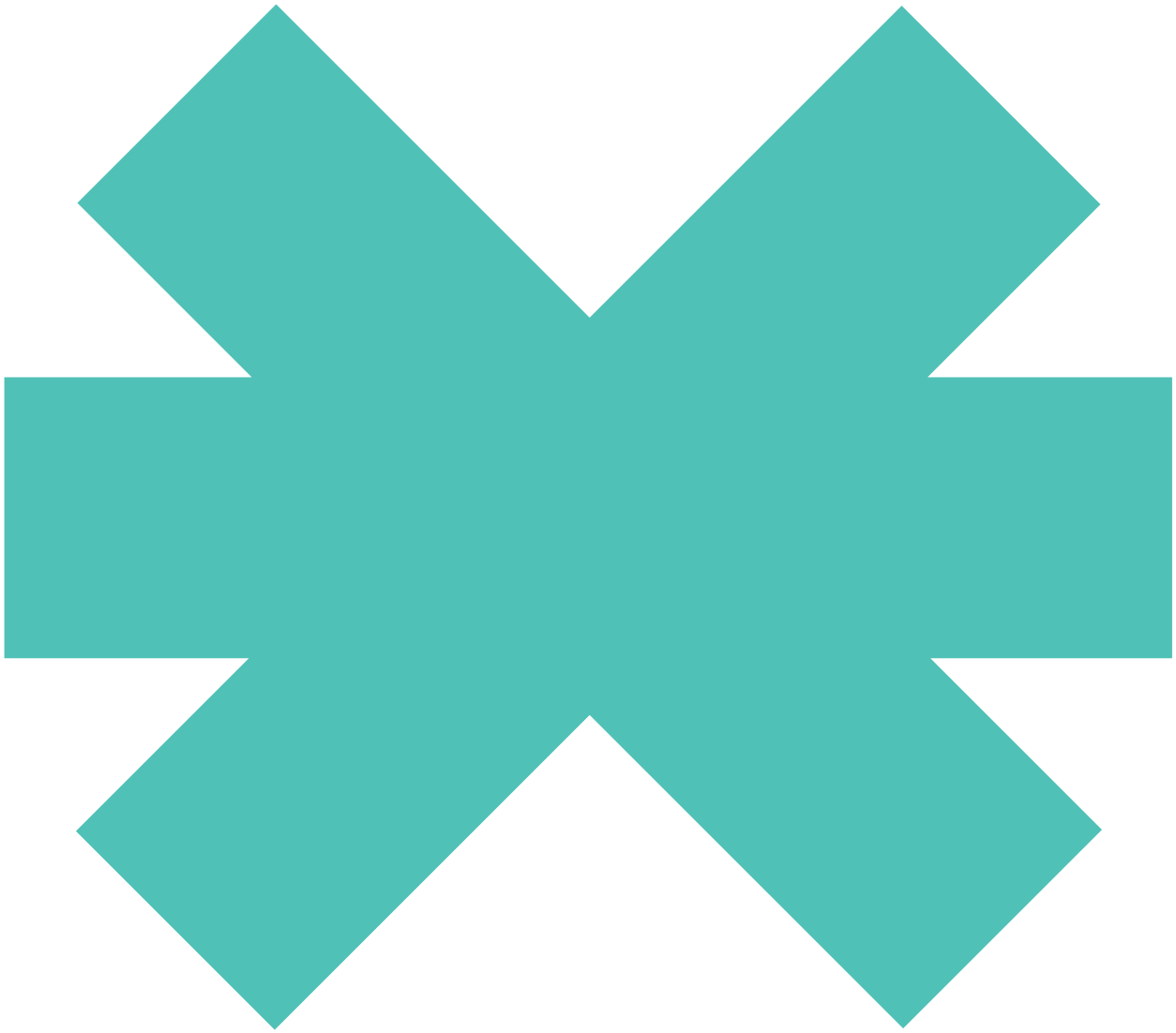
MÓDULO 2
Características básicas da linguagem
Programação Python

2

Operadores em Python

New Technology School

Tokio.



2 Operadores em Python

Sumário

Existem quatro tipos de operadores Python: **de atribuição, aritméticos, relacionais ou de comparação e lógica.**

2.1 Operadores de atribuição

Permitem atribuir um valor a uma variável, usando, para isso, o operador "=". Estes operadores permitem realizar a chamada atribuição composta, técnica que implica um código de escrita mais curto, ao mesmo tempo que são mais eficientes no tempo de execução. Esta atribuição composta realiza-se geralmente entre operadores numéricos.

Os operadores de atribuição são os seguintes:

=	Atribuição simples	Atribui o valor do operando da parte direita ao da parte esquerda x = y (atribui a x o valor de y)
+=	Adição (atribuição composta)	Soma o valor que está no operando da parte esquerda ao da direita e atribui-o ao da esquerda: x += y (x = x + y)
-=	Subtração (atribuição composta)	Subtrai e o valor que está no operando da parte esquerda ao da direita e atribui-o ao da esquerda: x -= y (x = x - y)
***=**	Multiplicação (atribuição composta)	Multiplica o valor que está no operando da parte esquerda pelo da direita e atribui-o ao da esquerda: x *= y (x = x * y)
****=**	Exponencial (atribuição composta)	Eleva o valor que está no operando da parte esquerda ao da direita e atribui-o ao da esquerda: x **= y (x = x ** y)
/=	Divisão (atribuição composta)	Divide o valor que está no operando da parte esquerda pelo da direita e atribui-o ao da esquerda: x /= y (x = x / y)
//=	Divisão inteira (atribuição composta)	Realiza a divisão inteira do valor do operando da parte esquerda pelo da direita e atribui-o ao da esquerda: x //= y (x = x // y)
%=	Módulo (atribuição composta)	Realiza a divisão inteira do valor do operando da parte esquerda pelo da direita e atribui a divisão restante ao da esquerda: x %= y (x = x % y)

Nesta imagem vemos a forma de realizar uma **atribuição simples**:

```
In [8]: # Atribuição de um valor a uma variável
        n = 3
        n
Out[8]: 3
```

Nesta imagem vemos como realizar **atribuições compostas** e a sua relação com a atribuição simples:

```
In [1]: a = 0 # stribuição, a variável 'a' passa a conter o valor 0
        a
Out[1]: 0

In [2]: a = a + 5 # a = 0 + 5
        a += 5    # o mesmo que a = a + 5
        a
Out[2]: 10

In [3]: a -= 5 # o mesmo que a = a - 5
        a
Out[3]: 5

In [4]: a *= 4 # o mesmo que a = a * 4
        a
Out[4]: 20

In [5]: a /= 2 # o mesmo que a = a / 2
        a
Out[5]: 10.0

In [6]: a %= 4 # o mesmo que a = a % 4
        a
Out[6]: 2.0

In [7]: a **= 3 # o mesmo que a = a ** 3
        a
Out[7]: 8.0
```

Também podemos realizar atribuições múltiplas, da forma mostrada seguidamente. Vemos como em "x" se guardará o 1 e em "y" se guardará o 2, através da instrução que se pode ver:

```
In [4]: x, y = 1, 2
        x, y
Out[4]: (1, 2)
```

Ao criar uma estrutura de cálculos com variáveis, podemos facilmente adaptar os seus valores para realizar verificações distintas.

```
In [28]: nota_1 = 2
         nota_2 = 5
         nota_media = (nota_1 + nota_2) / 2
         nota_media
Out[28]: 3.5
```

2.2 Operadores aritméticos

Permitem realizar qualquer operação aritmética que seja necessária, entre os operadores numéricos. Os operadores aritméticos são os seguintes, enumerados por ordem de precedência:

**	Exponenciação	x ** y (x elevado a y) [2**4 = 16]
-	Negação	- x (valor x em negativo) [- 5 = -5]
*	Multiplicação	x * y (x multiplicado por y) [3 * 2 = 6]
/	Divisão	x / y (x dividido entre y) [6 / 2 = 3]
//	Divisão inteira	x // y (quociente da divisão inteira x entre y) [7 // 2 = 3]
%	Módulo	x % y (resto da divisão inteira de x entre y) [7 // 2 = 1]
+	Soma	x + y (x mais y) [4 + 3 = 7]
-	Resto	x – y (x menos y) [5 – 3 = 2]

```
In [11]: n+3
Out[11]: 6

In [12]: n*2
Out[12]: 6

In [13]: n*n
Out[13]: 9

In [14]: m=10

In [15]: n+m
Out[15]: 13

In [16]: n*m+10
Out[16]: 40

In [17]: n=10
         m=15
         n+m
Out[17]: 25

In [18]: n=m

In [19]: n
Out[19]: 15

In [20]: m
Out[20]: 15

In [21]: n=m+10

In [22]: n
Out[22]: 25

In [23]: n=n+25

In [24]: n
Out[24]: 50
```

Nas imagens seguintes observamos distintas operações, realizadas com alguns destes operadores:

```
In [9]: 2 * 4 - (7 - 2) / 4 + 1.0
Out[9]: 7.75
```

O exemplo anterior pode ser realizado da mesma forma utilizando variáveis (o mais correto)

```
In [10]: a = 2
         b = 4
         c = 7
         d = 1.0
         a * b - (c - a) / b + d
Out[10]: 7.75
```

As divisões por zero apresentam um erro:

```
In [2]: 1 / 0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-b971f1464605> in <module>()
----> 1 1 / 0

ZeroDivisionError: division by zero
```

```
In [3]: 1.0 / 0.0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-3-d5317764bbf2> in <module>()
----> 1 1.0 / 0.0

ZeroDivisionError: float division by zero
```

Mais a diante, veremos como tratar desses erros. Quando usamos NumPy, devolverá "NaN".

A divisão entre inteiros em Python 3, devolve um número real, diferente de Python 2, onde devolve a parte inteira.

```
In [4]: 7 / 3
Out[4]: 2.3333333333333335
```

Pode-se forçar que a divisão seja inteira em Python 3 utilizando // :

```
In [5]: 7 // 3
Out[5]: 2
```

É possível elevar um número a outro utilizando * * :

```
In [6]: 2 ** 16
Out[6]: 65536
```

2.3 Operadores relacionais

Permitem-nos comparar dois ou mais valores e retornam-nos um resultado *booleano* (verdadeiro ou falso).

==	Igual a	Avalia se o valor do operando da esquerda é igual ao da direita, retornando *True* em caso afirmativo ou *False* em caso contrário. 5 == 6 (False) 6 == 6 (True)
!=	Distintos	Avalia se o valor do operando da esquerda é distinto do da direita, retornando *True* em caso afirmativo ou *False* em caso contrário. 5 != 6 (True) 6 != 6 (False)
<	Menor que	Avalia se o valor do operando da esquerda é menor que o da direita, retornando *True* em caso afirmativo ou *False* em caso contrário. 5 < 6 (True) 6 <6 (False)
>	Maior que	Avalia se o valor do operando da esquerda é maior que o da direita, retornando *True* em caso afirmativo ou *False* em caso contrário. 6 > 6 (False) 7 >6 (True)
<=	Menor ou igual a	Avalia se o valor do operando da esquerda é menor ou igual ao da direita, retornando *True* se for verdade ou *False* em caso contrário. 5 <= 6 (True) 7 <= 6 (False)
>=	Maior ou igual a	Avalia se o valor do operando da esquerda é maior ou igual ao da direita, retornando *True* se for verdade ou *False* em caso contrário. 5 >= 6 (False) 7 >= 6 (True)

Com estes operadores podemos comparar números:

```
In [8]: # Igualdade
        3 == 2
Out[8]: False

In [9]: # Diferente de
        3 != 2
Out[9]: True

In [10]: # Maior que
         3 > 2
Out[10]: True

In [11]: # Menor que
         3 < 2
Out[11]: False

In [12]: # Maior ou igual
         3 >= 2
Out[12]: True

In [13]: # Menor ou igual
         3 <= 2
Out[13]: False
```

Também podemos comparar variáveis:

```
In [15]: a = 10
         b = 5
         a > b
Out[15]: True

In [16]: b != a
Out[16]: True

In [17]: a == b*2
Out[17]: True
```

E outros tipos de dados, como listas, cadeias, o resultado de funções ou tipos de dados lógicos:

```
In [21]: "Olá" == "Olá"
Out[21]: True

In [22]: "Olá" != "Olá"
Out[22]: False

In [23]: c = "Olá"

In [24]: c[-1] == "a"
Out[24]: False

In [25]: l1 = [0,1,2]
         l2 = [2,3,4]
         l1 == l2
Out[25]: False

In [26]: len(l1) == len(l2)
Out[26]: True

In [27]: l1[-1] == l2[0]
Out[27]: True

In [28]: True == True
Out[28]: True

In [29]: False == True
Out[29]: False

In [30]: False != True
```

Como podemos verificar na imagem seguinte, a representação aritmética de *True* e *False* equivale a 1 e 0, respetivamente:

```
In [30]: True * 3
Out[30]: 3

In [31]: False / 5
Out[31]: 0.0

In [32]: True * False
Out[32]: 0
```

2.4 Operadores lógicos

Permitem-nos unir valores comparados ou negar (inverter) um valor. Retornam-nos um resultado *booleano* (verdadeiro ou falso).

&&	E (AND)	Retorna *True* se os dois operadores forem *True*, em caso contrário retorna *False*. True && True (True) True && False (False)
||	OU (OR)	Retorna *True* se algum dos operadores for verdadeiro, em caso contrário retorna *False*. True || False (True) False || False (False)
!	NÃO (NOT)	Inverte o valor do operando sobre o qual atua. ! True (False)

Nesta imagem vemos a conjunção lógica (**E** ou **AND**):

```
In [31]: True and True
Out[31]: True

In [32]: True and False
Out[32]: False

In [33]: False and True
Out[33]: False

In [34]: False and False
Out[34]: False

In [35]: a = 45
         a > 10 and a < 20
Out[35]: False

In [36]: c = "Olá Mundo"
         len(c) >= 20 and c[0] == "H"
Out[36]: False
```

Aqui a disjunção lógica (**OU** ou **OR**):

```
In [37]: True or True
Out[37]: True

In [38]: True or False
Out[38]: True

In [39]: False or True
Out[39]: True

In [40]: False or False
Out[40]: False

In [41]: c = "Outra Coisa"
         c == "Exit" or c == "Sair" or c == "Fim"
Out[41]: False

In [42]: c = "Antonio"
         c[0] == "H" or c[0] == "h"
Out[42]: False
```

Por último, a negação lógica (**NÃO** ou **NOT**):

```
In [1]: not True
Out[1]: False

In [2]: not True == False
Out[2]: True
```

A tabela de verdade dos operadores lógicos é a tabela onde são mostradas todas as combinações possíveis de operadores lógicos com dois operadores. É o resultado de aplicar operadores lógicos a duas entradas, no caso de E e de OU, e de uma no caso do NÃO. Temos de ter em conta que um **0 equivale a *False*** e um **1 a *True***.

ENTRADA		SAÍDA			
A	B	A && B	A || B	!A	!B
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	1
1	1	1	1	0	0

2.5 Precedência dos operadores

Já vimos operadores relacionais, lógicos e aritméticos. Contudo, quando se unem diferentes tipos de operadores, existem **regras de precedência**:

Primeiro, os parêntesis, porque têm prioridade máxima.

- Em segundo lugar, as expressões aritméticas, respeitando as suas próprias regras.
- Em terceiro lugar, as expressões relacionais ou de comparação.
- E em quarto, e último lugar, as expressões lógicas.

Nesta imagem vemos um exemplo desta precedência:

```
In [1]: a = 10
        b = 5
        a * b - 2**b >= 20 and not (a % b) != 0
Out[1]: False
```

2.6 Alteração de tipos de variáveis

A conversão de variáveis de um tipo para outro denomina-se de ***casting***. Em Python podemos verificar o tipo de uma variável através da função *type*(), ou também através da função *isInstance*(), como é mostrado seguidamente:

```
In [3]: a = 2.0
        b = 2
        print(type(a))
        print(type(b))

        <class 'float'>
        <class 'int'>
```

```
In [4]: print(isinstance(a, float))
        print(isinstance(a, int))
        print(isinstance(b, float))
        print(isinstance(b, int))

        True
        False
        False
        True
```

Podemos modificar o tipo de uma variável através das funções *int(), float(), complex() e str()*, como podemos ver na imagem:

```
In [5]: print(int(18.6))

        18
```

```
In [7]: print(float(1))

        1.0
```

```
In [8]: print(complex(2))

        (2+0j)
```

```
In [9]: print(str(256568))

        256568
```

```
In [3]: print(int('1234'))

        1234
```

Contudo, como todas as alterações efetuadas anteriormente são executadas imediatamente, os dados não são armazenados em nenhuma variável, pelo que depois não podemos verificar se a modificação de tipo foi a correta, com a função *type*(). Vamos fazê-lo um pouco melhor:

```
In [45]: cadena_texto = "12345"
         print("Cadeia original: ")
         print(cadena_texto)
         print("Tipo: ")
         print(type(cadena_texto))

         Cadeia original:
         12345
         Tipo:
         <class 'str'>
```

```
In [46]: cadena_num = int(cadena_texto)
         print("Cadeia com tipo modificado: ")
         print(cadena_num)
         print("Tipo: ")
         print(type(cadena_num))

         Cadeia com tipo modificado:
         12345
         Tipo:
         <class 'int'>
```

Se tentarmos realizar um *casting* de uma variável de tipo texto (*string*) para um inteiro, apenas funcionará se a cadeia de caracteres, no seu interior, contiver caracteres que possam ser transformados em números; se existirem caracteres do tipo letra, será produzido um erro:

```
In [47]: cadena_texto = "123abc"
         cadena_num = int(cadena_texto)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-47-7f984aa96500> in <module>
      1 cadena_texto = "123abc"
----> 2 cadena_num = int(cadena_texto)

ValueError: invalid literal for int() with base 10: '123abc'
```

Outras funções que podem ajudar-nos bastante na conversão e tratamento dos números são:

- ***round*(),** que nos permite arredondar um número, para o seu número inteiro mais próximo.
- ***max*(),** que nos retornará o número maior, de uma sequência de números passada como parâmetro.
- ***min*(),** que nos retornará o número menor, de uma sequência de números passada como parâmetro.

De seguida, vejamos um exemplo:

```
In [6]: print(round(18.6))
        19

In [11]: print(max(1,5,8,7))
         8

In [12]: print(min(-1,1,0))
         -1
```

2.7 Trabalhar com cadeias de caracteres

Para incluir aspas, dentro de uma cadeia de caracteres, devemos recorrer a qualquer uma das duas seguintes opções, alterar entre **aspas duplas e simples** ou utilizar o carácter de **barra invertida** ou *backslash* (\):

```
In [52]: 'Este texto inclui umas " " '
Out[52]: 'Este texto inclui umas " " '

In [53]: "Esta 'palavra' encontra-se escrita entre áspas simples"
Out[53]: "Esta 'palavra' encontra-se escrita entre áspas simples"

In [54]: "Esta \"palavra\" encontra-se escrita entre áspas duplas"
Out[54]: 'Esta "palavra" encontra-se escrita entre áspas duplas'

In [55]: 'Esta \'palavra\' encontra-se escrita entre áspas duplas'
Out[55]: "Esta 'palavra' encontra-se escrita entre áspas duplas"
```

Devemos ter em conta que, ao copiar o código de um processador de texto, devemos ter especial cuidado porque é possível que as aspas não correspondam às utilizadas em Python, pelo que o interpretador não o entenderá como uma cadeia e será produzido em erro. Este erro também pode ocorrer ao descarregar o código fonte, codificado noutra região do mundo, onde a codificação de carateres seja diferente e o carácter das aspas não seja igual ao de Portugal. Normalmente há menos problemas com as aspas duplas do que com as simples.

Uma das operações mais comuns que podem ser realizadas com cadeias é a ligação ou soma de cadeias. Podem ser realizadas com o operador +, ainda que haja várias formas de as realizar, como veremos seguidamente:

```
In [58]: c = "Isto é uma cadeia\ncom duas linhas"
         print(c)
         c+c

         Isto é uma cadeia
         com duas linhas

Out[58]: 'Isto é uma cadeia\ncom duas linhasIsto é uma cadeia\ncom duas linhas'

In [59]: print(c+c)

         Isto é uma cadeia
         com duas linhasIsto é uma cadeia
         com duas linhas

In [60]: s = "Uma cadeia" " composta por duas cadeias"
         print(s)

         Uma cadeia composta por duas cadeias

In [62]: c1 = "Uma cadeia"
         c2 = "outra cadeia"
         print("Uma cadeia " + c2)

         Uma cadeia outra cadeia
```

Também é possível realizar uma **multiplicação de cadeias**:

```
In [63]: dez_espaços = " " * 10
         print(dez_espaços + "Um texto com dez espaços")
```

```
          Um texto com dez espaços
```

Os índices permitem-nos posicionar num carácter específico de uma cadeia e aceder-lhe. Representam um número [índice] que, começando no 0, indica o carácter da primeira posição e assim sucessivamente:

```
In [64]: palavra = "Python"

In [65]: palavra[0] # caracter na primeira posição (posição 0)
Out[65]: 'P'

In [66]: palavra[3]
Out[66]: 'h'
```

Um **índice negativo** faz referência ao carácter da cadeia, começando pela última posição e em sentido inverso. Isto significa que o índice -1 será igual à última posição, o -2 à penúltima e assim sucessivamente:

```
In [67]: palavra[-1]
Out[67]: 'n'

In [68]: palavra[-0]
Out[68]: 'P'

In [69]: palavra[-2]
Out[69]: 'o'

In [70]: palavra[-6]
Out[70]: 'P'

In [71]: palavra[5]
Out[71]: 'n'
```

O slice (fatiar) é uma capacidade que as cadeias têm de retornar um subconjunto ou subcadeia, utilizando os índices [início:fim]. O primeiro índice indica onde começa a subcadeia (incluindo o carácter) e o segundo índice indica onde acaba a subcadeia (excluindo o carácter).

```
In [72]: palavra = "Python"

In [73]: palavra[0:2]
Out[73]: 'Py'

In [74]: palavra[2:]
Out[74]: 'thon'

In [75]: palavra[:2]
Out[75]: 'Py'
```

Se no *slice* não é indicado um índice, é considerado, por defeito, o princípio e o fim (incluídos).

```
In [76]: palavra[:]
Out[76]: 'Python'

In [77]: palavra[:2] + palavra[2:]
Out[77]: 'Python'

In [78]: palavra[-2:]
Out[78]: 'on'
```

Se um índice se encontrar fora do intervalo da cadeia, o interpretador do *Python* irá produzir um erro:

```
In [79]: palavra[99]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-79-7be8141a858a> in <module>
----> 1 palavra[99]

IndexError: string index out of range
```

Com o *slice* isto não acontece e o espaço é simplesmente considerado como vazio:

```
In [80]: palavra[:99]
Out[80]: 'Python'

In [81]: palavra[99:]
Out[81]: ''
```

Uma propriedade das cadeias é a **imutabilidade,** não podem ser modificadas. Se tentarmos reatribuir um carácter, não nos deixará:

```
In [82]: palavra[0] = "N"
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-82-709287fbf9a5> in <module>
----> 1 palavra[0] = "N"

TypeError: 'str' object does not support item assignment
```

Contudo, utilizando *slice* e ligação, podemos facilmente gerar novas cadeias:

```
In [83]: palavra = "N" + palavra[1:]
         palavra
Out[83]: 'Nython'
```

Uma função muito útil, que as cadeias suportam é *len*(), que nos permite saber o seu comprimento (o número de caracteres que contêm):

```
In [84]: len(palavra)
Out[84]: 6
```

2.8 A função *print()*

Para ligar cadeias dentro da função *print*, podemos fazê-lo de duas formas diferentes. Separando as cadeias **com uma vírgula**, na qual será inserido um espaço em branco entre elas:

```
In [85]: print("Uma cadeia" , "Outra cadeia")
         Uma cadeia Outra cadeia
```

Ou separá-las **com um espaço em branco**, em vez de utilizar a vírgula, desta forma não será introduzido qualquer espaço em branco entre elas:

```
In [86]: print("Uma cadeia"  "Outra cadeia")
         Uma cadeiaOutra cadeia
```

Como vimos anteriormente, a função *print*() insere uma quebra de linha cada vez que é executada. Se queremos que o Python não adicione uma quebra de linha, no final de um *print*(), definir-se-á o argumento *end* com o carácter que desejemos implementar como fim de linha:

```
In [94]: print("Uma cadeia" , end = " ")
         print("outra cadeia")
         Uma cadeia outra cadeia

In [95]: print("Uma cadeia" , end = ", ")
         print("outra cadeia")
         Uma cadeia, outra cadeia

In [96]: textoDeFinalizacao = " e para finalizar "
         print("Uma cadeia", end=f"{textoDeFinalizacao}")
         print("outra cadeia")
         Uma cadeia e para finalizar outra cadeia
```

É possível atribuir cadeias a variáveis e a forma correta de as mostrar é através da função *print*().

```
In [97]: c = "Isto é uma cadeia\ncom duas linhas"
         c
Out[97]: 'Isto é uma cadeia\ncom duas linhas'

In [98]: print(c)
         Isto é uma cadeia
         com duas linhas
```