Pascal ORTIZ

Les chaînes formatées

Version du 3 juin 2021

Table des matières

Chapitre I

Les f-chaînes

1 L'essentiel

La problématique des chaînes formatées

Le formatage de chaînes est une possibilité offerte par de nombreux langages de programmation de créer une chaîne de caractères (comprendre du texte) à partir d'un modèle prédéfini. Typiquement, on peut utiliser le formatage de chaîne pour générer une date en français puisqu'une telle date a la structure `jj/mm/aaaa` connue à l'avance ou encore l'heure qui a le format `hh:mm:ss`. Mais, au-delà de ces formats figés, de nombreuses compositions structurées comme un sudoku à remplir, une table de conjugaison ou un calendrier :

```
    Janvier 2038
di lu ma me je ve sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

sont avantageusement construites en utilisant des chaînes formatées.

La notion de chaîne formatée

Supposons qu'on veuille générer de façon automatisée des phrases ayant la structure suivante :

```
Son nom est titi et il a 8 ans.
Son nom est Arthur et il a 30 ans.
Son nom est Alfred et il a 100 ans.
```

D'une phrase à l'autre, seuls le nom et le nombre représentant l'âge changent. Les chaînes ci-dessus sont dites des *chaînes formatées* : toutes les chaînes ont même structure, ici de la forme

`Son nom est XXX et il a YYY ans.`

Seuls varient le nom `XXX` et l'âge `YYY`.

Dans le présent document j'appelerai la chaîne-modèle telle que

```
Son nom est XXX et il a YYY ans.
```

un *gabarit* ou encore un *modèle* ou parfois une *chaîne de formatage*. Les parties à remplacer sont appelées *champs de remplacement* (en anglais *remplacement field*). Un gabarit est le modèle à partir duquel d'autres chaînes vont être produites par substitution.

Les f-chaînes : exemple typique

La version 3.6 de Python a introduit une simplification syntaxique très intéressante pour le formatage de chaînes : les *chaînes littérales formatées* ou encore les *f-string* (en français, les f-chaînes, terme que j'emploierai). Un simple exemple montre bien la lisibilité de cette syntaxe :

```
1 nom = "Arthur"
2 age = 30
3 s = f"Son nom est {nom} et il a {age + 10} ans"
4 print(s)
```

```
5 Son nom est Arthur et il a 40 ans
```

Elles ont l'apparence de chaînes littérales habituelles, cf. les quotes ligne 3, mais elles sont précédées d'une lettre `f` (pour « formaté »). Dans la chaîne littérale, entre accolades, on place une *expression* Python (et pas seulement une variable) et le remplacement se fait après évaluation de l'expression. Ci-dessus, noter que l'expression `age + 10` a été évaluée.

Pour des situations simples et dans de nombreuses situations courantes, les f-chaînes fournissent la méthode la plus souple et la plus lisible pour effectuer un formatage de chaînes. En interne, elles utilisent la méthode `format` dont elles reprennent largement la syntaxe. La méthode `format` est plus générale mais, peut-être, moins lisible.

Documentation

— Lien vers la documentation officielle : Formatted string literals

— Discussions sur stackoverflow : python, fstring

Spécification de formatage dans une f-chaîne

Jusqu'à présent un champ de remplacement (ie une paire d'accolades) dans une f-chaîne, contient uniquement une expression :

```
1 fruits = ["pomme", "poire", "prune"]
2 s = f"une {fruits[1]}, une {fruits[2]} et une {fruits[1]}"
3
4 print(s)
```

```
5 une poire, une prune et une poire
```

— Ligne 2 : chaque champ de remplacement contient juste une expression.

Mais, cette syntaxe peut être enrichie pour préciser dans chaque champ de remplacement une *spécification de formatage*, par exemple ici comment on formate une date :

```
1 jsemaine = "dimanche"
2 mois = 9
3 jmois = 1
4 an =2030
5
6 s = f"{jsemaine} {jmois:02}/{mois:02}/{an}"
7 print(s)
```

```
8 dimanche 01/09/2030
```

— Ligne 6 : on compte 4 champs de remplacement (les 4 paires d'accolades) ; le 2e et le 3e contiennent une *spécification de formatage*. Par exemple, dans le champ `{jmois:02}`, la partie `02` à droite du symbole `:` est une spécification de formatage.

À droite du nom du champ, il est possible de placer un « spécification de formatage » qui précise **comment** la valeur à substituer doit être formatée. Pour signifier une spécification de formatage, après le nom du champ, on place le symbole `:` (*deux-points*) ; la spécification de formatage suit le symbole `:` jusqu'à l'accolade fermante.

En pratique, une spécification de formatage sert surtout à formater des **nombres** suivant leurs types (entiers, décimaux, etc) et à **gérer l'espacement** autour du champ de remplacement.

Reprenons l'exemple précédent :

```
1 jsemaine = "dimanche"
2 mois = 9
3 jmois = 1
4 an =2030
5
6 s = f"{jsemaine} {jmois:02}/{mois:02}/{an}"
7 print(s)
```

```
8 dimanche 01/09/2030
```

— Ligne 1 : par exemple, le champ `{jmois:02}` signifie (ce sera expliqué en détail ultérieurement) que la valeur de `jmois` doit être un nombre et qu'il sera formaté en plaçant un **zéro** devant le nombre si le nombre n'est constitué que d'un seul chiffre.

— Ligne 8 : `jmois` qui vaut `9` comme on le voit ligne 2, est donc formaté sous la forme `09`.

Spécificateurs de formatage

Précisons maintenant ce qu'on entend par « spécificateurs de formatage » :

```
1 jsemaine = "dimanche"
2 mois = 9
3 jmois = 1
4 an =2030
5
6 s = f"{jsemaine} {jmois:02}/{mois:02}/{an}"
7 print(s)
```

```
8 dimanche 01/09/2030
```

— Ligne 6 : `02` est une **spécification de formatage**. En réalité, cette spécification se décompose en deux « spécificateurs » :

— le spécificateur `0` pour spécifier que le nombre est, si nécessaire, précédé de zéros ;
— le spécificateur `2` pour spécifier que deux chiffres seront représentés, cf. le `09` de l'affichage.

Plus généralement, chaque spécification est constituée d'une succession de spécificateurs, jusqu'à 8 spécificateurs. On prendra soin de ne pas placer par mégarde d'espaces dans la spécification, après les deux points sinon l'espace sera visible, voire aura un effet inattendu :

```
1 jsemaine = "dimanche"
2 mois = 9
3 jmois = 1
4 an =2030
5
6 s = f"{jsemaine} {jmois:02}/{mois: 02}/{an}"
7 print(s)
```

```
8 dimanche 01/ 9/2030
```

— Ligne 6 : on a rajouté un espace dans l'avant-dernière accolade, juste avant le `02`
— Ligne 8 : le zéro a disparu avant le `2` et a été remplacé par un un espace.

Par ailleurs, l'ordre des spécificateurs n'est pas sans importance. Au lieu de *spécificateur*, le terme *d'option de formatage* est aussi employé, et parfois aussi de *spécifieur* (anglicisme).
Les règles s'appliquant à une spécification de formatage utilisée par les f-chaines sont les mêmes que celles utilisées pour les gabarits de la méthode `format`.

Partie décimale d'un flottant dans une f-chaîne

Le formatage des valeurs des flottants en Python peut faire apparaître jusqu'à 6 décimales. Le programmeur souhaite parfois n'en faire apparaître qu'un nombre plus réduit (ou plus important). Une spécificateur dit de *précision* permet de contrôler le nombre de chiffres décimaux nécessaires pour représenter un nombre flottant donné :

```
1 x=22/7
2 print(x)
3
4 print(f"{x:.2f}")
5 print(f"{x:.50f}")
```

```
6 3.142857142857143
7 3.14
8 3.14285714285714279370154144999105483293533325195312
```

— Ligne 4 : la spécification est ici `.2f` et contient deux spécificateurs :
  — point suivi de la valeur de la précision (un entier positif)
  — le type `f` (pour *flottant*)
— Ligne 7 : l'approximation donnée est incorrecte, par défaut, on ne peut pas espérer plus d'une quinzaine de chiffres significatifs corrects.

Même comportement si le nombre à formater est de type entier ou s'il possède moins de chiffres significatif que la précision demandée :

```
x=42.5
print(x)
print(f"{x:.2f}")

x=42
print(x)
print(f"{x:.2f}")
```

```
42.5
42.50
42
42.00
```

La représentation obtenue avec le spécificateur de précision est un arrondi et non pas une troncature, autrement dit, c'est la valeur la plus proche du nombre et ayant le nombre de chiffres indiqué qui est retenue :

```
1 x=2/3
2 print(x)
3
4 print(f"{x:.2f}")
```

```
5 0.6666666666666666
6 0.67
```

— Ligne 6 : si le formatage avait tronqué la valeur, le résultat aurait été `0.66`.

Échappement des accolades dans une f-chaîne

L'accolade est un caractère typographique utilisé dans certains domaines. Par exemple, en mathématiques, le contenu d'un ensemble est représenté en le plaçant entre deux accolades. Ainsi, $E = \{42, 421\}$ est l'ensemble formé de deux éléments 42 et 421.

Ainsi, on peut se demander comment, par exemple, créer une f-chaîne qui affiche un ensemble mathématique ayant deux éléments ? La réponse est que pour représenter une accolade dans une f-chaîne, on **double** l'accolade (ce qu'on appelle un *échappement*) :

```
a = 42
b = 421
s=f"E = {{{10*a}, {b}}}"

print(s)
```

```
E = {420, 421}
```

— Ligne 3 : les deux premières accolades permettent de désigner la première accolade ligne 6 ; la paire d'accolade qui suit (`{10*a}`) est un champ de remplacement qui permet de placer l'élément `420` de la ligne 2. De même pour `{b}` qui place 421.

Quelques possibilités et limitations des f-chaînes

Le problème du séparateur de chaîne littérale

Comme pour des chaînes littérales, un caractère apostrophe ou guillemet appartenant à la chaîne, comme le guillemet dans l'exemple ci-dessous :

```
La fable "Le loup et l'agneau" de la Fontaine
```

peut entrer en collision avec le délimiteur de la chaîne. On NE peut PAS y remédier en échappant le quote avec une contre-oblique mais on peut y remédier :

— soit en changeant de quote (c'est rarement impossible),
— soit en utilisant une variable temporaire.

Ainsi :

```
x=3

# changement de quote
s = f"{'Pomme'*x}"
print(s)

# variable
fruit = "Pomme"
s = f"{fruit * x}"
print(s)
```

```
PommePommePomme
PommePommePomme
```

Le problème du backslash

L'expression à évaluer ne peut pas contenir de vrai backslash litteral :

```
print(f"Le nombre Pi : { '\u03C0'}")
```

```
    print(f"Le nombre Pi : { '\u03C0'}")
          ^
SyntaxError: f-string expression part cannot include a backslash
```

même si le backslah est une continuation de ligne dans une chaîne triple :

```
s = f"""{x+\
1}"""
```

```
    1}"""
SyntaxError: f-string expression part cannot include a backslash
```

Pour y remédier, on utilise une variable temporaire :

```
pi = "\u03C0"
print(f"Le nombre Pi : {pi}")
```

```
Le nombre Pi :
```

Toutefois, une double contre-oblique est possible (pour échapper la contre-oblique donc) :

```
nom= "pascal"
s = f"Dossier C:\\User\\{nom}"

print(s)
```

```
Dossier C:\User\pascal
```

Le problème du symbole deux-points (:)

Les expressions `lambda` peuvent poser un problème syntaxique car elles contiennent un caractère : qui sert à introduire une spécification de formatage. On peut néanmoins employer des expressions `lambda` à condition qu'elles soient entourées de parenthèses :

```
L=[42, -81, -12, 31, 82]
s=f"Les négatifs puis les positifs : {sorted(L, key=lambda z:z>=0)}"
print(s)
```

```
Les négatifs puis les positifs : [-81, -12, 42, 31, 82]
```

C'est le même problème pour l'opérateur walrus :

```
s =f"Le suivant de {(x:=42)} est {x+1}"
print(s)
```

```
Le suivant de 42 est 43
```

Chaînes triples

Les f-chaînes et les chaînes brutes sont compatibles :

```
z="Saut de ligne"

jours  = rf"{z} : \n"
print(jours)

jours  = fr"{z} : \n"
print(jours)
```

```
Saut de ligne : \n
Saut de ligne : \n
```

Espaces libres dans un champ de remplacement

On est libre de placer des espaces dans les expressions entre accolades et de placer des saut de lignes dans une f-chaîne triple, cela n'aurra pas d'influence sur le contenu de la chaîne créée :

```
nom = "Arthur"
age = 30

s = f"Son nom est {nom} et il a {      age      +      10     } ans\n"
print(s)
```

```
s = f"""Son nom est {nom}.
Il a {age + 10} ans"""
print(s)
```

```
Son nom est Arthur et il a 40 ans

Son nom est Arthur.
Il a 40 ans
```

Justaposition de chaînes littérales

Comme pour les chaînes littérales, il est possible de juxtaposer dans du code Python des f-chaînes et des chaînes littérales en les séparant par des espaces sur une même ligne pour obtenir une f-chaîne :

```
nom = "Arthur"
age= 30
s= "Son "  f"nom est {nom}" f"et il a {age+10}" " ans"
print(s)
```

```
Son nom est Arthuret il a 40 ans
```

Toutefois, un champ de remplacement ouvert dans une chaîne littérale juxtaposée doit se refermer dans la même chaîne littérale. Par exemple, le code suivant est invalide :

```
nom = "Arthur"
age= 30
s= f"Son nom est {nom " f" * 2} et il a {age+10} ans"
print(s)
```

```
    s= f"Son nom est {nom " f" * 2} et il a {age+10} ans"
      ^
SyntaxError: f-string: expecting '}'
```

Divers

Les accolades doivent contenir une expression à évaluer :

```
x=42
s=f"x+1={}"
print(s)
```

```
  File "a.py", line 2
    s=f"x+1={}"
     ^
SyntaxError: f-string: empty expression not allowed
```

Il n'est pas interdit qu'une f-chaîne n'ait aucun champ de remplacement :

```
s=f"Coucou !"
print(s)
```

```
Coucou !
```

Une f-chaîne peut aussi être préfixée par F majuscule :

```
nom = "Arthur"
age= 30
s= F"Son nom est {nom} et il a {age+10} ans"
print(s)
```

```
Son nom est Arthur et il a 40 ans
```

2 Largeur d'un champ

Fenêtre de champ dans une f-chaîne

On souhaite parfois placer un espacement autour d'un champ de remplacement, souvent pour des raisons d'alignement vertical. Par exemple, pour obtenir les chaînes formatées suivantes

```
1 coloration : blanc        → noir
2 taille     : gigantesque  → minuscule
3 vitesse    : lent         → rapide
```

— Lignes 1-3 : on observe que les mots correspondants sont alignés verticalement ainsi que les séparateurs : et →.

Un gabarit sans spécificateur de formatage permet difficilement d'obtenir le même effet d'alignement vertical :

```
LL = [["blanc", "noir", "coloration"],
      ["gigantesque", "minuscule", "taille"],
      ["lent", "rapide", "vitesse"]]

for L in LL:
    s = f"{L[2]} : {L[0]} → {L[1]}"
    print(s)
```

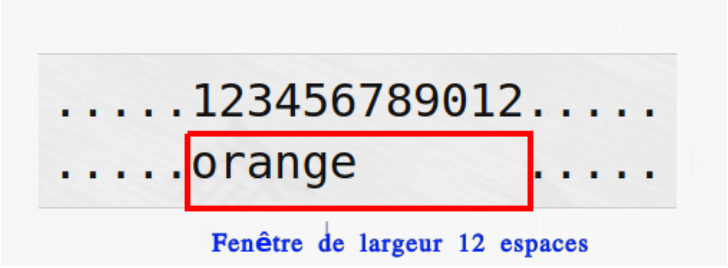
```
coloration : blanc → noir
taille : gigantesque → minuscule
vitesse : lent → rapide
```

Maintenant, regardons cet exemple pour illustrer la syntaxe définissant une largeur du champ de remplacement :

```
fruit = "orange"
print(".....123456789012.....")
s=f".....{fruit:12}....."
print(s)
```

```
.....123456789012.....
.....orange      .....
```

On voit, dans la sortie, que la valeur du champ de remplacement, la chaîne `orange`, est placée dans une « fenêtre » de largeur 12 :



Ainsi, pour placer la valeur d'une expression, disons `expr`, dans un fenêtre de largeur `w`, le champ de remplacement à placer dans la f-chaîne aura la syntaxe `{expr :w}`.

La largeur est le nombre total de caractères de la « fenêtre ». Par défaut, des caractères `" "` (espace) constituent le « bourrage » de la fenêtre de largeur donnée.

Le formatage présenté tout au début du paragraphe a été obtenu avec le code suivant :

```
LL = [["blanc", "noir", "coloration"],
      ["gigantesque", "minuscule", "taille"],
      ["lent", "rapide", "vitesse"]]

for L in LL:
    s = f"{L[2]:10} : {L[0]:12} → {L[1]:1}"
    print(s)
```

```
coloration : blanc        → noir
taille     : gigantesque  → minuscule
vitesse    : lent         → rapide
```

Noter que, parfois, le terme de *champ* se réfère autant à une valeur de remplacement qu'à l'emplacement de cette valeur dans la chaîne formatée.

Voyons sur un exemple comment le remplissage se fait en fonction de la dimension de l'objet qui remplace :

```
1 for v in ['', "Hello!", "Hello World!"]:
2     s = f"~~~~~{v:10}~~~~~"
3     print(s)
```

```
4 ~~~~~          ~~~~~
5 ~~~~~Hello!    ~~~~~
6 ~~~~~Hello World!~~~~~
```

— Ligne 2 : un champ de 10 caractères sera inséré.

— Ligne 2 : les tildes servent de ligne-étalon pour comparer avec les lignes suivantes.

— Ligne 6 : le champ est rempli des 6 caractères du mot `Hello!` et de 4 espaces (cf. la ligne-étalon)

— Ligne 7 : si le remplacement déborde le champ, le remplacement complet est effectué et aucun blanc n'est placé. Dans ce cas, le champ n'a eu aucun effet visible.

Par défaut, chaque élément est placé **à gauche** du champ. S'il reste de la place dans le champ, le formatage y place des espaces. La largeur du champ est le nombre total de caractères de la

fenêtre. Si l'objet à placer dans le champ est plus large que la largeur du champ, l'objet se place normalement et le champ n'est pas visible.

Alignement, remplissage dans une f-chaîne

Par défaut, l'alignement est fait au début du champ, c'est-à-dire à gauche. Il est possible de modifier ce comportement en utilisant un des spécificateurs d'alignement > ou ^ :

```
1 v = "bonjour"
2
3 s = f"----|{v:20}|----"
4 print(s)
5
6 s = f"----|{v:>20}|----"
7 print(s)
8
9 s = f"----|{v:^20}|----"
10 print(s)
11
12 s = f"----|{v:<20}|----"
13 print(s)
```

```
14 ----|bonjour              |----
15 ----|             bonjour|----
16 ----|      bonjour        |----
17 ----|bonjour              |----
```

— Ligne 3 : rappel du comportement par défaut : alignement à gauche.
— Ligne 6 : spécificateur > pour obtenr un alignement à droite du champ.
— Ligne 9 : spécificateur ^ pour obtenr un alignement centré dans le champ
— Ligne 12 : par homogénéité, le comportement par défaut a aussi son spécificateur : <.

Les parties invisibles d'un champ sont, par défaut, remplies de caractères espace.
< est le spécificateur du comportement par défaut.

Remplissage

Il est possible de remplir les parties vides d'un champ avec, par exemple, une suite de caractères comme des points (penser à des points de suspension), des tirets, etc. Ainsi :

```
Nom : .......Nobel........
Prénom : .......Alfred.......
```

Ce caractère est dit *caractère de remplissage*. Le caractère de remplissage doit immédiatement suivre le séparateur : et précéder un spécificateur d'alignement.

```
1 v = "maintenant"
2
3 s = f"hier {v:_<20} demain"
4 print(s)
5
```

```
6 s = f"hier {v:_>20} demain"
7 print(s)
8
9 s = f"hier {v:_^20} demain"
10 print(s)
```

```
11 hier maintenant__________ demain
12 hier __________maintenant demain
13 hier _____maintenant_____ demain
```

— Lignes 5 : la spécification de formatage est _>20 donc le remplissage est accompli par le caractère « blanc souligné » _.

Bien que le comportement par défaut soit l'alignement à gauche, le spécificateur < d'alignement à gauche doit être mentionné pour remplir par un autre caractère que l'espace :

```
v = "bonjour"
s = f"----|{v:_20}|----"
print(s)
```

```
    s = f"----|{v:_20}|----"
ValueError: Invalid format specifier
```

Pour disposer d'un champ intact, il suffit de remplir avec le caractère vide :

```
v = ''
s = f"hier {v:_<20} demain"
print(s)
```

```
hier ____________________ demain
```

Le caractère accolade fermante ne peut être utilisé pour effectuer un remplissage :

```
v = ''
s = f"hier {v:}}<20} demain"
print(s)
```

```
    s = f"hier {v:}}<20} demain"
          ^
SyntaxError: f-string: single '}' is not allowed
```

Champs pour des nombres dans une f-chaîne

Le principe de la fenêtre de champ pour un nombre est le même que pour n'importe quelle chaîne : un nombre (vu comme chaîne de caractères) est placé dans une fenêtre ayant une certain largeur, avec la possibilité de choisir entre un placement à gauche, à droite ou au centre de la fenêtre. Cela permet d'obtenir des effets d'alignement.

```
1 x=10
2 y=2030
3 z=8
4
```

```
5 addition=f"{x:>6}\n+{y:>5}\n+{z:>5}\n------\n{x+y+z:>6}"
6
7 print(addition)
```

```
8      10
9  + 2030
10 +    8
11 ------
12   2048
```

— Lignes 5 : tous les nombres sont alignés à droite dans un champ de largeur 6 ou 5 (à cause du signe +).

— Ligne 8-12 : les nombres sont convenablement alignés.

Il était même plus lisible d'utiliser une chaîne triple :

```
x=10
y=2030
z=8

addition=f"""\
{x:>6}
+{y:>5}
+{z:>5}
------
{x+y+z:>6}"""

print(addition)
```

Signe et premier chiffre dans une f-chaîne

C'est une option de remplissage assez rare d'utilisation et permettant de répéter un caractère entre le signe d'un nombre et son premier chiffre :

```
1 s = f"____{42:X=9}____{421:X=3}____{-42:X=9}____"
2 print(s)
```

```
3 ____XXXXXXX42____421____-XXXXXX42____
```

— Ligne 1 : par exemple, dans le dernier champ de remplacement `{:X=9}`, un champ de largeur 9 est réservé pour placer un nombre ainsi que son signe. L'espace entre le signe éventuel et le premier chiffre du nombre est rempli par des caractères `X`. Ainsi, dans notre exemple, moins de 9 caractères vaudront `"X"`, plus précisément, il y en aura 6=9-3 (le nombre `-42` utilise 3 caractères).

L'usage le plus courant est de placer des zéros. Cet usage tellement usuel qu'il est aussi rendu possible par le spécificateur de type `0n` où `n` est un entier littéral qui indique le nombre total de chiffres à formater :

```
s = f"Agent {7:0=3}"
print(s)
```

```
modele = f"Agent {7:03}"
print(s)
```

```
Agent 007
Agent 007
```

3 Les nombres

Formatage d'un flottant dans une f-chaîne

Le spécificateur de formatage le plus usuel pour les flottants est `f` :

```
x=22/7
print(x)

s = f"{x}"
print(s)

s = f"{x:f}"
print(s)
```

```
3.142857142857143
3.142857142857143
3.142857
```

Utiliser ce spécificateur `f` sans rien d'autre est d'intérêt réduit. Le formatage limite la partie décimale a au plus 6 décimales.

La dernière décimale du formatage peut avoir subi un arrondi :

```
x=3.141592653589793
print(x)

s = f"{x:f}"
print(s)
```

```
3.141592653589793
3.141593
```

Appliquer un formatage flottant à un entier fera apparaître 6 décimales valant toutes 0 :

```
s = f"{2038:f}"
print(s)
```

```
2038.000000
```

Zéros initiaux dans une f-chaîne

Considérons une date telle que **01/09/2030**. On observe que le nombre 1 ou le nombre 9 sont précédés d'un chiffre 0. Il existe une option de formatage qui permet de placer un certain nombre de zéros avant un nombre présent dans une chaîne.

Voici un exemple

```
1 s = f"____{42:09}________{421:02}_______"
2 print(s)
```

```
3 ____000000042________421_______
```

— Ligne 1 : la spécificateur `09` indique

  — qu'un champ de 9 caractères de large et destiné à recevoir un nombre est créé

  — que le formatage placera le nombre dans le champ et le fera précéder d'autant de zéros que nécessaire pour remplir le champ.

— Ligne 3 : si le nombre déborde du champ, aucun zéro n'est placé.

Si le nombre est négatif, les zéros sont placés, comme on s'y attend, entre le signe moins et le premier chiffre :

```
s = f"____{-42:09}________{-421:02}_______"
print(s)
```

```
____-00000042________-421_______
```

Conversion dans certaines bases dans une f-chaîne

Par défaut, les nombres sont formatés en base 10. On souhaite parfois que le nombre soit converti dans l'une des bases suivantes : 2, 8 ou 16. Pour cela il suffit de placer, en dernière position de la spécification de formatage un code de conversion donné dans le tableau suivant :

Base	Spécificateur	Nom usuel
2	`b`	binaire
8	`o`	octal
10	`d`	décimal
16	`x`	hexadécimal
16	`X`	hexadécimal

Voici un exemple d'utilisation de chaque spécificateur :

```
1 print(f"{42:b}")
2 print(f"{42:o}")
3 print(f"{42:d}")
4 print(f"{42:}")
5 print(f"{42:x}")
6 print(f"{42:X}")
```

```
7 101010
8 52
9 42
10 42
11 2a
12 2A
```

— Lignes 7-12 : noter qu'il n'y a aucune marque dans la chaîne qui montre en quelle base le nombre est représenté.

— Lignes 3 et 4 :le spécificateur `d` correspond au comportement par défaut et peut être omis.

— Lignes 5-6 et 11-12 : noter que la seule différence porte sur la casse majuscule/minuscule des chiffres en lettres.

Le spécificateur `x` formate les chiffres hexadécimaux sous forme de lettres minuscules (a, b, c, d, e, f) et le spécificateur `X` formate les chiffres hexadécimaux sous forme de lettres majuscules (A, B, C, D, E, F).

Un spécificateur de conversion doit toujours être le dernier spécificateur (autrement dit, le plus à droite) de la spécification de formatage.

Forme alternée

Python (à l'instar de beaucoup de langages) permet de représenter les entiers en base 2, 8 ou 16 sous leur forme littérale (je parle de nombres littéraux, pas de chaînes littérales). Ainsi, voici les nombres littéraux représentant 42 en Python dans ces bases :

```
b2 = 0b101010
b8 = 0o52
b16 = 0x2a
B16 = 0X2A

print(b2, b8, b16, B16)
```

```
42 42 42 42
```

Comme on le voit ci-dessus, pour représenter 42 par exemple en base 8, on utilise son écriture en base 8, qui est 52 puisque $5 \times 8^1 + 2 \times 8^0 = 42$ et que l'on fait précéder du préfixe **0o**.

Ainsi, ces représentations utilisent un préfixe commençant par l'entier 0 et un caractère.

Par ailleurs, le formatage de chaînes permet, à partir d'un entier donné, de produire des chaînes littérales représentant cet entier dans les bases ci-dessus. Par exemple, à partir de l'entier 42, on peut produire par formatage la chaîne **0b101010** (`b` comme *binary*). Pour cela, on utilise le spécificateur de formatage `#`. Par exemple, pour 42 et la base 2 :

```
x = 42
s = f"{x:#b}"

print(s)
```

```
0b101010
```

Le tableau suivant indique les préfixes pour chaînes littérales et les spécificateurs de formatage utilisés en Python :

Base	Spécificateur	Préfixe
2	`b`	`0b`
8	`o`	`0o`
10	`d`	aucun
16	`x`	`0x`
16	`X`	`0X`

— Ligne 2 : pour la base 8, noter que le préfixe n'est pas le préfixe classique (qui est `0` et non un `o` minuscule).

Ainsi :

```
print(f"{42:#b}")
print(f"{42:#o}")
print(f"{42:#d}")
print(f"{42:#x}")
print(f"{42:#X}")
```

```
0b101010
0o52
42
0x2a
0X2A
```

Signalons que lorsque le spécificateur contient le signe `#`, on le qualifie de forme alternée, cette terminologie vient du C et de Java.

Notation scientifique dans une f-chaîne

Python permet de représenter des constantes flottantes avec la notation scientifique stricte, à savoir $a \times 10^n$ où a est un flottant, appelé parfois *mantisse* dont la partie entière au signe près est un flottant entre 1 (inclus) et 10 (exclu).

Le spécificateur de formatage en notation scientifique est `e` ou `E` (cette lettre rappelle la première lettre du mot *exposant*) :

```
print(f"{2030.4248 :f}")
print(f"{2030.4248 :e}")
print(f"{2030.4248 :E}")
```

```
2030.424800
2.030425e+03
2.030425E+03
```

La précision de la mantisse est de 6 décimales. L'exposant est placé dans un champ de 2 chiffres. Le spécificateur `e` formate la lettre `e` en minuscule, et le spécificateur `E` formate en majuscule.

Mettre en relation avec unité sur les nombres littéraux.

Nombre de chiffres

Pour formater un nombre en notation scientifique, en utilisant un spécificateur de précision, on peut contrôler le nombre de chiffres significatifs de la mantisse :

```
print(f"{2030.4248:.2e}")
```

```
2.03e+03
```

Signe + dans une f-chaîne

Espace et signe

On souhaite parfois obtenir un formatage de nombres comme ci-dessous :

```
a = -42
b =  2024
```

— Lignes 1-2 : les nombres 42 et 2024 sont alignés verticalement du côté gauche.

Plus précisément, pour des raisons d'alignement, on souhaite qu'un nombre négatif soit formaté comme on en a l'habitude (unique signe qui précède le premier chiffre) et qu'un nombre positif soit formaté sans signe mais précédé d'un unique espace.

Le code correspondant à l'exemple ci-dessus est :

```
1 s=f"a = {-42: }"
2 t=f"b = {2024: }"
3 print(s)
4 print(t)
```

— Lignes 1 ou 2 : le spécificateur consiste seulement en un espace.

Pour placer un espace à la place du signe + devant un nombre, on utilise un spécificateur constitué d'un seul caractère espace qui est donc placé matériellement après le séparateur : et avant l'accolade fermante.

Signe + obligatoire

Parfois, on souhaite que le formatage d'un nombre montre toujours le signe + ou - (ce dernier étant obligatoire si le nombre est négatif). Pour cela, il suffit d'utiliser le spécificateur +

```
s=f"Rennes : {42:+}°C\nMoscou : {-24:+}°C"
print(s)
```

```
Rennes : +42°C
Moscou : -24°C
```

Noter bien qu'un spécificateur de formatage est nécessaire, le code naïf suivant ne fonctionne pas :

```
1 print(+42)
```

```
2 42
```

— Ligne 1 : le signe + est bien présent

— Ligne 2 : le signe + est absent : en effet l'expression +42 est *évaluée* avant d'être affichée.

Séparateur de milliers dans une f-chaîne

Pour des raisons de lisibilité, l'usage veut qu'on formate des nombres longs en séparant les chiffres par groupes de trois. Un spécificateur permet cette séparation :

```
p = -6542120359.4521263
s = f"prix = {p:,} euros"
print(s)

p = 421
s = f"prix = {p:,} euros"
print(s)
```

```
prix = -6,542,120,359.4521265 euros
prix = 421 euros
```

Le spécificateur consiste juste en une virgule :

,

Le séparateur est le séparateur utilisé dans le monde anglo-saxon, à savoir une virgule.

La partie décimale n'est pas formatée par groupe de 3 chiffres et il ne semble pas qu'un spécificateur le permette.

Si le nombre a moins de trois chiffres, aucune action de formatage spécifique n'est réalisée.

Respect des conventions linguistiques

Par défaut, le séparateur des chiffres d'un nombre en bloc de trois chiffres est la virgule. L'usage dans les pays francophones est d'utiliser plutôt un espace. Le spécificateur `n` permet de formater en respectant cet usage. Cela suppose que Python soit installé sur une machine où la locale est francophone.

```
1 import locale
2 ma_locale = locale.setlocale(locale.LC_ALL, '')
3 print(ma_locale)
4
5 x = 2099.236569633
6 print(f"{x:n}")
```

```
7 fr_FR.UTF-8
8 2099,24
```

— Lignes 2-3 : chargement de la locale pour toutes les catégories et affichage (ligne 6) de la locale utilisable.

— Ligne 6 : le spécificateur consiste juste en `n`

— Ligne 8 : on observe la séparation par un espace des deux blocs de chiffres.

La partie décimale a été tronquée à deux chiffres. Même si on tente de garder plus de chiffres significatifs, la partie décimale n'est pas découpée par groupes de 3 chiffres :

```
1 import locale
2 ma_locale = locale.setlocale(locale.LC_ALL, '')
3
4 x = 2099.236569633
5 print(f"{x:.10n}")
```

```
6 2099,23657
```

Pourcentage dans une f-chaîne

Le spécificateur % permet de formater un nombre en pourcentage.

```
print("{:%}".format(3/4))
```

```
75.000000%
```

— On notera que le séparateur décimal est ici un point et non une virgule.

Plus précisément, si x est le nombre à formater, le formatage consiste :

— à générer en flottant le pourcentage $100x$
— à adjoindre le symbole %.

L'intérêt semble limité.

4 Avancé, spécialisé ou secondaire

Usage de = pour débogage

L'exemple ci-dessous va servir d'illustration :

```
x = 4
s  = f"Le nombre {    10 * x + 2    =       } est universel"
print(s)
```

```
Le nombre     10 * x + 2    =       42 est universel
```

Depuis Python 3.8, après une expression dans un champ de remplacement, on peut placer un signe =. Ce simple ajout permet un débogage du code car la chaîne contiendra à l'affichage successivement :

— le code Python de l'expression (dans l'exemple, c'est `10 * x + 2`)
— le signe =
— la valeur de l'expression.

En outre, tous les placements d'espaces autour de l'expression et du signe égal seront rendus à l'identique dans l'affichage. Une fois débogué, il suffit de retirer le signe =, ce qui dans notre exemple donne :

```
x = 4
s  = f"Le nombre {    10 * x + 2          } est universel"
print(s)
```

```
Le nombre 42 est universel
```

et on observe que les espaces entre les accolades sont, cette fois, ignorés.
En outre, l'utilisation de = n'empêche pas de placer une spécification :

```
x=4
s = f"Le nombre {    10 * x + 2    =       :§^20} est universel"
```

```
print(s)
```

```
Le nombre     10 * x + 2    =       §§§§§§§§§42§§§§§§§§§ est universel
```

Particularité des f-chaînes

Une f-chaîne est autant une chaîne que du **code**. Il n'est pas possible de générer dynamiquement une f-chaîne à partir d'une chaîne arbitraire (ce que l'on qualifierait « d'effifier » une chaîne), à moins d'utiliser `eval`, voir les discussions ou ressources suivantes :

— How do I convert a string into an f-string?
— How to postpone/defer the evaluation of f-strings?
— PEP 501 – General purpose string interpolation
— What is the name of the internal function that "executes" an f-string?
— way to create fstring inside a fstring

Il est possible de créer un gabarit utilisable par la méthode `format` qui soit capable de recevoir un nombre arbitraire de champs de remplacement. Par exemple, soit à créer un gabarit qui permette, à partir d'une liste `L`, de formater une chaîne formée des éléments de `L` séparés par un tiret et des espaces, par exemple si `L` vaut

```
L = ["Mars", "Jupiter", "Uranus", "Neptune"]
```

alors le formatage attendu est

```
Mars - Jupiter - Uranus - Neptune
```

Voici comment créer (et utiliser) un tel gabarit utilisable par la méthode `format` :

```
def make_template(L):
    return ' - '.join(["{}"]*len(L))

L = ["Mars", "Jupiter", "Uranus", "Neptune"]
template = make_template(L)

print(template)
print(template.format(*L))
```

```
{} - {} - {} - {}
Mars - Jupiter - Uranus - Neptune
```

Pour réaliser la même chose avec des f-chaînes, je vois difficilement comment se passer de la fonction `eval` :

```
def make(L):
    t=' - '.join([f"{{L[{i}]}}" for i in range(len(L))])
    fstring = f'f"{t}"'
    print(fstring)
    return eval(fstring)

L = ["Mars", "Jupiter", "Uranus", "Neptune"]
```

```
print(make(L))
```

```
f"{L[0]} - {L[1]} - {L[2]} - {L[3]}"
Mars - Jupiter - Uranus - Neptune
```

Cumul de spécificateurs dans une f-chaîne

Une spécification de formatage peut contenir plusieurs spécificateurs successifs. Voici un exemple commenté :

```
for x in [1000 * 22/7, 1000 * 333/106]:
    s = f"1000 * pi = {x:~^ 20,.2f}"
    print(s)
```

```
1000 * pi = ~~~~~ 3,142.86~~~~~~
1000 * pi = ~~~~~ 3,141.51~~~~~~
```

Dans l'exemple, il n'y a qu'un champ de remplacement. Sa spécification est `:~^ 20,.2f` qui est composée de six spécificateurs :

— un spécificateur de remplissage centré (cf. le spécificateur `^`) par des caractères `~`
— l'espace (résultat : place un unique espace avant un nombre positif)
— un champ de largeur 20
— le séparateur de milliers (spécificateur virgule)
— précision (2 décimales)
— type flottant

L'ordre des spécificateurs n'est pas arbitraire ; de gauche à droite, on trouve des spécificateurs

— d'alignement (`<`, `^`, `>` et plus rarement `=`), précédé d'un **unique** caractère de remplissage
— de signe (`+` ou espace)
— de préfixe de conversion dans certaines bases (`#`)
— de remplissage par des zéros (`0`)
— de largeur de champ (un entier)
— de groupement par blocs de 3 chiffres de la partie entière (une virgule)
— de précision (de la forme `.n` où `n` est un entier)
— de type (par exemple `f`, `x`, `n` ou `%`)

Les spécificateurs sont optionnels, certains peuvent être omis (comme le type). Certains spécificateurs ne s'appliquent qu'à des nombres, voire des nombres d'un certain type. La plupart du temps, le bon sens permet de trancher.

La documentation affirme (sans que ce soit très clair : *The result (?) is then formatted using ...*) que la spécification de formatage est identique à celle qu'on utilise avec la méthode `format` mais la grammaire indiquée pour les f-chaînes ne le reflète pas.

Voici le lien vers la documentation concernant l'ordre des spécificateurs pour la méthode format : Format Specification Mini-Language

Champs emboîtés dans une f-chaîne

Considérons la f-chaîne suivante :

```
s = f"__{x:.>20}____{y:-^10}__"
```

Voici un exemple d'utilisation de cette f-chaîne :

```
x, y = 2030, 2038
s = f"__{x:.>20}____{y:-^10}__"
print(s)
```

```
__..............2030____---2038---__
```

Décrivons cette f-chaîne. Elle contient deux champs de remplacement, à savoir `{x:.>20}__` et `{y:-^10}` ; ces champs sont **séparés**. Chacun de ces champs est composé

— de l'expression dont la valeur est à remplacer (`x` puis `y`)
— du séparateur :
— d'une spécification de formatage (`.>20` et `-^10`).

A priori, des champs de remplacement sont toujours **séparés**. Cependant, il est possible, sous certaines conditions, de placer des champs de remplacement dans une spécification de format :

```
1 x, y = 2030, 2038
2 fen1 = ".>20"
3 fen2 = "-^10"
4 s = f"__{x:{fen1}}____{y:{fen2}}__"
5 print(s)
```

— Ligne 4 : le premier champ de remplacement `{x:{fen1}}` a une spécification `{fen1}` qui contient un autre champ de remplacement (cf. les accolades). On parle alors de *champs emboîtés*.

Action d'un champ emboîté

Exécutons l'exemple précédent :

```
x, y = 2030, 2038
fen1 = ".>20"
fen2 = "-^10"
s = f"__{x:{fen1}}____{y:{fen2}}__"
print(s)
```

```
__..............2030____---2038---__
```

On observe que le résultat est le même que la première utilisation de la f-chaîne `s` ci-dessus. Le processus d'exécution est le suivant :

— tout champ, emboîté ou non, contient une expression. Par exemple, le champ `{fen1}` ci-dessus réfère à l'argument `".>20"`, la valeur de la variable `fen1` ;
— les champs de remplacement emboîtés sont d'abord remplacés par les arguments correspondants ce qui produit une f-chaîne implicite, ici `"{x:.>20}____{y:-^10}"`
— suivant le processus habituel, les champs non emboîtés sont remplacés par les expressions qu'ils contiennent.

Intérêt

L'intérêt des champs emboîtés est que cela permet de générer des spécificateurs de format dynamiques, par exemple avec des champs de largeur paramétrable.

Un exemple typique, lu dans ce message de forum, permet de paramétrer le nombre de décimales d'un flottant :

```
1 for prec in [0, 3, 6, 9]:
2     s = f"{22/7:.{prec}f}"
3     print(s)
```

```
4 3
5 3.143
6 3.142857
7 3.142857143
```

— Ligne 1 : une même f-chaîne va permettre de générer les chaînes de largeur variable (lignes 4-7)

— Ligne 4 : on formate successivement `"{22/7:.0f}"`, puis `"{22/7:.3f}"`, etc.

On peut faire des formatages plus complexes :

```
x=2038

for L in [['?', '>', ' ', 42,','], ['~', '^', '+', 20,'e']]:
    s = f"__{x:{L[0]}{L[1]}{L[2]}{L[3]}{L[4]}}__"
    print(s)
```

```
__???????????????????????????????? 2,038__
__~~~+2.038000e+03~~~~__
```

Drapeau de conversion dans une f-chaîne

Soit le code :

```
x = "Hello"
y = "Rose"
s = f"{x}{y:.^20}"

print(s)
```

```
Hello........Rose........
```

La f-chaîne `f"{x}{y:.^20}"` comporte 2 champs de remplacement. Examinons par exemple le 2e champ : `{y:.^20}`. Il comporte deux composantes :

— l'expression dont la valeur est à remplacer : y

— un spécificateur de formatage : `.^20`

Cette syntaxe peut-être enrichie. Pour que mes explications ne soient pas trop abstraites, voici la nouveauté sur un exemple :

```
x = "Hello"
y = "Rose"
s = f"{x}{y!a:.^20}"

print(s)
```

qui affiche

```
Hello.......'Rose'.......
```

Un champ de remplacement peut disposer, outre une expression et le spécificateur de formatage, d'une troisième composante : un drapeau de conversion. Ce drapeau est placé en 2e position, entre l'expression dont la valeur est à remplacer et avant les deux-points qui précèdent le spécificateur de formatage, cf. ci-dessus, les deux caractères `!a`.

Un drapeau peut prendre une des trois formes suivantes :

`!s` ou `!r` ou `!a`

Voici un exemple :

```
x = "Hello"
y = "Rosé"

s = f"{x}{y!s:.^20}"
print(s)

s = f"{x}{y!r:.^20}"
print(s)

s = f"{x}{y!a:.^20}"
print(s)
```

```
Hello........Rosé........
Hello.......'Rosé'.......
Hello.....'Ros\xe9'......
```

Lorqu'un objet `X` à remplacer est associé à un drapeau de conversion, l'objet `X` est converti en une chaîne `S` et c'est la chaîne qui fait l'objet du remplacement. La chaîne `S` dépend du drapeau :

— si le drapeau est `!s` alors S est `str(X)`

— si le drapeau est `!r` alors S est `repr(X)`

— si le drapeau est `!a` alors S est `ascii(X)` où `ascii` est une fonction de la bibliothèque standard (par exemple, en français, ça réécrit avec un échappement s'il y a un accent).

Dans les cas usuels, le drapeau `!s` n'a pas d'action :

```
x = "Hello"
y = "Rosé"

s = f"{x}{y!s:.^20}"
print(s)
```

```
s = f"{x}{y:.^20}"
print(s)
```

```
Hello........Rosé........
Hello........Rosé........
```

Il se peut, bien sûr, qu'il n'y ait pas de spécification :

```
x= "codé"
print(f"C'est {x!a}")
```

```
C'est 'cod\xe9'
```

Exemple avec une liste

Montrons un exemple où on veut placer des listes dans des champs de largeur prédéfinie afin d'obtenir un alignement :

```
[5, 12, 8]    est une liste
[50, 120, 80] est une liste
```

Si on essaye d'obtenir le résultat via le code suivant :

```
1 L = [5, 12, 8]
2 M = [50, 120, 80]
3
4 s = f"""{L:13} est une liste
5 {M:13} est une liste"""
6
7 print(s)
```

```
8     {M:13} est une liste"""
9 TypeError: unsupported format string passed to list.__format__
```

on obtiendra une erreur. En effet, on essaye d'appliquer une spécification de largeur (lignes 4 et 5) à une liste qui ne la définit pas.

En revanche, si on place un drapeau de conversion, la spécification s'appliquera à une chaîne ce qui a un sens cette fois :

```
L = [5, 12, 8]
M = [50, 120, 80]

s = f"""{L!s:13} est une liste
{M!s:13} est une liste"""

print(s)
```

```
[5, 12, 8]    est une liste
[50, 120, 80] est une liste
```

Troncature dans une f-chaîne

On peut vouloir qu'un champ de remplacement reçoive une chaîne dont on ne garde qu'un certain nombre des premiers caractères :

```
1 semaine=['lundi', 'mardi', 'mercredi', 'jeudi',
2           'vendredi', 'samedi', 'dimanche']
3
4 for i in range(7):
5     print(f"{semaine[i]:.3} {i + 1}/10/2030")
```

```
6 lun 1/10/2030
7 mar 2/10/2030
8 mer 3/10/2030
9 jeu 4/10/2030
10 ven 5/10/2030
11 sam 6/10/2030
12 dim 7/10/2030
```

— Ligne 5 : seuls les 3 premiers caractères du jour de la semaine seront utilisés pour remplacer le champ.

Cette utilisation de troncature semble être assez rare.

La même syntaxe sert à contrôler le nombre de décimales d'un flottant.

Concaténation avec une f-chaîne

Le formatage de chaînes permet accessoirement de concaténer ou de joindre des chaînes

```
a, b, c = "Uni","Ver","Sel"

s = f"{a}{b}{c}"
print(s)

s = f"{a.upper()}-{b.upper()}-{c.upper()}"
print(s)
```

```
UniVerSel
UNI-VER-SEL
```

Chapitre II

La méthode format

1 L'essentiel

La problématique des chaînes formatées

Le formatage de chaînes est une possibilité offerte par de nombreux langages de programmation de créer une chaîne de caractères (comprendre du texte) à partir d'un modèle prédéfini. Typiquement, on peut utiliser le formatage de chaîne pour générer une date en français puisqu'une telle date a la structure `jj/mm/aaaa` connue à l'avance ou encore l'heure qui a le format `hh:mm:ss`. Mais, au-delà de ces formats figés, de nombreuses compositions structurées comme un sudoku à remplir, une table de conjugaison ou un calendrier :

```
    Janvier 2038
di lu ma me je ve sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

sont avantageusement construites en utilisant des chaînes formatées.

La notion de chaîne formatée

Supposons qu'on veuille générer de façon automatisée des phrases ayant la structure suivante :

```
Son nom est titi et il a 8 ans.
Son nom est Arthur et il a 30 ans.
Son nom est Alfred et il a 100 ans.
```

D'une phrase à l'autre, seuls le nom et le nombre représentant l'âge changent. Les chaînes ci-dessus sont dites des *chaînes formatées* : toutes les chaînes ont même structure, ici de la forme

`Son nom est XXX et il a YYY ans.`

Seuls varient le nom `XXX` et l'âge `YYY`.

Dans le présent document j'appelerai la chaîne-modèle telle que

`Son nom est XXX et il a YYY ans.`

un *gabarit* ou encore un *modèle* ou parfois une *chaîne de formatage*. Les parties à remplacer sont appelées *champs de remplacement* (en anglais *remplacement field*). Un gabarit est le modèle à partir duquel d'autres chaînes vont être produites par substitution.

Les chaînes formatées avec la méthode `format`

Python permet de créer des chaînes formatées basées sur un gabarit.
Considérons le gabarit informel suivant :

`Son nom est XXX et il a YYY ans.`

Voici le code Python permettant de produire les chaînes formatées basées sur ce gabarit :

```
1 modele = "Son nom est {} et il a {} ans."
2
3 a = modele.format("titi", 8)
4 print(a)
5
6 b = modele.format("Arthur", 30)
7 print(b)
8
9 print(modele.format("Alfred", 100))
```

```
10 Son nom est titi et il a 8 ans.
11 Son nom est Arthur et il a 30 ans.
12 Son nom est Alfred et il a 100 ans.
```

— Ligne 1 : c'est la traduction Python du gabarit informel `Son nom est XXX et il a YYY ans.` C'est ici une chaîne littérale. Les champs de remplacement sont représentés par des paires d'accolades `{}` au lieu de XXX et YYY.

— Ligne 3 : construction de la chaîne formatée (qui est affichée ligne 10) à l'aide de la méthode standard `format`. La première paire d'accolades est remplacée dans le gabarit par le premier argument, de même pour la deuxième paire d'accolades.

— Lignes 6 et 9 : analogue à la ligne 3.

En résumé, les deux étapes de création d'une chaîne formatée sont

— la **construction du gabarit** : un gabarit est réalisé en Python avec une chaîne de caractères Python, le plus souvent une chaîne littérale (c'est-à-dire délimitée par des guillemets). Les parties « variables » du gabarit sont placées entre des paires d'accolades. Dans les cas les plus simples, les paires d'accolades ne contiennent rien ;

— la **génération de la chaîne formatée** : pour « remplir » le gabarit, on appelle la méthode `format` du type chaîne en la préfixant avec la chaîne gabarit. Dans le cas où les accolades sont de la forme `{}` ie ne contiennent rien, le remplissage du gabarit est positionnel : le k-e champ de remplacement du gabarit est remplacé par le k-e argument de la fonction. La méthode `format` peut prendre un nombre arbitraire d'arguments. Elle renvoie la chaîne formatée.

Numérotation des arguments

On peut placer des numéros au début d'une paire d'accolades présentes dans un gabarit pour se référer aux arguments de la méthode `format` suivant leur numéro d'ordre d'apparition en tant qu'arguments :

```
1 modele = "une {1}, une {2} et une {0}"
2 print(modele.format("pomme", "poire", "prune"))
```

```
3 une poire, une prune et une pomme
```

— Ligne 1 : le gabarit contient trois champs. Par exemple, le champ `{2}` se réfère à l'argument n°2 de la liste des arguments de la méthode `format`, c'est-à-dire le 3e argument de la liste.

Le premier argument de la méthode `format` est numéroté 0. Chaque paire d'accolades avec son contenu est appelé *champ de remplacement*. On dit aussi que le gabarit utilise les *arguments positionnels* de la méthode `format`. Chaque nombre placé entre les accolades est appelé `nom de remplacement`.

Les arguments de la méthode `format` peuvent servir plusieurs fois dans le gabarit ou certains arguments peuvent ne pas servir :

```
1 modele = "une {1}, une {2} et une {1}"
2 print(modele.format("pomme", "poire", "prune"))
```

```
3 une poire, une prune et une poire
```

— L'argument numéro 0 ne sert pas et l'argument numéro 1 sert deux fois.

Arguments nommés dans un champ de remplacement

Pour permettre une plus grande facilité d'utilisation, la méthode `format` accepte des arguments dits « nommés » :

```
1 modele = "Son nom est {prenom} et il a {age} ans."
2 a=modele.format(age=42, prenom= "Arthur")
3 print(a)
```

```
4 Son nom est Arthur et il a 42 ans.
```

— Ligne 1 : la gabarit `modele` possède deux champs de remplacement nommés : `{prenom}` et `{age}`.

— Ligne 2 : la méthode `format` est appelée sur le gabarit. Les arguments de l'appel sont nommés, l'un `age`, l'autre `prenom`. Les valeurs de ces arguments (respectivement `42` et `"Arthur"`) vont remplacer les champs du gabarit portant le même nom.

On peut aussi obtenir le même effet si les arguments sont des clés d'un dictionnaire que l'on décompresse dans l'appel :

```
1 modele = "Son nom est {prenom} et il a {age} ans."
2 dico={"age":42,"prenom":"Arthur"}
3
4 a=modele.format(**dico)
```

```
5 print(a)
```

```
6 Son nom est Arthur et il a 42 ans.
```

— Ligne 4 : le dictionnaire est décompressé en des arguments nommés, ce qui donne le même résultat que dans l'exemple précédent.

Dans le même registre, si les arguments sont en fait des variables locales (et donc présentes dans le dictionnaire des variables locales), on peut obtenir le même effet :

```
1 age=42
2 prenom = "Arthur"
3 modele = "Son nom est {prenom} et il a {age} ans."
4
5 a=modele.format(**locals())
6 print(a)
```

```
7 Son nom est Arthur et il a 42 ans.
```

— Lignes 1-3 : trois variables locales `age`, `prenom`, `modele`.

— Ligne 4 : `locals()` renvoie un dictionnaire dont les clés sont des chaînes de même nom que les variables locales. Comme dans l'exemple précédent, le dictionnaire est décompressé.

Toutefois, la syntaxe des arguments nommés n'est pas aussi souple que celle des f-chaînes puisque le champ de remplacement ne peut contenir qu'une clé de dictionnaire et pas une expression. Ainsi, le code suivant échoue :

```
1 modele = "Son nom est {prenom} et il a {age+10} ans."
2 a=modele.format(age=42, prenom= "Arthur")
3 print(a)
```

alors qu'une f-chaîne permettait d'écrire :

```
1 age=42
2 prenom= "Arthur"
3 a = f"Son nom est {prenom} et il a {age+10} ans."
4 print(a)
```

```
5 Son nom est Arthur et il a 52 ans.
```

Enfin, il est possible d'utiliser à la fois des arguments positionnels numérotés et des arguments nommés mais tous les arguments positionnels doivent être au début de la liste des arguments de la méthode `format` :

```
1 modele = "Son nom est {prenom} et, dans {1} mois et {0} jours, il aura {age} ans."
2 a=modele.format(2, 5, age=42, prenom= "Arthur")
3 print(a)
```

```
4 Son nom est Arthur et, dans 5 mois et 2 jours, il aura 42 ans.
```

Arguments sous forme de liste

Examinons l'exemple suivant :

```
1 modele = "le {1[2]} est {0}"
2 objets=["soleil", "stylo", "papier"]
3 a = modele.format("rouge", objets)
4 print(a)
```

```
5 le papier est rouge
```

— Ligne 1 : dans le champ de remplacement `{1[2]}`, le numéro `1` correspond à l'argument numéro 1 de l'appel de la méthode `format`, ici la liste à la ligne 2.

— Lignes 3 et 5 : le champ `{1[2]}` est remplacé par `objets[2]` et le champ `{0}` est remplacé par l'argument numéro 0, ie la chaîne `"rouge"`.

Un nom de champ de la forme `i[j]` signifie que le champ sera remplacé par `t[j]` où `t` désigne l'objet référencé par l'argument numéro `i` de la méthode `format`, ce qui suppose que l'objet `t` est accessible par l'opération d'indexation (c'est le cas, par exemple, si `t` est une liste).

De même, un nom de champ de la forme `toto[j]` signifie que le champ sera remplacé par `t[j]` où `t` désigne l'objet référencé par l'argument nommé `toto` de la méthode `format` :

```
1 modele = "le {objets[2]} est {0}"
2 t=["soleil", "stylo", "papier"]
3 a = modele.format("rouge", objets=t)
4 print(a)
```

```
5 le papier est rouge
```

— Ligne 1 : le champ de remplacement `{objets[2]}`, le nom `objets` correspond à un argument nommé de l'appel de la méthode `format`, ici la liste `t` à la ligne 2.

— Ligne 5 : le champ `{objets[2]}` est remplacé par t[2][2].

Échappement des accolades avec la méthode `format`

L'accolade est un caractère typographique utilisé dans certains domaines. Par exemple, en mathématiques, le contenu d'un ensemble est représenté en le plaçant entre deux accolades. Ainsi, $E = \{42, 421\}$ est l'ensemble formé de deux éléments 42 et 421.

Ainsi, on peut se demander comment, par exemple, créer un gabarit qui affiche un ensemble mathématique ayant deux éléments ? La réponse est que pour représenter une accolade dans un gabarit, on **double** l'accolade (ce qu'on appelle un *échappement*) :

```
1 print("E = {{{}, {}}}".format(42,421))
```

```
2 E = {42, 421}
```

— Ligne 1 : les deux premières accolades permettent de désigner la première accolade ligne 2 ; la paire d'accolade qui suit est un champ de remplacement qui permet de placer l'élément 42 de la ligne 2.

Cela peut servir à créer dynamiquement des gabarits utilisables par `format`, voici un exemple venant du code source de Panda :

```
1 fmt = '{{x: .{prec:d}g}}'
2 fmt_str = fmt.format(prec=get_option("display.precision"))
```

Spécification de formatage avec la méthode `format`

Jusqu'à présent un champ de remplacement (ie une paire d'accolades) dans un gabarit, contient uniquement une référence à un argument de la méthode `format`, ce qu'on appelle un *nom de champ* :

```
1 modele = "une {1}, une {2} et une {1}"
2
3 print(modele.format("pomme", "poire", "prune"))
```

— Ligne 1 : chaque champ de remplacement contient juste une référence à un argument de l'appel de `format`.

Mais, cette syntaxe peut être enrichie pour préciser dans chaque champ de remplacement une *spécification de formatage*, par exemple ici comment on formate une date :

```
1 modele = "{0} {2:02}/{1:02}/{3}"
2 a=modele.format("dimanche", 9, 1, 2030)
3 print(a)
```

```
4 dimanche 01/09/2030
```

— Ligne 1 : on compte 4 champs de remplacement (les 4 paires d'accolades) ; le 2e et le 3e contiennent une *spécification de formatage*. Par exemple, dans le champ `{1:02}`, la partie `02` à droite du symbole `:` est une spécification de formatage.

À droite du nom du champ, il est possible de placer un « spécification de formatage » qui précise **comment** la valeur à substituer doit être formatée. Pour signifier une spécification de formatage, après le nom du champ, on place le symbole `:` (*deux-points*) ; la spécification de formatage suit le symbole `:` jusqu'à l'accolade fermante.

En pratique, une spécification de formatage sert surtout à formater des **nombres** suivant leurs types (entiers, décimaux, etc) et à **gérer l'espacement** autour du champ de remplacement.

Reprenons l'exemple précédent :

```
1 modele = "{0} {2:02}/{1:02}/{3}"
2 a=modele.format("dimanche", 9, 1, 2030)
3 print(a)
```

```
4 dimanche 01/09/2030
```

— Ligne 1 : par exemple, le champ `{1:02}` signifie (ce sera expliqué en détail ultérieurement) que l'argument numéro 1 de la méthode `format` doit être un nombre et qu'il sera formaté en plaçant un **zéro** devant le nombre si le nombre n'est constitué que d'un seul chiffre.

— Ligne 4 : l'argument numéro 1, qui vaut `9` comme on le voit ligne 2, est donc formaté sous la forme `09`.

Spécificateurs de formatage

Précisons maintenant ce qu'on entend par « spécificateurs de formatage » :

```
1 modele = "{0} {2:02}/{1:02}/{3}"
2 a=modele.format("dimanche", 9, 1, 2030)
3 print(a)
```

```
4 dimanche 01/09/2030
```

— Ligne 1 : 02 est une **spécification de formatage**. En réalité, cette spécification se décompose en deux « spécificateurs » :

  — le spécificateur 0 pour spécifier que le nombre est, si nécessaire, précédé de zéros ;

  — le spécificateur 2 pour spécifier que deux chiffres seront représentés, cf. le 09 ligne 4.

Plus généralement, chaque spécification est constituée d'une succession de spécificateurs, jusqu'à 8 spécificateurs. On prendra soin de ne pas placer par mégarde d'espaces dans la spécification, après les deux points sinon l'espace sera visible, voire aura un effet inattendu :

```
1 modele = "{0} {2:02}/{1:02}/{3}"
2 a=modele.format("dimanche", 9, 1, 2030)
3 print(a)
```

```
4 dimanche 01/ 9/2030
```

— Ligne 1 : on a rajouté un espace dans l'avant-dernière accolade, juste avant le 02

— Ligne 4 : le zéro a disparu avant le 2 et a été remplacé par un un espace.

Par ailleurs, l'ordre des spécificateurs n'est pas sans importance. Au lieu de *spécificateur*, le terme d'*option de formatage* est aussi employé, et parfois aussi de *spécifieur* (anglicisme).

Partie décimale d'un flottant avec la méthode `format`

Le formatage des valeurs des flottants en Python peut faire apparaître jusqu'à 6 décimales. Le programmeur souhaite parfois n'en faire apparaître qu'un nombre plus réduit (ou plus important). Une spécificateur dit de *précision* permet de contrôler le nombre de chiffres décimaux nécessaires pour représenter un nombre flottant donné :

```
1 x=22/7
2 print(x)
3 print("{:.2f}".format(x))
4 print("{:.50f}".format(x))
```

```
5 3.142857142857143
6 3.14
7 3.14285714285714279370154144999105483293533325195312 5
```

— Ligne 3 : la spécification est ici `.2f` et contient deux spécificateurs :

  — point suivi de la valeur de la précision (un entier positif)

  — le type `f` (pour *flottant*)

— Ligne 7 : l'approximation donnée est incorrecte, par défaut, on ne peut pas espérer plus d'une quinzaine de chiffres significatifs corrects.

Même comportement si le nombre à formater est de type entier ou s'il possède moins de chiffres significatif que la précision demandée :

```
1 x=42.5
2 print(x)
```

```
3 print("{:.2f}".format(x))
4 x=42
5 print(x)
6 print("{:.2f}".format(x))
```

```
7 42.5
8 42.50
9 42
10 42.00
```

La représentation obtenue avec le spécificateur de précision est un arrondi et non pas une troncature, autrement dit, c'est la valeur la plus proche du nombre et ayant le nombre de chiffres indiqué qui est retenue :

```
1 x=2/3
2 print(x)
3 print("{:.2f}".format(x))
```

```
4 0.6666666666666666
5 0.67
```

— Ligne 5 : si le formatage avait tronqué la valeur, le résultat aurait été `0.66`.

Avec une f-chaîne, le contrôle du nombre de décimales se fait comme suit :

```
x=22/7
print(f"{x:.2f}")
```

d'où :

```
1 3.14
```

2 Largeur d'un champ

Fenêtre de champ avec la méthode `format`

On souhaite parfois placer un espacement autour d'un champ de remplacement, souvent pour des raisons d'alignement vertical. Par exemple, pour obtenir les chaînes formatées suivantes

```
1 coloration : blanc       → noir
2 taille     : gigantesque → minuscule
3 vitesse    : lent        → rapide
```

— Lignes 1-3 : on observe que les mots correspondants sont alignés verticalement ainsi que les séparateurs : et →.

Un gabarit sans spécificateur de formatage permet difficilement d'obtenir le même effet d'alignement vertical :

```
1 modele = "{2} : {0} → {1}"
2
3 print(modele.format("blanc", "noir", "coloration"))
4 print(modele.format("gigantesque", "minuscule", "taille"))
```

```
print(modele.format("lent", "rapide", "vitesse"))
```

```
coloration : blanc → noir
taille : gigantesque → minuscule
vitesse : lent → rapide
```

Maintenant, regardons cet exemple pour illustrer la syntaxe définissant une largeur du champ de remplacement :

```
fruit = "orange"
print(".....123456789012.....")
s=".....{:12}.....".format(fruit)
print(s)
```

```
.....123456789012.....
.....orange      .....
```

On voit, dans la sortie, que la valeur du champ de remplacement, la chaîne `orange`, est placée dans une « fenêtre » de largeur 12.

Ainsi, pour placer la valeur d'une expression, disons `expr`, dans un fenêtre de largeur, par exemple, `w`, le champ de remplacement à placer dans la f-chaîne aura la syntaxe `{expr :w}`, par exemple.

La largeur est le nombre total de caractères. Par défaut, des caractères `" "` (espace) constituent le « bourrage » de la fenêtre de largeur donnée.

Le formatage présenté tout au début du paragraphe a été obtenu avec le code suivant :

```
modele = "{2:10} : {0:12} → {1}"

print(modele.format("blanc", "noir", "coloration"))
print(modele.format("gigantesque", "minuscule", "taille"))
print(modele.format("lent", "rapide", "vitesse"))
```

```
coloration : blanc        → noir
taille     : gigantesque  → minuscule
vitesse    : lent         → rapide
```

Noter que, parfois, le terme de *champ* se réfère autant à une valeur de remplacement qu'à l'emplacement de cette valeur dans la chaîne formatée.

Voyons sur un exemple comment le remplissage se fait en fonction de la dimension de l'objet qui remplace :

```
modele = "~~~~~{:10}~~~~~"
print(modele.format(''))
print(modele.format("Hello!"))
print(modele.format("Hello World!"))
```

```
~~~~~          ~~~~~
~~~~~Hello!    ~~~~~
~~~~~Hello World!~~~~~
```

— Ligne 1 : un champ de 10 caractères sera inséré.

— Ligne 2 : La ligne-étalon pour comparer avec les lignes suivantes.

— Ligne 3 : le champ est rempli des 6 caractères du mot `Hello!` et de 4 espaces (cf. la ligne-étalon)

— Ligne 4 et 7 : si le remplacement déborde le champ, le remplacement complet est effectué et aucun blanc n'est placé. Dans ce cas, le champ n'a eu aucun effet visible.

Par défaut, chaque élément est placé **à gauche** du champ. S'il reste de la place dans le champ, le formatage y place des espaces. La largeur du champ est le nombre total de caractères du conteneur. Si l'objet à placer dans le champ est plus large que la largeur du champ, l'objet se place normalement et le champ n'est pas visible.

Alignement, remplissage avec la méthode `format`

Par défaut, l'alignement est fait au début du champ, c'est-à-dire à gauche. Il est possible de modifier ce comportement en utilisant un des spécificateurs d'alignement `>` ou `^` :

```
1  modele = "----|{:20}|----"
2  print(modele.format("bonjour"))
3
4  modele = "----|{:>20}|----"
5  print(modele.format("bonjour"))
6
7  modele = "----|{:^20}|----"
8  print(modele.format("bonjour"))
9
10 modele = "----|{:<20}|----"
11 print(modele.format("bonjour"))
```

```
12 ----|bonjour              |----
13 ----|              bonjour|----
14 ----|      bonjour        |----
15 ----|bonjour              |----
```

— Ligne 1 : rappel du comportement par défaut : alignement à gauche.

— Ligne 4 : spécificateur `>` pour obtenr un alignement à droite du champ.

— Ligne 7 : spécificateur `^` pour obtenr un alignement centré dans le champ

— Ligne 10 : par homogénéité, le comportement par défaut a aussi son spécificateur : `<`.

Les parties invisibles d'un champ sont, par défaut, remplies de caractères espace.

`<` est le spécificateur du comportement par défaut.

Remplissage

Il est possible de remplir les parties vides d'un champ avec, par exemple, une suite de caractères comme des points (penser à des points de suspension), des tirets, etc. Ainsi :

```
Nom : .......Nobel........
Prénom : .......Alfred.......
```

Ce caractère est dit *caractère de remplissage*. Le caractère de remplissage doit immédiatement suivre le séparateur `:` et précéder un spécificateur d'alignement.

```
1 modele = "hier {:_<20} demain"
2 print(modele.format("maintenant"))
3
4 modele = "hier {:_>20} demain"
5 print(modele.format("maintenant"))
6
7 modele = "hier {:_^20} demain"
8 print(modele.format("maintenant"))
```

```
9 hier maintenant__________ demain
10 hier __________maintenant demain
11 hier _____maintenant_____ demain
```

— Lignes 4 : la spécification de formatage est _>20 donc le remplissage est accompli par le caractère « blanc souligné » _.

Bien que le comportement par défaut soit l'alignement à gauche, le spécificateur < d'alignement à gauche doit être mentionné pour remplir par un autre caractère que l'espace :

```
1 modele = "----|{:_20}|----"
2 print(modele.format("bonjour"))
```

```
3 Traceback (most recent call last):
4 File "a.py", line 2, in <module>
5   print(modele.format("bonjour"))
6     ValueError: Invalid format specifier
```

Pour disposer d'un champ intact, il suffit de remplir avec le caractère vide :

```
1 modele = "hier {:_<20} demain"
2 print(modele.format(""))
```

```
3 hier ____________________ demain
```

Le caractère accolade fermante ne peut être utilisé pour effectuer un remplissage :

```
1 modele = "hier {:}<20} demain"
2 print(modele.format(""))
```

```
3 Traceback (most recent call last):
4   File "a.py", line 2, in <module>
5     print(modele.format(""))
6 ValueError: Single '}' encountered in format string
```

Champs pour des nombres avec la méthode `format`

Le principe de la fenêtre de champ pour un nombre est le même que pour n'importe quelle chaîne : un nombre (vu comme chaîne de caractères) est placé dans une fenêtre ayant une certain largeur, avec la possibilité de choisir entre un placement à gauche, à droite ou au centre de la fenêtre. Cela permet d'obtenir des effets d'alignement.

```
1 addition="{:>6}\n+{:>5}\n+{:>5}\n------\n{:>6}"
2 x=10
3 y=2030
4 z=8
5 print(addition.format(x,y,z, x+y+z))
```

```
6     10
7 + 2030
8 +    8
9 ------
10   2048
```

— Lignes 1 : tous les nombres sont alignés à droite dans un champ de largeur 6 ou 5 (à cause du signe +).

— Ligne 6-10 : les nombres sont convenablement alignés.

Il était même plus lisible d'utiliser une chaîne triple :

```
addition="""\
{:>6}
+{:>5}
+{:>5}
------
{:>6}"""
x=10
y=2030
z=8
print(addition.format(x,y,z, x+y+z))
```

Signe et premier chiffre avec la méthode `format`

C'est une option de remplissage assez rare d'utilisation et permettant de répéter un caractère entre le signe d'un nombre et son premier chiffre :

```
1 modele="____{:X=9}____{:X=3}____{:X=9}____"
2 print(modele.format(42, 421, -42))
```

```
3 ____XXXXXXX42____421____-XXXXXX42____
```

— Ligne 1 : par exemple, dans le dernier champ de remplacement `{:X=9}`, un champ de largeur 9 est réservé pour placer un nombre ainsi que son signe. L'espace entre le signe éventuel et le premier chiffre du nombre est rempli par des caractères `X`. Ainsi, dans notre exemple, moins de 9 caractères vaudront `"X"`, plus précisément, il y en aura 6=9-3 (le nombre `-42` utilise 3 caractères).

L'usage le plus courant est de placer des zéros. Cet usage tellement usuel qu'il est aussi rendu possible par le spécificateur de type `0n` où `n` est un entier littéral qui indique le nombre total de chiffres à formater :

```
1 modele="Agent {:0=3}"
2 print(modele.format(7))
3
```

```
4 modele="Agent {:03}"
5 print(modele.format(7))
```

qui affiche

```
1 Agent 007
2 Agent 007
```

3 Les nombres

Formatage d'un flottant avec la méthode `format`

Le spécificateur de formatage le plus usuel pour les flottants est `f` :

```
1 x=22/7
2 print(x)
3 print("{:f}".format(x))
```

```
4 3.142857142857143
5 3.142857
```

Le formatage limite la partie décimale a au plus 6 décimales.
La dernière décimale du formatage peut avoir subi un arrondi :

```
1 x=3.141592653589793
2 print(x)
3 print("{:f}".format(x))
```

```
4 3.141592653589793
5 3.141593
```

Appliquer un formatage flottant à un entier fera apparaître 6 décimales valant toutes 0 :

```
1 print("{:f}".format(2030))
```

```
2 2030.000000
```

Zéros initiaux avec la méthode `format`

Considérons une date telle que **01/09/2030**. On observe que le nombre 1 ou le nombre 9 sont précédés d'un chiffre 0. Il existe une option de formatage qui permet de placer un certain nombre de zéros avant un nombre présent dans une chaîne.
Voici un exemple

```
1 modele="____{:09}________{:02}_______"
2 print(modele.format(42, 421))
```

```
3 ____000000042________421_______
```

— Ligne 1 : la spécificateur `09` indique

  — qu'un champ de 9 caractères de large et destiné à recevoir un nombre est créé

— que le formatage placera le nombre dans le champ et le fera précéder d'autant de zéros que nécessaire pour remplir le champ.

— Ligne 2 : si le nombre déborde du champ, aucun zéro n'est placé.

Si le nombre est négatif, les zéros sont placés, comme on s'y attend, entre le signe moins et le premier chiffre :

```
1 modele="____{:09}________{:02}_______"
2 print(modele.format(-42, -421))
```

```
3 ____-00000042________-421_______
```

Conversion dans certaines bases avec la méthode `format`

Par défaut, les nombres sont formatés en base 10. On souhaite parfois que le nombre soit converti dans l'une des bases suivantes : 2, 8 ou 16. Pour cela il suffit de placer, en dernière position de la spécification de formatage un code de conversion donné dans le tableau suivant :

Base	Spécificateur	Nom usuel
2	`b`	binaire
8	`o`	octal
10	`d`	décimal
16	`x`	hexadécimal
16	`X`	hexadécimal

Voici un exemple d'utilisation de chaque spécificateur :

```
1 print("{:b}".format(42))
2 print("{:o}".format(42))
3 print("{:d}".format(42))
4 print("{:}".format(42))
5 print("{:x}".format(42))
6 print("{:X}".format(42))
```

```
7 101010
8 52
9 42
10 42
11 2a
12 2A
```

101010

52

42

2a

2A

— Lignes 7-12 : noter qu'il n'y a aucune marque dans la chaîne qui montre en quelle base le nombre est représenté.

— Lignes 3 et 4 :le spécificateur `d` correspond au comportement par défaut et peut être omis.

— Lignes 5-6 et 11-12 : noter que la seule différence porte sur la casse majuscule/minuscule des chiffres en lettres.

Le spécificateur `x` formate les chiffres hexadécimaux sous forme de lettres minuscules (a, b, c, d, e, f) et le spécificateur `X` formate les chiffres hexadécimaux sous forme de lettres majuscules (A, B, C, D, E, F).

Un spécificateur de conversion doit toujours être le dernier spécificateur (autrement dit, le plus à droite) de la spécification de formatage.

Forme alternée

Python (à l'instar de beaucoup de langages) permet de représenter les entiers en base 2, 8 ou 16 sous leur forme littérale (je parle de nombres littéraux, pas de chaîne littérales). Ainsi, voici les nombres littéraux représentant 42 en Python dans ces bases :

```
b2 = 0b101010
b8 = 0o52
b16 = 0x2a
B16 = 0X2A

print(b2, b8, b16, B16)
```

```
42 42 42 42
```

Comme on le voit ci-dessus, pour représenter 42 par exemple en base 8, on utilise son écriture en base 8, qui est 52 puisque $5 \times 8^1 + 2 \times 8^0 = 42$ et que l'on fait précéder du préfixe **0o**.

Ainsi, ces représentations utilisent un préfixe commençant par l'entier 0 et un caractère.

Par ailleurs, le formatage de chaînes permet, à partir d'un entier donné, de produire des chaînes littérales représentant cet entier dans les bases ci-dessus. Par exemple, à partir de l'entier 42, on peut produire par formatage la chaîne **0b101010** (`b` comme *binary*). Pour cela, on utilise le spécificateur de formatage `#`. Par exemple, pour 42 et la base 2 :

```
fmt = "{0:#b}"
x = 42
s = fmt.format(x)
print(s)
```

```
0b101010
```

Le tableau suivant indique les préfixes pour chaînes littérales et les spécificateurs de formatage utilisés en Python :

Base	Spécificateur	Préfixe
2	`b`	`0b`
8	`o`	`0o`
10	`d`	aucun
16	`x`	`0x`
16	`X`	`0X`

— Ligne 2 : pour la base 8, noter que le préfixe n'est pas le préfixe classique (qui est `0` et non un `o` minuscule).

Ainsi :

```
print("{0:#b}".format(42))
print("{0:#o}".format(42))
print("{0:#d}".format(42))
print("{0:#x}".format(42))
print("{0:#X}".format(42))
```

```
0b101010
0o52
42
0x2a
0X2A
```

Signalons que lorsque le spécificateur contient le signe `#`, on le qualifie de forme alternée, cette terminologie vient du C et de Java.

Notation scientifique avec la méthode `format`

Python permet de représenter des constantes flottantes avec la notation scientifique stricte, à savoir $a \times 10^n$ où a est un flottant, appelé parfois *mantisse* dont la partie entière au signe près est un flottant entre 1 (inclus) et 10 (exclu).

Le spécificateur de formatage en notation scientifique est `e` ou `E` (cette lettre rappelle la première lettre du mot *exposant*) :

```
print("{:f}".format(2030.4248))
print("{:e}".format(2030.4248))
print("{:E}".format(2030.4248))
```

```
2030.424800
2.030425e+03
2.030425E+03
```

La précision de la mantisse est de 6 décimales. L'exposant est placé dans un champ de 2 chiffres. Le spécificateur `e` formate la lettre `e` en minuscule, et le spécificateur `E` formate en majuscule.

Nombre de chiffres

Pour formater un nombre en notation scientifique, en utilisant un spécificateur de précision, on peut contrôler le nombre de chiffres significatifs de la mantisse :

```
print("{:.2e}".format(2030.4248))
```

```
2.03e+03
```

Signe + avec la méthode `format`

Espace et signe

On souhaite parfois obtenir un formatage de nombres comme ci-dessous :

```
a = -42
b =  2024
```

— Lignes 1-2 : les nombres 42 et 2024 sont alignés verticalement du côté gauche.

Plus précisément, pour des raisons d'alignement, on souhaite qu'un nombre négatif soit formaté comme on en a l'habitude (unique signe qui précède le premier chiffre) et qu'un nombre positif soit formaté sans signe mais précédé d'un unique espace.

Le code correspondant à l'exemple ci-dessus est :

```
1 modele="{} = {: }"
2
3 print(modele.format("a", -42))
4 print(modele.format("b", 2024))
```

— Ligne 1 : le spécificateur consiste seulement en un espace.

Pour placer un espace à la place du signe + devant un nombre, on utilise un spécificateur constitué d'un seul caractère espace qui est donc placé matériellement après le séparateur : et avant l'accolade fermante.

Signe + obligatoire

Parfois, on souhaite que le formatage d'un nombre montre toujours le signe + ou - (ce dernier étant obligatoire si le nombre est négatif). Pour cela, il suffit d'utiliser le spécificateur +

```
1 modele="Rennes : {:+}°C\nMoscou : {:+}°C"
2 print(modele.format(42, -24))
```

```
3 Rennes : +42°C
4 Moscou : -24°C
```

Noter bien qu'un spécificateur de formatage est nécessaire, le code naïf suivant ne fonctionne pas :

```
1 print(+42)
```

```
2 42
```

— Ligne 1 : le signe + est bien présent

— Ligne 2 : le signe + est absent : en effet l'expression +42 est *évaluée* avant d'être affichée.

Séparateur de milliers avec la méthode `format`

Pour des raisons de lisibilité, l'usage veut qu'on formate des nombres longs en séparant les chiffres par groupes de trois. Un spécificateur permet cette séparation :

```
1 p = -6542120359.4521263
2 s = f"prix = {p:,} euros"
3 print(s)
4
5 p = 421
6 s = f"prix = {p:,} euros"
```

```
7 print(s)
```

```
8 prix = -6,542,120,359.4521265 euros
9 prix = 421 euros
```

Le spécificateur consiste juste en une virgule :

,

Le séparateur est le séparateur utilisé dans le monde anglo-saxon, à savoir une virgule.
La partie décimale n'est pas formatée par groupe de 3 chiffres et il ne semble pas qu'un spécificateur le permette.
Si le nombre a moins de trois chiffres, aucune action de formatage spécifique n'est réalisée.

Respect des conventions linguistiques

Par défaut, le séparateur des chiffres d'un nombre en bloc de trois chiffres est la virgule. L'usage dans les pays francophones est d'utiliser plutôt un espace. Le spécificateur `n` permet de formater en respectant cet usage. Cela suppose que Python soit installé sur une machine où la locale est francophone.

```
1 import locale
2 ma_locale = locale.setlocale(locale.LC_ALL, '')
3 print(ma_locale)
4
5 print("{:n}".format(2099.236569633))
```

```
6 fr_FR.UTF-8
7 2099,24
```

— Lignes 2-3 : chargement de la locale pour toutes les catégories et affichage (ligne 6) de la locale utilisable.
— Ligne 5 : le spécificateur consiste juste en `n`
— Ligne 7 : on observe la séparation par un espace des deux blocs de chiffres.

La partie décimale a été tronquée à deux chiffres. Même si on tente de garder plus de chiffres significatifs, la partie décimale n'est pas découpée par groupes de 3 chiffres :

```
1 import locale
2 ma_locale = locale.setlocale(locale.LC_ALL, '')
3
4 print("{:.10n}".format(2099.236569633))
```

```
5 2099,23657
```

Pourcentage avec la méthode `format`

Le spécificateur `%` permet de formater un nombre en pourcentage.

```
1 s = f"{3/4:%}"
2 print(s)
```

```
75.000000%
```

— On notera que le séparateur décimal est ici un point et non une virgule.

Plus précisément, si x est le nombre à formater, le formatage consiste :

— à générer en flottant le pourcentage $100x$
— à adjoindre le symbole %.

L'intérêt semble limité.

4 Avancé, spécialisé ou secondaire

Contexte d'utilisation de la méthode `format`

Une chaîne formatée est une chaîne comme les autres. Elle n'est pas exclusivement destinée à être affichée, bien que ce sera assez fréquent en pratique. Une chaîne formatée possède toutes les propriétés d'une chaîne, par exemple une longueur :

```
modele = "Son nom est {} et il a {} ans."

naissance= 2000
aujourdhui= 2030

a = modele.format("Arthur", aujourdhui - naissance)
print(len(a))
print(a)
```

```
34
Son nom est Arthur et il a 30 ans.
```

Exemple typique

Les chaînes formatées servent souvent à remplacer des valeurs dans une chaîne-type puis à effectuer l'affichage (en général avec print). Par exemple :

```
modele = "{0} = {2}, {1} = {3} => {0} + {1} = {4}"

x=42
y=18
print(modele.format("x", "y", x, y, x+y))
print(modele.format("too", "titi", x, y, x+y))
```

```
x = 42, y = 18 => x + y = 60
too = 42, titi = 18 => too + titi = 60
```

Un affichage pourrait être produit sans utilisation de chaînes formatées mais le code serait moins lisible :

```
modele = "{0} = {2}, {1} = {3} => {0} + {1} = {4}"

x=42
```

```
4 y=18
5 print("x =", x, ", y =", y, "=> x + y =", x+y)
```

```
6 x = 42 , y = 18 => x + y = 60
```

En outre, la virgule juste après la valeur de x, n'est pas placée comme attendue (il y a un blanc de trop).

Exemple plus complexes

Les chaînes formatées servent aussi à construire des chaînes complexes, représentant par exemple des alignements particuliers, pas forcément dans un but immédiat d'affichage. Ainsi, pour réaliser une table de multiplication comme ceci,

```
 ------------------------------------------------------------------------
| 1 x  1 =   1| 2 x  1 =   2| 3 x  1 =   3| 4 x  1 =   4| 5 x  1 =   5|
| 1 x  2 =   2| 2 x  2 =   4| 3 x  2 =   6| 4 x  2 =   8| 5 x  2 =  10|
| 1 x  3 =   3| 2 x  3 =   6| 3 x  3 =   9| 4 x  3 =  12| 5 x  3 =  15|
| 1 x  4 =   4| 2 x  4 =   8| 3 x  4 =  12| 4 x  4 =  16| 5 x  4 =  20|
| 1 x  5 =   5| 2 x  5 =  10| 3 x  5 =  15| 4 x  5 =  20| 5 x  5 =  25|
| 1 x  6 =   6| 2 x  6 =  12| 3 x  6 =  18| 4 x  6 =  24| 5 x  6 =  30|
| 1 x  7 =   7| 2 x  7 =  14| 3 x  7 =  21| 4 x  7 =  28| 5 x  7 =  35|
| 1 x  8 =   8| 2 x  8 =  16| 3 x  8 =  24| 4 x  8 =  32| 5 x  8 =  40|
| 1 x  9 =   9| 2 x  9 =  18| 3 x  9 =  27| 4 x  9 =  36| 5 x  9 =  45|
| 1 x 10 =  10| 2 x 10 =  20| 3 x 10 =  30| 4 x 10 =  40| 5 x 10 =  50|
------------------------------------------------------------------------
| 6 x  1 =   6| 7 x  1 =   7| 8 x  1 =   8| 9 x  1 =   9|10 x  1 =  10|
| 6 x  2 =  12| 7 x  2 =  14| 8 x  2 =  16| 9 x  2 =  18|10 x  2 =  20|
| 6 x  3 =  18| 7 x  3 =  21| 8 x  3 =  24| 9 x  3 =  27|10 x  3 =  30|
| 6 x  4 =  24| 7 x  4 =  28| 8 x  4 =  32| 9 x  4 =  36|10 x  4 =  40|
| 6 x  5 =  30| 7 x  5 =  35| 8 x  5 =  40| 9 x  5 =  45|10 x  5 =  50|
| 6 x  6 =  36| 7 x  6 =  42| 8 x  6 =  48| 9 x  6 =  54|10 x  6 =  60|
| 6 x  7 =  42| 7 x  7 =  49| 8 x  7 =  56| 9 x  7 =  63|10 x  7 =  70|
| 6 x  8 =  48| 7 x  8 =  56| 8 x  8 =  64| 9 x  8 =  72|10 x  8 =  80|
| 6 x  9 =  54| 7 x  9 =  63| 8 x  9 =  72| 9 x  9 =  81|10 x  9 =  90|
| 6 x 10 =  60| 7 x 10 =  70| 8 x 10 =  80| 9 x 10 =  90|10 x 10 = 100|
 ------------------------------------------------------------------------
```

Typiquement, le code va utiliser un gabarit qui représente l'élément formaté élémentaire qui sera répété pour constituer la totalité de la table, ici tel un élément serait représenté par exemple par `|   6 x  1 =   6` qui correspond au gabarit Python `"{0}|{1}{2:2} x {3:2} = {4:3}{5}"`.

La table de multiplication est placée dans une grande chaîne au fur et à mesure. Ensuite, cette chaîne est affichée en une seule instruction.

Exemples d'utilisation de chaînes formatées

Voici une liste de situations typiques d'utilisation de chaînes formatées :

— date, par exemple mardi 19/01/2038

— heure, par exemple `03:14:42`

— numéro de téléphone : `08-81-42-99-00`

— grilles en tous genres :

  - sudoku,
  - tables de multiplication
  - conjugaison de verbes
  - calendrier
  - tabulation des valeurs d'une fonction
- Opération comme :

```
   45
+ 253
+ 940
-----
 1238
```

Formatage par défaut d'un objet

Jusqu'à présent les remplacements effectués dans un gabarit portent sur des chaînes de caractères ou sur des nombres entiers ou flottants :

```
1 modele = "{} {:02}/{:02}/{}"
2 a=modele.format("dimanche", 1, 9, 2030)
3 print(a)
```

```
4 dimanche 01/09/2030
```

- Les objets formatés (qui correspondent aux accolades) sont ici soit une chaîne ("dimanche") soit des entiers comme 2030.

Mais que se passe-t-il si l'argument correspondant à un champ de remplacement est d'un autre type ? Voyons un exemple :

```
1 class Point:
2     def __init__(self, x, y, name):
3         self.x = x
4         self.y = y
5         self.name = name
6
7 a = Point(5, 8, "A")
8
9 gabarit = "Le point {0.name} ... {0}".format(a)
10 print(gabarit)
11 print(str(a))
```

```
12 Le point A ... <__main__.Point object at 0xb71842ac>
13 <__main__.Point object at 0xb71842ac>
```

Lorsqu'un argument, disons A, de la méthode `format` doit être remplacé dans un gabarit, Python recherche une méthode `A.__format__`. Si aucune telle méthode n'a été explicitement définie alors le remplacement est effectué par `str(A)`. Par défaut, `str(A)` est une chaîne peu lisible pour un utilisateur humain, cf. ligne 12 ou 13.

D'autre part, comme le remplacement est effectué par `str(A)`, il est possible de placer dans le champ de remplacement une spécification de formatage valide pour une chaîne. Voici un exemple artificiel :

```
1 class Point:
2     def __init__(self, x, y, name):
3         self.x = x
4         self.y = y
5         self.name = name
6 
7 a = Point(5, 8, "A")
8 
9 gabarit = "Le point {0.name} ... {0:X^60}".format(a)
10 print(gabarit)
```

```
11 Le point A ... XXXXXXXXXXX<__main__.Point object at 0xb715528c>XXXXXXXXXXX
```

La spécification de formatage `X^60` (ligne 9) a entouré de lettres majuscules `X` le champ de remplacement dans un champ de largeur 60. Cette utilisation est possible mais ici de peu d'intérêt.

Formatage personnalisé d'un objet

Montrons sur un exemple comment on peut définir une méthode spécifique de formatage aux objets d'une classe :

```
1 class Point:
2     def __init__(self, x, y, name):
3         self.x = x
4         self.y = y
5         self.name = name
6 
7     def __format__(self, format_spec):
8         if format_spec == "x":
9             return "x_{0.name} = {0.x}".format(self)
10         if format_spec == "y":
11             return "y_{0.name} = {0.y}".format(self)
12         return "{0.name} =  ({0.x}, {0.y})".format(self)
13 
14 a = Point(9, 8, "A")
15 print("Le point {0}".format(a))
16 print("L'abscisse de {0} vaut {1:x}".format(a.name, a))
```

```
17 Le point A =  (9, 8)
18 L'abscisse de A vaut x_A = 9
```

On définit (lignes 7-12) dans la classe `Point` une méthode `__format__`. Lorsqu'un champ de remplacement se réfère à un objet de type `Point` (lignes 14-16), le remplacement est effectué par ce que renvoie la méthode `__format__`. Un champ de remplacement peut même disposer après le symbole : comme dans le champ `{0.x}` d'un spécificateur de formatage sous la forme d'une chaîne `format_spec` (ligne 7) et qui est passé en argument (le x à la ligne 16) à la méthode

`__format__`.

Remplacement par un attribut d'argument

Un champ de remplacement peut référer un attribut d'un des arguments de la méthode `format`. Ainsi, soit le code :

```
class Point:
    def __init__(self, x, y, name):
        self.x = x
        self.y = y
        self.name = name

a = Point(5, 8, "A")
b = Point(2, -1, "B")
modele = "x_{0.name} = {0.x}\ny_{1.name} = {1.y}"
s = modele.format(a, b)

print(s)
```

```
x_A = 5
y_B = -1
```

Tout objet de la classe `Point` possède trois attributs : `x`, `y` et `name`. Dans un gabarit, tout champ de remplacement (entre accolades) peut référer à l'un de ces attributs. Par exemple, `1.y` réfère à l'attribut `y` du 2e argument de `format`.

Si l'ordre des champs de remplacement correspond à l'ordre des arguments, la présence des numéros n'est pas indispensable :

```
class Point:
    def __init__(self, x, y, name):
        self.x = x
        self.y = y
        self.name = name

a = Point(5, 8, "A")
b = Point(2, -1, "B")

s = "{.name} et {.name}".format(a, b)

print(s)
```

```
A et B
```

Positions après décompression d'une liste

Les arguments de la méthode `format` peuvent être récupérés après une décompression de liste avec la syntaxe `*L` où `L` est une liste, comme pour n'importe quelle fonction admettant une liste à décompresser en argument. Tout numéro d'argument dans un nom de remplacement se réfère

aux numéros obtenus après décompression.

```
1 modele = "le {0} de couleurs {2}, {1} et {2}"
2 couleurs=["rouge", "jaune", "vert"]
3 a = modele.format("drapeau", *couleurs)
4 print(a)
```

```
5 le drapeau de couleurs jaune, rouge et jaune
```

— Ligne 1 : le gabarit se réfère aux arguments numéros 0, 1 et 2.

— Ligne 3 : après décompression de l'argument `couleurs`, c'est comme si la liste des arguments de `format` était : `"drapeau"`, `"rouge"`, `"jaune"`, `"vert"`. La substitution se poursuit comme avec des arguments positionnels.

Nombre d'accolades

Considérons un gabarit Python de formatage, par exemple

```
modele = "une {1}, une {2} et une {0}"
```

Si n est le plus grand numéro apparaissant entre des accolades (dans l'exemple, c'est 2), il doit y avoir au moins $n + 1$ arguments dans l'appel de la méthode `format`, par exemple

```
modele.format("pomme", "poire", "prune")
```

Sinon, une exception est levée :

```
modele = "une {1}, une {2} et une {0}"
ch=modele.format("pomme", "poire")
print(ch)
```

```
Traceback (most recent call last):
  File "a.py", line 2, in <module>
    ch=modele.format("pomme", "poire")
IndexError: tuple index out of range
```

D'autre part, si une paire d'accolades est vide dans le gabarit, toutes les paires d'accolades suivantes doivent être également vides sinon on reçoit un message d'erreur :

```
1 print("{}{1}".format(42,17))
```

```
2 Traceback (most recent call last):
3   File "vide_et_non_vide.py", line 1, in <module>
4     print("{}{1}".format(42,17))
5 ValueError: cannot switch from automatic field numbering
6 to manual field specification
```

Enfin, bien que semblant avoir peu d'intérêt, il n'est pas interdit que le gabarit ne reçoive aucune accolade et que pourtant `format` reçoive des arguments :

```
1 s = "19 janvier 2038".format(42)
2 print(s)
```

```
19 janvier 2038
```

Bien sûr, cela n'a aucune action sur le gabarit.

Cela peut être utile lorsque la méthode `format` est appliquée à des chaînes générées automatiquement.

Spécification de formatage et champ anonyme

Le signe : entre des accolades marque la présence d'une spécification de formatage. À gauche de : figure en principe une référence à l'argument de la méthode `format` qui va servir de remplacement. La référence à l'argument n'est pas nécessaire si les remplacements s'effectuent suivant l'ordre où les arguments apparaissent écrits dans l'appel :

```
1 modele = "{} {:02}/{:02}/{}"
2 a=modele.format("dimanche", 1, 9, 2030)
3 print(a)
```

```
4 dimanche 01/09/2030
```

— Ligne 1 : tous les champs sont anonymes puisque rien ne précède le séparateur :

— ligne 2 : les remplacements sont faits dans l'ordre des arguments de la méthode `format` : `"dimanche"` puis `1` puis `9` et `2030`.

Cumuls de spécificateurs avec la méthode `format`

Une spécification de formatage peut contenir plusieurs spécificateurs successifs. Voici un exemple commenté :

```
1 modele = "1000 * pi = {:~^ 20,.2f}"
2 print(modele.format(1000 * 22/7))
3 print(modele.format(1000 * 333/106))
```

```
4 1000 * pi = ~~~~~ 3,142.86~~~~~~
5 1000 * pi = ~~~~~ 3,141.51~~~~~~
```

Dans l'exemple, il n'y a qu'un champ de remplacement. Sa spécification est :`~^ 20,.2f` qui est composée de six spécificateurs :

— un spécificateur de remplissage centré (cf. le spécificateur ^) par des caractères ~

— l'espace (résultat : place un unique espace avant un nombre positif)

— un champ de largeur 20

— le séparateur de milliers (spécificateur virgule)

— précision (2 décimales)

— type flottant

L'ordre des spécificateurs n'est pas arbitraire ; de gauche à droite, on trouve des spécificateurs

— d'alignement (<, ^, > et plus rarement =), précédé d'un **unique** caractère de remplissage

— de signe (+ ou espace)

— de préfixe de conversion dans certaines bases (`#`)
— de remplissage par des zéros (`0`)
— de largeur de champ (un entier)
— de groupement par blocs de 3 chiffres de la partie entière (une virgule)
— de précision (de la forme `.n` où `n` est un entier)
— de type (par exemple `f`, `x`, `n` ou `%`)

Les spécificateurs sont optionnels, certains peuvent être omis (comme le type). Certains spécificateurs ne s'appliquent qu'à des nombres, voire des nombres d'un certain type. La plupart du temps, le bon sens permet de trancher.

Champs emboîtés avec la méthode `format`

Considérons le gabarit suivant :

```
gabarit= "__{0:.>20}____{1:-^10}__"
```

Voici un exemple d'utilisation de ce gabarit :

```
gabarit= "__{0:.>20}____{1:-^10}__"
print(gabarit.format(2030, 2038))
```

```
__.................2030____---2038---__
```

Décrivons ce gabarit. Il contient deux champs de remplacement, à savoir `{0:.>20}__` et `{1:-^10}` ; ces champs sont **séparés**. Chacun de ces champs est composé

— de la référence à l'argument de la méthode `format` (0 puis 1)
— du séparateur :
— d'une spécification de formatage (`.>20` et `-^10`).

A priori, des champs de remplacement sont toujours **séparés**. Cependant, il est possible, sous certaines conditions, de placer des champs de remplacement dans une spécification de format :

```
gabarit0= "__{0:{2}}____{1:{3}}__"
print(gabarit0.format(2030, 2038, ".>20", "-^10"))
```

— Ligne 1 : le premier champ de remplacement `{0:{2}}` a une spécification `{2}` qui contient un autre champ de remplacement (cf. les accolades). On parle alors de *champs emboîtés*.

Action d'un champ emboîté

Exécutons l'exemple précédent :

```
1 gabarit0= "__{0:{2}}____{1:{3}}__"
2 print(gabarit0.format(2030, 2038, ".>20", "-^10"))
```

```
3 __.................2030____---2038---__
```

On observe que le résultat est le même que la première utilisation de `gabarit` ci-dessus. Le processus d'exécution est le suivant :

— tout champ, emboîté ou non, a un nom qui réfère à l'argument de la méthode `format`. Par exemple, le champ {2} ci-dessus réfère à l'argument `".>20"`, l'argument n°2 de `format`.
— les champs de remplacement emboîtés sont d'abord remplacés par les arguments correspondants ce qui produit un gabarit implicite, ici `"__{0:.>20}____{1:-^10}__"`
— suivant le processus habituel, les champs non emboîtés sont remplacés par les arguments correspondants.

Codes équivalents

Le code ci-dessus est équivalent à :

```
1 gabarit0="__{0:{2}}____{1:{3}}__"
2 gabarit= gabarit0.replace("{2}",".>20").replace("{3}","-^10")
3 print(gabarit)
4 print(gabarit.format(2030, 2038))
```

```
5 __{0:.>20}____{1:-^10}__
6 __.................2030____---2038---__
```

Intérêt

L'intérêt des champs emboîtés est que cela permet de générer des spécificateurs de format dynamiques, par exemple avec des champs de largeur paramétrable.

Un exemple typique, lu dans ce message de forum, permet de paramétrer le nombre de décimales d'un flottant :

```
1 modele="{:.{}f}"
2
3 for prec in [0, 3, 6, 9]:
4     print(modele.format(22/7,prec))
```

```
5 3
6 3.143
7 3.142857
8 3.142857143
```

— Ligne 1 : un même gabarit va permettre de générer les chaînes de largeur variable (lignes 5-8)
— Ligne 4 : on formate successivement `"{22/7:.0f}"`, puis `"{22/7:.3f}"`, etc.

On peut faire des formatages plus complexes :

```
1 modele = "__{0:{1}{2}{3}{4}{5}}__"
2 x=2038
3 print(modele.format(x, '?', '>', ' ', 42,','))
4 print(modele.format(x, '~', '^', '+', 20,'e'))
```

```
5 __???????????????????????????????????? 2,038__
6 __~~~+2.038000e+03~~~~__
```

Drapeau de conversion avec la méthode `format`

Soit le code :

```
print("{0}{1:.^20}".format("Hello", "Rose"))
```

```
Hello........Rose........
```

La chaîne `"{0}{1:.^20}"` comporte 2 champs de remplacement. Examinons par exemple le 2e champ : `{1:.^20}`. Il comporte deux composantes :

— un numéro d'argument : 1

— un spécificateur de formatage : `.^20`

Cette syntaxe peut-être enrichie. Pour que mes explications ne soient pas trop abstraites, voici la nouveauté sur un exemple :

```
print("{0}{1!a:.^20}".format("Hello", "Rose"))
```

qui affiche

```
Hello.......'Rose'.......
```

Un champ de remplacement peut disposer d'une troisième composante : un drapeau de conversion. Ce drapeau est placé en 2e position, entre le représentant de l'argument et avant les deux-points qui précèdent le spécificateur de formatage, cf. ci-dessus, les deux caractères `!a`.

Un drapeau peut prendre une des trois formes suivantes :

`!s` ou `!r` ou `!a`

Voici un exemple :

```
print("{0}{1!s:.^20}".format("Hello", "Rose"))
print("{0}{1!r:.^20}".format("Hello", "Rose"))
print("{0}{1!a:.^20}".format("Hello", "Rosé"))
```

```
Hello........Rose........
Hello.......'Rose'.......
Hello.....'Ros\xe9'......
```

Lorqu'un objet X à remplacer est associé à un drapeau de conversion, l'objet X est converti en une chaîne S et c'est la chaîne qui fait l'objet du remplacement. La chaîne S dépend du drapeau :

— si le drapeau est `!s` alors S est `str(X)`

— si le drapeau est `!r` alors S est `repr(X)`

— si le drapeau est `!a` alors S est `ascii(X)` où `ascii` est une fonction de la bibliothèque standard (par exemple, en français, ça réécrit avec un échappement s'il y a un accent).

Dans les cas usuels, le drapeau `!s` n'a pas d'action :

```
print("{0}{1!s:.^20}".format("Hello", "Rose"))
print("{0}{1:.^20}".format("Hello", "Rose"))
```

```
Hello........Rose........
Hello........Rose........
```

Il se peut, bien sûr, que le drapeau soit seul dans le champ de remplacement :

```
1 print("C'est {!a}".format("codé"))
```

```
2 C'est 'cod\xe9'
```

Exemple avec une liste

Montrons un exemple où on veut placer des listes dans des champs de largeur prédéfinie afin d'obtenir un alignement :

```
[5, 12, 8]    est une liste
[50, 120, 80] est une liste
```

Si on essaye d'obtenir le résultat via le code suivant :

```
1 L = [5, 12, 8]
2 M = [50, 120, 80]
3
4 s = """{:13} est une liste
5 {:13} est une liste"""
6
7 print(s.format(L, M))
```

```
8     print(s.format(L, M))
9 TypeError: unsupported format string passed to list.__format__
```

on obtiendra une erreur. En effet, on essaye d'appliquer une spécification de largeur (lignes 4 et 5) à une liste qui ne la définit pas.

En revanche, si on place un drapeau de conversion, la spécification s'appliquera à une chaîne ce qui a un sens cette fois :

```
L = [5, 12, 8]
M = [50, 120, 80]

s = """{!s:13} est une liste
{!s:13} est une liste"""

print(s.format(L, M))
```

```
[5, 12, 8]    est une liste
[50, 120, 80] est une liste
```

Intérêt du drapeau de conversion

Soit un appel de la méthode `format` de la forme `gabarit.format` où gabarit est une chaîne. Si gabarit admet un champ de remplacement par un objet `X` alors le champ est remplacé par le retour d'un appel de `X.__format__`. Parfois ce comportement par défaut n'est pas souhaité et pour l'éviter, on peut placer un drapeau dans le champ de remplacement.

Par exemple :

```
class Point:
    def __init__(self, x, y, name):
        self.x = x
        self.y = y
        self.name = name

    def __format__(self, format_spec):
        if format_spec == "x":
            return "x_{0.name} = {0.x}".format(self)
        if format_spec == "y":
            return "y_{0.name} = {0.y}".format(self)
        return "{0.name} =  ({0.x}, {0.y})".format(self)

a = Point(9, 8, "A")
print("Le point {0}".format(a))
print(str(a))
print("Le point {0!s}".format(a))
```

```
Le point A =  (9, 8)
<__main__.Point object at 0xb713d28c>
Le point <__main__.Point object at 0xb713d28c>
```

— Pas de drapeau : la méthode `__format__` de Point est appelée

— Le drapeau `!s` empêche l'appel de la méthode `__format__` et le champ est remplacé par `str(a)`

Troncature avec la méthode `format`

On peut vouloir qu'un champ de remplacement reçoive une chaîne dont on ne garde qu'un certain nombre des premiers caractères :

```
semaine=['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche']
for i in range(7):
    print("{0:.3} {1}/10/2030".format(semaine[i], i+1))
```

```
lun 1/10/2030
mar 2/10/2030
mer 3/10/2030
jeu 4/10/2030
ven 5/10/2030
sam 6/10/2030
dim 7/10/2030
```

— Lignes 1-2 : seuls les 3 premiers caractères du jour de la semaine seront utilisés pour remplacer le champ.

Cette utilisation de troncature semble être assez rare.

La même syntaxe sert à contrôler le nombre de décimales d'un flottant.

Concaténation avec la méthode `format`

Le formatage de chaînes permet accessoirement de concaténer ou de joindre des chaînes

```
modele = "{}{}{}"
print(modele.format("Uni","Ver","Sel"))

modele = "{}-{}-{}"
print(modele.format("Uni","Ver","Sel"))
```

```
UniVerSel
Uni-Ver-Sel
```

Documentation sur la méthode `format`

Voici les liens vers la documentation officielle sur le formatage avec la méthode `format` :

— le formatage avec la méthode format

— la méthode spéciale `__format__`

— des exemples de traduction de chaînes formatées de style classique en des chaînes formatées avec la méthode `format`.

Chapitre III

Chaînes formatées classiques

1 L'essentiel

Les chaînes formatées classiques

La méthode historique de formatage de chaînes en Python (qualifiée de méthode « classique » dans ce document) héritée du C et disponible en Java, reste utilisable mais la méthode `format` lui est explicitement préférée dans la documentation officielle. En outre, depuis l'apparition des f-chaînes, la méthode classique présente moins d'avantages en concision ou lisibilité.

Présentons justement la méthode classique. Soient les chaînes formatées suivantes :

```
Son nom est titi et il a 8 ans.
Son nom est Arthur et il a 30 ans.
Son nom est Alfred et il a 100 ans.
```

Voici le code Python permettant de produire les chaînes formatées ci-dessus :

```
1 modele = "Son nom est %s et il a %s ans."
2
3 a = modele %("titi", 8)
4 print(a)
5
6 b = modele %("Arthur", 30)
7 print(b)
8
9 print("Son nom est %s et il a %s ans." %("Alfred", 100))
```

```
10 Son nom est titi et il a 8 ans.
11 Son nom est Arthur et il a 30 ans.
12 Son nom est Alfred et il a 100 ans.
```

— Ligne 1 : le gabarit. C'est une chaîne Python. Les zones à remplacer sont marquées dans le gabarit par `%s` ; dans l'exemple, il y a deux zones de remplacement dans le gabarit.

— Ligne 3 : construction de la chaîne formatée (qui est affichée ligne 10). La syntaxe `%("titi", 8)` indique ce par quoi les zones de la forme `%s` vont être remplacées dans le gabarit. Le premier "

— Lignes 9 et 12 : noter que l'on n'est pas obligé de placer la gabarit dans une variable.

Description d'une chaîne formatée

On va décrire la syntaxe d'une chaîne formatée en se basant sur l'exemple suivant :

```
1 s = "Son nom est %s et il a %s ans." %("Alfred", 100)
2 print(s)
```

```
3 Son nom est Alfred et il a 100 ans.
```

L'expression complète

```
"Son nom est %s et il a %s ans." %("Alfred", 100)
```

est une *chaîne formatée*.

Elle se décompose en trois parties, de gauche à droite :

— le format

— l'opérateur %

— le tuple de remplacement

Le format

Le format se présente comme une simple chaîne Python. Dans l'exemple, il s'agit de la chaîne littérale

```
1 "Je m'appelle %s et j'ai %s ans"
```

Chaque valeur à remplacer dans cette chaîne (dans l'exemple le nom et l'âge) est signalée par le symbole % suivi d'une lettre particulière (ici s).

Le format représente un modèle (un gabarit) dans lequel seront effectuées des substitutions.

L'opérateur %

L'opérateur % se place après le format et avant l'objet contenant les valeurs de remplacement.

Cet opérateur est l'opérateur modulo de division entière mais dans ce contexte on l'appelle plutôt l'opérateur de *formatage de chaîne*, parfois aussi connu sous le nom d'opérateur d'*interpolation* de chaîne.

L'opérateur de formatage % est un opérateur comme un autre, au même titre que + ou *. Il peut être entouré d'autant d'espaces avant et après :

```
1 s = "Son nom est %s et il a %s ans."    %       ("Alfred", 100)
2 print(s)
```

```
3 Son nom est Alfred et il a 100 ans.
```

— Ligne 1 : on a placé des espaces avant et après l'opérateur %

Double usage du symbole %

On notera donc que le symbole % s'utilise dans des contextes différents :

— dans le format pour marquer le début des emplacements à remplacer ;

— dans la chaîne formatée pour signaler l'objet contenant les valeurs de remplacement correspondantes.

Dans le format, % est juste un marqueur interne et dans le deuxième cas, % est un opérateur du langage Python.

Spécificateur de formatage

Les endroits de remplacement dans le format sont signalés par le symbole % suivi d'un ou plusieurs caractères précisant **comment** le remplacement doit être fait.

Dans l'exemple ci-dessus, il s'agissait de %s. Ces groupes placés dans le format et commençant par le symbole % sont appelés des *spécificateurs de formatage*. Ils peuvent être plus complexes que ceux de l'exemple, comme %(toto) 010.2f.

Type de conversion

Un spécificateur de formatage commence par % et se termine par une lettre appelée le *type de conversion*.

La lettre indique quelle conversion la valeur de remplacement doit subir avant le remplacement. Par exemple, la lettre s fait référence au type *string* (chaîne), la lettre f fait référence au type *float* (flottant).

Dans l'exemple

```
1 s = "Son nom est %s et il a %s ans." %("Alfred", 100)
```

le deuxième spécificateur %s dans le format précise que la valeur de remplacement, ici 100, doit être convertie en une chaîne, plus précisément, la chaîne qui représente l'entier 100 en base 10.

Pour un autre type de conversion, le formatage aurait pu être différent. Par exemple, au lieu du type s, on peut utiliser un type f qui convertit la valeur en flottant :

```
1 s = "Son nom est %s et il a %f ans" %("Alfred", 100)
2 print(s)
```

```
3 Son nom est Alfred et il a 100.000000 ans
```

— L'entier 100 est converti en flottant avant remplacement

— Le remplacement se fait suivant un formatage flottant (séparateur décimal et 6 chiffres derrière).

Le formatage en *string* (avec s donc) convient à beaucoup d'utilisations usuelles.

Valeurs de remplacement

Dans l'exemple, les valeurs de remplacement sont au nombre de deux : la chaîne *Alfred* et l'entier 100. Les valeurs de remplacement apparaissent séparées par des virgules et entourées de parenthèses. La paire de parenthèses représente en fait un tuple. Le tuple qui suit l'opérateur % doit contenir autant de valeurs qu'il y a de spécificateurs de formatage présents dans le format.

Les valeurs de remplacement doivent figurer dans un **tuple** et **pas dans une liste**.

Tuple de remplacement

Soit le code suivant :

```
1 gabarit = "Son nom est %s et il a %s ans."
2
```

```
3 a = gabarit %("titi", 8)
4 print(a)
```

```
5 Son nom est titi et il a 8 ans.
```

Les valeurs de remplacement dans le gabarit apparaissent après l'opérateur % et sont placées dans un tuple. Ainsi le code peut-être récrit :

```
1 gabarit = "Son nom est %s et il a %s ans."
2 t = ("titi", 8)
3 a = gabarit %t
4 print(a)
```

Il est toutefois rare que l'on procède ainsi en pratique.

Cas d'une unique valeur de remplacement

Lorsqu'il y a une seule valeur à remplacer, la syntaxe d'une chaîne formatée peut être simplifiée en ignorant la notation de tuple pour le remplacement :

```
1 print("Son nom est %s !!!" %"toto")
2
3 x = "titi"
4 print("Son nom est %s !!!" %x)
5
6 gabarit = "Son nom est %s !!!"
7 x = "titi"
8 print(gabarit %x)
```

```
9 Son nom est toto !!!
10 Son nom est titi !!!
11 Son nom est titi !!!
```

— 3 variantes de la même syntaxe sont présentées ; dans chaque cas, un seul remplacement est fait et donc l'opérateur % n'est pas suivi d'une parenthèse.

Priorité de l'opérateur de remplacement

Observons le code suivant au comportement inattendu :

```
1 print("Toto a %s ans" %20)
2 print("Toto a %s ans" %10*2)
```

```
3 Toto a 20 ans
4 Toto a 10 ansToto a 10 ans
```

— Ligne 1 : %s a été remplacé par 10

— Ligne 2 : bien que `10 * 2` vaille 20, %s n'a pas été remplacé par 20.

Lorsque la valeur de remplacement fait l'objet d'une opération (ici une multiplication), il peut y avoir un conflit de priorité ou d'association avec l'opérateur % de formatage.

Dans l'exemple ci-dessus, le comportement inattendu est en fait équivalent à :

```
1 s = "Toto a %s ans" %10
2 print(s*2)
```

```
3 Toto a 10 ansToto a 10 ans
```

Par précaution dans ce type de situation, on entourera de parenthèses la valeur à remplacer :

```
1 print("Toto a %s ans" %(10*2))
```

```
2 Toto a 20 ans
```

Priorité de %

La priorité de l'opérateur % de formatatge de chaîne est la même que celle de l'opérateur arithmétique %.

Les opérateurs % et * ayant même priorité et la syntaxe du langage Python prévoyant que ces opérateurs s'associent de la gauche vers la droite, il en résulte que la chaîne formatée `mon_format % v * 2` s'interprète `(mon_format % v) * 2`.

En fonction des situations, on placera ou non des parenthèses pour changer les priorités. Voici quelques exemples :

```
1 t = ("mardi", 19)
2 u = ("janvier", 2038)
3
4 z = "%s %s %s %s"%(t+u)
5 print(z)
6
7 z = "%s %s "%t + "%s %s"%u
8 print(z)
```

```
9 mardi 19 janvier 2038
10 mardi 19 janvier 2038
```

Autre exemple :

```
1 x = 10
2 y = 42
3 z= "température : %s" %-max(x, y)
4
5 print(z)
```

```
6 température : -42
```

Options de formatage

Un formatage avec une chaîne formatée peut être affiné à l'aide de nombreuses options. Ces options sont plus ou moins utiles et plus ou moins complexes.

Pour les plus importantes, elles permettent, par exemple, de

— placer la valeur à formater dans un « champ » avec des espaces blancs à gauche ou à droite de la valeur,

— placer des zéros avant un nombre entier ou décimal,

— pour un nombre flottant, contrôler le nombre de décimales à placer dans la chaîne de remplacement

Dans le but de montrer la syntaxe des options, voici un exemple artificiellement complexe de chaîne formatée avec des options :

```
1 dico = {"nom":"Alfred", "age": 100}
2 s = "Son nom est %(nom)s et il a %(age)0 8.3f ans" %dico
3 print(s)
```

```
4 Son nom est Alfred et il a  100.000 ans
```

Les options de formatage se lisent (dans l'exemple, ligne 2) dans les spécificateurs de formatage entre le signe `%` et la dernière lettre du spécificateur (et qui désigne le type de conversion). Dans l'exemple, le spécificateur de formatage `%(age)0 8.3f` contient un certain nombre d'options (5 très précisément), que l'on peut extraire de la suite `(age)0 8.3`