Universidade Federal do Ceará
Departamento de Estatística e Matemática Aplicada
Fundamentos de Programação (CK0087), período 2020.2
Professor: Tibérius O. Bonates (`tb@ufc.br`).

Funções

1 Introdução

A maioria das linguagens de programação modernas (tais como Python, C++, Java, etc) fornecem ao programador a capacidade de organizar código em grupos de instruções chamados de *funções*. Uma função nada mais é do que uma sequência de instruções da linguagem que recebe um nome escolhido pelo usuário. Por meio deste nome, o código da função pode ser *chamado* (isto é, executado) a partir de diferentes partes do programa.

Já utilizamos várias funções do Python a esta altura, tais como a função `len`, que, ao ser utilizada, fornece o número de elementos em alguma coleção (um conjunto, uma lista, etc). O próprio `print` é um exemplo de função, que realiza a tarefa de imprimir certo conteúdo na tela. As funções `len` e `print` são chamadas de funções *nativas* do Python. Isto significa que elas estão disponíveis para uso em qualquer programa, sem haver a necessidade de nenhuma ação particular antes de serem utilizadas. Nesta parte do curso, vamos aprender a criar nossas próprias funções.

1.1 Sintaxe

O seguinte código em Python mostra a definição de uma função que calcula a média de dois valores que são fornecidos como parâmetros (ou argumentos):

```
def media(a,b):
    return (a + b)/2
```

A instrução `def` é utilizada para comunicar ao Python que estamos definindo uma nova função. Perceba o caractere de dois pontos (":") ao final da linha "`def media(a,b):`". Observe também a indentação da linha que vem logo abaixo do `def`. Estas características indicam que a definição de uma função segue um esquema similar àquele de um `if` ou um `for`: tudo aquilo que é parte da função fica indentado nas linhas abaixo dela. E o fim da indentação marca o fim da função.

Após o uso de `def`, deve ser especificado o nome da função. Este nome deve seguir as mesmas regras para criação de nomes de variáveis. Além destas partes, temos também a lista de parâmetros da função, que aparecem entre parênteses, imediatamente após o nome da função.

1.2 Retorno

Além estes elementos, vemos neste exemplo a instrução `return`, que não havíamos utilizado ainda. Ela informa o valor que a função assume.

Fazendo um paralelo com funções matemáticas, uma função em Python tipicamente vai calcular algum valor com base em um ou mais valores. A instrução `return` faz com que a chamada feita à função seja substituída pelo valor retornado. Isto é semelhante ao que acontece quando escrevemos $f(x)$ em uma expressão matemática para representar um valor calculado a partir do argumento x, de acordo com uma regra ou fórmula. A função `media` mostrada neste exemplo é capaz de calcular, com base em dois valores numéricos fornecidos (os *argumentos* da função), um valor que é a média aritmética dos dois argumentos. Este valor calculado e retornado é chamado de *retorno* da função.

É importante notar que a instrução `return` encerra a execução da função: a função é concluída imediatamente e a chamada feita à função é substituída pelo valor de retorno.

1.3 Argumentos (ou parâmetros)

Os argumentos `a` e `b` da função `media` são normalmente chamados de *parâmetros*. Eles são variáveis normais do Python e permitem que o código escrito seja genérico – isto é, ele pode ser utilizado para calcular a média aritmética entre quaisquer dois números. Vamos ver um exemplo.

```
def media(a,b):
    return (a + b)/2

x = float(input('Digite um número: '))
y = float(input('Digite outro número: '))
m = media(x,y)

print('A média de {} e {} é {}.'.format(x, y, m))
```

Neste exemplo, perceba que, além da função `media`, temos um código adicional, que lê dois números reais a partir do teclado, chama a função `media` e imprime uma frase na tela. As instruções adicionais, que não fazem parte da definição da função `media` são o que comumente chamamos de *programa principal*. O objetivo do programa principal costuma ser o de controlar o funcionamento geral do nosso programa, enquanto as funções são responsáveis por tarefas mais específicas.

Em nosso exemplo, chamamos a função `media` com os parâmetros `x` e `y`. Quando isso acontece, os parâmetros `a` e `b` que aparecem na definição da função recebem cópias dos valores que as variáveis `x` e `y` possuiam no ponto em que a chamada da função acontece, isto é, na linha `m = media(x,y)`. Isto significa que o código da função será executado com o valor de `a` igual ao valor de `x` e com o valor de `b` igual ao valor de `y`.

A principal vantagem de se escrever uma função é que o código dela é escrito uma única vez, mas pode ser utilizado várias vezes. Na prática, isso significa que nosso programa pode fazer várias chamadas à função `media`, podendo utilizar diferentes parâmetros em cada chamada. Ou seja, podemos realizar chamadas tais como `media(x,y)`, `media(7,4)`, `media(2,x)`, etc, em um mesmo programa. O que acontece na prática é que cada execução da função `media` é independente de qualquer outra chamada feita a ela. Cada vez que `media` é chamada, novas variáveis `a` e `b` são criadas na memória e recebem cópias dos argumentos fornecidos. Ao final desta execução da função, as variáveis `a` e `b`, que só existem dentro da função, deixam de existir.

Exercício 1.1 Vamos escrever uma função que recebe dois valores numéricos como parâmetros e retorna o maior deles. Para isso, vamos escolher um nome informativo para a função. Que tal o nome `maximo`? Assim, fica muito claro qual tarefa a função realiza.

Além do nome, precisamos dizer que a função recebe dois parâmetros. Estes parâmetros também precisam de nomes. No código a seguir, estamos usando os nomes `x` e `y`. O código da função é bem direto: se `x` for maior ou igual a `y`, a função retorna o valor de `x`; caso contrário (`else`), a função retorna o valor de `y`.

```
def maximo(x, y):
    if x >= y:
        return x
    else:
        return y
```

Esta é apenas uma das maneiras de escrever uma função que realiza esta tarefa. Mai adiante, veremos outras maneiras de escrever funções diferentes, mas que realizam exatamente a mesma tarefa. ▮

Exercício 1.2 E se desejarmos encontrar o máximo de três valores numéricos? Será que a função fica muito diferente?

Novamente, existe mais de um jeito de escrever uma função deste tipo. Uma das maneiras é adaptar um pouco a ideia que utilizamos no exercício anterior. Podemos chamar a função de `maximoDe3`. Precisamos informar ao Python que a função receberá três parâmetros, que podemos chamar de `x`, `y` e `z`.

```
def maximoDe3(x, y, z):
    if x >= y and x >= z:
        return x
    elif y >= x and y >= z:
        return y
    else:
        return z
```

Neste código, testamos se `x` é maior ou igual a `y` e também maior ou igual a `z`. Se o teste for verdadeiro, a função deve retornar o valor de `x`. Se o teste for falso, então já sabemos que o maior dos três valores é `y` ou `z` (o `x` já está fora da jogada). Se `y` for maior ou igual a `x` e maior ou igual a `z`, então a função deve retornar `y`. Por fim, se ambos os testes forem falsos, então sabemos que o maior dos três valores é seguramente `z`. Nesse ponto do código, podemos simplesmente retornar `z` sem precisar realizar nenhum teste adicional. ▮

1.4 Ausência de parâmetros ou de retorno

Diferentemente do que costuma acontecer quando usamos funções matemáticas, o Python permite definir funções sem parâmetros e também funções que não fornecem um valor de retorno. Vamos ver separadamente cada uma das duas situações.

O seguinte código mostra uma função que pergunta à usuária seu nome e retorna o nome lido. Perceba que a função não recebe nenhum parâmetro, afinal ela não precisa de nenhum valor externo para realizar sua tarefa. Isso fica claro quando olhamos para a linha "`def ler_nome():`" e percebemos os parênteses sem nenhum argumento entre eles.

```
def ler_nome():
    nome = input('Por favor, digite seu nome: ')
    return nome

nome_usuaria = ler_nome()
print('Olá, {}. Sejam bem-vinda(o).'.format(nome_usuaria))
```

Note que a chamada de uma função sempre requer o uso de parênteses, mesmo que a função não receba parâmetros. Isto fica claro quando olhamos para a linha "`nome_usuaria = ler_nome()`".

Um exemplo de função que não retorna nenhum valor é mostrado a seguir. A função `saudar` recebe um nome e imprime na tela uma saudação à usuária, mas não utiliza a instrução `return` para retornar nenhum valor. De fato, seu funcionamento não produz nenhum tipo de resultado que possa ser retornado.

```
def ler_nome():
    nome = input('Por favor, digite seu nome: ')
    return nome

def saudar(nome):
    print('Olá, {}. Sejam bem-vinda(o).'.format(nome))

nome_usuaria = ler_nome()
saudar(nome_usuaria)
```

É importante observar que o programa principal ficou menor quando incluimos a função "`saudar`". Esta observação reforça a ideia de que funções tendem a simplificar o código e deixá-lo mais legível. O programa principal agora se resume a duas linhas de código, que podem ser facilmente interpretadas pela pessoa que lê o programa.

1.5 Modificando o valor de um parâmetro

Como vimos, os parâmetros recebem cópias dos valores que são fornecidos na chamada da função. Por causa deste fato, se o código da função alterar o valor de um parâmetro, isso não causará nenhuma alteração nas variáveis usadas na chamada da função. Vamos ver um exemplo para esclarecer melhor essa afirmação.

Exemplo 1.3 O código na figura 1 contém uma função que retorna o maior de dois valores fornecidos como parâmetros. Vamos entender o funcionamento da função `maximo`. Se o valor do parâmetro `x` for maior ou igual ao valor do parâmetro `y`, então a função retorna o valor de `x`. Caso contrário, a função faz com que `x` receba o valor armazenado em `y` e, depois, retorna este valor. Os valores das variáveis `a` e `b` no programa principal nos permitem afirmar que o valor retornado será o do parâmetro `y`, pois $y > x$.

Figura 1: Exemplo de modificação do valor de um parâmetro.

É interessante ver que, após a execução da função, os valores das variáveis `a` e `b` no programa principal não foram alteraods, apesar da instrução `x = y` na linha 3. Isto pode ser visto ao execu-

tarmos o programa e percebermos que os comandos `print(a)` e `print(b)` imprimem os valores 9 e 15, nesta ordem. ▮

Este mesmo comportamento acontece não apenas quando estamos lidando com parâmetros do tipo `int` (inteiro), mas também com parâmetros dos tipos `float`, `str` e `bool`. Quando lidamos com parâmetros mais complexos (`list`, `set`, `dict`), o Python funciona de maneira diferente: mudanças feitas nestes parâmetros são refletidas nas variáveis fornecidas no momento da chamada da função. O exemplo 1.4 demonstra este comportamento.

Exemplo 1.4 A função `removerZeros` foi projetada para receber como parâmetro uma lista. Enquanto a lista contiver algum valor igual a zero, a função remove um destes valores iguais a zero. Ao final da execução da função, a lista terá todo seu conteúdo original, exceto pelos valores iguais a zero que existiam nela.

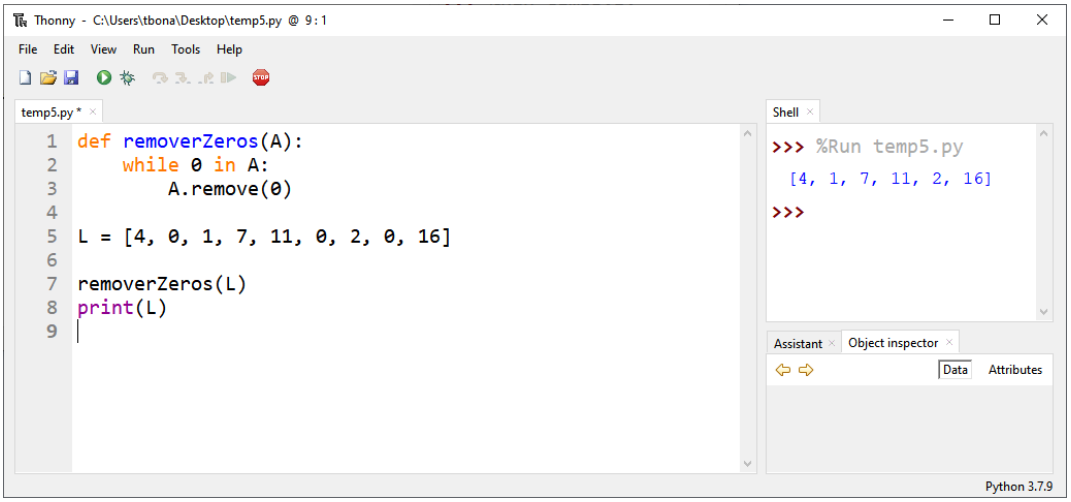

Figura 2: Exemplo de modificação do valor de um parâmetro do tipo `list`.

Ao final do programa principal, o conteúdo da lista `L` é impresso na tela, o que deixa claro que s lista foi realmente modificada: todos os zeros foram removidos dela. ▮

1.6 Funções como subprogramas

Cada uma das funções que vimos funciona como um pequeno programa (por esta razão, funções também são chamadas de *subprogramas* ou *subrotinas*). O significado disso é: da mesma forma que eu posso executar um programa várias vezes, talvez em momentos diferentes do dia e possivelmente utilizando dados diferentes a cada execução, um programa pode executar várias vezes alguma função, também em diferentes momentos e possivelmente utilizando dados diferentes a cada execução.

É possível, inclusive, copiar a função `media` para outro programa e utilizá-la sem haver qualquer necessidade de modificação na definição da função. Trechos de código que são executados mais de uma vez em um mesmo programa são candidatos naturais a se tornarem uma ou mais funções. Em vez de escrevermos mais de uma vez o mesmo trecho de código, podemos criar uma função e simplesmente chamá-la sempre que necessário.

Uma consequência do uso de uma mesma função em vários pontos do código é que, se a função contiver um erro, este erro só precisará ser corrigido uma vez. O outro lado da moeda também é verdade: se cometermos um erro na escrita de uma função que é chamada várias vezes, o erro acontecerá em vários pontos do programa. Portanto, aqui vale a máxima: "com grandes poderes vêm grandes responsabilidades".

Exercício 1.5 Será que você consegue escrever uma versão mais simples da função `maximoDe3`, desenvolvida no exercício 1.2? Uma sugestão é fazer uso da função `maximo` que retorna o maior de dois valores numéricos.

Antes de mais nada, vamos nos convencer que a seguinte propriedade é verdadeira: o máximo de três valores numéricos, x, y e z, pode ser calculado por meio do uso da função `maximo`, que retorna o maior de dois valores numéricos. De fato, se calcularmos o máximo de x e y (vamos chamar este máximo de t), então podemos calcular o máximo entre t e z. Este valor final é o máximo dos três valores.

Com isso em mente, podemos escrever o seguinte programa:

```
def maximo(x, y):
    if x >= y:
        return x
    else:
        return y

def maximoDe3(x, y, z):
    t = maximo(x, y)
    return maximo(t, z)

a = 6
b = 9
c = 2
print(maximoDe3(a,b,c))
```

Observe que a função `maximoDe3` primeor calcula o máximo entre `x` e `y`. Em seguida, ela calcula o valor entre este máximo e `z`. Ao final da execução do programa, o valor impresso na tela será 9. Este exemplo ilustra o uso oportunista de uma função que já estava pronta (`maximo`) para facilitar a escrita de uma nova função (`maximoDe3`). ∎

1.7 Escopo

Quando falamos sobre parâmetros, dissemos que eles são variáveis que existem somente dentro da função onde aparecem. Essa é uma ideia importante, que tem muito a ver com o que chamamos de *escopo*.

Nós dizemos que o *escopo de uma função* é todo o código da função, desde a linha que contém o `def` até a última linha de código que pertence à função. A importância de falar sobre o escopo de uma função é que isso nos permite entender a região do programa onde as variáveis da função existem e são acessíveis (isto é, podem ter seus valores lidos ou modificados).

Por exemplo, quando criamos uma variável dentro de uma função, a variável só existe desde o momento em que ela é utilizada pela primeira vez (o momento de sua criação), até o final do escopo

da função. Esta região do código é o *escopo da variável*.

Algo semelhante vale para os parâmetros da função: eles existem desde o momento em que a função começa a executar (a chamada da função) até o momento em que ela termina. Ou seja, o escopo dos parâmetros coincide com o escopo da função. A figura 3 ilustra estes conceitos.

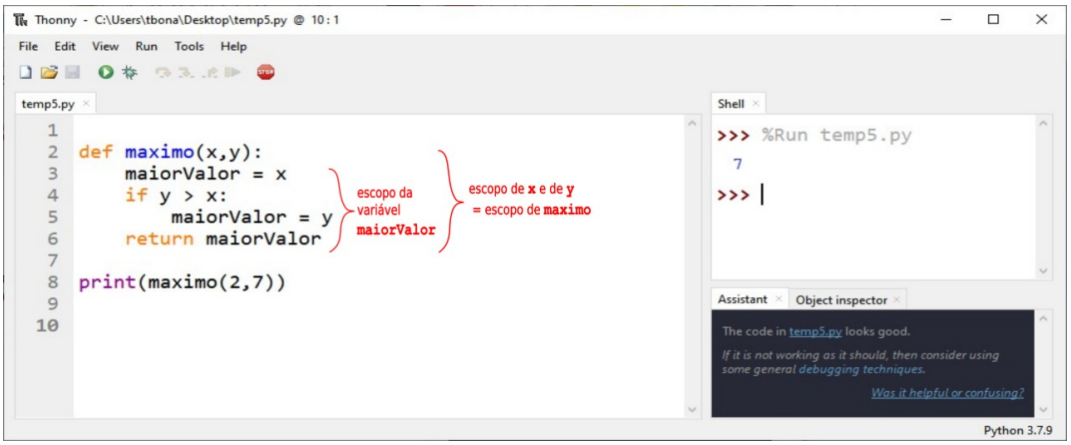

Figura 3: Escopo das variáveis de uma função.

Um parâmetro ou uma variável de uma função só pode ser acessado(a) a partir de uma linha de código que esteja dentro de seu escopo. Isto quer dizer que, por exemplo, os parâmetros ou variáveis de uma função não podem ser acessados a partir de outras funções, nem a partir do programa principal. Se tentarmos escrever `print(maiorValor)` na linha 9 do programa da figura 3, teremos um erro, pois esta instrução estaria fora do escopo da variável `maiorValor`.

O programa principal tem seu próprio escopo também. As variáveis definidas no programa principal existem desde o primeiro momento em que são utilizadas até o final do programa. Isto significa que uma variável não pode ser utilizada antes de ser criada. Por outro lado, o escopo do programa principal é especial e pode ser acessado a partir de qualquer função. Dessa forma, todas as variáveis do programa principal podem ser acessadas a partir de qualquer função, desde que a chamada da função aconteça dentro do escopo da variável – isto é, depois que a variável já foi criada. A figura 4 ilustra esta situação.

Como a chamada da função `saudacao` na linha 6 está dentro do escopo da variável `nome`, a função é capaz de acessar o valor da variável. Se a chamada acontecesse na linha 4 do código, ela seria inválida, pois estaria fora do escopo da variável `nome`.

Esta forma de organizar o acesso aos dados do programa complementa bem o que falamos sobre parâmetros e valores de retorno. Como cada variável está acessível apenas dentro de seu escopo, utilizamos os parâmetros e os valores de retorno para enviar dados de uma parte do programa para outra.

1.8 Variáveis com o mesmo nome

O Python permite que duas variáveis tenham o mesmo nome, desde que estejam em escopos independentes. Por exemplo, é possível que duas funções tenham parâmetros com o mesmo nome, sem

Figura 4: Escopo das variáveis do programa principal.

que haja nenhum conflito. O código a seguir mostra duas funções que possuem parâmetros com os mesmos nomes.

```
def maximo(x, y):
    maiorValor = x
    if y > x:
    maiorValor = y
    return maiorValor

def media(x, y):
    return (x + y)/2
```

A partir do contexto, o Python sabe a qual variável `x` estamos nos referindo cada vez que `x` aparece no código das duas funções. Esta é uma boa notícia, já que não precisamos inventar um nome diferente para cada parâmetro de nossas funções!

Uma situação parecida, mas diferente, é quando uma função utiliza um nome de variável que existe no programa principal. Como o escopo do programa principal é acessível a partir do código das funções, o Python precisa cuidar para saber se estamos nos referindo à variável do programa principal ou a uma variável criada dentro da função. Vamos ver dois exemplos.

Exemplo 1.6 No código a seguir, temos uma função chamada `bomDia`, que recebe um parâmetro `nome` e imprime uma mensagem educada. Para isso, a função utiliza uma variável chamada `mensagem`.

```
def bomDia(nome):
    mensagem = 'Bom dia, ' + nome
    print(mensagem)

mensagem = 'Digite seu nome, por gentileza:'
usuario = input(mensagem)
bomDia(usuario)
```

O programa principal também possui uma variável chamada `mensagem`. Isso é um problema? A resposta é não. Como vimos antes, o programa principal não consegue acessar a variável `mensagem`

que existe na função. Mas, a função `bomDia` é capaz de acessar a variável do programa principal. O que acontece neste exemplo é que a instrução "`mensagem = 'Bom dia, ' + nome`" é interpretada pelo Python como sendo a criação de uma variável dentro da função. A partir deste ponto, toda vez que usamos `mensagem` dentro dafunção, o Python entende que estamos falando variável que foi criada no escopo da função. ▮

Exemplo 1.7 No código abaixo, temos uma situação um pouco diferente: a função `boaTarde` apenas utiliza o valor da variável `usuario` do programa principal, isto é, ela consulta o valor armazenado na variável. Algo diferente acontece nesse exemplo: o Python compreende que não existe uma variável `usuario` criada dentro da função, e que o programador deseja acessar o valor da variável que existe no problema principal.

```
def boaTarde():
    mensagem = 'Boa tarde, ' + usuario
    print(mensagem)

mensagem = 'Digite seu nome, por gentileza:'
usuario = input(mensagem)
boaTarde()
```

Dessa forma, o programa imprime a frase "`Bom dia, João`" ou "`Bom dia, Ana`", dependendo do nome que for fornecido por meio do teclado. ▮