Capítulo 12

Programação com objetos

And I haven't seen the two Messengers, either. They're both gone to the town. Just look along the road, and tell me if you can see either of them.

Lewis Carroll, *Through the Looking Glass*

Neste capítulo introduzimos o conceito de objeto, uma entidade que contém um conjunto de variáveis internas e um conjunto de operações que manipulam essas variáveis, e abordamos alguns aspetos associados à programação com objetos.

A utilização de objetos em programação dá origem a um paradigma de programação conhecido por *programação com objetos*[1].

12.1 Objetos

Na secção 9.4 discutimos a necessidade de garantir que, após definido um tipo abstrato de dados, é proibido o acesso direto à sua representação. Dissemos também que a abordagem que usámos não nos permite impor esta proibição.

As linguagens de programação desenvolvidas antes do aparecimento da metodologia para os tipos abstratos de dados não possuem mecanismos para garantir que toda a utilização dos elementos de um dado tipo é efetuada recorrendo ex-

[1]Do inglês, "object-oriented programming". Em português, este tipo de programação também é conhecido por programação orientada a objetos, programação orientada por objetos, programação por objetos e programação orientada aos objetos.

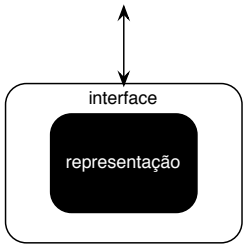

Figura 12.1: Módulo correspondente a um tipo abstrato de dados.

clusivamente às operações específicas desse tipo. Algumas das linguagens de programação mais recentes (por exemplo, Java e C++) fornecem mecanismos que garantem que as manipulações efetuadas sobre os elementos de um tipo apenas utilizam as operações básicas desse tipo. Embora o Python forneça estes mecanismos, os quais serão usados para apresentar os conceitos deste capítulo, alguns deles podem ser contornados.

A proibição do acesso direto à representação de um tipo abstrato é conseguida recorrendo a dois conceitos chamados encapsulação de dados e anonimato da representação.

A *encapsulação de dados*[2] corresponde ao conceito de que o conjunto de funções que corresponde ao tipo de dados engloba toda a informação referente ao tipo. Estas funções estão representadas dentro de um módulo que está protegido dos acessos exteriores. Este módulo comporta-se de certo modo como um bloco, com a diferença que "exporta" certas operações, permitindo apenas o acessos às operações exportadas. O *anonimato da representação*[3] corresponde ao conceito de que este módulo guarda como segredo o modo escolhido para representar os elementos do tipo. O único acesso que o programa tem aos elementos do tipo é através das operações básicas definidas dentro do módulo que corresponde ao tipo.

Na Figura 12.1 apresentamos de uma forma esquemática o módulo concetual correspondente a um tipo abstrato de dados. Este módulo contém uma parte que engloba a representação do tipo, representação essa que está escondida do resto do programa, e um conjunto de funções que efetuam a comunicação (interface)

[2]Do inglês, "data encapsulation".
[3]Do inglês, "information hiding".

entre o programa e a representação interna.

A vantagem das linguagens que incorporam os mecanismos da metodologia dos tipos abstratos de dados reside no facto de estas *proibirem efetivamente* a utilização de um tipo abstrato através da manipulação direta da sua representação: toda a informação relativa ao tipo está contida no módulo que o define, o qual guarda como segredo a representação escolhida para o tipo.

De modo a poder garantir a encapsulação de dados e anonimato da representação em Python, precisamos de recorrer ao conceito de objeto. Um *objeto* é uma entidade computacional que apresenta, não só informação interna (no caso dos tipos de dados, a representação de um elemento de um tipo), como também um conjunto de operações que definem o seu comportamento.

A manipulação de objetos não é baseada em funções que calculam valores, mas sim no conceito de uma entidade que recebe solicitações e que reage apropriadamente a essas solicitações, recorrendo à informação interna do objeto. As funções associadas a um objeto são vulgarmente designadas por *métodos*. É importante distinguir entre solicitações e métodos: uma solicitação é um pedido feito a um objeto; um método é a função utilizada pelo objeto para processar essa solicitação.

Para definir objetos em Python, criamos um módulo (chamado uma *classe*) com o nome do objeto, associando-lhe os métodos correspondentes a esse objeto. Em notação BNF, um objeto em Python é definido do seguinte modo[4]:

⟨definição de objeto⟩ ::= `class` ⟨nome⟩ {`(`⟨nome⟩`)`}`:` CR
TAB ⟨definição de método⟩$^+$

⟨definição de método⟩ ::= `def` ⟨nome⟩ `(self`{`,` ⟨parâmetros formais⟩}`):` CR
TAB ⟨corpo⟩

Os símbolos não terminais ⟨nome⟩, ⟨parâmetros formais⟩ e ⟨corpo⟩ foram definidos nos capítulos 2 e 3.

Nesta definição:

1. a palavra `class` (um símbolo terminal) indica que se trata da definição de um objeto. Existem duas alternativas para esta primeira linha: (1) pode apenas ser definido um nome, ou (2) pode ser fornecido um nome seguido

[4]Esta definição corresponde em Python a uma ⟨definição⟩.

de outro nome entre parênteses. O nome imediatamente após a palavra `class` representa o nome do objeto que está a ser definido;

2. A ⟨definição de método⟩ corresponde à definição de um método que está associado ao objeto. Podem ser definidos tantos métodos quantos se desejar, existindo tipicamente pelo menos um método cujo nome é `__init__` (este aspeto não está representado na nossa expressão em notação BNF);

3. Os parâmetros formais de um método contêm sempre, como primeiro elemento, um parâmetro com o nome `self`[5].

Os objetos contêm informação interna. Esta informação interna é caraterizada por uma ou mais variáveis. Os nomes destas variáveis utilizam a notação de ⟨nome composto⟩ apresentada na página 97, em que o primeiro nome é sempre `self`.

Para caraterizar a informação interna associada a um objeto correspondente a um número complexo, sabemos que precisamos de duas entidades, nomeadamente, a parte real e a parte imaginária desse número complexo. Vamos abandonar a ideia de utilizar um dicionário para representar números complexos, optando por utilizar duas variáveis separadas, `self.r` e `self.i`, hipótese que tínhamos abandonado à partida no início deste capítulo (página 267). A razão da nossa decisão prende-se com o facto destas duas variáveis existirem dentro do objeto que corresponde a um número complexo e consequentemente, embora sejam variáveis separadas, elas estão mutuamente ligadas por existirem dentro do mesmo objeto.

Consideremos a seguinte definição de um objeto com o nome `compl`[6] correspondente a um número complexo:

```
class compl:

    def __init__(self, real, imag):
        if isinstance(real, (int, float)) and \
           isinstance(imag, (int, float)):
```

[5] A palavra inglesa "self" corresponde à individualidade ou identidade de uma pessoa ou de uma coisa.

[6] Por simplicidade de apresentação, utilizamos o operador relacional `==` para comparar os elementos de um número complexo, quando, na realidade, deveríamos utilizar os predicados `zero` (apresentado na página 278) e `igual` (apresentado na página 279).

```
            self.r = real
            self.i = imag
        else:
            raise ValueError ('complexo: argumento errado')

    def p_real(self):
        return self.r

    def p_imag(self):
        return self.i

    def compl_zero(self):
        return self.r == 0 and self.i == 0

    def imag_puro(self):
        return self.r == 0

    def compl_iguais(self, outro):
        return self.r == outro.p_real() and \
               self.i == outro.p_imag()

    def escreve(self):
        if self.i >= 0:
            print(str(self.r) + '+' + str(self.i) + 'i')
        else:
            print(str(self.r) + '-' + str(abs(self.i)) + 'i')
```

A execução desta definição pelo Python dá origem à criação do objeto `compl`, estando este objeto associado a métodos que correspondem aos nomes `__init__`, `p_real`, `p_imag`, `compl_zero`, `imag_puro`, `compl_iguais` e `escreve`.

A partir do momento em que um objeto é criado, passa a existir uma nova função em Python cujo nome corresponde ao nome do objeto e cujos parâmetros correspondem a todos os parâmetros formais da função `__init__` que lhe está associada, com exceção do parâmetro `self`. O comportamento desta função é definido pela função `__init__` associada ao objeto, a qual devolve um objecto. No nosso exemplo, esta é a função `compl` de dois argumentos. A chamada à função `compl(3, 2)` cria o objeto correspondente ao número complexo $3 + 2\,i$, devolvendo esse objeto. Podemos, assim, originar a interação:

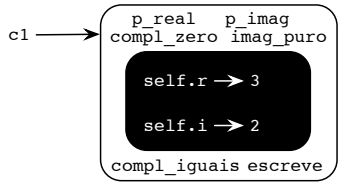

Figura 12.2: Objeto correspondente ao número complexo $3 + 2\,i$.

```
>>> compl
<class '__main__.compl'>
>>> c1 = compl(3, 2)
>>> c1
<__main__.compl object at 0x11fab90>
```

As duas primeiras linhas desta interação mostram que `compl` é uma classe (um tipo de entidades computacionais que existem nos nossos programas) e as duas últimas linhas mostram que o nome `c1` está associado a um objeto, localizado na posição de memória `0x11fab90`.

O resultado da execução da instrução `c1 = compl(3, 2)`, foi a criação do objeto apresentado na Figura 12.2 e a associação do nome `c1` a este objeto. Este objeto "esconde" do exterior a representação do seu número complexo (dado pelos valores das variáveis `self.r` e `self.i`, respetivamente, `3` e `2`), fornecendo seis métodos para interactuar com o número complexo, cujos nomes são `p_real`, `p_imag`, `compl_zero`, `imag_puro`, `compl_iguais` e `escreve`.

Os métodos associados a um objeto correspondem a funções que manipulam as variáveis associadas ao objeto, variáveis essas que estão guardadas *dentro* do objeto, correspondendo a um ambiente local ao objeto, o qual existe enquanto o objeto correspondente existir. Para invocar os métodos associados a um objeto utiliza-se a notação de ⟨nome composto⟩ apresentada na página 97, em que o nome é composto pelo nome do objeto, seguido do nome do método[7]. Por exemplo `c1.p_real` é o nome do método que devolve a parte real do complexo correspondente a `c1`. Para invocar a execução de um método, utilizam-se todos os parâmetros formais da função correspondente ao método com exceção do

[7]Note-se que a função `lower` apresentada na página 243 do Capítulo 8 e as funções que manipulam ficheiros apresentadas no Capítulo 11, na realidade correspondem a métodos.

parâmetro `self`[8]. Assim, podemos originar a seguinte interação:

```
>>> c1 = compl(3, 2)
>>> c1.p_real()
3
>>> c1.p_imag()
2
>>> c1.escreve()
3+2i
>>> c2 = compl(2, -2)
>>> c2.escreve()
2-2i
```

Na interação anterior criámos dois objetos `c1` e `c2` correspondentes a números complexos. Cada um destes objetos armazena a parte inteira e a parte imaginária de um número complexo, apresentando comportamentos semelhantes, apenas diferenciados pela sua identidade, um corresponde ao complexo $3 + 2\,i$ e o outro ao complexo $2 - 2\,i$.

Devemos fazer as seguintes observações em relação à nossa definição do objeto `compl`:

1. Não definimos um reconhecedor correspondente à operação básica *e_complexo*. Na realidade, ao criar um objeto, o Python associa automaticamente o nome do objeto à entidade que o representa. A função embutida `type` devolve o tipo do seu argumento, como o mostra a seguinte interação:

   ```
   >>> c1 = compl(3, 5)
   >>> type(c1)
   <class '__main__.compl'>
   ```

   A função `isinstance`, introduzida na página 277, permite determinar se uma dada entidade corresponde a um complexo:

   ```
   >>> c1 = compl(9, 6)
   >>> isinstance(c1, compl)
   True
   ```

[8] Na realidade, o nome do objeto é associado ao parâmetro `self`.

Existe assim, uma solicitação especial que pode ser aplicada a qualquer objeto e que pede a identificação do objeto. Por esta razão, ao criarmos um tipo abstrato de dados recorrendo a objetos, não necessitamos de escrever o reconhecedor que distingue os elementos do tipo dos restantes elementos.

2. Na definição das operações básicas para complexos que apresentámos no Capítulo 9, definimos a função *compl_iguais* (ver página 275) que permite decidir se dois complexos são iguais. Na nossa implementação, o método `compl_iguais` apenas tem um argumento correspondente a um complexo. Este método compara o complexo correspondente ao seu objeto (`self`) com outro complexo qualquer. Assim, por omissão, o primeiro argumento da comparação com outro complexo é o complexo correspondente ao objeto em questão, como a seguinte interação ilustra:

```
>>> c1 = compl(9, 6)
>>> c2 = compl(3, 2)
>>> c3 = compl(9, 6)
>>> c1.compl_iguais(c2)
False
>>> c1.compl_iguais(c3)
True
```

3. Quando no Capítulo 9 definimos as operações básicas para complexos, dissemos que essas operações básicas deveriam ser usadas na utilização de complexos, independentemente da escolha da representação. No entanto, as operações que apresentámos da página 324 à página 325 a interação que apresentámos com números complexos foge à terminologia das operações básicas apresentada no Capítulo 9.

   No entanto, definindo as seguintes funções:

```
def cria_compl(r, i):
    return compl(r, i)

def p_real(c):
    return c.p_real()

def p_imag(c):
    return c.p_imag()
```

```
def e_complexo(c):
    return isinstance(c, compl)

def e_compl_zero(c):
    return c.compl_zero()

def e_imag_puro(c):
    return c.imag_puro()

def compl_iguais(c1, c2):
    return c1.compl_iguais(c2)

def escreve_compl(c):
    c.escreve()
```

Podemos obter a mesma interação do que a apresentada na página 9.2[9]:

```
>>> c1 = cria_compl(3, 5)
>>> p_real(c1)
3
>>> p_imag(c1)
5
>>> escreve_compl(c1)
3+5i
>>> c2 = cria_compl(1, -3)
>>> escreve_compl(c2)
1-3i
```

Em programação com objetos existe uma abstração chamada classe. Uma *classe* corresponde a uma infinidade de objetos com as mesmas variáveis e com o mesmo comportamento. É por esta razão, que na definição de um objeto o Python usa a palavra "`class`"(ver a definição apresentada na página 323). Assim, os complexos correspondem a uma classe de objetos que armazenam uma parte real, `self.r`, e uma parte imaginária, `self.i`, e cujo comportamento é definido pelas funções `__init__`, `p_real`, `p_imag`, `compl_zero`, `imag_puro`, `compl_iguais` e `escreve`. Os complexos `c1` e `c2` criados na interação anterior correspondem

[9]Com exceção das operações `soma_compl` e `subtrai_compl` que só definimos na Secção 12.6.

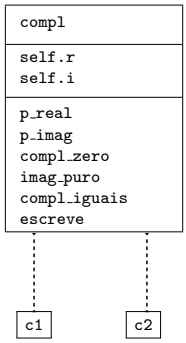

Figura 12.3: Representação da classe `compl` e duas de suas instâncias.

a instanciações do objeto `compl`, sendo conhecidos como *instâncias* da classe `compl`.

Uma classe é um potencial gerador de instâncias, objetos cujas variáveis e comportamento são definidos pela classe. São as instâncias que contêm os valores das variáveis associadas à classe e que reagem a solicitações – a classe apenas define como são caraterizadas as variáveis e o comportamento das suas instâncias.

As classes podem ser representadas graficamente como se indica na Figura 12.3. Uma classe é representada por um retângulo com três partes, contendo, respetivamente, o nome da classe, o nome das variáveis e o nome dos métodos. As instâncias são caraterizadas pelo seu nome (dentro de um retângulo) e estão ligadas à classe correspondente por uma linha a tracejado.

12.2 Objetos mutáveis

Na seção anterior apresentámos um primeiro exemplo de objetos cujas instâncias correspondem a constantes. Nesta seção começamos por apresentar objetos cujas variáveis internas variam ao longo do tempo, por esta razão, correspondentes um *tipo mutável*, e analisamos qual a influência que este aspeto apresenta no seu comportamento.

Dizemos que uma entidade tem *estado* se o seu comportamento é influenciado pela sua história. Vamos ilustrar o conceito de entidade com estado através da manipulação de uma conta bancária. Para qualquer conta bancária, a resposta

à questão "posso levantar 20 euros?" depende da história dos depósitos e dos levantamentos efetuados nessa conta. A situação de uma conta bancária pode ser caraterizada por um dos seguintes aspetos: o saldo num dado instante, ou a sequência de todos os movimentos na conta desde a sua criação. A situação da conta bancária é caraterizada computacionalmente através do conceito de estado. O estado de uma entidade é caraterizado por uma ou mais variáveis, as *variáveis de estado*, as quais mantêm informação sobre a história da entidade, de modo a poder determinar o comportamento da entidade.

Suponhamos que desejávamos modelar o comportamento de uma conta bancária. Por uma questão de simplificação, caraterizamos uma conta bancária apenas pelo seu saldo, ignorando todos os outros aspetos, que sabemos estarem associados a contas bancárias, como o número, o titular, o banco, a agência, etc.

Para o tipo conta definimos as seguintes operações básicas[10]:

1. *Construtores*:

   - $cria\_conta : inteiro \mapsto conta$
     $cria\_conta(s)$ tem como valor uma conta bancária cujo saldo inicial é s.

2. *Seletores*:

   - $consulta : conta \mapsto inteiro$
     $consulta(c)$ tem como valor o saldo da conta c.

3. *Modificadores*:

   - $deposito : conta \times inteiro \mapsto inteiro$
     $deposito(c, q)$ altera destrutivamente o valor do saldo, s, da conta c para $s + q$, tendo como valor $s + q$.
   - $levantamento : conta \times inteiro \mapsto inteiro$
     O valor de $levantamento(c, q)$ depende da relação entre o saldo da conta c e a quantia q: se o saldo, s, da conta c for maior ou igual à quantia, significando que a quantia pode ser removida do saldo, sem que o saldo se torne negativo, muda destrutivamente o valor do saldo

[10] Assumimos, sem perda de generalidade, que o saldo de uma conta e os valores dos depósitos e levantamentos correspondem a inteiros.

para $s-q$, devolvendo o valor do novo saldo, $s-q$; em caso contrário, não altera o saldo da conta e o seu valor é indefinido. Ou seja:

$$levantamento(c,q) = \begin{cases} consulta(c) - q & \text{se } consulta(c) \geq q \\ \bot & \text{se } consulta(c) < q \end{cases}$$

4. *Reconhecedores*:

   - $e\_conta : universal \mapsto lógico$
     $e\_conta(arg)$ tem o valor $verdadeiro$ se arg é uma conta e tem o valor $falso$ em caso contrário.

Para modelar o comportamento de uma conta bancária, utilizamos uma classe, chamada `conta`, cujo estado interno é definido pela variável `saldo` e está associada a funções que efetuam depósitos, levantamentos e consultas de saldo. Esta classe, para além do construtor (`__init__`), apresenta um seletor que devolve o valor do saldo (`consulta`) e dois modificadores, `deposito` e `levantamento`, que, respetivamente, correspondem às operações de depósito e de levantamento.

```
class conta:

    def __init__(self, quantia):
        self.saldo = quantia

    def consulta(self):
        return self.saldo

    def deposito(self, quantia):
        self.saldo = self.saldo + quantia
        return self.saldo

    def levantamento(self, quantia):
        if self.saldo - quantia >= 0:
            self.saldo = self.saldo - quantia
            return self.saldo
        else:
            print('Saldo insuficiente')
```

Com esta classe podemos obter a interação:

```
>>> conta_01 = conta(100)
>>> conta_01.deposito(50)
150
>>> conta_01.consulta()
150
>>> conta_01.levantamento(120)
30
>>> conta_01.levantamento(50)
Saldo insuficiente
>>> conta_02 = conta(250)
>>> conta_02.deposito(50)
300
>>> conta_02.consulta()
300
```

De modo análogo ao que fizémos em relação aos número complexos, podemos definir funções que garantem que a interface com contas é feita através da assinatura do tipo conta:

```
def cria_conta(saldo):
    return conta(saldo)

def consulta(c):
    return c.consulta()

def deposito(c, q):
    return c.deposito(q)

def levantamento(c, q):
    return c.levantamento(q)

def e_conta(x):
    return isinstance(x, conta)
```

Obtendo a interação:

```
>>> conta_01 = cria_conta(100)
>>> deposito(conta_01, 50)
```

```
150
>>> consulta(conta_01)
150
>>> levantamento(conta_01, 120)
30
>>> levantamento(conta_01, 50)
Saldo insuficiente
>>> conta_02 = cria_conta(250)
>>> deposito(conta_02, 50)
300
>>> consulta(conta_02)
300
```

Daqui em diante, não apresentaremos mais funções que garantam a interface através da assinatura do tipo e utilizaremos a notação associada a objetos.

Uma conta bancária, tal como a que acabámos de definir, corresponde a uma entidade que não só possui um estado (o qual é caraterizado pelo seu saldo) mas também está associada a um conjunto de comportamentos (as funções que efetuam depósitos, levantamentos e consultas). A conta bancária evolui de modo diferente quando fazemos um depósito ou um levantamento.

Sabemos que em programação com objetos existe uma abstração chamada classe. Uma *classe* corresponde a uma infinidade de objetos com as mesmas variáveis de estado e com o mesmo comportamento. Por exemplo, a conta bancária que definimos corresponde a uma classe de objetos cujo estado é definido pela variável `saldo` e cujo comportamento é definido pelas funções `levantamento`, `deposito` e `consulta`. As contas `conta_01` e `conta_02`, definidas na nossa interação, correspondem a *instâncias* da classe `conta`. Já vimos que as classes e instâncias são representadas graficamente como se mostra na Figura 12.4.

Ao trabalhar com objetos, é importante distinguir entre identidade e igualdade. Considerando a classe `conta`, podemos definir as contas `conta_01` e `conta_02` do seguinte modo:

```
>>> conta_01 = conta(100)
>>> conta_02 = conta(100)
```

Será que os dois objetos `conta_01` e `conta_02` são iguais? Embora eles tenham

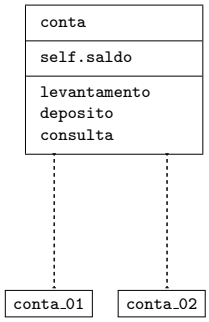

Figura 12.4: Representação da classe `conta` e suas instâncias.

sido definidos exatamente da mesma maneira, como instâncias da mesma classe, estes dois objetos têm estados distintos, como se mostra na seguinte interação:

```
>>> conta_01 = conta(100)
>>> conta_02 = conta(100)
>>> conta_01.levantamento(25)
75
>>> conta_01.levantamento(35)
40
>>> conta_02.deposito(120)
220
>>> conta_01.consulta()
40
>>> conta_02.consulta()
220
```

Suponhamos agora que definíamos:

```
>>> conta_01 = conta(100)
>>> conta_02 = conta_01
```

Repare-se que agora `conta_01` e `conta_02` são o *mesmo* objeto (correspondem na realidade a uma conta conjunta). Por serem o mesmo objeto, qualquer alteração a uma destas contas reflete-se na outra:

```
>>> conta_01 = conta(100)
```

```
>>> conta_02 = conta_01
>>> conta_01.consulta()
100
>>> conta_02.levantamento(30)
70
>>> conta_01.consulta()
70
```

12.3 Classes, subclasses e herança

Uma das vantagens da programação com objetos é a possibilidade de construir entidades reutilizáveis, ou seja, componentes que podem ser usados em programas diferentes através da especialização para necessidades específicas.

Para ilustrar o aspeto da reutilização, consideremos, de novo, a classe `conta`. Esta classe permite a criação de contas bancárias, com as quais podemos efetuar levantamentos e depósitos. Sabemos, no entanto, que qualquer banco oferece aos seus clientes a possibilidade de abrir diferentes tipos de contas, existem contas ordenado que permitem apresentar saldos negativos (até um certo limite), existem contas jovem que não permitem saldos negativos, mas que não impõem restrições mínimas para o saldo de abertura, mas, em contrapartida, não pagam juros, e assim sucessivamente. Quando um cliente abre uma nova conta, tipicamente define as caraterísticas da conta desejada: "quero uma conta em que não exista pagamento adicional pela utilização de cheques", "quero uma conta que ofereça 2% de juros por ano", etc.

Todos estes tipos de contas são "contas bancárias", mas apresentam caraterísticas (ou especializações) que variam de conta para conta. A programação tradicional, aplicada ao caso das contas bancárias, levaria à criação de funções em que o tipo de conta teria de ser testado quando uma operação de abertura, de levantamento ou de depósito era efetuada. Em programação com objetos, a abordagem seguida corresponde à criação de classes que especializam outras classes.

Comecemos por considerar uma classe de contas genéricas, `conta_gen`, a qual sabe efetuar levantamentos, depósitos e consultas ao saldo. Esta classe pressupõe que existe uma variável de estado `self.saldo_min` que define qual o

saldo mínimo após um levantamento.

A classe `conta_gen`, correspondente a uma conta genérica, pode ser realizada através da seguinte definição[11]. Note-se que a definição de `conta_gen` parece contradizer o que dissémos na página 324, no sentido em que não contém o método `__init__`. Este aspeto é clarificado mais à frente.

```
class conta_gen:

    def consulta(self):
        return self.saldo

    def deposito(self,quantia):
        self.saldo = self.saldo + quantia
        return self.saldo

    def levantamento(self,quantia):
        if self.saldo - quantia >= self.saldo_min:
            self.saldo = self.saldo - quantia
            return self.saldo
        else:
            print('Saldo insuficiente')
```

Suponhamos agora que estávamos interessados em criar os seguintes tipos de contas:

- *conta ordenado*: esta conta está associada à transferência mensal do ordenado do seu titular e está sujeita às seguintes condições:
  - — a sua abertura exige um saldo mínimo igual ao valor do ordenado a ser transferido para a conta;
  - — permite saldos negativos, até um montante igual ao ordenado.
- *conta jovem*: conta feita especialmente para jovens e que está sujeita às seguintes condições:
  - — a sua abertura não exige um saldo mínimo;

[11] Agradeço à Prof. Maria dos Remédios Cravo a sugestão desta definição.

— não permite saldos negativos.

Para criar as classes `conta_ordenado` e `conta_jovem`, podemos pensar em duplicar o código associado à classe `conta_gen`, fazendo as adaptações necessárias nos respetivos métodos. No entanto, esta é uma má prática de programação, pois não só aumenta desnecessariamente a quantidade de código, mas também porque qualquer alteração realizada sobre a classe `conta_gen` não se propaga automaticamente às classes `conta_ordenado` e `conta_jovem`, as quais são contas.

Para evitar estes inconvenientes, em programação com objetos existe o conceito de subclasse. Diz-se que uma classe é uma *subclasse* de outra classe, se a primeira corresponder a uma especialização da segunda. Ou seja, o comportamento da subclasse corresponde ao comportamento da superclasse, exceto no caso em que comportamento específico está indicado para a subclasse. Diz-se que a subclasse *herda* o comportamento da superclasse, exceto quando este é explicitamente alterado na subclasse.

É na definição de subclasses que temos que considerar a parte opcional da ⟨definição de objeto⟩ apresentada na página 323, e reproduzida aqui para facilitar a leitura:

⟨definição de objeto⟩ ::= `class` ⟨nome⟩ {(⟨nome⟩)}: CR
TAB ⟨definição de método⟩$^+$

Ao considerar uma definição de classe da forma `class`⟨nome$_1$⟩ (⟨nome$_2$⟩), estamos a definir a classe ⟨nome$_1$⟩ como uma subclasse de ⟨nome$_2$⟩. Neste caso, a classe ⟨nome$_1$⟩ "herda" automaticamente todas as variáveis e métodos definidos na classe ⟨nome$_2$⟩.

Por exemplo, definindo a classe `conta_ordenado` como

```
class conta_ordenado(conta_gen),
```

se nada for dito em contrário, a classe `conta_ordenado` passa automaticamente a ter os métodos `deposito`, `levantamento` e `consulta` definidos na classe `conta_gen`. Se, associado à classe `conta_ordenado`, for definido um método com o mesmo nome de um método da classe `conta_gen`, este método, na classe `conta_ordenado`, sobrepõe-se ao método homónimo da classe `conta_gen`. Se forem definidos novos métodos na classe `conta_ordenado`, estes pertencem apenas

a essa classe, não existindo na classe `conta_gen`.

Em resumo, as classes podem ter *subclasses* que correspondem a especializações dos seus elementos. As subclasses *herdam* as variáveis de estado e os métodos das superclasses, salvo indicação em contrário.

A classe `conta_ordenado` pode ser definida do seguinte modo:

```
class conta_ordenado(conta_gen):

    def __init__(self, saldo_inicial, ordenado):
        if saldo_inicial >= ordenado:
            self.saldo = saldo_inicial
            self.saldo_min = -ordenado
        else:
            print('O saldo deve ser maior que o ordenado')
```

Nesta classe, definimos o método `__init__`, o qual recebe o saldo inicial e o valor do ordenado. O ordenado define que o saldo mínimo corresponde a `-ordenado`. Note-se que associado à classe `conta_ordenado` aparece o método `__init__`, que não existia na classe `conta_gen`. Isto significa que não estamos interessados em criar instâncias da classe `conta_gen`, mas que iremos criar instâncias da classe `conta_ordenado`. Na realidade, a classe `conta_gen` apenas serve para definir o comportamento genérico de uma conta bancária.

Analogamente, a classe `conta_jovem` é definida como uma subclasse de `conta_gen`, definindo apenas o método `__init__`, no qual o valor do saldo mínimo é zero:

```
class conta_jovem(conta_gen):

    def __init__(self, saldo_inicial):
        self.saldo = saldo_inicial
        self.saldo_min = 0
```

Estas classes permitem-nos efetuar a seguinte interação:

```
>>> conta_ord_01 = conta_ordenado(300, 500)
O saldo deve ser maior que o ordenado
>>> conta_ord_01 = conta_ordenado(800, 500)
```

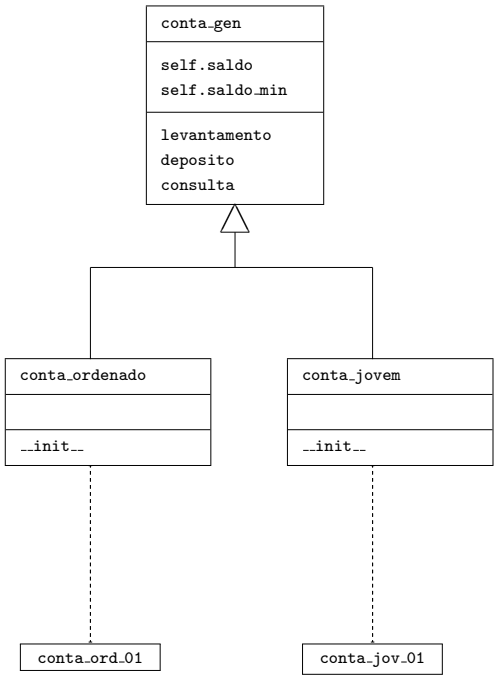

Figura 12.5: Hierarquia de contas e respetivas instâncias.

```
>>> conta_ord_01.levantamento(500)
300
>>> conta_ord_01.levantamento(500)
-200
>>> conta_jov_01 = conta_jovem(50)
>>> conta_jov_01.consulta()
50
>>> conta_jov_01.levantamento(100)
Saldo insuficiente
```

As classes correspondentes aos diferentes tipos de contas estão relacionadas entre si, podendo ser organizadas numa *hierarquia*. Nesta hierarquia, ao nível mais alto está a classe `conta_gen`, com duas subclasses, `conta_ordenado` e `conta_jovem`. Na Figura 12.5 mostramos esta hierarquia, bem como as instâncias `conta_ord_01` e `conta_jov_01`.

A importância da existência de várias classes e instâncias organizadas numa hierarquia, que estabelece relações entre classes e suas subclasses e instâncias, provém de uma forma de propagação associada, designada por *herança*. A herança consiste na transmissão das caraterísticas duma classe às suas subclasses e instâncias. Isto significa, por exemplo, que todos os métodos associados à classe `conta_gen` são herdados por todas as subclasses e instâncias dessa classe, exceto se forem explicitamente alterados numa das subclasses.

Suponhamos agora que desejamos criar uma conta jovem que está protegida por um código de segurança, normalmente conhecido por PIN[12]. Esta nova conta, sendo uma `conta_jovem`, irá herdar as caraterísticas desta classe. A nova classe que vamos definir, `conta_jovem_com_pin` terá um estado interno constituído por três variáveis, uma delas corresponde ao código de acesso, `pin`, a outra será uma `conta_jovem`, a qual só é acedida após o fornecimento do código de acesso correto, finalmente, a terceira variável, `contador`, regista o número de tentativas falhadas de acesso à conta, bloqueando o acesso à conta assim que este número atingir o valor 3. Deste modo, o método que cria contas da classe `conta_jovem_com_pin` é definido por:

```
def __init__(self, quantia, codigo):
    self.conta = conta_jovem(quantia)
    self.pin = codigo
    self.contador = 0
```

Para aceder a uma conta do tipo `conta_jovem_com_pin` utilizamos o método `acede`, o qual, sempre que é fornecido um PIN errado aumenta em uma unidade o valor do contador de tentativas de acesso incorretas. Ao ser efetuado um acesso com o PIN correto, o valor do contador de tentativas incorretas volta a ser considerado zero. O método `acede` devolve `True` se é efetuado um acesso com o PIN correto e devolve `False` em caso contrário.

```
def acede(self, pin):
    if self.contador == 3:
        print('Conta bloqueada')
        return False
    elif pin == self.pin:
        self.contador = 0
```

[12]Do inglês, "Personal Identification Number".

```
            return True
        else:
            self.contador = self.contador + 1
            if self.contador == 3:
                print('Conta bloqueada')
            else:
                print('PIN incorreto')
                print('Tem mais', 3 - self.contador, 'tentativas')
            return False
```

Os métodos `levantamento`, `deposito` e `consulta` utilizam o método `acede` para verificar a correção do PIN e, em caso de este acesso estar correto, utilizam o método correspondente da conta `conta_jovem` para aceder à conta `self.conta`. Note-se que `self.conta` é a instância da classe `conta_jovem` que está armazenada na instância da classe `conta_jovem_com_pin` que está a ser manipulada.

Finalmente, a classe `conta_jovem_com_pin` apresenta um método que não existe na classe `conta_jovem` e que corresponde à ação de alteração do PIN da conta.

A classe `conta_jovem_com_pin` é definida do seguinte modo:

```
class conta_jovem_com_pin (conta_jovem):

    def __init__(self, quantia, codigo):
        self.conta = conta_jovem (quantia)
        self.pin = codigo
        self.contador = 0


    def levantamento(self, quantia, pin):
        if self.acede(pin):
            return self.conta.levantamento(quantia)


    def deposito(self, quantia, pin):
        if self.acede(pin):
            return self.conta.deposito(quantia)


    def altera_codigo(self, pin):
```

```
        if self.acede(pin):
            novopin = input('Introduza o novo PIN\n--> ')
            print('Para verificação')
            verifica = input('Volte a introduzir o novo PIN\n--> ')
            if novopin == verifica:
                self.pin = novopin
                print('PIN alterado')
            else:
                print('Operação sem sucesso')


    def consulta(self, pin):
        if self.acede(pin):
            return self.conta.consulta()


    def acede(self, pin):
        if self.contador == 3:
            print('Conta bloqueada')
            return False
        elif pin == self.pin:
            self.contador = 0
            return True
        else:
            self.contador = self.contador + 1
            if self.contador == 3:
                print('Conta bloqueada')
            else:
                print('PIN incorreto')
                print('Tem mais', 3 - self.contador, 'tentativas')
            return False
```

Com esta classe, podemos originar a seguinte interação:

```
>>> conta_jcp_01 = conta_jovem_com_pin(120, '1234')
>>> conta_jcp_01.levantamento(30, 'abcd')
PIN incorreto
Tem mais 2 tentativas
```

```
>>> conta_jcp_01.levantamento(30, 'ABCD')
PIN incorreto
Tem mais 1 tentativas
>>> conta_jcp_01.levantamento(30, '1234')
90
>>> conta_jcp_01.altera_codigo('abcd')
PIN incorreto
Tem mais 2 tentativas
>>> conta_jcp_01.altera_codigo('1234')
Introduza o novo PIN
--> abcd
Para verificação
Volte a introduzir o novo PIN
--> abcd
PIN alterado
>>> conta_jcp_01.levantamento(30, 'abcd')
60
>>> conta_jcp_01.levantamento(30, '1234')
PIN incorreto
Tem mais 2 tentativas
>>> conta_jcp_01.levantamento(30, '1234')
PIN incorreto
Tem mais 1 tentativas
>>> conta_jcp_01.levantamento(30, '1234')
Conta bloqueada
```

12.4 Número abitrário de argumentos

A abordagem que seguimos na seção anterior para definir a classe `conta_jovem_com_pin` não foi a mais elegante: redefinimos os métodos associados à classe `conta_jovem`; teríamos que redefinir estes mesmos métodos para criar qualquer outro tipo de conta com PIN; não capturámos a abstração de entidade com PIN.

O modo mais correto de criar uma conta com PIN seria o de começar por definir o comportamento de uma entidade com PIN, dentro da qual existiria uma conta de qualquer tipo. Uma entidade com PIN é caraterizada por variáveis de estado

correspondentes ao PIN, a um contador de acessos incorretos e a um objeto protegido pelo PIN.

Podemos definir a classe `obj_com_pin` do seguinte modo:

```
class obj_com_pin:

    def __init__(self, pin, obj):
        self.pin = pin
        self.contador = 0
        self.obj = obj

    def verifica_PIN(self, pin):
        if self.contador == 3:
            raise ValueError('Conta bloqueada')
        elif pin == self.pin:
            self.contador = 0
            return True
        else:
            self.contador = self.contador + 1
            if self.contador == 3:
                raise ValueError('Conta bloqueada')
            else:
                print('PIN incorreto')
                mensagem = 'Tem mais ' + str(3-self.contador) +\
                    ' tentativas'
                raise ValueError(mensagem)

    def acede(self, pin):
        if self.verifica_PIN(pin):
            return self.obj

    def altera_PIN(self, pin):

        if self.verifica_PIN(pin):
            novopin = input('Introduza o novo PIN\n--> ')
            print('Para verificação')
```

```
            verifica = input('Volte a introduzir o novo PIN\n--> ')
            if novopin == verifica:
                self.pin = novopin
                print('PIN alterado')
            else:
                print('Operação sem sucesso')
```

Com esta classe podemos gerar a interação:

```
>>> conta = obj_com_pin('ABC', conta_jovem(100))
>>> conta.acede('ABC').consulta()
100
>>> conta.acede('ABC').levantamento(50)
50
>>> conta.acede('XPTO').consulta()
PIN incorreto
Tem mais 2 tentativas
>>> conta.altera_PIN('ABC')
Introduza o novo PIN
--> XPTO
Para verificação
Volte a introduzir o novo PIN
--> XPTO
PIN alterado
>>> conta.acede('XPTO').consulta()
50
```

Uma outra alternativa para definir a classe `obj_com_pin` corresponde, não em criar o objeto protegido pelo PIN como um argumento do método que cria a instância, mas sim criá-lo durante a inicialização das variáveis de estado da instância. Para isso, teremos que fornecer ao método `__init__` o nome da classe correspondente ao objeto a criar e os argumentos necessários para a criação desse objeto. Esta abordagem levanta-nos o problema que instâncias de diferentes classes podem ter que ser criadas com diferente número de argumentos. Veja-se, por exemplo, os argumentos necessários para criar uma `conta_ordenado` (página 339) e os argumentos necessários para criar uma `conta_jovem` (página 339). Vamos precisar da possibilidade de criar funções que aceitam um número

arbitrário de argumentos.

A utilização de funções com um número arbitrário de argumentos já tem sido feita nos nossos programas. Por exemplo, a função embutida `print` aceita qualquer número de argumentos. A execução de `print()` origina uma linha em branco, e a execução de `print` com qualquer número de argumentos origina a escrita dos valores dos seus argumentos seguidos de um salto de linha. Até agora, as funções que escrevemos têm um número fixo de argumentos, o que contrasta com algumas das funções embutidas do Python, por exemplo, a função `print`.

Vamos considerar um novo aspeto na definição de funções correspondente à utilização de um número arbitrário de argumentos. Para podermos utilizar este aspeto, teremos que rever a definição de função apresentada na página 76. Uma forma mais completa da definição de funções em Python é dada pelas seguintes expressões em notação BNF:

⟨definição de função⟩ ::= `def` ⟨nome⟩ `(`⟨parâmetros formais⟩`): CR`
`TAB` ⟨corpo⟩

⟨parâmetros formais⟩ ::= ⟨nada⟩{`*`⟨nome⟩} | ⟨nomes⟩{`, *`⟨nome⟩}

A diferença desta definição em relação à definição apresentada na página 76, corresponde à possibilidade dos parâmetros formais terminarem com um nome que é antecedido por `*`. Por exemplo, esta definição autoriza-nos a criar uma função em Python cuja primeira linha corresponde a "`def ex_fn(a, b, *c):`".

Ao encontrar uma chamada a uma função cujo último parâmetro formal seja antecedido por um asterisco, o Python associa os $n-1$ primeiros parâmetros formais com os $n-1$ primeiros parâmetros concretos e associa o último parâmetro formal (o nome que é antecedido por um asterisco) com o tuplo constituído pelos restantes parâmetros concretos. A partir daqui, a execução da função segue os mesmos passos que anteriormente.

Consideremos a seguinte definição de função:

```
def ex_fn(a, b, *c):
    print('a =', a)
    print('b =', b)
    print('c =', c)
```

Esta função permite gerar a seguinte interação, a qual revela este novo aspeto da definição de funções:

```
>>> ex_fn(3, 4)
a = 3
b = 4
c = ()
>>> ex_fn(5, 7, 9, 1, 2, 5)
a = 5
b = 7
c = (9, 1, 2, 5)
>>> ex_fn(2)
TypeError: ex_fn() takes at least 2 arguments (1 given)
```

A função `ex_fn` aceita, no mínimo, dois argumentos. Os dois primeiros parâmetros concretos são associados com os parâmetros formais `a` e `b` e tuplo contendo os restantes parâmetros concretos é associado ao parâmetro formal `c`.

Podemos agora escrever a seguinte versão da classe `obj_com_pin`[13], na qual omitimos propositadamente a possibilidade de alteração do PIN:

```
class obj_com_pin:

    def __init__(self, pin, cria_obj, *args):
        self.pin = pin
        self.contador = 0
        self.obj = cria_obj(*args)

    def acede(self, pin):
        if self.contador == 3:
            raise ValueError('Conta bloqueada')
        elif pin == self.pin:
            self.contador = 0
            return self.obj
        else:
            self.contador = self.contador + 1
            if self.contador == 3:
```

[13] Agradeço à Prof. Maria dos Remédios Cravo a sugestão desta abordagem.

```
            raise ValueError('Conta bloqueada')
        else:
            mensagem = 'PIN incorreto\n' + 'Tem mais ' + \
                str(3-self.contador) + ' tentativas'
            raise ValueError(mensagem)
```

Podemos agora gerar a interação:

```
>>> co = obj_com_pin('ABC', conta_ordenado, 5000, 1000)
>>> co.acede('ABC').deposito(500)
5500
>>> cj = obj_com_pin('XPTO', conta_jovem, 100)
>>> cj.acede('XPTO').levantamento(50)
50
```

12.5 Objetos em Python

O Python é uma linguagem baseada em objetos. Embora a nossa utilização do Python tenha fundamentalmente correspondido à utilização da sua faceta imperativa, na realidade, todos os tipos de dados embutidos em Python correspondem a classes. Recorrendo à função `type`, introduzida na página 327, podemos obter a seguinte interação que revela que os tipos embutidos com que temos trabalhado não são mais do que objetos pertencentes a determinadas classes:

```
>>> type(2)
<class 'int'>
>>> type(True)
<class 'bool'>
>>> type(3.5)
<class 'float'>
>>> type('abc')
<class 'str'>
>>> type([1, 2, 3])
<class 'list'>
```

Recordemos a função que lê uma linha de um ficheiro, ⟨fich⟩`.readline()`, apresentada na Seção 11.2. Dissemos na altura que o nome desta função podia ser tratado como um nome composto. O que na realidade se passa é que ao abrirmos um ficheiro, estamos a criar um objeto, objeto esse em que um dos métodos tem o nome de `readline`.

12.6 Polimorfismo

Já vimos que muitas das operações do Python são operações sobrecarregadas, ou seja, aplicam-se a vários tipos de dados. Por exemplo, a operação `+` pode ser aplicada a inteiros, a reais, a tuplos, a cadeias de caracteres, a listas e a dicionários.

Diz-se que uma operação é *polimórfica*, ou que apresenta a propriedade do *polimorfismo*[14], quando é possível definir funções diferentes que usam a mesma operação para lidar com tipos de dados diferentes. Consideremos a operação de adição, representada em Python por "`+`". Esta operação aplica-se tanto a inteiros como a reais. Recorde-se da página 37, que dissemos que internamente ao Python existiam duas operações, que representámos por $+_{\mathbb{Z}}$ e $+_{\mathbb{R}}$, que, respetivamente, são invocadas quando os argumentos da operação `+` são números inteiros ou são números reais. Esta capacidade do Python de associar a mesma representação externa de uma operação, `+`, a diferentes operações internas corresponde a uma faceta do polimorfismo[15]. Esta faceta do polimorfismo dá origem a operações sobrecarregadas.

Após definirmos o tipo correspondente a números complexos, no Capítulo 9, estaremos certamente interessados em escrever funções que executem operações aritméticas sobre complexos. Por exemplo, a soma de complexos pode ser definida por:

```
def soma_compl(c1, c2):
    r = c1.p_real() + c2.p_real()
    i = c1.p_imag() + c2.p_imag()
    return compl(r, i)
```

[14] A palavra "polimorfismo" é derivada do grego e significa apresentar múltiplas formas.

[15] O tipo de polimorfismo que definimos nesta seção corresponde ao conceito inglês de "subtype polymorphism".

com a qual podemos gerar a interação:

```
>>> c1 = compl(9, 6)
>>> c2 = compl(7, 6)
>>> c3 = soma_compl(c1, c2)
>>> c3.escreve()
16 + 12 i
```

No entanto, quando trabalhamos com números complexos em matemática usamos o símbolo da operação de adição, +, para representar a soma de números complexos. Através da propriedade do polimorfismo, o Python permite especificar que a operação `+` também pode ser aplicada a números complexos e instruir o computador em como aplicar a operação "`+`" a números complexos.

Sabemos que em Python, todos os tipos embutidos correspondem a classes. Um dos métodos que existe numa classe tem o nome de `__add__`, recebendo dois argumentos, `self` e outro elemento do mesmo tipo. A sua utilização é semelhante à da função `compl_iguais` que apresentámos na página 325, como o mostra a seguinte interação:

```
>>> a = 5
>>> a.__add__(2)
7
```

A invocação deste método pode ser feita através da representação externa "`+`", ou seja, sempre que o Python encontra a operação `+`, invoca automaticamente o método `__add__`, aplicado às instâncias envolvidas. Assim, se associado à classe `compl`, definirmos o método

```
def __add__(self, outro):
    r = self.p_real() + outro.p_real()
    i = self.p_imag() + outro.p_imag()
    return compl(r, i)
```

podemos originar a interação

```
>>> c1 = compl(2, 4)
>>> c2 = compl(5, 10)
```

```
>>> c3 = c1 + c2
>>> c3.escreve()
7 + 14 i
```

Ou seja, usando o polimorfismo, criámos uma nova forma da operação `+` que sabe somar complexos. A "interface" da operação para somar complexos passa a ser o operador `+`, sendo este operador transformado na operação de somar complexos quando os seus argumentos são números complexos.

De um modo análogo, existem métodos embutidos, com os nomes `__sub__`, `__mul__` e `__truediv__` que estão associados às representações das operações `-`, `*` e `/`. O método `__eq__` está associado à operação `==`. Existe também um método, `__repr__` que transforma a representação interna de uma instância da classe numa cadeia de caracteres que corresponde à sua representação externa. Esta representação externa é usada diretamente pela função `print`.

Sabendo como definir em Python operações aritméticas e a representação externa, podemos modificar a classe `compl` com os seguintes métodos. Repare-se que com o método `__eq__` deixamos de precisar do método `compl_iguais` e que com o método `__repr__` deixamos de precisar do método `escreve` para gerar a representação externa de complexos[16]. Em relação à classe `compl`, definimos como operações de alto nível, funções que adicionam, subtraem, multiplicam e dividem complexos.

```
class compl:

    def __init__(self, real, imag):
        if isinstance(real, (int, float)) and \
           isinstance(imag, (int, float)):
            self.r = real
            self.i = imag
        else:
            raise ValueError ('complexo: argumento errado')


    def p_real(self):
        return self.r
```

[16]No método `__repr__` utilizamos a função embutida `str`, a qual foi apresentada na Tabela 4.3, que recebe uma constante de qualquer tipo e tem como valor a cadeia de caracteres correspondente a essa constante.

```
    def p_imag(self):
        return self.i


    def compl_zero(self):
        return self.r == 0 and self.i == 0


    def __eq__(self, outro):
        return self.r == outro.p_real() and \
               self.i == outro.p_imag()


    def __add__(self, outro):
        r = self.p_real() + outro.p_real()
        i = self.p_imag() + outro.p_imag()
        return compl(r, i)


    def __sub__(self, outro):
        r = self.p_real() - outro.p_real()
        i = self.p_imag() - outro.p_imag()
        return compl(r, i)


    def __mul__(self, outro):
        r = self.p_real() * outro.p_real() - \
            self.p_imag() * outro.p_imag()
        i = self.p_real() * outro.p_imag() + \
            self.p_imag() * outro.p_real()
        return compl(r, i)



    def __truediv__(self, outro):
        try:
            den = outro.p_real() * outro.p_real() + \
                outro.p_imag() * outro.p_imag()
            r = (self.p_real() * outro.p_real() + \
```

```
                self.p_imag() * outro.p_imag()) / den
            i = (self.p_imag() * outro.p_real() - \
                self.p_real() * outro.p_imag()) / den
            return compl(r, i)
        except ZeroDivisionError:
            print('complexo: divisão por zero')



    def __repr__(self):
        if self.p_imag() >= 0:
            return str(self.p_real()) + '+' + \
                   str(self.p_imag()) + 'i'
        else:
            return str(self.p_real()) + '-' + \
                   str(abs(self.p_imag())) + 'i'
```

Podendo agora gerar a seguinte interação:

```
>>> c1 = compl(2, 5)
>>> c1
2+5i
>>> c2 = compl(-9, -7)
>>> c2
-9-7i
>>> c1 + c2
-7-2i
>>> c1 * c2
17-59i
>>> c3 = compl(0, 0)
>>> c1 / c3
complexo: divisão por zero
>>> c1 == c2
False
>>> c4 = compl(2, 5)
>>> c1 == c4
True
```

12.7 Notas finais

Neste capítulo apresentámos um estilo de programação, conhecido como programação com objetos, que é centrado em objetos, entidades que possuem um estado interno e reagem a mensagens. Os objetos correspondem a entidades, tais como contas bancárias, livros, estudantes, etc. O conceito de objeto agrupa as funções que manipulam dados (os métodos) com os dados que representam o estado do objeto.

O conceito de objeto foi introduzido com a linguagem SIMULA[17], foi vulgarizado em 1984 pelo sistema operativo do Macintosh, foi adotado pelo sistema operativo Windows, uma década mais tarde, e está hoje na base de várias linguagens de programação, por exemplo, o C++, o CLOS, o Java e o Python.

12.8 Exercícios

1. Defina uma classe em Python, chamada `estacionamento`, que simula o funcionamento de um parque de estacionamento. A classe `estacionamento` recebe um inteiro que determina a lotação do parque e devolve um objeto com os seguintes métodos: `entra()`, corresponde à entrada de um carro; `sai()`, corresponde à saída de um carro; `lugares()` indica o número de lugares livres no estacionamento. Por exemplo,

```
>>> ist = estacionamento(20)
>>> ist.lugares()
20
>>> ist.entra()
>>> ist.entra()
>>> ist.entra()
>>> ist.entra()
>>> ist.sai()
>>> ist.lugares()
17
```

2. Defina uma classe que corresponde a uma urna de uma votação. A sua classe deve receber a lista dos possíveis candidatos e manter como estado

[17] [Dahl and Nygaard, 1967].

interno o número de votos em cada candidato. Esta classe pode receber um voto num dos possíveis candidatos, aumentando o número de votos nesse candidato em um. Deve também permitir apresentar os resultados da votação.

3. Suponha que desejava criar o tipo *conjunto*. Considere as seguintes operações para conjuntos:

   Construtores:

   - $novo\_conj : \{\} \mapsto conjunto$
     $novo\_conj()$ tem como valor um conjunto sem elementos.
   - $insere\_conj : elemento \times conjunto \mapsto conjunto$
     $insere\_conj(e, c)$ tem como valor o resultado de inserir o elemento e no conjunto c; se e já pertencer a c, tem como valor c.

   Seletores:

   - $retira\_conj : elemento \times conjunto \mapsto conjunto$
     $retira\_conj(e, c)$ tem como valor o resultado de retirar do conjunto c o elemento e; se e não pertencer a c, tem como valor c.

   Reconhecedores:

   - $e\_conjunto : universal \mapsto lógico$
     $e\_conjunto(arg)$ tem o valor $verdadeiro$ se arg é um conjunto, e tem o valor $falso$, em caso contrário.
   - $e\_conj\_vazio : conjunto \mapsto lógico$
     $e\_conj\_vazio(c)$ tem o valor $verdadeiro$ se o conjunto c é o conjunto vazio, e tem o valor $falso$, em caso contrário.

   Testes:

   - $conjuntos\_iguais : conjunto \times conjunto \mapsto lógico$
     $conjuntos\_iguais(c_1, c_2)$ tem o valor $verdadeiro$ se os conjuntos c_1 e c_2 forem iguais, ou seja, se tiverem os mesmos elementos, e tem o valor $falso$, em caso contrário.
   - $pertence : elemento \times conjunto \mapsto lógico$
     $pertence(e, c)$ tem o valor $verdadeiro$ se o elemento e pertence ao conjunto c e tem o valor $falso$, em caso contrário.

Operações adicionais:

- $cardinal : conjunto \mapsto inteiro$
  $cardinal(c)$ tem como valor o número de elementos do conjunto c.
- $subconjunto : conjunto \times conjunto \mapsto lógico$
  $subconjunto(c_1, c_2)$ tem o valor $verdadeiro$, se o conjunto c_1 for um subconjunto do conjunto c_2, ou seja, se todos os elementos de c_1 pertencerem a c_2, e tem o valor $falso$, em caso contrário.
- $uniao : conjunto \times conjunto \mapsto conjunto$
  $uniao(c_1, c_2)$ tem como valor o conjunto união de c_1 com c_2, ou seja, o conjunto formado por todos os elementos que pertencem a c_1 ou a c_2.
- $interseccao : conjunto \times conjunto \mapsto conjunto$
  $interseccao(c_1, c_2)$ tem como valor o conjunto intersecção de c_1 com c_2, ou seja, o conjunto formado por todos os elementos que pertencem simultaneamente a c_1 e a c_2.
- $diferenca : conjunto \times conjunto \mapsto conjunto$
  $diferenca(c_1, c_2)$ tem como valor o conjunto diferença de c_1 e c_2, ou seja, o conjunto formado por todos os elementos que pertencem a c_1 e não pertencem a c_2.

(a) Escreva uma axiomatização para as operações do tipo conjunto.

(b) Defina uma representação para conjuntos utilizando listas.

(c) Defina a classe `conjunto` com base na representação escolhida.

4. Considere a função de Ackermann apresentada nos exercícios do Capítulo 7. Como pode verificar, a sua função calcula várias vezes o mesmo valor. Para evitar este problema, podemos definir uma classe, `mem_A`, cujo estado interno contém informação sobre os valores de `A` já calculados, apenas calculando um novo valor quando este ainda não é conhecido. Esta classe possui um método `val` que calcula o valor de `A` para os inteiros que são seus argumentos. Por exemplo,

```
>>> A = mem_A()
>>> A.val(2, 2)
7
```

Defina a classe `mem_A`.