Capítulo 13

Definindo funções

Na Seção 8.5 introduzimos as funções que fazem parte da biblioteca Python. Essas funções facilitam muito nossa vida e economizam muito nosso trabalho. Imagine se, a cada vez que precisássemos computar o seno de um ângulo, tivéssemos que utilizar a série de Taylor, como vimos em um exercício anterior. De qualquer forma, o valor do seno que utilizamos quando invocamos a função `math.sin(x)` é calculado por um trecho de programa que alguém implementou.

A Figura 13.1 mostra o que acontece quando chamamos uma função e ela nos retorna um valor, no caso o seno do número passado como parâmetro. Na Figura 13.1(A), nosso programa é executado até o ponto em que a função é chamada. A execução do nosso programa é interrompida momentaneamente e o interpretador passa a executar o código da função `sin`, que está na biblioteca Python (Figura (B)). Esse programa vai utilizar o valor que passamos como parâmetro para computar o seno, utilizando algum método como a série de Taylor, ou qualquer outro método que, na verdade, não nos interessa. Terminada a função, a execução volta para o nosso programa, devolvendo também o resultado computado pela função (Figura (C)). Nosso programa pode, então, prosseguir a execução .

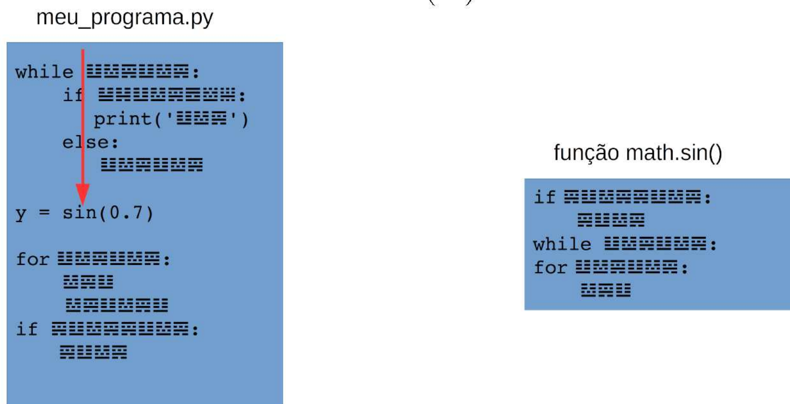
13.1 Definindo nossas próprias funções

Nós também podemos definir funções dentro dos nossos programas Python. Vamos usar como exemplo o nosso programa que calcula as estatísticas sobre um conjunto de dados. Se olharmos o programa, vemos que temos um código longo, e, talvez, difícil de se compreender. Isso acontece porque temos muitas coisas sendo feitas juntas, uma após a outra.

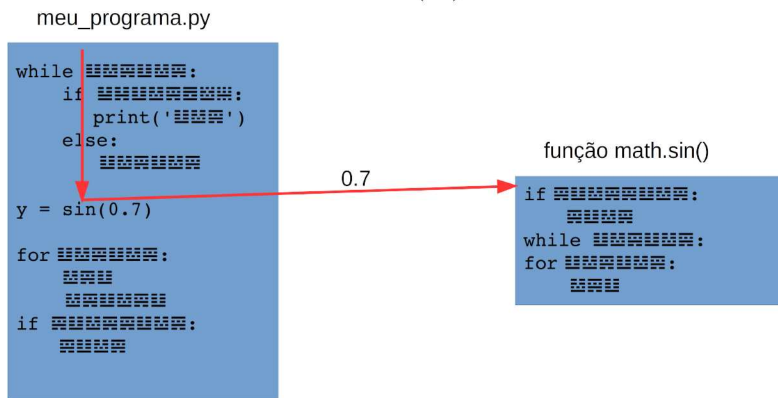
Vamos, então, tentar dividir as coisas. Vamos supor que temos diferentes funções para:

- ler os dados e armazená-los em uma lista;
- computar a média;
- computar a variância;
- computar mediana; e

(A)



(B)



(C)

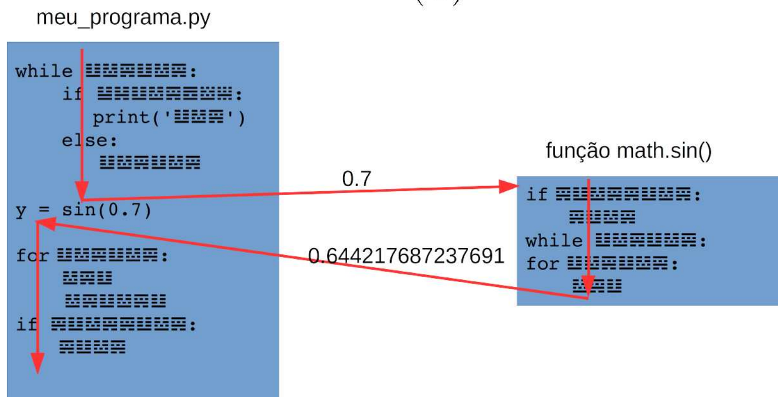


Figura 13.1: Mecanismo de chamada de uma função

- computar a moda.

Se tivermos essas funções disponíveis, podemos reescrever nosso programa

como mostrado a seguir. Percebe-se que o programa ficou muito mais claro e mais elegante.

Programa 13.1 Programa que calcula as estatísticas, usando funções

```
import math

x = le_dados()
print('Valor de média: {:.4f}'.format(media(x)))
print('Valor de variância: {:.4f}'.format(\
                                        variancia(x)))
print('Valor do desvio padrão: {:.4f}'.format(\
                            math.sqrt(variancia(x))))
print('Valor de mediana: {:.4f}'.format(mediana(x)))
print('Valor de moda: {}'.format(moda(x)))
```

Precisamos, então, implementar aquele código que estava no nosso programa original na forma de funções. Vamos começar pela função que lê os dados de entrada. Para fazê-lo, utilizamos o seguinte comando:

```
def le_dados():
```

Com esse comando dizemos que os comandos a seguir fazem parte da definição da função que batizamos de `le_dados`. Os nomes de função devem seguir as mesmas restrições que vimos para nomes de variáveis. Os parênteses que seguem o nome indicam que essa função não utiliza nenhum parâmetro, ou seja, não precisa que nosso programa forneça nenhum dado para que ela faça o que deve fazer.

Vejamos, então, os comandos que executaremos dentro da função `le_dados`. Repare a indentação dos comandos. Todos eles devem estar pelo menos uma indentação além do comando `def`.

Programa 13.2 Função para ler um conjunto de dados e armazená-lo em uma lista

```
1  def le_dados():
2      dados = []
3      r = 0
4      i = 1
5      while r >= 0:
6          msg = 'Digite o valor {}:'.format(i)
7          r = float(input(msg))
8          if r < 0:
9              print('Entrada de dados terminou')
10         elif r > 100:
11             print('Valor deve estar entre 0 e 100')
12         else:
13             dados.append(r)
14             i += 1
15     return dados
```

O significado de cada um dos comandos não é mais novidade. Vamos, então, nos concentrar em alguns pontos que são importantes. O primeiro é que "o que acontece na função, fica na função". Ou seja, todas as variáveis utilizadas na função não fazem sentido fora dela. Por exemplo, se tentarmos utilizar as variáveis `dados` ou `r` ou `i` lá no nosso Programa 13.1 obteremos uma mensagem de erro do interpretador. Por isso são consideradas "locais".

Mas se não podemos usar os dados processados na função, para que ela serve? Existe uma forma de transferir dados que foram obtidos ou calculados dentro da função para o código que a chamou. Isso é feito com o comando `return` na linha 15 da nossa função. Esse comando está dizendo que o resultado da chamada da função é o conteúdo da variável `dados`, que é uma lista. Então, ao executarmos o comando da linha 3 do Programa 13.1, estamos atribuindo à variável `x` esse resultado. Da mesma forma que podemos fazer `y = sin(0.7)`, podemos fazer `x = le_dados()`.

A diferença entre chamar a função do seno e a `le_dados()`, é que para o seno precisamos dizer qual é o ângulo sobre o qual queremos calcular o seno. Significa que precisamos passar um parâmetro para essa função, o que não acontece com a função `le_dados()`, que não necessita de parâmetros. Já a função `media`, que chamamos na linha 4 do Programa 13.1, necessita de um parâmetro, que é a lista de valores sobre a qual desejamos calcular a média. Então, na definição da função `media` precisamos indicar que ela aceita um parâmetro.

Programa 13.3 Função para computar a média de valores armazenados em uma lista

```
1 def media(dados):
2     soma = 0.0
3     for r in dados:
4         soma += r
5     return soma / len(dados)
```

Isso é feito colocando-se entre os parênteses, após o nome da função, uma lista de parâmetros que devem ser usados para chamar-se a função. Se tivermos mais do que um parâmetro, os seus nomes aparecem separados por vírgula. O nome dos parâmetros são tratados como variáveis locais da função.

A função para o cálculo da variância é mostrada a seguir. Não temos muitas novidades a destacar, a não ser o fato que essa função chama a função `media`. Isso é perfeitamente válido, ou seja, ter chamada de uma função dentro de outra função.

Programa 13.4 Função para computar a variância de valores armazenados em uma lista

```
1 def variancia(dados):
2     m = media(dados)
3     var = 0.0
4     for r in dados:
5         var += (r - m) ** 2
6     return var / (len(dados) -1)
```

Podemos melhorar um pouco essa função, deixando o nosso código mais elegante. Nas linhas 5 e 7 do Programa 13.1, chamamos duas vezes a função `variancia`. Na segunda, extraímos o raiz quadrada do resultado para computar o desvio padrão. Usando uma tupla, podemos fazer com que a função `variancia` retorne mais do que um resultado. No caso, a variância e o desvio padrão. A nova versão da função é mostrada a seguir.

Programa 13.5 Função para computar a variância e o desvio padrão de valores armazenados em uma lista

```
1 def variancia(dados):
2     m = media(dados)
3     var = 0.0
4     for r in dados:
5         var += (r - m) ** 2
6     var /= (len(dados) -1)
7     dp = math.sqrt(var)
8     return  (var,dp)
```

E a parte “principal” do nosso programa é mostrada a seguir. Note que temos uma tupla, com duas variáveis que irá receber o retorno da função `variancia`: a variável `v` recebe o primeiro elemento da tupla retornada, que é a variância, e a variável `dp` recebe o segundo, que é o desvio padrão.

Programa 13.6 Chamada de função que retorna uma tupla

```
x = le_dados()
print('Valor de média: {:.4f}'.format(media(x)))
(v,dp) = variancia(x)
print('Valor de variância: {:.4f}'.format(v))
print('Valor do desvio padrão: {:.4f}'.format(dp))
print('Valor de mediana: {:.4f}'.format(mediana(x)))
print('Valor de moda: {}'.format(moda(x)))
```

No Programa 13.6, a última linha mostra uma lista ou uma tupla, já que podemos ter mais do que um valor como moda. Por isso, não é possível utilizar o mesmo comando `format` usado nos comandos anteriores. Para melhorar um pouco isso, podemos criar uma função `print_moda`, que mostra os resultados da moda formatados da maneira que desejamos. Note, no programa a seguir, que essa função não retorna nenhum valor. Ela simplesmente faz o processamento necessário e sua execução termina com um comando `return`, sem nenhum valor associado a ele.

Programa 13.7 Função sem retorno de valor

```
def mostra_moda(moda):
    if len(moda) == 1:
        s = 'O valor da moda é: '
    else:
        s = 'Os valores da moda são: '
    for x in moda:
        s += '{:.4f} '.format(x)
    print(s)
    return
```

13.1.1 Exercícios

1. Complete o programa das estatísticas, implementando as demais funções.
2. Escreva uma função que recebe três parâmetros que representam os lados de um triângulo. O programa deve retornar: 0 se os valores não correspondem a um triângulo; 1 se correspondem a um triângulo equilátero; 2 se correspondem a um triângulo isósceles; 3 se correspondem a um triângulo escaleno.

3. Faça um programa com duas funções (uma para homens e uma para mulheres) que computam o peso ideal, baseado na altura de uma pessoa:
    - para homem: (72.7 x h) – 58
    - para mulher: (62.1 x h) – 44.7
4. Escreva uma função que recebe como parâmetro o valor do salário de um trabalhador e calcula o valor do imposto de renda a ser recolhido na fonte.
5. Escreva uma função que verifica se um determinado ano é ou não bissexto. A função deve retornar o valor False se o ano não for bissexto e True se ele for bissexto.
6. Implemente as funções de seno, cosseno e tangente, usando a série de Taylor, vista anteriormente.
7. Escreva uma função que transforma qualquer ângulo dado em radianos em um ângulo em graus, no intervalo [0 – 360).
8. Utilize funções para simplificar o programa que implementa o método de Gauss (Programas 12.10 a 12.13) para resolução de sistemas de equações.

13.2 Recursão

Quando falamos de uma linguagem de programação, o termo "recursão" refere-se à possibilidade de uma função fazer chamadas a ela mesma. Em termos mais gerais, isso significa que para resolver um determinado problema usamos a solução de um caso mais simples do mesmo problema.

Um exemplo típico é o cálculo do fatorial. Podemos definir $n!$ como:

- a multiplicação de todos os inteiros de 1 até n; ou
- $n \times (n-1)!$. E $0! = 1$

A primeira definição é iterativa. A segunda é recursiva pois para definir o fatorial de um número n, usamos a definição do fatorial de $n-1$. Temos, também, que definir um caso base, que é o caso mais simples, e que serve para dar fim à recursão. Nesse caso, precisamos definir qual o valor de 1!. Dessa forma, se quisermos saber qual o valor de 4!, teremos:

$$4! = 4\times 3! = 4\times 3\times 2! = 4\times 3\times 2\times 1! = 4\times 3\times 2\times 1\times 0! = 4\times 3\times 2\times 1\times 1 = 24$$

Se formos traduzir essa ideia em uma função Python, teremos o trecho de programa a seguir. Na execução dessa função, o interpretador vai criando várias chamadas da função `fatorial`, cada uma com um valor menor do parâmetro `n`, até que esse chegue a 0. Quando isso acontece, é possível calcular o valor para todas as chamadas, com valores maiores. Quer, dizer, tendo computado `fatorial(0)`, que retorna o resultado 1, é possível computar `fatorial(1)` que também retorna o valor 1, e assim por diante.

Programa 13.8 Função fatorial recursiva

```
1 def fatorial(n):
2     if n == 0:
3         return 1;
4     return n * fatorial(n-1)
```

Um segundo exemplo de função que pode ser definida recursivamente é o Máximo Divisor Comum entre dois números inteiros. Sabe-se que para computar essa função, valem as seguintes propriedades:

- $MDC(x, x) = x$;
- $MDC(x, y) = MDC(x - y, y)$, se $x > y$;
- $MDC(x, y) = MDC(y, x)$.

A primeira propriedade espelha o caso base, ou seja, é onde devemos chegar para terminar a recursão. A segunda, mostra como calculamos o valor do MDC, baseado em um caso mais simples. E o terceiro nos ajuda no caso em que $x < y$. O programa a seguir mostra como implementar essa função.

Programa 13.9 Função MDC recursiva

```
1  def mdc(x,y):
2      if x == y:
3          return x
4      if x > y:
5          return mdc(x-y, y)
6      return mdc(y,x)
7
8  x = int(input('Entre com o 1o. valor: '))
9  y = int(input('Entre com o 2o. valor: '))
10
11 print('O valor do MDC é: {}'.format(mdc(x,y)))
```

Muitas vezes, a definição de uma função recursiva acaba sendo bem mais simples e mais elegante do que a versão iterativa. Porém, devemos ser cuidadosos na sua utilização. Cada chamada de função em Python vai gastar uma certa quantidade de memória. Se tivermos muita chamadas de funções simultâneas, todas "abertas" ao mesmo tempo, é possível que tenhamos um erro apontado pelo interpretador. Por exemplo, se executarmos o programa do MDC com os valores 1234567 e 890, obtemos o resultado mostrado a seguir.

```
> python mdc.py
Entre com o 1o. valor: 1234567
Entre com o 2o. valor: 890
Traceback (most recent call last):
  File "mdc.py", line 13, in <module>
    print('O valor do MDC é: '.format(mdc(x,y)))
  File "mdc.py", line 6, in mdc
    return mdc(x-y, y)
  [Previous line repeated 994 more times]
  File "mdc.py", line 3, in mdc
    if x == y:
RecursionError: maximum recursion depth exceeded in
comparison
```

13.2.1 Exercícios

Resolva os problemas abaixo, sempre utilizando funções recursivas.

1. Crie uma função recursiva que receba um número inteiro N e calcule a soma dos números de 1 até N.

2. Escreva uma função recursiva para somar os elementos de uma lista de números. Ou seja, a função recebe uma lista como parâmetro e retorna um número, que é a soma dos elementos da lista.

3. Escreva uma função recursiva para encontrar o maior valor de uma lista de números.

4. Implemente uma função `inverte` que recebe uma lista como parâmetro e inverte a ordem dos seus elementos.

5. Para calcular o resto da divisão inteira entre dois números (função MOD), pode-se usar a seguinte definição:

   - $MOD(x, x) = 0$;
   - $MOD(x, y) = x$, se $x < y$;
   - $MOD(x, y) = MOD(x - y, y)$, se $x > y$.
     Escreva uma função recursiva para calcular a função MOD.

6. Escreva uma função recursiva que determine quantas vezes um dígito K ocorre em um número natural N. Por exemplo, o dígito 2 ocorre 3 vezes em 762021192.

7. Escreva a função recursiva que une duas listas, sem elementos repetidos, ordenadas de forma crescente, considerando que as duas listas não têm elementos em comum. Ou seja, o resultado da função deve ser uma lista ordenada de forma crescente que contém todos os elementos das listas originais.

8. Escreva a função recursiva que une duas listas, sem elementos repetidos, ordenadas de forma crescente, considerando que as duas listas podem ter elementos em comum e que a lista resultante pode ter elementos repetidos. Ou seja, o resultado da função deve ser uma lista ordenada de forma crescente que contém todos os elementos das listas originais.

9. Escreva a função recursiva que une duas listas, sem elementos repetidos, ordenadas de forma crescente, considerando que as duas listas podem ter elementos em comum e que a lista resultante não pode ter elementos repetidos. Ou seja, o resultado da função deve ser uma lista ordenada de forma crescente que contém todos os elementos das listas originais, sem repetições.

13.3 Aplicação: o método de Cramer

Como vimos anteriormente, uma das formas para resolver um sistema de equações, que não seja muito grande, é o método de Cramer. Para aplicar esse método, é essencial computar o determinante de algumas matrizes. Por isso, vamos definir uma função, recursiva, que calcula o determinante pelo método de Laplace, como mostrado na primeira parte deste livro.

No Programa 13.10 vemos essa função, que recebe um único parâmetro, a matriz da qual se deseja o determinante. Se a matriz tem ordem um, ou seja, é composta de um único elemento, então o valor desse elemento é o determinante. Essa condição e verificada nas linhas 2 e 3 da função.

Caso contrário, iremos calcular e somar os cofatores dos elementos da primeira coluna, ou no caso da nossa lista, a coluna de número 0. Isso é feito nas linhas 6 a 14. Note que estamos trabalhando em cima da coluna 0 ou da linha 0, dependendo de como estamos representando a nossa matriz, conforme apresentado na Seção 12.5. Como veremos adiante, nesse programa, escolhemos a representação menos convencional, que é a de uma coluna por posição da nossa lista.

Programa 13.10 Função para calcular o determinante

```
def determinante(M):
    if len(M) == 1:
        return M[0][0]

    det = 0.0
    for i in range(len(M)):
        #faz uma copia da matriz
        #sem coluna 0 e sem linha i
        M2 = copia_matriz(M,0,i)
        if i % 2 == 0:
            sig = 1
        else:
            sig = -1
        d = determinante(M2)
        det += M[0][i] * sig * d

    return det
```

A função `copia_matriz` retorna uma matriz `M2` igual à matriz `M`, mas sem a coluna 0 e sem a linha `i`, ou seja, com a ordem uma unidade inferior. Sobre essa nova matriz chamamos a mesma função `determinante`. Note que assim vamos diminuindo a ordem das matriz que necessitamos computar o determinante até chegar à ordem um. Como mencionado anteriormente, o problema é que para matrizes grandes, o número de matrizes torna-se absurdamente grande, fazendo com que torne-se impraticável aplicar este método. Na função, a variável `sig` indica qual o sinal do cofator que devemos usar na soma dos cofatores (1 ou -1). Como usamos sempre a coluna 0, esse valor depende apenas do valor de `i`.

A função `copia_matriz` é apresentada a seguir. Ela é relativamente simples. Inicialmente, faz uma cópia exatamente igual do parâmetro `M`. Em seguida, percorre todas as posições da nova lista `M2`, que também são listas, eliminando os elementos na posição `j` dessas listas. Ou seja, elimina a linha `j` da matriz. Em seguida elimina o elemento na posição `i` da matriz `M2`, eliminando, portanto, toda a linha `i`. Essa função requer que nosso programa contenha um `import copy` para o uso da função `deepcopy`.

Programa 13.11 Função que elimina uma linha e uma coluna da matriz

```
def copia_matriz(M, i, j):
    M2 = copy.deepcopy(M)
    for l in M2:
        del l[j]
    del M2[i]
    return M2
```

Finalmente, falta implementar o método de Cramer, propriamente dito.

Para isso, inicialmente computamos o determinante da matriz de coeficientes e armazenamos na variável `detM`, no Programa 13.12. Em seguida, substituímos cada coluna dessa mesma matriz, pela última coluna da matriz estendida, que representa os valores à direita da igualdade. Para cada uma dessas substituições, computamos o determinante da matriz obtida e armazenada na variável `A2` (linha 11). Finalmente, dividimos esses determinantes pelo determinante da matriz de coeficientes, obtendo os valores desejados das incógnitas.

Programa 13.12 Programa que implementa o método de Cramer

```
1  #cada posição da lista representa uma COLUNA da matriz
2  A = [[6,5,-2,3], [0,-2,3,1], [0,3,2,1], [2,3,-11,0],
3  [6,5,4,3]]
4  detM = determinante(A[:-1])
5  if detM == 0:
6      print('Sistema sem solução.')
7      sys.exit()
8  b = A[-1]
9  r = []
10 for i in range(len(A)-1):
11     A2 = A[:-1]
12     A2[i] = b
13     det = determinante(A2)
14     r.append(det/detM)
15 print(r)
```

Um ponto importante a ressaltar é que na representação escolhida, cada elemento das listas `A` e `A2` representam uma coluna da matriz. Essa representação não é a mais habitual, como já mencionamos, mas facilita a substituição de cada uma das colunas da matriz de coeficientes, na linha 11 do Programa 13.12.