Departamento de Engenharia Informática e de Sistemas

Metodologias de Otimização e Apoio à Decisão

Conceitos básicos da programação em Python

1. Variáveis

• Tipos de variáveis

Não é necessário declarar o tipo de uma variável, sendo que este tipo muda, consoante o valor atribuído a essa variável.

Existem definidos os seguintes tipos: ***int***, ***long***, ***float***, ***complex***, ***bool***, ***str***

```
In [2]: x=3+4

In [3]: type(x)
Out[3]: int

In [4]: x=4.5*2

In [5]: type(x)
Out[5]: float

In [6]: x=4>3

In [7]: type(x)
Out[7]: bool

In [8]: x
Out[8]: True

In [9]: x=2+3j

In [10]: type(x)
Out[10]: complex
```

As *strings* (tipo ***str***), podem ser definidas com aspas simples ou duplas.

```
In [7]: x='teresa'

In [8]: type(x)
Out[8]: str

In [9]: x
Out[9]: 'teresa'

In [10]: x="teresa"

In [11]: type(x)
Out[11]: str

In [12]: x
Out[12]: 'teresa'
```

Para confirmar o tipo de uma variável, pode usar-se o comando *type*. Para mostrar o valor de uma variável, pode simplesmente digitar-se o nome desta na linha de comandos, ou usar o comando *print* (ver exemplo seguinte).

```
n=23
pi=3.141592653589
nome="Teresa"
print(n)
print(pi)
print(nome)
print(n,pi,nome)
```

Para o bloco de código anterior o resultado seria:

```
In [26]: runfile('C:/Users/Teresa/Arquivos/MOAD/AULAS PRÁTICAS/Introdução ao
Python/Projetos/untitled0.py', wdir='C:/Users/Teresa/Arquivos/MOAD/AULAS
PRÁTICAS/Introdução ao Python/Projetos')
23
3.141592653589
Teresa
23 3.141592653589 Teresa
```

Alternativamente, a atribuição de um valor a uma variável pode ser feita através do comando *input()*.

```
nome=input("Indique o seu nome:")
idade=int(input("Idade:"))
altura=float(input("Altura (em cm):"))
peso=float(input("Peso (em Kg):"))
```

O resultado deste bloco de código (na consola) seria:

```
Indique o seu nome:Carlos Santos

Idade:31

Altura (em cm):179

Peso (em Kg):71.3

In [11]: type(nome)
Out[11]: str

In [12]: type(idade)
Out[12]: int

In [13]: type(altura)
Out[13]: float

In [14]: type(peso)
Out[14]: float
```

- ## Nomes de variáveis

Os nomes das variáveis podem ser arbitrariamente longos. Eles podem conter letras e números, mas não podem começar com um número. É válido usar letras maiúsculas, mas normalmente inicia-se o nome com uma letra minúscula. O caractere '_' pode ser usado para separar várias palavras no nome. Exemplo: *nome_completo_do_aluno*.

Existem palavras-chave reservadas do Python que não podem ser usadas nos nomes das variáveis. A saber:

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

2. Operadores

• Operadores aritméticos

Os operadores **+**, **-**, ***, */*, *//*, e ** realizam a **adição**, **subtração**, **multiplicação**, **divisão**, **divisão inteira** e **exponenciação**, respetivamente.

```
In [27]: 2+3
Out[27]: 5

In [28]: 2-3
Out[28]: -1

In [29]: 2/3
Out[29]: 0.6666666666666666

In [30]: 2*3
Out[30]: 6

In [31]: 2**3
Out[31]: 8

In [32]: 2//3
Out[32]: 0
```

O operador **%** permite obter o **resto da divisão inteira** de dois números.

```
quociente=5/2
print(quociente)
resto_divisao=5%2
print(resto_divisao)
```

O resultado deste bloco de código seria:

```
In [36]: runfile('C:/Users/Teresa/Arquivos/MOAD/AULAS PRÁTICAS/Introdução
ao Python/Projetos/untitled0.py', wdir='C:/Users/Teresa/Arquivos/MOAD/
AULAS PRÁTICAS/Introdução ao Python/Projetos')
2.5
1
```

• Operadores lógicos

Os operadores lógicos mais importantes são ***and***, ***or*** e ***not***.

O resultado da operação é sempre um valor ***True*** ou ***False***.

• Operadores relacionais

Os operadores relacionais (de comparação) são: **<**, **>**, **<=**, **>=**, **==** e **!=**.

O resultado da operação é sempre um valor ***True*** ou ***False***.

• Operadores de atribuição

Dentro desta categoria, existem os seguintes operadores:

- ❖ **=** : operador de atribuição simples (x=valor).
- ❖ **+=** : operador de atribuição composta (x+=valor ⇔ x = x + valor). Os operadores **-=**, ***/=***, ***//=***, **%=** e ***=**, funcionam da mesma forma que o **+=**.

• Precedência de operadores

A tabela seguinte mostra qual a precedência dos vários operadores numa expressão (https://www.tutorialspoint.com/python/operators_precedence_example.htm):

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'td>
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Note-se que quando os operadores têm a mesma precedência, são avaliados da esquerda para direita.

- **Operadores + e * com *strings***

O operador **+**, que anteriormente foi apresentado como operador aritmético de adição, funciona como operador de **concatenação** quando os operandos forem do tipo *string*.

```
In [37]: "Lança"+"mento"
Out[37]: 'Lançamento'
```

Por sua vez, o operador ***** também pode ser usado com um operando do tipo *string*, multiplicando o conteúdo desta por um operando (obrigatoriamente) inteiro.

```
In [38]: "Fixe!"*3
Out[38]: 'Fixe!Fixe!Fixe!'

In [43]: 2*"Ai!"
Out[43]: 'Ai!Ai!'
```

3. Strings

- **Funções**

Para obter o tamanho de uma *string*, pode usar-se a função *len()* que devolve o nº de caracteres da mesma:

```
frase="Hoje está um dia lindo!"
print("Tamanho da frase: ",len(frase))
```

O resultado seria:

```
In [49]: runfile('C:/Users/Teresa/Arquivos/MOAD/
AULAS PRÁTICAS/PYTHON/Projetos')
Tamanho da frase:  23
```

• Métodos

Em Python existem vários métodos para manipular *strings*. Alguns dos mais úteis são apresentados na tabela que se segue.

Método	Descrição
lower()	Devolve uma *string* com todas as letras da *string* original em minúsculas
upper()	Devolve uma *string* com todas as letras da *string* original em maiúsculas
replace(x,y)	Devolve uma *string* com todas as ocorrências de *x* substituídas por *y*
count(x)	Conta o nº de ocorrências de *x* na *string*
index(x)	Devolve a localização da 1ª ocorrência de *x* na *string*
Isalpha()	Devolve *True* se todos os caracteres da *string* forem letras

Para conhecer outros métodos, digitar *help(str)* na consola. De notar que os métodos *lower()*, *upper()* e *replace()*, não alteram a *string* original. Se o método *isalpha()* for aplicado à *string*, só devolverá o valor *True* se todos os caracteres forem letras (os espaços em branco não estão nessa categoria).

Alguns exemplos:

```
In [32]: frase="Hoje tenho um exame no ISEC"

In [33]: frase.lower()
Out[33]: 'hoje tenho um exame no isec'

In [34]: frase.upper()
Out[34]: 'HOJE TENHO UM EXAME NO ISEC'

In [35]: frase.replace("E","*")
Out[35]: 'Hoje tenho um exame no IS*C'

In [36]: frase.replace("e","*")
Out[36]: 'Hoj* t*nho um *xam* no ISEC'

In [37]: frase.replace("SEC","PO")
Out[37]: 'Hoje tenho um exame no IPO'
```

```
In [39]: frase.count("e")
Out[39]: 4

In [40]: frase.index("e")
Out[40]: 3

In [41]: frase.isalpha()
Out[41]: False

In [42]: frase[0].isalpha()
Out[42]: True
```

- ## Indexar (*indexing*) *strings*

Para selecionar caracteres individuais de uma *string*, usa-se parênteses retos. A tabela que se segue dá alguns exemplos de indexação da *string frase* = 'Python'.

Instrução	Resultado	Descrição
frase[0]	P	1º caractere de *frase*
frase[1]	Y	2º caractere de *frase*
frase[-1]	N	Último caractere de *frase*
frase[-2]	O	Penúltimo caractere de *frase*

Note-se que o primeiro caractere de *frase* é *frase[0]*, não *frase[1]*. Por outro lado, índices negativos contam para trás, a partir do final da *string*.

Se tentarmos aceder a uma posição que não existe, por exemplo, *frase[10]*, obtemos o seguinte erro:

```
IndexError: string index out of range
```

- ## Fatiar *(slicing) strings*

Uma *fatia* (*slice*) é usada para selecionar uma parte de uma *string*.

SINTAXE: *Nome_string [Limite_Inferior : Limite_Superior : passo]*

Devolve uma *string* com os elementos que ocupam as posições desde o limite inferior, até o limite superior - 1. O 3º argumento (*passo*) é 1, por omissão. Se for negativo, inverte a *string*.

Considerando o mesmo exemplo com a *string frase* = 'Python':

```
In [23]: frase[1:4]
Out[23]: 'yth'

In [24]: frase[2:]
Out[24]: 'thon'

In [25]: frase[:4]
Out[25]: 'Pyth'

In [30]: frase[1:6:2]
Out[30]: 'yhn'

In [31]: frase[::-1]
Out[31]: 'nohtyP'
```

4. Listas

Uma lista é um conjunto sequencial de valores, onde cada valor é identificado através de um índice, sendo que o primeiro valor tem índice 0.

SINTAXE: *Nome_lista = [valor1, valor2, ..., valorN]*

Uma lista pode ter elementos de qualquer tipo, inclusivamente, outras listas.

Os exemplos que se seguem mostram como criar uma lista, aceder aos seus elementos ou alterá-los (através de uma simples atribuição de valor).

```
In [43]: Lista = [8 , 'setembro' , 5.7 , [1 , 4 , 3] , "Coimbra" ]

In [44]: Lista[2]
Out[44]: 5.7

In [45]: Lista[3]
Out[45]: [1, 4, 3]

In [46]: Lista[4]
Out[46]: 'Coimbra'

In [47]: Lista[3]='evento'

In [48]: Lista
Out[48]: [8, 'setembro', 5.7, 'evento', 'Coimbra']
```

Se tentarmos atribuir um valor à lista, acedendo a um índice que não existe dentro desta, obtemos um erro.

```
In [49]: Lista[5]=23
Traceback (most recent call last):

  File "C:\Users\HP\AppData\Local\Temp/ipykernel_2676/2812850633.py",
line 1, in <module>
    Lista[5]=23

IndexError: list assignment index out of range
```

- **Semelhança com as *strings***

Há vários pontos em comum entre listas e *strings*, como sejam:

- ❖ Função *len()* - Devolve o número de itens da lista
- ❖ Indexação e *slicing* - Funcionam exatamente como nas *strings*.
- ❖ Métodos *index()* e *count()* – Semelhantes aos que existem para as *strings*.
- ❖ Operadores + e * - O operador + adiciona uma lista ao final de outra; o operador * repete uma lista.

Alguns exemplos:

```
In [54]: Lista
Out[54]: [8, 'setembro', 5.7, 'evento', 'Coimbra']

In [55]: len(Lista)
Out[55]: 5

In [56]: Lista+[5,"Animais"]
Out[56]: [8, 'setembro', 5.7, 'evento', 'Coimbra', 5, 'Animais']

In [57]: Lista*2
Out[57]:
[8,
 'setembro',
 5.7,
 'evento',
 'Coimbra',
 8,
 'setembro',
 5.7,
 'evento',
 'Coimbra']
```

(Cont.)

```
In [61]: Lista=Lista*2

In [63]: Lista.count(8)
Out[63]: 2

In [65]: Lista.index("Coimbra")
Out[65]: 4
```

- ## Funções

Função	Descrição
len()	Devolve o nº de elementos da lista
sum()	Devolve a soma dos elementos da lista *(se forem todos numéricos)*
min()	Devolve o menor valor da lista *(se todos os elementos forem numéricos)*
max()	Devolve o maior valor da lista *(se todos os elementos forem numéricos)*

Exemplo:

```
In [70]: Idades=[12,13,11,11,14,12]

In [71]: media=sum(Idades)/len(Idades)

In [72]: media
Out[72]: 12.166666666666666
```

- ## Métodos

Método	Descrição
append(x)	Adiciona *x* ao final da lista
sort()	Ordena a lista
count(x)	Devolve o nº de vezes que *x* ocorre na lista
index(x)	Devolve a localização da 1ª ocorrência de *x* na lista
reverse()	Inverte a lista
remove(x)	Remove a 1ª ocorrência de *x* na lista
pop(p)	Remove o elemento na localização *p* da lista e devolve o seu valor
insert(p,x)	Insere x na localização *p* da lista

Para conhecer outros métodos, digitar *help(list)* na consola. Note-se que há uma grande diferença entre métodos de lista e métodos de *string*: os métodos de *string* não alteram a *string* original, mas os métodos de lista alteram a lista original.

Alguns exemplos:

```
In [73]: L=[4,6,7,1,9,3,5,0]

In [74]: L.append(8)

In [75]: L
Out[75]: [4, 6, 7, 1, 9, 3, 5, 0, 8]

In [76]: L.reverse()

In [77]: L
Out[77]: [8, 0, 5, 3, 9, 1, 7, 6, 4]

In [78]: L.sort()

In [79]: L
Out[79]: [0, 1, 3, 4, 5, 6, 7, 8, 9]

In [80]: L.remove(0)

In [81]: L
Out[81]: [1, 3, 4, 5, 6, 7, 8, 9]
```

- ## Criação de listas através da função *range()*

A função *range()* define um intervalo de valores inteiros. Na sua forma mais simples, *range(n)*, gera valores entre *0* e *n-1*.

Formas possíveis de utilização:

Instrução	Resultado	Comentário
range(5)	Intervalo de valores 0, 1, 2, 3 e 4	
range(1,5)	Intervalo de valores 1, 2, 3, 4	Sequência com início em valor diferente de 0
range(1,10,2)	Intervalo de valores 1, 3, 5, 7, 9	Sequência tem passo diferente de 1
range(5,1,-1)	Intervalo de valores 5, 4, 3, 2	Sequência decrescente

Uma das utilizações da função *range()* é na criação de listas, conjugando-a com a função *list()*. Esta última recebe o intervalo de valores criados pela 1ª função e converte-os numa lista.

Alguns exemplos:

```
In [90]: L1=list(range(6))

In [91]: L1
Out[91]: [0, 1, 2, 3, 4, 5]

In [92]: L2=list(range(2,7))

In [93]: L2
Out[93]: [2, 3, 4, 5, 6]

In [94]: L3=list(range(1,8,2))

In [95]: L3
Out[95]: [1, 3, 5, 7]

In [96]: L4=list(range(8,2,-1))

In [97]: L4
Out[97]: [8, 7, 6, 5, 4, 3]
```

5. Tuplos

Um tuplo é, essencialmente, uma lista imutável, em que os seus elementos não podem ser alterados.

SINTAXE: *Nome_tuplo = (valor1, valor2, ..., valor)*

Os tuplos são delimitados por parênteses curvos, em vez de parênteses retos. A indexação e o *slicing* funcionam da mesma forma que nas listas. Pode obter-se o tamanho dos tuplos usando a função *len()* e também pode usar-se os métodos *count()* e *index()*. No entanto, ao contrário das listas, não dispõem de métodos como *sort()* ou *reverse()*, uma vez que são imutáveis.

Alguns exemplos:

```
In [98]: T=(1,2,3)

In [99]: T
Out[99]: (1, 2, 3)

In [100]: T[0]
Out[100]: 1

In [101]: T.index(3)
Out[101]: 2

In [102]: len(T)
Out[102]: 3
```

6. Dicionários

Um dicionário é uma versão mais genérica de uma lista. Contém um conjunto de valores, onde cada valor é associado a uma chave de acesso.

SINTAXE: *Nome_dicionario = { chave1 : valor1,*

chave2 : valor2,

chave3 : valor3,

.....

chaveN : valorN }

Para perceber melhor a diferença entre um dicionário e uma lista, vamos considerar o exemplo do nº de dias dos meses do ano. Em primeiro lugar, vamos definir a seguinte lista:

dias = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

Se quisermos o número de dias de janeiro, usamos *dias[0]*, de dezembro, usamos *dias[11]* ou *dias[-1]*.

Alternativamente, podemos definir um dicionário com os dias nos meses do ano da seguinte forma:

dias = {'janeiro':31, 'fevereiro':28, 'março':31, 'abril':30,

'maio': 31, 'junho': 30, 'julho': 31, 'agosto': 31,

'setembro':30, 'outubro':31, 'novembro':30, 'dezembro':31}

Neste caso, para obter o número de dias de janeiro, usamos *dias['janeiro']*. Uma vantagem de usar dicionários, é que o código fica mais legível, e não precisamos de descobrir qual é índice da lista correspondente a um determinado mês.

As chaves geralmente são *strings*, mas também podem ser números inteiros, números reais, entre outros. Com efeito, podem ser misturados diferentes tipos de chaves no mesmo dicionário, bem como diferentes tipos de valores.

O seguinte exemplo mostra como criar um novo dicionário e aceder a elementos individuais, através da sua chave. Se esta não existir, é gerado um erro.

```
D={"Teresa":11,"Guilherme":6,"Luís":20}
print("----------------------------")
print(D)

print("----------------------------")
print(D["Guilherme"])

print("----------------------------")
print(D["Ana"])
```

Para o código anterior é obtido o seguinte resultado:

```
In [105]: runfile('C:/Users/Teresa/Arquivos/MOAD/AULAS PRÁTICAS/
Introdução ao Python/Projetos/untitled0.py', wdir='C:/Users/Teresa/
Arquivos/MOAD/AULAS PRÁTICAS/Introdução ao Python/Projetos')
----------------------------
{'Teresa': 11, 'Guilherme': 6, 'Luís': 20}
----------------------------
6
----------------------------
Traceback (most recent call last):

  File "C:\Users\Teresa\Arquivos\MOAD\AULAS PRÁTICAS\Introdução ao
Python\Projetos\untitled0.py", line 16, in <module>
    print(D["Ana"])

KeyError: 'Ana'
```

Também é possível modificar ou acrescentar novos elementos ao dicionário:

```
In [106]: D["Teresa"]=10

In [107]: D
Out[107]: {'Teresa': 10, 'Guilherme': 6, 'Luís': 20}

In [108]: D["Ana"]=14

In [109]: D
Out[109]: {'Teresa': 10, 'Guilherme': 6, 'Luís': 20, 'Ana': 14}
```

Note-se que os elementos do dicionário não são ordenados, por isso a ordem de impressão dos valores pode não ser sempre a mesma.

• Operações em dicionários

Seguidamente são apresentadas algumas das operações normalmente efetuadas em dicionários.

	Descrição	Exemplo

del()	*Função que elimina um elemento de um dicionário através da chave*	`In [113]: del(D["Ana"])`
		
		`In [114]: D`
		`Out[114]: {'Teresa': 10, 'Guilherme': 6, 'Luís': 20}`

in	*Operador que verifica se uma chave existe num dicionário*	`In [115]: "Teresa" in D`
		`Out[115]: True`
		
		`In [116]: "Fernando" in D`
		`Out[116]: False`

keys()	*Método que obtém as chaves de um dicionário*	`In [117]: D.keys()` `Out[117]: dict_keys(['Teresa', 'Guilherme', 'Luís'])`
values()	*Método que obtém os valores de um dicionário*	`In [118]: D.values()` `Out[118]: dict_values([10, 6, 20])`
Items()	*Método que obtém os pares (chave,valor) de um dicionário*	`In [119]: D.items()` `Out[119]: dict_items([('Teresa', 10), ('Guilherme', 6), ('Luís', 20)])`

copy()	*Método que cria uma cópia de um dicionário*	`In [120]: D2=D.copy()`
		
		`In [121]: D2`
		`Out[121]: {'Teresa': 10, 'Guilherme': 6, 'Luís': 20}`

Usando a função *list()* (anteriormente apresentada), é possível criar uma lista de chaves, uma lista de valores, ou uma lista de pares *(chave,valor)*, a partir de um dicionário.

```
In [122]: list(D)
Out[122]: ['Teresa', 'Guilherme', 'Luís']

In [123]: list(D.values())
Out[123]: [10, 6, 20]

In [124]: list(D.items())
Out[124]: [('Teresa', 10), ('Guilherme', 6), ('Luís', 20)]
```

7. Bibliotecas

As bibliotecas armazenam funções pré-definidas, que podem ser usadas pelos nossos programas. Para tal, devemos utilizar o comando *import*:

O exemplo seguinte mostra a importação de uma biblioteca de funções matemáticas, para calcular o fatorial de um número:

```
import math

print(math.factorial(5))
```

Alternativamente, pode importar-se apenas uma função específica de uma biblioteca. Usando o mesmo exemplo:

```
from math import factorial

print(factorial(5))
```

Em qualquer das situações, o resultado obtido seria:

```
In [1]: runfile('C:/Users/Teresa/Arquivos/MOAD/AULAS PRÁTICAS/PYTHON/Projetos/
AULAS PRÁTICAS/PYTHON/Projetos')
120
```

De referir que existem bibliotecas padrão que são instaladas juntamente com o Python, como é o caso da ***math***, havendo também bibliotecas externas disponíveis para instalação, como a ***PuLP*** que iremos usar na resolução de problemas de programação linear.

8. Estruturas de seleção

- **if**

SINTAXE:

if <condição>:

<Bloco de instruções>

Exemplo:

```
idade = int(input("Indique a sua idade:"))
if idade >= 18:
    print("Já é adulto!")
```

Resultado:

```
In [3]: runfile('C:/Users/Teresa/Arquivos/MOAD/
AULAS PRÁTICAS/PYTHON/Projetos')

Indique a sua idade:35
Já é adulto!
```

- **if..else**

SINTAXE:

if <condição>:
    <Bloco de instruções para condição verdadeira>
else:
    <Bloco de instruções para condição falsa>

Exemplo:

```
idade = int(input("Indique a sua idade:"))
if idade >= 18:
    print("Já é adulto!")
else:
    print("É menor de idade!")
```

Resultado:

```
In [4]: runfile('C:/Users/Teresa/Arquivos/MOAD
AULAS PRÁTICAS/PYTHON/Projetos')

Indique a sua idade:7
É menor de idade!
```

- **if..elif..else**

SINTAXE:

if <condição 1>:
    <Bloco de instruções 1>
elif <condição 2>:
    <Bloco de instruções 2>
elif <condição 3>:
    <Bloco de instruções 3>
…
else:
    <Bloco de instruções *default*>

Exemplo:

```
idade = int(input("Indique a sua idade:"))
if idade < 18:
    print("Pertence ao escalão 1!")
elif idade >=18 and idade <40:
    print("Pertence ao escalão 2!")
elif idade >=40 and idade <65:
    print("Pertence ao escalão 3!")
else:
    print("Pertence ao escalão 4!")
```

Resultado:

```
In [5]: runfile('C:/Users/Teresa/Arquivos/MOAD/
AULAS PRÁTICAS/PYTHON/Projetos')

Indique a sua idade:25
Pertence ao escalão 2!
```

9. Estruturas de repetição

- **while**

SINTAXE:

while <condição>:

<Bloco de instruções>

Exemplo:

```
idade=-1
while idade <=0:
    idade = int(input("Indique a sua idade:"))
    if idade <= 0:
        print("Idade inválida! Insira de novo!")
    else:
        print("A sua idade=",idade)
```

Resultado:

```
In [7]: runfile('C:/Users/Teresa/Arquivos/MOAD/
AULAS PRÁTICAS/PYTHON/Projetos')

Indique a sua idade:-3
Idade inválida! Insira de novo!

Indique a sua idade:34
A sua idade= 34
```

- **for**

SINTAXE:

for <variável> in range (início, limite, passo):
<Bloco de instruções>

ou

for <variável> in <lista>:
<Bloco de instruções>

Exemplo:

```
# Soma de 1 + 2 + 3 + ... + 99 + 100

soma=0
for x in range(1,100,1):
    soma+=x;
print("Soma=",soma)
```

Resultado:

```
In [9]: runfile('C:/Users/Teresa/Arquivos/MOAD/
AULAS PRÁTICAS/PYTHON/Projetos')
Soma= 4950
```

Exemplo:

```
Lista_idades= [13,64,26,7,18,39]

soma=0
for x in Lista_idades:
    soma+=x;
print("Soma das idades=",soma)
```

Resultado:

```
In [11]: runfile('C:/Users/Teresa/Arquivos/MOAD/
AULAS PRÁTICAS/PYTHON/Projetos')
Soma das idades= 167
```

10. Funções

Funções são pequenos blocos de código reutilizáveis, agrupados sob um determinado nome (*nome da função*), as quais podem ser executadas a partir de qualquer local de um programa.

• Definição de uma função

Para definir uma função usa-se a palavra ***def***, de acordo com a sintaxe apresentada em seguida. De referir que a lista de parâmetros é opcional.

SINTAXE:

def <nome_da_função> (<lista de parâmetros>):

<Bloco de instruções>

Exemplo:

```
def mensagem():
    print ("Bom dia a toda a gente!!!")
```

Resultado da chamada da função:

```
In [14]: mensagem()
Bom dia a toda a gente!!!
```

• Parâmetros / argumentos

Parâmetros são as variáveis que podem ser colocadas dentro dos parênteses quando se define uma função. Quando a função é chamada, são passados valores para essas variáveis, valores esses que têm o nome de argumentos.

Exemplo:

```
def soma(x,y):
    print("soma=",x+y)
```

Resultado da chamada da função:

```
In [22]: soma(3,8)
soma= 11
```

- ## Variáveis locais e globais

Uma variável utilizada dentro de uma função é **local** a essa função, ou seja, não poderá ser usada por outras funções ou pelo programa principal. Se fora da função existir outra variável com o mesmo nome, serão duas variáveis completamente independentes.

Exemplo:

```
# Função
def produto(x,y):
   total = x*y
   print("Valor do total (função) = ",total)

# Programa principal
total = 20
produto(2,4)
print("Valor do total (programa principal) = ",total)
```

Resultado:

```
In [3]: runfile('C:/Users/Teresa/Arquivos/MOAD/
AULAS PRÁTICAS/PYTHON/Projetos')
Valor do total (função) =  8
Valor do total (programa principal) =  20
```

Uma variável **global** poderá ser partilhada por várias funções e pelo programa principal, sendo que é necessário declará-la como tal, através da instrução ***global*** em todas essas funções.

Exemplo:

```
# Função
def produto(x,y):
    global total
    total = x*y
    print("Valor do total (função) = ",total)

# Programa principal
global total
total = 20
produto(2,4)
print("Valor do total (programa principal) = ",total)
```

Resultado:

```
In [4]: runfile('C:/Users/Teresa/Arquivos/MOAD/
AULAS PRÁTICAS/PYTHON/Projetos')
Valor do total (função) =  8
Valor do total (programa principal) =  8
```

- ## Instrução *return*

Esta instrução consiste na palavra-chave ***return***, seguida de um valor de retorno opcional, a qual pode ser usado dentro de uma função para devolver o resultado desta, terminando a sua execução. Se nenhum valor de retorno for especificado ou a instrução ***return*** for simplesmente omitida, a função retornará o valor *None* (que corresponde a não retornar nada).

De notar que o valor de retorno pode ser qualquer objeto Python (valor numérico, lista, dicionário, função, …).

Exemplo:

```
# Função
def produto(x,y):
    total = x*y
    return total

# Programa principal

p=produto(2,4)
print("Valor do produto = ",p)
```

Resultado:

```
In [5]: runfile('C:/Users/Teresa/Arquivos/MOAD/
AULAS PRÁTICAS/PYTHON/Projetos')
Valor do produto =  8
```