Capítulo 4

Funções

4.1 Chamadas de Função

No contexto da programação, uma *função* é a denominação dada a um conjunto de instruções que realiza uma computação. Quando você define uma função, você especifica o nome e a sequência de instruções. Posteriormente, você poderá "chamar" a função pelo seu nome. Nós já haviamos visto um exemplo de uma *chamada de função*:

```
>>> type(32)
<class 'int'>
```

O nome da função é `type`. A expressão em parênteses é chamada de *argumento* da função. O argumento é um valor ou variável que estamos passando para a função como entrada. O resultado, para a função `type`, é o tipo do argumento.

É comum dizer que uma função "recebe" um argumento e "retorna" um resultado. O resultado é chamado de *valor retornado*.

4.2 Funções internas

O Python fornece várias funções internas importantes que nós podemos usar sem precisar definí-las. Os criadores do Python escreveram um conjunto de funções para solucionar problemas comuns e incluíram elas na linguagem para nosso uso.

A funções `max` e `min` nos dão o maior e o menor valores de uma lista, respectivamente:

```
>>> max('Alô mundo')
'ô'
>>> min('Alô mundo')
' '
>>>
```

A função `max` nos diz qual é o "maior caractere" da string (que acaba sendo a letra "ô") e a função `min` nos mostra o menor caractere (que acaba sendo o espaço).

Outra função interna muito comum é a `len`, que nos diz quantos itens há em seu argumento. Caso este argumento seja uma string, a função retorna o número de caracteres que a compõem.

```
>>> len('Alô mundo')
9
>>>
```

Essas funções não são limitadas à análise apenas de strings. Elas podem operar em qualquer conjunto de valores, como veremos em capítulos posteriores.

Você deve encarar os nomes das funções internas como palavras reservadas (por exemplo, evite usar "max" como nome de uma variável).

4.3 Funções de conversão de tipo

O Python também fornece funções internas que convertem valores de um tipo para outro. A função `int` recebe qualquer valor e converte ele em um inteiro, se for possível, ou reclama, caso contrário:

```
>>> int('32')
32
>>> int('Olá')
ValueError: invalid literal for int() with base 10: 'Hello'
```

`int` pode converter valores em pontos-flutuante em inteiros, mas não os arredonda; ela trunca a parte decimal:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` converte inteiros e strings em números em ponto-flutuante:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finalmente, `str` converte seu argumento para uma string:

```
{.python}
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

4.4 Funções matemáticas

Python tem um módulo `math` que possui a maioria das funções matemáticas familiares. Antes que possamos utilizar o módulo, nós temos que importá-lo:

```
>>> import math
```

Essa declaração cria um *objeto do módulo* chamado math. Se você usar print() nele, você receberá algumas informações sobre ele:

```
>>> print(math)
<module 'math' (built-in)>
```

O objeto do módulo contém as funções e variáveis definidas no módulo. Para acessar uma dessas funções, você deve especificar o nome do módulo e o nome da função, separados por um ponto. Esse formato é conhecido como *notação de ponto*.

```
>>> razao = potencia_do_sinal / intensidade_do_ruido
>>> decibeis = 10 * math.log10(razao)
>>> radianos = 0.7
>>> modulo = math.sin(radianos)
```

O primeiro exemplo computa o logaritmo de base 10 da razão sinal-ruído. O módulo *math* também fornece uma função chamada `log` que computa logaritmos de base e.

O segundo exemplo encontra o seno de `radianos`. O nome da variável é uma

dica que `sin` e outras funções trigonométricas (`cos`, `tan`, etc.) recebem argumentos em radianos. Para converter de graus para radianos, dividimos por 360 e multiplicamos por 2π:

```
>>> graus = 45
>>> radianos = graus / 360.0 * 2 * math.pi
>>> math.sin(radianoss)
0.7071067811865476
```

A expressão `math.pi` pega a variável `pi` do módulo *math*. O valor dessa variável é uma aproximação de π, com precisão de 15 dígitos.

Se você sabe trigonometria, pode checar o resultado anterior comparando-o com a raiz quadrada de 2 dividida por 2:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

4.5 Números Aleatórios

Dadas as mesmas entradas, a maior parte dos programas geram sempre as mesmas saídas, de forma que eles são ditos *determinísticos*. Determinismo é geralmente uma boa coisa, já que nós esperamos que o mesmo cálculo obtenha sempre o mesmo resultado. Para algumas aplicações, porém, nós queremos que o computador seja imprevisível. Jogos são um exemplo óbvio, mas existem outros.

Criar um programa totalmente não determinístico não é tão fácil, mas há formas de que eles pareçam ser não determinísticos. Uma delas é utilizar *algoritmos* que geram *números pseudoaleatórios*. Números pseudoaleatórios não são verdadeiramente aleatórios, pois eles são gerados por má computação determinística, mas apenas olhando para os números, é impossível distinguí-los de números aleatórios.

O módulo `random` oferece funções que geram números pseudoaleatórios (os quais eu passarei simplesmente a chamar de "aleatórios" a partir daqui).

A função `random` retorna um número float entre 0.0 e 1.0 (incluindo 0.0, mas não 1.0). Todas as vezes que você chamar a função `random`, você receberá o próximo número de uma série longa. Para ver uma amostra, rode esse *loop*:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

Este programa produz a seguinte lista de 10 números aleatórios entre 0.0 e até, mas não incluindo, 1.0.

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

Exercício 1: Rode o programa anterior no seu computador para ver quais números aparecem. Rode o programa mais uma vez e veja quais números aparecem.

A função `random` é apenas uma das funções que utilizam números aleatórios. A função `randint` recebe os parâmetros 'menor' e 'maior' e retorna um inteiro entre 'menor' e 'maior' (incluindo ambos).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Para escolher um elemento de uma sequência aleatoriamente você pode utilizar a função `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

O módulo `random` também disponibiliza funções para gerar valores aleatórios de acordo com distribuições contínuas, incluindo gaussiana, exponencial, gamma e outras.

4.6 Adicionando novas funções

Até agora, nós temos utilizado apenas as funções que já vêm com o Python, mas também é possível adicionar novas funções. Uma *definição de função* especifica o nome de uma nova função e a sequência de comandos que serão executados quando ela for chamada. Uma vez que nós definimos uma função, podemos reutilizá-la repetidas vezes ao longo do nosso programa:

Veja aqui um exemplo:

```
def print_LetraDaMusica():
    print("Eu sou um lenhador, e eu estou bem.")
    print('Eu durmo a noite toda e trabalho o dia todo.')
```

`def` é uma palavra-chave que indica que isso é uma definição de função. O nome da função é `print_LetraDeMusica`. As regras para nomes de funções são as mesmas que para nomes de variáveis: letras, números e alguns sinais de pontuação são permitidos, mas o primeiro caractere não pode ser um número. Você não pode usar uma palavra-chave como nome de função, e deve evitar ter uma variável e uma função com nomes iguais.

Os parênteses vazios após o nome indicam que essa função não aceita argumentos. Mais tarde nós construiremos funções que levam argumentos como suas entradas.

A primeira linha da definição da função é chamada de *cabeçalho*; o restante é o *corpo*. O cabeçalho tem que encerrar com dois pontos e o corpo tem que estar identado. Por convenção, a identação é sempre de quatro espaços. O corpo pode conter qualquer número de comandos.

Se você fizer uma definição de função no modo interativo, o interpretador usa reticências (. . .) para te informar que a definição não está completa:

```
>>> def print_LetraDeMusica():
...     print("Eu sou um lenhador, e eu estou bem.")
...     print('Eu durmo a noite toda e trabalho o dia todo.')
...
```

Para finalizar a função, você precisa inserir uma linha em branco (isso não é necessário no modo comum).

Definir uma função cria uma variável com o mesmo nome.

```
>>> print(print_LetraDeMusica)
<function print_ LetraDeMusica at 0xb7e99e9c>
>>> print(type(print_ LetraDeMusica))
<class 'function'>
```

O valor de `print_LetraDeMusica` é um *objeto de função*, o qual é do tipo "*function*".

A sintaxe para chamar a nova função é a mesma que para funções internas:

```
>>> print_LetraDeMusica()
Eu sou um lenhador, e eu estou bem.
Eu durmo a noite toda e trabalho o dia todo.
```

Uma vez que sua função está definida, você pode usá-la dentro de outra função. Por exemplo, para repetir o refrão acima, nós poderíamos escrever uma função chamada `repetir_LetraDeMusica`:

```
def repetir_LetraDeMusica():
    print_ LetraDeMusica()
    print_ LetraDeMusica()
```

E depois chamar `repetir_LetraDeMusica`:

```
>>> repetir_LetraDeMusica()
Eu sou um lenhador, e eu estou bem.
Eu durmo a noite toda e trabalho o dia todo.
Eu sou um lenhador, e eu estou bem.
Eu durmo a noite toda e trabalho o dia todo.
```

Mas não é bem assim que a música continua.

4.7 Definições e usos

Colocando junto os fragmentos de código da seção anterior, o programa inteiro fica assim:

```
>>> def print_LetraDeMusica():
...     print("Eu sou um lenhador, e eu estou bem.")
...     print('Eu durmo a noite toda e trabalho o dia todo.')
...
```

```
def repetir_LetraDeMusica():
    print_ LetraDeMusica()
    print_ LetraDeMusica()

>>> repetir_LetraDeMusica()

def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')


def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()

# Code: http://www.py4e.com/code3/lyrics.py
```

Esse programa contém duas definições de função: `print_lyrics` e `repeat_lyrics`. Definições de função são executadas assim como outras declarações, mas o efeito é criar objetos de função. Os comandos dentro da função não são executados até que a função seja chamada, e a definição de função não gera saída.

Como você deve imaginar, será necessário criar a função antes de poder executá-la. Em outras palavras, a definição de função será executada antes da primeira vez que seja chamada.

Exercício 2: Mova a última linha desse programa para o início, assim a chamada da função aparece antes das definições. Execute o programa e veja qual mensagem de erro aparece.

Exercício 3: Mova a chamada da função de volta ao final e mova a definição de `print_lyrics` para depois da definição de `repeat_lyrics`. O que acontece quando você executa esse programa?

4.8 Fluxo de Execução

Para garantir que a função é definida antes de seu primeiro uso, você precisa saber a ordem que cada declaração é executada, o que é chamado de *fluxo de execução*.

A execução sempre começa na primeira declaração do programa. Declarações são executadas uma por vez, na ordem de cima para baixo.

Definições de função não alteram o fluxo da execução do programa, mas lembrem-se de que comandos dentro da função não são executados até que a função seja chamada.

Uma chamada de função é como um desvio no fluxo da execução. Em vez de ir para a próxima declaração, o fluxo pula para o corpo da função, executa todas os comandos ali, e então volta para onde havia parado.

Isso parece bastante simples, até você lembrar que uma função pode chamar outra. Enquanto estiver no meio de uma função, o programa pode ter que executar as instruções em outra função. Então, enquanto estiver executando a nova função, o programa pode ter que executar ainda outra função!

Felizmente, Python é bom em manter controle de onde está, então cada vez que uma função é concluída o programa continua de onde parou na função que o chamou. Quando chega ao fim do programa, se encerra.

Qual é a moral desse conto sórdido? Quando você lê um programa, você nem sempre quer lê-lo de cima para baixo. Às vezes faz mais sentido se você seguir o fluxo de execução.

4.9 Parâmetros e argumentos

Algumas das funções internas que vimos requerem argumentos. Por exemplo, quando você chama `math.sin` você passa um número como argumento. Algumas funções requerem mais de um argumento: `math.pow` recebe dois, a base e o expoente.

Dentro da função, os argumentos são atribuídos à variáveis chamadas *parâmetros*. Aqui está um exemplo de uma função definida pelo usuário que recebe um argumento:

```
def mostra_duas_vezes(bruce):
    print(bruce)
    print(bruce)
```

Essa função atribui o argumento à um parâmetro nomeado `bruce`. Quando a função é chamada, ela mostra o valor do parâmetro (qualquer que seja ele) duas vezes.

Essa função funciona com qualquer valor que possa ser mostrado.

```
>>> mostra_duas_vezes('Spam')
Spam
Spam
>>> mostra_duas_vezes(17)
17
17
>>> import math
>>> mostra_duas_vezes(math.pi)
3.141592653589793
3.141592653589793
```

As mesmas regras de composição que se aplicam às funções internas também se aplicam à funções definidas pelo usuário, então podemos usar qualquer tipo de expressão como argumento para `mostra_duas_vezes`:

```
>>> mostra_duas_vezs('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> mostra_duas_vezes(math.cos(math.pi))
-1.0
-1.0
```

O argumento é avaliado antes da função ser chamada, então nos exemplos as expressões `'Spam '*4` e `math.cos(math.pi)` são avaliadas apenas uma vez.

Você também pode usar variáveis como argumento:

```
>>> michael = 'Eric, a meia abelha.'
>>> mostra_duas_vezes(michael)
Eric, a meia abelha.
Eric, a meia abelha.
```

O nome da variável que passamos como um argumento (`michael`) não tem nada a ver com o nome do parâmetro (`bruce`). Não importa como o valor é chamado em casa (na chamada da função); aqui em `mostra_duas_vezes`, chamamos todo mundo de `bruce`.

4.10 Funções frutíferas e funções vazias

Algumas das funções que estamos usando, como as funções matemáticas, produzem resultados; por falta de um nome melhor, eu os chamo *funções frutíferas*. Outras funções, como `mostra_duas_vezes`, executam uma ação mas não retornam um valor. Elas são chamadas *funções vazias*.

Quando você chama uma função frutífera, você quase sempre quer fazer alguma coisa com resultado; por exemplo, você pode atribuí-la a uma variável ou usá-la como parte de uma expressão:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

Quando você chama uma função no modo interativo, o Python exibe o resultado:

```
>>> math.sqrt(5)
2.23606797749979
```

Mas em um `script`, se você chama uma função frutífera e não armazena o resultado da função em uma variável, o valor de retorno desaparece na névoa!

```
math.sqrt(5)
```

Esse `script` calcula a raiz quadrada de 5, mas como não se armazena o resultado em uma variável ou exibe o resultado, isso não é muito útil.

Funções vazias podem exibir algo na tela ou ter algum outro efeito, mas elas não têm um valor de retorno. Se você tentar atribuir o resultado a uma variável, você obtém um valor especial chamado `None`.

```
>>> resultado = mostra_duas_vezes('Bing')
Bing
Bing
>>> print(resultado)
None
```

O valor `None` não é o mesmo que uma `string` `"None"`. Ele é um valor especial que possui seu próprio tipo:

```
>>> print(type(None))
<class 'NoneType'>
```

Para retornar um resultado de uma função, nós usamos o comando `return` em nossa função. Por exemplo, nós podemos fazer uma função muito simples chamada `soma_dois` que soma dois números juntos e retorna um resultado.

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

>>> x = soma_dois(3, 5)
>>> print(x)
```

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)

# Code: http://www.py4e.com/code3/addtwo.py
```

Quando esse `script` é executado, a declaração `print` mostrará "8" porque a função `soma_dois` foi chamada com 3 e 5 como argumentos. Dentro de uma função, os parâmetros `a` e `b` eram 3 e 5, respectivamente. A função calculou a soma dos dois números e atribuiu à variável local da função nomeada `soma`. Então foi usado o comando `return` para enviar o valor calculado de volta para o código de chamada como resultado da função, o qual foi atribuído à variável `x` e mostrada em tela.

4.11 Por que funções?

Pode não parecer claro o motivo de dividir o programa em funções. Existem varias razões:

- Criar uma nova função lhe dar a oportunidade de nomear um grupo de instruções, o que torna seu programa mais fácil de ler, de entender e de depurar.
- As funções podem tornar um programa menor, eliminando as repetições. Mais tarde, se você fizer uma mudança, só precisará mudar em um lugar.
- Dividindo um programa em funções permite que você depure as partes uma de cada vez e, em seguida, montá-las em um só trabalho.
- Funções bem projetadas são muitas vezes úteis para muitos programas. Depois de escrever e depurar um, você pode reutilizá-lo.

Ao longo do resto do livro, muitas vezes usaremos uma definição de função para explicar um conceito. A habilidade de criar e usar funções é ter uma função que capture a ideia corretamente, como "encontrar o menor valor em uma lista de valores". Mais tarde vamos lhe mostrar o código que encontra o menor valor em uma lista de valores e vamos apresentá-lo para você como uma função chamada `min` que leva uma lista de valores como seu argumento e retorna o menor valor na lista.

4.12 Depuração

Se você estiver usando um editor de texto para escrever seus `scripts`, você pode encontrar problemas com espaços e tabulações (tabs). A melhor maneira de evitar esses problemas é usar espaços exclusivamente (sem tabs). A maioria dos editores de texto que sabem sobre Python fazem isso por padrão, mas alguns não.

Tabulações e espaços são geralmente invisíveis, o que os torna difíceis de depurar, então tente encontrar um editor que gerencia a identação para você.

Além disso, não se esqueça de salvar seu programa antes de executá-lo. Alguns ambientes de desenvolvimento fazem isso automaticamente, mas alguns não. Nesse caso, o programa que você está olhando no editor de texto não é o mesmo que o programa que você está executando.

A depuração pode levar um longo tempo se você estiver executando o programa errado repetidamente!

Certifique-se de que o código que você está olhando é o código que você está executando. Se você não tiver certeza, coloque algo como `print ("Olá")` no início do programa e execute-o novamente. Se você não vê 'Olá', você não está executando o programa certo!

4.13 Glossário

algoritmo Um processo geral para resolver uma categoria de problemas. algoritmo

argumento Um valor fornecido para uma função quando a função é chamada. Esse valor é atribuído ao parâmetro correspondente na função. argumento

cabeçalho A primeira linha de uma definição de função. cabeçalho

chamada de função Uma declaração que executa uma função. Consiste na função nome seguido por uma lista de argumentos. chamada de função

composição Utilizar uma expressão como parte de uma expressão maior ou uma instrução como parte de uma declaração maior. composição

corpo A sequência de instruções dentro de uma definição de função. corpo

declaração de importação Uma instrução que lê um arquivo do módulo e cria um objeto do módulo.

definição de função Uma declaração que cria uma nova função, especificando seu nome, parâmetros e as instruções que executa. definição de função

determinista Pertence a um programa que faz a mesma coisa toda vez que é executado, dadas as mesmas entradas. determinista

fluxo de execução A ordem em que as instruções são executadas durante a execução de um programa. fluxo de execução

função Uma sequência nomeada de instruções que realiza alguma operação útil. As funções podem ou não aceitar argumentos e podem ou não produzir um resultado. função

função frutífera Uma função que retorna um valor. função frutífera

função vazia Uma função que não retorna um valor.

notação de ponto A sintaxe utilizada para chamar uma função em outro módulo especificando o nome do módulo seguido por um ponto e o nome da função. notação de ponto

objeto de função Um valor criado por uma definição de uma função. O nome da função é uma variável que se refere a um objeto de função. objeto de função

objeto do módulo Um valor criado por uma declaração `import` que fornece acesso aos dados e código definidos em um módulo.

parâmetro Um nome usado dentro de uma função para se referir ao valor passado como argumento.

pseudoaleatório Pertence a uma sequência de números que parecem ser aleatórios, mas são gerados por um programa determinístico.

valor de retorno O resultado de uma função. Se uma chamada de função for usada como expressão, o valor de retorno é o valor da expressão.

4.14 Exercícios

Exercício 4: Qual a finalidade da palavra-chave "def" em Python

a) É uma gíria que significa "o código a seguir é muito maneiro"

b) Ela indica o início de uma função

c) Ela indica que a próxima seção endentada do código será guardada para mais tarde

d) b e c são verdade

e) Nenhuma das opções anteriores

Exercício 5: O que o programa em Python a seguir mostrará em tela?

```
def fred():
   print("Zap")

def jane():
   print("ABC")

jane()
fred()
jane()
```

a) Zap ABC jane fred jane

b) Zap ABC Zap

c) ABC Zap jane

d) ABC Zap ABC

e) Zap Zap Zap

Exercício 6: Reescreva seu programa de cálculo de pagamento com um 1.5 o valor de hora de trabalho por hora extra, crie uma função chamada `calculoPagamento` que aceita dois parâmetros(`horas` e `TaxaHora`).

```
Insira as Horas: 45
Insira o valor da Hora de Trabalho: 10
pagamento: 475.0
```

Exercício 7: Reescreva o programa de notas do capítulo anterior usando a função `computarNotas` que recebe a pontuação como parâmetro e retorna a nota como uma string.

```
Pontuação   Nota
>= 0.9       A
>= 0.8       B
>= 0.7       C
>= 0.6       D
 < 0.6       F

Insira a pontuação: 0.95
A
```

```
Insira a pontuação: perfeito
Pontuação Inválida

Insira a pontuação: 10.0
Pontuação Inválida

Insira a pontuação: 0.75
C

Insira a pontuação: 0.5
F
```

Execute o programa repetitivamente para testar vários valores diferentes como entrada.