CAPÍTULO 04

4. Mais ferramentas de controle de fluxo

Além do comando `while` recém apresentado, Python tem as estruturas usuais de controle de fluxo conhecidas em outras linguagens, com algumas particulares.

4.1. Comandos `if`

Provavelmente o mais conhecido comando de controle de fluxo é o `if`. Por exemplo:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

Pode haver zero ou mais partes `elif`, e a parte `else` é opcional. A palavra-chave '`elif`' é uma abreviação para 'else if', e é útil para evitar indentação excessiva. Uma sequência `if` ... `elif` ... `elif` ... substitui os comandos `switch` ou `case`, encontrados em outras linguagens.

Se você está comparando o mesmo valor com várias constantes, ou verificando por tipos ou atributos específicos, você também pode achar a instrução `match` útil. Para mais detalhes veja Instruções match.

4.2. Comandos `for`

O comando `for` em Python é um pouco diferente do que costuma ser em C ou Pascal. Ao invés de sempre iterar sobre uma progressão aritmética de números (como no Pascal), ou permitir ao usuário definir o passo de iteração e a condição de parada (como C), o comando `for` do Python itera sobre os itens de qualquer sequência (seja uma lista ou uma string), na ordem que aparecem na sequência. Por exemplo:

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Código que modifica uma coleção sobre a qual está iterando pode ser inseguro. No lugar disso, usualmente você deve iterar sobre uma cópia da coleção ou criar uma nova coleção:

```
# Create a sample collection
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎':
'active'}

# Strategy:  Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy:  Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3. A função `range()`

Se você precisa iterar sobre sequências numéricas, a função embutida `range()` é a resposta. Ela gera progressões aritméticas:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

O ponto de parada fornecido nunca é incluído na lista; `range(10)` gera uma lista com 10 valores, exatamente os índices válidos para uma sequência de

comprimento 10. É possível iniciar o intervalo com outro número, ou alterar a razão da progressão (inclusive com passo negativo):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

Para iterar sobre os índices de uma sequência, combine `range()` e `len()` da seguinte forma:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Na maioria dos casos, porém, é mais conveniente usar a função `enumerate()`, veja Técnicas de iteração.

Uma coisa estranha acontece se você imprime um intervalo:

```
>>> range(10)
range(0, 10)
```

Em muitos aspectos, o objeto retornado pela função `range()` se comporta como se fosse uma lista, mas na verdade não é. É um objeto que retorna os itens sucessivos da sequência desejada quando você itera sobre a mesma, mas na verdade ele não gera a lista, economizando espaço.

Dizemos que um objeto é iterável, isso é, candidato a ser alvo de uma função ou construção que espera alguma coisa capaz de retornar sucessivamente seus elementos um de cada vez. Nós vimos que o comando `for` é um exemplo de construção, enquanto que um exemplo de função que recebe um iterável é `sum()`:

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

Mais tarde, veremos mais funções que retornam iteráveis e tomam iteráveis como argumentos. No capítulo Estruturas de dados, iremos discutir em mais detalhes sobre `list()`.

4.4. Comandos `break` e `continue`, e cláusula `else`, nos laços de repetição

O comando `break`, como no C, sai imediatamente do laço de repetição mais interno, seja `for` ou `while`.

Laços podem ter uma cláusula `else`, que é executada sempre que o laço se encerra por exaustão do iterável (no caso do `for`) ou quando a condição se torna falsa (no caso do `while`), mas nunca quando o laço é interrompido por um `break`. Isto é exemplificado no próximo exemplo que procura números primos:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Sim, o código está correto. Olhe atentamente: a cláusula `else` pertence ao laço `for`, e **não** ao comando `if`.)

Quando usado em um laço, a cláusula `else` tem mais em comum com a cláusula `else` de um comando `try` do que com a de um comando `if`: a cláusula `else` de um comando `try` executa quando não ocorre exceção, e o `else` de um laço executa quando não ocorre um `break`. Para mais informações sobre comando `try` e exceções, veja Tratamento de exceções.

A instrução `continue`, também emprestada da linguagem C, continua com a próxima iteração do laço:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found an odd number", num)
...
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5. Comandos `pass`

O comando `pass` não faz nada. Pode ser usada quando a sintaxe exige um comando mas a semântica do programa não requer nenhuma ação. Por exemplo:

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

Isto é usado muitas vezes para se definir classes mínimas:

```
>>> class MyEmptyClass:
...     pass
...
```

Outra ocasião em que o `pass` pode ser usado é como um substituto temporário para uma função ou bloco condicional, quando se está trabalhando com código novo, ainda indefinido, permitindo que mantenha-se o pensamento num nível mais abstrato. O `pass` é silenciosamente ignorado:

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

4.6. Instruções `match`

Uma instrução de correspondência pega uma expressão e compara seu valor com padrões sucessivos fornecidos como um ou mais blocos de case. Isso é superficialmente semelhante a uma instrução switch em C, Java ou JavaScript (e muitas outras linguagens), mas também pode extrair componentes (elementos de sequência ou atributos de objeto) do valor em variáveis.

A forma mais simples compara um valor de assunto com um ou mais literais:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Observe o último bloco: o “nome da variável” `_` atua como um *curinga* e nunca falha em corresponder. Se nenhum caso corresponder, nenhuma das ramificações será executada.

Você pode combinar vários literais em um único padrão usando `|` (“ou”):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Os padrões podem se parecer com atribuições de desempacotamento e podem ser usados para vincular variáveis:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
```

```
        raise ValueError("Not a point")
```

Estude isso com cuidado! O primeiro padrão tem dois literais e pode ser considerado uma extensão do padrão literal mostrado acima. Mas os próximos dois padrões combinam um literal e uma variável, e a variável *vincula* um valor do assunto (`point`). O quarto padrão captura dois valores, o que o torna conceitualmente semelhante à atribuição de desempacotamento `(x, y) = point`.

Se você estiver usando classes para estruturar seus dados, você pode usar o nome da classe seguido por uma lista de argumentos semelhante a um construtor, mas com a capacidade de capturar atributos em variáveis:

```
class Point:
    x: int
    y: int

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

Você pode usar parâmetros posicionais com algumas classes embutidas que fornecem uma ordem para seus atributos (por exemplo, classes de dados). Você também pode definir uma posição específica para atributos em padrões configurando o atributo especial `__match_args__` em suas classes. Se for definido como (“x”, “y”), os seguintes padrões são todos equivalentes (e todos vinculam o atributo `y` à variável `var`):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

Uma maneira recomendada de ler padrões é vê-los como uma forma estendida do que você colocaria à esquerda de uma atribuição, para entender quais variáveis seriam definidas para quê. Apenas os nomes autônomos (como `var` acima) são atribuídos por uma instrução de correspondência. Nomes pontilhados (como `foo.bar`), nomes de atributos (o `x=` e `y=` acima) ou nomes

de classes (reconhecidos pelo "(…)" próximo a eles, como `Point` acima) nunca são atribuídos.

Os padrões podem ser aninhados arbitrariamente. Por exemplo, se tivermos uma pequena lista de pontos, poderíamos correspondê-la assim:

```
match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

Podemos adicionar uma cláusula `if` a um padrão, conhecido como "guarda". Se a guarda for falsa, `match` continua para tentar o próximo bloco de caso. Observe que a captura de valor ocorre antes que a guarda seja avaliada:

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Not on the diagonal")
```

Vários outros recursos importantes desta instrução:

- Assim como desempacotar atribuições, os padrões de tupla e lista têm exatamente o mesmo significado e realmente correspondem a sequências arbitrárias. Uma exceção importante é que eles não correspondem a iteradores ou strings.
- Os padrões de sequência têm suporte ao desempacotamento estendido: `[x, y, *rest]` e `(x, y, *rest)` funcionam de forma semelhante ao desempacotamento de atribuições. O nome depois de `*` também pode ser `_`, então `(x, y, *_)` corresponde a uma sequência de pelo menos dois itens sem ligar os itens restantes.
- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dictionary. Unlike sequence patterns, extra keys are ignored. An unpacking like `**rest` is also supported. (But `**_` would be redundant, so it is not allowed.)
- Subpadrões podem ser capturados usando a palavra reservada `as`:

```
case (Point(x1, y1), Point(x2, y2) as p2): ...
```

Vai capturar o segundo elemento da entrada como `p2` (se a entrada for uma sequência de dois pontos)

- A maioria dos literais são comparados por igualdade, no entando os singletons `True`, `False` e `None` são comparados por identidade.
- Padrões podem usar constantes nomeadas. Estas devem ser nomes pontilhados para prevenir que sejam interpretadas como variáveis de captura:

```
from enum import Enum
class Color(Enum):
    RED = 0
    GREEN = 1
    BLUE = 2

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```

Para uma explicação mais detalhada e exemplos adicionais, você pode olhar **PEP 636** que foi escrita em formato de tutorial.

ATIVIDADES

1. Faça um programa que peça uma nota, entre zero e dez. Mostre uma mensagem caso o valor seja inválido e continue pedindo até que o usuário informe um valor válido.
2. Faça um programa que leia 5 números e informe o maior número.
3. Faça um programa que imprima na tela apenas os números ímpares entre 1 e 50.
4. Numa eleição existem três candidatos. Faça um programa que peça o número total de eleitores. Peça para cada eleitor votar e ao final mostrar o número de votos de cada candidato.

4.7. Definindo funções

Podemos criar uma função que escreve a série de Fibonacci até um limite arbitrário:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

A palavra reservada `def` inicia a *definição* de uma função. Ela deve ser seguida do nome da função e da lista de parâmetros formais entre parênteses. Os comandos que formam o corpo da função começam na linha seguinte e devem ser indentados.

Opcionalmente, a primeira linha do corpo da função pode ser uma literal string, cujo propósito é documentar a função. Se presente, essa string chama-se *docstring*. (Há mais informação sobre docstrings na seção Strings de documentação.) Existem ferramentas que utilizam docstrings para produzir automaticamente documentação online ou para imprimir, ou ainda, permitir que o usuário navegue interativamente pelo código. É uma boa prática incluir sempre docstrings em suas funções, portanto, tente fazer disso um hábito.

A *execução* de uma função cria uma nova tabela de símbolos para as variáveis locais da função. Mais precisamente, todas as atribuições de variáveis numa função são armazenadas na tabela de símbolos local; referências a variáveis são buscadas primeiro na tabela de símbolos local, em seguida na tabela de símbolos locais de funções delimitadoras ou circundantes, depois na tabela de símbolos global e, finalmente, na tabela de nomes da própria linguagem. Embora possam ser referenciadas, variáveis globais e de funções externas não podem ter atribuições (a menos que seja utilizado o comando `global`, para variáveis globais, ou `nonlocal`, para variáveis de funções externas).

Os parâmetros reais (argumentos) de uma chamada de função são introduzidos na tabela de símbolos local da função no momento da chamada; portanto, argumentos são passados *por valor* (onde o *valor* é sempre uma *referência* para objeto, não o valor do objeto). 1 Quando uma função chama outra função, ou chama a si mesma recursivamente, uma nova tabela de símbolos é criada para tal chamada.

Uma definição de função associa o nome da função com o objeto função na tabela de símbolos atual. O interpretador reconhece o objeto apontado pelo nome como uma função definida pelo usuário. Outros nomes também podem apontar para o mesmo objeto função e também pode ser usados pra acessar a função:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Conhecendo outras linguagens, pode-se questionar que `fib` não é uma função, mas um procedimento, pois ela não devolve um valor. Na verdade, mesmo funções que não usam o comando `return` devolvem um valor, ainda que pouco interessante. Esse valor é chamado `None` (é um nome embutido). O interpretador interativo evita escrever `None` quando ele é o único resultado de uma expressão. Mas se quiser vê-lo pode usar a função `print()`:

```
>>> fib(0)
>>> print(fib(0))
None
```

É fácil escrever uma função que retorna uma lista de números da série de Fibonacci, ao invés de exibi-los:

```
>>> def fib2(n):  # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to
n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)     # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)     # call it
>>> f100                 # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Este exemplo demonstra novos recursos de Python:

- A instrução `return` finaliza a execução e retorna um valor da função. `return` sem qualquer expressão como argumento retorna `None`. Atingir o final da função também retorna `None`.

- A instrução `result.append(a)` chama um *método* do objeto lista `result`. Um método é uma função que 'pertence' a um objeto, e é chamada `obj.nomemetodo`, onde `obj` é um objeto qualquer (pode ser uma expressão), e `nomemetodo` é o nome de um método que foi definido pelo tipo do objeto. Tipos diferentes definem métodos diferentes. Métodos de diferentes tipos podem ter o mesmo nome sem ambiguidade. (É possível definir seus próprios tipos de objetos e métodos, utilizando *classes*, veja em Classes) O método `append()`, mostrado no exemplo é definido para objetos do tipo lista; adiciona um novo elemento ao final da lista. Neste exemplo, ele equivale a `result = result + [a]`, só que mais eficiente.

4.8. Mais sobre definição de funções

É possível definir funções com um número variável de argumentos. Existem três formas, que podem ser combinadas.

4.8.1. Argumentos com valor padrão

A mais útil das três é especificar um valor padrão para um ou mais argumentos. Isso cria uma função que pode ser invocada com menos argumentos do que os que foram definidos. Por exemplo:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

Essa função pode ser chamada de várias formas:

- fornecendo apenas o argumento obrigatório: `ask_ok('Do you really want to quit?')`
- fornecendo um dos argumentos opcionais: `ask_ok('OK to overwrite the file?', 2)`
- ou fornecendo todos os argumentos: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

Este exemplo também introduz a palavra-chave `in`, que verifica se uma sequência contém ou não um determinado valor.

Os valores padrões são avaliados no momento da definição da função, e no escopo em que a função foi *definida*, portanto:

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

irá exibir `5`.

Aviso importante: Valores padrões são avaliados apenas uma vez. Isso faz diferença quando o valor é um objeto mutável, como uma lista, dicionário, ou instâncias de classes. Por exemplo, a função a seguir acumula os argumentos passados, nas chamadas subsequentes:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

Isso exibirá:

```
[1]
[1, 2]
[1, 2, 3]
```

Se não quiser que o valor padrão seja compartilhado entre chamadas subsequentes, pode reescrever a função assim:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.8.2. Argumentos nomeados

Funções também podem ser chamadas usando argumentos nomeados da forma `chave=valor`. Por exemplo, a função a seguir:

```
def parrot(voltage, state='a stiff', action='voom',
type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

aceita um argumento obrigatório (`voltage`) e três argumentos opcionais (`state`, `action`, e `type`). Esta função pode ser chamada de qualquer uma dessas formas:

```
parrot(1000)                                          # 1
positional argument
parrot(voltage=1000)                                  # 1
keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2
keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2
keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3
positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1
positional, 1 keyword
```

mas todas as formas a seguir seriam inválidas:

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a
keyword argument
parrot(110, voltage=220)     # duplicate value for the same
argument
parrot(actor='John Cleese')  # unknown keyword argument
```

Em uma chamada de função, argumentos nomeados devem vir depois dos argumentos posicionais. Todos os argumentos nomeados passados devem corresponder com argumentos aceitos pela função (ex. `actor` não é um argumento nomeado válido para a função `parrot`), mas sua ordem é irrelevante. Isto também inclui argumentos obrigatórios

(ex.: `parrot(voltage=1000)` funciona). Nenhum argumento pode receber mais de um valor. Eis um exemplo que não funciona devido a esta restrição:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

Quando um último parâmetro formal no formato `**nome` está presente, ele recebe um dicionário (veja Tipo mapeamento — dict) contendo todos os argumentos nomeados, exceto aqueles que correspondem a um parâmetro formal. Isto pode ser combinado com um parâmetro formal no formato `*nome` (descrito na próxima subseção) que recebe uma tupla contendo os argumentos posicionais, além da lista de parâmetros formais. (`*nome` deve ocorrer antes de `**nome`.) Por exemplo, se definirmos uma função assim:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

Pode ser chamada assim:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

e, claro, exibiria:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Observe que a ordem em que os argumentos nomeados são exibidos é garantida para corresponder à ordem em que foram fornecidos na chamada da função.

4.8.3. Parâmetros especiais

Por padrão, argumentos podem ser passadas para uma função Python tanto por posição quanto explicitamente pelo nome. Para uma melhor legibilidade e desempenho, faz sentido restringir a maneira pelo qual argumentos possam ser passados, assim um desenvolvedor precisa apenas olhar para a definição da função para determinar se os itens são passados por posição, por posição e nome, ou por nome.

A definição de uma função pode parecer com:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        Positional or keyword   |
        |                                - Keyword only
         -- Positional only
```

onde `/` e `*` são opcionais. Se usados, esses símbolos indicam o tipo de parâmetro pelo qual os argumentos podem ser passados para as funções: somente-posicional, posicional-ou-nomeado, e somente-nomeado. Parâmetros nomeados são também conhecidos como parâmetros palavra-chave.

4.8.3.1. Argumentos posicional-ou-nomeados

Se `/` e `*` não estão presentes na definição da função, argumentos podem ser passados para uma função por posição ou por nome.

4.8.3.2. Parâmetros somente-posicionais

Olhando com um pouco mais de detalhes, é possível definir certos parâmetros como *somente-posicional*. Se *somente-posicional*, a ordem do parâmetro importa, e os parâmetros não podem ser passados por nome. Parâmetros somente-posicional são colocados antes de `/` (barra). A `/` é usada para logicamente separar os argumentos somente-posicional dos demais parâmetros. Se não existe uma `/` na definição da função, não existe parâmetros somente-posicionais.

Parâmetros após a `/` podem ser *posicional-ou-nomeado* ou *somente-nomeado*.

4.8.3.3. Argumentos somente-nomeados

Para definir parâmetros como *somente-nomeado*, indicando que o parâmetro deve ser passado por argumento nomeado, colocamos um `*` na lista de argumentos imediatamente antes do primeiro parâmetro *somente-nomeado*.

4.8.3.4. Exemplos de funções

Considere o seguinte exemplo de definição de função com atenção redobrada para os marcadores `/` e `*`:

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

A definição da primeira função, `standard_arg`, a forma mais familiar, não coloca nenhuma restrição para a chamada da função e argumentos podem ser passados por posição ou nome:

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

A segunda função `pos_only_arg` está restrita ao uso de parâmetros somente posicionais, uma vez que existe uma `/` na definição da função:

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got some positional-only arguments
passed as keyword arguments: 'arg'
```

A terceira função `kwd_only_args` permite somente argumentos nomeados como indicado pelo `*` na definição da função:

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: kwd_only_arg() takes 0 positional arguments but 1
was given

>>> kwd_only_arg(arg=3)
3
```

E a última usa as três convenções de chamada na mesma definição de função:

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but
3 were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError:  combined_example()  got  some  positional-only
arguments passed as keyword arguments: 'pos_only'
```

Finalmente, considere essa definição de função que possui uma potencial colisão entre o argumento posicional `name` e `**kwds` que possui `name` como uma chave:

```
def foo(name, **kwds):
    return 'name' in kwds
```

Não é possível essa chamada devolver `True`, uma vez que a chave `'name'` sempre será aplicada para o primeiro parâmetro. Por exemplo:

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
>>>
```

Mas usando `/` (somente argumentos posicionais), isso é possível já que permite `name` como um argumento posicional e `'name'` como uma chave nos argumentos nomeados:

```
def foo(name, /, **kwds):
    return 'name' in kwds
>>> foo(1, **{'name': 2})
True
```

Em outras palavras, o nome de parâmetros somente-posicional podem ser usados em `**kwds` sem ambiguidade.

4.8.3.5. Recapitulando

A situação irá determinar quais parâmetros usar na definição da função:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

Como guia:

- Use somente-posicional se você não quer que o nome do parâmetro esteja disponível para o usuário. Isso é útil quando nomes de parâmetros não tem um significado real, se você quer forçar a ordem dos argumentos da função quando ela é chamada ou se você precisa ter alguns parâmetros posicionais e alguns nomeados.
- Use somente-nomeado quando os nomes tem significado e a definição da função fica mais clara deixando esses nomes explícitos ou se você quer evitar que usuários confiem na posição dos argumentos que estão sendo passados.
- Para uma API, use somente-posicional para evitar quebras na mudança da API se os nomes dos parâmetros forem alterados no futuro.

4.8.4. Listas de argumentos arbitrárias

Finalmente, a opção menos usada é especificar que a função pode ser chamada com um número arbitrário de argumentos. Esses argumentos serão empacotados em uma tupla (ver Tuplas e Sequências). Antes dos argumentos em número variável, zero ou mais argumentos normais podem estar presentes.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Normalmente, esses argumentos `variádicos` estarão no final da lista de parâmetros formais, porque eles englobam todos os argumentos de entrada restantes, que são passados para a função. Quaisquer parâmetros formais que ocorrem após o parâmetro `*args` são argumentos ‘somente-nomeados’ , o que significa que eles só podem ser usados como chave-valor, em vez de argumentos posicionais:

```
>>> def concat(*args, sep="/"):
```

```
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

4.8.5. Desempacotando listas de argumentos

A situação inversa ocorre quando os argumentos já estão numa lista ou tupla mas ela precisa ser explodida para invocarmos uma função que requer argumentos posicionais separados. Por exemplo, a função `range()` espera argumentos separados, *start* e *stop*. Se os valores já estiverem juntos em uma lista ou tupla, escreva a chamada de função com o operador `*` para desempacotá-los da sequência:

```
>>> list(range(3, 6))            # normal call with separate
arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))           # call with arguments unpacked
from a list
[3, 4, 5]
```

Da mesma forma, dicionários podem produzir argumentos nomeados com o operador `**`:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end='
')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin'
demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts
through it. E's bleedin' demised !
```

4.8.6. Expressões lambda

Pequenas funções anônimas podem ser criadas com a palavra-chave `lambda`. Esta função retorna a soma de seus dois argumentos: `lambda a, b: a+b`. As funções lambda podem ser usadas sempre que objetos função forem

necessários. Eles são sintaticamente restritos a uma única expressão. Semanticamente, eles são apenas açúcar sintático para uma definição de função normal. Como definições de funções aninhadas, as funções lambda podem referenciar variáveis contidas no escopo:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

O exemplo acima usa uma expressão lambda para retornar uma função. Outro uso é passar uma pequena função como um argumento:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

ATIVIDADES

1. Faça um programa, com uma função que necessite de três argumentos, e que forneça a soma desses três argumentos.

2. Faça um programa que converta da notação de 24 horas para a notação de 12 horas. Por exemplo, o programa deve converter 14:25 em 2:25 P.M. A entrada é dada em dois inteiros. Deve haver pelo menos duas funções: uma para fazer a conversão e uma para a saída. Registre a informação A.M./P.M. como um valor ‘A’ para A.M. e ‘P’ para P.M. Assim, a função para efetuar as conversões terá um parâmetro formal para registrar se é

A.M. ou P.M. Inclua um loop que permita que o usuário repita esse cálculo para novos valores de entrada todas as vezes que desejar.

3. **Reverso do número.** Faça uma função que retorne o reverso de um número inteiro informado. Por exemplo: 127 -> 721.

4. **Data com mês por extenso**. Construa uma função que receba uma data no formato *DD/MM/AAAA* e devolva uma string no formato *D de mesPorExtenso de AAAA*. Opcionalmente, valide a data e retorne NULL caso a data seja inválida.

5. Faça uma função que informe a quantidade de dígitos de um determinado número inteiro informado.