Funções

Imagine que você fez um programinha para gerar estatísticas sobre vários dados dos funcionários: média dos salários, média de vendas, média de *feedback* positivo, média de *feedback* negativo, média de notas atribuídas pelos gestores... Você tem uma lista com os salários, uma lista com o total de vendas de cada funcionário, e assim sucessivamente. Então você fez o seguinte trecho de código:

```
soma = 0
for elemento in lista:
    soma = elemento
media = soma/len(lista)
```

Em seguida, você copiou e colou esse trecho de código várias vezes mudando "lista" pelo nome de cada lista individual, e "media" pelo nome do atributo. Trabalhoso, certo? Imagine que agora você percebeu o erro no trecho acima, e terá que sair corrigindo em todos os lugares onde colou o código errado. Imagina que conveniente se você pudesse arrumar apenas uma vez e todas as ocorrências fossem corrigidas automaticamente...

1. Funções

Uma função é um pedacinho de programa. Nós podemos dar um nome para nossa função, e toda vez que precisarmos que esse pedacinho de programa seja executado, nós o chamamos pelo nome.

Com isso, podemos evitar repetição de código, tornando nossos códigos mais enxutos e legíveis. Além disso, fica mais fácil corrigir problemas como o ilustrado no início deste capítulo.

1.1. Criando funções

1.1.1. Sintaxe básica

Em Python, podemos criar funções com o comando *def*, e em seguida damos um nome para nossa função.

```
def minha_primeira_funcao():
    print('Olá Mundo')
```

Se você executar o código acima, o que aparecerá na tela? Nada. Tudo que o código acima faz é **definir** *minha_primeira_funcao*, mas ela só será **executada** quando for chamada pelo nome.

```
# criando a função
def minha_primeira_funcao():
    print('Olá Mundo')

# o programa começa de verdade aqui:
minha_primeira_funcao() # chamada para a função
```

Quando **chamamos** uma função, a execução do programa principal é pausada, o fluxo de execução é desviado para a função, e ao final dela ele retornará para o ponto onde parou. Veja o exemplo abaixo:

```
# criando a função
def minha_primeira_funcao():
    print('Olá Mundo')

# o programa começa de verdade aqui:
print('aaa')
minha_primeira_funcao() # chamada para a função
print('bbb')
```

O resultado na tela será:

```
aaa
Olá Mundo
bbb
```

1.1.2. Escopo de uma variável

Se você criar uma variável dentro de uma função, dizemos que o escopo dessa variável é a função onde ela foi criada. Isso significa que essa variável existe apenas dentro da função e apenas durante sua execução. Se o seu programa principal possui uma variável chamada "nome" e uma função também possui uma variável chamada "nome", elas **não** são a mesma variável. Qualquer alteração feita nessa variável dentro da função não irá afetar de maneira alguma a variável externa.

1.1.3. Parâmetros de uma função

Nossas funções devem ser tão genéricas quanto possível se quisermos reaproveitá-las ao máximo.

Um dos pontos onde devemos tomar cuidado é na entrada de dados da função: se usarmos um *input* dentro da função, teremos uma função que resolverá um certo problema *desde que o usuário vá digitar os dados do problema*. Mas e se quisermos usar a função em um trecho do programa onde o usuário digita os dados e em outro ponto onde os dados são lidos de um arquivo?

Podemos resolver isso fazendo a leitura de dados no programa principal, fora de nossa função, e então **passaremos** os dados para a função. Dados passados para a função são chamados de *parâmetros* ou *argumentos* de uma função. Observe o exemplo abaixo:

```
def soma(a, b):
    resultado = a + b
    print(resultado)

soma(3, 2) # resultado na tela: 5
soma(4, 7) # resultado na tela: 11
x = 5
soma(10, x) # resultado na tela: 15
```

Quando colocamos "a" e "b" entre parênteses na criação da função, estamos especificando que a função recebe 2 parâmetros. O primeiro valor que for **passado** entre parênteses para nossa função será referenciado por "a" e o segundo será referenciado por "b". É como se "a" e "b" fossem variáveis que vão receber a cópia dos valores passados para a função. Note que podemos passar valores puros ou variáveis (como fizemos com "x" na última linha), e não precisamos criar variáveis "a" e "b" em nosso programa principal para "casar" com os parâmetros da função.

1.1.4. Retorno de uma função

Certas funções possuem uma "resposta": elas resolvem um problema (por exemplo, uma equação matemática) e nós estamos interessados no resultado. No exemplo anterior, tínhamos uma soma e nós imprimíamos a soma na tela.

Porém, ainda pensando na questão da função ser genérica: será que nós sempre queremos o resultado na tela? Imagine que você esteja utilizando a fórmula de Bháskara para resolver uma equação de segundo grau. No meio da fórmula existe uma raiz quadrada. Nós não queremos o resultado da raiz quadrada na tela, nós queremos o resultado dentro do nosso programa em uma variável para jogar em outra equação.

Bom, parece fácil: vamos tentar pegar o resultado fora da função... Certo?

```
def soma(a, b):
    resultado = a + b

media = resultado/2
print(media)
```

Se você executar o programa acima, verá uma mensagem de erro dizendo que "resultado" não existe. Toda variável criada dentro de uma função é **privada**. Ela só pode ser acessada dentro da função e será destruída ao final da execução da função. Para disponibilizar para o programa um valor que foi gerado dentro da função, utilizamos o comando *return*.

```
def soma(a, b):
    resultado = a + b
```

```
    return resultado


s = soma(10, 5)
media = s/2
print(media)
```

Quando fizemos `s = soma(10, 5)`, a função *soma* foi chamada, e ao final da execução, *s* recebeu o valor retornado por ela. Deste ponto em diante podemos utilizar a "resposta" da nossa função em nosso programa principal.

O *return*, além de disponibilizar um valor, **encerra** a execução da função. Se a sua função possuir outras linhas após o *return*, elas serão ignoradas.

> **Importante:** o *return* não torna a **variável** disponível fora da função, e sim o seu valor. Mesmo fazendo `return resultado`, se você tentar acessar `resultado` fora da função, você encontrará um erro. Imagine que a variável interna da sua função é uma caixinha, e o seu valor é o conteúdo da caixinha. Ao final da execução da função, a caixinha sempre será jogada fora, mas o seu conteúdo pode ser resgatado pelo programa principal ou por outra função, e eles ficam responsáveis por usar esse conteúdo e/ou armazená-lo em outra caixinha.

2. Recursividade

Uma função pode chamar outra função? Sim. Rode o programa abaixo e observe que ele funciona:

```
def soma(a, b):
    resultado = a + b
    return resultado

def media(x, y):
    s = soma(x, y)
    resultado = s/2
    return resultado

m = media(10, 5)
print(m)
```

Mas e se uma função referenciasse ela mesma? Isso também funciona, e chama-se **função recursiva**, ou **recursão**.

A ideia vem da matemática. Vejamos um exemplo. Considere a função fatorial. O fatorial de um número n qualquer é igual ao produto entre n e todos os seus antecessores inteiros positivos: n! = n x (n-1) x (n-2) x ... x 2 x 1.

Considere o fatorial de 5: 5! = 5x**4x3x2x1**

Pense agora no fatorial de 4: 4! = 4x3x2x1

Note que temos destacado em negrito a expressão completa do fatorial de 4 dentro do fatorial de 5. Então é possível reescrecer o fatorial de 5 em função do fatorial de 4:

5! = 5x(4!)

Porém, dentro do fatorial de 4, temos o fatorial de 3, e assim sucessivamente. Podemos generalizar da seguinte maneira:

f(n) =

1, se n = 1
n * f(n-1), se x > 1

Ou seja, imagine que você queira calcular f(4). Como 4 > 1, teremos: f(4) = 4 * f(3)

Precisamos expandir f(3): f(4) = 4 * (3 * f(2))

E assim sucessivamente: f(4) = 4 * (3 * (2 * f(1)))

Opa, f(1) nós conhecemos: está definido lá em cima como 1. Portanto: f(4) = 4 * 3 * 2 * 1 f(4) = 24

Note que nós decompomos um problema em várias instâncias "menores" do problema. Quebramos a formulação de uma multiplicação enorme por vários casos de n x f(n-1). Chamamos essa estratégia de *dividir para conquistar*, e ela envolve identificar 2 etapas bastante claras do problema:

Caso base: é um caso para o qual temos um valor conhecido (no exemplo acima, f(1) = 1)
Caso geral: é a chamada recursiva, onde faremos referência à própria função.

Note também que esse comportamento tem o comportamento de *pilha*: se colocamos 3 pratos empilhados sobre a mesa, precisamos tirar primeiro o último que colocamos, certo? Caso contrário, a pilha toda tomba. No caso da recursão, para obter f(4) caímos em f(3), depois f(2), depois f(1), depois f(0) e foi para ele que obtivemos a primeira resposta, que em seguida usamos para calcular f(1), depois calcular f(2), depois f(3) e só então chegamos em f(4). O primeiro passo do problema foi o último a ser resolvido.

Em Python, nossa função ficaria assim:

```
def fatorial(n):
    if n == 1:
        return 1
    else:
        return n * fatorial (n-1)
```

Se chamarmos `fatorial(4)`, o que acontecerá? O programa começará a executar a função, cairá no *else* e encontrará a função chamada novamente. Neste caso, ele salva x valendo 4 e salva que a execução foi interrompida nessa linha. Então ele cria um **novo** x valendo 3, cai novamente no *else* e salva que a execução foi interrompida nessa linha, e assim sucessivamente.

Note que para cada passo recursivo, as variáveis da função são copiadas e também é salvo o ponto onde a execução parou. Ou seja, funções recursivas podem consumir **bastante** memória, além de tempo de processamento para ficar criando cópias. A vantagem delas é

o rigor matemático: podemos transcrever funções matemáticas quase exatamente como elas são, sem criar *loops* e variáveis para ficar guardando estados.

3. Documentando funções

3.1. *Type hints*

O Python utiliza uma abordagem para tipos conhecida como *duck typing*, que é baseada no ditado "*if it walks like a duck and it quacks like a duck, it's a duck*", que pode ser traduzido como "*se ele anda como um pato e grasna como um pato, ele é um pato*".

Isso significa que o Python não se importa muito com o tipo de variáveis. Nossa função soma recebe dois parâmetros e aplica o operador "+" entre eles. A gente pode até ter pensado em número ao criar essa função, mas o operador + também funciona entre 2 strings. Ou seja, se alguém chamar `soma('olá', 'mundo')`, a função irá funcionar.

Nem sempre esse comportamento é desejável. Muitas vezes bolamos nossa função com tipos específicos em mente, e a possibilidade de outros tipos serem passados cria o risco da função não executar corretamente, ou de retornar algum tipo de dado que pode causar problemas na integração com outros sistemas.

Para evitar esse tipo de problema existe o conceito de *type hint*, onde nós podemos deixar anotado em nossas funções o tipo esperado de cada parâmetro e do retorno. Utilizaremos dois pontos entre o nome do parâmetro e o tipo esperado, e uma setinha após os parênteses para indicar o tipo de retorno:

```
def soma(a:int, b:int) -> int:
    resultado = a + b
    return resultado
```

Note que **ainda** é possível passar *float*, *str* ou outros tipos para a função, e nesses casos ela também retornará outros tipos. Porém, as *type hints* são uma espécie de anotação, e sempre que um programador começar a digitar uma chamada para essa função, as IDEs mais modernas irão destacar para o programador quais tipos ele **deveria** passar e quais tipos ele **deveria** esperar como retorno.

É possível também especificar que uma função não retorna nada:

```
def ola_mundo() -> None:
    print('Olá mundo!')
```

Quando tentamos pegar o retorno de uma função que não retorna nada (ex: `x = ola_mundo()`), o programa não dará erro. Ele apenas irá armazenar na variável a constante *None*, que é um valor especial em Python que representa justamente a ausência de valor. Uma variável contendo *None* é uma variável que existe mas não possui valor atribuído. Dessa maneira, ao anotarmos que nossa função "retorna *None*", estamos na prática documentando que ela não apresenta retorno.

> Experimente pegar o retorno da função `print`: execute algo como `x = print('olá')` e tente visualizar o valor de `x`.

Caso você queira que algum parâmetro ou o retorno seja uma coleção (ex: uma lista), basta utilizar o tipo correspondente à coleção. Porém, não é possível especificar que deve ser uma lista de strings ou uma lista de int, por exemplo, mas simplesmente uma lista.

```
def somatorio(numeros:list) -> int:
    soma = 0
    for n in numeros:
        soma += n
    return soma
```

A partir da versão **3.10** do Python é possível utilizar o operador *pipe* (a barra vertical: |) com o efeito de "ou" para indicar que mais de um tipo é permitido. A função abaixo recebe uma lista e promete que pode retornar int ou float:

```
def somatorio(numeros:list) -> int | float:
    soma = 0
    for n in numeros:
        soma += n
    return soma
```

3.2. *Docstrings*

Além das *type hints*, podemos também escrever comentários especiais explicando o que as nossas funções fazem. IDEs modernas são capazes de identificar esses comentários e exibi-los em um popup na tela para o programador que irá usar a função. Esses comentários são chamados de *docstrings*.

Para criar uma *docstring*, a primeira linha da sua função deve ser uma string envolta em 3 aspas simples ou duplas:

```
def somatorio(numeros:list) -> int | float:
'''Recebe uma lista de números e retorna a soma de todos eles'''
    soma = 0
    for n in numeros:
        soma += n
    return soma
```

Experimente colar a função acima em uma IDE de sua preferência (como o VS Code) e comece a digitar uma chamada para ela. Ele irá automaticamente mostrar para você toda a documentação: parâmetros (com tipo), retorno e a breve descrição que digitamos.