

Módulo 03

Programação Estruturada

1. Introdução às Funções

O que são funções - uma função é um bloco de código que só é executado quando chamado.

```
In [ ]: print("Olá")  
  
#definir a função  
def MinhaFuncao():  
    print("World")  
  
print("Mundo")
```

O código da função não é executado porque nunca foi chamado.

```
In [ ]: print("Olá")  
  
#definir a função  
def MinhaFuncao():  
    print("Hello World")  
  
print("Mundo")  
#chamar a função  
MinhaFuncao()
```

Execução do código passo a passo

Quando a função é chamada a execução do código **salta** para dentro da função até que todo o código da função é executado, de seguida a execução continua na linha seguinte onde a chamada da função tinha sido efetuada.

```
In [ ]: print("Antes da função")  
  
def FuncaoA():  
    print("Dentro da função")  
  
#chama a função  
FuncaoA()  
  
print("Depois da função")
```

Enquanto o código da função não terminar de ser executado o resto do programa fica à espera.

Porquê utilizar funções?

1. Reutilizar código: evitar repetir código
2. Organização: criar blocos de código facilita a sua leitura e compreensão
3. Abstração: funções fazem o que têm de fazer e não temos de saber os detalhes todos
4. Manutenção: se é necessário alterar alguma coisa, podemos simplesmente alterar o código de uma função sem ter de alterar o resto do código

2. Definindo e chamando funções

Para definir uma função utilizamos a palavra reservada **def** seguida do nome da função e :

Por exemplo: def MostrarMensagem():

O nome das funções seguem as mesmas regras que o nome das variáveis: não podem começar por um número e não podem ter espaços em branco. Cuidado com as palavras reservadas ou nomes de funções repetidos. O nome da função inclui sempre () - parentesis

```
In [ ]: def SomaDoisNumeros():
    x = 10
    y = 20
    resultado = x + y
    print(f"A soma é {resultado}")
```

Para **executar** a função utilizamos o nome seguido dos parentesis:

```
In [ ]: SomaDoisNumeros()
```

Os () - parentesis - indica que a função deve ser executada, sem os parentesis estamos a referir o objeto da função sem que esta seja executada, ou seja, é só uma referência para a função.

```
In [ ]: SomaDoisNumeros
```

Para ser possível executar uma função esta tem de ser definida primeiro, se o computador ainda não encontrou a definição da função esta não pode ser chamada

```
In [ ]: funcaoA() #erro porque a função ainda não foi definida

def funcaoA():
    print("Dentro da função A")
```

Função main

Apesar de não ser obrigatório é uma boa prática definir uma função **main** no ficheiro que tem o código principal do programa. Esta função será o ponto de entrada onde o código começa a ser executado. Desta forma também podemos utilizar este ficheiro

como um módulo que pode ser importado para outros projetos sem que o código seja executado inadvertidamente, o que acontece se importarmos um módulo que tem código fora das funções.

```
In [ ]: def main():
    # Código principal do programa
    pass

# só é executada a função main se a execução do programa começou por este ficheiro
if __name__ == "__main__":
    main()
```

3. Parâmetros e argumentos

A função **SomaDoisNumeros()** que definimos anteriormente não é muito útil. Porquê?

Para ser mais útil ela devia permitir somar quaisquer dois números e não somente 10 com o 20.

Para tornar a função mais útil vamos definir dois parâmetros para podermos "dar" à função os números que pretendemos somar.

```
In [ ]: """Função que soma dois números e mostra o resultado

Keyword arguments:
argument -- os dois números a somar
Return: nada
"""

def SomaDoisNumeros(x,y):
    #os valores a somar não estão definidos dentro da função
    resultado = x + y
    print(f"A soma é {resultado}")
```

Esta versão da função é diferente da anterior uma vez que ela **não sabe** quais são os valores que vai somar. Agora quando chamamos a função temos de passar os argumentos, ou seja, os valores para os parâmetros definidos.

```
In [ ]: SomaDoisNumeros(10,20)
```

```
In [ ]: SomaDoisNumeros(5,12)
```

Agora a função é muito mais útil porque permite somar quaisquer dois números.

Os **parâmetros** são as variáveis que **só vão existir dentro da função** e cujos valores são definidos no momento em que a função é chamada.

Os **argumentos** são os valores que atribuímos aos parâmetros no momento da execução da função.

```
In [ ]: def MostraMensagem(mensagem):
    print(mensagem)
```

```
MostraMensagem("Olá mundo")
# esta Linha dá erro porque a variável mensagem é um parâmetro da função MostraM
# e só existe dentro da função
print(mensagem)
```

Este conceito é MUITO IMPORTANTE: todos os parâmetros de uma função, bem como as variáveis que são definidas dentro de uma função, têm uma utilização (scope) limitado ao corpo da função. Isto significa que estas variáveis são criadas quando a função é chamada e são destruídas quando a execução da função termina.

Considerando o seguinte código:

```
In [ ]: def SubtraiDoisValores(x,y):
    resultado = x - y
    print(resultado)

SubtraiDoisValores(10,5)
```

A função **SubtraiDoisValores** tem dois parâmetros: **x** e **y** e uma variável local: **resultado**

Qualquer um destes parâmetros e/ou variáveis estão limitados ao corpo da função. Isto quer dizer que NÃO EXISTEM FORA DA FUNÇÃO.

```
In [ ]: def SubtraiDoisValores(x,y):
    resultado = x - y
    print(resultado)

SubtraiDoisValores(10,5)
# estas Linhas dão erros porque as variáveis
# x,y e resultado só existem dentro da função SubtraiDoisValores
# todas elas são locais
print(x)
print(y)
print(resultado)
```

A única diferença entre a variável **resultado** e as variáveis **x** e **y** (que são parâmetros) é que estes últimos têm um valor atribuído quando a função é chamada.

Os argumentos podem ser valores constantes ou variáveis.

```
In [ ]: def SubtraiDoisValores(x,y):
    resultado = x - y
    print(resultado)

a = 10
b = 5
SubtraiDoisValores(a,b)
```

Assim podemos colocar a questão: o que acontece à variável que é utilizada num argumento se o valor do parâmetro for alterado dentro da função?

```
In [ ]: def SubtraiDoisValores(x,y):
    resultado = x - y
    print(resultado)
```

```

x = 0
y = 0

a = 10
b = 5
SubtraiDoisValores(a,b)
print(a,b)

```

E a resposta é: NADA.

A razão deve-se ao facto de em Python existirem **tipos de dados imutáveis**, como por exemplo: inteiros, strings e outros. Isto quer dizer que sempre que atribuimos um valor novo a uma destes variáveis é criada uma variável nova com o novo valor. Assim, no exemplo, a variável **a** continua com o valor 10 e ao atribuir 0 ao parâmetro **x** na realidade é criada uma nova variável **x** e não é alterado o valor original do **a**

Para demonstrar este conceito podemos utilizar a função **id()** que mostra o endereço de memória de uma variável. Quando alteramos o valor da variável o endereço muda porque é criada uma variável nova.

```
In [ ]: x = 10
         print(id(x))
         print(id(x))
         x=15
         print(id(x))
```

Mas se pretendemos passar um valor que é calculado dentro da função para o resto do programa uma solução passa por fazer com que a função devolva um valor.

```
In [ ]: def Troco(pagar,dinheiro):
          dinheiro = dinheiro-pagar #esta Linha não altera o valor da variável carteira
          print(f"Pagou {pagar} e ficou com {dinheiro}")

          carteira =100
          Troco(50,carteira)
          print(f"Tem na carteira {carteira}")
```

```
In [ ]: def Troco(pagar,dinheiro):
          dinheiro = dinheiro - pagar
          print(f"Pagou {pagar} e ficou com {dinheiro}")
          #vamos devolver o valor
          return dinheiro

          carteira =100
          #Vamos chamar a função e guardar o valor que retorna
          carteira=Troco(50,carteira)
          print(f"Tem na carteira {carteira}")
```

A instrução **return** também permite terminar a execução do código da função, à semelhança da instrução **break** dentro de um ciclo.

```
In [ ]: def Troco(pagar,dinheiro):
          if dinheiro==0:
              print("Não tem dinheiro para pagar")
              return
```

```

dinheiro = dinheiro - pagar
print(f"Pagou {pagar} e ficou com {dinheiro}")
return dinheiro

carteira = 100
#Vamos chamar a função sem guardar o valor que retorna
Troco(50, carteira)
Troco(50,0)

```

Neste exemplo a função é executada sem guardar o valor de retorno.

Valores padrão para os parâmetros

É possível definir um valor padrão para um parâmetro, esse valor só será utilizado caso a função seja chamada sem um argumento para esse parâmetro.

```

In [ ]: def Saudacao(texto="mundo"):
          print(f"Olá, {texto}")

Saudacao() #função chamada sem argumentos por isso texto assume o valor padrão
Saudacao("Joaquim") #neste caso o valor padrão não é utilizado

```

Neste exemplo o parâmetro texto tem um valor padrão que é utilizado da primeira vez que a função é chamada porque nessa chamada não é fornecido nenhum valor, ou seja, a função foi chamada sem argumentos. Da segunda vez a função é chamada com o argumento "Joaquim" sendo esse o valor utilizado pelo parâmetro ao invés do valor padrão.

NB: Quando definimos um valor padrão para um parâmetro temos de definir para todos os parâmetros que se encontram à sua direita, uma vez que o computador não conseguiria saber se o valor que era indicado era para o primeiro ou para o segundo parâmetro

```

In [ ]: # se definimos um valor padrão para o parâmetro x, este deve ser o parâmetro maior
         # ou temos de definir para todos os parâmetros que se encontram à sua direita
def Soma(x=0,y):
    resultado = x + y
    print(resultado)

```

Nomear os parâmetros

Ao chamar a função os argumentos são atribuídos aos parâmetros pela ordem que foram definidos, ou seja, da esquerda para a direita, no entanto podemos tornar o nosso código mais legível se indicarmos o nome dos parâmetros e assim até podemos trocar a ordem dos parâmetros.

```

In [ ]: def Subtrai(x,y):
          resultado = x - y
          print(resultado)

Subtrai(10,5) #chamada da função com os valores 5 e 10 que são atribuidos aos parâmetros x e y
Subtrai(y=5,x=10) #chamada da função nomeando os parâmetros e invertendo a ordem
Subtrai(5,10)

```

Variáveis Globais

Como já foi dito todos os parâmetros e variáveis declaradas dentro das funções são consideradas variáveis locais, ou seja, a sua utilização (scope) está limitada ao corpo da função.

```
In [ ]: def MostraValor(numero):
    print(numero) # a variável numero só existe dentro da função

MostraValor(10)
print(f"Valor do número: {numero}") #esta linha dá erro porque a variável numero
```

No entanto podemos definir **variáveis globais** que é possível utilizar em qualquer parte do nosso código, dentro e fora das funções.

```
In [ ]: numero = 10 #esta variável é global assim é possível utilizá-la em qualquer parte

def SomaDoisNumeros():
    print(numero) # não dá erro porque a variável é global

MostraValor()
print(numero) #variável é global
```

Assim podemos cair na tentação de definir as nossas variáveis todas sempre como globais e não temos o problema de ter de controlar se a variável existe ou não em alguma parte do nosso código, mas as variáveis globais podem criar outros problemas, como se pode ver pelo código que se segue:

```
In [ ]: resultado = 0 #esta variável é global assim é possível utilizá-la em qualquer parte

def SomaDoisNumeros(x,y):
    resultado = x + y #estamos a utilizar a variável global ou a criar uma nova
    print(resultado) # não dá erro porque a variável é global

MostraValor()
print(resultado) #variável é global
```

Na realidade o código apresentado cria uma variável local com o mesmo nome da variável global, se o que pretendemos é mesmo utilizar a variável global temos de alterar o código:

```
In [ ]: soma = 0 #esta variável é global assim é possível utilizá-la em qualquer parte d

def SomaDoisNumeros(x,y):
    global soma # assim indicamos que a variável soma que vamos utilizar dentro
    soma = x + y #estamos a utilizar a variável global
    print(soma) # não dá erro porque a variável é global

SomaDoisNumeros(5,10)
print(soma) #variável é global
```

A instrução **global** indica que a variável que vai utilizada é a variável global e assim não é criada uma variável local dentro da função.

A utilização de variáveis globais deve ser limitada pois cria código mais complicado de perceber e de manter!

Call Stack

Nesta parte vamos tentar perceber, com maior detalhe, como é que as funções são executadas pelo computador.

Como já vimos ao chamar uma função a execução salta para dentro da função e o código desta é executado. E este princípio aplicação da mesma forma se dentro da função for chamada outra função.

```
In [ ]: variavel_global=10

def FuncaoA():
    variavel_local_funcao_a=5
    print("Esta é a função A")

def FuncaoB():
    variavel_local_funcao_b=15
    print("Inicio da função B")
    FuncaoA()
    print("Fim da função B")

print("No programa principal")
FuncaoA()
print("No programa principal")
FuncaoB()
print("No programa principal")
```

Sempre que uma função é chamada é criado um espaço de memória denominado de frame. É neste frame de memória que são criadas as variáveis locais. Ao concluir a execução da função esta frame é apagada e toda a informação que tinha perde-se.

Cada frame é criada e gerida pelo computador como se de uma pilha (stack) de pratos de tratar-se, em que o último prato colocado na pilha é sempre o primeiro a ser retirado, é por isso que, no código exemplo, ao chamar a FuncaoA dentro da FuncaoB estamos a colocar nessa pilha

[Demonstração da call stack](#)

Módulos

Agora que já sabemos criar funções podemos criar os nossos próprios módulos de funções e importar para os nossos programas e assim maximizar a reutilização do código.

Vamos criar um módulo com duas funções

```
In [ ]: %%writefile meu_modulo.py

def saudacao(nome):
    return f"Olá, {nome}!"

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

Agora para utilizar este módulo temos de o importar

```
In [ ]: import meu_modulo #importação do módulo

resultado = meu_modulo.soma(10, 5)
print(resultado) # Saída: 15

mensagem = meu_modulo.saudacao("Maria")
print(mensagem) # Saída: Olá, Maria!
```

Instalar módulos antes de importar

Alguns módulos são disponibilizados em repositórios e têm de ser instalados no nosso computador antes de serem importados para o programa.

Para instalar módulos utiliza-se a ferramente *pip*

Por exemplo, um módulo que pode ser útil é o `termcolor` que permite a utilização de texto com cores no terminal.

Para instalar o módulo no computador executamos o comando: **pip install termcolor**

Depois já podemos importar:

```
In [ ]: from termcolor import colored

def imprimir_texto_cor(texto, cor):
    print(colored(texto, cor))

imprimir_texto_cor("Olá, mundo!", "green")
imprimir_texto_cor("Olá, mundo!", "red")
```

Recursividade

Uma função recursiva é aquela que se chama a si mesma para resolver uma versão menor do problema. Na prática é um ciclo sem utilizar as instruções **for** ou **while**. Ao chamar a função dentro do seu próprio código estamos a promover a repetição da sua execução.

Um aspeto fundamental do código de uma função recursiva é a condição para parar de se chamar a si própria. É o caso base. Sem esta condição criamos um ciclo infinito de

chamadas à função que resulta num erro de Stack Overflow, em que esgotamos o limite de chamadas máximas da stack de execução do programa.

Exemplo: Uma função que calcula a exponenciação. Os parâmetros são o valor da base e do expoente. Como condição de paragem podemos definir o caso base em que o expoente é 1 que, como de certo sabe, resulta no valor da base, ou seja, qualquer valor elevado a 1 dá o próprio valor.

Assim para resolver 2^4 podemos dizer que é igual a $2 * 2^3$

Por sua vez 2^3 é $2 * 2^2$ e assim sucessivamente.

```
In [ ]: def Expoente(base,expoente):
    #Condição de paragem
    if expoente==1:
        return base
    else:
        #recursividade: a função chama-se a si própria
        return base*Expoente(base,expoente-1)

print(Expoente(2,4))
```

Funções com yield

As funções são blocos de código em que as variáveis são destruídas quando a execução da função termina, por isso não guardam o estado entre chamadas da mesma função.

No entanto podemos colocar a execução de uma função num estado de pausa, em que a execução é interrompida até que seja novamente chamada.

```
In [ ]: def gerar_pares(limite):
    n = 0
    while n <= limite:
        yield n      # interrompe a execução e devolve o valor de n
        #na próxima vez que a função é chamada continua na linha seguinte
        #à instrução yield
        n += 2

    # Usando o gerador
    for numero_par in gerar_pares(10):
        print(numero_par)
```