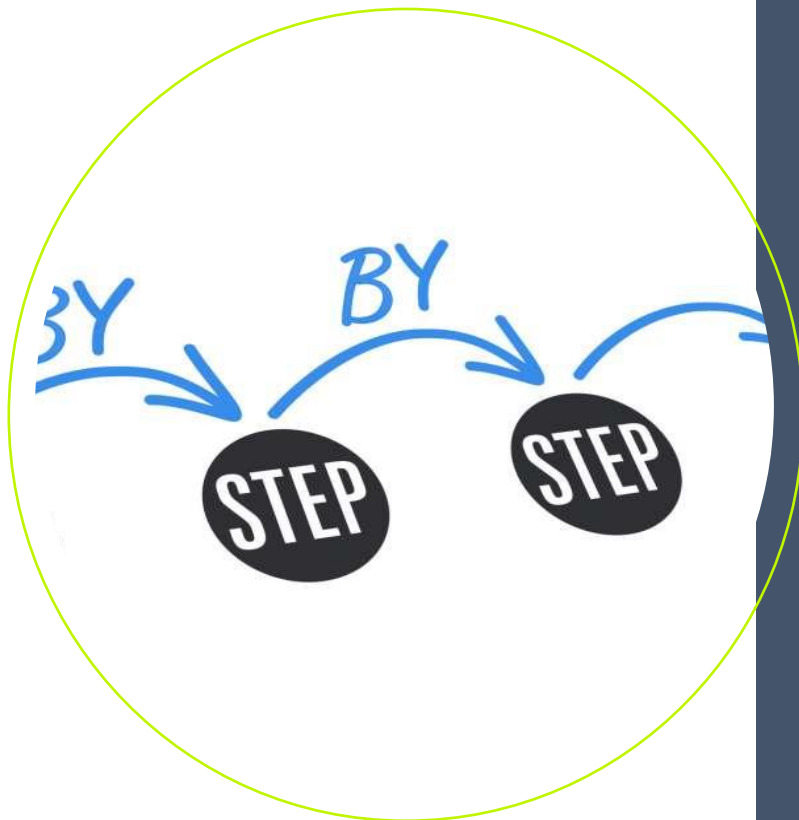




## SUMÁRIO

FLUXO DE EXECUÇÃO.....	2
ENTENDENDO A ORDEM.....	2
PARÂMETROS E ARGUMENTOS .....	3
IMPRIMEDOBRADO (BRUNO) .....	4
VARIÁVEIS E PARÂMETROS SÃO LOCAIS .....	5
ENTENDENDO O ESCOPO .....	5
FUNÇÕES COM RESULTADOS.....	6
ARQUIVO DE FUNÇÕES (IMPORT) .....	7



## ENTENDENDO A ORDEM

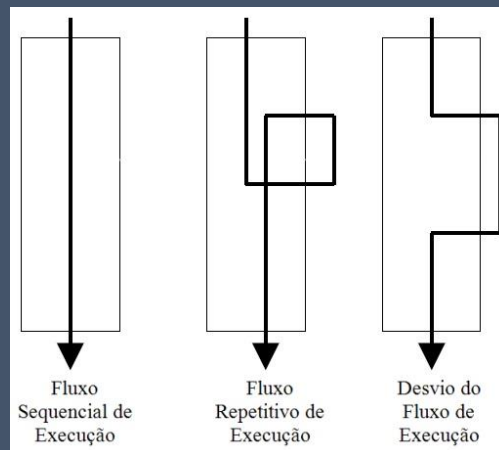
Neste caso, a definição mais interna não é executada até que a função mais externa seja chamada. Chamadas de função são como um desvio no fluxo de execução. Em vez de ir para o próximo comando, o fluxo salta para a primeira linha da função chamada, executa todos os comandos lá e então volta atrás para retomar de onde havia deixado.

Parece muito simples, até a hora em que você lembra que uma função pode chamar outra. Enquanto estiver no meio de uma função, o programa poderia executar os comandos em uma outra função. Mas enquanto estivesse executando esta nova função, o programa poderia ter de executar ainda outra função e assim por diante.

Felizmente, Python é adepto de monitorar a posição onde está, assim, cada vez que uma função se completa, o programa retoma de onde tinha parado na função que a chamou. Quando chega ao fim do programa, ele termina. Qual a moral dessa história sórdida? Quando você for ler um programa, não o leia de cima para baixo. Em vez disso, siga o fluxo de execução.

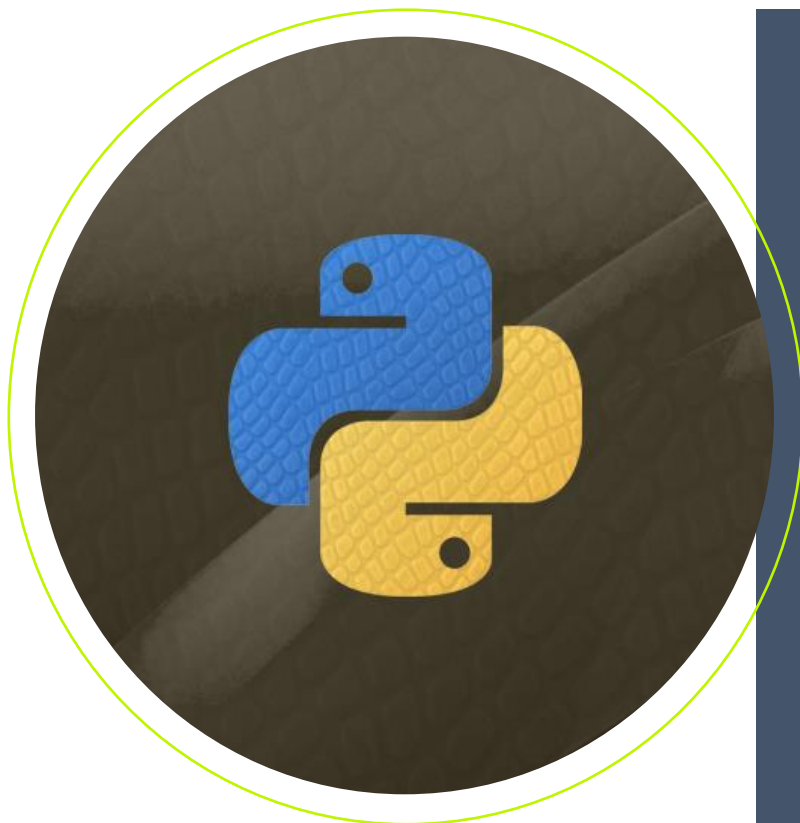
## FLUXO DE EXECUÇÃO

Para assegurar que uma função esteja definida antes do seu primeiro uso, é preciso saber em que ordem os comandos são executados, ou seja, descobrir qual o fluxo de execução do programa.



A execução sempre começa com o primeiro comando do programa. Os comandos são executados um de cada vez, pela ordem, de cima para baixo. As definições de função não alteram o fluxo de execução do programa, mas lembre-se que comandos dentro da função não são executados até a função ser chamada. Embora não seja comum, você pode definir uma função dentro de outra.





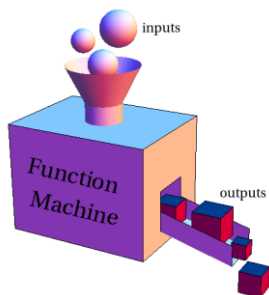
## PARÂMETROS E ARGUMENTOS

Algumas das funções nativas que você já usou requerem argumentos, aqueles valores que controlam como a função faz seu trabalho. Por exemplo, se você quer achar o seno de um número, você tem que indicar qual número é. Deste modo, `sin` recebe um valor numérico como um argumento.

Algumas funções recebem mais de um argumento. Por exemplo, `pow` recebe dois argumentos, a base e o expoente.

Dentro da função, os valores que lhe são passados são atribuídos a variáveis chamadas parâmetros. Veja um exemplo de uma função definida pelo usuário, que recebe um parâmetro:

```
def imprimeDobrado(bruno):  
    print (bruno, bruno)
```



Algumas funções podem receber um ou mais de um argumento. Tudo depende da finalidade da mesma.



## IMPRIME DOBRADO (BRUNO)

Esta função recebe um único argumento e o atribui a um parâmetro chamado `bruno`. O valor do parâmetro (a essa altura, não sabemos qual será) é impresso duas vezes, seguido de uma nova linha. Estamos usando `bruno` para mostrar que o nome do parâmetro é decisão sua, mas claro que é melhor escolher um nome que seja mais ilustrativo. A função `imprimeDobrado` funciona para qualquer tipo que possa ser impresso:

```
>>> imprimeDobrado('Spam')
```

```
Spam Spam
```

```
>>> imprimeDobrado(5)
```

```
5 5
```

Na primeira chamada da função, o argumento é uma string. Na segunda, é um inteiro. Na terceira é um float. As mesmas regras de composição que se aplicam a funções nativas também se aplicam às funções definidas pelo usuário, assim, podemos usar qualquer tipo de expressão como um argumento para `imprimeDobrado`:

```
>>> imprimeDobrado('Spam'*4)
```

```
SpamSpamSpamSpam SpamSpamSpamSpam
```

```
>>> imprimeDobrado(math.cos(math.pi))
```

```
-1.0 -1.0
```

Também podemos usar uma variável como argumento:

```
>>> miguel = 'Eric, the half a bee.'
```

```
>>> imprimeDobrado(miguel)
```

```
Eric, the half a bee. Eric, the half a bee.
```

Obs.: “Eric, the half a bee” é uma música do grupo humorístico britânico Monty Python. A linguagem Python foi batizada em homenagem ao grupo e, por isso, os programadores gostam de citar piadas deles em seus exemplos.

Repare numa coisa importante: o nome da variável que passamos como um argumento (`miguel`) não tem nada a ver com o nome do parâmetro (`bruno`). Não importa de que modo o valor foi chamado de onde veio (do ‘chamador’); aqui, em `imprimeDobrado`, chamamos a todo mundo de `bruno`.



## ENTENDENDO O ESCOPO

A função ao lado (concatDupla) recebe dois argumentos, concatena-os, e então imprime o resultado duas vezes. Podemos chamar a função com duas strings:

```
>>> canto1 = 'Pie Jesu domine, '
>>> canto2 = 'dona eis requiem. '
>>> concatDupla(canto1, canto2)
```

Pie Jesu domine, Dona eis requiem. Pie Jesu domine, Dona eis requiem.

Quando a função concatDupla termina, a variável concat é destruída. Se tentarmos imprimi-la, teremos um erro:

```
>>> print (concat)

NameError: concat
```

Parâmetros são sempre locais. Por exemplo, fora da função imprimeDobrado, não existe nada que se chama bruno. Se você tentar utilizá-la, o Python vai reclamar.

## VARIÁVEIS E PARÂMETROS SÃO LOCAIS

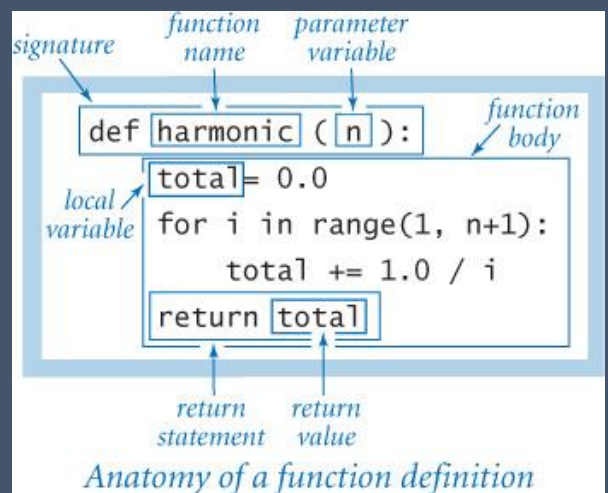
Quando você cria uma variável local dentro de uma função, ela só existe dentro da função e você não pode usá-la fora de lá. Por exemplo:

```
def concatDupla(partel1, parte2):

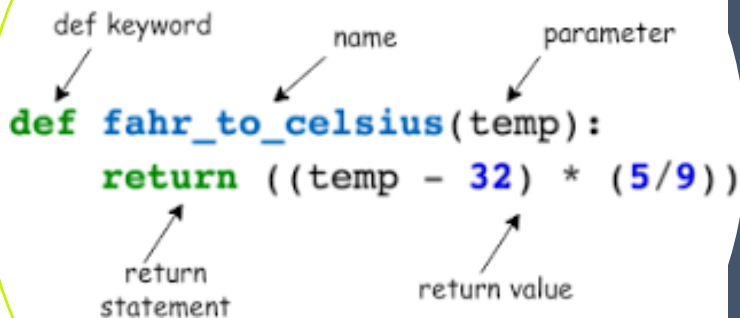
    concat = parte1 + parte2

    imprimeDobrado(concat)
```

NameError: name 'concat' is not defined







```
def fahr_to_celsius(temp):
    return ((temp - 32) * (5/9))
```

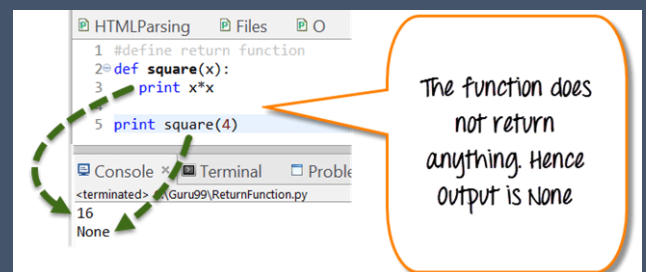
## FUNÇÕES COM RESULTADOS

A essa altura, você deve ter percebido que algumas das funções que estamos usando, tais como as funções matemáticas, produzem resultados. Outras funções, como `novaLinha`, executam uma ação, mas não retornam um valor. O que levanta algumas questões. O que acontece se você chama uma função e não faz nada com o resultado (por exemplo, não atribui o resultado a uma variável ou o usa como parte de uma expressão maior)?

O que acontece se você usa uma função que não produz resultado em uma expressão tal como `novaLinha() + 7`?

Você pode escrever funções que produzem resultados, ou está preso a funções como `novaLinha` e `imprimeDobrado`?

```
def concatDupla(partel1, parte2)
    concat = partel1 + parte2
    return concat
```



```
def fahr_to_celsius(temp):
    return ((temp - 32) * (5/9))
```

## ARQUIVO DE FUNÇÕES (IMPORT)

Uma boa prática é isolar as funções em arquivos segmentados para tal tarefa. Supomos o arquivo 'funcoes.py' na mesma pasta do arquivo 'main.py'. O arquivo de funções ficará responsável pelas funções e o main pela execução principal do projeto

**'funcoes.py'**

```
def blankLine ():
    print()
```

**'main.py'**

```
from funcoes import blankLine

blankLine()

blankLine()

blankLine()

print("Olá Mundo")
```

No fluxo principal, você pode receber o valor de uma função e armazená-lo em uma variável, a fim de executar determinadas ações em outro momento.

Outro ponto que denota a importância de usar return em funções, é a reutilização de tais códigos em outros projetos. Suponha que você crie uma função para converter um valor float no formato Brasileiro de moeda, por exemplo, 8.2 retornaria R\$ 8,20. Essa função poderia ser utilizada em diversos softwares.

```
def add(x, y):
    print(f'arguments are {x} and {y}')
    return x + y
```

Na figura acima, podemos visualizar um exemplo de como devemos pensar a criação de uma função, seguindo os 6 passos básicos para uma arquitetura funcional.

