CAPÍTULO 5

Funções

Objetivos:

- entender o conceito de função
- saber e usar algumas funções embutidas da linguagem
- criar uma função

5.1 O QUE É UMA FUNÇÃO?

O conceito de função é um dos mais importantes na matemática. Em computação, uma função é uma sequência de instruções que computa um ou mais resultados que chamamos de parâmetros. No capítulo anterior utilizamos algumas funções já prontas do Python como o **print()**, **input()**, **format()** e **type()**.

Também podemos criar nossas próprias funções. Por exemplo, quando queremos calcular a razão do espaço pelo tempo podemos definir uma função recebendo estes parâmetros:

```
f(espaco, tempo) = espaco/tempo
```

Essa razão do espaço pelo tempo é o que chamamos de velocidade média na física. Podemos então dar este nome a nossa função:

```
velocidade(espaco, tempo) = espaco/tempo
```

Se um carro percorreu uma distância de 100 metros em 20 segundos podemos calcular sua velocidade média:

```
velocidade(100, 20) = 100/20 =     5 m/s
```

O Python permite definirmos funções como essa da velocidade média. A sintaxe é muito parecida com a da matemática. Para definirmos uma função no Python utilizamos o comando **def**:

```
def velocidade(espaco, tempo):
    pass
```

Logo após o `def` vem o nome da função e entre parêntese vêm os seus parâmetros. Uma função também tem um escopo, um bloco de instruções em que colocamos os cálculos e estes devem seguir a identação padrão do Python (4 espaços a direita).

Como nossa função ainda não faz nada, utilizamos a palavra chave **pass** para dizer ao interpretador que definiremos os cálculo depois. A palavra `pass` não é usada apenas em funções, podemos usar em qualquer bloco de comandos como nas instruções `if`, `while` e `for`, por exemplo.

Vamos substituir a palavra `pass` pelos cálculos que nossa função deve executar:

```
def velocidade(espaco, tempo):
    v = espaco/tempo
    print('velocidade: {} m/s'.format(v))
```

Nossa função faz o cálculo da velocidade média e utiliza a função `print()` do Python para imprimir na tela. Vamos testar nossa função:

```
>>> velocidade(100, 20)
velocidade: 5 m/s
```

Executar código

De maneira geral, uma função é um estrutura para agrupar um conjunto de instruções que podem ser reutilizadas. Agora qualquer parte do nosso programa pode chamar a função velocidade quando precisar calcular a velocidade média de um veículo, por exemplo. E podemos chamá-la mais de uma vez, o que significa que não precisamos escrever o mesmo código novamente.

Funções são conhecidas por diversos nomes em linguagens de programação como subrotinas, rotinas, procedimentos, métodos e subprogramas.

Podemos ter funções sem parâmetros. Por exemplo, podemos ter uma função que diz 'oi' na tela:

```
>>> def diz_oi():
...     print("oi")
>>>
>>> diz_oi()
oi
```

Executar código

5.2 Parâmetros de Função

Um conjunto de parâmetros consiste em uma lista com nenhum ou mais elementos que podem ser obrigatórios ou opcionais. Para um parâmetro ser opcional atribuímos um valor padrão (default) para ele - o mais comum é utilizar **None**. Por exemplo:

```
def dados(nome, idade=None):
    print('nome: {}'.format(nome))
    if(idade is not None):
        print('idade: {}'.format(idade))
    else:
        print('idade: não informada')
```

O código da função acima recebe uma idade como parâmetro e faz uma verificação com uma instrução **if**: se a idade for diferente de *None* ela vai imprimir a idade, caso contrário vai imprimir *idade não informada*. Vamos testar passando os dois parâmetros e depois apenas o nome:

```
>>> dados('joão', 20)
nome: joão
idade: 20
```

Agora passando apenas o nome:

```
>>> dados('joão')
nome: joão
idade: não informada
```

Executar código

E o que acontece se passarmos apenas a idade?

```
>>> dados(20)
nome: 20
idade: não informada
```

Executar código

Veja que o Python obedece a ordem dos parâmetros. Nossa intenção era passar o número 20 como `idade` mas o interpretador vai entender que estamos passando o `nome` porque não avisamos isso à ele. Caso queiramos passar apenas a `idade`, devemos nomear o parâmetro:

```
>>> dados(idade=20)
 File "<stdin>", line 1, in <module>
TypeError: dados() missing 1 required positional argument: 'nome'
```

Executar código

O interpretador vai acusar um erro já que não passamos o atributo obrigatório `nome`.

5.3 Função com retorno

E se ao invés de apenas mostrar o resultado, quisermos utilizar a velocidade média para fazer outro cálculo como calcular a aceleração? Da maneira como está, nossa função `velocidade()` não conseguimos utilizar seu resultado final para cálculos.

```
Exemplo:
aceleracao = velocidade(parametros) / tempo
```

Para conseguirmos este comportamento, precisamos que nossa função **retorne** o valor calculado por ela. No Python, utilizamos o comando **return**:

```
def velocidade(espaco, tempo):
    v = espaco/tempo
    return v
```

Testando:

```
>>> velocidade(100, 20)
5.0
```

Executar código

Ou ainda, podemos atribuir a uma variável:

```
>>> resultado = velocidade(100, 20)
>>> resultado
5.0
```

Executar código

E conseguimos utilizar no cálculo da aceleração:

```
>>> aceleracao = velocidade(100, 20)/20
0.25
```

Uma função pode conter mais de um comando **return**. Por exemplo, nossa função `dados()` que imprime o nome e a idade, pode agora retornar uma *string*. Repare que, neste caso, temos duas situações possíveis: a que a idade é passada por parâmetro e a que ela não é passada. Aqui, teremos dois comandos **return**:

```
def dados(nome, idade=None):
    if(idade is not None):
        return ('nome: {} \nidade: {}'.format(nome, idade))
    else:
        return ('nome: {} \nidade: não informada'.format(nome))
```

Apesar da função possuir dois comandos **return**, ela tem apenas um retorno -- vai retornar um ou o outro. Quando a função encontra um comando **return** ela não executa mais nada que vier depois dele dentro de seu escopo.

5.4 Retornando múltiplos valores

Apesar de uma função executar apenas um retorno, em Python podemos retornar mais de uma valor. Vamos fazer uma função calculadora que vai retornar os resultados de operações básicas entre dois números: adição(+) e subtração(-), nesta ordem.

Para retornar múltiplos valores, retornamos os resultados separados por virgula:

```
def calculadora(x, y):
    return x+y, x-y

>>> calculadora(1, 2)
(3, -1)
```

Executar código

Qual será o tipo de retorno desta função? Vamos perguntar ao interpretador através da função type:

```
>>> type(calculadora(1,2))
<class 'tuple'>
```

Executar código

Da maneira que definimos o retorno, a função devolve uma tupla. Neste caso específico, poderíamos retornar um dicionário e usar um laço `for` para imprimir os resultados:

```
>>> def calculadora(x, y):
...         return {'soma':x+y, 'subtração':x-y}
...
>>> resultados = calculadora(1, 2)
>>> for key in resultados:
...     print('{}: {}'.format(key, resultados[key]))
...
soma: 3
subtração: -1
```

Executar código

5.5 Exercícios: Funções

1.

Defina uma função chamada velocidade_media() em um script chamado `funcoes.py` que recebe dois parâmetros: a distância percorrida (em metros) e o tempo (em segundos) gasto.

```
def velocidade_media(distancia, tempo):
    pass
```

2.

Agora vamos inserir as instruções, ou seja, o que a função deve fazer. Vamos inserir os comandos para calcular a velocidade média e guardar o resultado em uma variável velocidade:

```
def velocidade_media(distancia, tempo):
    velocidade = distancia/tempo
```

3.

Vamos fazer a função imprimir o valor da velocidade média calculada:

```
def velocidade_media(distancia, tempo):
    velocidade = distancia/tempo
    print(velocidade)
```

4.

Teste o seu código chamando a função para os valores abaixo e compare os resultados com seus colegas:

- distância: 100, tempo = 20
- distância: 150, tempo = 22
- distância: 200, tempo = 30
- distância: 50, tempo = 3

5.

Modifique a função `velocidade_media()` de modo que ela retorne o resultado calculado.

6.

Defina uma função `soma()` que recebe dois números como parâmetros e calcula a soma entre eles.

7.

Defina uma função `subtracao()`que recebe dois números como parâmetros e calcula a diferença entre eles.

8.

Agora faça uma função `calculadora()` que recebe dois números como parâmetros e retorna o resultado da soma e da subtração entre eles.

9.

Modifique a função `calculadora()` do exercício anterior e faça ela retornar também o resultado da multiplicação e divisão dos parâmetros.

10.

Chame a função `calculadora()` com alguns valores.

11.

(opcional) Defina uma função `divisao()` que recebe dois números como parâmetros, calcula e retorna o resultado da divisão do primeiro pelo segundo. Modifique a função `velocidade_media()` utilizando a função `divisao()` para calcular a velocidade. Teste o seu código chamando a função `velocidade_media()` com o valores abaixo: a. distância: 100, tempo = 20 b. distância: -20, tempo = 10 c. distância: 150, tempo = 0

5.6 Número arbitrário de parâmetros (*args)

Podemos passar um número arbitrário de parâmetros em uma função. Utilizamos as chamadas variáveis mágicas do Python: ***args** e ****kwargs**. Muitos programadores tem dificuldades em entender essas variáveis. Vamos entender o que elas são.

Não é necessário utilizar exatamente estes nomes: `*args` e `**kwargs`. Apenas o asterisco(*), ou dois deles(**), será necessário. Podemos optar, por exemplo, em escrever `*var` e `**vars`. Mas `*args` e `**kwargs` é uma convenção entre a comunidade que também seguiremos.

Primeiro aprenderemos a usar o `*args`. É usado, assim como o `**kwargs`, em definições de funções. `*args` e `**kwargs` permitem passar um número variável de argumentos de uma função. O que a variável significa é que o programador ainda não sabe de antemão quantos argumentos serão passados para sua função, apenas que são muitos. Então, neste caso usamos a palavra chave `*args`.

Veja um exemplo:

```
def teste(arg, *args):
    print('primeiro argumento normal: {}'.format(arg))
    for arg in args:
        print('outro argumento: {}'.format(arg))

teste('python', 'é', 'muito', 'legal')
```

Que vai gerar a saída:

```
primeiro argumento normal:  python
outro argumento:  é
outro argumento:  muito
outro argumento:  legal
```

Executar código

O parâmetro `arg` é como qualquer outro parâmetro de função, já o `*args` recebe múltiplos parâmetros. Viu como é fácil? Também poderíamos conseguir o mesmo resultado passando um `list` ou `tuple` de argumentos, acrescido do asterisco:

```
lista = ["é", "muito", "legal"]
teste('python', *lista)
```

Executar código

Ou ainda:

```
tupla = ("é", "muito", "legal")
teste('python', *tupla)
```

Executar código

O `*args` então é utilizado quando não sabemos de antemão quantos argumentos queremos passar para uma função. O asterisco (*) executa um empacotamento dos dados para facilitar a passagem de parâmetros, e a função que recebe este tipo de parâmetro é capaz de fazer o desempacotamento.

5.7 Número arbitrário de chaves (**kwargs)

O `**kwargs` permite que passemos o tamanho variável da palavra-chave dos argumentos para uma função. Você deve usar o `**kwargs` se quiser manipular argumentos nomeados em uma função. Veja um exemplo:

```
def minha_funcao(**kwargs):
    for key, value in kwargs.items():
        print('{0} = {1}'.format(key, value))

>>> minha_funcao(nome='caelum')
nome = caelum
```

Executar código

Também podemos passar um dicionário acrescido de dois símbolos asterisco já que se trata de chave e valor:

```
dicionario = {'nome': 'joao', 'idade': 25}
minha_funcao(**dicionario)
idade = 25
nome = joao
```

Executar código

A diferença é que o `*args` espera uma tupla de argumentos posicionais enquanto o `**kwargs` um dicionário com argumentos nomeados.

5.8 Exercício - *args e **kwargs

1.

Crie um arquivo com uma função chamada `teste_args_kwargs()` que recebe três argumentos e imprime cada um deles:

```
def teste_args_kwargs(arg1, arg2, arg3):
    print("arg1: ", arg1)
    print("arg2: ", arg2)
    print("arg3: ", arg3)
```

2.

Agora vamos chamar a função utilizando o ***args**:

```
args = ('um', 2, 3)
teste_args_kwargs(*args)
```

Que gera a saída:

```
arg1: um
arg2: 2
arg3: 3
```

3.

Teste a mesma função usando o ****kargs**. Para isso criaremos um dicionário com três argumentos:

```
kwargs = {'arg3': 3, 'arg2': 'dois', 'arg1': 'um'}
teste_args_kwargs(**kwargs)
```

Que deve gerar a saída:

```
arg1: um
arg2: dois
arg3: 3
```

4.

(Opcional) Tente chamar a mesma função mas adicionando um quarto argumento na variável args e kwargs dos exercícios anteriores. O que acontece se a função recebe mais do que 3 argumentos?

5.

De que maneira você resolveria o problema do exercício anterior?

Discuta com o instrutor e seus colegas quando usar ***args** e ****kwargs**.

5.9 Exercício - Função jogar()

1.

Vamos começar definindo uma função jogar que conterá toda lógica do jogo da forca. Abra o arquivo forca.py e coloque o código do jogo em uma função `jogar()`:

```
def jogar():
    #código do jogo aqui
```

2.

Vamos tentar executar nosso jogo pelo terminal. Navegue até a pasta jogos e execute forca.py através do comando python3:

```
$ cd jogos
$ python3 forca.py
$
```

Veja que nada aconteceu já que precisamos chamar a função `jogar()` do arquivo forca.py. Para fazer isso temos que importar o arquivo forca.py dentro do interpretador do python através do comando **import**:

```
$ python3.6
>>> import forca
```

E chamar a função através do arquivo 'forca':

```
>>> forca.jogar()
"*********************************"
"***Bem vindo ao jogo da Forca!***"
"*********************************"
['_', '_', '_', '_', '_', '_']
Qual letra?
```

Agora nosso jogo funciona como esperado.

3.

Faça o mesmo com o jogo da adivinhação e execute o jogo.

5.10 Módulos e o comando import

Ao importar o arquivo forca.py estamos importando um **módulo** de nosso programa, que nada mais é do que um arquivo. Vamos verificar o tipo de `forca`:

```
>>> type(forca)
<class 'module'>
```

Veja que o tipo é um **módulo**. Antes de continuarmos com nosso jogo, vamos aprender um pouco mais sobre arquivos e módulos. Vamos melhorar ainda mais nosso jogo da Forca e utilizar o que aprendemos de funções para organizar nosso código.

CAPÍTULO ANTERIOR:

[Estrutura de dados]

PRÓXIMO CAPÍTULO:

[Arquivos]

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter