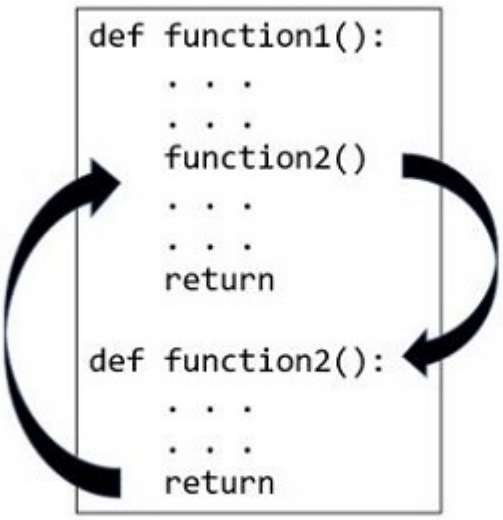
Python - Funções

Uma função Python é um bloco de código organizado e reutilizável usado para executar uma única ação relacionada. As funções fornecem melhor modularidade para seu aplicativo e um alto grau de reutilização de código.

Uma abordagem de cima para baixo para construir a lógica de processamento envolve a definição de blocos de funções reutilizáveis independentes. Uma função Python pode ser invocada a partir de qualquer outra função passando os dados necessários (chamados **parâmetros** ou **argumentos**). A função chamada retorna seu resultado ao ambiente de chamada.



Tipos de funções Python

Python fornece os seguintes tipos de funções -

- Funções integradas
- Funções definidas em módulos integrados
- Funções definidas pelo usuário

A biblioteca padrão do Python inclui várias funções integradas. Algumas das funções integradas do Python são print(), int(), len(), sum(), etc. Essas funções estão sempre disponíveis, pois são carregadas na memória do computador assim que você inicia o interpretador Python.

A biblioteca padrão também agrupa vários módulos. Cada módulo define um grupo de funções. Essas funções não estão prontamente disponíveis. Você precisa importá-los para a memória a partir de seus respectivos módulos.

Além das funções e funções integradas nos módulos integrados, você também pode criar suas próprias funções. Essas funções são chamadas de funções definidas pelo usuário .

Definindo uma função Python

Você pode definir funções personalizadas para fornecer a funcionalidade necessária. Aqui estão regras simples para definir uma função em Python.

- Os blocos de função começam com a palavra-chave **def** seguida pelo nome da função e parênteses (( )).
- Quaisquer parâmetros ou argumentos de entrada devem ser colocados entre parênteses. Você também pode definir parâmetros entre parênteses.
- A primeira instrução de uma função pode ser opcional; a string de documentação da função ou docstring.
- O bloco de código dentro de cada função começa com dois pontos (:) e é recuado.
- A instrução **return [expressão]** sai de uma função, opcionalmente devolvendo uma expressão ao chamador. Uma instrução **return** sem argumentos é o mesmo que return None.

Sintaxe para definir uma função Python

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

Por padrão, os parâmetros possuem um comportamento posicional e é necessário informá-los na mesma ordem em que foram definidos.

Depois que a função for definida, você poderá executá-la chamando-a de outra função ou diretamente do prompt do Python.

Exemplo para definir uma função Python

O exemplo a seguir mostra como definir uma função saudações(). O colchete está vazio, portanto não há parâmetros.

A primeira linha é a documentação. O bloco de funções termina com a instrução return. quando esta função for chamada, a mensagem **Hello world** será impressa.

```
def greetings():
   "This is docstring of greetings function"
   print ("Hello World")
```

```
    return

greetings()
```

Chamando uma função Python

Definir uma função apenas lhe dá um nome, especifica os parâmetros que devem ser incluídos na função e estrutura os blocos de código.

Depois que a estrutura básica de uma função estiver finalizada, você poderá executá-la chamando-a de outra função ou diretamente do prompt do Python.

Exemplo para chamar uma função Python

A seguir está o exemplo para chamar a função printme() -

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return;

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

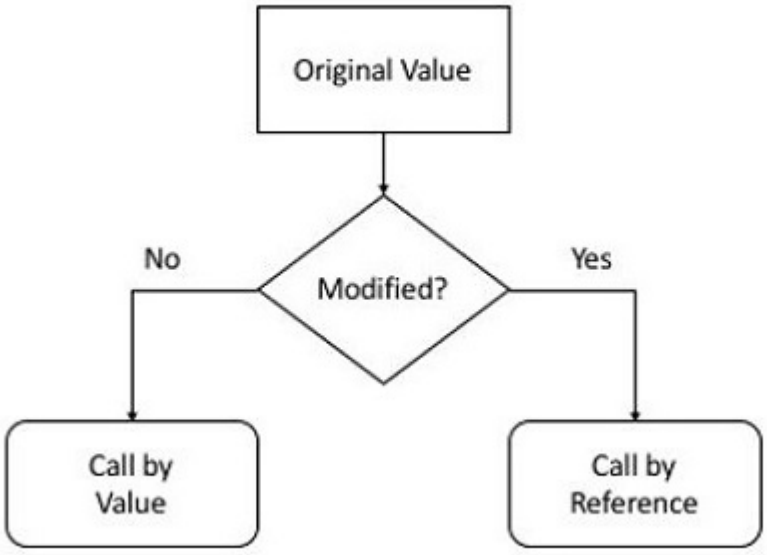
Quando o código acima é executado, ele produz a seguinte **saída** -

```
I'm first call to user defined function!
Again second call to the same function
```

Passagem por referência vs valor

O **mecanismo de chamada de função** do Python difere daquele de C e C++. Existem dois mecanismos principais de chamada de função: **Chamada por Valor** e **Chamada por Referência** .

Quando uma variável é passada para uma função, o que a função faz com ela? Se alguma alteração em sua variável não for refletida no argumento real, ele usará o mecanismo de chamada por valor. Por outro lado, se a mudança for refletida, então ela passa a ser chamada por mecanismo de referência.



Diz-se que as funções C/C++ são chamadas por value . Quando uma função em C/C++ é chamada, o valor dos argumentos reais é copiado para as variáveis que representam os argumentos formais. Se a função modifica o valor do argumento formal, ela não reflete a variável que foi passada para ela.

Python usa mecanismo de passagem por referência . Como variável em Python é um rótulo ou referência ao objeto na memória, tanto as variáveis usadas como argumento real quanto os argumentos formais realmente se referem ao mesmo objeto na memória. Podemos verificar esse fato verificando o id() da variável passada antes e depois da passagem.

```
def testfunction(arg):
   print ("ID inside the function:", id(arg))

var="Hello"
print ("ID before passing:", id(var))
testfunction(var)
```

Se o código acima for executado, o id() antes de passar e dentro da função é o mesmo.

```
ID before passing: 1996838294128
ID inside the function: 1996838294128
```

O comportamento também depende se o objeto passado é mutável ou imutável. O objeto numérico Python é imutável. Quando um objeto numérico é passado e a função altera o valor do argumento formal, ela na verdade cria um novo objeto na memória, deixando a variável original inalterada.

```
def testfunction(arg):
   print ("ID inside the function:", id(arg))
   arg=arg+1
```

```
   print ("new object after increment", arg, id(arg))

var=10
print ("ID before passing:", id(var))
testfunction(var)
print ("value after function call", var)
```

Ele produzirá a seguinte **saída** -

```
ID before passing: 140719550297160
ID inside the function: 140719550297160
new object after increment 11 140719550297192
value after function call 10
```

Vamos agora passar um objeto mutável (como uma lista ou dicionário) para uma função. Também é passado por referência, pois o id() do lidt antes e depois da passagem é o mesmo. Porém, se modificarmos a lista dentro da função, sua representação global também refletirá a mudança.

Aqui passamos uma lista, acrescentamos um novo item e vemos o conteúdo do objeto de lista original, que descobriremos que foi alterado.

```
def testfunction(arg):
   print ("Inside function:",arg)
   print ("ID inside the function:", id(arg))
   arg=arg.append(100)

var=[10, 20, 30, 40]
print ("ID before passing:", id(var))
testfunction(var)
print ("list after function call", var)
```

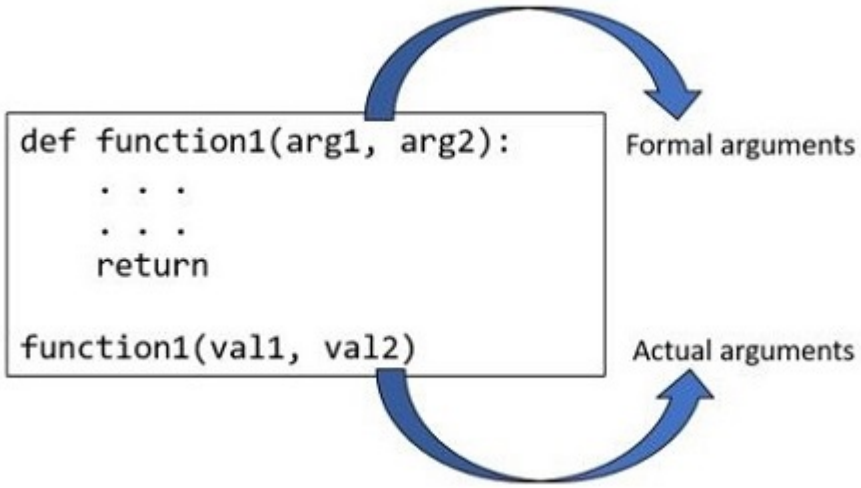
Ele produzirá a seguinte **saída** -

```
ID before passing: 2716006372544
Inside function: [10, 20, 30, 40]
ID inside the function: 2716006372544
list after function call [10, 20, 30, 40, 100]
```

Argumentos de função Python

O processo de uma função geralmente depende de certos dados fornecidos durante sua chamada. Ao definir uma função, você deve fornecer uma lista de variáveis nas quais os dados passados para ela são coletados. As variáveis entre parênteses são chamadas de argumentos formais.

Quando a função é chamada, o valor de cada um dos argumentos formais deve ser fornecido. Esses são chamados de argumentos reais.



Exemplo

Vamos modificar a função de saudações e nomear um argumento. Uma string passada para a função como argumento real torna-se uma variável de nome dentro da função.

```
def greetings(name):
   "This is docstring of greetings function"
   print ("Hello {}".format(name))
   return

greetings("Samay")
greetings("Pratima")
greetings("Steven")
```

Ele produzirá a seguinte **saída** -

```
Hello Samay
Hello Pratima
Hello Steven
```

Tipos de argumentos de função Python

Com base em como os argumentos são declarados ao definir uma função Python, eles são classificados nas seguintes categorias -

- Argumentos posicionais ou obrigatórios
- Argumentos de palavras-chave
- Argumentos padrão
- Argumentos apenas posicionais
- Argumentos somente de palavras-chave
- Argumentos arbitrários ou de comprimento variável

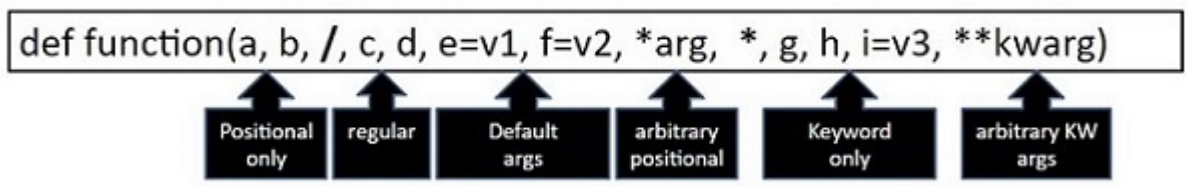
Nos próximos capítulos, discutiremos detalhadamente esses argumentos de função.

Ordem dos argumentos da função Python

Uma função pode ter argumentos de qualquer um dos tipos definidos acima. No entanto, os argumentos devem ser declarados na seguinte ordem -

- A lista de argumentos começa com argumentos somente posicionais, seguidos pelo símbolo de barra (/).
- É seguido por argumentos posicionais regulares que podem ou não ser chamados como argumentos de palavras-chave.
- Então pode haver um ou mais argumentos com valores padrão.
- A seguir, argumentos posicionais arbitrários representados por uma variável prefixada com um único asterisco, que é tratada como tupla. É o próximo.
- Se a função tiver argumentos apenas de palavras-chave, coloque um asterisco antes do início dos nomes. Alguns dos argumentos somente de palavras-chave podem ter um valor padrão.
- O último entre colchetes é o argumento com dois asteriscos ** para aceitar um número arbitrário de argumentos de palavras-chave.

O diagrama a seguir mostra a ordem dos argumentos formais -



Função Python com valor de retorno

A palavra-chave **return** como a última instrução na definição da função indica o fim do bloco funcional e o fluxo do programa volta para a função de chamada. Embora o recuo reduzido após a última instrução do bloco também implique retorno, usar retorno explícito é uma boa prática.

Junto com o controle de fluxo, a função também pode retornar o valor de uma expressão para a função de chamada. O valor da expressão retornada pode ser armazenado em uma variável para processamento posterior.

Exemplo

Vamos definir a função add(). Ele soma os dois valores passados para ele e retorna a adição. O valor retornado é armazenado em uma variável chamada resultado.

```
def add(x,y):
   z=x+y
   return z

a=10
b=20
result = add(a,b)
print ("a = {} b = {} a+b = {}".format(a, b, result))
```

Ele produzirá a seguinte saída -

```
a = 10 b = 20 a+b = 30
```