Capítulo 6

Funções revisitadas

'When I use a word,' Humpty Dumpty said, in rather a scornful tone, 'it means just what I choose it to mean neither more nor less.'
'The question is,' said Alice, 'whether you can make words mean so many different things.'
'The question is,' said Humpty Dumpty, 'which is to be the master — that's all.'

Lewis Carroll, *Through the Looking Glass*

Neste capítulo voltamos a considerar as funções em Python, introduzidas no Capítulo 3. Apesar dos programas que temos desenvolvido utilizarem funções, existem muitos aspetos associados à sua utilização que ainda não foram por nós considerados. Começamos por introduzir uma estrutura nas funções, permitindo que estas contenham outras funções, e discutimos as vantagens desta estruturação. Apresentamos o conceito de função recursiva e de programação funcional. Finalmente apresentamos funções que recebem funções como argumentos e funções que devolvem funções.

6.1 Estruturação de funções

Para motivar a necessidade de introduzir uma estrutura nas funções, consideremos novamente a função `potencia` apresentada na Secção 3.4.2:

```
def potencia(x , n):
    res = 1
    while n != 0:
        res = res * x
        n = n - 1
    return res
```

Esta função apenas produz resultados para valores do expoente que sejam inteiros positivos ou nulos. Se fornecermos à função um valor negativo para o expoente, o Python gera um ciclo infinito, pois a condição do ciclo `while`, `n != 0`, nunca é satisfeita. Isto corresponde a uma situação que queremos obviamente evitar. Tendo em atenção que $x^{-n} = 1/x^n$, podemos pensar na seguinte solução para a função `potencia` que lida tanto com o expoente positivo como com o expoente negativo:

```
def potencia(x, n) :
    res = 1
    if n >= 0:
        while n != 0:
            res = res * x
            n = n - 1
    else:
        while n != 0:
            res = res / x
            n = n + 1
    return res
```

Com esta nova função podemos gerar a interação:

```
>>> potencia(3, 2)
9
>>> potencia(3, -2)
0.1111111111111111
>>> potencia(3, 0)
1
```

A desvantagem desta solução é a de exigir dois ciclos muito semelhantes, um que trata o caso do expoente positivo, multiplicando sucessivamente o resultado

por `x`, e outro que trata o caso do expoente negativo, dividindo sucessivamente o resultado por `x`.

Podemos pensar numa outra solução que recorre a uma função auxiliar (com o nome `potencia_aux`) para o cálculo do valor da potência, função essa que é sempre chamada com um expoente positivo:

```
def potencia(x, n):
    if n >= 0:
        return potencia_aux(x, n)
    else:
        return 1 / potencia_aux(x, -n)

def potencia_aux(b, e):
    res = 1
    while e != 0:
        res = res * b
        e = e - 1
    return res
```

Com esta solução resolvemos o problema da existência dos dois ciclos semelhantes, mas potencialmente criámos um outro problema: nada impede que a função `potencia_aux` seja diretamente utilizada para calcular a potência, eventualmente gerando um ciclo infinito se os seus argumentos não forem os corretos.

O Python apresenta uma solução para abordar problemas deste tipo, baseada no conceito de *estrutura de blocos*. A estrutura de blocos é muito importante em programação, existindo uma classe de linguagens de programação, chamadas *linguagens estruturadas em blocos*, que são baseadas na definição de blocos. Historicamente, a primeira linguagem desta classe foi o ALGOL, desenvolvida na década de 50, e muitas outras têm surgido, o Python é uma destas linguagens. A ideia subjacente à estrutura de blocos consiste em definir funções dentro das quais existem outras funções. Em Python, qualquer função pode ser considerada como um bloco, dentro da qual podem ser definidos outros blocos.

Nas linguagens estruturadas em blocos, toda a informação definida dentro de um bloco pertence a esse bloco, e só pode ser usada por esse bloco e pelos blocos definidos dentro dele. Esta regra permite a proteção efetiva dos dados definidos em cada bloco da utilização não autorizada por parte de outros blocos. Podemos

pensar nos blocos como sendo egoístas, não permitindo o acesso aos seus dados pelos blocos definidos fora deles.

A razão por que os blocos não podem usar a informação definida num bloco interior é que essa informação pode não estar pronta para ser utilizada até que algumas ações lhe sejam aplicadas ou pode exigir que certas condições sejam verificadas. No caso da função `potencia_aux`, estas condições correspondem ao facto de o expoente ter que ser positivo e a estrutura de blocos vai permitir especificar que a única função que pode usar a função `potencia_aux` é a função `potencia`, a qual garante que função `potencia_aux` é usada com os argumentos corretos. O bloco onde essa informação é definida é o único que sabe quais as condições de utilização dessa informação e, consequentemente, o único que pode garantir que essa sequência de ações é aplicada antes da informação ser usada.

Recorde-se da página 76 que uma função em Python foi definida do seguinte modo:

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

⟨corpo⟩ ::= ⟨definição de função⟩* ⟨instruções em função⟩

Até agora, o corpo de todas as nossas funções tem sido apenas constituído por instruções e o componente opcional ⟨definição de função⟩ não tem existido. No entanto, são as definições de funções dentro do ⟨corpo⟩ que permitem a definição da estrutura de blocos num programa em Python: o corpo de cada função corresponde a um bloco.

Recorrendo a esta definição, podemos escrever uma nova versão da função `potencia`:

```
def potencia(x , n):

    def potencia_aux(b, e):
        res = 1
        while e != 0:
            res = res * b
            e = e - 1
        return res
```

```
    if n >= 0:
        return potencia_aux(x, n)
    else:
        return 1 / potencia_aux(x, -n)
```

O corpo desta função contém a definição de uma função, `potencia_aux`, e uma instrução `if`. Com esta função, podemos gerar a seguinte interação:

```
>>> potencia(3, 2)
9
>>> potencia(3, -2)
0.1111111111111111
>>> potencia
<function potencia at 0x10f45d0>
>>> potencia_aux
NameError: name 'potencia_aux' is not defined
```

Esta interação mostra que a função `potencia` é conhecida pelo Python e que a função `potencia_aux` não é conhecida pelo Python, mesmo depois de este a ter utilizado na execução de `potencia(3, 2)`!

Na Figura 6.1 apresentamos de um modo informal a estrutura de blocos que definimos para a nova versão da função `potencia`. Nesta figura, um bloco é indicado como um retângulo, dentro do qual aparece a informação do bloco. Note-se que dentro do bloco da função `potencia` existe um bloco que corresponde à função `potencia_aux`. Nesta ótica, o comportamento que obtivemos na interação anterior está de acordo com a regra associada à estrutura de blocos. De acordo com esta regra, toda a informação definida dentro de um bloco pertence a esse bloco, e só pode ser usada por esse bloco e pelos blocos definidos dentro dele. Isto faz com que a função `potencia_aux` só possa ser utilizada pela função `potencia`, o que efetivamente evita a possibilidade de utilizarmos diretamente a função `potencia_aux`, como a interação anterior o demonstra.

Para compreender o funcionamento da função `potencia`, iremos seguir o funcionamento do Python durante a interação anterior. Ao definir a função `potencia` cria-se, no ambiente global, a associação entre o nome `potencia` e uma entidade computacional correspondente a uma função com certos parâmetros e um corpo (Figura 6.2).

```
def potencia(x, n):
    def potencia_aux(b, e):
        res = 1
            while e != 0:
                res = res * b
                e = e - 1
        return res
    if n >= 0:
        return potencia_aux(x, n)
    else:
        return 1 / potencia_aux(x, -n)
```

Figura 6.1: Estrutura de blocos da função `potencia`.

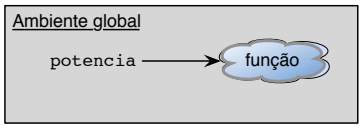


Figura 6.2: Ambiente global com o nome `potencia`.

Até agora tudo corresponde ao que sabemos, o nome `potencia` está associado a uma função, cujo corpo é constituído por uma definição de uma função e por uma instrução `if`. Neste âmbito, o Python "sabe" que `potencia` é uma função e desconhece o nome `potencia_aux`.

Ao avaliar a expressão `potencia(3, 2)`, o Python associa os parâmetros formais da função `potencia` (`x` e `n`) aos parâmetros concretos (`3` e `2`), cria um ambiente local e executa o corpo da função. Durante a execução do corpo da função, o Python encontra a definição da função `potencia_aux`, criando um nome correspondente no ambiente local como se mostra na Figura 6.3. A segunda instrução do corpo de `potencia` corresponde a uma instrução `if`, na qual ambas as alternativas levam à execução da função `potencia_aux`, a qual existe no Ambiente local a `potencia`. A execução desta função leva à criação de um novo ambiente (indicado na Figura 6.4 com o nome Ambiente local a `potencia_aux`), no qual esta função é executada. Depois da execução de `potencia` terminar, os dois ambientes locais desaparecem, voltando-se à situação apresentada na Figura 6.2, e o Python deixa de conhecer o significado de `potencia_aux`, o que justifica a interação apresentada.

Uma questão pertinente a levantar neste momento é a de quando definir funções dentro de outras funções e quando definir funções cujos nomes são colocados no ambiente global. Para responder a esta pergunta devemos considerar que a ativi-

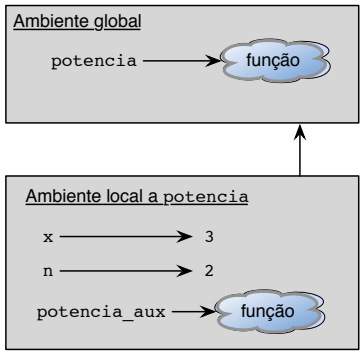


Figura 6.3: Ambiente criado pela execução de `potencia`.

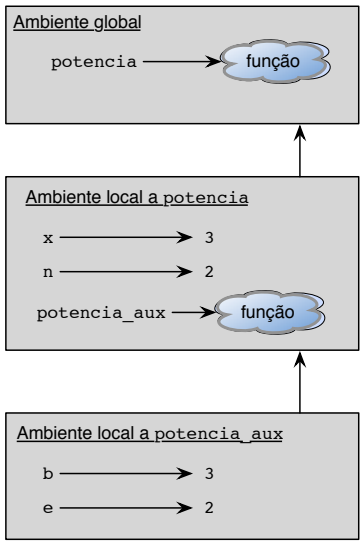


Figura 6.4: Ambientes durante a execução de `potencia`.

dade de programação corresponde à construção de componentes computacionais – por exemplo, funções – que podem ser utilizados como caixas pretas por outros componentes. Sob esta perspetiva, durante a atividade de programação a nossa linguagem de programação está a ser sucessivamente enriquecida no sentido em que vão surgindo operações mais potentes, que podem ser usadas como componentes para desenvolver operações adicionais. Ao escrever um programa devemos decidir se as funções que desenvolvemos devem ser ou não públicas. Por *função pública* entenda-se uma função que existe no ambiente global e, consequentemente, pode ser utilizada por qualquer outra função.

A decisão de definir funções no ambiente global vai prender-se com a utilidade da função e com as restrições impostas à sua utilização. No exemplo do cálculo da potência, é evidente que a operação `potencia` é suficientemente importante para ser disponibilizada publicamente; no entanto a operação auxiliar a que esta recorre, `potencia_aux`, é, como discutimos, privada da função `potencia`, existindo como consequência do algoritmo usado por `potencia`, e deve estar escondida do exterior, evitando utilizações indevidas.

No caso do cálculo da raiz quadrada (apresentado na Secção 3.4.5) deve também ser claro que a função `calcula_raiz` não deve ser tornada pública, no sentido em que não queremos que o palpite inicial seja fornecido ao algoritmo; as funções `bom_palpite` e `novo_palpite` são claramente específicas ao modo de calcular a raiz quadrada, e portanto devem ser privadas. De acordo com esta discussão, a função `raiz` deverá ser definida do seguinte modo (na Figura 6.5 apresentamos a estrutura de blocos da função `raiz`):

```
def raiz(x):

    def calcula_raiz(x, palpite):

        def bom_palpite(x, palpite):
            return abs(x - palpite * palpite) < delta


        def novo_palpite(x, palpite):
            return (palpite + x / palpite) / 2

        while not bom_palpite(x, palpite):
```

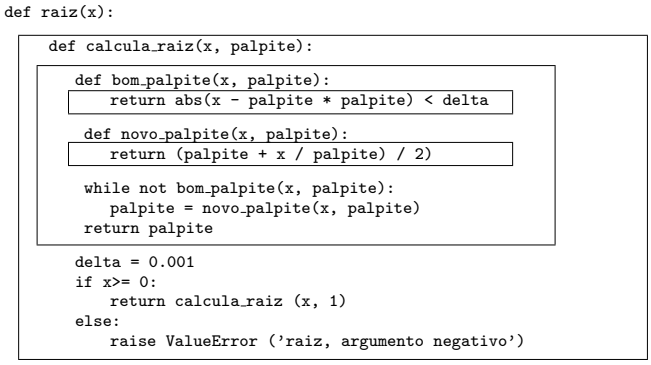


Figura 6.5: Estrutura de blocos da função `raiz`.

```
        palpite = novo_palpite(x, palpite)
    return palpite

delta = 0.001
if x >= 0:
    return calcula_raiz (x, 1)
else:
    raise ValueError ('raiz, argumento negativo')
```

Consideremos agora de um modo mais detalhado a noção de *ambiente*. Sabemos do Capítulo 2 (página 41) que um ambiente contém associações entre nomes e valores. Tipicamente, um ambiente não aparece isolado, estando ligado a um outro ambiente, chamado o *ambiente envolvente* (o qual tem sido indicado nas nossas figuras por uma seta que aponta do ambiente em questão para o ambiente envolvente). Existe um (e só um) ambiente especial, o *ambiente global*, que não tem nenhum ambiente envolvente. Os ambientes estão assim organizados numa estrutura hierárquica, em que cada ambiente está associado ao seu ambiente envolvente.

A sequência de ambientes, desde um dado ambiente ao ambiente global, é utilizada para definir o valor de uma variável num determinado ambiente. O *valor de uma variável* em relação a um ambiente é o valor da associação dessa variável no primeiro ambiente que contém uma associação para essa variável. Se uma variável não tem valor num dado ambiente, ou seja, se não existe nenhuma

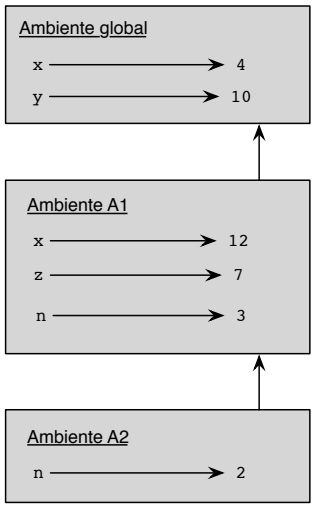


Figura 6.6: Exemplo de ambientes.

ligação para a variável nesse ambiente, a variável diz-se *não ligada* (do inglês "*unbound*"). Esta definição significa que o valor de uma variável *depende* do ambiente em que é considerada.

Consideremos os ambientes apresentados na Figura 6.6. Em relação ao Ambiente A2 a variável `n` tem o valor `2`, a variável `z` tem o valor `7` e a variável `x` tem o valor `12`; em relação ao Ambiente A1 a variável `n` tem o valor `3`, a variável `z` tem o valor `7` e a variável `x` tem o valor `12`; em relação ao Ambiente global as variáveis `n` e `z` não estão ligadas e a variável `x` tem o valor `4`.

Com a introdução da estrutura de blocos, podemos classificar os nomes utilizados por uma função em três categorias, nomes locais, livres e globais. Recordemos, da página 76, a definição do procedimento para calcular o quadrado de um número:

```
def quadrado(x):
    return x * x
```

Sabemos que o nome `x` será associado a um ambiente local durante a avaliação de uma expressão que utilize a função `quadrado`. Esta associação estará ativa durante a avaliação do corpo da função. Numa função, qualquer parâmetro

formal corresponde a um *nome local*, ou seja, apenas tem significado no âmbito do corpo da função.

Nada nos impede de escrever funções que utilizem nomes que não sejam locais, por exemplo, a função

```
def potencia_estranha(n):
    pot = 1
    while n != 0:
        pot = pot * x
        n = n - 1
    return pot
```

contém no seu corpo uma referência a um nome (`x`) que não é um nome local. Com esta função podemos gerar a seguinte interação:

```
>>> potencia_estranha(3)
NameError: global name 'x' is not defined
>>> x = 5
>>> potencia_estranha(3)
125
```

Na interação anterior, ao tentarmos executar a função `potencia_estranha`, o Python gerou um erro, pois não sabia qual o significado do nome `x`. A partir do momento em que dizemos que o nome `x` está associado ao valor 5, através da instrução `x = 5`, podemos executar, sem gerar um erro, a função `potencia_estranha`. No entanto esta tem uma particularidade: calcula sempre a potência para a base cujo valor está associado a `x` (no nosso caso, 5). Um nome, tal como `x` no nosso exemplo, que apareça no corpo de uma função e que não seja um nome local é chamado um *nome não local*.

Entre os nomes não locais, podemos ainda considerar duas categorias: os *nomes globais*, aqueles que pertencem ao ambiente global e que são conhecidos por todas as funções (como é o caso do nome `x` na interação anterior), e os que não são globais, a que se dá o nome de *livres*. Consideremos a seguinte definição da função `potencia_estranha_2` que utiliza um nome livre:

```
def potencia_tambem_estranha(x, n):

    def potencia_estranha_2(n):
        pot = 1
        while n != 0:
            pot = pot * x
            n = n - 1
        return pot

    return potencia_estranha_2(n)
```

A função `potencia_estranha_2` utiliza o nome `x` que não é local, mas, neste caso, `x` também não é global, ele é um nome local (na realidade é um parâmetro formal) da função `potencia_tambem_estranha`.

Durante a execução da função `potencia_tambem_estranha`, o nome `x` está ligado a um objeto computacional, e função `potencia_estranha_2` vai utilizar essa ligação. Na Figura 6.7 apresentamos a estrutura de ambientes criados durante a avaliação de `potencia_tambem_estranha(4, 2)`. Note-se que no Ambiente local a `potencia_t_estranha_2`, a variável `x` não existe. Para calcular o seu valor, o Python vai explorar a sequência de ambientes obtidos seguindo as setas que ligam os ambientes: o Python começa por procurar o valor de `x` no Ambiente local a `potencia_estranha_2`, como este ambiente não contém o nome `x`, o Python procura o seu valor no ambiente Ambiente local a `potencia_tambem_estranha`, encontrando o valor `4`. A utilização de nomes livres é útil quando se desenvolvem funções com muitos parâmetros e que utilizam a estrutura de blocos.

Em resumo, os nomes podem ser locais ou não locais, sendo estes últimos ainda divididos em livres e globais:

$$\text{nome} \begin{cases} \text{local} \\ \text{não local} \begin{cases} \text{global} \\ \text{livre} \end{cases} \end{cases}$$

Define-se *domínio* de um nome como a gama de instruções pelas quais o nome é conhecido, ou seja, o conjunto das instruções onde o nome pode ser utilizado. O domínio dos nomes locais é o corpo da função de que são parâmetros formais ou na qual são definidos; o domínio dos nomes globais é o conjunto de todas as

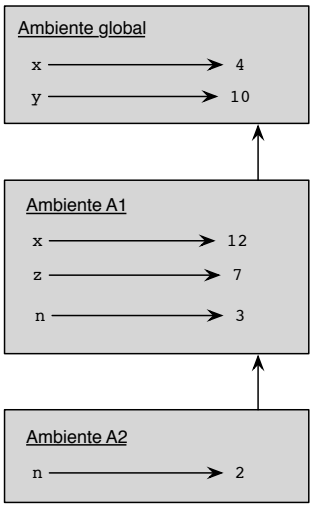


Figura 6.7: Ambientes criados pela avaliação de `potencia_tambem_estranha(4, 2)`.

instruções numa interação em Python.

O tipo de domínio utilizado em Python é chamado *domínio estático*: o domínio de um nome é definido em termos da estrutura do programa (a hierarquia dos seus blocos) e não é influenciado pelo modo como a execução do programa é feita.

O Python permite a utilização de nomes não locais mas não permite a sua alteração. Isto significa que se numa função se executar uma instrução de atribuição a um nome que não é local, o Python cria uma variável local correspondente a esse nome e altera essa variável local, não alterando a variável não local. Este aspeto é ilustrado com a seguinte função:

```
def potencia_ainda_mais_estranha(x, n):

    def potencia_estranha_3(n):
        pot = 1
        x = 7
        while n > 0:
            pot = pot * x
```

```
            n = n - 1
        return pot

    print('x=', x)
    res = potencia_estranha_3(n)
    print('x=', x)
    return res
```

a qual permite gerar a seguinte interação:

```
>>> potencia_ainda_mais_estranha(4, 2)
x= 4
x= 4
49
```

Com efeito, a instrução `x = 7` no corpo da função `potencia_estranha_3` cria o nome `x` como uma variável local à função `potencia_estranha_3`, sendo o seu valor utilizado no cálculo da potência. Fora desta função, a variável `x` mantém o seu valor.

De modo a permitir a partilha de variáveis não locais entre funções, existe em Python a instrução `global`, a qual tem a seguinte sintaxe em notação BNF[1]:

⟨instrução `global`⟩ ::= `global` ⟨nomes⟩

Quando a instrução `global` aparece no corpo de uma função, o Python considera que ⟨nomes⟩ correspondem a variáveis partilhadas e permite a sua alteração como variáveis não locais. A instrução `global` não pode referir parâmetros formais de funções. A utilização da instrução `global` é ilustrada na seguinte função:

```
def potencia_ainda_mais_estranha_2(n):

    def potencia_estranha_4(n):
        global x
        pot = 1
        x = 7
        while n > 0:
```

[1]Na realidade, esta instrução corresponde a uma *diretiva* para o interpretador do Python.

```
            pot = pot * x
            n = n - 1
        return pot

    print('x (antes de potencia_estranha_4) =', x)
    res = potencia_estranha_4(n)
    print('x (depois de potencia_estranha_4) =', x)
    return res
```

a qual permite gerar a interação:

```
>>> x = 4
>>> potencia_ainda_mais_estranha_2(2)
x (antes de potencia_estranha_4) = 4
x (depois de potencia_estranha_4) = 7
49
```

Devido à restrição imposta pela instrução `global` de não poder alterar parâmetros formais, na função `potencia_ainda_mais_estranha_2` utilizamos `x` como uma variável global.

Ao terminar esta secção é importante dizer que a utilização exclusiva de nomes locais permite manter a independência entre funções, no sentido em que toda a comunicação entre elas é limitada à associação dos parâmetros concretos com os parâmetros formais. Quando este tipo de comunicação é mantido, para utilizar uma função, apenas é preciso saber o que ela faz, e não como foi programada. Já sabemos que isto se chama abstração procedimental.

Embora a utilização de nomes locais seja vantajosa e deva ser utilizada sempre que possível, é por vezes conveniente permitir o acesso a nomes não locais. O facto de um nome ser não local significa que o objeto computacional associado a esse nome é compartilhado por vários blocos do programa, que o podem consultar e, eventualmente, modificar.

6.2 Funções recursivas

Nesta secção abordamos um processo de definição de funções, denominado definição *recursiva* ou *por recorrência*. Diz-se que uma dada entidade é *recursiva* se

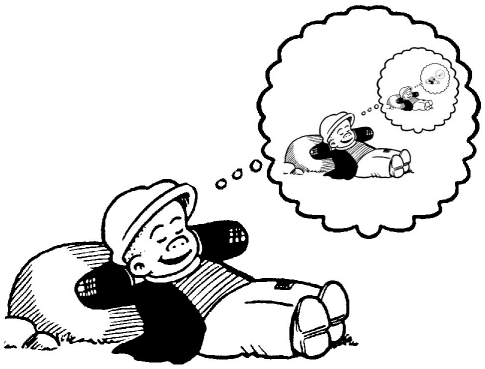


Figura 6.8: Exemplo de uma imagem recursiva.

ela for definida em termos de si própria. As definições recursivas são frequentemente utilizadas em matemática. Ao longo deste livro temos utilizado definições recursivas na apresentação das expressões em notação BNF. Por exemplo, a definição de ⟨nomes⟩ apresentada na página 76:

⟨nomes⟩ ::= ⟨nome⟩ |
⟨nome⟩ , ⟨nomes⟩

é recursiva pois ⟨nomes⟩ é definido em termos de si próprio. Esta definição afirma que ⟨nomes⟩ é ou um ⟨nome⟩ ou um ⟨nome⟩, seguido de uma vírgula, seguida de ⟨nomes⟩. A utilização de recursão é também frequente em imagens, um exemplo das quais apresentamos na Figura 6.8[2].

Como um exemplo utilizado em matemática de uma definição recursiva, consideremos a seguinte definição do conjunto dos números naturais: (1) 1 é um número natural; (2) o sucessor de um número natural é um número natural. A segunda parte desta definição é recursiva, porque utiliza o conceito de número natural para definir número natural: 2 é um número natural porque é o sucessor do número natural 1 (sabemos que 1 é um número natural pela primeira parte desta definição).

Outro exemplo típico de uma definição recursiva utilizada em matemática é a

[2]Reproduzido com autorização de United Feature Syndicate. © United Feature Syndicate, Inc. Nancy and Sluggo are a registered trademark of United Feature Syndicate, Inc.

seguinte definição da função fatorial:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases}$$

Esta definição deve ser lida do seguinte modo: o fatorial de 0 ($n = 0$) é 1; se n é maior do que 0, então o fatorial de n é dado pelo produto entre n e o fatorial de $n - 1$. A definição recursiva da função fatorial é mais sugestiva, e rigorosa, do que a seguinte definição que frequentemente é apresentada:

$$n! = n \cdot (n-1) \cdot (n-2) \ldots 2 \cdot 1.$$

Note-se que na definição não recursiva de fatorial, "..." significa que existe um padrão que se repete, padrão esse que deve ser descoberto por quem irá calcular o valor da função, ao passo que na definição recursiva, o processo de cálculo dos valores da função está completamente especificado.

O poder das definições recursivas baseia-se na possibilidade de definir um conjunto infinito utilizando uma frase finita. Do mesmo modo, um número arbitrariamente grande de cálculos pode ser especificado através de uma função recursiva, mesmo que esta função não contenha, explicitamente, estruturas de repetição.

Suponhamos que desejávamos definir a função fatorial em Python. Podemos usar a definição não recursiva, descodificando o padrão correspondente às reticências, dando origem à seguinte função (a qual corresponde a uma variante, utilizando um ciclo `for`, da função apresentada na Secção 3.4.3):

```
def fatorial(n):
    fact = 1
    for i in range(n, 0, -1):
        fact = fact * i
    return fact
```

ou podemos utilizar diretamente a definição recursiva, dando origem à seguinte função:

```
def fatorial(n):
    if n == 0:
```

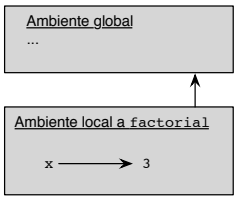


Figura 6.9: Primeira chamada a `fatorial`.

```
        return 1
    else:
        return n * fatorial(n-1)
```

Esta segunda versão de `fatorial` não contém explicitamente nenhum ciclo.

Como exercício, iremos seguir o funcionamento da versão recursiva da função `fatorial` para calcular o fatorial de 3. Ao avaliar `fatorial(3)`, o valor 3 é associado ao parâmetro formal `n` da função `fatorial`, criando-se o ambiente local apresentado na Figura 6.9 (nesta figura e nas seguintes, estamos propositadamente a ignorar as outras ligações que possam existir no ambiente global), no qual o corpo da função é executado. Como a expressão `3 == 0` tem o valor `False`, o valor devolvido será `3 * fatorial(2)`.

Encontramos agora uma outra expressão que utiliza a função `fatorial` e esta expressão inicia o cálculo de `fatorial(2)`. É importante recordar que neste momento existe um cálculo suspenso, o cálculo de `fatorial(3)`. Para calcular `fatorial(2)`, o valor 2 é associado ao parâmetro formal `n` da função `fatorial`, criando-se o ambiente local apresentado na Figura 6.10. Como a expressão `2 == 0` tem o valor `False`, o valor devolvido por `fatorial(2)` é `2 * fatorial(1)` e o valor de `fatorial(1)` terá que ser calculado. Dois cálculos da função fatorial estão agora suspensos: o cálculo de `fatorial(3)` e o cálculo de `fatorial(2)`.

Encontramos novamente outra expressão que utiliza a função `fatorial` e esta expressão inicia o cálculo de `fatorial(1)`. Para calcular `fatorial(1)`, o valor 1 é associado ao parâmetro formal `n` da função `fatorial`, criando-se o ambiente local apresentado na Figura 6.11. Como a expressão `1 == 0` tem o valor `False`, o valor devolvido por `fatorial(1)` é `1 * fatorial(0)` e o valor de `fatorial(0)` terá que ser calculado. Três cálculos da função fatorial estão agora suspensos: o cálculo de `fatorial(3)`, `fatorial(2)` e `fatorial(1)`.

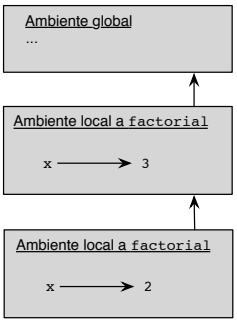


Figura 6.10: Segunda chamada a `fatorial`.

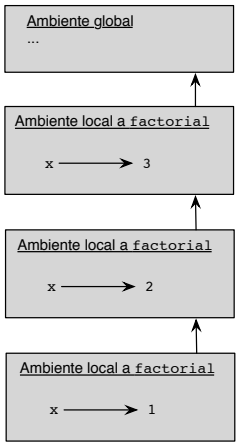


Figura 6.11: Terceira chamada a `fatorial`.

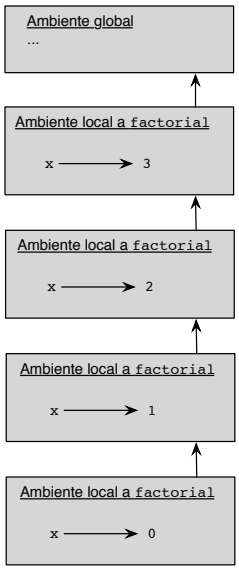


Figura 6.12: Quarta chamada a `fatorial`.

Para calcular `fatorial(0)` associa-se o valor 0 ao parâmetro formal `n` da função `fatorial`, criando-se o ambiente local apresentado na Figura 6.12 e executa-se o corpo da função. Neste caso a expressão `0 == 0` tem valor `True`, pelo que o valor de `fatorial(0)` é 1.

Pode-se agora continuar o cálculo de `fatorial(1)`, que é `1 * fatorial(0) = 1 * 1 = 1`, o que, por sua vez, permite calcular `fatorial(2)`, cujo valor é `2 * fatorial(1) = 2 * 1 = 2`, o que permite calcular `fatorial(3)`, cujo valor é `3 * fatorial(2) = 3 * 2 = 6`.

Em resumo, uma função diz-se *recursiva* se for definida em termos de si própria. A ideia fundamental numa função recursiva consiste em definir um problema em termos de uma versão semelhante, embora mais simples, de si próprio. Com efeito, quando definirmos $n!$ como $n \cdot (n-1)!$ estamos a definir fatorial em termos de uma versão mais simples de si próprio, pois o número de que queremos calcular o fatorial é mais pequeno. Esta definição é repetida sucessivamente até se atingir uma versão do problema para a qual a solução seja conhecida. Utiliza-se então essa solução para sucessivamente calcular a solução de cada um dos subproblemas gerados e produzir a resposta desejada.

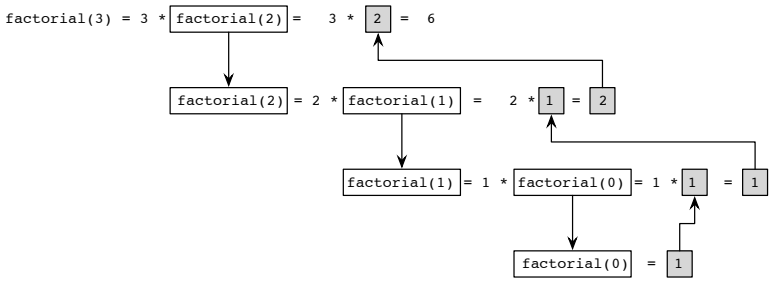


Figura 6.13: Encadeamento das operações no cálculo de `fatorial(3)`.

Como vimos, durante a avaliação de uma expressão cujo operador é `fatorial`, gera-se um encadeamento de operações suspensas à espera do valor de outras operações cujo operador é o próprio `fatorial`. Na Figura 6.13 mostramos o encadeamento das operações geradas durante o cálculo de `fatorial(3)`.

Tanto as definições recursivas como as funções recursivas são constituídas por duas partes distintas:

1. Uma *parte básica*, também chamada *caso terminal*, a qual constitui a versão mais simples do problema para o qual a solução é conhecida. No caso da função fatorial esta parte corresponde ao caso em que $n = 0$.

2. Uma *parte recursiva*, também chamada *caso geral*, na qual o problema é definido em termos de uma versão mais simples de si próprio. No caso da função fatorial, isto corresponde ao caso em que $n > 0$.

Como segundo exemplo, consideremos novamente o algoritmo de Euclides (apresentado na Secção 3.4.4) para o cálculo do máximo divisor comum entre dois números. O algoritmo apresentado para o cálculo do máximo divisor comum afirma que:

1. O máximo divisor comum entre um número e zero é o próprio número.

2. Quando dividimos um número por um menor, o máximo divisor comum entre o resto da divisão e o divisor é o mesmo que o máximo divisor comum entre o dividendo e o divisor.

Este algoritmo corresponde claramente a uma definição recursiva. No entanto,

na Secção 3.4.4 analisámos o seu comportamento e traduzimo-lo numa função não recursiva. O máximo divisor comum pode ser definido do seguinte modo[3]:

$$mdc(m,n) = \begin{cases} m & \text{se } n = 0 \\ mdc(n, m\%n) & \text{se } n > 0 \end{cases}$$

Podemos agora aplicar diretamente o algoritmo, dado origem à seguinte função que calcula o máximo divisor comum:

```
def mdc(m, n):
    if n == 0:
        return m
    else:
        return mdc(n, m % n)
```

Esta função origina o seguinte processo de cálculo, por exemplo, para calcular `mdc(24, 16)`:

```
mdc(24, 16)
mdc(16, 8)
mdc(8, 0)
8
```

Para ilustrar o poder das funções recursivas, consideremos a função `alisa` apresentada na página 110, que recebe como argumento um tuplo, cujos elementos podem ser outros tuplos, e que devolve um tuplo contendo todos os elementos correspondentes a tipos elementares de dados (inteiros, reais ou valores lógicos) do tuplo original. Esta função pode ser definida recursivamente do seguinte modo:

[3]Nesta definição usamos o símbolo do resto da divisão inteira do Python.

```
def alisa(t):
    if t == ():
        return t
    elif isinstance(t[0], tuple):
        return alisa(t[0]) + alisa(t[1:])
    else:
        return (t[0],) + alisa(t[1:])
```

Se o tuplo recebido pela função for o tuplo vazio, o seu valor é o tuplo vazio; se o primeiro elemento do tuplo for um tuplo, o valor da função corresponde a juntar o tuplo que resulta de "alisar" o primeiro elemento ao tuplo que resulta de "alisar" os restantes elementos do tuplo; em caso contrário, o resultado da função é o tuplo que se obtém adicionando o primeiro elemento do tuplo ao resultado de "alisar" os restantes elementos do tuplo.

6.3 Programação funcional

Existe um paradigma de programação, chamado *programação funcional* que é baseado exclusivamente na utilização de funções. Um *paradigma de programação* é um modelo para abordar o modo de raciocinar durante a fase de programação e, consequentemente, o modo como os programas são escritos.

O tipo de programação que temos utilizado até agora tem o nome de *programação imperativa*. Em programação imperativa, um programa é considerado como um conjunto de ordens dadas ao computador, e daí a designação de "imperativa", por exemplo, atualiza o valor desta variável, chama esta função, repete a execução destas instruções até que certa condição se verifique.

Uma das operações centrais em programação imperativa corresponde à instrução de atribuição através da qual é criada uma associação entre um nome (considerado como uma variável) e um valor ou é alterado o valor que está associado com um nome (o que corresponde à alteração do valor da variável). A instrução de atribuição leva a considerar variáveis como referências para o local onde o seu valor é guardado e o valor guardado nesse local pode variar. Um outro aspeto essencial em programação imperativa é o conceito de ciclo, uma sequência de instruções associada a uma estrutura que controla o número de vezes que essas instruções são executadas.

Em programação funcional, por outro lado, um programa é considerado como uma função matemática que cumpre os seus objetivos através do cálculo dos valores (ou da avaliação) de outras funções. Em programação funcional não existe o conceito de instrução de atribuição e podem nem existir ciclos. O conceito de repetição é realizado exclusivamente através da recursão.

Para ilustrar a utilização da programação funcional, consideremos a função potência apresentada na Secção 3.4.2. Uma alternativa para definir x^n é através da seguinte análise de casos:

$$x^n = \begin{cases} 1 & \text{se } n = 0 \\ x.(x^{n-1}) & \text{se } n > 1 \end{cases}$$

ou seja, x^0 é 1, e para $n > 0$, x^n é o produto de x pela potência de x com expoente imediatamente inferior $x.(x^{n-1})$.

Podemos traduzir diretamente esta definição para a seguinte função em Python[4]:

```
def potencia(x, n):
    if n == 0:
        return 1
    else:
        return x * potencia(x, n-1)
```

Repare-se que estamos perante uma uma função que corresponde a uma definição recursiva. A função `potencia` é definida através de uma parte básica, para a qual a solução é conhecida (se `n` = 0, o valor da potência é `1`), e de uma parte recursiva, para qual a potência é definida através de uma versão mais simples de si própria (o expoente é menor).

Nesta função não existe uma definição explícita de repetição, sendo essa repetição tratada implicitamente pela parte recursiva da função. Note-se que também não existe nenhuma instrução de atribuição, os parâmetros formais recebem os seus valores quando a função é chamada (em programação funcional, diz-se que a *função é avaliada*), sendo os valores devolvidos pela função utilizados nos cálculos subsequentes.

Como segundo exemplo, consideremos a função `raiz` apresentada na Secção 3.4.5.

[4]Propositadamente, não estamos a fazer nenhuma verificação em relação aos possíveis valores fornecidos a esta função.

A seguinte função corresponde a uma versão funcional de `raiz`:

```
def raiz(x):

    def calcula_raiz(x, palpite):

        def bom_palpite(x, palpite):
            return abs(x - palpite * palpite) < 0.001

        def novo_palpite(x, palpite):
            return (palpite + x / palpite) / 2

        if bom_palpite(x, palpite):
            return palpite
        else:
            return calcula_raiz(x, novo_palpite(x, palpite))

    if x >= 0:
        return calcula_raiz (x, 1)
    else:
        raise ValueError ('raiz: argumento negativo')
```

A função `calcula_raiz` apresentada na página 88 foi substituída por uma versão recursiva que evita a utilização do ciclo `while`.

Como exemplo final, o seguinte programa devolve a soma dos dígitos do número fornecido pelo utilizador, recorrendo à função `soma_digitos`. Repare-se na não existência de instruções de atribuição, sendo o valor fornecido pelo utilizador o parâmetro concreto da função `soma_digitos`.

```
def prog_soma():
    return soma_digitos(eval(input('Escreva um inteiro\n? ')))

def soma_digitos(n):
    if n == 0:
        return n
    else:
        return n % 10 + soma_digitos(n // 10)
```

6.4 Funções como parâmetros

Nos capítulos anteriores argumentámos que as funções correspondem a abstrações que definem operações compostas, independentemente dos valores por estas utilizados. Na realidade, ao definirmos a função `quadrado` como

```
def quadrado(x):
    return x * x
```

não estamos a falar do quadrado de um número em particular, mas sim de um padrão de cálculo que permite calcular o quadrado de qualquer número. Poderíamos escrever programas sem nunca recorrer à função `quadrado` mas sempre que precisássemos de utilizar o quadrado de uma entidade particular teríamos que recorrer à expressão que calcula o quadrado, `25 * 25`, `x * x`, `y * y`. A abstração procedimental permite introduzir o *conceito de quadrado*, evitando que tenhamos sempre de exprimi-lo em termos das operações elementares da linguagem. Sem a introdução do conceito de quadrado, poderíamos calcular o quadrado de qualquer número, mas faltava-nos a abstração através da qual podemos referir-nos à operação capturada pelo conceito.

Nesta secção apresentamos conceitos mais abstratos do que os que temos utilizado até aqui com a noção de função. Recordemos mais uma vez que uma função captura um padrão de cálculo.

Consideremos três funções que, embora distintas, englobam um padrão de cálculo comum[5]. Estas funções calculam, respetivamente, a soma dos números naturais entre `l_inf` e `l_sup`

```
def soma_naturais(l_inf, l_sup):
    soma = 0
    while l_inf <= l_sup:
        soma = soma + l_inf
        l_inf = l_inf + 1
    return soma
```

a soma dos quadrados dos números naturais entre `l_inf` e `l_sup`

[5]Este exemplo foi adaptado de [Abelson et al., 1996], páginas 57 e 58.

```
def soma_quadrados(l_inf, l_sup):
    soma = 0
    while l_inf <= l_sup:
        soma = soma + quadrado(l_inf)
        l_inf = l_inf + 1
    return soma
```

e a soma dos inversos dos quadrados dos números naturais ímpares entre `l_inf` e `l_sup`[6]

```
def soma_inv_quadrados_impares(l_inf, l_sup):
    soma = 0
    while l_inf <= l_sup:
        soma = soma + 1 / quadrado(l_inf)
        l_inf = l_inf + 2
    return soma
```

Estas funções permitem gerar a interação:

```
>>> soma_naturais(1, 40)
820
>>> soma_quadrados(1, 40)
22140
>>> soma_inv_quadrados_impares(1, 40)
1.2212031520286797
```

Estas funções partilham um padrão de cálculo em comum, diferindo entre si nos seguintes aspetos: (1) o seu nome; (2) o processo de cálculo do termo a ser adicionado; e (3) o processo do cálculo do próximo termo a adicionar. Poderíamos criar cada ums destas funções substituindo os símbolos não terminais no seguinte padrão de cálculo:

```
def ⟨nome⟩ (linf, lsup):
   soma = 0
   while linf <= lsup:
      soma = soma + ⟨calc_termo⟩(linf)
```

[6] Esta função assume que `l_inf` e `l_sup` são números ímpares.

```
      linf = ⟨prox⟩(linf)
   return soma
```

Os símbolos não terminais no padrão anterior assumem valores particulares para cada uma das funções. A existência deste padrão mostra que existe uma abstração escondida subjacente a este cálculo. Os matemáticos identificaram esta abstração através do conceito de somatório:

$$\sum_{n=l_{inf}}^{l_{sup}} f(n) = f(l_{inf}) + \cdots + f(l_{sup})$$

O poder da abstração correspondente ao somatório permite lidar com o *conceito de soma* em vez de tratar apenas com somas particulares. A existência da abstração correspondente ao somatório leva-nos a pensar em definir uma função correspondente a esta abstração em vez de apenas utilizar funções que calculam somas particulares.

Um dos processos para tornar as funções mais gerais corresponde a utilizar parâmetros adicionais que indicam quais as operações a efetuar sobre os objetos computacionais manipulados pelas funções, e assim idealizar uma função correspondente a um somatório que, para além de receber a indicação sobre os elementos extremos a somar (`l_inf` e `l_sup`), recebe também como parâmetros as funções para calcular um termo do somatório (`calc_termo`) e para calcular o próximo termo a adicionar (`prox`):

```
def somatorio(calc_termo, linf, prox, lsup):
    soma = 0
    while linf <= lsup:
        soma = soma + calc_termo(linf)
        linf = prox(linf)
    return soma
```

Definindo agora funções para adicionar uma unidade ao seu argumento (`inc1`), para devolver o seu próprio argumento (`identidade`), para adicionar duas unidades ao seu argumento (`inc2`), para calcular o quadrado do seu argumento (`quadrado`) e para calcular o inverso do quadrado do seu argumento (`inv_quadrado`):

```
def inc1(x):
    return x + 1


def identidade(x):
    return x

def inc2(x):
    return x + 2

def quadrado (x):
    return x * x

def inv_quadrado(x):
    return 1 / quadrado(x)
```

Obtemos a seguinte interação:

```
>>> somatorio(identidade, 1, inc1, 40)
820
>>> somatorio(quadrado, 1, inc1, 40)
22140
>>> somatorio(inv_quadrado, 1, inc2, 40)
1.2212031520286797
```

Ou seja, utilizando funções como parâmetros, somos capazes de definir a função `somatorio` que captura o conceito de somatório utilizado em Matemática. Com esta função, cada vez que precisamos de utilizar um somatório, em lugar de escrever um programa, utilizamos um programa existente. As funções que utilizam funções como argumentos são conhecidas por *funções de ordem superior* ou *funcionais*.

Existem dois aspetos que é importante observar em relação à função anterior. Em primeiro lugar, a função `somatorio` não foi exatamente utilizada com funções como parâmetros, mas sim com parâmetros que correspondem a nomes de funções. Em segundo lugar, esta filosofia levou-nos a criar funções que podem ter pouco interesse, fora do âmbito das somas que estamos interessados em calcular, por exemplo, a função `identidade`.

Recorde-se a discussão sobre funções apresentada nas páginas 74–75. Podemos reparar que o nome da função é, de certo modo, supérfluo, pois o que na realidade nos interessa é saber *como* calcular o valor da função para um dado argumento – a função é o conjunto dos pares ordenados. A verdadeira importância do nome da função é a de fornecer um modo de podermos *falar sobre* ou *designar* a função. Em 1941, o matemático Alonzo Church inventou uma notação para modelar funções a que se dá o nome de *cálculo lambda*[7]. No cálculo lambda, a função que soma 3 ao seu argumento é representada por $\lambda(x)(x+3)$. Nesta notação, imediatamente a seguir ao símbolo λ aparece a lista dos argumentos da função, a qual é seguida pela expressão designatória que permite calcular o valor da função. Uma das vantagens do cálculo lambda é permitir a utilização de funções sem ter que lhes dar um nome. Para representar a aplicação de uma função a um elemento do seu domínio, escreve-se a função seguida do elemento para o qual se deseja calcular o valor. Assim, $(\lambda(x)(x+3))(3)$ tem o valor 6; da mesma forma, $(\lambda(x,y)(x \cdot y))(5,6)$ tem o valor 30.

Em Python existe a possibilidade de definir funções sem nome, as *funções anónimas*, recorrendo à notação lambda. Uma função anónima é definida através da seguinte expressão em notação BNF:

⟨função anónima⟩ ::= `lambda` ⟨parâmetros formais⟩ `:` ⟨expressão⟩

Nesta definição, ⟨expressão⟩ corresponde a uma expressão em Python.

Ao encontrar uma ⟨função anónima⟩, o Python cria uma função cujos parâmetros formais correspondem a ⟨parâmetros formais⟩ e cujo corpo corresponde a ⟨expressão⟩. Quando esta função anónima é executada, os parâmetros concretos são associados aos parâmetros formais, e o valor da função é o valor da ⟨expressão⟩. Note-se que numa função anónima não existe uma instrução `return` (estando esta implicitamente associada a ⟨expressão⟩).

Por exemplo, `lambda x : x + 1` é uma função anónima que devolve o valor do seu argumento mais um. Com esta função anónima podemos gerar a interação:

```
>>> (lambda x : x + 1)(3)
4
```

O que nos interessa fornecer à função `somatorio` não são os nomes das funções que calculam um termo do somatório e que calculam o próximo termo a adicio-

[7] Do inglês, "lambda calculus" [Church, 1941].

nar, mas sim as próprias funções. Tendo em atenção, por exemplo, que a função que adiciona 1 ao seu argumento é dado por `lambda x : x + 1`, independentemente do nome com que é batizada, podemos utilizar a função `somatorio`, recorrendo a funções anónimas como mostra a seguinte interação (a qual pressupõe a definição da função `quadrado`):

```
>>> somatorio(lambda x : x, 1, lambda x : x + 1, 40)
820
>>> somatorio(quadrado, 1, lambda x : x + 1, 40)
22140
>>> somatorio(lambda x : 1/quadrado(x), 1, lambda x : x + 2, 40)
1.2212031520286797
```

Note-se que, a partir da definição da função `somatorio`, estamos em condições de poder calcular qualquer somatório. Por exemplo, usando a definição da função `fatorial` apresentada na página 181, podemos calcular a soma dos fatoriais dos 20 primeiros números naturais através de:

```
>>> somatorio(fatorial, 1, lambda x : x + 1, 20)
2561327494111820313
```

6.4.1 Funcionais sobre listas

Com a utilização de listas é vulgar recorrer a um certo número de funcionais, os quais se classificam em transformadores, filtros e acumuladores.

- Um *transformador* é um funcional que recebe como argumentos uma lista e uma operação aplicável aos elementos da lista, e devolve uma lista em que cada elemento resulta da aplicação da operação ao elemento correspondente da lista original. Podemos realizar um transformador através da seguinte função:

  ```
  def transforma(tr, lst):
      res = list()
      for e in lst:
          res = res + [tr(e)]
      return res
  ```

Em alternativa, a função `transforma` pode ser escrita como uma função recursiva do seguinte modo:

```
def transforma(tr, lst):
    if lst == []:
        return lst
    else:
        return [tr(lst[0])] + transforma(tr, lst[1:])
```

A seguinte interação corresponde a uma utilização de um transformador, utilizando a função `quadrado` da página 76:

```
>>> transforma(quadrado, [1, 2, 3, 4, 5, 6])
[1, 4, 9, 16, 25, 36]
```

- Um *filtro* é um funcional que recebe como argumentos uma lista e um predicado aplicável aos elementos da lista, e devolve a lista constituída apenas pelos elementos da lista original que satisfazem o predicado. Podemos realizar um filtro através da seguinte função:

```
def filtra(teste, lst):
    res = list()
    for e in lst:
        if teste(e):
            res = res + [e]
    return res
```

ou, alternativamente, através da seguinte função recursiva:

```
def filtra(teste, lst):
    if lst == []:
        return lst
    elif teste(lst[0]):
        return [lst[0]] + filtra(teste, lst[1:])
    else:
        return filtra(teste, lst[1:])
```

A seguinte interação corresponde a uma utilização de um filtro que testa se um número é par:

```
>>> filtra(lambda x : x % 2 == 0, [1, 2, 3, 4, 5, 6])
[2, 4, 6]
```

- Um *acumulador* é um funcional que recebe como argumentos uma lista e uma operação aplicável aos elementos da lista, e aplica sucessivamente essa operação aos elementos da lista original, devolvendo o resultado da aplicação da operação a todos os elementos da lista. Podemos realizar um acumulador através da seguinte função[8]:

  ```
  def acumula(fn, lst):
      res = lst[0]
      for i in range(1,len(lst)):
          res = fn(res, lst[i])
      return res
  ```

  ou, alternativamente, através da seguinte função recursiva:

  ```
  def acumula(fn, lst):
      if len(lst) == 1:
          return lst[0]
      else:
          return fn(lst[0], acumula(fn, lst[1:]))
  ```

  A seguinte interação corresponde a uma utilização de um acumulador, calculando o produto de todos os elementos da lista:

  ```
  >>> acumula(lambda x, y : x * y, [1, 2, 3, 4, 5])
  120
  ```

A seguinte utilização de funcionais calcula a soma dos quadrados dos elementos ìmpares da lista `[2, 4, 5, 29, 30]`:

```
acumula(lambda x, y : x + y,
        transforma(quadrado,
                   filtra(lambda x : x % 2 == 1,
                          [2, 4, 5, 29, 30])))
```

[8] Como exercício, o leitor deverá explicar a razão da variável `res` ser inicializada para `lst[0]` e não para `0`.

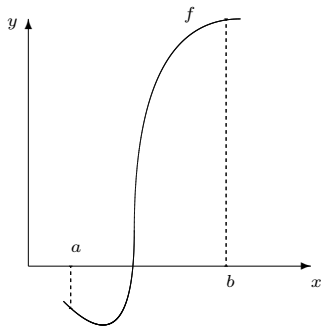


Figura 6.14: Situação à qual o método do intervalo é aplicável.

6.4.2 Funções como métodos gerais

Nesta secção apresentamos um exemplo que mostra como a utilização de funções como argumentos de funções pode originar métodos gerais de computação, independentemente das funções envolvidas.

Para isso, apresentamos um processo de cálculo de raízes de equações pelo *método do intervalo*. Este método é aplicável ao cálculo de raízes de funções contínuas entre dois pontos e baseia-se no corolário do teorema de Bolzano, o qual afirma que, se $f(x)$ é uma função contínua entre os pontos a e b, tais que $f(a) < 0 < f(b)$, então podemos garantir que $f(x)$ tem um zero entre os pontos a e b (Figura 6.14).

Nesta situação, para calcular a raiz da função f no intervalo $[a,\ b]$, calcula-se o valor de $f(x)$ no ponto médio do intervalo. Se este valor for positivo, podemos garantir que $f(x)$ tem um zero entre a e o valor médio, $(a+b)/2$; se este valor for negativo, podemos garantir que $f(x)$ tem um zero entre o valor médio, $(a+b)/2$, e b. Podemos então repetir o processo com o novo intervalo. Este processo será repetido até que o intervalo em consideração seja suficientemente pequeno.

Consideremos a seguinte função em Python que recebe como argumentos uma função contínua (`f`) e os extremos de um intervalo no qual a função assume um valor negativo e um valor positivo, respetivamente, `p_neg` e `p_pos`. Esta função calcula o zero da função no intervalo especificado, utilizando o método do intervalo.

```
def calcula_raiz(f, p_neg, p_pos):
    while not suf_perto(p_neg, p_pos):
        p_medio = (p_neg + p_pos) / 2
        if f(p_medio) > 0:
            p_pos = p_medio
        elif f(p_medio) < 0:
            p_neg = p_medio
        else:
            return p_medio
    return (p_neg + p_pos) / 2
```

Esta função parte da existência de uma função para decidir se dois valores estão suficientemente próximos (`suf_perto`). Note-se ainda que o `else` da instrução de seleção serve para lidar com os casos em que `f(p_medio)` não é nem positivo, nem negativo, ou seja, `f(p_medio)` corresponde a um valor nulo e consequentemente é um zero da função.

Para definir quando dois valores estão suficientemente próximos utilizamos a mesma abordagem que usámos no cálculo da raiz quadrada apresentado na Secção 3.4.5: quando a sua diferença em valor absoluto for menor do que um certo limiar (suponhamos que este limiar é 0.001). Podemos assim escrever a função `suf_perto`:

```
def suf_perto(a, b):
    return abs(a - b) < 0.001
```

Notemos que a função `calcula_raiz` parte do pressuposto que os valores extremos do intervalo correspondem a valores da função, `f`, com sinais opostos e que `f(p_neg)` $<$ 0 $<$ `f(p_pos)`. Se algum destes pressupostos não se verificar, a função não calcula corretamente a raiz. Para evitar este problema potencial, definimos a seguinte função:

```
def met_intervalo(f, a, b):
    fa = f(a)
    fb = f(b)
    if fa < 0 < fb:
        return calcula_raiz(f, a, b)
    elif fb < 0 < fa:
```

```
        return calcula_raiz(f, b, a)
    else:
        print('Metodo do intervalo: valores não opostos')
```

Com base nesta função, podemos obter a seguinte interação:

```
>>> met_intervalo(lambda x : x * x * x - 2 * x - 3, 1, 2)
1.8935546875
>>> met_intervalo(lambda x : x * x * x - 2 * x - 3, 1, 1.2)
Metodo do intervalo: valores não opostos
```

Sabemos também que devemos garantir que a função `raiz` apenas é utilizada pela função `met_intervalo`, o que nos leva a definir a seguinte estrutura de blocos:

```
def met_intervalo(f, a, b):

    def calcula_raiz(f, p_neg, p_pos):
        while not suf_perto(p_neg, p_pos):
            p_medio = (p_neg + p_pos) / 2
            if f(p_medio) > 0:
                p_pos = p_medio
            elif f(p_medio) < 0:
                p_neg = p_medio
            else:
                return p_medio
        return (p_neg + p_pos) / 2

    def suf_perto(a, b):
        return abs(a - b) < 0.001

    fa = f(a)
    fb = f(b)
    if fa < 0 < fb:
        return calcula_raiz(f, a, b)
    elif fb < 0 < fa:
        return calcula_raiz(f, b, a)
```

```
    else:
        print('Metodo do intervalo: valores não opostos')
```

6.5 Funções como valor de funções

Na secção anterior vimos que em Python as funções podem ser utilizadas como argumentos de outras funções. Esta utilização permite a criação de abstrações mais gerais do que as obtidas até agora. Nesta secção discutimos funções como valores produzidos por funções.

6.5.1 Cálculo de derivadas

Suponhamos que queremos desenvolver uma função que calcula o valor da derivada de uma função real de variável real, f, num dado ponto. Esta função real de variável real pode ser, por exemplo, o quadrado de um número, e deverá ser um dos argumentos da função.

Por definição, sendo a um ponto do domínio da função f, a derivada de f no ponto a, representada por $f'(a)$, é dada por:

$$f'(a) = \lim_{x \to a} \frac{f(x) - f(a)}{x - a}$$

ou, fazendo $h = x - a$,

$$f'(a) = \lim_{h \to 0} \frac{f(a + h) - f(a)}{h}$$

Sendo dx um número suficientemente pequeno, podemos considerar que a seguinte fórmula fornece uma boa aproximação para a derivada da função f no ponto a:

$$f'(a) \cong \frac{f(a + dx) - f(a)}{dx}$$

Podemos agora definir a função `derivada_ponto`, a qual recebe como argumento uma função correspondente ao argumento `f`, e um ponto do seu domínio, `a`, e produz o valor da derivada nesse ponto:

```
def derivada_ponto(f, a):
    return (f(a + dx) - f(a)) / dx
```

Esta função, juntamente com a definição do que se entende por algo suficientemente pequeno, `dx` (por exemplo, 0.00001),

```
dx = 0.00001
```

permite calcular a derivada de funções arbitrárias em pontos particulares do seu domínio. A seguinte interação calcula a derivada da função `quadrado` (apresentada na página 76) para os pontos 3 e 10^9:

```
>>> derivada_ponto(quadrado, 3)
6.000009999951316
>>> derivada_ponto(quadrado, 10)
20.00000999942131
```

Em Matemática, após a definição de derivada num ponto, é possível definir a *função derivada*, a qual a cada ponto do domínio da função original associa o valor da derivada nesse ponto. O conceito de derivada de uma função é suficientemente importante para ser capturado como uma abstração. Repare-se que a derivada num ponto arbitrário x é calculada substituindo x por a na função que apresentámos. Consideremos a seguinte função:

```
def fn_derivada(x):
    return (f(x + dx) - f(x)) / dx
```

Esta função utiliza uma variável livre, `f`, correspondente a uma função e calcula o valor da derivada de `f` para qualquer ponto do seu domínio. Com esta função podemos gerar a interação, a qual pressupõe a definição da função `quadrado`:

```
>>> f = quadrado
>>> dx = 0.00001
>>> fn_derivada(3)
6.000009999951316
>>> fn_derivada(10)
20.00000999942131
```

[9] Note-se que a definição da função `quadrado` não é necessária, pois podemos usar `lambda x : x * x`. No entanto, a definição da função `quadrado` torna as nossas expressões mais legíveis.

O que ainda nos falta na função anterior é capturar o "conceito de função". Assim, podemos definir a seguinte função:

```
def derivada(f):

    def fn_derivada(x):
        return (f(x + dx) - f(x)) / dx

    return fn_derivada
```

A função `derivada` recebe como parâmetro uma função, `f`, e devolve como valor a função (ou seja, o valor de `derivada(f)` é uma função) que, quando aplicada a um valor, `a`, produz a derivada da função `f` para o ponto `a`. Ou seja, `derivada(f)(a)` corresponde à derivada de `f` no ponto `a`.

Podemos agora gerar a interação:

```
>>> derivada(quadrado)
<function fn_derivada at 0x10f45d0>
>>> derivada(quadrado)(3)
6.000009999951316
>>> derivada(quadrado)(10)
20.00000999942131
```

Nada nos impede de dar um nome a uma função derivada, obtendo a interação:

```
>>> der_quadrado = derivada(quadrado)
>>> der_quadrado(3)
6.000009999951316
```

6.5.2 Raízes pelo método de Newton

Na Secção 6.4.2 apresentámos uma solução para o cálculo de raízes de equações pelo método do intervalo. Outro dos métodos muito utilizados para determinar raízes de equações é o método de Newton, o qual é aplicável a funções diferenciáveis, e consiste em partir de uma aproximação, x_n, para a raiz de uma função diferenciável, f, e calcular, como nova aproximação, o ponto onde

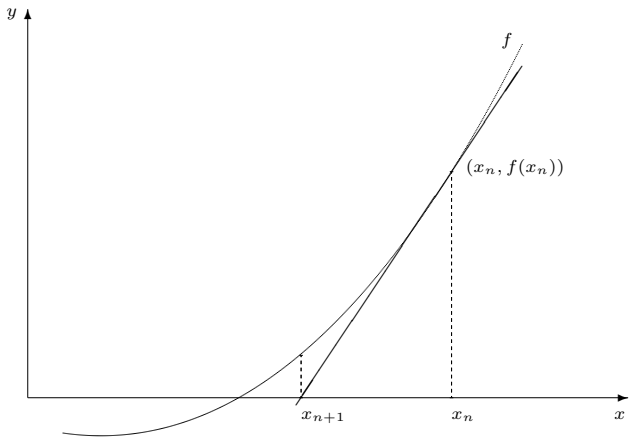


Figura 6.15: Representação gráfica subjacente ao método de Newton.

a tangente ao gráfico da função no ponto $(x_n, f(x_n))$ interseta o eixo dos xx (Figura 6.15). É fácil de concluir que a nova aproximação será dada por:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

A função matemática que para um dado valor de uma aproximação calcula uma nova aproximação é chamada *transformada de Newton*. A transformada de Newton é definida por:

$$t_N(x) = x - \frac{f(x)}{f'(x)}$$

A determinação da raiz de uma função f, utilizando o método de Newton, corresponde a começar com uma primeira aproximação para a raiz (um palpite), x_0, e gerar os valores:

$$\begin{aligned} &x_0 \\ &x_1 = t_N(x_0) \\ &x_2 = t_N(x_1) \\ &\quad\vdots \\ &x_n = t_N(x_{n-1}) \end{aligned}$$

Podemos agora definir as seguintes funções para calcular a raiz de uma função

(representada por `f`), com base num palpite inicial (`palpite`):

```
def met_newton(f, palpite):
    tf_N = transf_newton(f)
    while not boa_aprox(f(palpite)):
        palpite = tf_N(palpite)
    return palpite


def transf_newton(f):
    def t_n(x):
        return x - f(x) / derivada(f)(x)
    return t_n


def boa_aprox(x):
    return abs(x) < dx
```

Com base no método de Newton, e definindo `dx` como `0.0001`, podemos agora calcular os zeros das seguintes funções:

```
>>> met_newton(lambda x : x * x * x - 2 * x - 3, 1.0)
1.8932892212475259
>>> from math import *
>>> met_newton(sin, 2.0)
3.1415926536589787
```

Suponhamos agora que queríamos definir a função arco de tangente (*arctg*) a partir da função tangente (*tg*). Sabemos que $arctg(x)$ é o número y tal que $x = tg(y)$, ou seja, para um dado x, o valor de $arctg(x)$ corresponde ao zero da equação $tg(y) - x = 0$. Recorrendo ao método de Newton, podemos então definir:

```
def arctg(x):
    # tg no módulo math tem o nome tan
    return met_newton(lambda y : tan(y) - x, 1.0)
```

obtendo a interação:

```
>>> from math import *
>>> arctg(0.5)
0.4636478065118169
```

6.6 Notas finais

Neste capítulo, introduzimos o conceito de função recursiva, uma função que se utiliza a si própria. Generalizámos o conceito de função através da introdução de dois aspetos, a utilização de funções como argumentos para funções e a utilização de funções que produzem objetos computacionais que correspondem a funções. Esta generalização permite-nos definir abstrações de ordem superior através da construção de funções que correspondem a métodos gerais de cálculo.

Introduzimos também um paradigma de programação conhecido por programação funcional.

O livro [Hofstader, 1979] (também disponível em português [Hofstader, 2011]), galardoado com o Prémio Pulitzer em 1980, ilustra o tema da recursão (auto-referência) discutindo como a utilização de regras formais permite que sistemas adquiram significado apesar de serem constituídos por símbolos sem significado.

6.7 Exercícios

1. Escreva uma função recursiva em Python que recebe um número inteiro positivo e devolve a soma dos seus dígitos pares. Por exemplo,

   ```
   >>> soma_digitos_pares(234567)
   12
   ```

2. Escreva uma função recursiva em Python que recebe um número inteiro positivo e devolve o inteiro correspondente a inverter a ordem dos seus dígitos. Por exemplo,

   ```
   >>> inverte_digitos(7633256)
   6523367
   ```

3. Utilizando os funcionais sobre listas escreva uma função que recebe uma lista de inteiros e que devolve a soma dos quadrados dos elementos da lista.

4. Defina uma função de ordem superior que recebe funções para calcular as funções reais de variável real f e g e que se comporta como a seguinte função matemática:

$$h(x) = f(x)^2 + 4g(x)^3$$

5. A função `piatorio` devolve o produto dos valores de uma função, `fn`, para pontos do seu domínio no intervalo `[a, b]` espaçados pela função `prox`:

```
def piatorio(fn, a, prox, b):
    res = 1
    valor = a
    while valor <= b:
        res = res * fn(valor)
        valor = prox(valor)
    return res
```

Use a função `piatorio` para definir a função *sin* que calcula o seno de um número, usando a seguinte aproximação:

$$sin(x) = x\left(1-\left(\frac{x}{\pi}\right)^2\right)\left(1-\left(\frac{x}{2\pi}\right)^2\right)\left(1-\left(\frac{x}{3\pi}\right)^2\right)\cdots$$

A sua função deverá receber o número para o qual se pretende calcular o seno e o número de factores a considerar.

6. Escreva a função `faz_potencia` que recebe, como argumento, um inteiro `n` não negativo e devolve uma função que calcula a potência `n` de um número. Por exemplo, `faz_potencia(3)` devolve a função que calcula o cubo do seu argumento. Esta função é ilustrada na seguinte interação:

```
>>> faz_potencia(3)
<function f_p at 0x10f4540>
>>> faz_potencia(3)(2)
8
```

Este exercício é um exemplo da definição de uma função de dois argumentos como uma função de um argumento, cujo valor é uma função de

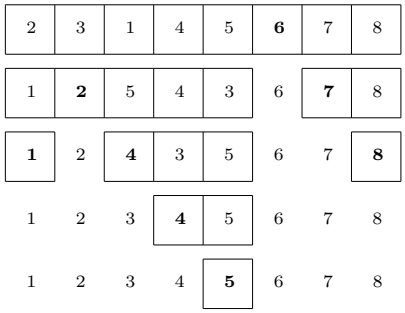
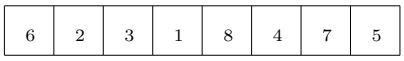


Figura 6.16: Passos seguidos no *quick sort*.

um argumento. Esta técnica, que é útil quando desejamos manter um dos argumentos fixo, enquanto o outro pode variar, é conhecida como método de *Curry*, e foi concebida por Moses Schönfinkel [Schönfinkel, 1977] e baptizada em honra do matemático americano Haskell B. Curry (1900–1982).

7. Um método de ordenação muito eficiente, chamado *quick sort*, consiste em considerar um dos elementos a ordenar (em geral, o primeiro elemento da lista), e dividir os restantes em dois grupos, um deles com os elementos menores e o outro com os elementos maiores que o elemento considerado. Este é colocado entre os dois grupos, que são por sua vez ordenados utilizando *quick sort*. Por exemplo, os passos apresentados na Figura 6.16 correspondem à ordenação da lista

   utilizando *quick sort* (o elemento escolhido em cada lista representa-se a carregado).

   Escreva uma função em Python para efetuar a ordenação de uma lista utilizando *quick sort*.

8. Considere a função `derivada` apresentada na página 203. Com base na sua definição escreva uma função que recebe uma função correspondente a uma função e um inteiro n ($n \geq 1$) e devolve a derivada de ordem n da

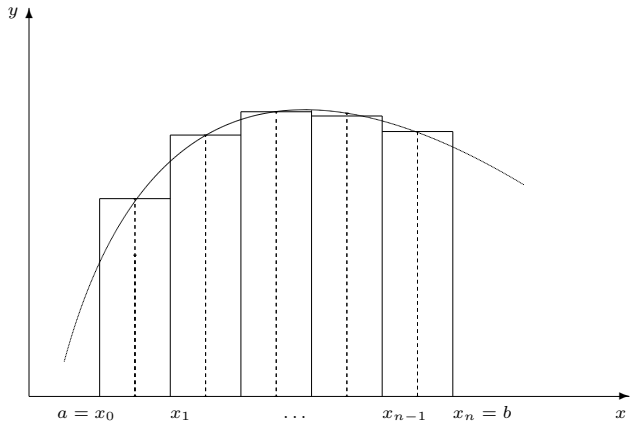


Figura 6.17: Área aproximada sob a curva.

função. A derivada de ordem n de uma função é a derivada da derivada de ordem $n-1$.

9. Tendo em atenção que $\sqrt{x}$ é o número y tal que $y^2 = x$, ou seja, para um dado x, o valor de $\sqrt{x}$ corresponde ao zero da equação $y^2 - x = 0$, utilize o método de Newton para escrever uma função que calcula a raiz quadrada de um número.

10. Dada uma função f e dois pontos a e b do seu domínio, um dos métodos para calcular a área entre o gráfico da função e o eixo dos xx no intervalo $[a,\ b]$ consiste em dividir o intervalo $[a,\ b]$ em n intervalos de igual tamanho, $[x_0,\ x_1]$, $[x_1,\ x_2]$, ..., $[x_{n-2},\ x_{n-1}]$, $[x_{n-1},\ x_n]$, com $x_0 = a$, $x_n = b$, e $\forall i, j\ x_i - x_{i-1} = x_j - x_{j-1}$. A área sob o gráfico da função será dada por (Figura 6.17):

$$\sum_{i=1}^{n} f\left(\frac{x_i + x_{i-1}}{2}\right)\ (x_i - x_{i-1})$$

Escreva em Python uma função de ordem superior que recebe como argumentos uma função (correspondente à função f) e os valores de a e b e que calcula o valor da área entre o gráfico da função e o eixo dos xx no intervalo $[a,\ b]$. A sua função poderá começar a calcular a área para

o intervalo $[x_0,\ x_1] = [a,\ b]$ e ir dividindo sucessivamente os intervalos $[x_{i-1},\ x_i]$ ao meio, até que o valor da área seja suficientemente bom.

11. Escreva uma função chamada `rasto` que recebe como argumentos uma cadeia de caracteres correspondendo ao nome de uma função, e uma função de um argumento.

    A função `rasto devolve` uma função de um argumento que escreve no écrã a indicação de que a função foi avaliada e o valor do seu argumento, escreve também o resultado da função e tem como valor o resultado da função. Por exemplo, partindo do princípio que a função `quadrado` foi definida, podemos gerar a seguinte interação:

```
>>> rasto_quadrado = rasto('quadrado', quadrado)
>>> rasto_quadrado(3)
Avaliação de  quadrado com argumento  3
Resultado  9
9
```