

# Paradigmas de Linguagem de Programação em Python



# Funções

# Prof. Henrique Mota



[funcoes.ipynb](#)

colab + 

## Introdução

- Até agora, estivemos satisfeitos em utilizar as funções pré-definidas do Python.
- Já sabemos como usar várias delas, como o `len`, `int`, `float`, `print` e `input`
- Mas além de usar funções já existentes, também é possível **criar as nossas próprias funções**

## Introdução

- Funções são especialmente interessantes para isolar uma tarefa específico em um trecho do programa.
- Isto permite que a solução de um problema seja reutilizada em outras partes do programa, evitando repetição.
- Além disso, tentaremos sempre fazer com que uma função seja o **mais especializada** possível para resolver uma determinada tarefa.

## Definindo uma função

- Para definir uma nova função, utilizamos a instrução **def**
- Em seguida, vem o **nome** da função, e, entre parênteses, os **parâmetros** ou **argumentos** que a função receberá
- Vejamos então como declarar uma função de soma, que recebe dois números como parâmetros e imprime sua soma na tela:

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

## Definindo [e usando] uma função

- Entretanto, apenas criar uma função não garante que ela está sendo usada.
- Para utilizá-las, devemos **chamá-las**, assim como fazemos com o **print**, por exemplo.
- Note também que usamos **:** após os parâmetros para indicar o início do bloco de código da função

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

```
soma(3, 4)
```

## Retornando valores

- Podemos fazer com que as funções **retornem** a informação computada por ela (sejam valores ou variáveis)
- Para isso, utilizamos o **return**
- Vamos escrever um programa para procurar o índice de determinado número numa lista, e retorná-lo para quem chamou a função

## Retornando valores

```
def pesquise(lista , valor):  
    for x, e in enumerate(lista):  
        if e == valor:  
            return x  
  
    return None
```

```
valores = [13, 17, 19, 23]  
print(pesquise(valores, 19))  
print(pesquise(valores, 31))
```



## Retornando valores

- O **enumerate()** retorna o índice e o valor correspondente a este índice na lista.
- **x** começa em **0** e vai até **len(lista) - 1**
- **e** começa em **lista[0]** e vai até **lista[len(lista)-1]**
- Caso o valor seja encontrado, ele será retornado e, fora da função, será printado
- Caso não seja encontrado, o retorno será **None**

## Retornando valores

- Também é possível retornar mais de um valor numa função
- Imagine que agora queremos retornar quantas vezes um determinado número apareceu na lista, e qual foi a sua última posição encontrada
- Para tal, devemos separar os valores a serem retornado por **vírgula**

## Retornando valores

```
def pesquise(lista, valor):  
    count, ult_indice = 0, None  
    for x, e in enumerate(lista):  
        if e == valor:  
            count += 1  
            ult_indice = x  
  
    return count, ult_indice
```

```
valores = [13, 17, 19, 19]  
print(pesquise(valores, 19))  
print(pesquise(valores, 31))
```

## Parâmetros e Argumentos

- Note que **podemos passar qualquer tipo de dado para as funções!** Mas, na **assinatura** da função (i.e., **`pesquise(lista, valor)`**), não forçamos que **lista** seja realmente uma lista, ou que **valor** seja realmente um número.
- Ou seja, estamos **confiando** que os nomes das variáveis indiquem os tipos que estamos esperando.
- Podemos dar "dicas" ao colocar comentários, o que é uma boa prática.
- Podemos, também, verificar se o tipo do parâmetro é mesmo o esperado com o **`isinstance()`**

## Parâmetros e Argumentos

```
def pesquisa(lista , valor):  
    if not isinstance(lista, list):  
        print("Nao eh uma lista")  
        raise TypeError  
  
    for x, e in enumerate(lista):  
        if e == valor:  
            return x  
  
    return None
```

## Parâmetros e Argumentos

### Chamada por nome

- Ainda, funções podem ter vários parâmetros, tornando sua chamada complicada se não lembrarmos da ordem exata da assinatura
- A função `calc(esqd, dirt, op)`, realiza uma operação matemática `op` em `esqd` (operando à esquerda) e `dirt` (operando à direita)
- Se não lembrarmos que `esqd` é o 1º argumento, podemos obter um resultado indesejado
- Para resolver isso, Python permite que chamemos os parâmetros pelo seu respectivo **nome**.

## Parâmetros e Argumentos

### Chamada por nome

```
def calc(esqd, dirt, op):  
    if op == '+':  
        return esqd + dirt  
    elif op == "-":  
        return esqd - dirt  
    elif op == "*":  
        return esqd * dirt  
    elif op == "/":  
        return esqd / dirt
```

```
# chamada valida (em ordem e  
# todos nomeados)  
calc(esqd=3, dirt=1, op='-')  
# chamada valida (fora de  
# ordem, mas todos nomeados)  
calc(esqd=3, op='+', dirt=1)  
# chamada invalida (último  
# argumento sem nomeacao)  
calc(esqd=3, dirt=1, '-')
```

## Parâmetros e Argumentos

### Chamada por nome

#### [+ Opcionais]

- Se chamarmos um parâmetro pelo nome, **todos** a partir deverão também ser chamados pelo nome.
- Se houver parâmetros **opcionais**, que são aqueles que já tem um valor padrão, eles não precisam ser explicitamente chamados
  - Mas se forem, e se já houver parâmetro tendo sido chamado pelo nome, os opcionais também precisam ser chamados pelo nome



## Parâmetros e Argumentos

### Chamada por nome

#### [+ Opcionais]

```
def calc(esqd, dirt, op="+"):
    if op == '+':
        return esqd + dirt
    elif op == "-":
        return esqd - dirt
    elif op == "*":
        return esqd * dirt
    elif op == "/":
        return esqd / dirt
```

```
# chamada valida (opcional
omitido)
calc(esqd=3, dirt=1)
# chamada valida (opcional
explicito e nomeado)
calc(esqd=3, dirt=1, op='-')
# chamada invalida (último
argumento sem nomeacao)
calc(esqd=3, dirt=1, '-')
```

## Módulos

- A medida que criamos mais e mais funções, os programas poderão ficar cada vez maiores
- É comum, também, que essas funções sejam, reaproveitadas em outros programas
- Em Python, resolvem-se esses dois problemas com o conceito de **módulos**

## Módulos

- A medida que criamos mais e mais funções, os programas poderão ficar cada vez maiores
- É comum, também, que essas funções sejam, reaproveitadas em outros programas
- Em Python, resolvem-se esses dois problemas com o conceito de **módulos**
  - Todo programa que escrevemos em Python (arquivo **.py**) é um módulo que pode ser reaproveitado por outros programas via **import**

## Módulos

```
# arquivo util.py
def conta_letra(palavra, letra):
    soma = 0
    for x in palavra:
        if x == letra:
            soma += 1
    return soma
```

```
def troca_valor(lista, v1, v2):
    for i, x in enumerate(lista):
        if x == v1:
            lista[i] = v2
    return lista
```

## Módulos

```
# arquivo teste.py
import util
print(util.conta_letra("abigail", 'a'))
l = [41, 3, 3, 13]
nova_lista = util.troca_valor(l, 3, 19)
```

## Módulos

- Note que **util.py** não utiliza suas próprias funções!
  - Ele apenas as cria
- Quem as utiliza é o arquivo **test.py**
- Para o **import** funcionar "simples" assim, é necessário apenas que os arquivos estejam no mesmo diretório
- Caso contrário, é preciso alterar algumas configurações do seu Sistema Operacional

Voltando....

## Parâmetros e Argumentos

### Escopo

- Quando usamos funções, começamos a trabalhar com **variáveis internas** (ou **locais**) e **variáveis externas** (ou **globais**)
- A **diferença** entre elas é a **visibilidade** (ou **escopo**)
- Uma **variável local** a uma função existe **apenas dentro dela**, sendo normalmente inicializada a cada chamada, e **não podendo ser acessível fora**.
- Uma **variável global** é **definida fora de uma função**, podendo ser **vista por todas as funções do módulo** e por todos os módulos que importam o módulo que a definiu.

Voltando....

## Parâmetros e Argumentos Escopo

```
def func1(a):  
    a = 13
```

```
def func2(lista):  
    lista[0] = 13
```

```
a = 11  
lista = [10, 10, 10]  
func1(a)  
func2(lista)  
print(a)  
print(lista)
```



Voltando....

## Parâmetros e Argumentos

### Escopo

- Ao executarmos o programa anterior, nota-se que a variável **a** continua com o valor 11 mesmo depois da chamada de **func1 ()**
- Entretanto, **lista** teve seu conteúdo alterado.
- Por que isso ocorre?

Voltando....

## Parâmetros e Argumentos

### Escopo

- Isso ocorre porque, em Python, valores imutáveis como **inteiros**, **floats** e **strings**, quando passados como argumentos, são copiados **localmente** na função.
- Essas cópias são **inacessíveis** para o **escopo global** do programa.
- Logo, todas as alterações nas variáveis imutáveis são locais.
- Se quisermos que se tornem globais, devemos utilizar o **return**, e alterar o seu valor no respectivo escopo.

Voltando....

## Parâmetros e Argumentos

### Escopo

- Entretanto, quando as variáveis são **mutáveis**, como listas e dicionários, elas são passadas como **referência**, ou seja, a própria variável é passada como argumento, e a função não realiza a cópia local.
- Logo, as alterações reverberam no escopo global.

Obrigado!

Alguma dúvida?

Prof. Henrique Mota



profhenriquemota@gmail.com