14

Itérateurs, générateurs et coroutines

L'itération est un concept fondamental en algorithmique et dans les langages de programmation qui décrit la répétition d'une action. La formulation la plus simple de l'itération pour les programmeurs est la boucle, formulée par le mot-clé `for` ou `while`. En programmation fonctionnelle, l'itération est souvent exprimée par des appels récursifs à des fonctions.

L'itération peut également être vue comme une abstraction, un *service générique fourni par des structures itérables*. Ces structures sont alors capables de fournir des éléments un par un, sans avoir à les charger intégralement en mémoire, ce qui est souvent impossible pour de gros traitements de données. Toutes les structures de collection que nous avons abordées précédemment (☞ p. 49, § **4**) sont itérables. Il est alors possible :

- de les parcourir par une boucle `for`,
- de construire de nouvelles structures en *consommant* les anciennes (en passant une structure itérable à la fonction `list()` par exemple),
- de les manipuler par compréhension, (☞ p. 13, § **1.5**)
- de les déballer (*unpacking*).

Depuis Python 3, le mot-clé `range` ne renvoie pas de liste, mais un objet de type `range`.

```
>>> range(10)
range(0, 10)
>>> type(range(10))  # ceci n'est pas une liste
range
```

Il est alors possible d'itérer sur un `range`, que ce soit avec une boucle `for` ou avec des constructeurs d'autres structures conteneurs, comme les listes.

```
>>> for i in range(10):
...     print(i, end=" ")
0 1 2 3 4 5 6 7 8 9
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

14.1. Les générateurs

L'écriture en compréhension permet de produire de nouvelles structures itérables, pour les accumuler dans des nouvelles structures de collection ou pour les réduire. La notation en compréhension permet notamment d'appliquer les schémas `map` et `filter` (☞ p. 158, § 12.3) et améliore la performance par rapport à une construction à base de boucle et de `list.append()`.

```
%%timeit
nouveau = [2 * x for x in range(1000000)]
# 115 ms ± 4.96 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

%%timeit
nouveau = []
for x in range(1000000):
    nouveau.append(2*x)
# 165 ms ± 5.63 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Cette notation peut être parenthésée avant d'être *consommée*. L'objet produit est alors un *générateur*. On peut itérer ou construire une nouvelle collection à partir d'un générateur mais, une fois ce générateur utilisé, ou *consommé*, il n'est pas possible de le redémarrer.

```
>>> g = (str(i) for i in range(10))
>>> type(g)
generator
>>> list(g)
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> list(g)  # cette fois-ci, c'est vide!
[]
```

Attention !

Les deux expressions suivantes ne sont pas équivalentes : les parenthèses ont leur importance dans la définition du générateur.

```
>>> [str(i) for i in range(10)]
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> [(str(i) for i in range(10))]
[<generator object <genexpr> at 0x7ffe0bec2cf0>]
```

Au-delà des constructeurs de collection de base, d'autres fonctions natives manipulent des générateurs et des structures itérables. La fonction `sorted()` (☞ p. 23, § 2.1) construit une liste triée, la fonction `max()` renvoie l'élément maximal d'une structure itérable pourvu que les éléments renvoyés un par un soient comparables :

```
>>> list(i * (-1) ** (i) for i in range(10))
[0, -1, 2, -3, 4, -5, 6, -7, 8, -9]
>>> sorted(i * (-1) ** (i) for i in range(10))
[-9, -7, -5, -3, -1, 0, 2, 4, 6, 8]
>>> max(i * (-1) ** (i) for i in range(10))
8
```

14.2. Le mot-clé `yield`

Le mot-clé `yield` permet d'écrire des générateurs dans des fonctions. Une fonction qui contient le mot-clé `yield` renvoie un générateur.

Quand le programme rencontre le mot-clé `yield` :

1. il renvoie (*yields*) la valeur courante,
2. attend la prochaine itération dans la boucle,
3. reprend le programme là où il s'était arrêté.

Le générateur s'interrompt quand la fonction retourne.

```
>>> def exemple_yield() -> "generator":
...     yield 0
...
>>> type(exemple_yield())
generator
```

Comme décrit précédemment, les générateurs sont consommés pendant l'itération. À la fin d'une itération, il n'est plus possible de les redémarrer. L'avantage des fonctions avec le mot-clé `yield` est qu'elles retournent un nouveau générateur avec le même comportement à chaque fois qu'on les appelle.

```
>>> list(exemple_yield())
[0]
>>> list(exemple_yield())
[0]
```

Sur des générateurs écrits par compréhension, les deux syntaxes sont alors équivalentes :

```
def eq1() -> "generator":
    return (i for i in range(5))
```

```
def eq2() -> "generator":
    for i in range(5):
        yield i
```

Comme la suite de Fibonacci, la suite de Syracuse est un bon exemple pour illustrer le fonctionnement des fonctions qui renvoient des générateurs. La suite de Syracuse démarre sur un entier positif. À chaque itération, si le dernier entier est pair, on renvoie le résultat de sa division par 2 ; sinon on le multiplie par 3 avant d'ajouter 1.

Une conjecture prédit que cette suite converge systématiquement vers 1. Le chiffre 1 étant impair, les valeurs suivantes sont 4, puis 2, puis 1 : aussi l'usage est d'interrompre cette suite quand la valeur 1 est atteinte.

Les résultats intéressants pour cette suite peuvent être :

— la séquence complète de valeurs qui démarrent à l'entier n,
— la longueur de cette suite : combien faut-il d'itérations pour atteindre la valeur 1 ?
— la hauteur de cette suite : quelle est la valeur maximale atteinte par la suite avant de converger vers 1 ?

On pourrait alors imaginer une fonction qui renvoie la séquence complète, une autre fonction qui renvoie sa longueur et encore un autre qui renvoie sa hauteur. Pour factoriser cette spécification, la définition par générateur est confortable :

```
def syracuse(n: int) -> "generator":
    """Calcule la suite de Syracuse.

    >>> list(p for p in syracuse(28))
    [28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
    """
    yield n
    while n != 1:
        if n & 1 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        yield n
```

Un générateur n'a pas de longueur dans la définition de son interface. En effet, il existe des générateurs infinis qui n'ont pas de longueur (par exemple une instruction `yield` dans une boucle infinie).

Il existe une réduction (☞ p. 158, § **12.3**) qui permet de trouver la longueur d'une telle séquence : on ajoute 1 pour chaque nouvelle valeur retournée. Dans le code suivant, on peut utiliser la variable muette _ pour insister sur le fait que la valeur récupérée dans la structure itérable n'a pas d'importance :

```
def length(iterable):
    "Renvoie la longueur d'une structure itérable finie."
    return sum(1 for _ in iterable)

length(syracuse(58))  # 20
```

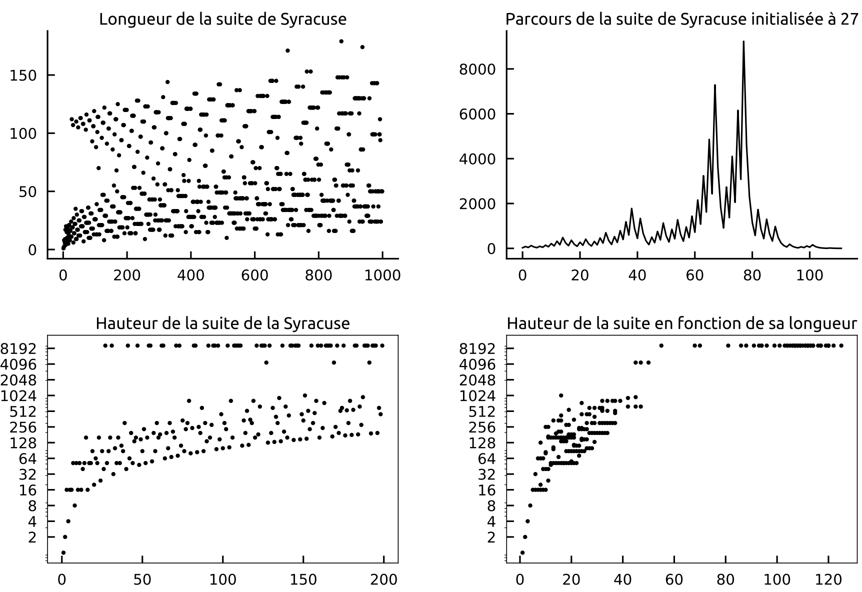

Figure 14.1 – Suite de Syracuse : longueur, parcours, hauteur et hauteur de la suite en fonction de sa longueur

On peut tracer (Figure 14.1) la longueur de la suite de Syracuse pour tous les entiers (de 1 à 1000 ici) pour faire ressortir des schémas surprenants. Dans l'expression suivante, en combinant application/filtrage sous forme de générateur en compréhension et réduction par l'opérateur `"".join()`, on peut raffiner l'affichage de la suite de Syracuse qui part de la valeur 27.

```
>>> " -> ".join(str(i) for i in syracuse(27))
'27 -> 82 -> 41 -> 124 -> 62 -> 31 -> 94 -> 47 -> 142 -> 71 -> 214 -> 107 ->
 322 -> 161 -> 484 -> 242 -> 121 -> 364 -> 182 -> 91 -> 274 -> 137 -> 412 ->
 206 -> 103 -> 310 -> 155 -> 466 -> 233 -> 700 -> 350 -> 175 -> 526 -> 263 ->
 790 -> 395 -> 1186 -> 593 -> 1780 -> 890 -> 445 -> 1336 -> 668 -> 334 -> 167 ->
 502 -> 251 -> 754 -> 377 -> 1132 -> 566 -> 283 -> 850 -> 425 -> 1276 -> 638 ->
 319 -> 958 -> 479 -> 1438 -> 719 -> 2158 -> 1079 -> 3238 -> 1619 -> 4858 ->
 2429 -> 7288 -> 3644 -> 1822 -> 911 -> 2734 -> 1367 -> 4102 -> 2051 -> 6154 ->
 3077 -> 9232 -> 4616 -> 2308 -> 1154 -> 577 -> 1732 -> 866 -> 433 -> 1300 ->
 650 -> 325 -> 976 -> 488 -> 244 -> 122 -> 61 -> 184 -> 92 -> 46 -> 23 -> 70 ->
 35 -> 106 -> 53 -> 160 -> 80 -> 40 -> 20 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1'
```

Sous forme de liste, on suit le parcours de la suite avant sa convergence :

```
list(syracuse(27))
```

On trouve sa hauteur (la valeur maximale prise) à l'aide de la fonction de réduction `max()` :

```
list(max(syracuse(i)) for i in range(200))
```

14.3. Itérables et itérateurs

Un autre opérateur particulier applicable sur les générateurs est la fonction `next()`, qui renvoie la première valeur d'un générateur. Il est utile si l'on souhaite connaître par exemple la première valeur pour laquelle la longueur de la suite de Syracuse est supérieure à 100.

```
>>> next(i for i in range(1, 50) if length(syracuse(i)) > 100)
27
```

Si on souhaite connaître l'entier suivant pour lequel la longueur de la suite de Syracuse est supérieure à 100, il est possible de stocker le générateur dans une variable, et d'appeler `next()` une deuxième fois :

```
>>> g = (i for i in range(1, 50) if length(syracuse(i)) > 100)
>>> next(g), next(g)
27, 31
```

Quand un générateur est épuisé, une exception `StopIteration` est levée :

```
>>> next(i for i in range(10) if i > 10)
Traceback (most recent call last):
  ...
StopIteration
```

Il est alors possible de définir en deuxième argument une valeur par défaut (souvent `None`) pour éviter les exceptions :

```
>>> next((i for i in range(10) if i > 10), None)  # None
```

La fonction `next()` s'applique dans un cadre plus général que celui des générateurs. Pour autant elle ne s'applique pas à n'importe quelle structure itérable :

```
>>> next([1, 2, 3])
Traceback (most recent call last):
   ...
TypeError: 'list' object is not an iterator
```

La clé est dans le message d'erreur : une liste n'est pas un *itérateur*.

Python fait la distinction entre deux termes, *itérable* et *itérateur* : une structure *itérable* est une structure à partir de laquelle il est possible, de démarrer une ou plusieurs itérations, de produire un *itérateur*. Un *itérateur* applique l'itération. À chaque étape il se *consomme*, avant de *s'épuiser* avec une exception `StopIteration`.

La fonction *built-in* `next()` ne s'applique qu'à un *itérateur*, c'est-à-dire une structure qui se consomme, comme un générateur. Il est possible de créer un *itérateur* à partir d'une structure *itérable* à l'aide de la fonction *built-in* `iter()`.

```
>>> it = iter([1, 2, 3])
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
...
StopIteration
```

Enfin, un *itérateur* est également *itérable* : la fonction `iter()` appliquée à un itérateur renvoie simplement l'itérateur passé en argument.

14.4. Le module `itertools`

La librairie standard fournit de nombreux générateurs ou itérateurs, par exemple `range`, `Path.glob("*")`, le résultat de `map()`, de `filter()`. Une arithmétique des itérateurs est également fournie par le langage, essentiellement dans le module `itertools`.

Cette section décrit l'usage de certaines de ces fonctions pour chaîner, combiner, accumuler, fusionner ou réduire des itérateurs.

Dans notre premier exemple sur `next()`, on souhaite connaître la première valeur pour laquelle la longueur de la suite de Syracuse est supérieure à 100 :

```
>>> next(i for i in range(1, 50) if length(syracuse(i)) > 100)
27
```

Ce code ne fonctionne plus pour une longueur supérieure à 120, parce que la question que nous avons posée était « quelle est la première valeur inférieure à 50 [...] ? ».

```
>>> next(i for i in range(1, 50) if length(syracuse(i)) > 120)
Traceback (most recent call last):
...
StopIteration
```

Si nous n'avons pas de moyen de borner notre itération, il est possible d'utiliser l'itérateur `count()`, un itérateur infini qui renvoie les entiers un par un :

```
>>> import itertools
>>> # itertools.count(start: int, step: int) -> Iterator[int]
>>> next(i for i in itertools.count(start=1) if length(syracuse(i)) > 120)
129
```

Chaînage, l'opérateur `yield from`. Un cas d'usage courant est celui du chaînage d'itérateurs. Il est facile de concaténer deux listes à itérer à l'aide de l'opérateur `+`. Pour plusieurs itérateurs `i1`, `i2`, etc., on pourrait écrire une fonction génératrice :

```
def chain(*iterables) -> "Iterator":
    """Combine un ensemble d'itérateurs.

    >>> max(chain([1, 2], {7, 9}))
    9
    """
    for it in iterables:
        for elt in it:
            yield elt
```

La double boucle peut alourdir les notations dans le code ; aussi l'opérateur `yield from` a été introduit dès Python 3.3 (PEP 380). La fonction `itertools.chain` de la librairie remplit exactement la même spécification que le code suivant :

```
def chain(*iterables) -> "Iterator":
    """Combine un ensemble d'itérateurs.

    >>> max(chain([1, 2], {7, 9}))
    9
    """
    for it in iterables:
        yield from it
```

Notons le parallèle à tirer entre d'une part les éléments de syntaxe `yield` et `yield from`, et, d'autre part, les implémentations de fonctions *récursives terminales*. Reprenons l'exemple de la factorielle :

```
def factorielle(n: int) -> int:
    if n == 0:
        return 1
    return n * factorielle(n - 1)  # ①
```

Les langages de programmation fonctionnelle (ce n'est pas le cas de Python) sont capables d'optimiser les appels aux fonctions récursives si les appels sont terminaux, c'est-à-dire que la dernière instruction appelée avant le `return` est l'appel à la fonction récursive. Sur la ligne ①, l'appel n'est pas récursif terminal (*tail-recursive* en anglais) parce que le résultat de la factorielle sera multiplié par n.

On peut modifier cette spécification de la manière suivante, avec une variable qui transmet les résultats intermédiaires à l'appel suivant :

```
def factorielle(n: int, cumul: int = 1) -> int:
    if n == 0:
        return cumul
    return factorielle(n - 1, n * cumul)
```

Cette syntaxe est alors à rapprocher de la fonction suivante, à base d'itérateurs : le `yield` simple renvoie le cas de base, et le `yield from` délègue la production des valeurs suivantes à l'appel récursif.

```
def fact_iter(n: int, cumul: int = 1) -> "Iterator[int]":
    if n == 0:  # nécessaire pour interrompre la récursion
        return
    yield cumul
    yield from fact_iter(n - 1, n * cumul)
```

Cette manière de procéder permet ici d'obtenir tous les résultats intermédiaires transmis dans la pile pendant la récursion : le dernier élément de la liste est le résultat de la factorielle. On peut aussi se contenter du résultat final par déballage ②.

```
>>> list(fact_iter(10))
[1, 10, 90, 720, 5040, 30240, 151200, 604800, 1814400, 3628800]
>>> *_, result = fact_iter(10)  # ②
>>> result
3628800
```

Filtrage. Les fonctions suivantes permettent de filtrer des éléments d'un itérable, c'est-à-dire de ne retourner que les valeurs qui retournent un certain critère :

```
phrase = "Python, un langage idéal!"
```

Nous avons déjà parlé de la fonction `filter` (☞ p. 158, § **12.3**), qui ne renvoie que les éléments évalués comme vrais par la fonction passée en paramètre.

```
>>> # uniquement les caractères alphabétiques
>>> "".join(filter(str.isalpha, phrase))
'Pythonunlangageidéal'
```

— `filterfalse(fun, iter)` renvoie les éléments évalués comme faux :

```
>>> # le complément
>>> "".join(itertools.filterfalse(str.isalpha, phrase))
',   !'
```

— `takewhile(fun, iter)` renvoie tous les éléments jusqu'au premier test qui échoue :

```
>>> # on arrête au premier caractère non alphabétique
>>> "".join(itertools.takewhile(str.isalpha, phrase))
'Python'
```

— `dropwhile(fun, iter)` renvoie tous les éléments à partir du premier test réussi :

```
>>> "".join(itertools.dropwhile(str.isupper, phrase))
'ython, un langage idéal!'
```

— `compress(iter1, iter2)` agit comme un masque NumPy : il renvoie tous les éléments de `iter1` qui correspondent à un élément évaluable comme vrai dans `iter2` :

```
>>> "".join(
...     itertools.compress(
...         phrase, [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1]
...     )
... )
'un ange'
```

— `islice(iter, stop)` (ou `islice(iter, start, stop[, step])`) produit un équivalent à la notation `[start:stop(:step)]` pour les itérateurs :

```
>>> "".join(itertools.islice(s, 10))  # s[:10]
'Python, un'
```

Application. Nous avons déjà parlé de la fonction `map`, qui crée un itérateur constitué du résultat de l'application d'une fonction à chacun des éléments d'un itérateur en entrée :

```
>>> sequence = [2, 3, 7, 6, 4, 5, 8, 9, 1]
>>> list(map(lambda x: x + 1, sequence))
[3, 4, 8, 7, 5, 6, 9, 10, 2]
```

— `accumulate(iter[, fun])` renvoie une somme cumulée des éléments passés. Si une fonction est passée en paramètre, elle est appliquée à la place de la somme :

```
>>> list(itertools.accumulate(sequence))
[2, 5, 12, 18, 22, 27, 35, 44, 45]
>>> [2, 5, 12, 18, 22, 27, 35, 44, 45]
[2, 3, 7, 7, 7, 7, 8, 9, 9]
>>> # calcul de la factorielle
>>> list(itertools.accumulate(range(1, 10), operator.mul))
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

— `starmap(fun, iter)` applique `fun` à chacun des éléments `elt` de `iter`; chaque `elt` doit être à son tour itérable pour appeler `fun(*elt)`.
Dans l'exemple ci-dessous, la fonction `zip` renvoie le premier élément de chaque collection, plus le deuxième, et ainsi de suite. On calcule alors $0 + 9$, $1 + 8$, etc.

```
>>> list(itertools.starmap(operator.add, zip(range(10), reversed(range(10)))))
[9, 9, 9, 9, 9, 9, 9, 9, 9, 9]
```

L'exemple suivant produit une moyenne cumulée : `accumulate()` calcule la somme, `enumerate(iter, 1)` compte le nombre d'éléments (en démarrant à 1), et la fonction anonyme se charge de la division :

```
>>> list(
...     itertools.starmap(
...         lambda a, b: b / a,
...         enumerate(itertools.accumulate(sequence), start=1)
...     )
... )
[2.0, 2.5, 4.0, 4.5, 4.4, 4.5, 5.0, 5.5, 5.0]
```

Produits.

— `product(iter1, iter2, ...)` génère le produit cartésien de tous les itérateurs passés en paramètres.

```
>>> couleurs = ["♠", "♥", "♦", "♣"]
>>> valeurs = ["A", "R", "D", "V", "10", "9", "8", "7"]
>>> "".join(("A", "♠"))
'A♠'
>>> " ".join("".join(carte) for carte in itertools.product(valeurs, couleurs))
'A♠ A♥ A♦ A♣ R♠ R♥ R♦ R♣ D♠ D♥ D♦ D♣ V♠ V♥ V♦ V♣ 10♠ 10♥ 10♦ 10♣ 9♠ 9♥ 9♦ 9♣
 8♠ 8♥ 8♦ 8♣ 7♠ 7♥ 7♦ 7♣'
```

— `combinations(iter, i)` génère l'ensemble des combinaisons possibles de i éléments parmi ceux fournis par l'itérateur. On peut alors compter le nombre de jeux de 7 cartes qu'il est possible de tirer au jeu de la belote :

```
>>> sum(1 for _ in itertools.combinations(itertools.product(valeurs, couleurs), 7))
3365856
```

— `permutations(iter, i)` est similaire à `combinations` mais prend en compte l'ordre dans lequel sont placés les élements en sortie :

```
>>> list(itertools.combinations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'C')]
>>> list(itertools.permutations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
```

— à l'image de `count()`, `cycle(iter)` renvoie un itérateur infini qui redémarre l'itérateur courant une fois celui-ci épuisé. Nous utilisons `islice` pour en extraire quelques éléments :

```
>>> list(itertools.islice(itertools.count(), 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(itertools.islice(itertools.cycle(couleurs), 10))
['♠', '♥', '♦', '♣', '♠', '♥', '♦', '♣', '♠', '♥']
```

— enfin, `repeat()` permet de répéter un élément donné. Cet élément de syntaxe est confortable pour éviter de créer des listes intermédiaires. Si l'argument entier de `repeat` n'est pas indiqué, alors la répétition est infinie :

```
>>> list(zip(valeurs, "♥"))
[('A', '♥')]
>>> list(zip(valeurs, itertools.repeat("♥", 4)))
[('A', '♥'), ('R', '♥'), ('D', '♥'), ('V', '♥')]
>>> list(zip(valeurs, itertools.repeat("♥")))
[('A', '♥'), ('R', '♥'), ('D', '♥'), ('V', '♥'), ('10', '♥'), ('9', '♥'),
 ('8', '♥'), ('7', '♥')]
```

Réarrangement. Ces deux fonctions sont moins connues que les précédentes, mais leur fonctionnement est familier des utilisateurs du shell Unix (`itertools.tee()`) et de la bibliothèque Pandas (`itertools.groupby()`, ☞ p. 121, § **10**) :

— l'outil shell `tee` duplique la sortie standard d'un programme pour rediriger cette duplication vers l'entrée standard du programme suivant. Dans le module `itertools`, la fonction `tee(iter, n=2)` permet de dupliquer la sortie des itérateurs pour la consommer plusieurs fois :

```
>>> t1, t2 = itertools.tee((i * 2 for i in range(10)))
>>> next(t1), next(t1), next(t1)
(0, 2, 4)
>>> next(t2)
0
```

On peut ainsi produire un itérateur qui calcule la différence entre deux éléments consécutifs :

```
>>> sequence
[2, 3, 7, 6, 4, 5, 8, 9, 1]
>>> t1, t2 = itertools.tee(sequence)
>>> _ = next(t1)  # ignorer le résultat
>>> sub = list(itertools.starmap(operator.sub, zip(t1, t2)))
>>> sub
[1, 4, -1, -2, 1, 3, 1, -8]
```

— `groupby(iter, key=None)` renvoie des tuples `clé, iterateur` avec tous les éléments qui vérifient le critère clé. Contrairement à Pandas, `groupby` suppose que les éléments de `iter` sont regroupés (triés par exemple ③) :

```
def signe(x):
    return int(x / abs(x))

"    ".join(
    f"{key:2d} -> {list(it)}"
    for key, it in itertools.groupby(sorted(sub), key=signe)  # ③
)
# '-1 -> [-8, -2, -1]     1 -> [1, 1, 1, 3, 4]'
```

14.5. Les coroutines

Les coroutines partagent un élément de syntaxe avec les générateurs : le mot-clé `yield`, à ceci près que celui-ci est précédé du signe égal. Dans un générateur, la ligne `yield elt` produit une valeur `elt` qui sera consommée par la fonction qui utilise le mot-clé `next()`, et se met attente du prochain appel à `next()`.

Dans une coroutine, le mot-clé `yield` est à droite du signe égal. Cette fois, la coroutine va consommer des données fournies par la fonction appelante à l'aide du mot-clé `.send()`.

```
def allo():
    x = yield
    print(f"Allô, j'écoute: {x}")
```

```
>>> coco = allo()
>>> coco
<generator object allo at 0x7f527aea1820>
>>> next(coco)
Traceback (most recent call last):
...
TypeError: can't send non-None value to a just-started generator
```

Pour pouvoir commencer à faire consommer des données par la coroutine, il est nécessaire de la démarrer à l'aide de la fonction `next()`. Comme `coco` est un générateur, il se termine systématiquement par une exception `StopIteration`.

```
>>> next(coco)
>>> coco.send("Mille sabords!")
Allô, j'écoute: Mille sabords!
Traceback (most recent call last):
...
StopIteration
```

Il est courant de démarrer les coroutines à l'aide du décorateur suivant qui initialise (on utilise le verbe *to prime (a coroutine)* en anglais) automatiquement les coroutines ① :

```
import functools

def coroutine(fun):
    @functools.wraps(fun)
    def wraps(*args, **kwargs):
        gen = fun(*args, **kwargs)
        next(gen)  # ①
        return gen
    return wraps
```

```
@coroutine
def allo():
    x = yield
    print(f"Allô, j'écoute: {x}")
```

```
>>> coco = allo()
>>> # next(coco) a été exécuté lors de l'appel à allo(), à la ligne ①
>>> coco.send("Non, Madame, ce n'est pas la boucherie Sanzot!")
Allô, j'écoute: Non, Madame, ce n'est pas la boucherie Sanzot!
Traceback (most recent call last):
...
StopIteration
```

On peut choisir en exemple d'utilisation des coroutines une fonction qui reçoit des valeurs pour retourner la moyenne des valeurs consommées :

```
@coroutine
def moyenne():
    total = 0.0
    average = None
    for compteur in count(1):  # ②
        terme = yield average
        total += terme
        average = total / compteur
```

À chaque appel de `.send()`, une itération se fait dans la boucle infinie ② qui incrémente un compteur, et accumule la somme des valeurs reçues pour renvoyer la moyenne des valeurs reçues.

```
>>> moy = moyenne()
>>> ", ".join(f"{elt} -> {moy.send(elt)}" for elt in sequence)
'2 -> 2.0, 3 -> 2.5, 7 -> 4.0, 6 -> 4.5, 4 -> 4.4, 5 -> 4.5, 8 -> 5.0,
 9 -> 5.5, 1 -> 5.0'
```

Attention !

La coroutine n'étant pas terminée, il est possible de poursuivre le calcul de la moyenne en ajoutant des valeurs. Il est donc normal de ne pas obtenir la même sortie que précédemment.

```
>>> ", ".join(f"{elt} -> {moy.send(elt):.2f}" for elt in sequence)
'2 -> 4.70, 3 -> 4.55, 7 -> 4.75, 6 -> 4.85, 4 -> 4.79, 5 -> 4.80, 8 -> 5.00,
 9 -> 5.24, 1 -> 5.00'
```

Attention !

Si une exception non rattrapée **à l'intérieur de** la coroutine a lieu, la coroutine est alors quittée.

```
>>> try:
...     moy.send("grossière erreur")
... except TypeError as exc:
...     print(exc)
unsupported operand type(s) for +=: 'float' and 'str'
>>> moy.send(1)
Traceback (most recent call last):
...
StopIteration
```

Il faut plutôt rattraper l'exception dans la coroutine pour éviter d'interrompre celle-ci.

```
@coroutine
def moyenne():
    total = 0.0
    count = 0
    average = None
    while True:
        try:
            terme = yield average
            total += terme
            count += 1
            average = total / count
        except TypeError:
            print("On n'a rien vu...")
```

```
>>> moy = moyenne()
>>> moy.send("grossière erreur")
On n'a rien vu...
>>> moy.send(1)
1.0
>>> moy.throw(TypeError)  # on peut envoyer directement une exception
On n'a rien vu...
```

```
1.0
>>> moy.send(2)
1.5
```

Il est également courant de prévoir une garde ④ pour interrompre la coroutine et renvoyer une valeur. La valeur de retour est alors contenue dans l'exception `StopIteration` :

```
@coroutine
def moyenne():
    total = 0.0
    count = 0
    average = None
    while True:
        try:
            terme = yield
            if terme is None:  # ④
                break
            total += terme
            count += 1
            average = total / count
        except TypeError:
            print("On n'a rien vu...")

    return average

>>> moy = moyenne()
>>> moy.send(1)
>>> moy.send(None)
Traceback (most recent call last):
...
StopIteration: 1.0
```

On peut récupérer la valeur dans une variable à l'aide des gardes d'exception :

```
moy = moyenne()
moy.send(1)
try:
    moy.send(None)
except StopIteration as exc:
    result = exc.value
result  # 1.0
```

Le parallèle entre générateurs, qui produisent des données, et coroutines, qui consomment des données, est également intéressant lors du chaînage de fonctions. Dans le premier exemple, on consomme les données sorties de `range(10)`, pour les faire passer successivement dans `mul_2`, `add_1`, `add_1`, et ainsi de suite, avant de constituer une liste :

```
def add_1(it):
    for elt in it:
        yield elt + 1
```

```
def mul_2(it):
    for elt in it:
        yield 2 * elt
```

```
chaine = functools.reduce(
    lambda x, f: f(x),
    [mul_2, add_1, add_1, mul_2, add_1],  # ③
    range(10)
)
list(chaine)  # [5, 9, 13, 17, 21, 25, 29, 33, 37, 41]
```

Dans le second exemple, la liste de résultats est au plus profond de la pile d'appel : c'est la coroutine `start_elt` qui *nourrira* cette liste, à partir des éléments reçus de `add_1`, qui les reçoit de `mul_2`, de `add_1`, et ainsi de suite. La composition des fonctions est alors faite dans le sens inverse :

```
@coroutine
def add_1(output):
    while True:
        elt = yield
        output.send(elt + 1)

@coroutine
def mul_2(output):
    while True:
        elt = yield
        output.send(elt * 2)
```

```
@coroutine
def ajoute(liste):
    while True:
        elt = yield
        liste.append(elt)

resultat = list()
chaine = functools.reduce(
    lambda x, f: f(x),
    [add_1, mul_2, add_1, add_1, mul_2],  # ④
    ajoute(resultat),
)
```

```
for elt in range(10):
    chaine.send(elt)

resultat  # [5, 9, 13, 17, 21, 25, 29, 33, 37, 41]
```

> **Attention !**
>
> Bien noter que les deux listes de fonctions ③ et ④ sont chaînées dans des sens opposés comme sur la figure 14.2.

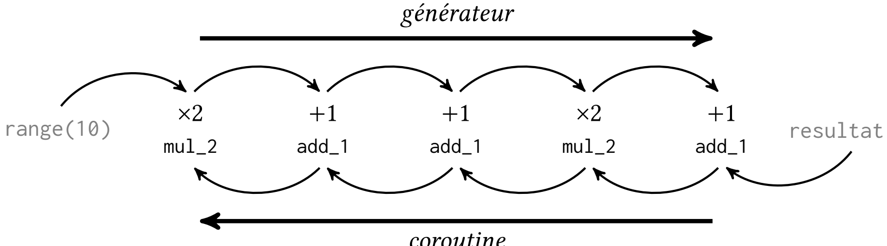

Figure 14.2 – Générateurs et coroutines

En quelques mots…

Python propose une syntaxe et un formalisme riches autour du protocole de l'itération. Deux structures fonctionnent de façons opposées : les *générateurs*, qui produisent des données à chaque itération, et les *coroutines*, qui en consomment.

Les générateurs peuvent être produits par les notations en compréhension ou avec des fonctions qui utilisent le mot-clé `yield`. C'est un mécanisme extrêmement puissant qui trouve de nombreuses applications réelles, notamment pour personnaliser des motifs d'itération.

Quand `yield` est situé à droite du signe égal, la fonction devient une coroutine. Les coroutines sont un concept des années 1960 qui ont laissé la place aux processus légers (appelés aussi *threads*, ☞ p. 261, § **18.2**). Néanmoins, la facilité avec laquelle elles permettent d'interrompre une exécution a permis de construire les bases du module `asyncio` en Python (☞ p. 266, § **18.5**).