La notion d'opérateur adoptée dans ce chapitre est très large :

```
x + k          # addition (concaténation)
not x          # négation
x[k]           # accès à l'item de clé/indice k
x(k)           # appel de x avec passage de k en argument
[x, k, v]      # agencement d'items dans une liste
lambda x: k    # fonction lambda
etc.
```

1 Arité

Python possède des opérateurs d'arité 2, comme l'addition, par exemple :

```
>>> 5 + 3    # arité 2, ie 2 opérandes
8
```

et des opérateurs d'arité 1, comme la négation :

```
>>> not True     # arité 1, ie 1 opérande
False
```

Il possède aussi des opérateurs d'arité indéterminée. L'opérateur d'agencement permettant d'écrire une liste, par exemple :

```
[5, 7, 1, 0]
```

n'a pas d'arité prédéfinie. Dans l'exemple ci-dessus, il est d'arité 4, mais on peut l'écrire avoir n'importe quelle arité, comme 2 :

```
[5, 7]
```

ou zéro :

```
[]
```

2 Chaînage et sens de regroupement

Un opérateur d'arité 2 possédant une syntaxe comme celle de l'addition :

```
x ✠ y
```

permet le chaînage, c'est-à-dire l'écriture d'expressions où l'on répète l'opération :

```
a ✠ b ✠ c ✠ d ✠ e
```

Dans un chaînage, il existe 2 sens de regroupement des opérandes :

```
# de gauche à droite :
(((a ✠ b) ✠ c) ✠ d) ✠ e
# de droite à gauche :
a ✠ (b ✠ (c ✠ (d ✠ e)))
```

La plupart du temps, Python regroupe de gauche à droite :

```
>>> 4 - 10 - 3 - 1 # regr de gauche à droite
-10
>> 4 - (10 - (3 - 1)) # reg de droite à gauche
-4
```

mais il existe des exceptions, comme avec l'exponentiation :

```
>>> 5 ** 2 ** 3   # regr de droite à gauche
390625
>>> (5 ** 2) ** 3 # regr de gauche à droite
15625
```

Notons que l'opérateur d'accès à un item (par indice ou par clé)

```
a[k]
```

et l'opérateur d'appel

```
a(k)
```

entrent dans la catégorie des opérateurs d'arité 2 chaînables qui ne laissent pas le choix dans le sens du regroupement — impossible de faire autrement (de par la syntaxe elle-même) que d'aller de la gauche vers la droite :

```
(((a[b])[c])[d])[e]
```

(le parenthésage de droite à gauche n'aurait aucune signification ici !)

Nous donnons plus loin le sens de regroupement des opérateurs d'arité 2 chaînables.

Notons qu'il existe des opérateurs d'arité 2 dont la syntaxe empêche le chaînage. Pour eux, le sens de regroupement n'a aucune signification. C'est le cas de l'opérateur `lambda`, par exemple :

```
lambda a: b     # opérandes : a et b
```

dont la syntaxe correspond à la notation fonctionnelle `lambda(a,b)`. On rappelle la signification des deux opérandes ici :

```
lambda <paramètres>: <expression>
```

On peut faire du chaînage avec un opérateur ∇ d'arité 1 comme ceci :

```
∇∇∇∇∇∇∇∇x
```

mais la question du sens de regroupement ne se pose pas ici :

```
∇(∇(∇(∇(∇(∇(∇(∇x)))))))    # pas le choix !
```

3 Fonction `reduce`

La fonction `reduce` n'existe plus comme fonction interne depuis la version 3 de Python mais on la trouve dans le module `functools` :

```
>>> from functools import reduce
```

Soit `f` est une fonction d'arité 2 et notons `a` ✠ `b` la valeur retournée par `f(a,b)`. Alors un appel `reduce(f, [a,b,c,d,e])`, par exemple, retourne `a` ✠ `b` ✠ `c` ✠ `d` ✠ `e` où le calcul est effectué de gauche à droite :

```
>>> def f(x,y): return x / y
>>> reduce(f, [5, 2, 0.4, 3, 10])
0.20833333333333334
```

4 Ordre de priorité

Il est important de connaître l'ordre dans lequel Python exécute les opérations lorsqu'il évalue une expression :

```
>>> 2 + 5 * 10
52
>>> (2 + 5) * 10
```

```
70
```

Le tableau ci-dessous liste les opérateurs du langage Python, du plus prioritaire au moins prioritaire. Les opérateurs de même ordre de priorité sont mis dans la même case.

Nous indiquons le sens de regroupement (s'il y a lieu) et l'arité de chaque opérateur.

//Number 1 : (inside 'center')

Catégorie	**Opérateurs**	**Arité**	**Sens de regroup.**

Agencement	`(a,b,c)`
	`[a,b,c]`
	`{a,b,c}`
	`{a:b, c:d)`
		(*)	

Accès
et appels
	`a[b]`
	`a[b:c]`
	`a(b)`
	`a.b`
		2	$\longrightarrow$

	`await x`	1	
Exponentiation	`a ** b`	2	$\longleftarrow$

positif, opposé, inverse	`+a`
	`-a`
	`~a`
		1	

Multiplication
et cousines
	`a * b`
	`a @ b`
	`a / b`
	`a // b`
	`a % b`
		2	$\longrightarrow$

Addition et soustraction	`a + b` `a - b`	2	$\longrightarrow$
Décalages	`a << b` `a >> b`	2	$\longrightarrow$
Conjonction bit à bit	`a & b`	2	$\longrightarrow$
Disjontion exclusive bit à bit	`a ^ b`	2	$\longrightarrow$
Disjonction bit à bit	`a | b`	2	$\longrightarrow$

Comparaisons,
appartenance,
identité
	`a == b`
	`a != b`
	`a < b`
	`a > b`
	`a <= b`
	`a >= b`
	`a is b`
	`a is not b`
	`a in b`
	`a not in b`
		2	$\longrightarrow$

Négation	`not a`	1	
Conjonction	`a and b`	2	$\longrightarrow$
Disjonction	`a or b`	2	$\longrightarrow$
Expression `if-else`	`a if b else c`	3	
Fonction lambda	`lambda a:b`	2	
Expression-affectation (walrus)	`a := b`	2	(**)
Affectation	`a = b`	2	$\longleftarrow$

Affectation-opération	`a += b`
	`a -= b`
	`a *= b`
	etc.
		2	(**)

// Number 2 : inside a Text environement, inside a math displayed formula

Catégorie	**Opérateurs**	**Arité**	**Sens de regroup.**

Agencement	`(a,b,c)`
	`[a,b,c]`
	`{a,b,c}`
	`{a:b, c:d)`
		(*)	

Accès
et appels
	`a[b]`
	`a[b:c]`
	`a(b)`
	`a.b`
		2	$\longrightarrow$

	`await x`	1	
Exponentiation	`a ** b`	2	$\longleftarrow$

positif, opposé, inverse	`+a`
	`-a`
	`~a`
		1	

Multiplication et cousines	`a * b`
	`a @ b`
	`a / b`
	`a // b`
	`a % b`
		2	⟶

Addition et soustraction	`a + b` `a - b`	2	⟶
Décalages	`a << b` `a >> b`	2	⟶
Conjonction bit à bit	`a & b`	2	⟶
Disjontion exclusive bit à bit	`a ^ b`	2	⟶
Disjonction bit à bit	`a | b`	2	⟶

Comparaisons, appartenance, identité	`a == b`
	`a != b`
	`a < b`
	`a > b`
	`a <= b`
	`a >= b`
	`a is b`
	`a is not b`
	`a in b`
	`a not in b`
		2	⟶

Négation	`not a`	1	
Conjonction	`a and b`	2	⟶
Disjonction	`a or b`	2	⟶
Expression `if-else`	`a if b else c`	3	
Fonction lambda	`lambda a:b`	2	
Expression-affectation (walrus)	`a := b`	2	(**)
Affectation	`a = b`	2	⟵

Affectation-opération	`a += b`
	`a -= b`
	`a *= b`
	etc.
		2	(**)

(*) Il est évident que les opérateurs d'agencement sont prioritaires : Python ne peut faire autrement puisqu'ils sont mis entre parenthèses (ou l'équivalent de parenthèses). L'arité dépend du nombre d'items. Ci-dessous, par exemple

```
>>> x = (5, 7, 8, 1)
```

nous avons écrit un opérateur `()` d'arité 4.

(**) Il est interdit d'enchaîner des opérateurs walrus `:=`. Même chose avec les 12 opération-affectations telles que `+=` (il existe 12 opérateurs *in place*).

Nous avons omis l'opérateur virgule `,`. Il s'agit d'un opérateur binaire. On l'utilise pour agencer des items dans un uplet :

```
>>> x = 5, 1, 9, 0
>>> x
(5, 1, 9, 0)
```

Elle n'est prioritaire que devant l'affectation `=`.

Nous n'avons pas évoqué l'opérateur complexe `j` permettant d'écrire des objets de type `complex`. Il est évident que dans une expression, tout fragment comme `7j`, par exemple, reçoit la priorité.