`, a negação de `__lt__`; `__le__`, a combinação de `__lt__` com `__eq__`; e `__ge__`, a combinação de `__gt__` com `__eq__`, implementado, assim, todos os operadores de comparação (`==`, `!=`, `>`, `<`, `>=`, `<=`).

Vejamos o resultado no interpretador:

```

>>> from nome import *
>>> A=("Nilo")
>>> A=Nome("Nilo")
>>> A == Nome("Nilo")
__eq__ Chamado
True
>>> A != Nome("Nilo")
__eq__ Chamado
False
>>> A > Nome("Nilo")
__lt__ Chamado
__eq__ Chamado
False
>>> A < Nome("Nilo")
__lt__ Chamado
False
>>> A <= Nome("Nilo")
__lt__ Chamado
__eq__ Chamado
True
>>> A >= Nome("Nilo")
__lt__ Chamado
True

```

Observe que no programa da listagem 10.15, utilizamos um outro decorador `@staticmethod` antes da definição do método `CriaChave`. Veja também que o método `CriaChave` não possui o parâmetro `self`. Esse decorador cria um método estático, isto é, um método que pode ser chamado apenas com o nome da classe, não necessitando de um objeto para ser chamado. Vejamos no interpretador:

```
>>> A.CriaChave("X")
'x'
>>> Nome.CriaChave("X")
'x'
```

Tanto a chamada de `CriaChave` com o objeto `A` quanto a chamada com o nome da classe em `Nome.CriaChave` funcionaram como esperado. Métodos estáticos são utilizados para implementar métodos que não acessam ou que não precisam acessar as propriedades de uma instância da classe. No caso do método `CriaChave`, apenas o parâmetro `nome` é necessário para seu funcionamento. O decorador `@staticmethod` é necessário para informar ao interpretador não passar o parâmetro `self` automaticamente, como faz nos métodos normais, não estáticos. A vantagem de definirmos métodos estáticos é agrupar certas funções em nossas classes. Dessa forma, fica claro que `CriaChave` funciona no contexto de nomes. Se fosse definido como uma função fora de uma classe, sua implementação poderia parecer mais genérica do que realmente é. Desta forma, métodos estáticos ajudam a manter o programa coeso e a manter o contexto da implementação do método.

O programa da listagem 10.15 introduz um problema de consistência de dados. Veja, que ao construirmos o objeto em `__init__`, verificamos o valor de `nome` e também atualizamos a chave de pesquisa. O problema é que podemos alterar `nome` sem qualquer validação e deixar chave em um estado inválido. Vejamos isso no interpretador:

```
>>> A=Nome("Teste")
>>> A
<Classe Nome em 0x3000f10 Nome: Teste Chave: teste>
>>> A.nome="Nilo"
>>> A
<Classe Nome em 0x3000f10 Nome: Nilo Chave: teste>
>>> A.chave="TST"
>>> A
<Classe Nome em 0x3000f10 Nome: Nilo Chave: TST>
```

Veja que depois da construção do objeto, o nome e a chave estão corretos, pois essa operação é garantida pela nossa implementação de `__init__`. No entanto, um acesso à `A.nome` não atualizará o valor da chave, que continua com o valor configurado pelo construtor. O mesmo acontece com o atributo `chave` que pode ser alterado sem qualquer ligação com o `nome` em si. Na programação orientada a objetos, uma classe é responsável por gerenciar seu estado interno e manter a

consistência do objeto entre chamadas de métodos. Para garantir que `nome` e `chave` sempre sejam atualizados corretamente, vamos fazer uso de um recurso do Python que utiliza métodos para realizar a atribuição e a leitura de valores. Esse recurso é chamado de propriedade. O programa da listagem 10.16 também utilizará um outro recurso, uma convenção especial de nomenclatura de nossos métodos para esconder os atributos da classe e impedir o acesso direto a eles de fora da classe.

► Listagem 10.16 – Classe Nome com propriedades (nome.py)

```
from functools import total_ordering

@total_ordering
class Nome:
    def __init__(self, nome):
        self.__nome = nome ❶

    def __str__(self):
        return self.nome

    def __repr__(self):
        return "<Classe {} em {} Nome: {} Chave: {}>".format(
            id(self), self.__nome, self.__chave, type(self).__name__) ❷

    def __eq__(self, outro):
        return self.nome == outro.nome

    def __lt__(self, outro):
        return self.nome < outro.nome

    @property ❸
    def nome(self):
        return self.__nome ❹

    @nome.setter ❺
    def nome(self, valor):
        if valor == None or not valor.strip():
            raise ValueError("Nome não pode ser nulo nem em branco")
        self.__nome = valor ❻
        self.__chave = Nome.CriaChave(valor) ❼

    @staticmethod
    def CriaChave(nome):
        return nome.strip().lower()
```

O programa da listagem 10.16 utiliza outros decoradores como `@property`, em ❸, e `@nome.setter`, em ❹. Esses decoradores modificam o método logo abaixo deles, transformando-os em propriedades. O primeiro decorador, `@property`, transforma o método `nome` na propriedade `nome`; dessa forma, sempre que escrevermos `objeto.nome`, estaremos chamando esse método, que retorna `self.__nome`. Já o segundo decorador, `@nome.setter`, transforma o método `nome` na propriedade usada para alterar o valor de `__nome`. Dessa forma, quando escrevermos `objeto.nome = valor`, esse método será chamado para efetuar as alterações. Observe que copiamos a verificação de tipo e a criação da chave do método `__init__` (em ❶) para o método marcado por `@nome.setter` em ❹. Assim, em ❷, chamamos o método de ❹ ao escrevermos: `self.nome = nome`, uma vez que `self.nome` é agora uma propriedade. Outro detalhe importante é que acrescentamos duas sublinhadas (`__`) antes do nome dos atributos `nome` e `chave`, fazendo-os `__nome` e `__chave`. Dessa forma, `__nome` e `__chave` ficam escondidos quando acessados de fora da classe. Esse "esconder" é apenas um detalhe da implementação do Python, que modifica o nome desses atributos de forma a torná-los inacessíveis (*name mangling*). No entanto, seus nomes foram apenas transformados pela adição de `_` e do nome da classe, fazendo com que `__chave` se torne `_Nome__chave`. Mas essa proteção é suficiente para garantir que `nome` e `chave` estejam sincronizadas e que nossa classe funcione como esperado. Em Python, preste muita atenção ao utilizar nomes que começam com `_` ou `__`. Esses símbolos indicam que esses atributos não devem ser acessados, exceto pelo código da própria classe. Não confundir os atributos protegidos, cujo nome começa por `__`, com o nome dos métodos mágicos os especiais que já vimos, que começam e terminam por `__`. Em ❸, podemos ver que `__nome` e `__chave` continuam acessíveis dentro do código da classe por `self.__nome` e `self.__chave`.

Vamos experimentar o novo código no interpretador:

```
>>> from nome import *
>>> A=Nome("Nilo") ❶
>>> A
<Classe Nome em 0x3140f10 Nome: Nilo Chave: nilo>
>>> A.nome="Nilo Menezes" ❷
>>> A
<Classe Nome em 0x3140f10 Nome: Nilo Menezes Chave: nilo menezes>
>>> A.__nome ❸
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

```
AttributeError: 'Nome' object has no attribute '__nome'
>>> A.__chave ❹
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Nome' object has no attribute '__chave'
>>> A.chave ❺
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Nome' object has no attribute 'chave'
>>> A._Nome__chave ❻
'nilo menezes'
```

Em ❶, criamos o objeto `A`, como anteriormente. Veja que o `self.nome` dentro do `__init__` chamou corretamente o método marcado por `@nome.setter`, e que `chave` e `nome` foram corretamente configuradas. Em ❷, alteramos o `nome` e logo podemos ver que a chave foi atualizada ao mesmo tempo. Observe que `A.nome` é acessível de fora da classe e que agora faz parte de nosso objeto como antes fazia o atributo `self.nome`. ❸, ❹ e ❺ mostram que `__nome`, `__chave` e `chave` são inacessíveis de fora da classe. Em ❻, mostramos o acesso à chave, com o truque de usar o nome completo após a escondida do Python (*name mangling*): `_nomedaclasse__atributo`, gerando: `_Nome__chave`. Entenda esse tipo de acesso como uma curiosidade e que, quando um programador marcar um atributo com `__`, está dizendo: não utilize esse atributo fora da classe, salvo se tiver certeza do que está fazendo.

O truque dos nomes que começam por `__` também funciona com métodos. Se você quiser marcar um método para ser utilizado apenas dentro da classe, adicione `__` antes de seu nome, da mesma forma que fizemos com nossos atributos.

Em todos os casos, nunca crie métodos ou atributos que comecem e terminem por `__`, esse tipo de construção é reservado aos métodos mágicos (especiais) da linguagem.

Você pode criar atributos que podem ser lidos definindo apenas o método de acesso com `@property`. Em nosso exemplo, `@nome.setter` é que permite alterarmos o nome. Se não utilizássemos `@nome.setter`, `A.nome` seria acessível apenas para leitura, e uma exceção seria gerada se tentássemos alterar seu valor. Vejamos como isso funciona no programa da listagem 10.17.

► Listagem 10.17 – Chave como propriedade apenas para leitura (nome.py)

```
from functools import total_ordering
@total_ordering
class Nome:
    def __init__(self, nome):
        self.nome = nome
    def __str__(self):
        return self.nome
    def __repr__(self):
        return "<Classe {} em 0x{:x} Nome: {} Chave: {}>".format(
            id(self), self.__nome, self.__chave, type(self).__name__)
    def __eq__(self, outro):
        return self.nome == outro.nome
    def __lt__(self, outro):
        return self.nome < outro.nome
    @property
    def nome(self):
        return self.__nome
    @nome.setter
    def nome(self, valor):
        if valor == None or not valor.strip():
            raise ValueError("Nome não pode ser nulo nem em branco")
        self.__nome = valor
        self.__chave = Nome.CriaChave(valor)
    @property
    def chave(self):
        return self.__chave
    @staticmethod
    def CriaChave(nome):
        return nome.strip().lower()
```

Veja que na listagem 10.17, marcamos apenas o método com `@property` e não criamos um método com `@chave.setter` para processar as modificações. No interpretador, vejamos o que acontece se tentarmos alterar chave:

```
>>> from nome import *
>>> A=Nome("Nilo")
>>> A.chave
'nilo'
>>> A.chave="nilo menezes"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> A.nome="Nilo Menezes"
>>> A.chave
'nilo menezes'
```

10.6 Revisitando a agenda

No capítulo 9, desenvolvemos uma agenda de nomes e telefones. Agora que conhecemos classes, podemos adaptar o código da agenda de forma a melhorar seu funcionamento e a utilizar alguns conceitos de programação orientada a objetos.

Vamos revisitar a agenda de forma a definir suas estruturas como objetos e modelar a aplicação da agenda em uma classe separada. Nas seções anteriores, definimos as classes `Nome` e `ListaÚnica`. Essas classes serão utilizadas no modelo de dados de nossa nova agenda.

Podemos visualizar nossa agenda como uma lista de nomes e telefones. A agenda original do capítulo 9 suportava apenas um telefone por nome, mas modificamos isso no exercício 9.28, logo nossa nova agenda deverá suportar vários telefones, e cada telefone terá um tipo específico (celular, trabalho, fax etc.).

Vamos começar pela classe que vai gerenciar os tipos de telefone. Vejamos o código na listagem 10.18.

► Listagem 10.18 – A classe `TipoTelefone`

```
from functools import total_ordering
@total_ordering
class TipoTelefone:
    def __init__(self, tipo):
        self.tipo = tipo
```

```

def __str__(self):
    return "{0}{1}.format(self.tipo)

def __eq__(self, outro):
    if outro is None:
        return False
    return self.tipo == outro.tipo

def __lt__(self, outro):
    return self.tipo < outro.tipo

```

O programa da listagem 10.18 implementa nosso tipo de telefone. Veja que implementamos o método `__str__` para exibir o nome do tipo entre parênteses, e o método `__eq__` e `__lt__` para ativar a comparação por tipo. Nesse exemplo, não usamos propriedades, pois na classe `TipoTelefone` não estamos fazendo qualquer verificação. Dessa forma, `self.tipo` oferecerá a mesma forma de acesso, seja como atributo ou propriedade. A grande vantagem de utilizar propriedades é que podemos, mais tarde, se necessário, transformar `self.tipo` em uma propriedade, sem alterar o código que utiliza a classe, como fizemos com o nome na classe `Nome`. Observe que no método `__eq__`, incluímos uma verificação se a outra parte for `None`. Essa verificação é necessária porque nossa futura agenda utilizará `None` quando não soubermos o tipo de um telefone. Assim, `self.tipo == outro.tipo` falharia, pois `None` não possui um atributo chamado `tipo`. O teste se `outro` for `None` foi feito com o operador `is` do Python. Isso é importante, pois, se utilizarmos `==`, estaremos chamando recursivamente o método `__eq__` que é chamado pelo operador de comparação (`==`).

A classe `Telefone` será implementada pelo programa da listagem 10.19.

► Listagem 10.19 – A classe `Telefone`

```

class Telefone:
    def __init__(self, número, tipo=None):
        self.número = número
        self.tipo = tipo

    def __str__(self):
        if self.tipo!=None:
            tipo = self.tipo
        else:
            tipo = ""
        return "{0} {1}.format(self.número, tipo)

```

```

def __eq__(self, outro):
    return self.número == outro.número and (
        self.tipo == outro.tipo) or (
        self.tipo == None or outro.tipo == None)

@property
def número(self):
    return self.__número

@número.setter
def número(self, valor):
    if valor == None or not valor.strip():
        raise ValueError("Número não pode ser None ou em branco")
    self.__número = valor

```

Na listagem 10.19, fazemos o mesmo tipo de verificação de tipo que fizemos na classe `Nome`. Dessa forma, um objeto da classe `Telefone` não aceita números vazios. No método `__eq__`, implementamos a verificação de forma a ignorar um `Telefone` sem `TipoTelefone`, ou seja, um telefone onde o tipo é `None`. Dessa forma, se compararmos dois objetos de `Telefone` e um ou ambos possuírem o tipo valendo `None`, o resultado da comparação será decidido apenas se o número for idêntico. Se as duas instâncias de `Telefone` possuírem um tipo válido (diferente de `None`), então o tipo fará parte da comparação, sendo iguais apenas se os números e os tipos forem iguais. Essas decisões foram tomadas para a implementação da agenda e são decisões de projeto. Isso significa que para outras aplicações, essas decisões poderiam ser diferentes. O processo ficará mais claro quando leremos o programa completo da agenda.

Agora que temos as classes `Nome`, `Telefone` e `TipoTelefone`, podemos passar a pensar em como organizar os objetos dessas classes. Podemos ver nossa agenda como uma grande lista, onde cada elemento é um composto de `Nome` com uma lista de objetos do tipo `Telefone`. Vamos chamar cada entrada em nossa agenda de `DadoAgenda` e implementar uma classe para agrupar instâncias das duas classes. Veja o programa da listagem 10.20 com a classe `DadoAgenda`.

► Listagem 10.20 – Classe `DadoAgenda`

```

import listaúnica
class Telefones(ListaÚnica):
    def __init__(self):
        super().__init__(Telefone)

```

```

class DadoAgenda:
    def __init__(self, nome):
        self.nome = nome
        self.telefones = Telefones()
    @property
    def nome(self):
        return self.__nome
    @nome.setter
    def nome(self, valor):
        if type(valor) != Nome:
            raise TypeError("nome deve ser uma instância da classe Nome")
        self.__nome = valor
    def pesquisaTelefone(self, telefone):
        posição = self.telefones.pesquisa(Telefone(telefone))
        if posição == -1:
            return None
        else:
            return self.telefones[posição]

```

Como cada objeto de `DadoAgenda` pode conter vários telefones, criamos uma classe `Telefones` que herda da classe `ListaÚnica` seu comportamento. Dessa forma `self.nome` de `DadoAgenda` é uma instância da classe `Nome`, e `self.telefones` uma instância de `Telefones`. Veja que adicionamos uma propriedade `nome` para facilitar o acesso ao objeto de `self.__nome` e fazer as verificações de tipo necessárias. Adicionamos também um método `pesquisaTelefone` que transforma uma string em um objeto da classe `Telefone`, sem tipo. Esse objeto é então utilizado pelo método `pesquisa`, vindo da implementação original de `ListaÚnica` que retorna a posição do objeto na lista, ou `-1`, caso ele não seja encontrado. Veja que o método `pesquisaTelefone` retorna uma instância de `Telefone` se ela for encontrada na lista ou `None`, caso contrário.

Vejamos agora a classe `Agenda` na listagem 10.21.

► Listagem 10.21 – Listagem parcial do programa da agenda

```

class TiposTelefone(ListaÚnica):
    def __init__(self):
        super().__init__(TipoTelefone)

```

```

class Agenda(ListaÚnica):
    def __init__(self):
        super().__init__(DadoAgenda)
        self.tiposTelefone = TiposTelefone()
    def adicionaTipo(self, tipo):
        self.tiposTelefone.adiciona(TipoTelefone(tipo))
    def pesquisaNome(self, nome):
        if type(nome) == str:
            nome = Nome(nome)
        for dados in self.lista:
            if dados.nome == nome:
                return dados
        else:
            return None
    def ordena(self):
        super().ordena(lambda dado: str(dado.nome))

```

O programa da listagem 10.21 é uma listagem parcial, apenas para a explicação da classe `Agenda`. O programa completo precisa importar as definições de classe feitas anteriormente. Observe que a classe `Agenda` é uma subclasse de `ListaÚnica` configurada para aceitar somente objetos do tipo `DadoAgenda`. Veja que definimos uma classe `TiposTelefone`, também herdando de `ListaÚnica`, para manter a lista de tipos de telefone. Em nossa agenda, os tipos de telefone são pré-configurados na classe `Agenda`. Para facilitar o trabalho de inclusão de novos tipos, incluímos o método `adicionaTipo` que prepara uma string, transformando-a em objeto de `TipoTelefone` e o incluindo na lista de tipos válidos. Outro método a notar é `pesquisaNome`, que recebe o nome a pesquisar como objeto da classe `Nome` ou como string. Veja que utilizamos a verificação do tipo do parâmetro `nome` para transformá-lo em objeto de `Nome`, caso necessário, deixando nossa classe funcionar com strings ou com objetos do tipo `Nome`. A função então pesquisa na lista interna de dados e retorna o objeto, caso encontrado. É importante notar a diferença dessa função de pesquisa e outras funções de pesquisa que definimos anteriormente. Em `pesquisaNome`, o objeto é retornado ou `None`, caso não seja encontrado. Da mesma forma que vimos o problema de referências em listas no capítulo 6, vamos utilizar as referências retornadas por `pesquisaNome` para editar os valores da instância de `DadoAgenda` diretamente, sem a necessidade de reinseri-los na lista. Isso ficará mais claro quando analisarmos o programa da agenda completo. Para finalizar, o método `ordena` cria uma função para extrair a chave de ordenação de `DadoAgenda`; nesse caso, o `nome`.

Antes de passarmos para o programa da agenda, vamos construir uma classe `Menu` para exibir o menu principal. Veja o programa da listagem 10.22.

► Listagem 10.22 – Listagem parcial da agenda: classe `Menu`

```
class Menu:
    def __init__(self):
        self.opções = [ ["Sair", None] ]
    def adicionaopção(self, nome, função):
        self.opções.append([nome, função])
    def exibe(self):
        print("====")
        print("Menu")
        print("====\n")
        for i, opção in enumerate(self.opções):
            print("[{0}] - {1}".format(i, opção[0]))
        print()
    def execute(self):
        while True:
            self.exibe()
            escolha = valida_faixa_inteiro("Escolha uma opção: ",
                0, len(self.opções)-1)
            if escolha == 0:
                break
            self.opções[escolha][1]()

```

Nosso menu é bem simples, adicionando a opção para "Sair" como valor padrão. O método `exibe` mostra o menu na tela, percorrendo a lista de opções. Já o método `execute` mostra continuamente o menu, pedindo uma escolha e chamando o método correspondente. Vejamos como utilizar a classe `Menu` na inicialização da classe `AppAgenda` na listagem 10.23:

► Listagem 10.23 – Listagem completa da nova agenda

```
import sys
import pickle
from functools import total_ordering
def nulo_ou_vazio(texto):
    return texto == None or not texto.strip()
```

```
def valida_faixa_inteiro(pergunta, inicio, fim, padrão = None):
    while True:
        try:
            entrada = input(pergunta)
            if nulo_ou_vazio(entrada) and padrão != None:
                entrada = padrão
            valor = int(entrada)
            if inicio <= valor <= fim:
                return(valor)
        except ValueError:
            print("Valor inválido, favor digitar entre %d e %d" %
                (inicio, fim))
def valida_faixa_inteiro_ou_branco(pergunta, inicio, fim):
    while True:
        try:
            entrada = input(pergunta)
            if nulo_ou_vazio(entrada):
                return None
            valor = int(entrada)
            if inicio <= valor <= fim:
                return(valor)
        except ValueError:
            print("Valor inválido, favor digitar entre %d e %d" %
                (inicio, fim))
class ListaÚnica:
    def __init__(self, elem_class):
        self.lista = []
        self.elem_class = elem_class
    def __len__(self):
        return len(self.lista)
    def __iter__(self):
        return iter(self.lista)
    def __getitem__(self, p):
        return self.lista[p]
```

```

def indiceVálido(self, i):
    return i >= 0 and i < len(self.lista)
def adiciona(self, elem):
    if self.pesquisa(elem) == -1:
        self.lista.append(elem)
def remove(self, elem):
    self.lista.remove(elem)
def pesquisa(self, elem):
    self.verifica_tipo(elem)
    try:
        return self.lista.index(elem)
    except ValueError:
        return -1
def verifica_tipo(self, elem):
    if type(elem) != self.elem_class:
        raise TypeError("Tipo inválido")
def ordena(self, chave = None):
    self.lista.sort(key=chave)

@total_ordering
class Nome:
    def __init__(self, nome):
        self.nome = nome
    def __str__(self):
        return self.nome
    def __repr__(self):
        return "<Classe {} em 0x{:x} Nome: {} Chave: {}>".format(
            id(self), self.__nome, self.__chave,
            type(self).__name__)
    def __eq__(self, outro):
        return self.nome == outro.nome
    def __lt__(self, outro):
        return self.nome < outro.nome
@property
def nome(self):
    return self.__nome

```

```

@nome.setter
def nome(self, valor):
    if nulo_ou_vazio(valor):
        raise ValueError("Nome não pode ser nulo nem em branco")
    self.__nome = valor
    self.__chave = Nome.CriaChave(valor)
@property
def chave(self):
    return self.__chave
@staticmethod
def CriaChave(nome):
    return nome.strip().lower()

@total_ordering
class TipoTelefone:
    def __init__(self, tipo):
        self.tipo = tipo
    def __str__(self):
        return "({})".format(self.tipo)
    def __eq__(self, outro):
        if outro is None:
            return False
        return self.tipo == outro.tipo
    def __lt__(self, outro):
        return self.tipo < outro.tipo
class Telefone:
    def __init__(self, número, tipo=None):
        self.número = número
        self.tipo = tipo
    def __str__(self):
        if self.tipo != None:
            tipo = self.tipo
        else:
            tipo = ""
        return "{} {}".format(self.número, tipo)

```

```

def __eq__(self, outro):
    return self.número == outro.número and (
        (self.tipo == outro.tipo) or (
            self.tipo == None or outro.tipo == None))

@property
def número(self):
    return self._número

@número.setter
def número(self, valor):
    if nulo_ou_vazio(valor):
        raise ValueError("Número não pode ser None ou em branco")
    self._número = valor

class Telefones(ListaÚnica):
    def __init__(self):
        super().__init__(Telefone)

class TiposTelefone(ListaÚnica):
    def __init__(self):
        super().__init__(TipoTelefone)

class DadoAgenda:
    def __init__(self, nome):
        self.nome = nome
        self.telefones = Telefones()

    @property
    def nome(self):
        return self._nome

    @nome.setter
    def nome(self, valor):
        if type(valor) != Nome:
            raise TypeError("nome deve ser uma instância da classe Nome")
        self._nome = valor

    def pesquisaTelefone(self, telefone):
        posição = self.telefones.pesquisa(Telefone(telefone))
        if posição == -1:
            return None
        else:
            return self.telefones[posição]

```

```

class Agenda(ListaÚnica):
    def __init__(self):
        super().__init__(DadoAgenda)
        self.tiposTelefone = TiposTelefone()

    def adicionaTipo(self, tipo):
        self.tiposTelefone.adiciona(TipoTelefone(tipo))

    def pesquisaNome(self, nome):
        if type(nome) == str:
            nome = Nome(nome)
        for dados in self.lista:
            if dados.nome == nome:
                return dados
        else:
            return None

    def ordena(self):
        super().ordena(lambda dado: str(dado.nome))

class Menu:
    def __init__(self):
        self.opções = [ ["Sair", None] ]

    def adicionaopção(self, nome, função):
        self.opções.append([nome, função])

    def exibe(self):
        print("====")
        print("Menu")
        print("====\n")
        for i, opção in enumerate(self.opções):
            print("[{0}] - {1}".format(i, opção[0]))
        print()

    def execute(self):
        while True:
            self.exibe()
            escolha = valida_faixa_inteiro("Escolha uma opção: ",
                                            0, len(self.opções)-1)
            if escolha == 0:
                break
            self.opções[escolha][1]()

```

```

class AppAgenda:
    @staticmethod
    def pede_nome():
        return(input("Nome: "))
    @staticmethod
    def pede_telefone():
        return(input("Telefone: "))
    @staticmethod
    def mostra_dados(dados):
        print("Nome: %s" % dados.nome)
        for telefone in dados.telefones:
            print("Telefone: %s" % telefone)
        print()
    @staticmethod
    def mostra_dados_telefone(dados):
        print("Nome: %s" % dados.nome)
        for i, telefone in enumerate(dados.telefones):
            print("{0} - Telefone: {1}".format(i, telefone))
        print()
    @staticmethod
    def pede_nome_arquivo():
        return(input("Nome do arquivo: "))
    def __init__(self):
        self.agenda = Agenda()
        self.agenda.adicionaTipo("Celular")
        self.agenda.adicionaTipo("Residência")
        self.agenda.adicionaTipo("Trabalho")
        self.agenda.adicionaTipo("Fax")
        self.menu = Menu()
        self.menu.adicionaopção("Novo", self.novo)
        self.menu.adicionaopção("Altera", self.altera)
        self.menu.adicionaopção("Apaga", self.apaga)
        self.menu.adicionaopção("Lista", self.lista)
        self.menu.adicionaopção("Grava", self.grava)
        self.menu.adicionaopção("Lê", self.lê)

```

```

        self.menu.adicionaopção("Ordena", self.ordena)
        self.ultimo_nome = None
    def pede_tipo_telefone(self, padrão = None):
        for i,tipo in enumerate(self.agenda.tiposTelefone):
            print(" {0} - {1} ".format(i, tipo), end=None)
        t = valida_faixa_inteiro("Tipo: ", 0, len(self.agenda.tiposTelefone)-1, padrão)
        return self.agenda.tiposTelefone[t]
    def pesquisa(self, nome):
        dado = self.agenda.pesquisaNome(nome)
        return dado
    def novo(self):
        novo = AppAgenda.pede_nome()
        if nulo_ou_vazio(novo):
            return
        nome = Nome(novo)
        if self.pesquisa(nome) != None:
            print("Nome já existe!")
            return
        registro = DadoAgenda(nome)
        self.menu_telefones(registro)
        self.agenda.adiciona(registro)
    def apaga(self):
        if len(self.agenda)==0:
            print("Agenda vazia, nada a apagar")
            nome = AppAgenda.pede_nome()
            if(nulo_ou_vazio(nome)):
                return
            p = self.pesquisa(nome)
            if p != None:
                self.agenda.remove(p)
                print("Apagado. A agenda agora possui apenas: %d registros" % len(self.agenda))
            else:
                print("Nome não encontrado.")
    def altera(self):
        if len(self.agenda)==0:
            print("Agenda vazia, nada a alterar")

```

```

nome = AppAgenda.pede_nome()
if(nulo_ou_vazio(nome)):
    return
p = self.pesquisa(nome)
if p != None:
    print("\nEncontrado:\n")
    AppAgenda.mostra_dados(p)
    print("Digite enter caso não queira alterar o nome")
    novo = AppAgenda.pede_nome()
    if not nulo_ou_vazio(novo):
        p.nome = Nome(novo)
    self.menu_telefones(p)
else:
    print("Nome não encontrado!")
def menu_telefones(self, dados):
    while True:
        print("\nEditando telefones\n")
        AppAgenda.mostra_dados_telefone(dados)
        if(len(dados.telefones)>0):
            print("\n[A] - alterar\n[D] - apagar\n", end="")
        print("[N] - novo\n[S] - sair\n")
        operação = input("Escolha uma operação: ")
        operação = operação.lower()
        if operação not in ["a","d","n", "s"]:
            print("Operação inválida. Digite A, D, N ou S")
            continue
        if operação == 'a' and len(dados.telefones)>0:
            self.altera_telefones(dados)
        elif operação == 'd' and len(dados.telefones)>0:
            self.apaga_telefone(dados)
        elif operação == 'n':
            self.novo_telefone(dados)
        elif operação == "s":
            break

```

```

def novo_telefone(self, dados):
    telefone = AppAgenda.pede_telefone()
    if nulo_ou_vazio(telefone):
        return
    if dados.pesquisaTelefone(telefone) != None:
        print("Telefone já existe")
        tipo = self.pede_tipo_telefone()
        dados.telefones.adiciona(Telefone(telefone, tipo))
def apaga_telefone(self, dados):
    t = valida_faixa_inteiro_ou_branco(
        "Digite a posição do número a apagar, enter para sair: ",
        0, len(dados.telefones)-1)
    if t == None:
        return
    dados.telefones.remove(dados.telefones[t])
def altera_telefones(self, dados):
    t = valida_faixa_inteiro_ou_branco(
        "Digite a posição do número a alterar, enter para sair: ",
        0, len(dados.telefones)-1)
    if t == None:
        return
    telefone = dados.telefones[t]
    print("Telefone: %s" % telefone)
    print("Digite enter caso não queira alterar o número")
    novotelefone = AppAgenda.pede_telefone()
    if not nulo_ou_vazio(novotelefone):
        telefone.numero = novotelefone
    print("Digite enter caso não queira alterar o tipo")
    telefone.tipo = self.pede_tipo_telefone(
        self.agenda.tiposTelefone.pesquisa(telefone.tipo))
def lista(self):
    print("\nAgenda")
    print("-"*60)
    for e in self.agenda:
        AppAgenda.mostra_dados(e)
    print("-"*60)

```

```

def lê(self, nome_arquivo = None):
    if nome_arquivo == None:
        nome_arquivo = AppAgenda.pede_nome_arquivo()
    if nulo_ou_vazio(nome_arquivo):
        return
    with open(nome_arquivo, "rb") as arquivo:
        self.agenda = pickle.load(arquivo)
    self.ultimo_nome = nome_arquivo
def ordena(self):
    self.agenda.ordena()
    print("\nAgenda ordenada\n")
def grava(self):
    if self.ultimo_nome != None:
        print("Último nome utilizado foi '%s'" % self.ultimo_nome)
        print("Digite enter caso queira utilizar o mesmo nome")
    nome_arquivo = AppAgenda.pede_nome_arquivo()
    if nulo_ou_vazio(nome_arquivo):
        if self.ultimo_nome != None:
            nome_arquivo = self.ultimo_nome
        else:
            return
    with open(nome_arquivo, "wb") as arquivo:
        pickle.dump(self.agenda, arquivo)
def execute(self):
    self.menu.execute()
if __name__ == "__main__":
    app = AppAgenda()
    if len(sys.argv) > 1:
        app.lê(sys.argv[1])
    app.execute()

```

A listagem 10.23 contém o programa completo, onde todas as classes necessárias foram escritas em um só arquivo para facilitar a leitura. A classe AppAgenda é nossa aplicação agenda em si. Veja a implementação de `__init__`, onde criamos uma instância de Agenda para conter nossos dados e populamos os tipos de telefone que queremos trabalhar. Também criamos uma instância de Menu e adicionamos

as opções. Observe que passamos o nome da opção e o método correspondente a cada escolha. O atributo `self.ultimo_nome` é usado para guardar o nome usado na última leitura do arquivo.

Na nova agenda, note que dividimos o tratamento de nomes e telefones. Como podemos ter vários telefones, outro menu com opções para o gerenciamento de telefones aparece. Você pode ver os detalhes na implementação do método `menu_telefones`. Em nossa agenda, adotamos o seguinte comportamento quando um valor não muda, digitamos apenas `Enter`. Em todas as opções, você deve perceber o tratamento de entradas em branco, até criarmos uma função de suporte para isso: `nulo_ou_vazio`.

Em todos os métodos de edição, você pode perceber que utilizamos apenas os métodos fornecidos pela classe `ListaÚnica`. Veja também que as pesquisas retornam os objetos e não apenas a posição deles na lista.

Os métodos `grava` e `lê` foram modificados para utilizar o módulo `pickle` do Python. Como nossa agenda agora possui vários telefones, cada um com um tipo, a utilização de arquivos simples se tornaria muito trabalhosa. O módulo `pickle` fornece métodos que gravam um objeto ou uma lista de objetos em disco. No método `grava`, utilizamos a função `pickle.dump` que grava no arquivo o objeto agenda inteiro. No método `lê`, utilizamos `pickle.load` para recriar nossa agenda a partir do arquivo em disco. A utilização do `pickle` facilita muito a tarefa de serializar os dados em um arquivo, ou seja, representar os objetos de forma que possam ser reconstruídos, caso necessário (deserialização).

Como última modificação, adicionamos a possibilidade de passar o nome do arquivo da agenda como parâmetro. Caso o nome seja passado, o arquivo é lido antes de apresentar o menu. Observe também que o método `execute` tem um loop infinito que roda até sairmos do menu principal.