

Bases d'algorithmique et de programmation

Sébastien Gagné, Université d'Orléans

L1 Algo - Prog - S1

Cours 02 - Opérateurs, Fonctions, Méthodes

Plan du cours

- Opérateurs arithmétiques
- Opérateurs logiques
- Opérateurs de présence
- Opérateurs d'identité
- Opérateurs de comparaison
- Opérateurs particuliers : $+$ et $*$
- Opérateur `[ ]`
- Fonctions
- Méthodes

Opérateurs arithmétiques

Les opérateurs arithmétiques python pourront être utilisés en algo : $+$, $*$, $-$, $/$, $//$, $**$, $\%$.

Ils agissent sur des int ou des float. Versions contractées $+=$, $-=$, ... disponibles en python.

<code>4.1 + 5.2</code>	<code>:</code>	<code>9.3</code>	addition
<code>- 1</code>	<code>:</code>	<code>-1</code>	changement de signe
<code>4 - 1</code>	<code>:</code>	<code>3</code>	soustraction
<code>3 * 7</code>	<code>:</code>	<code>21</code>	multiplication
<code>7 / 4</code>	<code>:</code>	<code>1.75</code>	float
<code>7 // 4</code>	<code>:</code>	<code>1</code>	quotient
<code>7 % 4</code>	<code>:</code>	<code>3</code>	reste
<code>2 ** 5</code>	<code>:</code>	<code>32</code>	puissance

Opérateurs arithmétiques

Les opérateurs arithmétiques agissent sur des valeurs de même type.

Ainsi, des opérations comme

```
>>> 3 + 4.0  
7.0
```

cachent en fait un transtypage : l'*int* 3 est converti en *float* 3.0 avant l'opération $3.0 + 4.0$.

Opérateurs particuliers // et %

- cadre naturel : les int (mais adapté aux float)
- $q = a // b$ et $r = a \% b$ revient à

$$a = b * q + r \quad \text{et} \quad 0 \leq r < b$$

- exemples :

► $139 // 8 = 17$ et $139 \% 8 = 3$

$$\begin{array}{r|l} 139 & 8 \\ 3 & 17 \end{array} \quad 139 = 17 * 8 + 3$$

► $30 // 6 = 5$ et $30 \% 6 = 0$

► $-31 // 4 = -8$ et $-31 \% 4 = 1$

Priorités

Les priorités des opérateurs arithmétiques sont, du moins prioritaire au plus prioritaire :

moins prioritaire				plus prioritaire
- (soustraction) +	* / // %	- (chgt de signe)	** (puissance)	

En cas de priorités égales, les calculs se font en respectant les règles d'associativité :

- de gauche à droite pour +, -, *, /, //,
- de droite à gauche pour - (changement de signe)

Priorités et parenthèses

Un expression entre parenthèses est évaluée en priorité supérieure.

S'il y a plusieurs niveaux de parenthèses, le calcul priorise les parenthèses les plus profondes.

Que valent les expressions suivantes ?

- $5 + 3 * 4 - 7 // 2 + 6 * 5 \% 3$
- $5 + (3 * 4 - 7) // 2 + (6 * 5) \% 3$
- $((5 + 3) * 4 - 7 // (2 + 6)) * 5 \% 3$
- $(5 + 3) * 4 - 7 // (((2 + 6) * 5) \% 3)$

Opérateurs logiques

En python, les opérateurs logiques classiques sont **and** (ET), **or** (OU) et **not** (NON)

x	not x
True	False
False	True

x	y	x or y	x and y
True	True	True	True
True	False	True	False
False	True	True	False
False	False	False	False

Priorité des opérations

not > and > or

Opérateurs de présence - python

L'opérateur python *in* permet de dire si une valeur est dans un itérable. Il peut être utilisé avec *not* (pour donner *not in*).

```
2 in (3, 2, 5)           : True      # tuple
3 in [0, 1, 2]           : False     # list
5 in range(8)            : True      # range
5 not in range(2, 10, 2) : True      # range
5 in {'x' : 3, 'y' : 5}  : False     # dict
'x' in {'x' : 3, 'y' : 5} : True     # dict
'a' in 'bcd'             : False     # str
'a' not in 'bcd'         : True      # str
'ab' in 'ebcabd'         : True      # str
```

L'opérateur *in* dit aussi si une chaîne est sous-chaîne d'une autre. Par contre, il ne dit pas si un tuple (ou une list) est extrait d'un tuple (d'une list).

Opérateurs d'identité - python

L'opérateur *is* indique si deux variables (ou deux valeurs) référencent le même emplacement mémoire.

(a is b)	revient à	(id(a) == id(b))
(a == b)	ne revient pas à	(a is b)
(a is b)	implique	(a == b)

```
a = 1, b = 1 : a is b vaut True
l1 = [1, 2, 3], l2 = [1, 2, 3] : l1 is l2 vaut False
l1 = [1, 2, 3], l2 = [1, 2, 3] : l1 == l2 vaut True
id(l1) = 127523933853888
id(l2) = 127523932317504
```

Remarque 1 : **pour les int**, la gestion spéciale de la mémoire en python permet de dire que : ***a == b*** revient à : ***a is b***.

Remarque 2 : on peut utiliser aussi *is not*. La syntaxe python assimile *a is not b* et *not(a is b)*.

Opérateurs de comparaison

Les opérateurs `<`, `<=`, `>`, `>=`, `==` et `!=` permettent de comparer les valeurs de deux variables.

On peut aussi comparer des chaînes de caractères.

```
1 < 4           : True
2 > 3           : False
1 <= 1          : True
2 == 1 + 1      : True
1 != 2          : True
1 < 4 < 6       : True      teste 1 < 4 and 4 < 6
1 < 7 < 6       : False     teste 1 < 7 and 7 < 6
1 < 7 > 6       : True      teste 1 < 7 and 7 > 6
1 < 7 > 9       : False     teste 1 < 7 and 7 > 9
'avant' < 'après' : False   car 'p' < 'v'
'a' <= 'm' <= 'z' : True    'm' est une minuscule
'A' <= 'P' <= 'Z' : True    'P' est une majuscule
'0' <= '2' <= '9' : True    '2' est un caractère
                        représentant un chiffre
```

Comparaison de caractères

Chaque caractère correspond en fait à un code ASCII (American Standard Code for Information Interchange).

Python compare deux caractères en comparant les deux int qui les identifient dans la table ASCII.

Par exemple, 'a' correspond à 97 et '2' à 50.
Comme $50 < 97$, python évalue 'a' < '2' à False

Python permet d'accéder à la valeur du code ASCII par la fonction ord (la fonction inverse est chr).

```
>>> 'a' < '2'  
False  
>>> ord('a')  
97  
>>> ord('2')  
50
```

```
>>> chr(98)  
'b'  
>>> chr(66)  
'B'  
>>> 'B' < 'b'  
True
```

Tableau ASCII

Dec	Hex	char		Dec	Hex	char	Dec	Hex	char	Dec	Hex	char
0	00	NUL	<i>caract. null</i>	32	20	espace	64	40	@	96	60	'
1	01	SOH		33	21	!	65	41	A	97	61	a
2	02	STX		34	22	"	66	42	B	98	62	b
3	03	ETX		35	23	#	67	43	C	99	63	c
4	04	EOT		36	24	\$	68	44	D	100	64	d
5	05	ENQ		37	25	%	69	45	E	101	65	e
6	06	ACK		38	26	&	70	46	F	102	66	f
7	07	BEL	<i>bell</i>	39	27	'	71	47	G	103	67	g
8	08	BS	<i>backspace</i>	40	28	(	72	48	H	104	68	h
9	09	TAB	<i>tabul. horiz</i>	41	29	)	73	49	I	105	69	i
10	0A	LF	<i>line feed</i>	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	<i>Tabul. vertic</i>	43	2B	+	75	4B	K	107	6B	k
12	0C	FF		44	2C	,	76	4C	L	108	6C	l
13	0D	CR	<i>retour ligne</i>	45	2D	-	77	4D	M	109	6D	m
14	0E	SO		46	2E	.	78	4E	N	110	6E	n
15	0F	SI		47	2F	/	79	4F	O	111	6F	o
16	10	DLE		48	30	0	80	50	P	112	70	p
17	11	DC1		49	31	1	81	51	Q	113	71	q
18	12	DC2		50	32	2	82	52	R	114	72	r
19	13	DC3		51	33	3	83	53	S	115	73	s
20	14	DC4		52	34	4	84	54	T	116	74	t
21	15	NAK		53	35	5	85	55	U	117	75	u
22	16	SYN		54	36	6	86	56	V	118	76	v
23	17	ETB		55	37	7	87	57	W	119	77	w
24	18	CAN		56	38	8	88	58	X	120	78	x
25	19	EM		57	39	9	89	59	Y	121	79	y
26	1A	SUB		58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	<i>escape</i>	59	3B	;	91	5B	[	123	7B	{
28	1C	FS		60	3C	<	92	5C	\	124	7C	
29	1D	GS		61	3D	=	93	5D	]	125	7D	}
30	1E	RS		62	3E	>	94	5E	^	126	7E	~
31	1F	US		63	3F	?	95	5F	_	127	7F	DEL

Tableau de priorités

plus prioritaire



moins prioritaire

opération	description	associativité
**	élévation à une puissance	droite à gauche
-	changement de signe	droite à gauche
*	multiplication	gauche à droite
/	division	
//	division entière	
%	modulo	
+	addition	gauche à droite
-	soustraction	
<	infériorité stricte	gauche à droite
≤	infériorité large	
>	supériorité stricte	
≥	supériorité large	
==	égalité	
!=	différence	
in	présence dans ...	
not in	non présence dans ...	
is	comparaison	
is not	différence	
not	négation	gauche à droite
and	conjonction	gauche à droite
or	inclusion	gauche à droite

Opérateurs particuliers

Les opérateurs `+` et `*` agissent sur les types numériques, mais aussi sur d'autres types : *str*, *tuple* et *list*.

```
type str
    "abc" + "de"           "abcde"
    "abc" * 3              "abccabccabc"
type tuple
    (1, 4, 5) + (2, 3)     (1, 4, 5, 2, 3)
    (1, 4, 5) * 2          (1, 4, 5, 1, 4, 5)
type list
    [1, 4, 5] + [2, 3]     [1, 4, 5, 2, 3]
    [1, 4, 5] * 2          [1, 4, 5, 1, 4, 5]
```

Les opérateurs `+` et `*` prennent dans ce cas un sens totalement différent de l'addition et de la multiplication numériques.

L'opérateur d'extraction []

- L'opérateur [] accède à un élément particulier d'une valeur itérable, donnée par l'indice fourni.
- Pour des indices **positifs**, de 0 à taille - 1 (lecture de gauche à droite).

indices positifs	0	1	2	3	4	5	6
chaîne = B o n j o u r							
indices négatifs	-7	-6	-5	-4	-3	-2	-1

- Pour des indices **négatifs**, de - taille à - 1 (lecture de gauche à droite).
- En dehors de ces deux plages : erreur.

L'opérateur d'extraction []

```
1 str1 = 'Programme'
2 range1 = range(4, 48, 7)
3 tuple1 = ('a', 'b', 'c', 'd')
4 list1 = [3.14, False, 'Bye', [1, 3], ('a', 0, True, 'toto')]
5
6 print(str1[0])
7 print(str1[3])
8 print(range1[4])
9 print(tuple1[2])
10 print(list1[4])
11
12 print(str1[-7])
13 print(str1[-1])
14 print(range1[-5])
15 print(tuple1[-3])
16 print(list1[-2])
17
18 print(str1[9])
19 print(tuple1[-5])
```

Qu'affiche ce programme, ligne par ligne ?

Extraction avec []

Extraction d'un "sous-itérable" de même type :

`<iterable>[<debut> : <fin>]`

- l'élément d'indice *debut* est inclus ;
- l'élément d'indice *fin* est exclu.

L'extraction se fait toujours de gauche à droite.

```
1 str1 = 'Bonjour'
2 list1 = [3.14, False, 'Bye', [1, 3], ('a', 0, True, 'toto')]
3
4 print(str1[1 : 4])
5 print(str1[-5 : 5])
6 print(str1[-4 : -2])
7
8 print(list1[1 : 3])
9 print(list1[-3 : 4])
10 print(list1[-4 : -1])
```

Extraction avec []

Si une borne est manquante, cela revient à "tout jusqu'à" ou à "tout à partir de".

```
1 str1 = 'Algorithme'
2
3 print(str1[3 :])    # à partir de l'indice 3 (inclus)
4 print(str1[: 5])    # jusqu'à l'indice 5 (exclus)
5
6 print(str1[-4 :])   # à partir de l'indice -4 (inclus)
7 print(str1[: -6])   # jusqu'à l'indice -6 (exclus)
```

Extraction avec []

Contrairement à l'accès à un élément particulier, qui ne tolère pas les dépassements de bornes, une extraction peut être faite avec une borne hors-limite.

```
1 str1 = 'Algorithme'
2
3 print(str1[4 : 17])
4 print(str1[-11 : 6])
5 print(str1[-11 : 17])
```

Extraction avec []

Extraction d'un "sous-itérable" de même type :

$\langle iterable \rangle [\langle debut \rangle : \langle fin \rangle : pas]$

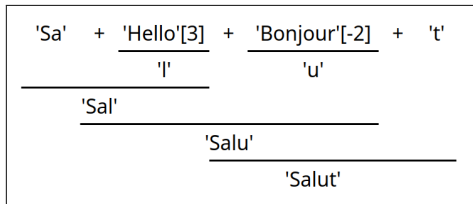
Le principe est le même que pour **range**, à savoir :

- l'élément d'indice *debut* est inclus ;
- l'élément d'indice *fin* est exclu ;
- les éléments extraits commencent à l'indice *debut*, avec un pas égal à la valeur *pas*.

```
1 chaîne = "abcdefghijklmnpqrstuvwxy"
2
3 print(chaine[1:12:2])
4 print(chaine[2:22:4])
```

Priorité et associativité avec []

- Priorité supérieure à tout ce qui a été vu avant



- Associativité de gauche à droite

```
1 list1 = [3.14, 'Bye', ('a', [0, 2, 7, 8, 4, 1, 6, 3], True)]
2
3 print(list1[2])
4 print(list1[2][1])
5 print(list1[2][1][3 : 6])
6 print(list1[2][1][3 : 6][1])
```

Fonctions – des exemples

Python propose des *fonctions* utilisables à tout moment, par l'import (ou non) d'une bibliothèque spécifique. Exemples :

- la fonction **print** qui affiche sur la sortie ce qui lui est passé en argument ;
- la fonction **input** qui affiche la chaîne de caractère passée en argument, et récupère la chaîne saisie par l'utilisateur ;
- la fonction **len** qui retourne la longueur de l'objet itérable en argument ;
- la fonction **int** qui retourne la conversion de l'argument sous forme d'**int** (quand c'est possible) ;
- la fonction **float** qui retourne la conversion de l'argument sous forme de **float** (quand c'est possible) ;

Fonctions – des exemples

- la fonction **str** qui retourne sous forme de chaîne de caractères la valeur en argument ;
- la fonction **sqrt** qui retourne la racine carrée du float en argument (de la bibliothèque **math**) ;
- la fonction **randint** qui retourne un int aléatoire entre les bornes passées en arguments (bibliothèque **random**) ;
- la fonction **random** qui retourne un float aléatoire entre 0 et 1 (bibliothèque **random**) ;

Fonctions – des exemples

- les fonctions **min** et **max** qui retournent la valeur minimum (ou maximum) d'une série d'éléments tous comparables ;
- la fonction **abs** qui retourne la valeur absolue d'un nombre ;
- la fonction **round** qui retourne l'arrondi d'un nombre à un certain nombre de décimales ;
- les fonctions **sorted** qui prend en argument un itérable composé d'éléments comparables, et qui retourne une liste de ces éléments, triés par ordre croissant. La liste initiale n'est pas modifiée ;

Fonctions

Vous définirez vos propres fonctions pour optimiser votre code.

Une maladresse classique : la répétition de code, qui contredit un principe fondamental :

Don't Repeat Yourself

```
1 print("=====")
2 print("Ceci est le premier paragraphe")
3 print("=====")
4 print("Ceci est le deuxième paragraphe")
5 print("=====")
```

```
=====
Ceci est le premier paragraphe
=====
Ceci est le deuxième paragraphe
=====
```

Fonctions

- On peut isoler du code dans une **fonction**.
- Une fonction est définie après le mot-clé **def**.
- Elle a un **nom**, un **identifiant**, un **type**.
- Elle peut ou non utiliser des **arguments**, qui sont précisés entre les **parenthèses** qui suivent le nom de la fonction.
- Elle est appelée à tout endroit du programme par l'évocation de son nom.
- Elle effectue une série d'opérations, et **retourne** ou non un **résultat**.

Fonctions

Retour sur l'exemple précédent :

```
1 # centralisation du code dans une fonction
2 def separation() :
3     print("=====")
4
5
6 # appels de la fonction
7 separation()
8 print("Ceci est le premier paragraphe")
9 separation()
10 print("Ceci est le deuxième paragraphe")
11 separation()
```

Cette fonction ne prend **pas d'argument**, et ne retourne **aucun résultat**. Elle ne fait qu'afficher une chaîne de caractères qu'elle a construite.

Fonctions

```
1 # fonction recevant des arguments et ne retournant rien
2 def separation(motif, n) :
3     print(motif * n)
4
5 # utilisation de la fonction dans le programme principal
6 separation("#", 15)
7 print("Ceci est le premier paragraphe")
8 separation("<=>", 6)
9 print("Ceci est le deuxième paragraphe")
10 separation("/", 15)
```

Cette fonction prend **des arguments**, et ne retourne **aucun résultat**. Elle affiche une chaîne construite avec les arguments.

```
#####
Ceci est le premier paragraphe
<=><=><=><=><=><=>
Ceci est le deuxième paragraphe
////////////////
```

Fonctions

```
1 # fonction ne recevant aucun argument et retournant un résultat
2 def carte() :
3     couleurs = ('coeur', 'pique', 'carreau', 'trèfle')
4     valeurs = ('7', '8', '9', '10', 'valet', 'dame', 'roi', 'as')
5     c = randint(0, 3)
6     v = randint(0, 7)
7     carte_tiree = valeurs[v] + ' de ' + couleurs[c]
8     return carte_tiree
9
10 # utilisation de la fonction dans le programme principal
11 from random import randint
12 print("la carte tirée est : " + carte())
```

Cette fonction ne prend **pas d'argument**, et **retourne un résultat** sous forme de chaîne de caractères.

```
la carte tirée est : 8 de coeur
```

Fonctions

```
1 # fonction recevant un argument et retournant un résultat
2 def carte(n) :
3     couleurs = ('coeur', 'pique', 'carreau', 'trèfle')
4     if n == 32 :
5         valeurs = ('7', '8', '9', '10', 'valet', 'dame', 'roi', 'as')
6     elif n == 52 :
7         valeurs = ('2', '3', '4', '5', '6', '7', '8', '9', '10', 'valet', 'dame', 'roi', 'as')
8     c = randint(0, 3)
9     v = randint(0, len(valeurs) - 1)
10    carte_tiree = valeurs[v] + ' de ' + couleurs[c]
11    return carte_tiree
12
13 # utilisation de la fonction dans le programme principal
14 from random import randint
15 print("la carte tirée est : " + carte(32))
16 print("la carte tirée est : " + carte(52))
```

Cette fonction prend **un argument**, et retourne un **résultat** sous forme de chaîne de caractères.

```
la carte tirée est : as de coeur
la carte tirée est : 3 de carreau
```

Intérêt des fonctions

Toutes les situations faisant intervenir des paramètres et fournissant des résultats vont naturellement pouvoir se traduire en fonctions.

Cela rappellera souvent le concept mathématique des fonctions, mais en élargissent le contexte d'utilisation.

Exemple mathématique : modéliser par une fonction comment on peut calculer la surface d'un rectangle de périmètre constant égal à 100.

Intérêt des fonctions



→ La contrainte du périmètre impose la relation $2x + 2y = 100$ et donc $x + y = 50$.

→ y s'exprime en fonction de x : $y = 50 - x$.

→ L'aire est $x * y$, donc $aire = x * (50 - x)$.

Intérêt des fonctions

Cette situation met en évidence :

- Une variable, note ici x , qui représente la longueur du côté horizontal du rectangle.
- Deux fonctions, qui permettent de calculer, à partir de x , la longueur y et l'aire du rectangle.

On pourra alors utiliser les écritures

- $y(x) = 50 - x$
- $aire(x) = x * (50 - x)$

Traduction en python

```
1 def y(x) :  
2     return 50 - x  
3  
4 def aire(x) :  
5     return x * (50 - x)  
6  
7 def afficher_resultat(x) :  
8     print("pour x =", x)  
9     print("- y vaut", y(x))  
10    print("- l'aire vaut", aire(x))  
11  
12 # programme principal  
13 afficher_resultat(7)  
14 afficher_resultat(19)  
15 c = int(input("entrez votre valeur de x : "))  
16 afficher_resultat(c)
```

- **def** → définition de la fonction : **Obligatoire**.
- **return** → renvoie un résultat à l'endroit de l'appel de la fonction : **Facultatif**. Toute instruction après **return** est ignorée !

Traduction en python

```
1 def y(x) :  
2     return 50 - x  
3  
4 def aire(x) :  
5     return x * (50 - x)  
6  
7 def afficher_resultat(x) :  
8     print("pour x =", x)  
9     print("- y vaut", y(x))  
10    print("- l'aire vaut", aire(x))  
11  
12 # programme principal  
13 afficher_resultat(7)  
14 afficher_resultat(19)  
15 c = int(input("entrez votre valeur de x : "))  
16 afficher_resultat(c)
```

```
pour x = 7  
- y vaut 43  
- l'aire vaut 301  
pour x = 19  
- y vaut 31  
- l'aire vaut 589  
entrez votre valeur de x : 15  
pour x = 15  
- y vaut 35  
- l'aire vaut 525
```

Méthodes

Une **méthode** ressemble de près à une fonction

- Elle peut utiliser des arguments ;
- Elle peut retourner un résultat ;
- Mais elle est appelée par la **notation pointée**, sur l'objet auquel elle appartient ;
- Et elle est de type *method*.

```
>>> import random
>>> type(random.randint)
<class 'method'>
>>> random.randint(1, 49)
31
```

Quelques méthodes sur les *str*

Voici quelques méthodes qui s'appliquent aux chaînes de caractères. Il y en a d'autres...

<code>ch.lower()</code>	: retourne la chaîne convertie en minuscules
<code>ch.upper()</code>	: retourne la chaîne convertie en majuscules
<code>ch.islower()</code>	: retourne True si ch est en minuscules
<code>ch.isupper()</code>	: retourne True si ch est en majuscules
<code>ch.find(ch2)</code>	: retourne l'indice de la première occurrence de ch2 dans ch (retourne -1 si ch2 n'est pas trouvé dans ch)
<code>ch.split(' ')</code>	: découpe ch avec le séparateur ' ', et renvoie la liste correspondante
<code>ch.replace('o', '-')</code>	: retourne la chaîne où tous les 'o' sont remplacés par des '-'
<code>ch.count('a')</code>	: retourne le nombre de caractères 'a' dans la chaîne ch
<code>"{0} et {1}".format('toto', 'tata')</code> : remplace {0} par 'toto' et {1} par 'tata'	

Quelques méthodes sur les *str*

```
1 chaine = "Hello World !"
2 chaine_min = chaine.lower()
3 chaine_maj = chaine.upper()
4
5 message_generique = "Bonjour {0}, comment va {1} ?"
6 message_1 = message_generique.format('Antoine', 'Romain')
7 message_2 = message_generique.format('Gaël', 'Thomas')
8
9 print(chaine)
10 print(chaine_min)
11 print(chaine_maj)
12
13 print(message_generique)
14 print(message_1)
15 print(message_2)
```

```
Hello World !
hello world !
HELLO WORLD !
Bonjour {0}, comment va {1} ?
Bonjour Antoine, comment va Romain ?
Bonjour Gaël, comment va Thomas ?
```

Quelques méthodes sur les *list*

- `list1.append(n)` accole la valeur `n` en fin de *list1* ;
- `list1.remove(m)` supprime la 1ère occurrence de *m* ;
- `list1.pop()` supprime le dernier élément et le retourne ;
- `list1.reverse()` renverse la liste ;
- `list1.sort()` trie la liste ;

après sa création,	<code>liste = [4, 7, 3, 1, 7]</code>
après <code>liste.append(2)</code> ,	<code>liste = [4, 7, 3, 1, 7, 2]</code>
après <code>liste.remove(7)</code> ,	<code>liste = [4, 3, 1, 7, 2]</code>
après <code>dernier = liste.pop()</code> ,	<code>liste = [4, 3, 1, 7]</code> <code>dernier = 2</code>
après <code>liste.sort()</code> ,	<code>liste = [1, 3, 4, 7]</code>
après <code>liste.reverse()</code> ,	<code>liste = [7, 4, 3, 1]</code>