Types natifs

Les sections suivantes décrivent les types standards intégrés à l'interpréteur.

Les principaux types natifs sont les numériques, les séquences, les dictionnaires, les classes, les instances et les exceptions.

Certaines classes de collection sont muables. Les méthodes qui ajoutent, retirent, ou réorganisent leurs éléments sur place, et qui ne renvoient pas un élément spécifique, ne renvoient jamais l'instance de la collection elle-même, mais `None`.

Certaines opérations sont prises en charge par plusieurs types d'objets; en particulier, pratiquement tous les objets peuvent être comparés, testés (valeur booléenne), et convertis en une chaîne (avec la fonction `repr()` ou la fonction légèrement différente `str()`). Cette dernière est implicitement utilisée quand un objet est écrit par la fonction `print()`.

Valeurs booléennes

Tout objet peut être comparé à une valeur booléenne, typiquement dans une condition `if` ou `while` ou comme opérande des opérations booléennes ci-dessous.

Par défaut, tout objet est considéré vrai à moins que sa classe définisse soit une méthode `__bool__()` renvoyant `False` soit une méthode `__len__()` renvoyant zéro lorsqu'elle est appelée avec l'objet. [1] Voici la majorité des objets natifs considérés comme étant faux :

- les constantes définies comme étant fausses : `None` et `False`.
- zéro de tout type numérique: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- les chaînes et collections vides : `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

Les opérations et fonctions natives dont le résultat est booléen donnent toujours `0` ou `False` pour faux et `1` ou `True` pour vrai, sauf indication contraire. (Exception importante : les opérations booléennes `or` et `and` renvoient toujours l'une de leurs opérandes.)

Opérations booléennes --- `and`, `or`, `not`

Ce sont les opérations booléennes, classées par priorité ascendante :

Opération	Résultat	Notes
`x or y`	si *x* est faux, alors *y*, sinon *x*	(1)
`x and y`	si *x* est faux, alors *x*, sinon *y*	(2)
`not x`	si *x* est faux, alors `True`, sinon `False`	(3)

Notes :

1. Ceci est un opérateur court-circuit : il n'évalue le deuxième argument que si le premier est faux.
2. Ceci est un opérateur court-circuit, il n'évalue le deuxième argument si le premier est vrai.
3. `not` a une priorité inférieure à celle des opérateurs non-booléens, donc `not a == b` est interprété comme `not (a == b)` et `a == not b` est une erreur de syntaxe.

Comparaisons

Il y a huit opérations de comparaison en Python. Elles ont toutes la même priorité (qui est supérieure à celle des opérations booléennes). Les comparaisons peuvent être enchaînées arbitrairement; par exemple, `x < y <= z` est équivalent à `x < y and y <= z`, sauf que *y* n'est évalué qu'une seule fois (mais dans les deux cas *z* n'est pas évalué du tout quand `x < y` est faux).

Ce tableau résume les opérations de comparaison :

Opération	Signification
<	strictement inférieur

Opération	Signification
`<=`	inférieur ou égal
`>`	strictement supérieur
`>=`	supérieur ou égal
`==`	égal
`!=`	différent
`is`	identité d'objet
`is not`	contraire de l'identité d'objet

Les objets de différents types, à l'exception de différents types numériques, ne peuvent en aucun cas être égaux. En outre, certains types (par exemple, les objets fonction) ne gèrent qu'une une notion dégénérée de la comparaison où deux objets de ce type sont inégaux. Les opérateurs `<`, `<=`, `>` et `>=` lèvent une exception `TypeError` lorsqu'on compare un nombre complexe avec un autre type natif numérique, lorsque les objets sont de différents types qui ne peuvent pas être comparés, ou dans d'autres cas où il n'y a pas d'ordre défini.

Des instances différentes d'une classe sont normalement considérées différentes à moins que la classe ne définisse la méthode `__eq__()`.

Les instances d'une classe ne peuvent pas être ordonnées par rapport à d'autres instances de la même classe, ou d'autres types d'objets, à moins que la classe ne définisse suffisamment de méthodes parmi `__lt__()`, `__le__()`, `__gt__()` et `__ge__()` (en général, `__lt__()` et `__eq__()` sont suffisantes, si vous voulez les significations classiques des opérateurs de comparaison).

Le comportement des opérateurs `is` et `is not` ne peut pas être personnalisé ; aussi ils peuvent être appliqués à deux objets quelconques et ne lèvent jamais d'exception.

Deux autres opérations avec la même priorité syntaxique, `in` et `not in`, sont pris en charge par les types itérables ou qui implémentent la méthode `__contains__()`.

Types numériques — `int`, `float`, `complex`

Il existe trois types numériques distincts: *integers* (entiers), *floating point numbers* (nombres flottants) et *complex numbers* (nombres complexes). En outre, les booléens sont un sous-type des entiers. Les entiers ont une précision illimitée. Les nombres à virgule flottante sont généralement implémentés en utilisant des `double` en C ; des informations sur la précision et la représentation interne des nombres à virgule flottante pour la machine sur laquelle le programme est en cours d'exécution est disponible dans `sys.float_info`. Les nombres complexes ont une partie réelle et une partie imaginaire, qui sont chacune des nombres à virgule flottante. Pour extraire ces parties d'un nombre complexe *z*, utilisez `z.real` et `z.imag`. (La bibliothèque standard comprend d'autres types numériques, `fractions` qui stocke des rationnels et `decimal` qui stocke les nombres à virgule flottante avec une précision définissable par l'utilisateur.)

Les nombres sont créés par des littéraux numériques ou sont le résultat de fonctions natives ou d'opérateurs. Les entiers littéraux basiques (y compris leur forme hexadécimale, octale et binaire) donnent des entiers. Les nombres littéraux contenant un point décimal ou un exposant donnent des nombres à virgule flottante. Suffixer `'j'` ou `'J'` à un nombre littéral donne un nombre imaginaire (un nombre complexe avec une partie réelle nulle) que vous pouvez ajouter à un nombre entier ou un à virgule flottante pour obtenir un nombre complexe avec une partie réelle et une partie imaginaire.

Python gère pleinement l'arithmétique mixte : quand un opérateur arithmétique binaire a des opérandes de types numériques différents, l'opérande avec le type "le plus étroit" est élargie au type de l'autre, où l'entier est plus étroit que la virgule flottante, qui est plus étroite que les complexes. Les comparaisons entre des nombres de type mixte utilisent la même règle. [2] Les constructeurs `int()`, `float()` et `complex()` peuvent être utilisé pour produire des nombres d'un type spécifique.

Tous les types numériques (sauf complexe) gèrent les opérations suivantes, classées par priorité ascendante (toutes les opérations numériques ont une priorité plus élevée que les opérations de comparaison) :

Opération	Résultat	Notes	Documentation complète
`x + y`	somme de *x* et *y*		

Opération	Résultat	Notes	Documentation complète
`x - y`	différence de *x* et *y*		
`x * y`	produit de *x* et *y*		
`x / y`	quotient de *x* et *y*		
`x // y`	quotient entier de *x* et *y*	(1)	
`x % y`	reste de `x / y`	(2)	
`-x`	négatif de *x*		
`+x`	*x* inchangé		
`abs(x)`	valeur absolue de *x*		`abs()`
`int(x)`	*x* converti en nombre entier	(3)(6)	`int()`
`float(x)`	*x* converti en nombre à virgule flottante	(4)(6)	`float()`
`complex(re, im)`	un nombre complexe avec *re* pour partie réelle et *im* pour partie imaginaire. *im* vaut zéro par défaut.	(6)	`complex()`
`c.conjugate()`	conjugué du nombre complexe *c*		
`divmod(x, y)`	la paire `(x // y, x % y)`	(2)	`divmod()`
`pow(x, y)`	*x* à la puissance *y*	(5)	`pow()`
`x ** y`	*x* à la puissance *y*	(5)	

Notes :

1. Également appelé division entière. La valeur résultante est un nombre entier, bien que le type du résultat ne soit pas nécessairement *int*. Le résultat est toujours arrondi vers moins l'infini : `1//2` vaut `0`, `(-1)//2` vaut `-1`, `1//(-2)` vaut `-1`, et `(-1)//(-2)` vaut `0`.
2. Pas pour les nombres complexes. Convertissez-les plutôt en nombres flottants à l'aide de `abs()` si c'est approprié.
3. La conversion de virgule flottante en entier peut arrondir ou tronquer comme en C ; voir les fonctions `math.floor()` et `math.ceil()` pour des conversions bien définies.
4. *float* accepte aussi les chaînes *nan* et *inf* avec un préfixe optionnel `+` ou `-` pour *Not a Number* (*NaN*) et les infinis positif ou négatif.
5. Python définit `pow(0, 0)` et `0 ** 0` valant `1`, puisque c'est courant pour les langages de programmation, et logique.
6. Les littéraux numériques acceptés comprennent les chiffres `0` à `9` ou tout équivalent Unicode (caractères avec la propriété `Nd`).

   Voir http://www.unicode.org/Public/10.0.0/ucd/extracted/DerivedNumericType.txt pour une liste complète des caractères avec la propriété `Nd`.

Tous types `numbers.Real` (`int` et `float`) comprennent également les opérations suivantes :

Opération	Résultat
`math.trunc(x)`	*x* tronqué à l'`Integral`
`round(x[, n])`	*x* arrondi à *n* chiffres, arrondissant la moitié au pair. Si *n* est omis, la valeur par défaut à 0.
`math.floor(x)`	le plus grand `Integral` <= *x*
`math.ceil(x)`	le plus petit `Integral` >= *x*

Pour d'autres opérations numériques voir les modules `math` et `cmath`.

Opérations sur les bits des nombres entiers

Les opérations bit à bit n'ont de sens que pour les entiers relatifs. Le résultat d'une opération bit à bit est calculé comme si elle était effectuée en complément à deux avec un nombre infini de bits de signe.

Les priorités de toutes les opération à deux opérandes sur des bits sont inférieures aux opérations numériques et plus élevées que les comparaisons ; l'opération unaire `~` a la même priorité que les autres opérations numériques unaires (`+` et `-`).

Ce tableau répertorie les opérations binaires triées par priorité ascendante :

Opération	Résultat	Notes
`x | y`	*ou <or>* binaire de *x* et *y*	(4)
`x ^ y`	*ou <or>* exclusive binaire de *x* et *y*	(4)
`x & y`	*et binaire <and>* de *x* et *y*	(4)
`x << n`	*x* décalé vers la gauche de *n* bits	(1)(2)
`x >> n`	*x* décalé vers la droite de *n* bits	(1)(3)
`~x`	les bits de *x*, inversés	

Notes :

1. Des valeurs de décalage négatives sont illégales et provoquent une exception `ValueError`.
2. Un décalage à gauche de *n* bits est équivalent à la multiplication par `pow(2, n)` sans vérification de débordement.
3. Un décalage à droite de *n* les bits est équivalent à la division par `pow(2, n)` sans vérification de débordement.
4. Effectuer ces calculs avec au moins un bit d'extension de signe supplémentaire dans une représentation finie du complément à deux éléments (une largeur de bit fonctionnelle de `1 + max(x.bit_length(), y.bit_length()` ou plus) est suffisante pour obtenir le même résultat que s'il y avait un nombre infini de bits de signe.

Méthodes supplémentaires sur les entiers

Le type *int* implémente la classe de base abstraite `numbers.Integral`. Il fournit aussi quelques autres méthodes :

`int.`**`bit_length`**()

Renvoie le nombre de bits nécessaires pour représenter un nombre entier en binaire, à l'exclusion du signe et des zéros non significatifs :

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

Plus précisément, si `x` est différent de zéro, `x.bit_length()` est le nombre entier positif unique, `k` tel que `2**(k-1) <= abs(x) < 2**k`. Équivalemment, quand `abs(x)` est assez petit pour avoir un logarithme correctement arrondi, `k = 1 + int(log(abs(x), 2))`. Si `x` est nul, alors `x.bit_length()` donne `0`.

Équivalent à :

```
def bit_length(self):
    s = bin(self)       # binary representation:  bin(-37) --> '-0b100101'
    s = s.lstrip('-0b') # remove leading zeros and minus sign
    return len(s)       # len('100101') --> 6
```

Nouveau dans la version 3.1.

`int.`**`to_bytes`**(*length*, *byteorder*, *, *signed=False*)

Renvoie un tableau d'octets représentant un nombre entier.

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() + 7) // 8, byteorder='little')
```

```
b'\xe8\x03'
```

L'entier est représenté par *length* octets. Une exception `OverflowError` est levée s'il n'est pas possible de représenter l'entier avec le nombre d'octets donnés.

L'argument *byteorder* détermine l'ordre des octets utilisé pour représenter le nombre entier. Si *byteorder* est `"big"`, l'octet le plus significatif est au début du tableau d'octets. Si *byteorder* est `"little"`, l'octet le plus significatif est à la fin du tableau d'octets. Pour demander l'ordre natif des octets du système hôte, donnez `sys.byteorder` comme *byteorder*.

L'argument *signed* détermine si le complément à deux est utilisé pour représenter le nombre entier. Si *signed* est `False` et qu'un entier négatif est donné, une exception `OverflowError` est levée. La valeur par défaut pour *signed* est `False`.

Nouveau dans la version 3.2.

classmethod `int.`**`from_bytes`**(*bytes*, *byteorder*, *, *signed=False*)

Donne le nombre entier représenté par le tableau d'octets fourni.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

L'argument *bytes* doit être soit un bytes-like object soit un itérable produisant des *bytes*.

L'argument *byteorder* détermine l'ordre des octets utilisé pour représenter le nombre entier. Si *byteorder* est `"big"`, l'octet le plus significatif est au début du tableau d'octets. Si *byteorder* est `"little"`, l'octet le plus significatif est à la fin du tableau d'octets. Pour demander l'ordre natif des octets du système hôte, donnez `sys.byteorder` comme *byteorder*.

L'argument *signed* indique si le complément à deux est utilisé pour représenter le nombre entier.

Nouveau dans la version 3.2.

Méthodes supplémentaires sur les nombres à virgule flottante

Le type *float* implémente la classe de base abstraite `numbers.Real` et a également les méthodes suivantes.

`float.`**`as_integer_ratio`**()

Renvoie une paire de nombres entiers dont le rapport est exactement égal au nombre d'origine et avec un dénominateur positif. Lève `OverflowError` avec un infini et `ValueError` avec un NaN.

`float.`**`is_integer`**()

Donne `True` si l'instance de *float* est finie avec une valeur entière, et `False` autrement :

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

Deux méthodes prennent en charge la conversion vers et à partir de chaînes hexadécimales. Étant donné que les *float* de Python sont stockés en interne sous forme de nombres binaires, la conversion d'un *float* depuis ou vers une chaine décimale implique généralement une petite erreur d'arrondi. En revanche, les chaînes hexadécimales permettent de représenter exactement les nombres à virgule flottante. Cela peut être utile lors du débogage, et dans un travail numérique.

`float.`**`hex`**()

Donne une représentation d'un nombre à virgule flottante sous forme de chaîne hexadécimale. Pour les nombres à virgule flottante finis, cette représentation comprendra toujours un préfixe `0x`, un suffixe `p`, et un exposant.

classmethod `float.`**`fromhex`**(*s*)

Méthode de classe pour obtenir le *float* représenté par une chaîne de caractères hexadécimale *s*. La chaîne *s* peut contenir des espaces avant et après le chiffre.

Notez que `float.hex()` est une méthode d'instance, alors que `float.fromhex()` est une méthode de classe.

Une chaîne hexadécimale prend la forme :

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

où `sign` peut être soit `+` soit `-`, `integer` et `fraction` sont des chaînes de chiffres hexadécimales, et `exponent` est un entier décimal facultativement signé. La casse n'est pas significative, et il doit y avoir au moins un chiffre hexadécimal soit dans le nombre entier soit dans la fraction. Cette syntaxe est similaire à la syntaxe spécifiée dans la section 6.4.4.2 de la norme C99, et est aussi la syntaxe utilisée à partir de Java 1.5. En particulier, la sortie de `float.hex()` est utilisable comme valeur hexadécimale à virgule flottante littérale en C ou Java, et des chaînes hexadécimales produites en C via un format `%a` ou Java via `Double.toHexString` sont acceptées par `float.fromhex()`.

Notez que l'exposant est écrit en décimal plutôt qu'en hexadécimal, et qu'il donne la puissance de 2 par lequel multiplier le coefficient. Par exemple, la chaîne hexadécimale `0x3.a7p10` représente le nombre à virgule flottante `(3 + 10./16 + 7./16**2) *2.0**10`, ou `3740.0` :

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

L'application de la conversion inverse à `3740.0` donne une chaîne hexadécimale différente représentant le même nombre :

```
>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

Hachage des types numériques

Pour deux nombres égaux `x` et `y` (`x == y`), pouvant être de différents types, il est une requis que `hash(x) == hash(y)` (voir la documentation de `__hash__()`). Pour faciliter la mise en œuvre et l'efficacité à travers une variété de types numériques (y compris `int`, `float`, `decimal.Decimal` et `fractions.Fraction`) le hachage en Python pour les types numérique est basé sur une fonction mathématique unique qui est définie pour tout nombre rationnel, et donc s'applique à toutes les instances de `int` et `fractions.Fraction`, et toutes les instances finies de `float` et `decimal.Decimal`. Essentiellement, cette fonction est donnée par la réduction modulo `P` pour un nombre `P` premier fixe. La valeur de `P` est disponible comme attribut `modulus` de `sys.hash_info`.

CPython implementation detail: Actuellement, le premier utilisé est `P = 2 ** 31 - 1` sur des machines dont les *longs* en C sont de 32 bits `P = 2 ** 61 - 1` sur des machines dont les *longs* en C font 64 bits.

Voici les règles en détail :

- Si `x = m / n` est un nombre rationnel non négatif et `n` n'est pas divisible par `P`, définir `hash(x)` comme `m * invmod(n, P) % P`, où `invmod(n, P)` donne l'inverse de `n` modulo `P`.
- Si `x = m / n` est un nombre rationnel non négatif et `n` est divisible par `P` (mais `m` ne l'est pas), alors `n` n'a pas de modulo inverse `P` et la règle ci-dessus n'est pas applicable ; dans ce cas définir `hash(x)` comme étant la valeur de la constante `sys.hash_info.inf`.
- Si `x = m / n` est un nombre rationnel négatif définir `hash(x)` comme `-hash(-x)`. Si le résultat est `-1`, le remplacer par `-2`.
- Les valeurs particulières `sys.hash_info.inf`, `-sys.hash_info.inf` et `sys.hash_info.nan` sont utilisées comme valeurs de hachage pour l'infini positif, l'infini négatif, ou *nans* (respectivement). (Tous les *nans* hachables ont la même valeur de hachage.)
- Pour un nombre `complexe` `z`, les valeurs de hachage des parties réelles et imaginaires sont combinées en calculant `hash(z.real) + sys.hash_info.imag * hash(z.imag)`, réduit au modulo `2**sys.hash_info.width` de sorte qu'il se trouve dans `range(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1))`. Encore une fois, si le résultat est `-1`, il est remplacé par `-2`.

Afin de clarifier les règles ci-dessus, voici quelques exemples de code Python, équivalent à la fonction de hachage native, pour calculer le hachage d'un nombre rationnel, d'un `float`, ou d'un `complex` :

```
import sys, math
```

```
def hash_fraction(m, n):
    """Compute the hash of a rational number m / n.

    Assumes m and n are integers, with n positive.
    Equivalent to hash(fractions.Fraction(m, n)).

    """
    P = sys.hash_info.modulus
    # Remove common factors of P.  (Unnecessary if m and n already coprime.)
    while m % P == n % P == 0:
        m, n = m // P, n // P

    if n % P == 0:
        hash_value = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: pow(n, P-1, P) is 1, so
        # pow(n, P-2, P) gives the inverse of n modulo P.
        hash_value = (abs(m) % P) * pow(n, P - 2, P) % P
    if m < 0:
        hash_value = -hash_value
    if hash_value == -1:
        hash_value = -2
    return hash_value

def hash_float(x):
    """Compute the hash of a float x."""

    if math.isnan(x):
        return sys.hash_info.nan
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""

    hash_value = hash_float(z.real) + sys.hash_info.imag * hash_float(z.imag)
    # do a signed reduction modulo 2**sys.hash_info.width
    M = 2**(sys.hash_info.width - 1)
    hash_value = (hash_value & (M - 1)) - (hash_value & M)
    if hash_value == -1:
        hash_value = -2
    return hash_value
```

Les types itérateurs

Python supporte un concept d'itération sur les conteneurs. C'est implémenté en utilisant deux méthodes distinctes qui permettent aux classes définies par l'utilisateur de devenir itérables. Les séquences, décrites plus bas en détail, supportent toujours les méthodes d'itération.

Une méthode doit être définie afin que les objets conteneurs supportent l'itération :

`container.__iter__()`
: Donne un objet itérateur. L'objet doit implémenter le protocole d'itération décrit ci-dessous. Si un conteneur prend en charge différents types d'itération, d'autres méthodes peuvent être fournies pour obtenir spécifiquement les itérateurs pour ces types d'itération. (Exemple d'un objet supportant plusieurs formes d'itération : une structure d'arbre pouvant être parcourue en largeur ou en profondeur.) Cette méthode correspond à l'attribut `tp_iter` de la structure du type des objets Python dans l'API Python/C.

Les itérateurs eux-mêmes doivent implémenter les deux méthodes suivantes, qui forment ensemble le *protocole d'itérateur <iterator protocol>* :

`iterator.__iter__()`
: Donne l'objet itérateur lui-même. Cela est nécessaire pour permettre à la fois à des conteneurs et des itérateurs d'être utilisés avec les instructions `for` et `in`. Cette méthode correspond à l'attribut `tp_iter` de la structure des types des objets Python dans l'API Python/C.

`iterator.__next__()`
: Donne l'élément suivant du conteneur. S'il n'y a pas d'autres éléments, une exception `StopIteration` est levée. Cette méthode correspond à l'attribut `PyTypeObject.tp_iternext` de la structure du type des ob-

jets Python dans l'API Python/C.

Python définit plusieurs objets itérateurs pour itérer sur les types standards ou spécifiques de séquence, de dictionnaires et d'autres formes plus spécialisées. Les types spécifiques ne sont pas importants au-delà de leur implémentation du protocole d'itération.

Dès que la méthode `__next__()` lève une exception `StopIteration`, elle doit continuer à le faire lors des appels ultérieurs.Implémentations qui ne respectent pas cette propriété sont considérées cassées.

Types générateurs

Les generators offrent un moyen pratique d'implémenter le protocole d'itération. Si la méthode `__iter__()` d'un objet conteneur est implémentée comme un générateur, elle renverra automatiquement un objet *iterator* (techniquement, un objet générateur) fournissant les méthodes `__iter__()` et `__next__()`. Plus d'informations sur les générateurs peuvent être trouvés dans la documentation de l'expression yield.

Types séquentiels — `list`, `tuple`, `range`

Il existe trois types séquentiels basiques: les *lists*, *tuples* et les *range*. D'autres types séquentiels spécifiques au traitement de données binaires et chaînes de caractères sont décrits dans des sections dédiées.

Opérations communes sur les séquences

Les opérations dans le tableau ci-dessous sont pris en charge par la plupart des types séquentiels, variables et immuables. La classe de base abstraite `collections.abc.Sequence` est fournie pour aider à implémenter correctement ces opérations sur les types séquentiels personnalisés.

Ce tableau répertorie les opérations sur les séquences triées par priorité ascendante. Dans le tableau, *s,* et *t* sont des séquences du même type, *n*, *i*, *j* et *k* sont des nombres entiers et *x* est un objet arbitraire qui répond à toutes les restrictions de type et de valeur imposée par *s*.

Les opérations `in` et `not in` ont les mêmes priorités que les opérations de comparaison. Les opérations `+` (concaténation) et `*` (répétition) ont la même priorité que les opérations numériques correspondantes. [3]

Opération	Résultat	Notes
`x in s`	`True` si un élément de *s* est égal à *x*, sinon `False`	(1)
`x not in s`	`False` si un élément de *s* est égal à *x*, sinon `True`	(1)
`s + t`	la concaténation de *s* et *t*	(6)(7)
`s * n` or `n * s`	équivalent à ajouter *s* *n* fois à lui même	(2)(7)
`s[i]`	i^{e} élément de *s* en commençant par 0	(3)
`s[i:j]`	tranche (*slice*) de *s* de *i* à *j*	(3)(4)
`s[i:j:k]`	tranche (*slice*) de *s* de *i* à *j* avec un pas de *k*	(3)(5)
`len(s)`	longueur de *s*	
`min(s)`	plus petit élément de *s*	
`max(s)`	plus grand élément de *s*	
`s.index(x[, i[, j]])`	indice de la première occurrence de *x* dans *s* (à ou après l'indice *i* et avant indice *j*)	(8)
`s.count(x)`	nombre total d'occurrences de *x* dans *s*	

Les séquences du même type supportent également la comparaison. En particulier, les n-uplets et les listes sont comparés lexicographiquement en comparant les éléments correspondants. Cela signifie que pour que deux séquences soit égales, les éléments les constituant doivent être égaux deux à deux et les deux séquences doivent être du même type et de la même longueur. (Pour plus de détails voir Comparaisons dans la référence du langage.)

Notes :

1. Bien que les opérations `in` et `not in` ne soient généralement utilisées que pour les tests d'appartenance simple, certaines séquences spécialisées (telles que `str`, `bytes` et `bytearray`) les utilisent aussi pour tester l'existence de sous-séquences :

```
>>> "gg" in "eggs"
True
```

2. Les valeurs de *n* plus petites que `0` sont traitées comme `0` (ce qui donne une séquence vide du même type que *s*). Notez que les éléments de *s* ne sont pas copiés ; ils sont référencés plusieurs fois. Cela hante souvent de nouveaux développeurs Python, typiquement :

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

   Ce qui est arrivé est que `[[]]` est une liste à un élément contenant une liste vide, de sorte que les trois éléments de `[[]] * 3` sont des références à cette seule liste vide. Modifier l'un des éléments de `lists` modifie cette liste unique. Vous pouvez créer une liste des différentes listes de cette façon :

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

   De plus amples explications sont disponibles dans la FAQ à la question Comment créer une liste à plusieurs dimensions ?.
3. Si *i* ou *j* sont négatifs, l'indice est relatif à la fin de la séquence *s* : `len(s) + i` ou `len(s) + j` est substitué. Mais notez que `-0` est toujours `0`.
4. La tranche de *s* de *i* à *j* est définie comme la séquence d'éléments d'indice *k* tels que `i <= k < j`. Si *i* ou *j* est supérieur à `len(s)`, `len(s)` est utilisé. Si *i* est omis ou `None`, `0` est utilisé. Si *j* est omis ou `None`, `len(s)` est utilisé. Si *i* est supérieure ou égale à *j*, la tranche est vide.
5. La tranche de *s* de *i* à *j* avec un pas de *k* est définie comme la séquence d'éléments d'indice `x = i + n*k` tels que `0 <= n < (j-i)/k`. En d'autres termes, les indices sont `i`, `i+k`, `i+2*k`, `i+3*k` et ainsi de suite, en arrêtant lorsque *j* est atteint (mais jamais inclus). Si *k* est positif, *i* et *j* sont réduits, s'ils sont plus grands, à `len(s)`. Si *k* est négatif, *i* et *j* sont réduits à `len(s) - 1` s'ils sont plus grands. Si *i* ou *j* sont omis ou sont `None`, ils deviennent des valeurs "extrêmes" (où l'ordre dépend du signe de *k*). Remarquez, *k* ne peut pas valoir zéro. Si *k* est `None`, il est traité comme `1`.
6. Concaténer des séquences immuables donne toujours un nouvel objet. Cela signifie que la construction d'une séquence par concaténations répétées aura une durée d'exécution quadratique par rapport à la longueur de la séquence totale. Pour obtenir un temps d'exécution linéaire, vous devez utiliser l'une des alternatives suivantes :
   - si vous concaténez des `str`, vous pouvez construire une liste puis utiliser `str.join()` à la fin, ou bien écrire dans une instance de `io.StringIO` et récupérer sa valeur lorsque vous avez terminé
   - si vous concaténez des `bytes`, vous pouvez aussi utiliser `bytes.join()` ou `io.BytesIO`, ou vous pouvez faire les concaténation sur place avec un objet `bytearray`. Les objets `bytearray` sont muables et ont un mécanisme de sur-allocation efficace
   - si vous concaténez des `tuple`, utilisez plutôt *extend* sur une `list`
   - pour d'autres types, cherchez dans la documentation de la classe concernée
7. Certains types séquentiels (tels que `range`) ne supportent que des séquences qui suivent des modèles spécifiques, et donc ne prennent pas en charge la concaténation ou la répétition.
8. `index` lève une exception `ValueError` quand *x* ne se trouve pas dans *s*. Toutes les implémentations ne gèrent pas les deux paramètres supplémentaires *i* et *j*. Ces deux arguments permettent de chercher efficacement dans une sous-séquence de la séquence. Donner ces arguments est plus ou moins équivalent à `s[i:j].index(x)`, sans copier les données ; l'indice renvoyé alors relatif au début de la séquence plutôt qu'au début de la tranche.

Types de séquences immuables

La seule opération que les types de séquences immuables implémentent qui n'est pas implémentée par les types de séquences muables est le support de la fonction native `hash()`.

Cette implémentation permet d'utiliser des séquences immuables, comme les instances de `tuple`, en tant que clés de `dict` et stockées dans les instances de `set` et `frozenset`.

Essayer de hacher une séquence immuable qui contient des valeurs non-hachables lèvera une `TypeError`.

Types de séquences muables

Les opérations dans le tableau ci-dessous sont définies sur les types de séquences muables. La classe de base abstraite `collections.abc.MutableSequence` est prévue pour faciliter l'implémentation correcte de ces opérations sur les types de séquence personnalisées.

Dans le tableau *s* est une instance d'un type de séquence muable, *t* est un objet itérable et *x* est un objet arbitraire qui répond à toutes les restrictions de type et de valeur imposées par *s* (par exemple, `bytearray` accepte uniquement des nombres entiers qui répondent à la restriction de la valeur `0 <= x <= 255`).

Opération	Résultat	Notes
`s[i] = x`	élément *i* de *s* est remplacé par *x*	
`s[i:j] = t`	tranche de *s* de *i* à *j* est remplacée par le contenu de l'itérable *t*	
`del s[i:j]`	identique à `s[i:j] = []`	
`s[i:j:k] = t`	les éléments de `s[i:j:k]` sont remplacés par ceux de *t*	(1)
`del s[i:j:k]`	supprime les éléments de `s[i:j:k]` de la liste	
`s.append(x)`	ajoute *x* à la fin de la séquence (identique à `s[len(s):len(s)] = [x]`)	
`s.clear()`	supprime tous les éléments de *s* (identique à `del s[:]`)	(5)
`s.copy()`	crée une copie superficielle de *s* (identique à `s[:]`)	(5)
`s.extend(t)` or `s += t`	étend *s* avec le contenu de *t* (proche de `s[len(s):len(s)] = t`)	
`s *= n`	met à jour *s* avec son contenu répété *n* fois	(6)
`s.insert(i, x)`	insère *x* dans *s* à l'index donné par *i* (identique à `s[i:i] = [x]`)	
`s.pop([i])`	récupère l'élément à *i* et le supprime de *s*	(2)
`s.remove(x)`	supprime le premier élément de *s* pour lequel `s[i]` est égal à *x*	(3)
`s.reverse()`	inverse sur place les éléments de *s*	(4)

Notes :

1. *t* doit avoir la même longueur que la tranche qu'il remplace.
2. L'argument optionnel *i* vaut `-1` par défaut, afin que, par défaut, le dernier élément soit retiré et renvoyé.
3. `remove` lève une exception `ValueError` si *x* ne se trouve pas dans *s*.
4. La méthode `reverse()` modifie les séquence sur place pour économiser de l'espace lors du traitement de grandes séquences. Pour rappeler aux utilisateurs qu'elle a un effet de bord, elle ne renvoie pas la séquence inversée.
5. `clear()` et `copy()` sont incluses pour la compatibilité avec les interfaces des conteneurs muables qui ne supportent pas les opérations de découpage (comme `dict` et `set`)

   Nouveau dans la version 3.3: méthodes `clear()` et `copy()`.

6. La valeur *n* est un entier, ou un objet implémentant `__index__()`. Zéro et les valeurs négatives de *n* permettent d'effacer la séquence. Les éléments dans la séquence ne sont pas copiés ; ils sont référencés plusieurs fois, comme expliqué pour `s * n` dans Opérations communes sur les séquences.

Listes

Les listes sont des séquences muables, généralement utilisées pour stocker des collections d'éléments homogènes (où le degré de similitude variera selon l'usage).

class **`list`**([*iterable*])

Les listes peuvent être construites de différentes manières :

- En utilisant une paire de crochets pour indiquer une liste vide : `[]`
- Au moyen de crochets, séparant les éléments par des virgules : `[a]`, `[a, b, c]`
- En utilisant une liste en compréhension : `[x for x in iterable]`
- En utilisant le constructeur du type : `list()` ou `list(iterable)`

Le constructeur crée une liste dont les éléments sont les mêmes et dans le même ordre que les éléments d'*iterable*. *iterable* peut être soit une séquence, un conteneur qui supporte l'itération, soit un itérateur. Si *iterable* est déjà une liste, une copie est faite et renvoyée, comme avec `iterable[:]`. Par exemple, `list('abc')` renvoie `['a', 'b', 'c']` et `list( (1, 2, 3) )` renvoie `[1, 2, 3]`. Si aucun argument est donné, le constructeur crée une nouvelle liste vide, `[]`.

De nombreuses autres opérations produisent des listes, tel que la fonction native `sorted()`.

Les listes supportent toutes les opérations des séquences communes et muables. Les listes fournissent également la méthode supplémentaire suivante :

`sort`(*, *key=None*, *reverse=False*)

Cette méthode trie la liste sur place, en utilisant uniquement des comparaisons `<` entre les éléments. Les exceptions ne sont pas supprimées si n'importe quelle opération de comparaison échoue, le tri échouera (et la liste sera probablement laissée dans un état partiellement modifié).

`sort()` accepte deux arguments qui ne peuvent être fournis que par mot-clé (keyword-only arguments):

key spécifie une fonction d'un argument utilisée pour extraire une clé de comparaison de chaque élément de la liste (par exemple, `key=str.lower`). La clé correspondant à chaque élément de la liste n'est calculée qu'une seule fois, puis utilisée durant tout le processus. La valeur par défaut, `None`, signifie que les éléments sont triés directement sans en calculer une valeur "clé" séparée.

La fonction utilitaire `functools.cmp_to_key()` est disponible pour convertir une fonction *cmp* du style 2.x à une fonction *key*.

reverse, une valeur booléenne. Si elle est `True`, la liste d'éléments est triée comme si toutes les comparaisons étaient inversées.

Cette méthode modifie la séquence sur place pour économiser de l'espace lors du tri de grandes séquences. Pour rappeler aux utilisateurs cet effet de bord, elle ne renvoie pas la séquence triée (utilisez `sorted()` pour demander explicitement une nouvelle instance de liste triée).

La méthode `sort()` est garantie stable. Un tri est stable s'il garantit de ne pas changer l'ordre relatif des éléments égaux --- cela est utile pour trier en plusieurs passes (par exemple, trier par département, puis par niveau de salaire).

CPython implementation detail: L'effet de tenter de modifier, ou même inspecter la liste pendant qu'elle se fait trier est indéfini. L'implémentation C de Python fait apparaître la liste comme vide pour la durée du traitement, et lève `ValueError` si elle détecte que la liste a été modifiée au cours du tri.

Tuples

Les tuples (*n-uplets* en français) sont des séquences immuables, généralement utilisées pour stocker des collections de données hétérogènes (tels que les tuples de deux éléments produits par la fonction native `enumerate()`). Les tuples sont également utilisés dans des cas où une séquence homogène et immuable de données est nécessaire (pour, par exemple, les stocker dans un `set` ou un `dict`).

class **`tuple`**([*iterable*])

Les tuples peuvent être construits de différentes façons :

- En utilisant une paire de parenthèses pour désigner le tuple vide : `()`

- En utilisant une virgule, pour créer un tuple d'un élément : `a,` ou `(a,)`
- En séparant les éléments avec des virgules : `a, b, c` ou `(a, b, c)`
- En utilisant la fonction native `tuple()` : `tuple()` ou `tuple(iterable)`

Le constructeur construit un tuple dont les éléments sont les mêmes et dans le même ordre que les éléments de *iterable*. *iterable* peut être soit une séquence, un conteneur qui supporte l'itération, soit un itérateur. Si *iterable* est déjà un tuple, il est renvoyé inchangé. Par exemple, `tuple('abc')` renvoie `('a', 'b', 'c')` et `tuple( [1, 2, 3] )` renvoie `(1, 2, 3)`. Si aucun argument est donné, le constructeur crée un nouveau tuple vide, `()`.

Notez que c'est en fait la virgule qui fait un tuple et non les parenthèses. Les parenthèses sont facultatives, sauf dans le cas du tuple vide, ou lorsqu'elles sont nécessaires pour éviter l'ambiguïté syntaxique. Par exemple, `f(a, b, c)` est un appel de fonction avec trois arguments, alors que `f((a, b, c))` est un appel de fonction avec un tuple de trois éléments comme unique argument.

Les tuples implémentent toutes les opérations communes des séquences.

Pour les collections hétérogènes de données où l'accès par nom est plus clair que l'accès par index, `collections.namedtuple()` peut être un choix plus approprié qu'un simple tuple.

Ranges

Le type `range` représente une séquence immuable de nombres et est couramment utilisé pour itérer un certain nombre de fois dans les boucles `for`.

class **`range`**(*stop*)
class **`range`**(*start*, *stop*[, *step*])

Les arguments du constructeur de *range* doivent être des entiers (des `int` ou tout autre objet qui implémente la méthode spéciale `__index__`). La valeur par défaut de l'argument *step* est `1`. La valeur par défaut de l'argument *start* est `0`. Si *step* est égal à zéro, une exception `ValueError` est levée.

Pour un *step* positif, le contenu d'un *range* `r` est déterminé par la formule `r[i] = start + step*i` où `i >= 0` et `r[i] < stop`.

Pour un *step* négatif, le contenu du *range* est toujours déterminé par la formule `r[i] = start + step*i`, mais les contraintes sont `i >= 0` et `r[i] > stop`.

Un objet *range* sera vide si `r[0]` ne répond pas à la contrainte de valeur. Les *range* prennent en charge les indices négatifs, mais ceux-ci sont interprétées comme une indexation de la fin de la séquence déterminée par les indices positifs.

Les *range* contenant des valeurs absolues plus grandes que `sys.maxsize` sont permises, mais certaines fonctionnalités (comme `len()`) peuvent lever `OverflowError`.

Exemples avec *range* :

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

range implémente toutes les opérations communes des séquences sauf la concaténation et la répétition (en raison du fait que les *range* ne peuvent représenter que des séquences qui respectent un motif strict et que la répétition et la concaténation les feraient dévier de ce motif).

`start`

La valeur du paramètre *start* (ou `0` si le paramètre n'a pas été fourni)

`stop`

La valeur du paramètre *stop*

`step`

La valeur du paramètre *step* (ou `1` si le paramètre n'a pas été fourni)

L'avantage du type `range` sur une `list` classique ou `tuple` est qu'un objet `range` prendra toujours la même (petite) quantité de mémoire, peu importe la taille de la gamme qu'elle représente (car elle ne stocke que les valeurs `start`, `stop` et `step` , le calcul des éléments individuels et les sous-intervalles au besoin).

Les *range* implémentent la classe de base abstraite `collections.abc.Sequence` et offrent des fonctionnalités telles que les tests d'appartenance (avec *in*), de recherche par index, le tranchage et ils gèrent les indices négatifs (voir Types séquentiels — list, tuple, range):

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

Comparer des *range* avec `==` et `!=` les compare comme des séquences. Soit deux objets *range* sont considérées comme égaux si ils représentent la même séquence de valeurs. (Notez que deux objets *range* dits égaux pourraient avoir leurs attributs `start`, `stop` et `step` différents, par exemple `range(0) == range(2, 1, 3)` ou `range(0, 3, 2) == range(0, 4, 2)`.)

Modifié dans la version 3.2: Implémente la classe de base abstraite *Sequence*. Supporte le *slicing* et les indices négatifs. Tester l'appartenance d'un `int` en temps constant au lieu d'itérer tous les éléments.

Modifié dans la version 3.3: `==` et `!=` comparent des *range* en fonction de la séquence de valeurs qu'ils définissent (au lieu d'une comparaison fondée sur l'identité de l'objet).

Nouveau dans la version 3.3: Les attributs `start`, `stop` et `step`.

Voir aussi:

- La recette linspace montre comment implémenter une version paresseuse de *range* adaptée aux nombres à virgule flottante.

Type Séquence de Texte — `str`

Les données textuelles en Python est manipulé avec des objets `str` ou *strings*. Les chaînes sont des séquences immuables de points de code Unicode. Les chaînes littérales peuvent être écrites de différentes manières :

- Les guillemets simples : `'autorisent les "guillemets"'`
- Les guillemets : `"autorisent les guillemets 'simples'"`.
- Guillemets triples : `'''Trois guillemets simples'''`, `"""Trois guillemets"""`

Les chaînes entre triple guillemets peuvent couvrir plusieurs lignes, tous les espaces associés seront inclus dans la chaîne littérale.

Les chaînes littérales qui font partie d'une seule expression et ont seulement des espaces entre elles sont implicitement converties en une seule chaîne littérale. Autrement dit, `("spam " "eggs") == "spam eggs"`.

Voir Littéraux de chaînes de caractères et de suites d'octets pour plus d'informations sur les différentes formes de chaînes littérales, y compris des séquences d'échappement prises en charge, et le préfixe `r` (*raw* (brut)) qui désactive la plupart des traitements de séquence d'échappement.

Les chaînes peuvent également être créés à partir d'autres objets à l'aide du constructeur `str`.

Comme il n'y a pas de type "caractère" distinct, l'indexation d'une chaîne produit des chaînes de longueur 1. Autrement dit, pour une chaîne non vide *s*, `s[0] == s[0:1]`.

Il n'y a aucun type de chaîne muable, mais `str.join()` ou `io.StringIO` peuvent être utilisées pour construire efficacement des chaînes à partir de plusieurs fragments.

Modifié dans la version 3.3: Pour une compatibilité ascendante avec la série Python 2, le préfixe `u` est à nouveau autorisé sur les chaînes littérales. Elle n'a aucun effet sur le sens des chaînes littérales et ne peut être combiné avec le préfixe `r`.

class **`str`**(*object=''*)
class **`str`**(*object=b''*, *encoding='utf-8'*, *errors='strict'*)

Renvoie une représentation string de *object*. Si *object* n'est pas fourni, renvoie une chaîne vide. Sinon, le comportement de `str()` dépend de si *encoding* ou *errors* sont donnés, voir l'exemple.

Si ni *encoding* ni *errors* ne sont donnés, `str(object)` renvoie `object.__str__()`, qui est la représentation de chaîne "informelle" ou bien affichable de *object*. Pour les chaînes, c'est la chaîne elle-même. Si *object* n'a pas de méthode `__str__()`, `str()` utilise `repr(object)`.

Si au moins un des deux arguments *encoding* ou *errors* est donné, *object* doit être un bytes-like object (par exemple `bytes` ou `bytearray`). Dans ce cas, si *object* est un objet `bytes` (ou `bytearray`), alors `str(bytes, encoding, errors)` est équivalent à `bytes.decode(encoding, errors)`. Sinon, l'objet *bytes* du *buffer* est obtenu avant d'appeler `bytes.decode()`. Voir Séquences Binaires --- bytes, bytearray, memoryview et Protocole tampon pour plus d'informations sur les *buffers*.

Donner un objet `bytes` à `str()` sans ni l'argument *encoding* ni l'argument *errors* relève du premier cas, où la représentation informelle de la chaîne est renvoyé (voir aussi l'option `-b` de Python). Par exemple :

```
>>> str(b'Zoot!')
"b'Zoot!'"
```

Pour plus d'informations sur la classe `str` et ses méthodes, voir les sections Type Séquence de Texte — str et Méthodes de chaînes de caractères. Pour formater des chaînes de caractères, voir les sections Chaînes de caractères formatées littérales et Syntaxe de formatage de chaîne. La section Services de Manipulation de Texte contient aussi des informations.

Méthodes de chaînes de caractères

Les chaînes implémentent toutes les opérations communes des séquences, ainsi que les autres méthodes décrites ci-dessous.

Les chaînes gèrent aussi deux styles de mise en forme, l'un fournissant une grande flexibilité et de personnalisation (voir `str.format()`, Syntaxe de formatage de chaîne et Formatage personnalisé de chaîne) et l'autre basée sur `printf` du C qui gère une gamme plus étroite des types et est légèrement plus difficile à utiliser correctement, mais il est souvent plus rapide pour les cas, il peut gérer (Formatage de chaines à la printf).

La section Services de Manipulation de Texte de la bibliothèque standard couvre un certain nombre d'autres modules qui fournissent différents services relatifs au texte (y compris les expressions régulières dans le module `re`).

`str.`**`capitalize`**()

Renvoie une copie de la chaîne avec son premier caractère en majuscule et le reste en minuscule.

`str.`**`casefold`**()

Renvoie une copie *casefolded* de la chaîne. Les chaînes *casefolded* peuvent être utilisées dans des comparaison insensibles à la casse.

Le *casefolding* est une technique agressive de mise en minuscule, car il vise à éliminer toutes les distinctions de casse dans une chaîne. Par exemple, la lettre minuscule `'ß'` de l'allemand équivaut à `"ss"`. Comme il est déjà minuscule, `lower()` ferait rien à `'ß'`; `casefold()` le convertit en `"ss"`.

L'algorithme de *casefolding* est décrit dans la section 3.13 de la norme Unicode.

Nouveau dans la version 3.3.

`str.`**`center`**(*width*[, *fillchar*])

Donne la chaîne au centre d'une chaîne de longueur *width*. Le remplissage est fait en utilisant l'argument *fill-*

char (qui par défaut est un espace ASCII). La chaîne d'origine est renvoyée si *width* est inférieur ou égale à `len(s)`.

`str.`**`count`**(*sub*[, *start*[, *end*]])

Donne le nombre d'occurrences de *sub* ne se chevauchant pas dans le *range* [*start*, *end*]. Les arguments facultatifs *start* et *end* sont interprétés comme pour des *slices*.

`str.`**`encode`**(*encoding="utf-8"*, *errors="strict"*)

Donne une version encodée de la chaîne sous la forme d'un objet *bytes*. L'encodage par défaut est `'utf-8'`. *errors* peut être donné pour choisir un autre système de gestion d'erreur. La valeur par défaut pour *errors* est `'strict'`, ce qui signifie que les erreurs d'encodage lèvent une `UnicodeError`. Les autres valeurs possibles sont `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` et tout autre nom enregistré *via* `codecs.register_error()`, voir la section Error Handlers. Pour une liste des encodages possibles, voir la section Standard Encodings.

Modifié dans la version 3.1: Gestion des arguments par mot clef.

`str.`**`endswith`**(*suffix*[, *start*[, *end*]])

Donne `True` si la chaîne se termine par *suffix*, sinon `False`. *suffix* peut aussi être un tuple de suffixes à rechercher. Si l'argument optionnel *start* est donné, le test se fait à partir de cette position. Si l'argument optionnel *end* est fourni, la comparaison s'arrête à cette position.

`str.`**`expandtabs`**(*tabsize=8*)

Donne une copie de la chaîne où toutes les tabulations sont remplacées par un ou plusieurs espaces, en fonction de la colonne courante et de la taille de tabulation donnée. Les positions des tabulations se trouvent tous les *tabsize* caractères (8 par défaut, ce qui donne les positions de tabulations aux colonnes 0, 8, 16 et ainsi de suite). Pour travailler sur la chaîne, la colonne en cours est mise à zéro et la chaîne est examinée caractère par caractère. Si le caractère est une tabulation (`\t`), un ou plusieurs caractères d'espacement sont insérés dans le résultat jusqu'à ce que la colonne courante soit égale à la position de tabulation suivante. (Le caractère tabulation lui-même n'est pas copié.) Si le caractère est un saut de ligne (`\n`) ou un retour chariot (`\r`), il est copié et la colonne en cours est remise à zéro. Tout autre caractère est copié inchangé et la colonne en cours est incrémentée de un indépendamment de la façon dont le caractère est représenté lors de l'affichage.

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
```

`str.`**`find`**(*sub*[, *start*[, *end*]])

Donne la première la position dans la chaîne où *sub* est trouvé dans le *slice* `s[start:end]`. Les arguments facultatifs *start* et *end* sont interprétés comme dans la notation des *slice*. Donne `-1` si *sub* n'est pas trouvé.

Note: La méthode `find()` ne doit être utilisée que si vous avez besoin de connaître la position de *sub*. Pour vérifier si *sub* est une sous chaine ou non, utilisez l'opérateur `in` :

```
>>> 'Py' in 'Python'
True
```

`str.`**`format`**(**args*, ***kwargs*)

Formate une chaîne. La chaîne sur laquelle cette méthode est appelée peut contenir du texte littéral ou des emplacements de remplacement délimités par des accolades `{}`. Chaque champ de remplacement contient soit l'indice numérique d'un argument positionnel, ou le nom d'un argument donné par mot-clé. Renvoie une copie de la chaîne où chaque champ de remplacement est remplacé par la valeur de chaîne de l'argument correspondant.

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

Voir Syntaxe de formatage de chaîne pour une description des options de formatage qui peuvent être spécifiées dans les chaînes de format.

Note: Lors du formatage avec le format `n` (comme `'{:n}'.format(1234)`) d'un nombre (`int`, `float`, `complex`, `decimal.Decimal` et dérivées), la fonction met temporairement la variable `LC_CTYPE` à la va-

leur de `LC_NUMERIC` pour décoder correctement les attributs `decimal_point` et `thousands_sep` de `localeconv()`, s'ils ne sont pas en ASCII ou font plus d'un octet, et que `LC_NUMERIC` est différent de `LC_CTYPE`. Ce changement temporaire affecte les autres fils d'exécution.

Modifié dans la version 3.7: Lors du formatage d'un nombre avec le format `n`, la fonction change temporairement `LC_CTYPE` par la valeur de `LC_NUMERIC` dans certains cas.

`str.`**`format_map`**(*mapping*)

Semblable à `str.format(**mapping)`, sauf que `mapping` est utilisé directement et non copié dans un `dict`. C'est utile si, par exemple `mapping` est une sous-classe de `dict` :

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

Nouveau dans la version 3.2.

`str.`**`index`**(*sub*[, *start*[, *end*]])

Comme `find()`, mais lève une `ValueError` lorsque la chaîne est introuvable.

`str.`**`isalnum`**()

Donne `True` si tous les caractères de la chaîne sont alphanumériques et qu'il y a au moins un caractère, sinon `False`. Un caractère `c` est alphanumérique si l'un des tests suivants est vrais : `c.isalpha()`, `c.isdecimal()`, `c.isdigit()` ou `c.isnumeric()`.

`str.`**`isalpha`**()

Donne `True` si tous les caractères de la chaîne sont alphabétiques et qu'elle contient au moins un caractère, sinon `False`. Les caractères alphabétiques sont les caractères définis dans la base de données de caractères Unicode comme *Letter*, à savoir, ceux qui ont "Lm", "Lt", "Lu", "Ll", ou "Lo" comme catégorie générale. Notez que ceci est différent de la propriété *Alphabetic* définie dans la norme Unicode.

`str.`**`isascii`**()

Donne `True` si la chaîne est vide ou ne contient que des caractères ASCII, `False` sinon. Les caractères ASCII ont un code dans l'intervalle `"U+0000"`---`"U+007F"`.

Nouveau dans la version 3.7.

`str.`**`isdecimal`**()

Renvoie vrai si tous les caractères de la chaîne sont des caractères décimaux et qu'elle contient au moins un caractère, sinon faux. Les caractères décimaux sont ceux pouvant être utilisés pour former des nombres en base 10, tels que U+0660, ARABIC-INDIC DIGIT ZERO. Spécifiquement, un caractère décimal est un caractère dans la catégorie Unicode générale "Nd".

`str.`**`isdigit`**()

Renvoie vrai si tous les caractères de la chaîne sont des chiffres et qu'elle contient au moins un caractère, faux sinon. Les chiffres incluent des caractères décimaux et des chiffres qui nécessitent une manipulation particulière, tels que les *compatibility superscript digits*. Ça couvre les chiffres qui ne peuvent pas être utilisés pour construire des nombres en base 10, tel que les nombres de Kharosthi. Spécifiquement, un chiffre est un caractère dont la valeur de la propriété *Numeric_Type* est *Digit* ou *Decimal*.

`str.`**`isidentifier`**()

Donne `True` si la chaîne est un identifiant valide selon la définition du langage, section Identifiants et mots-clés.

Utilisez `keyword.iskeyword()` pour savoir si un identifiant est réservé, tels que `def` et `class`.

`str.`**`islower`**()

Donne `True` si tous les caractères capitalisables [4] de la chaîne sont en minuscules et qu'elle contient au moins un caractère capitalisable. Donne `False` dans le cas contraire.

`str.`**`isnumeric`**()

Donne `True` si tous les caractères de la chaîne sont des caractères numériques, et qu'elle contient au moins un caractère, sinon `False`. Les caractères numériques comprennent les chiffres, et tous les carac-

tères qui ont la propriété Unicode *numeric value*, par exemple U+2155, *VULGAR FRACTION OF FIFTH*. Formellement, les caractères numériques sont ceux avec les priorités *Numeric_Type=Digit*, *Numeric_Type=Decimal*, ou *Numeric_Type=Numeric*.

`str.`**`isprintable`**()

Donne `True` si tous les caractères de la chaîne sont affichables ou qu'elle est vide sinon, `False`. Les caractères non affichables sont les caractères définis dans la base de données de caractères Unicode comme *"Other"* ou *"Separator"*, à l'exception de l'espace ASCII (`0x20`) qui est considéré comme affichable. (Notez que les caractères imprimables dans ce contexte sont ceux qui ne devraient pas être protégés quand `repr()` est invoquée sur une chaîne. Ça n'a aucune incidence sur le traitement des chaînes écrites sur `sys.stdout` ou `sys.stderr`.)

`str.`**`isspace`**()

Donne `True` s'il n'y a que des caractères blancs dans la chaîne et il y a au moins un caractère, sinon `False`. Les caractères blancs sont les caractères définis dans la base de données de caractères Unicode comme *"Other"* ou *"Separator"* ainsi que ceux ayant la propriété bidirectionnelle valant "WS", "B" ou "S".

`str.`**`istitle`**()

Donne `True` si la chaîne est une chaîne *titlecased* et qu'elle contient au moins un caractère, par exemple les caractères majuscules ne peuvent suivre les caractères non capitalisables et les caractères minuscules ne peuvent suivre que des caractères capitalisables. Donne `False` dans le cas contraire.

`str.`**`isupper`**()

Donne `True` si tous les caractères différentiables sur la casse [4] de la chaîne sont en majuscules et il y a au moins un caractère différentiable sur la casse, sinon `False`.

`str.`**`join`**(*iterable*)

Donne une chaîne qui est la concaténation des chaînes contenues dans *iterable*. Une `TypeError` sera levée si une valeur d'*iterable* n'est pas une chaîne, y compris pour les objets `bytes`. Le séparateur entre les éléments est la chaîne fournissant cette méthode.

`str.`**`ljust`**(*width*[, *fillchar*])

Renvoie la chaîne justifiée à gauche dans une chaîne de longueur *width*. Le rembourrage est fait en utilisant *fillchar* (qui par défaut est un espace ASCII). La chaîne d'origine est renvoyée si *width* est inférieur ou égale à `len(s)`.

`str.`**`lower`**()

Renvoie une copie de la chaîne avec tous les caractères capitalisables [4] convertis en minuscules.

L'algorithme de mise en minuscules utilisé est décrit dans la section 3.13 de la norme Unicode.

`str.`**`lstrip`**([*chars*])

Renvoie une copie de la chaîne des caractères supprimés au début. L'argument *chars* est une chaîne spécifiant le jeu de caractères à supprimer. En cas d'omission ou `None`, la valeur par défaut de *chars* permet de supprimer des espaces. L'argument *chars* n'est pas un préfixe, toutes les combinaisons de ses valeurs sont supprimées :

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

static `str.`**`maketrans`**(*x*[, *y*[, *z*]])

Cette méthode statique renvoie une table de traduction utilisable pour `str.translate()`.

Si un seul argument est fourni, ce soit être un dictionnaire faisant correspondre des points de code Unicode (nombres entiers) ou des caractères (chaînes de longueur 1) à des points de code Unicode.

Si deux arguments sont fournis, ce doit être deux chaînes de caractères de même longueur. Le dictionnaire renvoyé fera correspondre pour chaque caractère de x un caractère de y pris à la même place. Si un troisième argument est fourni, ce doit être une chaîne dont chaque caractère correspondra à `None` dans le résultat.

`str.`**`partition`**(*sep*)

Divise la chaîne à la première occurrence de *sep*, et donne un *tuple* de trois éléments contenant la partie

avant le séparateur, le séparateur lui-même, et la partie après le séparateur. Si le séparateur n'est pas trouvé, le *tuple* contiendra la chaîne elle-même, suivie de deux chaînes vides.

`str.`**`replace`**(*old*, *new*[, *count*])

Renvoie une copie de la chaîne dont toutes les occurrences de la sous-chaîne *old* sont remplacés par *new*. Si l'argument optionnel *count* est donné, seules les *count* premières occurrences sont remplacées.

`str.`**`rfind`**(*sub*[, *start*[, *end*]])

Donne l'indice le plus élevé dans la chaîne où la sous-chaîne *sub* se trouve, de telle sorte que *sub* soit contenue dans `s[start:end]`. Les arguments facultatifs *start* et *end* sont interprétés comme dans la notation des *slices*. Donne `-1` en cas d'échec.

`str.`**`rindex`**(*sub*[, *start*[, *end*]])

Comme `rfind()` mais lève une exception `ValueError` lorsque la sous-chaîne *sub* est introuvable.

`str.`**`rjust`**(*width*[, *fillchar*])

Renvoie la chaîne justifié à droite dans une chaîne de longueur *width*. Le rembourrage est fait en utilisant le caractère spécifié par *fillchar* (par défaut est un espace ASCII). La chaîne d'origine est renvoyée si *width* est inférieure ou égale à `len(s)`.

`str.`**`rpartition`**(*sep*)

Divise la chaîne à la dernière occurrence de *sep*, et donne un tuple de trois éléments contenant la partie avant le séparateur, le séparateur lui-même, et la partie après le séparateur. Si le séparateur n'est pas trouvé, le *tuple* contiendra deux chaînes vides, puis par la chaîne elle-même.

`str.`**`rsplit`**(*sep=None*, *maxsplit=-1*)

Renvoie une liste des mots de la chaîne, en utilisant *sep* comme séparateur. Si *maxsplit* est donné, c'est le nombre maximum de divisions qui pourront être faites, celles "à droite". Si *sep* est pas spécifié ou est `None`, tout espace est un séparateur. En dehors du fait qu'il découpe par la droite, `rsplit()` se comporte comme `split()` qui est décrit en détail ci-dessous.

`str.`**`rstrip`**([*chars*])

Renvoie une copie de la chaîne avec des caractères finaux supprimés. L'argument *chars* est une chaîne spécifiant le jeu de caractères à supprimer. En cas d'omission ou `None`, les espaces sont supprimés. L'argument *chars* n'est pas un suffixe : toutes les combinaisons de ses valeurs sont retirées :

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

`str.`**`split`**(*sep=None*, *maxsplit=-1*)

Renvoie une liste des mots de la chaîne, en utilisant *sep* comme séparateur de mots. Si *maxsplit* est donné, c'est le nombre maximum de divisions qui pourront être effectuées, (donnant ainsi une liste de longueur `maxsplit+1`). Si *maxsplit* n'est pas fourni, ou vaut `-1`, le nombre de découpes n'est pas limité (Toutes les découpes possibles sont faites).

Si *sep* est donné, les délimiteurs consécutifs ne sont pas regroupés et ainsi délimitent des chaînes vides (par exemple, `'1,,2'.split(',')` donne `['1', '', '2']`). L'argument *sep* peut contenir plusieurs caractères (par exemple, `'1<>2<>3'.split('<>')` renvoie `['1', '2', '3']`). Découper une chaîne vide en spécifiant *sep* donne `['']`.

Par exemple :

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
```

Si *sep* n'est pas spécifié ou est `None`, un autre algorithme de découpage est appliqué : les espaces consécutifs sont considérés comme un seul séparateur, et le résultat ne contiendra pas les chaînes vides de début ou de la fin si la chaîne est préfixée ou suffixé d'espaces. Par conséquent, diviser une chaîne vide ou une chaîne composée d'espaces avec un séparateur `None` renvoie `[]`.

Par exemple :

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

`str.`**`splitlines`**([*keepends*])

Renvoie les lignes de la chaîne sous forme de liste, la découpe se fait au niveau des limites des lignes. Les sauts de ligne ne sont pas inclus dans la liste des résultats, sauf si *keepends* est donné, et est vrai.

Cette méthode découpe sur les limites de ligne suivantes. Ces limites sont un sur ensemble de universal newlines.

Représentation	Description
`\n`	Saut de ligne
`\r`	Retour chariot
`\r\n`	Retour chariot + saut de ligne
`\v` or `\x0b`	Tabulation verticale
`\f` or `\x0c`	Saut de page
`\x1c`	Séparateur de fichiers
`\x1d`	Séparateur de groupes
`\x1e`	Séparateur d'enregistrements
`\x85`	Ligne suivante (code de contrôle *C1*)
`\u2028`	Séparateur de ligne
`\u2029`	Séparateur de paragraphe

Modifié dans la version 3.2: `\v` et `\f` ajoutés à la liste des limites de lignes.

Par exemple :

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

Contrairement à `split()` lorsque *sep* est fourni, cette méthode renvoie une liste vide pour la chaîne vide, et un saut de ligne à la fin ne se traduit pas par une ligne supplémentaire :

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

À titre de comparaison, `split('\n')` donne :

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

`str.`**`startswith`**(*prefix*[, *start*[, *end*]])

Donne `True` si la chaîne commence par *prefix*, sinon `False`. *prefix* peut aussi être un tuple de préfixes à rechercher. Lorsque *start* est donné, la comparaison commence à cette position, et lorsque *end* est donné, la comparaison s'arrête à celle ci.

`str.`**`strip`**([*chars*])

Donne une copie de la chaîne dont des caractères initiaux et finaux sont supprimés. L'argument *chars* est une chaîne spécifiant le jeu de caractères à supprimer. En cas d'omission ou `None`, les espaces sont supprimés. L'argument *chars* est pas un préfixe ni un suffixe, toutes les combinaisons de ses valeurs sont suppri-

mées :

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

Les caractères de *char* sont retirés du début et de la fin de la chaîne. Les caractères sont retirés de la gauche jusqu'à atteindre un caractère ne figurant pas dans le jeu de caractères dans *chars*. La même opération à lieu par la droite. Par exemple :

```
>>> comment_string = '#....... Section 3.2.1 Issue #32 .......'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

`str.`**`swapcase`**()

Renvoie une copie de la chaîne dont les caractères majuscules sont convertis en minuscules et vice versa. Notez qu'il est pas nécessairement vrai que `s.swapcase().swapcase() == s`.

`str.`**`title`**()

Renvoie une version en initiales majuscules de la chaîne où les mots commencent par une capitale et les caractères restants sont en minuscules.

Par exemple :

```
>>> 'Hello world'.title()
'Hello World'
```

Pour l'algorithme, la notion de mot est définie simplement et indépendamment de la langue comme un groupe de lettres consécutives. La définition fonctionne dans de nombreux contextes, mais cela signifie que les apostrophes (typiquement de la forme possessive en Anglais) forment les limites de mot, ce qui n'est pas toujours le résultat souhaité :

```
>>> "they're bill's friends from the UK".title()
"They're Bill'S Friends From The Uk"
```

Une solution pour contourner le problème des apostrophes peut être obtenue en utilisant des expressions rationnelles :

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

`str.`**`translate`**(*table*)

Renvoie une copie de la chaîne dans laquelle chaque caractère a été changé selon la table de traduction donnée. La table doit être un objet qui implémente l'indexation via `__getitem__()`, typiquement un mapping ou une sequence. Pour un ordinal Unicode (un entier), la table peut donner soit un ordinal Unicode ou une chaîne pour faire correspondre un ou plusieurs caractère au caractère donné, soit `None` pour supprimer le caractère de la chaîne de renvoyée soit lever une exception `LookupError` pour ne pas changer le caractère.

Vous pouvez utiliser `str.maketrans()` pour créer une table de correspondances de caractères dans différents formats.

Voir aussi le module `codecs` pour une approche plus souple de changements de caractères par correspondance.

`str.`**`upper`**()

Renvoie une copie de la chaîne où tous les caractères capitalisables [4] ont été convertis en capitales. Notez que `s.upper().isupper()` peut être `False` si `s` contient des caractères non capitalisables ou si la catégorie Unicode d'un caractère du résultat n'est pas "Lu" (*Letter*, *uppercase*), mais par exemple "Lt" (*Letter*, *titlecase*).

L'algorithme de capitalisation utilisé est décrit dans la section 3.13 de la norme Unicode.

`str.`**`zfill`**(*width*)

Renvoie une copie de la chaîne remplie par la gauche du chiffre (le caractère ASCII) `'0'` pour faire une chaîne de longueur *width*. Un préfixe (`'+'` / `'-'`) est permis par l'insertion du caractère de rembourrage *après* le caractère désigne plutôt qu'avant. La chaîne d'origine est renvoyée si *width* est inférieur ou égale à `len(s)`.

Par exemple :

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

Formatage de chaines à la `printf`

Note: Ces opérations de mise en forme contiennent des bizarreries menant à de nombreuses erreurs classiques (telles que ne pas réussir à afficher des *tuples* ou des dictionnaires correctement). Utiliser les formatted string literals, la méthode `str.format()` ou les template strings aide à éviter ces erreurs. Chacune de ces alternatives apporte son lot d'avantages et inconvénients en matière de simplicité, de flexibilité et/ou de généralisation possible.

Les objets *str* n'exposent qu'une opération : L'opérateur `%` (modulo). Aussi connu sous le nom d'opérateur de formatage, ou opérateur d'interpolation. Étant donné `format % values` (où *format* est une chaîne), les marqueurs `%` de *format* sont remplacés par zéro ou plusieurs éléments de *values*. L'effet est similaire à la fonction `sprintf()` du langage C.

Si *format* ne nécessite qu'un seul argument, *values* peut être un objet unique. [5] Si *values* est un tuple, il doit contenir exactement le nombre d'éléments spécifiés par la chaîne de format, ou un seul objet de correspondances (*mapping object*, par exemple, un dictionnaire).

Un indicateur de conversion contient deux ou plusieurs caractères et comporte les éléments suivants, qui doivent apparaître dans cet ordre :

1. Le caractère `'%'`, qui marque le début du marqueur.
2. La clé de correspondance (facultative), composée d'une suite de caractères entre parenthèse (par exemple, `(somename)`).
3. Des options de conversion, facultatives, qui affectent le résultat de certains types de conversion.
4. Largeur minimum (facultative). Si elle vaut `'*'` (astérisque), la largeur est lue de l'élément suivant du tuple *values*, et l'objet à convertir vient après la largeur de champ minimale et la précision facultative.
5. Précision (facultatif), donnée sous la forme d'un `'.'` (point) suivi de la précision. Si la précision est `'*'` (un astérisque), la précision est lue à partir de l'élément suivant du tuple *values* et la valeur à convertir vient ensuite.
6. Modificateur de longueur (facultatif).
7. Type de conversion.

Lorsque l'argument de droite est un dictionnaire (ou un autre type de *mapping*), les marqueurs dans la chaîne *doivent* inclure une clé présente dans le dictionnaire, écrite entre parenthèses, immédiatement après le caractère `'%'`. La clé indique quelle valeur du dictionnaire doit être formatée. Par exemple :

```
>>> print('%(language)s has %(number)03d quote types.' %
...       {'language': "Python", "number": 2})
Python has 002 quote types.
```

Dans ce cas, aucune `*` ne peuvent se trouver dans le format (car ces `*` nécessitent une liste (accès séquentiel) de paramètres).

Les caractères indicateurs de conversion sont :

Option	Signification
`'#'`	La conversion utilisera la "forme alternative" (définie ci-dessous).
`'0'`	Les valeurs numériques converties seront complétée de zéros.
`'-'`	La valeur convertie est ajustée à gauche (remplace la conversion `'0'` si les deux sont données).

Option	Signification
`' '`	(un espace) Un espace doit être laissé avant un nombre positif (ou chaîne vide) produite par la conversion d'une valeur signée.
`'+'`	Un caractère de signe (`'+'` ou `'-'`) précède la valeur convertie (remplace le marqueur "espace").

Un modificateur de longueur (`h`, `l` ou `L`) peut être présent, mais est ignoré car il est pas nécessaire pour Python, donc par exemple `%ld` est identique à `%d`.

Les types utilisables dans les conversion sont :

Conversion	Signification	Notes
`'d'`	Entier décimal signé.	
`'i'`	Entier décimal signé.	
`'o'`	Valeur octale signée.	(1)
`'u'`	Type obsolète — identique à `'d'`.	(6)
`'x'`	Hexadécimal signé (en minuscules).	(2)
`'X'`	Hexadécimal signé (capitales).	(2)
`'e'`	Format exponentiel pour un *float* (minuscule).	(3)
`'E'`	Format exponentiel pour un *float* (en capitales).	(3)
`'f'`	Format décimal pour un *float*.	(3)
`'F'`	Format décimal pour un *float*.	(3)
`'g'`	Format *float*. Utilise le format exponentiel minuscules si l'exposant est inférieur à `-4` ou pas plus petit que la précision, sinon le format décimal.	(4)
`'G'`	Format *float*. Utilise le format exponentiel en capitales si l'exposant est inférieur à `-4` ou pas plus petit que la précision, sinon le format décimal.	(4)
`'c'`	Un seul caractère (accepte des entiers ou une chaîne d'un seul caractère).	
`'r'`	String (convertit n'importe quel objet Python avec `repr()`).	(5)
`'s'`	String (convertit n'importe quel objet Python avec `str()`).	(5)
`'a'`	String (convertit n'importe quel objet Python en utilisant `ascii()`).	(5)
`'%'`	Aucun argument n'est converti, donne un caractère de `'%'` dans le résultat.	

Notes :

1. La forme alternative entraîne l'insertion d'un préfixe octal (`'0o'`) avant le premier chiffre.
2. La forme alternative entraîne l'insertion d'un préfixe `'0x'` ou `'0X'` (respectivement pour les formats `'x'` et `'X'`) avant le premier chiffre.
3. La forme alternative implique la présence d'un point décimal, même si aucun chiffre ne le suit.

   La précision détermine le nombre de chiffres après la virgule, 6 par défaut.
4. La forme alternative implique la présence d'un point décimal et les zéros non significatifs sont conservés (ils ne le seraient pas autrement).

   La précision détermine le nombre de chiffres significatifs avant et après la virgule. 6 par défaut.
5. Si la précision est `N`, la sortie est tronquée à `N` caractères.
6. Voir la **PEP 237**.

Puisque les chaînes Python ont une longueur explicite, les conversions `%s` ne considèrent pas `'\0'` comme la fin de la chaîne.

Modifié dans la version 3.1: Les conversions `%f` pour nombres dont la valeur absolue est supérieure à `1e50` ne sont plus remplacés par des conversions `%g`.

Séquences Binaires --- `bytes`, `bytearray`, `memoryview`

Les principaux types natifs pour manipuler des données binaires sont `bytes` et `bytearray`. Ils sont supportés par `memoryview` qui utilise le buffer protocol pour accéder à la mémoire d'autres objets binaires sans avoir besoin d'en faire une copie.

Le module `array` permet le stockage efficace de types basiques comme les entiers de 32 bits et les *float* double précision IEEE754.

Objets *bytes*

Les *bytes* sont des séquences immuables d'octets. Comme beaucoup de protocoles binaires utilisent l'ASCII, les objets *bytes* offrent plusieurs méthodes qui ne sont valables que lors de la manipulation de données ASCII et sont étroitement liés aux objets *str* dans bien d'autres aspects.

class **`bytes`**([*source*[, *encoding*[, *errors*]]])

Tout d'abord, la syntaxe des *bytes* littéraux est en grande partie la même que pour les chaînes littérales, en dehors du préfixe `b` :

- Les guillemets simples : `b'autorisent aussi les guillemets "doubles"'`
- Les guillemets doubles : `b"permettent aussi les guillemets 'simples'"`.
- Les guillemets triples : `b'''3 single quotes'''`, `b"""3 double quotes"""`

Seuls les caractères ASCII sont autorisés dans les littéraux de *bytes* (quel que soit l'encodage du code source déclaré). Toutes les valeurs au delà de 127 doivent être entrés dans littéraux de *bytes* en utilisant une séquence d'échappement appropriée.

Comme avec les chaînes littérales, les *bytes* littéraux peuvent également utiliser un préfixe `r` pour désactiver le traitement des séquences d'échappement. Voir Littéraux de chaînes de caractères et de suites d'octets pour plus d'informations sur les différentes formes littérales de *bytes*, y compris les séquences d'échappement supportées.

Bien que les *bytes* littéraux, et leurs représentation, soient basés sur du texte ASCII, les *bytes* se comportent en fait comme des séquences immuables de nombres entiers, dont les valeurs sont restreintes dans `0 <= x < 256` (ne pas respecter cette restriction lève une `ValueError`. Ceci est fait délibérément afin de souligner que, bien que de nombreux encodages binaires soient compatibles avec l'ASCII, et peuvent être manipulés avec des algorithmes orientés texte, ce n'est généralement pas le cas pour les données binaires arbitraires (appliquer aveuglément des algorithmes de texte sur des données binaires qui ne sont pas compatibles ASCII conduit généralement à leur corruption).

En plus des formes littérales, des objets *bytes* peuvent être créés par de nombreux moyens :

- Un objet *bytes* rempli de zéros d'une longueur spécifiée : `bytes(10)`
- D'un itérable d'entiers : `bytes(range(20))`
- Copier des données binaires existantes via le *buffer protocol* : `bytes(obj)`

Voir aussi la fonction native bytes.

Puisque 2 chiffres hexadécimaux correspondent précisément à un seul octet, les nombres hexadécimaux sont un format couramment utilisé pour décrire les données binaires. Par conséquent, le type *bytes* a une méthode de classe pour lire des données dans ce format :

classmethod **`fromhex`**(*string*)

Cette méthode de la classe `bytes` renvoie un objet *bytes*, décodant la chaîne donnée. La chaîne doit contenir deux chiffres hexadécimaux par octet, les espaces ASCII sont ignorés.

```
>>> bytes.fromhex('2Ef0 F1f2  ')
b'.\xf0\xf1\xf2'
```

Modifié dans la version 3.7: `bytes.fromhex()` saute maintenant dans la chaîne tous les caractères ASCII "blancs", pas seulement les espaces.

Une fonction de conversion inverse existe pour transformer un objet *bytes* en sa représentation hexadécimale.

`hex`()

Renvoie une chaîne contenant deux chiffres hexadécimaux pour chaque octet du *byte*.

```
>>> b'\xf0\xf1\xf2'.hex()
```

```
'f0f1f2'
```

Nouveau dans la version 3.5.

Comme les objets *bytes* sont des séquences d'entiers (semblables à un tuple), pour une instance de *bytes b*, `b[0]` sera un entier, tandis que``b[0:1]`` sera un objet *bytes* de longueur 1. (Cela contraste avec les chaînes, où l'indexation et le *slicing* donne une chaîne de longueur 1)

La représentation des *bytes* utilise le format littéral (`b'...'`) car il est souvent plus utile que par exemple `bytes([46, 46, 46])`. Vous pouvez toujours convertir un *bytes* en liste d'entiers en utilisant `list(b)`.

Note: Pour les utilisateurs de Python 2.x : Dans la série 2.x de Python, une variété de conversions implicites entre les chaînes 8-bit (la chose la plus proche d'un type natif de données binaires offert par Python 2) et des chaînes Unicode étaient permises. C'était solution de contournement, pour garder la rétro-compatibilité, par rapport au fait que Python ne prenait initialement en charge que le texte 8 bits, le texte Unicode est un ajout ultérieur. En Python 3.x, ces conversions implicites ont disparues, les conversions entre les données binaires et texte Unicode doivent être explicites, et les *bytes* sont toujours différents des chaînes.

Objets *bytearray*

Les objets `bytearray` sont l'équivalent muable des objets `bytes`.

class **`bytearray`**([*source*[, *encoding*[, *errors*]]])

Il n'y a pas de syntaxe littérale dédiée aux *bytearray*, ils sont toujours créés en appelant le constructeur :

- Créer une instance vide: `bytearray()`
- Créer une instance remplie de zéros d'une longueur donnée : `bytearray(10)`
- À partir d'un itérable d'entiers : `bytearray(range(20))`
- Copie des données binaires existantes via le *buffer protocol* : `bytearray(b'Hi!')`

Comme les *bytearray* sont muables, ils prennent en charge les opérations de séquence muables en plus des opérations communes de *bytes* et *bytearray* décrites dans Opérations sur les bytes et bytearray.

Voir aussi la fonction native bytearray.

Puisque 2 chiffres hexadécimaux correspondent précisément à un octet, les nombres hexadécimaux sont un format couramment utilisé pour décrire les données binaires. Par conséquent, le type *bytearray* a une méthode de classe pour lire les données dans ce format :

classmethod **`fromhex`**(*string*)

Cette méthode de la classe `bytearray` renvoie un objet *bytearray*, décodant la chaîne donnée. La chaîne doit contenir deux chiffres hexadécimaux par octet, les espaces ASCII sont ignorés.

```
>>> bytearray.fromhex('2Ef0 F1f2  ')
bytearray(b'.\xf0\xf1\xf2')
```

Modifié dans la version 3.7: `bytearray.fromhex()` saute maintenant tous les caractères "blancs" ASCII dans la chaîne, pas seulement les espaces.

Une fonction de conversion inverse existe pour transformer un objet *bytearray* en sa représentation hexadécimale.

`hex`()

Renvoie une chaîne contenant deux chiffres hexadécimaux pour chaque octet du *byte*.

```
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

Nouveau dans la version 3.5.

Comme les *bytearray* sont des séquences d'entiers (semblables à une liste), pour un objet *bytearray b*, `b[0]` sera un entier, tandis que `b[0:1]` sera un objet *bytearray* de longueur 1. (Ceci contraste avec les chaînes de texte, où l'indexation et le *slicing* produit une chaîne de longueur 1)

La représentation des objets *bytearray* utilise le format littéral des *bytes* (`bytearray(b'...')`) car il est souvent plus utile que par exemple `bytearray([46, 46, 46])`. Vous pouvez toujours convertir un objet *bytearray* en une liste de nombres entiers en utilisant `list(b)`.

Opérations sur les *bytes* et *bytearray*

bytes et *bytearray* prennent en charge les opérations communes des séquences. Ils interagissent non seulement avec des opérandes de même type, mais aussi avec les bytes-like object. En raison de cette flexibilité, ils peuvent être mélangés librement dans des opérations sans provoquer d'erreurs. Cependant, le type du résultat peut dépendre de l'ordre des opérandes.

Note: Les méthodes sur les *bytes* et les *bytearray* n'acceptent pas les chaînes comme arguments, tout comme les méthodes sur les chaînes n'acceptent pas les *bytes* comme arguments. Par exemple, vous devez écrire :

```
a = "abc"
b = a.replace("a", "f")
```

et :

```
a = b"abc"
b = a.replace(b"a", b"f")
```

Quelques opérations de *bytes* et *bytesarray* supposent l'utilisation de formats binaires compatibles ASCII, et donc doivent être évités lorsque vous travaillez avec des données binaires arbitraires. Ces restrictions sont couvertes ci-dessous.

Note: Utiliser ces opérations basées sur l'ASCII pour manipuler des données binaires qui ne sont pas au format ASCII peut les corrompre.

Les méthodes suivantes sur les *bytes* et *bytearray* peuvent être utilisées avec des données binaires arbitraires.

`bytes.`**`count`**(*sub*[, *start*[, *end*]])
`bytearray.`**`count`**(*sub*[, *start*[, *end*]])

Renvoie le nombre d'occurrences qui ne se chevauchent pas de la sous-séquence *sub* dans l'intervalle [*start*, *end*]. Les arguments facultatifs *start* et *end* sont interprétés comme pour un *slice*.

La sous-séquence à rechercher peut être un quelconque bytes-like object ou un nombre entier compris entre 0 et 255.

Modifié dans la version 3.3: Accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

`bytes.`**`decode`**(*encoding="utf-8"*, *errors="strict"*)
`bytearray.`**`decode`**(*encoding="utf-8"*, *errors="strict"*)

Décode les octets donnés, et le renvoie sous forme d'une chaîne de caractères. L'encodage par défaut est `'utf-8'`. *errors* peut être donné pour changer de système de gestion des erreurs. Sa valeur par défaut est `'strict'`, ce qui signifie que les erreurs d'encodage lèvent une `UnicodeError`. Les autres valeurs possibles sont `'ignore'`, `'replace'` et tout autre nom enregistré via `codecs.register_error()`, voir la section Error Handlers. Pour une liste des encodages possibles, voir la section Standard Encodings.

Note: Passer l'argument *encoding* à `str` permet de décoder tout bytes-like object directement, sans avoir besoin d'utiliser un *bytes* ou *bytearray* temporaire.

Modifié dans la version 3.1: Gère les arguments nommés.

`bytes.`**`endswith`**(*suffix*[, *start*[, *end*]])
`bytearray.`**`endswith`**(*suffix*[, *start*[, *end*]])

Donne `True` si les octets se terminent par *suffix*, sinon `False`. *suffix* peut aussi être un tuple de suffixes à rechercher. Avec l'argument optionnel *start*, la recherche se fait à partir de cette position. Avec l'argument optionnel *end*, la comparaison s'arrête à cette position.

Les suffixes à rechercher peuvent être n'importe quel bytes-like object.

`bytes.`**`find`**(*sub*[, *start*[, *end*]])
`bytearray.`**`find`**(*sub*[, *start*[, *end*]])

Donne la première position où le *sub* se trouve dans les données, de telle sorte que *sub* soit contenue dans `s[start:end]`. Les arguments facultatifs *start* et *end* sont interprétés comme dans la notation des *slices*. Donne `-1` si *sub* n'est pas trouvé.

La sous-séquence à rechercher peut être un quelconque bytes-like object ou un nombre entier compris entre 0 et 255.

Note: La méthode `find()` ne doit être utilisée que si vous avez besoin de connaître la position de *sub*. Pour vérifier si *sub* est présent ou non, utilisez l'opérateur `in` :

```
>>> b'Py' in b'Python'
True
```

Modifié dans la version 3.3: Accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

`bytes.`**`index`**(*sub*[, *start*[, *end*]])
`bytearray.`**`index`**(*sub*[, *start*[, *end*]])

Comme `find()`, mais lève une `ValueError` lorsque la séquence est introuvable.

La sous-séquence à rechercher peut être un quelconque bytes-like object ou un nombre entier compris entre 0 et 255.

Modifié dans la version 3.3: Accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

`bytes.`**`join`**(*iterable*)
`bytearray.`**`join`**(*iterable*)

Donne un *bytes* ou *bytearray* qui est la concaténation des séquences de données binaires dans *iterable*. Une exception `TypeError` est levée si une valeur d'*iterable* n'est pas un bytes-like objects, y compris pour des `str`. Le séparateur entre les éléments est le contenu du *bytes* ou du *bytearray* depuis lequel cette méthode est appelée.

static `bytes.`**`maketrans`**(*from*, *to*)
static `bytearray.`**`maketrans`**(*from*, *to*)

Cette méthode statique renvoie une table de traduction utilisable par `bytes.translate()` qui permettra de changer chaque caractère de *from* par un caractère à la même position dans *to*; *from* et *to* doivent tous deux être des bytes-like objects et avoir la même longueur.

Nouveau dans la version 3.1.

`bytes.`**`partition`**(*sep*)
`bytearray.`**`partition`**(*sep*)

Divise la séquence à la première occurrence de *sep*, et renvoie un 3-tuple contenant la partie précédant le séparateur, le séparateur lui même (ou sa copie en *byterray*), et la partie suivant le séparateur. Si le séparateur est pas trouvé, le 3-tuple renvoyé contiendra une copie de la séquence d'origine, suivi de deux *bytes* ou *bytearray* vides.

Le séparateur à rechercher peut être tout bytes-like object.

`bytes.`**`replace`**(*old*, *new*[, *count*])
`bytearray.`**`replace`**(*old*, *new*[, *count*])

Renvoie une copie de la séquence dont toutes les occurrences de la sous-séquence *old* sont remplacées par *new*. Si l'argument optionnel *count* est donné, seules les *count* premières occurrences de sont remplacés.

La sous-séquence à rechercher et son remplacement peuvent être n'importe quel bytes-like object.

Note: La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.`**`rfind`**(*sub*[, *start*[, *end*]])
`bytearray.`**`rfind`**(*sub*[, *start*[, *end*]])

Donne la plus grande position de *sub* dans la séquence, de telle sorte que *sub* soit dans `s[start:end]`. Les arguments facultatifs *start* et *end* sont interprétés comme dans la notation des *slices*. Donne `-1` si *sub* n'est pas trouvable.

La sous-séquence à rechercher peut être un quelconque bytes-like object ou un nombre entier compris entre 0 et 255.

Modifié dans la version 3.3: Accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

`bytes.`**`rindex`**(*sub*[, *start*[, *end*]])
`bytearray.`**`rindex`**(*sub*[, *start*[, *end*]])

Semblable à `rfind()` mais lève une `ValueError` lorsque *sub* est introuvable.

La sous-séquence à rechercher peut être un quelconque bytes-like object ou un nombre entier compris entre 0 et 255.

Modifié dans la version 3.3: Accepte aussi un nombre entier compris entre 0 et 255 comme sous-séquence.

`bytes.`**`rpartition`**(*sep*)
`bytearray.`**`rpartition`**(*sep*)

Coupe la séquence à la dernière occurrence de *sep*, et renvoie un triplet de trois éléments contenant la partie précédent le séparateur, le séparateur lui même (ou sa copie, un *bytearray*), et la partie suivant le séparateur. Si le séparateur n'est pas trouvé, le triplet contiendra deux *bytes* ou *bytesarray* vides suivi d'une copie de la séquence d'origine.

Le séparateur à rechercher peut être tout bytes-like object.

`bytes.`**`startswith`**(*prefix*[, *start*[, *end*]])
`bytearray.`**`startswith`**(*prefix*[, *start*[, *end*]])

Donne `True` si les données binaires commencent par le *prefix* spécifié, sinon `False`. *prefix* peut aussi être un tuple de préfixes à rechercher. Avec l'argument *start* la recherche commence à cette position. Avec l'argument *end* option, la recherche s'arrête à cette position.

Le préfixe(s) à rechercher peuvent être n'importe quel bytes-like object.

`bytes.`**`translate`**(*table*, *delete=b''*)
`bytearray.`**`translate`**(*table*, *delete=b''*)

Renvoie une copie du *bytes* ou *bytearray* dont tous les octets de *delete* sont supprimés, et les octets restants changés par la table de correspondance donnée, qui doit être un objet *bytes* d'une longueur de 256.

Vous pouvez utiliser la méthode `bytes.maketrans()` pour créer une table de correspondance.

Donnez `None` comme *table* pour seulement supprimer des caractères :

```
>>> b'read this short text'.translate(None, b'aeiou')
b'rd ths shrt txt'
```

Modifié dans la version 3.6: delete est maintenant accepté comme argument nommé.

Les méthodes suivantes sur les *bytes* et *bytearray* supposent par défaut que les données traitées sont compatibles ASCII, mais peuvent toujours être utilisées avec des données binaires, arbitraires, en passant des arguments appropriés. Notez que toutes les méthodes de *bytearray* de cette section ne travaillent jamais sur l'objet lui même, mais renvoient un nouvel objet.

`bytes.`**`center`**(*width*[, *fillbyte*])
`bytearray.`**`center`**(*width*[, *fillbyte*])

Renvoie une copie de l'objet centrée dans une séquence de longueur *width*. Le remplissage est fait en utilisant *fillbyte* (qui par défaut est un espace ASCII). Pour les objets `bytes`, la séquence initiale est renvoyée si *width* est inférieur ou égal à `len(s)`.

Note: La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.`**`ljust`**(*width*[, *fillbyte*])
`bytearray.`**`ljust`**(*width*[, *fillbyte*])

Renvoie une copie de l'objet aligné à gauche dans une séquence de longueur *width*. Le remplissage est fait en utilisant *fillbyte* (par défaut un espace ASCII). Pour les objets `bytes`, la séquence initiale est renvoyée si *width* est inférieure ou égale à `len(s)`.

Note: La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.`**`lstrip`**([*chars*])
`bytearray.`**`lstrip`**([*chars*])

Renvoie une copie de la séquence dont certains préfixes ont été supprimés. L'argument *chars* est une séquence binaire spécifiant le jeu d'octets à supprimer. Ce nom se réfère au fait de cette méthode est généralement utilisée avec des caractères ASCII. En cas d'omission ou `None`, la valeur par défaut de *chars* permet de supprimer des espaces ASCII. L'argument *chars* n'est pas un préfixe, toutes les combinaisons de ses valeurs sont supprimées :

```
>>> b'   spacious   '.lstrip()
b'spacious   '
>>> b'www.example.com'.lstrip(b'cmowz.')
b'example.com'
```

La séquence de valeurs à supprimer peut être tout bytes-like object.

Note: La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.`**`rjust`**(*width*[, *fillbyte*])
`bytearray.`**`rjust`**(*width*[, *fillbyte*])

Renvoie une copie de l'objet justifié à droite dans une séquence de longueur *width*. Le remplissage est fait en utilisant le caractère *fillbyte* (par défaut est un espace ASCII). Pour les objets `bytes`, la séquence d'origine est renvoyée si *width* est inférieure ou égale à `len(s)`.

Note: La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.`**`rsplit`**(*sep=None*, *maxsplit=-1*)
`bytearray.`**`rsplit`**(*sep=None*, *maxsplit=-1*)

Divise la séquence d'octets en sous-séquences du même type, en utilisant *sep* comme séparateur. Si *maxsplit* est donné, c'est le nombre maximum de divisions qui pourront être faites, celles "à droite". Si *sep* est pas spécifié ou est `None`, toute sous-séquence composée uniquement d'espaces ASCII est un séparateur. En dehors du fait qu'il découpe par la droite, `rsplit()` se comporte comme `split()` qui est décrit en détail ci-dessous.

`bytes.`**`rstrip`**([*chars*])
`bytearray.`**`rstrip`**([*chars*])

Renvoie une copie de la séquence dont des octets finaux sont supprimés. L'argument *chars* est une séquence d'octets spécifiant le jeu de caractères à supprimer. En cas d'omission ou `None`, les espaces ASCII sont supprimés. L'argument *chars* n'est pas un suffixe : toutes les combinaisons de ses valeurs sont retirées :

```
>>> b'   spacious   '.rstrip()
b'   spacious'
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

La séquence de valeurs à supprimer peut être tout bytes-like object.

Note: La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.`**`split`**(*sep=None*, *maxsplit=-1*)
`bytearray.`**`split`**(*sep=None*, *maxsplit=-1*)

Divise la séquence en sous-séquences du même type, en utilisant *sep* comme séparateur. Si *maxsplit* est donné, c'est le nombre maximum de divisions qui pourront être faites (la liste aura donc au plus `maxsplit+1` éléments), Si *maxsplit* n'est pas spécifié ou faut `-1`, il n'y a aucune limite au nombre de découpes (elles sont toutes effectuées).

Si *sep* est donné, les délimiteurs consécutifs ne sont pas regroupés et ainsi délimitent ainsi des chaînes vides (par exemple, `b'1,,2'.split(b',')` donne `[b'1', b'', b'2']`). L'argument *sep* peut contenir plusieurs sous séquences (par exemple, `b'1<>2<>3'.split(b'<>')` renvoie `[b'1', b'2', b'3']`). Découper une chaîne vide en spécifiant *sep* donne `[b'']` ou `[bytearray(b'')]` en fonction du type de l'objet découpé. L'argument *sep* peut être n'importe quel bytes-like object.

Par exemple :

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,2,,3,'.split(b',')
[b'1', b'2', b'', b'3', b'']
```

Si *sep* n'est pas spécifié ou est `None`, un autre algorithme de découpe est appliqué : les espaces ASCII consécutifs sont considérés comme un seul séparateur, et le résultat ne contiendra pas les chaînes vides de début ou de la fin si la chaîne est préfixée ou suffixé d'espaces. Par conséquent, diviser une séquence vide ou une séquence composée d'espaces ASCII avec un séparateur `None` renvoie `[]`.

Par exemple :

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b'   1   2   3   '.split()
[b'1', b'2', b'3']
```

`bytes.`**`strip`**([*chars*])
`bytearray.`**`strip`**([*chars*])

Renvoie une copie de la séquence dont des caractères initiaux et finaux sont supprimés. L'argument *chars* est une séquence spécifiant le jeu d'octets à supprimer, le nom se réfère au fait de cette méthode est généralement utilisée avec des caractères ASCII. En cas d'omission ou `None`, les espaces ASCII sont supprimés. L'argument *chars* n'est ni un préfixe ni un suffixe, toutes les combinaisons de ses valeurs sont supprimées :

```
>>> b'   spacious   '.strip()
b'spacious'
>>> b'www.example.com'.strip(b'cmowz.')
b'example'
```

La séquence de valeurs à supprimer peut être tout bytes-like object.

Note: La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

Les méthodes suivantes sur les *bytes* et *bytearray* supposent l'utilisation d'un format binaire compatible ASCII, et donc doivent être évités lorsque vous travaillez avec des données binaires arbitraires. Notez que toutes les méthodes de *bytearray* de cette section *ne modifient pas* les octets, ils produisent de nouveaux objets.

`bytes.`**`capitalize`**()
`bytearray.`**`capitalize`**()

Renvoie une copie de la séquence dont chaque octet est interprété comme un caractère ASCII, le premier octet en capitale et le reste en minuscules. Les octets non ASCII ne sont pas modifiés.

Note: La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.`**`expandtabs`**(*tabsize=8*)
`bytearray.`**`expandtabs`**(*tabsize=8*)

Renvoie une copie de la séquence où toutes les tabulations ASCII sont remplacées par un ou plusieurs espaces ASCII, en fonction de la colonne courante et de la taille de tabulation donnée. Les positions des tabulations se trouvent tous les *tabsize* caractères (8 par défaut, ce qui donne les positions de tabulations aux colonnes 0, 8, 16 et ainsi de suite). Pour travailler sur la séquence, la colonne en cours est mise à zéro et la séquence est examinée octets par octets. Si l'octet est une tabulation ASCII (`b'\t'`), un ou plusieurs espaces sont insérés au résultat jusqu'à ce que la colonne courante soit égale à la position de tabulation suivante. (Le caractère tabulation lui-même n'est pas copié.) Si l'octet courant est un saut de ligne ASCII (`b'\n'`) ou un retour chariot (`b'\r'`), il est copié et la colonne en cours est remise à zéro. Tout autre octet est copié inchangé et la colonne en cours est incrémentée de un indépendamment de la façon dont l'octet est représenté lors de l'affichage :

```
>>> b'01\t012\t0123\t01234'.expandtabs()
b'01      012     0123    01234'
```

```
>>> b'01\t012\t0123\t01234'.expandtabs(4)
b'01  012 0123    01234'
```

Note: La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

bytes.**isalnum**()
bytearray.**isalnum**()

Donne `True` si tous les caractères de la chaîne sont des caractères ASCII alphabétiques ou chiffres. et que la séquence n'est pas vide, sinon `False`. Les caractères ASCII alphabétiques sont les suivants `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'` et les chiffres : `b'0123456789'`.

Par exemple :

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

bytes.**isalpha**()
bytearray.**isalpha**()

Donne `True` si tous les octets dans la séquence sont des caractères alphabétiques ASCII et la que la séquence n'est pas vide, sinon `False`. Les caractères ASCIIalphabétiques sont : `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Par exemple :

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

bytes.**isascii**()
bytearray.**isascii**()

Renvoie `True` si la séquence est vide, ou si tous ses octets sont des octets ASCII, renvoie faux dans le cas contraire. Les octets ASCII dans l'intervalle `0---0x7F`.

Nouveau dans la version 3.7.

bytes.**isdigit**()
bytearray.**isdigit**()

Donne `True` si tous les octets de la séquence sont des chiffres ASCII et que la séquence n'est pas vide, sinon `False`. Les chiffres ASCII sont `b'0123456789'`.

Par exemple :

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

bytes.**islower**()
bytearray.**islower**()

Donne `True` s'il y a au moins un caractère ASCII minuscule dans la séquence et aucune capitale, sinon `False`.

Par exemple :

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

bytes.**isspace**()

bytearray.**isspace**()

Donne `True` si tous les octets de la séquence sont des espaces ASCII et que la séquence n'est pas vide, sinon `False`. Les espèces ASCII sont `b' \t\n\r\x0b\f'` (espace, tabulation,saut de ligne, retour chariot, tabulation verticale, *form feed*).

bytes.**istitle**()
bytearray.**istitle**()

Donne `True` si la séquence ASCII est *titlecased*, et qu'elle ne soit pas vide, sinon `False`. Voir `bytes.title()` pour plus de détails sur la définition de *titlecase*.

Par exemple :

```
>>> b'Hello World'.istitle()
True
>>> b'Hello world'.istitle()
False
```

bytes.**isupper**()
bytearray.**isupper**()

Donne `True` s'il y a au moins un caractère alphabétique majuscule ASCII dans la séquence et aucun caractères ASCII minuscules, sinon `False`.

Par exemple :

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

bytes.**lower**()
bytearray.**lower**()

Renvoie une copie de la séquence dont tous les caractères ASCII en majuscules sont convertis en leur équivalent en minuscules.

Par exemple :

```
>>> b'Hello World'.lower()
b'hello world'
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Note: La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

bytes.**splitlines**(*keepends=False*)
bytearray.**splitlines**(*keepends=False*)

Renvoie une liste des lignes de la séquence d'octets, découpant au niveau des fin de lignes ASCII. Cette méthode utilise l'approche universal newlines pour découper les lignes. Les fins de ligne ne sont pas inclus dans la liste des résultats, sauf si *keepends* est donné et vrai.

Par exemple :

```
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
[b'ab c\n', b'\n', b'de fg\r', b'kl\r\n']
```

Contrairement à `split()` lorsque le délimiteur *sep* est fourni, cette méthode renvoie une liste vide pour la chaîne vide, et un saut de ligne à la fin ne se traduit pas par une ligne supplémentaire :

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([b''], [b'Two lines', b''])
>>> b"".splitlines(), b"One line\n".splitlines()
```

```
([], [b'One line'])
```

bytes.**swapcase**()
bytearray.**swapcase**()

Renvoie une copie de la séquence dont tous les caractères ASCII minuscules sont convertis en majuscules et vice-versa.

Par exemple :

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

Contrairement à `str.swapcase()`, `bin.swapcase().swapcase() == bin` est toujours vrai. Les conversions majuscule/minuscule en ASCII étant toujours symétrique, ce qui n'est pas toujours vrai avec Unicode.

Note: La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

bytes.**title**()
bytearray.**title**()

Renvoie une version *titlecased* de la séquence d'octets où les mots commencent par un caractère ASCII majuscule et les caractères restants sont en minuscules. Les octets non capitalisables ne sont pas modifiés.

Par exemple :

```
>>> b'Hello world'.title()
b'Hello World'
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les caractères ASCII majuscules sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. Aucun autre octet n'est capitalisable.

Pour l'algorithme, la notion de mot est définie simplement et indépendamment de la langue comme un groupe de lettres consécutives. La définition fonctionne dans de nombreux contextes, mais cela signifie que les apostrophes (typiquement de la forme possessive en Anglais) forment les limites de mot, ce qui n'est pas toujours le résultat souhaité :

```
>>> b"they're bill's friends from the UK".title()
b"They'Re Bill'S Friends From The Uk"
```

Une solution pour contourner le problème des apostrophes peut être obtenue en utilisant des expressions rationnelles :

```
>>> import re
>>> def titlecase(s):
...     return re.sub(rb"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0:1].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase(b"they're bill's friends.")
b"They're Bill's Friends."
```

Note: La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

bytes.**upper**()
bytearray.**upper**()

Renvoie une copie de la séquence dont tous les caractères ASCII minuscules sont convertis en leur équivalent majuscule.

Par exemple :

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

Les caractères ASCII minuscules sont `b'abcdefghijklmnopqrstuvwxyz'`. Les capitales ASCII sont `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

> **Note:** La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

`bytes.`**`zfill`**(*width*)
`bytearray.`**`zfill`**(*width*)

Renvoie une copie de la séquence remplie par la gauche du chiffre `b'0'` pour en faire une séquence de longueur *width*. Un préfixe (`b'+'` / `b'-'`) est permis par l'insertion du caractère de remplissage *après* le caractère de signe plutôt qu'avant. Pour les objets `bytes` la séquence d'origine est renvoyée si *width* est inférieur ou égale à `len(seq)`.

Par exemple :

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

> **Note:** La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

Formatage de *bytes* a la `printf`

> **Note:** Les opérations de formatage décrites ici présentent une variété de bizarreries qui conduisent à un certain nombre d'erreurs classiques (typiquement, échouer à afficher des tuples ou des dictionnaires correctement). Si la valeur à afficher peut être un tuple ou un dictionnaire, mettez le a l'intérieur d'un autre tuple.

Les objets *bytes* (`bytes` et `bytearray`) ont un unique opérateur : l'opérateur `%` (modulo). Il est aussi connu sous le nom d'opérateur de mise en forme. Avec `format % values` (où *format* est un objet *bytes*), les marqueurs de conversion `%` dans *format* sont remplacés par zéro ou plus de *values*. L'effet est similaire à la fonction `sprintf()` du langage C.

Si *format* ne nécessite qu'un seul argument, *values* peut être un objet unique. [5] Si *values* est un tuple, il doit contenir exactement le nombre d'éléments spécifiés dans le format en *bytes*, ou un seul objet de correspondances (*mapping object*, par exemple, un dictionnaire).

Un indicateur de conversion contient deux ou plusieurs caractères et comporte les éléments suivants, qui doivent apparaître dans cet ordre :

1. Le caractère `'%'`, qui marque le début du marqueur.
2. La clé de correspondance (facultative), composée d'une suite de caractères entre parenthèse (par exemple, `(somename)`).
3. Des options de conversion, facultatives, qui affectent le résultat de certains types de conversion.
4. Largeur minimum (facultative). Si elle vaut `'*'` (astérisque), la largeur est lue de l'élément suivant du tuple *values*, et l'objet à convertir vient après la largeur de champ minimale et la précision facultative.
5. Précision (facultatif), donnée sous la forme d'un `'.'` (point) suivi de la précision. Si la précision est `'*'` (un astérisque), la précision est lue à partir de l'élément suivant du tuple *values* et la valeur à convertir vient ensuite.
6. Modificateur de longueur (facultatif).
7. Type de conversion.

Lorsque l'argument de droite est un dictionnaire (ou un autre type de *mapping*), les marqueurs dans le *bytes doivent* inclure une clé présente dans le dictionnaire, écrite entre parenthèses, immédiatement après le caractère `'%'`. La clé indique quelle valeur du dictionnaire doit être formatée. Par exemple :

```
>>> print(b'%(language)s has %(number)03d quote types.' %
...       {b'language': b"Python", b"number": 2})
b'Python has 002 quote types.'
```

Dans ce cas, aucune `*` ne peuvent se trouver dans le format (car ces `*` nécessitent une liste (accès séquentiel) de paramètres).

Les caractères indicateurs de conversion sont :

Option	Signification
`'#'`	La conversion utilisera la "forme alternative" (définie ci-dessous).
`'0'`	Les valeurs numériques converties seront complétée de zéros.
`'-'`	La valeur convertie est ajustée à gauche (remplace la conversion `'0'` si les deux sont données).
`' '`	(un espace) Un espace doit être laissé avant un nombre positif (ou chaîne vide) produite par la conversion d'une valeur signée.
`'+'`	Un caractère de signe (`'+'` ou `'-'`) précède la valeur convertie (remplace le marqueur "espace").

Un modificateur de longueur (`h`, `l` ou `L`) peut être présent, mais est ignoré car il est pas nécessaire pour Python, donc par exemple `%ld` est identique à `%d`.

Les types utilisables dans les conversion sont :

Conversion	Signification	Notes
`'d'`	Entier décimal signé.	
`'i'`	Entier décimal signé.	
`'o'`	Valeur octale signée.	(1)
`'u'`	Type obsolète — identique à `'d'`.	(8)
`'x'`	Hexadécimal signé (en minuscules).	(2)
`'X'`	Hexadécimal signé (capitales).	(2)
`'e'`	Format exponentiel pour un *float* (minuscule).	(3)
`'E'`	Format exponentiel pour un *float* (en capitales).	(3)
`'f'`	Format décimal pour un *float*.	(3)
`'F'`	Format décimal pour un *float*.	(3)
`'g'`	Format *float*. Utilise le format exponentiel minuscules si l'exposant est inférieur à `-4` ou pas plus petit que la précision, sinon le format décimal.	(4)
`'G'`	Format *float*. Utilise le format exponentiel en capitales si l'exposant est inférieur à `-4` ou pas plus petit que la précision, sinon le format décimal.	(4)
`'c'`	Octet simple (Accepte un nombre entier ou un seul objet *byte*).	
`'b'`	*Bytes* (tout objet respectant le buffer protocol ou ayant la méthode `__bytes__()`).	(5)
`'s'`	`'s'` est un alias de `'b'` et ne devrait être utilisé que pour du code Python2/3.	(6)
`'a'`	*Bytes* (convertis n'importe quel objet Python en utilisant `repr(obj).encode('ascii', 'backslashreplace)`).	(5)
`'r'`	`'r'` est un alias de `'a'` et ne devrait être utilise que dans du code Python2/3.	(7)
`'%'`	Aucun argument n'est converti, donne un caractère de `'%'` dans le résultat.	

Notes :

1. La forme alternative entraîne l'insertion d'un préfixe octal (`'0o'`) avant le premier chiffre.
2. La forme alternative entraîne l'insertion d'un préfixe `'0x'` ou `'0X'` (respectivement pour les formats `'x'` et `'X'`) avant le premier chiffre.
3. La forme alternative implique la présence d'un point décimal, même si aucun chiffre ne le suit.

   La précision détermine le nombre de chiffres après la virgule, 6 par défaut.
4. La forme alternative implique la présence d'un point décimal et les zéros non significatifs sont conservés (ils ne le seraient pas autrement).

   La précision détermine le nombre de chiffres significatifs avant et après la virgule. 6 par défaut.
5. Si la précision est `N`, la sortie est tronquée à `N` caractères.
6. `b'%s'` est obsolète, mais ne sera pas retiré des version 3.x.
7. `b'%r'` est obsolète mais ne sera pas retiré dans Python 3.x.

8. Voir la **PEP 237**.

Note: La version *bytearray* de cette méthode *ne modifie pas* les octets, elle produit toujours un nouvel objet, même si aucune modification n'a été effectuée.

Voir aussi: **PEP 461** -- Ajout du formatage via % aux *bytes* et *bytesarray*

Nouveau dans la version 3.5.

Vues de mémoires

Les `memoryview` permettent a du code Python d'accéder sans copie aux données internes d'un objet pendant en charge le buffer protocol.

class **`memoryview`**(*obj*)

Crée une `memoryview` faisant référence à *obj*. *obj* doit supporter le *buffer protocol*. Les objets natifs pendant en charge le *buffer protocol* sont `bytes` et `bytearray`.

Une `memoryview` a la notion d'*element*, qui est l'unité de mémoire atomique géré par l'objet *obj* d'origine. Pour de nombreux types simples comme `bytes` et `bytearray`, l'élément est l'octet, mais pour d'autres types tels que `array.array` les éléments peuvent être plus grands.

`len(view)` est égal à la grandeur de `tolist`. Si `view.ndim = 0`, la longueur vaut 1. Si `view.ndim = 1`, la longueur est égale au nombre d'éléments de la vue. Pour les dimensions plus grandes, la longueur est égale à la longueur de la sous-liste représentée par la vue. L'attribut `itemsize` vous donnera la taille en octets d'un élément.

Une `memoryview` autorise le découpage et l'indiçage de ses données. Découper sur une dimension donnera une sous-vue :

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

Si le `format` est un des formats natif du module `struct`, indexer avec un nombre entier ou un *tuple* de nombres entiers est aussi autorisé et renvoie un seul *element* du bon type. Les *memoryview* à une dimension peuvent être indexées avec un nombre entier ou un *tuple* d'un entier. Les *memoryview* multi-dimensionnelles peuvent être indexées avec des *tuples* d'exactement *ndim* entiers où *ndim* est le nombre de dimensions. Les *memoryviews* à zéro dimension peuvent être indexées avec un *tuple* vide.

Voici un exemple avec un autre format que *byte* :

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

Si l'objet sous-jacent est accessible en écriture, la *memoryview* autorisera les assignations de tranches à une dimension. Redimensionner n'est cependant pas autorisé :

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
```

```
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different structures
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

Les *memoryviews* à une dimension de types hachables (lecture seule) avec les formats 'B', 'b', ou 'c' sont aussi hachables. La fonction de hachage est définie tel que `hash(m) == hash(m.tobytes())` :

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

Modifié dans la version 3.3: Les *memoryviews* à une dimension peuvent aussi être découpées. Les *memoryviews* à une dimension avec les formats 'B', 'b', ou 'c' sont maintenant hachables.

Modifié dans la version 3.4: *memoryview* est maintenant enregistrée automatiquement avec `collections.abc.Sequence`

Modifié dans la version 3.5: les *memoryviews* peut maintenant être indexées par un n-uplet d'entiers.

La `memoryview` dispose de plusieurs méthodes :

__eq__(*exporter*)

Une *memoryview* et un *exporter* de la **PEP 3118** sont égaux si leurs formes sont équivalentes et si toutes les valeurs correspondantes sont égales, le format respectifs des opérandes étant interprétés en utilisant la syntaxe de `struct`.

Pour le sous-ensemble des formats de `struct` supportés par `tolist()`, `v` et `w` sont égaux si `v.tolist() ==w.tolist()` :

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

Si l'un des format n'est pas supporté par le module de `struct`, les objets seront toujours considérés différents (même si les formats et les valeurs contenues sont identiques) :

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

Notez que pour les *memoryview*, comme pour les nombres à virgule flottante, `v is w` *n'implique pas* `v == w`.

Modifié dans la version 3.3: Les versions précédentes comparaient la mémoire brute sans tenir compte du format de l'objet ni de sa structure logique.

`tobytes`()

Renvoie les données du *buffer* sous forme de *bytes*. Cela équivaut à appeler le constructeur `bytes` sur le *memoryview*.

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

Pour les listes non contiguës le résultat est égal à la représentation en liste aplatie dont tous les éléments sont convertis en octets. `tobytes()` supporte toutes les chaînes de format, y compris celles qui ne sont pas connues du module `struct`.

`hex`()

Renvoie une chaîne contenant deux chiffres hexadécimaux pour chaque octet de la mémoire.

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

Nouveau dans la version 3.5.

`tolist`()

Renvoie les données de la mémoire sous la forme d'une liste d'éléments.

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

Modifié dans la version 3.3: `tolist()` prend désormais en charge tous les formats d'un caractère du module `struct` ainsi que des représentations multidimensionnelles.

`release`()

Libère le tampon sous-jacent exposé par l'objet *memoryview*. Beaucoup d'objets prennent des initiatives particulières lorsqu'ils sont liés à une vue (par exemple, un `bytearray` refusera temporairement de se faire redimensionner). Par conséquent, appeler *release()* peut être pratique pour lever ces restrictions (et en libérer les ressources liées) aussi tôt que possible.

Après le premier appel de cette méthode, toute nouvelle opération sur la *view* lève une `ValueError` (sauf `release()` elle-même qui peut être appelée plusieurs fois) :

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

Le protocole de gestion de contexte peut être utilisé pour obtenir un effet similaire, via l'instruction `with` :

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

Nouveau dans la version 3.2.

cast(*format*[, *shape*])

Change le format ou la forme d'une *memoryview*. Par défaut *shape* vaut `[byte_length//new_itemsize]`, ce qui signifie que la vue résultante n'aura qu'une dimension. La valeur renvoyée est une nouvelle *memoryview*, mais la mémoire elle-même n'est pas copiée. Les changements supportés sont une dimension vers C-contiguous et *C-contiguous* vers une dimension.

Le format de destination est limité à un seul élément natif de la syntaxe du module `struct`. L'un des formats doit être un *byte* ('B', 'b', ou 'c'). La longueur du résultat en octets doit être la même que la longueur initiale.

Transforme *1D/long* en *1D/unsigned bytes* :

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

Transforme *1D/unsigned bytes* en *1D/char* :

```
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview: invalid value for format "B"
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

Transforme *1D/bytes* en *3D/ints* en *1D/signed char* :

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

Cast 1D/unsigned long to 2D/unsigned long:

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

Nouveau dans la version 3.3.

Modifié dans la version 3.5: Le format de la source n'est plus restreint lors de la transformation vers une vue d'octets.

Plusieurs attributs en lecture seule sont également disponibles :

`obj`

L'objet sous-jacent de la *memoryview* :

```
>>> b  = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

Nouveau dans la version 3.3.

`nbytes`

`nbytes == product(shape) * itemsize == len(m.tobytes())`. Ceci est l'espace que la liste utiliserait en octets, dans une représentation contiguë. Ce n'est pas nécessairement égale à `len(m)` :

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

Tableaux multidimensionnels :

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96
```

Nouveau dans la version 3.3.

`readonly`

Un booléen indiquant si la mémoire est en lecture seule.

`format`

Une chaîne contenant le format (dans le style de `struct`) pour chaque élément de la vue. Une *memoryview* peut être crée depuis des exportateurs de formats arbitraires, mais certaines méthodes (comme `tolist()`) sont limitées aux formats natifs à un seul élément.

Modifié dans la version 3.3: le format `'B'` est maintenant traité selon la syntaxe du module *struct*. Cela signifie que `memoryview(b'abc')[0] == b'abc'[0] == 97`.

itemsize

La taille en octets de chaque élément d'une *memoryview* :

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim

Un nombre entier indiquant le nombre de dimensions d'un tableau multi-dimensionnel représenté par la *memoryview*.

shape

Un *tuple* d'entiers de longueur `ndim` donnant la forme de la *memoryview* sous forme d'un tableau à N dimensions.

Modifié dans la version 3.3: Un *tuple* vide au lieu de `None` lorsque *ndim = 0*.

strides

Un *tuple* d'entiers de longueur `ndim` donnant la taille en octets permettant d'accéder à chaque dimensions du tableau.

Modifié dans la version 3.3: Un *tuple* vide au lieu de `None` lorsque *ndim = 0*.

suboffsets

Détail de l'implémentation des *PIL-style arrays*. La valeur n'est donné qu'a titre d'information.

c_contiguous

Un booléen indiquant si la mémoire est C-contiguous.

Nouveau dans la version 3.3.

f_contiguous

Un booléen indiquant si la mémoire est Fortran contiguous.

Nouveau dans la version 3.3.

contiguous

Un booléen indiquant si la mémoire est contiguous.

Nouveau dans la version 3.3.

Types d'ensembles — `set`, `frozenset`

Un objet *set* est une collection non-triée d'objets hashable distincts. Les utilisations classiques sont le test d'appartenance, la déduplication d'une séquence, ou le calcul d'opérations mathématiques telles que l'intersection, l'union, la différence, ou la différence symétrique. (Pour les autres conteneurs, voir les classes natives `dict`, `list`, et `tuple`, ainsi que le module `collections`.)

Comme pour les autres collections, les ensembles supportent `x in set`, `len(set)`, et `for x in set`. En tant que collection non-triée, les ensembles n'enregistrent pas la position des éléments ou leur ordre d'insertion. En conséquence, les *sets* n'autorisent ni l'indexation, ni le découpage, ou tout autre comportement de séquence.

Il existe actuellement deux types natifs pour les ensembles, `set` et `fronzenset`. Le type `set` est muable --- son contenu peut changer en utilisant des méthodes comme `add()` et `remove()`. Puisqu'il est muable, il n'a pas de valeur de hachage et ne peut donc pas être utilisé ni comme clef de dictionnaire ni comme élément d'un autre ensemble. Le type `frozenset` est immuable et hashable --- son contenu ne peut être modifié après sa création, il peut ainsi être utilisé comme clef de dictionnaire ou élément d'un autre *set*.

Des *sets* (mais pas des *frozensets*) peuvent être crées par une liste d'éléments séparés par des virgules et entre accolades, par exemple : `{'jack', 'sjoerd'}`, en plus du constructeur de la classe `set`.

Les constructeurs des deux classes fonctionnent pareil :

class **set**([*iterable*])
class **frozenset**([*iterable*])

Renvoie un nouveau *set* ou *frozenset* dont les éléments viennent d'*iterable*. Les éléments d'un *set* doivent être hashable. Pour représenter des *sets* de *sets* les *sets* intérieurs doivent être des `frozenset`. Si *iterable* n'est pas spécifié, un nouveau *set* vide est renvoyé.

Les instances de `set` et `frozenset` fournissent les opérations suivantes :

len(s)
Donne le nombre d'éléments dans le *set s* (cardinalité de *s*).

x in s
Test d'appartenance de *x* dans *s*.

x not in s
Test de non-appartenance de *x* dans *s*.

isdisjoint(*other*)
Renvoie `True` si l'ensemble n'a aucun élément en commun avec *other*. Les ensembles sont disjoints si et seulement si leurs intersection est un ensemble vide.

issubset(*other*)
set <= other
Teste si tous les éléments du set sont dans *other*.

set < other
Teste si l'ensemble est un sous-ensemble de *other*, c'est-à-dire, `set <= other and set != other`.

issuperset(*other*)
set >= other
Teste si tous les éléments de *other* sont dans l'ensemble.

set > other
Teste si l'ensemble est un sur-ensemble de *other*, c'est-à-dire, `set >= other and set != other`.

union(**others*)
set | other | ...
Renvoie un nouvel ensemble dont les éléments viennent de l'ensemble et de tous les autres.

intersection(**others*)
set & other & ...
Renvoie un nouvel ensemble dont les éléments sont commun à l'ensemble et à tous les autres.

difference(**others*)
set - other - ...
Renvoie un nouvel ensemble dont les éléments sont dans l'ensemble mais ne sont dans aucun des autres.

symmetric_difference(*other*)
set ^ other
Renvoie un nouvel ensemble dont les éléments sont soit dans l'ensemble, soit dans les autres, mais pas dans les deux.

copy()
Renvoie une copie de surface du dictionnaire.

Remarque : Les méthodes `union()`, `intersection()`, `difference()`, et `symmetric_difference()`, `issubset()`, et `issuperset()` acceptent n'importe quel itérable comme argument, contrairement aux opérateurs équivalents qui n'acceptent que des *sets*. Il est donc préférable d'éviter les constructions comme `set('abc') & 'cbs'`, sources typiques d'erreurs, en faveur d'une construction plus lisible : `set('abc').intersection('cbs')`.

Les classes `set` et `frozenset` supportent les comparaisons d'ensemble à ensemble. Deux ensembles sont égaux si et seulement si chaque éléments de chaque ensemble est contenu dans l'autre (autrement dit que chaque ensemble est un sous-ensemble de l'autre). Un ensemble est plus petit qu'un autre ensemble si et seulement si le premier est un sous-ensemble du second (un sous-ensemble, mais pas égal). Un ensemble est plus grand qu'un autre ensemble si et seulement si le premier est un sur-ensemble du second

(est un sur-ensemble mais n'est pas égal).

Les instances de `set` se comparent aux instances de `frozenset` en fonction de leurs membres. Par exemple, `set('abc') == frozenset('abc')` envoie `True`, ainsi que `set('abc') in set([frozenset('abc')])`.

Les comparaisons de sous-ensemble et d'égalité ne se généralisent pas en une fonction donnant un ordre total. Par exemple, deux ensemble disjoints non vides ne sont ni égaux et ni des sous-ensembles l'un de l'autre, donc toutes ces comparaisons donnent `False` : `a<b`, `a==b`, et `a>b`.

Puisque les *sets* ne définissent qu'un ordre partiel (par leurs relations de sous-ensembles), la sortie de la méthode `list.sort()` n'est pas définie pour des listes d'ensembles.

Les éléments des *sets*, comme les clefs de dictionnaires, doivent être hashable.

Les opérations binaires mélangeant des instances de `set` et `frozenset` renvoient le type de la première opérande. Par exemple : `frozenset('ab') | set('bc')` renvoie une instance de `frozenset`.

La table suivante liste les opérations disponibles pour les `set` mais qui ne s'appliquent pas aux instances de `frozenset` :

`update`(**others*)
`set |= other | ...`
Met à jour l'ensemble, ajoutant les éléments de tous les autres.

`intersection_update`(**others*)
`set &= other & ...`
Met à jour l'ensemble, ne gardant que les éléments trouvés dans tous les autres.

`difference_update`(**others*)
`set -= other | ...`
Met à jour l'ensemble, retirant les éléments trouvés dans les autres.

`symmetric_difference_update`(*other*)
`set ^= other`
Met à jour le set, ne gardant que les éléments trouvés dans un des ensembles mais pas dans les deux.

`add`(*elem*)
Ajoute l'élément *elem* au set.

`remove`(*elem*)
Retire l'élément *elem* de l'ensemble. Lève une exception `KeyError` si *elem* n'est pas dans l'ensemble.

`discard`(*elem*)
Retire l'élément *elem* de l'ensemble s'il y est.

`pop`()
Retire et renvoie un élément arbitraire de l'ensemble. Lève une exception `KeyError` si l'ensemble est vide.

`clear`()
Supprime tous les éléments du *set*.

Notez que les versions non-opérateurs des méthodes `update()`, `intersection_update()`, `difference_update()`, et `symmetric_difference_update()` acceptent n'importe quel itérable comme argument.

Notez que l'argument *elem* des méthodes `__contains__()`, `remove()`, et `discard()` peut être un ensemble. Pour supporter la recherche d'un *frozenset* équivalent, un *frozenset* temporaire est crée depuis *elem*.

Les types de correspondances — `dict`

Un objet mapping fait correspondre des valeurs hashable à des objets arbitraires. Les *mappings* sont des objets muables. Il n'existe pour le moment qu'un type de *mapping* standard, le *dictionary*. (Pour les autres conteneurs, voir les types natifs `list`, `set`, et `tuple`, ainsi que le module `collections`.)

Les clefs d'un dictionnaire sont *presque* des données arbitraires. Les valeurs qui ne sont pas hashable, c'est-à-dire qui contiennent les listes, des dictionnaires ou autre type muable (qui sont comparés par valeur plutôt que par leur identité) ne peuvent pas être utilisées comme clef de dictionnaire. Les types numériques utilisés comme clef obéissent aux règles classiques en ce qui concerne les comparaisons : si deux nombres sont égaux (comme `1` et `1.0`) ils peuvent tous les deux être utilisés pour obtenir la même entrée d'un dictionnaire. (Notez cependant que puisque les ordinateurs stockent les nombres à virgule flottante sous forme d'approximations, il est généralement imprudent de les utiliser comme clefs de dictionnaires.)

Il est possible de créer des dictionnaires en plaçant entre accolades une liste de paires de `key: value` séparés par des virgules, par exemple: `{'jack': 4098, 'sjoerd': 4127}` ou `{4098: 'jack', 4127: 'sjoerd'}`, ou en utilisant le constructeur de `dict`.

class **dict**(***kwarg*)
class **dict**(*mapping*, ***kwarg*)
class **dict**(*iterable*, ***kwarg*)

Renvoie un nouveau dictionnaire initialisé depuis un argument positionnel optionnel, et un ensemble (vide ou non) d'arguments par mot clef.

Si aucun argument positionnel n'est donné, un dictionnaire vide est crée. Si un argument positionnel est donné et est un *mapping object*, un dictionnaire est crée avec les mêmes paires de clef-valeurs que le *mapping* donné. Autrement, l'argument positionnel doit être un objet iterable. Chaque élément de cet itérable doit lui même être un itérable contenant exactement deux objets. Le premier objet de chaque élément devient la une clef du nouveau dictionnaire, et le second devient sa valeur correspondante. Si une clef apparaît plus d'une fois, la dernière valeur pour cette clef devient la valeur correspondante à cette clef dans le nouveau dictionnaire.

Si des arguments nommés sont donnés, ils sont ajoutés au dictionnaire créé depuis l'argument positionnel. Si une clef est déjà présente, la valeur de l'argument nommé remplace la valeur reçue par l'argument positionnel.

Typiquement, les exemples suivants renvoient tous un dictionnaire valant `{"one": 1, "two": 2, "three": 3}` :

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Fournir les arguments nommés comme dans le premier exemple en fonctionne que pour des clefs qui sont des identifiants valide en Python. Dans les autres cas, toutes les clefs valides sont utilisables.

Voici les opérations gérées par les dictionnaires, (par conséquent, d'autres types de *mapping* peuvent les gérer aussi) :

len(d)

Renvoie le nombre d'éléments dans le dictionnaire *d*.

d[key]

Donne l'élément de *d* dont la clef est *key*. Lève une exception `KeyError` si *key* n'est pas dans le dictionnaire.

Si une sous-classe de *dict* définit une méthode `__missing__()` et que *key* manque, l'opération `d[key]` appelle cette méthode avec la clef *key* en argument. L'opération `d[key]` renverra la valeur, ou lèvera l'exception renvoyée ou levée par l'appel à `__missing__(key)`. Aucune autre opération ni méthode n'appellent `__missing__()`. If `__missing__()` n'est pas définie, une exception `KeyError` est levée. `__missing__()` doit être une méthode; ça ne peut être une variable d'instance :

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
```

```
1
```

L'exemple ci-dessus montre une partie de l'implémentation de `collections.Counter`. `collections.defaultdict` implémente aussi `__missing__`.

`d[key] = value`

Assigne `d[key]` à *value*.

`del d[key]`

Supprime `d[key]` de *d*. Lève une exception `KeyError` si *key* n'est pas dans le dictionnaire.

`key in d`

Renvoie `True` si *d* a la clef *key*, sinon `False`.

`key not in d`

Équivalent à `not key in d`.

`iter(d)`

Renvoie un itérateur sur les clefs du dictionnaire. C'est un raccourci pour `iter(d.keys())`.

`clear`()

Supprime tous les éléments du dictionnaire.

`copy`()

Renvoie une copie de surface du dictionnaire.

classmethod `fromkeys`(*iterable*[, *value*])

Crée un nouveau dictionnaire avec les clefs de *iterable* et les valeurs à *value*.

`fromkeys()` est une *class method* qui renvoie un nouveau dictionnaire. *value* vaut `None` par défaut.

`get`(*key*[, *default*])

Renvoie la valeur de *key* si *key* est dans le dictionnaire, sinon *default*. Si *default* n'est pas donné, il vaut `None` par défaut, de manière à ce que cette méthode ne lève jamais `KeyError`.

`items`()

Renvoie une nouvelle vue des éléments du dictionnaire (paires de `(key, value)`). Voir la documentation des vues.

`keys`()

Renvoie une nouvelle vue des clefs du dictionnaire. Voir la documentation des vues.

`pop`(*key*[, *default*])

Si *key* est dans le dictionnaire elle est supprimée et sa valeur est renvoyée, sinon renvoie *default*. Si *default* n'est pas donné et que *key* n'est pas dans le dictionnaire, une `KeyError` est levée.

`popitem`()

Supprime et renvoie une paire `(key, value)` du dictionnaire. Les paires sont renvoyées dans un ordre LIFO.

`popitem()` est pratique pour itérer un dictionnaire de manière destructive, comme souvent dans les algorithmes sur les ensembles. Si le dictionnaire est vide, appeler `popitem()` lève une `KeyError`.

Modifié dans la version 3.7: L'ordre "dernier entré, premier sorti" (LIFO) est désormais assuré. Dans les versions précédentes, `popitem()` renvoyait une paire clé/valeur arbitraire.

`setdefault`(*key*[, *default*])

Si *key* est dans le dictionnaire, sa valeur est renvoyée. Sinon, insère *key* avec comme valeur *default* et renvoie *default*. *default* vaut `None` par défaut.

`update`([*other*])

Met à jour le dictionnaire avec les paires de clef/valeur d'*other*, écrasant les clefs existantes. Renvoie `None`.

`update()` accepte aussi bien un autre dictionnaire qu'un itérable de clef/valeurs (sous forme de *tuples* ou autre itérables de longueur deux). Si des paramètres par mot-clef sont donnés, le dictionnaire et ensuite mis à jour avec ces pairs de clef/valeurs : `d.update(red=1, blue=2)`.

`values`()

Renvoie une nouvelle vue des valeurs du dictionnaire. Voir la documentation des vues.

Deux dictionnaires sont égaux si et seulement si ils ont les mêmes paires de clef-valeur. Les comparaisons d'ordre (<, <=, >=, >) lèvent une `TypeError`.

Les dictionnaires préservent l'ordre des insertions. Notez que modifier une clé n'affecte pas l'ordre. Les clés ajoutées après un effacement sont insérées à la fin.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

Modifié dans la version 3.7: L'ordre d'un dictionnaire est toujours l'ordre des insertions. Ce comportement était un détail d'implémentation de CPython depuis la version 3.6.

Voir aussi: `types.MappingProxyType` peut être utilisé pour créer une vue en lecture seule d'un `dict`.

Les vues de dictionnaires

Les objets renvoyés par `dict.keys()`, `dict.values()` et `dict.items()` sont des *vues*. Ils fournissent une vue dynamique des éléments du dictionnaire, ce qui signifie que si le dictionnaire change, la vue reflète ces changements.

Les vues de dictionnaires peuvent être itérées et ainsi renvoyer les données du dictionnaire, elle gèrent aussi les tests de présence :

`len(dictview)`

Renvoie le nombre d'entrées du dictionnaire.

`iter(dictview)`

Renvoie un itérateur sur les clefs, les valeurs, ou les éléments (représentés par des *tuples* de `(key, value)`) du dictionnaire.

Les clefs et les valeurs sont itérées dans l'ordre de leur insertion. Ceci permet la création de paires de `(key, value)` en utilisant `zip()` : `pairs = zip(d.values(), d.keys())`. Un autre moyen de construire la même liste est `pairs = [(v, k) for (k, v) in d.items()]`.

Parcourir des vues tout en ajoutant ou supprimant des entrées dans un dictionnaire peut lever une `RuntimeError` ou ne pas fournir toutes les entrées.

Modifié dans la version 3.7: L'ordre d'un dictionnaire est toujours l'ordre des insertions.

`x in dictview`

Renvoie `True` si *x* est dans les clefs, les valeurs, ou les éléments du dictionnaire sous-jacent (dans le dernier cas, *x* doit être un *tuple* `(key, value)`).

Les vues de clefs sont semblables à des ensembles puisque leurs entrées sont uniques et hachables. Si toutes les valeurs sont hachables, et qu'ainsi toutes les paires de `(key, value)` sont uniques et hachables, alors la vue donnée par *items()* est aussi semblable à un ensemble. (Les vues données par *items()* ne sont généralement pas traitées comme des ensembles, car leurs valeurs ne sont généralement pas uniques.) Pour les vues semblables aux ensembles, toutes les opérations définies dans la classe de base abstraite `collections.abc.Set` sont disponibles (comme ==, <, ou ^).

Exemple d'utilisation de vue de dictionnaire :

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
```

```
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order (insertion order)
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```

Le type gestionnaire de contexte

L'instruction `with` permet l'existence de contextes définis à l'exécution par des gestionnaires de contextes. C'est implémenté via une paire de méthodes permettant de définir un contexte, à l'exécution, qui est entré avant l'exécution du corps de l'instruction, et qui est quitté lorsque l'instruction se termine :

`contextmanager.__enter__()`

Entre dans le contexte à l'exécution, soit se renvoyant lui-même, soit en renvoyant un autre objet en lien avec ce contexte. La valeur renvoyée par cette méthode est liée à l'identifiant donné au `as` de l'instruction `with` utilisant ce gestionnaire de contexte.

Un exemple de gestionnaire de contexte se renvoyant lui-même est file object. Les *file objects* se renvoient eux-même depuis __enter__() et autorisent `open()` à être utilisé comme contexte à une instruction `with`.

Un exemple de gestionnaire de contexte renvoyant un objet connexe est celui renvoyé par `decimal.localcontext()`. Ces gestionnaires remplacent le contexte décimal courant par une copie de l'original, copie qui est renvoyée. Ça permet de changer le contexte courant dans le corps du `with` sans affecter le code en dehors de l'instruction `with`.

`contextmanager.__exit__`(*exc_type*, *exc_val*, *exc_tb*)

Sort du contexte et renvoie un booléen indiquant si une exception survenue doit être supprimée. Si une exception est survenue lors de l'exécution du corps de l'instruction `with`, les arguments contiennent le type de l'exception, sa valeur, et la trace de la pile (*traceback*). Sinon les trois arguments valent `None`.

L'instruction `with` inhibera l'exception si cette méthode renvoie une valeur vraie, l'exécution continuera ainsi à l'instruction suivant immédiatement l'instruction `with`. Sinon, l'exception continuera de se propager après la fin de cette méthode. Les exceptions se produisant pendant l'exécution de cette méthode remplaceront toute exception qui s'est produite dans le corps du `with`.

L'exception reçue ne doit jamais être relancée explicitement, cette méthode devrait plutôt renvoyer une valeur fausse pour indiquer que son exécution s'est terminée avec succès et qu'elle ne veut pas supprimer l'exception. Ceci permet au code de gestion du contexte de comprendre si une méthode `__exit__()` a échoué.

Python définit plusieurs gestionnaires de contexte pour faciliter la synchronisation des fils d'exécution, la fermeture des fichiers ou d'autres objets, et la configuration du contexte arithmétique décimal. Ces types spécifiques ne sont pas traités différemment, ils respectent simplement le protocole de gestion du contexte. Voir les exemples dans la documentation du module `contextlib`.

Les generators de Python et le décorateur `contextlib.contextmanager` permettent d'implémenter simplement ces protocoles. Si un générateur est décoré avec `contextlib.contextmanager`, elle renverra un gestionnaire de contexte implémentant les méthodes `__enter__()` et `__exit__()`, plutôt que l'itérateur produit

par un générateur non décoré.

Notez qu'il n'y a pas d'emplacement spécifique pour ces méthodes dans la structure de type pour les objets Python dans l'API Python/C. Les types souhaitant définir ces méthodes doivent les fournir comme une méthode accessible en Python. Comparé au coût de la mise en place du contexte d'exécution, les le coût d'un accès au dictionnaire d'une classe unique est négligeable.

Autres types natifs

L'interpréteur gère aussi d'autres types d'objets, la plupart ne supportant cependant qu'une ou deux opérations.

Modules

La seule opération spéciale sur un module est l'accès à ses attributs : `m.name`, où *m* est un module et *name* donne accès un nom défini dans la table des symboles de *m*. Il est possible d'assigner un attribut de module. (Notez que l'instruction `import` n'est pas strictement une opération sur un objet module. `import foo` ne nécessite pas qu'un objet module nommé *foo* existe, il nécessite cependant une *définition* (externe) d'un module nommé *foo* quelque part.)

Un attribut spécial à chaque module est `__dict__`. C'est le dictionnaire contenant la table des symbole du module. Modifier ce dictionnaire changera la table des symboles du module, mais assigner directement `__dict__` n'est pas possible (vous pouvez écrire `m.__dict__['a'] = 1`, qui donne `1` comme valeur pour `m.a`, mais vous ne pouvez pas écrire `m.__dict__ = {}`). Modifier `__dict__` directement n'est pas recommandé.

Les modules natifs à l'interpréteur sont représentés `<module 'sys' (built-in)>`. S'ils sont chargés depuis un fichier, ils sont représentés `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`.

Les classes et instances de classes

Voir Objets, valeurs et types et Définition de classes.

Fonctions

Les objets fonctions sont crées par les définitions de fonctions. La seule opération applicable à un objet fonction est de l'appeler : `func(argument-list)`.

Il existe en fait deux catégories d'objets fonctions : Les fonctions natives et les fonctions définies par l'utilisateur. Les deux gèrent les mêmes opérations (l'appel à la fonction), mais leur implémentation est différente, d'où les deux types distincts.

Voir Définition de fonctions pour plus d'information.

Méthodes

Les méthodes sont des fonctions appelées via la notation d'attribut. Il en existe deux variantes : Les méthodes natives (tel que `append()` sur les listes), et les méthodes d'instances de classes. Les méthodes natives sont représentées avec le type qui les supporte.

Si vous accédez à une méthode (une fonction définie dans l'espace de nommage d'une classe) via une instance, vous obtenez un objet spécial, une *bound method* (aussi appelée *instance method*). Lorsqu'elle est appelée, elle ajoute l'argument `self` à la liste des arguments. Les méthodes liées ont deux attributs spéciaux, en lecture seule : `m.__self__` est l'objet sur lequel la méthode travaille, et `m.__func__` est la fonction implémentant la méthode. Appeler `m(arg-1, arg-2, …, arg-n)` est tout à fait équivalent à appeler `m.__func__(m.__self__, arg-1, arg-2, …, arg-n)`.

Comme les objets fonctions, les objets méthodes, liées, acceptent des attributs arbitraires. Cependant, puisque les attributs de méthodes doivent être stockés dans la fonction sous-jacente (`meth.__func__`), affecter des attributs à des objets *bound method* est interdit. Toute tentative d'affecter un attribut sur un objet *bound method* lèvera une `AttributeError`. Pour affecter l'attribut, vous devrez explicitement l'affecter à sa fonction sous-jacente :

```
>>> class C:
...     def method(self):
...         pass
...
```

```
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

Voir Hiérarchie des types standards pour plus d'information.

Objets code

Les objets code sont utilisés par l'implémentation pour représenter du code Python "pseudo-compilé", comme un corps de fonction. Ils sont différents des objets fonction dans le sens où ils ne contiennent pas de référence à leur environnement global d'exécution. Les objets code sont renvoyés par la fonction native `compile()` et peuvent être obtenus des objets fonction via leur attribut `__code__`. Voir aussi le module `code`.

Les objets code peuvent être exécutés ou évalués en les passant (au lieu d'une chaîne contenant du code) aux fonction natives `exec()` ou `eval()`.

Voir Hiérarchie des types standards pour plus d'information.

Objets type

Les objets types représentent les différents types d'objets. Le type d'un objet est obtenu via la fonction native `type()`. Il n'existe aucune opération spéciale sur les types. Le module standard `types` définit les noms de tous les types natifs.

Les types sont représentés : `<class 'int'>`.

L'objet Null

Cet objet est renvoyé par les fonctions ne renvoyant pas explicitement une valeur. Il ne supporte aucune opération spéciale. Il existe exactement un objet *null* nommé `None` (c'est un nom natif). `type(None)()`.

C'est écrit `None`.

L'objet points de suspension

Cet objet est utilisé classiquement lors des découpes (voir Tranches). Il ne supporte aucune opération spéciale. Il n'y a qu'un seul objet *ellipsis*, nommé `Ellipsis` (un nom natif). `type(Ellipsis)()` produit le *singleton* `Ellipsis`.

C'est écrit `Ellipsis` ou `...`.

L'objet *NotImplemented*

Cet objet est renvoyé depuis des comparaisons ou des opérations binaires effectuées sur des types qu'elles ne supportent pas. Voir Comparaisons pour plus d'informations. Il n'y a qu'un seul objet `NotImplemented`. `type(NotImplemented)()` renvoie un *singleton*.

C'est écrit `NotImplemented`.

Valeurs booléennes

Les valeurs booléennes sont les deux objets constants `False` et `True`. Ils sont utilisés pour représenter les valeurs de vérité (bien que d'autres valeurs peuvent être considérées vraies ou fausses). Dans des contextes numériques (par exemple en argument d'un opérateur arithmétique), ils se comportent comme les nombres entiers 0 et 1, respectivement. La fonction native `bool()` peut être utilisée pour convertir n'importe quelle valeur en booléen tant que la valeur peut être interprétée en une valeur de vérité (voir Valeurs booléennes au dessus).

Ils s'écrivent `False` et `True`, respectivement.

Objets internes

Voir Hiérarchie des types standards. Ils décrivent les objets *stack frame*, *traceback*, et *slice*.

Attributs spéciaux

L'implémentation ajoute quelques attributs spéciaux et en lecture seule, à certains types, lorsque ça a du sens. Certains ne sont *pas* listés par la fonction native `dir()`.

`object.__dict__`
: Un dictionnaire ou un autre *mapping object* utilisé pour stocker les attributs (modifiables) de l'objet.

`instance.__class__`
: La classe de l'instance de classe.

`class.__bases__`
: Le *tuple* des classes parentes d'un objet classe.

`definition.__name__`
: Le nom de la classe, fonction, méthode, descripteur, ou générateur.

`definition.__qualname__`
: Le qualified name de la classe, fonction, méthode, descripteur, ou générateur.

Nouveau dans la version 3.3.

`class.__mro__`
: Cet attribut est un *tuple* contenant les classes parents prises en compte lors de la résolution de méthode.

`class.mro()`
: Cette méthode peut être surchargée par une méta-classe pour personnaliser l'ordre de la recherche de méthode pour ses instances. Elle est appelée à la l'initialisation de la classe, et son résultat est stocké dans l'attribut `__mro__`.

`class.__subclasses__()`
: Chaque classe garde une liste de références faibles à ses classes filles immédiates. Cette méthode renvoie la liste de toutes ces références encore valables. Exemple :

```
>>> int.__subclasses__()
[<class 'bool'>]
```

Notes

[1] Plus d'informations sur ces méthodes spéciales peuvent être trouvées dans le *Python Reference Manual* (Personnalisation de base).

[2] Par conséquent, la liste `[1, 2]` est considérée égale à `[1.0, 2.0]`. Idem avec des tuples.

[3] Nécessairement, puisque l'analyseur ne peut pas discerner le type des opérandes.

4(1,2,3,4)Les caractères capitalisables sont ceux dont la propriété Unicode *general category* est soit "Lu" (pour *Letter*, *uppercase*), soit "Ll" (pour *Letter*, *lowercase*), soit "Lt" (pour *Letter*, *titlecase*).

5(1,2)Pour insérer un *tuple*, vous devez donc donner un *tuple* d'un seul élément, contenant le *tuple* à insérer.