13 Containers, dictionnaires, tuples et *sets*

Dans ce chapitre nous allons voir trois nouveaux types d'objet qui s'avèrent extrêmement utiles : les dictionnaires, les tuples et les *sets*. Comme les listes ou les chaînes de caractères, ces trois nouveaux types sont appelés communémement des **containers**. Avant d'aborder en détail ces nouveaux types, nous allons définir les containers et leurs propriétés.

13.1 Containers

13.1.1 Définition

Définition

Un **container** est un nom générique pour définir un objet Python qui contient une collection d'autres objets.

Les containers que nous connaissons depuis le début de ce cours sont les listes et les chaînes de caractères. Même si on ne l'a pas vu explicitement, les objets de type *range* sont également des containers.

Dans la section suivante, nous allons examiner les différentes propriétés des containers. A la fin de ce chapitre, nous ferons un tableau récapitulatif de ces propriétés.

13.1.2 Propriétés

Examinons d'abord les propriétés qui caractérisent tous les types de container.

- Capacité à supporter le **test d'appartenance**. Souvenez-vous, il permettait de vérifier si un élément était présent dans une liste. Cela fonctionne donc aussi sur les chaînes de caractères ou tout autre container :

```
>>> l = [1, 2, 3]
>>> 1 in l
True
>>> "to" in "toto"
True
```

- Capacité à supporter la fonction `len()` renvoyant la longueur du container.

Voici d'autres propriétés générales que nous avons déjà croisées. Un container peut être :

- **ordonné** (*ordered* en anglais) : il y a un ordre précis des éléments ; cet ordre correspond à celui utilisé lors de la création ou de la modification du container (si cela est permis) ; ce même ordre est utilisé lorqu'on itère dessus ;
- **indexable** (*subscriptable* en anglais) : on peut retrouver un élément par son indice (i.e. sa position dans le container) ou plusieurs éléments avec une tranche ; en général, tout container indexable est ordonné ;
- **itérable** (*iterable* en anglais) : on peut faire une boucle dessus.

Certains containers sont appelés objets séquentiels ou séquence.

Définition

Un **objet séquentiel** ou **séquence** est un container itérable, ordonné et indexable. Les objets séquentiels sont les listes, les chaînes de caractères, les objets de type *range*, ainsi que les tuples (cf. plus bas).

Une autre propriété importante que l'on a déjà croisée et qui nous servira dans ce chapitre concerne la possiblité ou non de modifier un objet.

- Un objet est dit **non modifiable** lorsqu'on ne peut pas le modifier, ou lorsqu'on ne peut pas en modifier un de ses éléments si c'est un container. On parle aussi d'objet immuable (*immutable object* en anglais). Cela signifie qu'une fois créé, Python ne permet plus de le modifier par la suite.

Qu'en est-il des objets que nous connaissons ? Les listes sont modifiables, on peut modifier un ou plusieurs de ses éléments. Tous les autres types que nous avons vus précédemment sont quant à eux non modifiables : les chaînes de caractères ou *strings*, les objets de type *range*, mais également des objets qui ne sont pas des containers comme les entiers, les *floats* et les booléens.

On comprend bien l'immutabilité des *strings* comme vu au chapitre 10, mais c'est moins évident pour les entiers, *floats* ou booléens. Nous allons démontrer cela, mais avant nous avons besoin de définir la notion d'identifiant d'un objet.

Définition

L'**identifiant** d'un objet est un nombre entier qui est garanti constant pendant toute la durée de vie de l'objet. Cet identifiant est en général unique pour chaque objet. Toutefois, pour des raisons d'optimisation, Python crée parfois le même identifiant pour deux objets non modifiables différents qui ont la même valeur. L'identifiant peut être assimilé à l'adresse mémoire de l'objet qui elle aussi est unique. En Python, on utilise la fonction interne `id()` qui prend en argument un objet et renvoie son identifiant.

Maintenant que l'identifiant est défini, regardons l'exemple suivant qui montre l'immutabilité des entiers.

```
1  >>> a = 4
2  >>> id(a)
3  140318876873440
4  >>> a = 5
5  >>> id(a)
6  140318876873472
```

En ligne 1 on définit l'entier `a` puis on regarde son identifiant. En ligne 4, on pourrait penser que l'on modifie `a`. Toutefois, on voit que son identifiant en ligne 6 est différent de la ligne 3. En fait, l'affectation en ligne 4 `a = 5` écrase l'ancienne variable `a` et en crée une nouvelle, ce n'est pas la valeur de `a` qui a été changée puisque l'identifiant n'est plus le même. Le même raisonnement peut être tenu pour les autres types numériques comme les *floats* et booléens. Si on regarde maintenant ce qu'il se passe pour une liste :

```
1  >>> l = [1, 2, 3]
2  >>> id(l)
3  140318850324832
4  >>> l[1] = -15
5  >>> id(l)
6  140318850324832
7  >>> l.append(5)
```

```
>>> id(l)
140318850324832
```

La liste `l` a été modifiée en ligne 4 (changement de l'élément d'indice 1) et en ligne 7 (ajout d'un élément). Pour autant, l'identifiant de cette liste est resté identique tout du long. Ceci démontre la mutabilité des listes : quelle que soit la manière dont on modifie une liste, celle-ci garde le même identifiant.

- Une dernière propriété importante est la capacité d'un container (ou tout autre objet Python) à être **hachable**.

Définition

Un objet Python est dit **hachable** (*hashable* en anglais) s'il est possible de calculer une valeur de hachage sur celui-ci avec la fonction interne `hash()`. En programmation, la valeur de hachage peut être vue comme une empreinte numérique de l'objet. Elle est obtenue en passant l'objet dans une fonction de hachage et dépend du contenu de l'objet. En Python, cette empreinte est comme dans la plupart des langages de programmation un entier. Au sein d'une même session Python, deux objets hachables qui ont un contenu identique auront strictement la même valeur de hachage.

Attention

La valeur de hachage d'un objet renvoyée par la fonction `hash()` n'a pas le même sens que son identifiant renvoyé par la fonction `id()`. La valeur de hachage est obtenue en « moulinant » le contenu de l'objet dans une fonction de hachage. L'identifiant est quant à lui attribué par Python à la création de l'objet. Il est constant tout le le long de la durée de vie de l'objet, un peu comme une carte d'identité. Tout objet a un un identifiant, mais il doit être hachable pour avoir une valeur de hachage.

Pour aller plus loin

Pour aller plus loin, vous pouvez consulter la page Wikipedia sur les fonctions de hachage.

Pourquoi évoquer cette propriété de hachabilité ? D'abord, parce-qu'elle est étroitement liée à l'immutabilité. En effet, un objet non modifiable est la plupart du temps hachable. Cela permet de l'identifier **en fonction de son contenu**. Par ailleurs, l'hachabilité est une implémentation qui permet un accès rapide aux éléments des containers de type dictionnaire ou *set* (cf. rubriques suivantes).

Les objets hachables sont les chaînes de caractères, les entiers, les *floats*, les booléens, les objets de type *range*, les tuples (sous certaines conditions) et les *frozensets* ; par contre, les listes, les *sets* et les dictionnaires sont non hachables. Les dictionnaires, tuples, *sets* et *frozensets* seront vus plus bas dans ce chapitre.

Voici un exemple :

```
>>> hash("Plouf")
5085648805260210718
>>> hash(5)
5
>>> hash(3.14)
322818021289917443
>>> hash([1, 2, 3])
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Les valeurs de hachage renvoyées par la fonction `hash()` de Python sont systématiquement des entiers. Par contre, Python renvoie une erreur pour une liste car elle est non hachable.

13.1.3 Containers de type *range*

Revenons rapidement sur les objets de type *range*. Jusqu'à maintenant, on s'en est servi pour faire des boucles ou générer des listes de nombres. Toutefois, on a vu ci-dessus qu'ils étaient aussi des containers. Ils sont ordonnés, indexables, itérables, hachables et non modifiables.

```
>>> r = range(3)
>>> r[0]
0
>>> r[0:1]
range(0, 1)
>>> for i in r:
...     print(i)
...
0
1
2
>>> r[2] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'range' object does not support item assignment
>>> hash(r)
5050907061201647097
```

La tentative de modification d'un élément en ligne 12 conduit à la même erreur que lorsqu'on essaie de modifier un caractère d'une chaîne de caractères. Comme pour la plupart des objets Python non modifiables, les objets de type *range* sont hachables.

13.2 Dictionnaires

13.2.1 Définition

Les **dictionnaires** se révèlent très pratiques lorsque vous devez manipuler des structures complexes à décrire et que les listes présentent leurs limites. Les dictionnaires sont des collections non ordonnées d'objets (ceci est vrai jusqu'à la version 3.6 de Python, voir remarque ci-dessous). Il ne s'agit pas d'objets séquentiels comme les listes ou chaînes de caractères, mais plutôt d'objets dits de correspondance (*mapping objects* en anglais) ou tableaux associatifs. En effet, on accède aux **valeurs** d'un dictionnaire par des **clés**. Ceci semble un peu confus ? Regardez l'exemple suivant :

```
>>> ani1 = {}
>>> ani1["nom"] = "girafe"
>>> ani1["taille"] = 5.0
>>> ani1["poids"] = 1100
>>> ani1
{'nom': 'girafe', 'taille': 5.0, 'poids': 1100}
```

En premier, on définit un dictionnaire vide avec les accolades `{}` (tout comme on peut le faire pour les listes avec `[]`). Ensuite, on remplit le dictionnaire avec différentes clés (`"nom"`, `"taille"`, `"poids"`) auxquelles

on affecte des valeurs (`"girafe"` , `5.0` , `1100`). Vous pouvez mettre autant de clés que vous voulez dans un dictionnaire (tout comme vous pouvez ajouter autant d'éléments que vous voulez dans une liste).

Remarque

Jusqu'à la version 3.6 de Python, un dictionnaire était affiché sans ordre particulier. L'ordre d'affichage des éléments n'était pas forcément le même que celui dans lequel il avait été rempli. De même lorsqu'on itérait dessus, l'ordre n'était pas garanti. Depuis Python 3.7 (inclus), ce comportement a changé, un dictionnaire est toujours affiché dans le même ordre que celui utilisé pour le remplir. De même, si on itère sur un dictionnaire, cet ordre est respecté. Ce détail provient de l'implémentation interne des dictionnaires dans Python, mais cela nous concerne peu. Ce qui importe, c'est de se rappeler qu'on accède aux éléments par des clés, donc cet ordre n'a pas d'importance spéciale sauf dans de rares cas.

On peut aussi initialiser toutes les clés et les valeurs d'un dictionnaire en une seule opération :

```
>>> ani2 = {"nom": "singe", "poids": 70, "taille": 1.75}
```

Mais rien ne nous empêche d'ajouter une clé et une valeur supplémentaire :

```
>>> ani2["age"] = 15
```

Pour récupérer la valeur associée à une clé donnée, il suffit d'utiliser la syntaxe suivante `dictionnaire["cle"]` . Par exemple :

```
>>> ani1["taille"]
5.0
```

Après ce premier tour d'horizon, on voit tout de suite l'avantage des dictionnaires. Pouvoir retrouver des éléments par des noms (clés) plutôt que par des indices. Les humains retiennent mieux les noms que les chiffres. Ainsi, l'usage des dictionnaires rend en général le code plus lisible. Par exemple, si nous souhaitions stocker les coordonnées (x, y, z) d'un point dans l'espace : `coors = [0, 1, 2]` pour la version liste, `coors = {"x": 0, "y": 1, "z": 2}` pour la version dictionnaire. Un lecteur comprendra tout de suite que `coors["z"]` contient la coordonnée z, ce sera moins intuitif avec `coors[2]` .

13.2.2 Objets utilisables comme clé

Toutes les clés de dictionnaire utilisées jusqu'à présent étaient des chaînes de caractères. On peut utiliser d'autres types d'objets comme des entiers, des *floats*, voire même des *tuples* (cf. rubrique suivante), cela peut s'avérer parfois très utile. Une règle est toutefois requise, les objets utilisés comme clé doivent être **hachables** (cf. rubrique précédente pour la définition).

Pourquoi les clés doivent être des objets hachables ? C'est la raison d'être des dictionnaires, d'ailleurs ils sont aussi appelés table de hachage dans d'autres langages comme Perl. Convertir chaque clé en sa valeur de hachage permet un accès très rapide à chacun des éléments du dictionnaire ainsi que des comparaisons de clés entre dictionnaires extrêmement efficaces. Même si on a vu que deux objets pouvaient avoir la même valeur de hachage, par exemple `a = 5` et `b = 5` , on ne peut mettre qu'une seule fois la clé `5` . Ceci assure que deux clés d'un même dictionnaire ont forcément une valeur de hachage différente.

Conseils

Malgré les possibilités offertes, nous vous conseillons de n'utiliser que des chaînes de caractères pour vos clés de dictionnaire lorsque vous débutez.

13.2.3 Itération sur les clés pour obtenir les valeurs

Si on souhaite voir toutes les associations clés / valeurs, on peut itérer sur un dictionnaire de la manière suivante :

```
>>> ani2 = {'nom': 'singe', 'poids': 70, 'taille': 1.75}
>>> for key in ani2:
...     print(key, ani2[key])
...
poids 70
nom singe
taille 1.75
```

Par défaut, l'itération sur un dictionnaire se fait sur les clés. Dans cet exemple, la variable d'itération `key` prend successivement la valeur de chaque clé, `ani2[key]` donne la valeur correspondant à chaque clé.

13.2.4 Méthodes `.keys()`, `.values()` et `.items()`

Les méthodes `.keys()` et `.values()` renvoient, comme vous vous en doutez, les clés et les valeurs d'un dictionnaire :

```
>>> ani2.keys()
dict_keys(['poids', 'nom', 'taille'])
>>> ani2.values()
dict_values([70, 'singe', 1.75])
```

Les mentions `dict_keys` et `dict_values` indiquent que nous avons à faire à des objets un peu particuliers. Ils ne sont pas indexables (on ne peut pas retrouver un élément par indice, par exemple `dico.keys()[0]` renverra une erreur). Si besoin, nous pouvons les transformer en liste avec la fonction `list()` :

```
>>> ani2.values()
dict_values(['singe', 70, 1.75])
>>> list(ani2.values())
['singe', 70, 1.75]
```

Toutefois, ce sont des objets itérables, donc utilisables dans une boucle.

Conseil : pour les débutants, vous pouvez sauter cette fin de rubrique.

Enfin, il existe la méthode `.items()` qui renvoie un nouvel objet `dict_items` :

```
>>> dico = {0: "t", 1: "o", 2: "t", 3: "o"}
>>> dico.items()
dict_items([(0, 't'), (1, 'o'), (2, 't'), (3, 'o')])
```

Celui-ci n'est pas indexable (on ne peut pas retrouver un élément par un indice) mais il est itérable :

```
>>> dico.items()[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_items' object is not subscriptable
```

```
>>> for key, val in dico.items():
...     print(key, val)
...
0 t
1 o
2 t
3 o
```

Notez la syntaxe particulière qui ressemble à la fonction `enumerate()` vue au chapitre 5 *Boucles et comparaisons*. On itère à la fois sur `key` et sur `val`. On verra plus bas que cela peut-être utile pour construire des dictionnaires de compréhension.

13.2.5 Existence d'une clé ou d'une valeur

Pour vérifier si une clé existe dans un dictionnaire, on peut utiliser le test d'appartenance avec l'opérateur `in` qui renvoie un booléen :

```
>>> ani2 = {'nom': 'singe', 'poids': 70, 'taille': 1.75}
>>> if "poids" in ani2:
...     print("La clé 'poids' existe pour ani2")
...
La clé 'poids' existe pour ani2
>>> if "age" in ani2:
...     print("La clé 'age' existe pour ani2")
...
```

Dans le second test (lignes 5 à 7), le message n'est pas affiché car la clé `age` n'est pas présente dans le dictionnaire `ani2`.

Si on souhaite tester si une valeur existe dans un dictionnaire, on peut utiliser l'opérateur `in` avec l'objet renvoyé par la méthode `.values()` :

```
>>> ani2 = {'nom': 'singe', 'poids': 70, 'taille': 1.75}
>>> ani2.values()
dict_values(['singe', 70, 1.75])
>>> "singe" in ani2.values()
True
```

13.2.6 Méthode `.get()`

Par défaut, si on demande la valeur associée à une clé qui n'existe pas, Python renvoie une erreur :

```
>>> ani2 = {'nom': 'singe', 'poids': 70, 'taille': 1.75}
>>> ani2["age"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'age'
```

La méthode `.get()` s'affranchit de ce problème. Elle extrait la valeur associée à une clé mais ne renvoie pas d'erreur si la clé n'existe pas :

```
>>> ani2.get("nom")
'singe'
>>> ani2.get("age")
>>>
```

Ici la valeur associée à la clé `nom` est `singe` mais la clé `age` n'existe pas. On peut également indiquer à `.get()` une valeur par défaut si la clé n'existe pas :

```
>>> ani2.get("age", 42)
42
```

13.2.7 Tri par clés

On peut utiliser la fonction `sorted()` vue précédemment avec les listes pour trier un dictionnaire par ses clés :

```
>>> ani2 = {'nom': 'singe', 'taille': 1.75, 'poids': 70}
>>> sorted(ani2)
['nom', 'poids', 'taille']
```

Les clés sont triées ici par ordre alphabétique.

13.2.8 Tri par valeurs

Pour trier un dictionnaire par ses valeurs, il faut utiliser la fonction `sorted` avec l'argument `key` :

```
>>> dico = {"a": 15, "b": 5, "c":20}
>>> sorted(dico, key=dico.get)
['b', 'a', 'c']
```

L'argument `key=dico.get` indique explicitement qu'il faut réaliser le tri par les valeurs du dictionnaire. On retrouve la méthode `.get()` vue plus haut, mais sans les parenthèses : `key=dico.get` mais pas `key=dico.get()`. Une fonction ou méthode passée en argument sans les parenthèses est appelée *callback*, nous reverrons cela en détail dans le chapitre 20 *Fenêtres graphiques et Tkinter*.

Attention, ce sont les clés du dictionnaires qui sont renvoyées, pas les valeurs. Ces clés sont cependant renvoyées dans un ordre qui permet d'obtenir les clés triées par ordre croissant :

```
>>> dico = {"a": 15, "b": 5, "c":20}
>>> for key in sorted(dico, key=dico.get):
...     print(key, dico[key])
...
b 5
a 15
c 20
```

Enfin, l'argument `reverse=True` fonctionne également :

```
>>> dico = {"a": 15, "b": 5, "c":20}
>>> sorted(dico, key=dico.get, reverse=True)
['c', 'a', 'b']
```

Remarque

Lorsqu'on trie un dictionnaire par ses valeurs, il faut être sûr que cela soit possible. Ce n'est, par exemple, pas le cas pour le dictionnaire `ani2` car les valeurs sont des valeurs numériques et une chaîne de caractères :

```
>>> ani2 = {'nom': 'singe', 'poids': 70, 'taille': 1.75}
>>> sorted(ani2, key=ani2.get)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

On obtient ici une erreur car Python ne sait pas comparer une chaîne de caractères (`singe`) avec des valeurs numériques (`70` et `1.75`).

13.2.9 Clé associée au minimum ou au maximum des valeurs

Les fonctions `min()` et `max()`, que vous avez déjà manipulées dans les chapitres précédents, acceptent également l'argument `key=`. On peut ainsi obtenir la clé associée au minimum ou au maximum des valeurs d'un dictionnaire :

```
>>> dico = {"a": 15, "b": 5, "c":20}
>>> max(dico, key=dico.get)
'c'
>>> min(dico, key=dico.get)
'b'
```

13.2.10 Liste de dictionnaires

En créant une liste de dictionnaires qui possèdent les mêmes clés, on obtient une structure qui ressemble à une base de données :

```
>>> animaux = [ani1, ani2]
>>> animaux
[{'nom': 'girafe', 'poids': 1100, 'taille': 5.0}, {'nom': 'singe',
'poids': 70, 'taille': 1.75}]
>>>
>>> for ani in animaux:
...     print(ani["nom"])
...
girafe
singe
```

Vous constatez ainsi que les dictionnaires permettent de gérer des structures complexes de manière plus explicite que les listes.

13.2.11 Fonction `dict()`

Conseil : Pour les débutants vous pouvez sauter cette rubrique.

La fonction `dict()` va convertir l'argument qui lui est passé en dictionnaire. Il s'agit donc d'une fonction de *casting* comme `int()`, `str()`, etc. Toutefois, l'argument qui lui est passé doit avoir une forme particulière : un objet séquentiel contenant d'autres objets séquentiels de 2 éléments. Par exemple, une liste de listes de 2 éléments :

```
>>> liste_animaux = [["girafe", 2], ["singe", 3]]
>>> dict(liste_animaux)
{'girafe': 2, 'singe': 3}
```

Ou un *tuple* de *tuples* de 2 éléments (cf. rubrique suivante pour la définition d'un *tuple*), ou encore une combinaison liste / *tuple* :

```
>>> tuple_animaux = (("girafe", 2), ("singe", 3))
>>> dict(tuple_animaux)
{'girafe': 2, 'singe': 3}
>>>
>>> dict([("girafe", 2), ("singe", 3)])
{'girafe': 2, 'singe': 3}
```

Si un des sous-éléments a plus de 2 éléments (ou moins), Python renvoie une erreur :

```
>>> dict([("girafe", 2), ("singe", 3, 4)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: dictionary update sequence element #1 has length 3; 2 is required
```

13.3 Tuples

13.3.1 Définition

Les **tuples** (« n-uplets » en français) sont des objets séquentiels correspondant aux listes (itérables, ordonnés et indexables) mais ils sont toutefois **non modifiables**. On verra plus bas qu'ils sont hachables sous certaines conditions. L'intérêt des tuples par rapport aux listes réside dans leur immutabilité. Cela, accèlère considérablement la manière dont Python accède à chaque élément et ils prennent moins de place en mémoire. Par ailleurs, on ne risque pas de modifier un de ses éléments par mégarde. Vous verrez ci-dessous que nous les avons déjà croisés à plusieurs reprises !

Pratiquement, on utilise les parenthèses au lieu des crochets pour les créer :

```
1  >>> t = (1, 2, 3)
2  >>> t
3  (1, 2, 3)
4  >>> type(t)
5  <class 'tuple'>
6  >>> t[2]
7  3
8  >>> t[0:2]
9  (1, 2)
10 >>> t[2] = 15
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13 TypeError: 'tuple' object does not support item assignment
```

L'affectation et l'indiçage fonctionnent comme avec les listes. Mais si on essaie de modifier un des éléments du tuple (en ligne 10), Python renvoie un message d'erreur. Ce message est similaire à celui que nous avions rencontré quand on essayait de modifier une chaîne de caractères (cf. chapitre 10). De manière générale, Python renverra un message `TypeError: '[...]' does not support item assignment` lorsqu'on essaie de modifier un élément d'un objet non modifiable. Si vous voulez ajouter un élément (ou le modifier), vous devez créer un nouveau tuple :

```
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
>>> id(t)
```

```
139971081704464
>>> t = t + (2,)
>>> t
(1, 2, 3, 2)
>>> id(t)
139971081700368
```

La fonction `id()` montre que le tuple créé en ligne 6 est bien différent de celui créé en ligne 4 bien qu'ils aient le même nom. Comme on a vu plus haut, ceci est dû à l'opérateur d'affectation utilisé en ligne 6 (`t = t + (2,)`) qui crée un nouvel objet distinct de celui de la ligne 1. Cet exemple montre que les tuples sont peu adaptés lorsqu'on a besoin d'ajouter, retirer, modifier des éléments. La création d'un nouveau tuple à chaque étape s'avère lourde et il n'y a aucune méthode pour faire cela puisque les tuples sont non modifiables. Pour ce genre de tâche, les listes sont clairement mieux adaptées.

Remarque

Pour créer un tuple d'un seul élément comme ci-dessus, utilisez une syntaxe avec une virgule `(element,)`, pour éviter une ambiguïté avec une simple expression. Par exemple `(2)` équivaut à l'entier `2`, `(2,)` est un tuple avec l'élément `2`.

Autre particularité des tuples, il est possible de les créer sans les parenthèses, dès lors que ceci ne pose pas d'ambiguïté avec une autre expression :

```
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
>>> t = 1, 2, 3
>>> t
(1, 2, 3)
```

Toutefois, afin d'éviter les confusions, nous vous conseillons d'utiliser systématiquement les parenthèses lorsque vous débutez.

Les opérateurs `+` et `*` fonctionnent comme pour les listes (concaténation et duplication) :

```
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)
>>> (1, 2) * 4
(1, 2, 1, 2, 1, 2, 1, 2)
```

Enfin, on peut utiliser la fonction `tuple(sequence)` qui fonctionne exactement comme la fonction `list()`, c'est-à-dire qu'elle prend en argument un objet de type container et renvoie le tuple correspondant (opération de *casting*) :

```
>>> tuple([1,2,3])
(1, 2, 3)
>>> tuple("ATGCCGCGAT")
('A', 'T', 'G', 'C', 'C', 'G', 'C', 'G', 'A', 'T')
```

Remarque

Les listes, les dictionnaires et les tuples sont des containers, c'est-à-dire qu'il s'agit d'objets qui contiennent une collection d'autres objets. En Python, on peut construire des listes qui contiennent des dictionnaires, des tuples

ou d'autres listes, mais aussi des dictionnaires contenant des tuples, des listes, etc. Les combinaisons sont infinies !

13.3.2 Itérations sur plusieurs valeurs à la fois

Pratiquement, nous avons déjà croisé les tuples avec la fonction `enumerate()` dans le chapitre 5 *Boucles et comparaisons*. Cette dernière permettait d'itérer **en même temps** sur les indices et les éléments d'une liste :

```
>>> for indice, element in enumerate([75, -75, 0]):
...     print(indice, element)
...
0 75
1 -75
2 0
>>> for bidule in enumerate([75, -75, 0]):
...     print(bidule, type(bidule))
...
(0, 75) <class 'tuple'>
(1, -75) <class 'tuple'>
(2, 0) <class 'tuple'>
```

En fin de compte, la fonction `enumerate()` itère sur une série de *tuples*. Pouvoir séparer `indice` et `element` dans la boucle est possible du fait que Python autorise l'affectation multiple du style `indice, element = 0, 75` (voir rubrique suivante).

Dans le même ordre d'idée, nous avons vu précédemment la méthode `.dict_items()` qui permettait d'itérer sur des couples clé / valeur d'un dictionnaire :

```
>>> dico = {"pinson": 2, "merle": 3}
>>> for cle, valeur in dico.items():
...     print(cle, valeur)
...
pinson 2
merle 3
>>> for bidule in dico.items():
...     print(bidule, type(bidule))
...
('pinson', 2) <class 'tuple'>
('merle', 3) <class 'tuple'>
```

La méthode `.dict_items()` itère comme `enumerate()` sur une série de tuples.

De la même façon, on peut itérer sur 3 valeurs en même temps à partir d'une liste de tuples de 3 éléments :

```
>>> liste = [(i, i+1, i+2) for i in range(5, 8)]
>>> liste
[(5, 6, 7), (6, 7, 8), (7, 8, 9)]
>>> for x, y, z in liste:
...     print(x, y, z)
...
5 6 7
6 7 8
7 8 9
```

On pourrait concevoir la même chose sur 4, 5... éléments. La seule contrainte est d'avoir une correspondance systématique entre le nombre de variables d'itération (par exemple 3 variables dans l'exemple ci-dessus avec

`x, y, z`) et la longueur de chaque sous-*tuple* de la liste sur laquelle on itère (chaque sous-*tuple* a 3 éléments ci-dessus).

13.3.3 Affectation multiple et le nom de variable `_`

L'affectation multiple est un mécanisme très puissant et important en Python. Pour rappel, il permet d'effectuer sur une même ligne plusieurs affectations en même temps, par exemple : `x, y, z = 1, 2, 3`. Cette syntaxe correspond à un *tuple* de chaque côté de l'opérateur `=`. Notez qu'il serait possible de le faire également avec les listes : `[x, y, z] = [1, 2, 3]`. Toutefois, cette syntaxe est alourdie par la présence des crochets. On préfèrera donc la première syntaxe avec les *tuples* sans parenthèse.

Remarque

Nous avons appelé l'opération `x, y, z = 1, 2, 3` affectation multiple pour signifier que l'on affectait des valeurs à plusieurs variables en même temps. Toutefois, vous pourrez rencontrer aussi l'expression *tuple unpacking* que l'on pourrait traduire par « désempaquetage de tuple ». Cela signifie que l'on décompose le *tuple* initial `1, 2, 3` en 3 variables différentes.

Nous avions croisé l'importance de l'affectation multiple dans le chapitre 9 *Fonctions* lorsqu'une fonction renvoyait plusieurs valeurs.

```
>>> def ma_fonction():
...     return 3, 14
...
>>> x, y = ma_fonction()
>>> print(x, y)
3 14
```

La syntaxe `x, y = ma_fonction()` permet de récupérer les 2 valeurs renvoyées par la fonction et de les affecter à la volée dans 2 variables différentes. Cela évite l'opération laborieuse de récupérer d'abord le tuple, puis de créer les variables en utilisant l'indiçage :

```
>>> resultat = ma_fonction()
>>> resultat
(3, 14)
>>> x = resultat[0]
>>> y = resultat[1]
>>> print(x, y)
3 14
```

Conseils

Lorsqu'une fonction renvoie plusieurs valeurs sous forme de tuple, ce sera bien sûr la forme `x, y = ma_fonction()` qui sera privilégiée.

Quand une fonction renvoie plusieurs valeurs mais que l'on ne souhaite pas les utiliser toutes dans la suite du code, on peut utiliser le nom de variable `_` (caractère *underscore*) pour indiquer que certaines valeurs ne nous intéressent pas :

```
>>> def ma_fonction():
...     return 1, 2, 3, 4
```

```
...
>>> x, _, y, _ = ma_fonction()
>>> x
1
>>> y
3
```

Cela envoie le message à celui qui lit le code « je me fiche des valeurs récupérées dans ces variables `_` ». Notez que l'on peut utiliser une ou plusieurs variables *underscores(s)*. Dans l'exemple ci-dessus, la 2e et la 4e variable renvoyées par la fonction seront ignorées dans la suite du code. Cela a le mérite d'éviter de polluer l'attention du lecteur du code.

Remarque

Dans l'interpréteur interactif, la variable `_` a une signication différente. Elle prend automatiquement la dernière valeur affichée :

```
>>> 3
3
>>> _
3
>>> "mésange"
'mésange'
>>> _
'mésange'
```

Attention, cela n'est vrai que dans l'interpréteur !

Remarque

Le caractère *underscore* (`_`) est couramment utilisé dans les noms de variable pour séparer les mots et être explicite, par exemple `seq_ADN` ou `liste_listes_residus`. On verra dans le chapitre 15 *Bonnes pratiques en programmation Python* que ce style de nommage est appelé *snake_case*. Toutefois, il faut éviter d'utiliser les *underscores* en début et/ou en fin de nom de variable (par exemple : `_var`, `var_`, `__var`, `__var__`). On verra au chapitre 19 *Avoir la classe avec les objets* que ces *underscores* ont aussi une signification particulière.

13.3.4 Tuples contenant des listes

Conseil : pour les débutants, vous pouvez passer cette rubrique.

On a vu que les tuples étaient **non modifiables**. Que se passe-t-il alors si on crée un tuple contenant des objets modifiables comme des listes ? Examinons le code suivant :

```
>>> l1 = [1, 2, 3]
>>> t = (l1, "Plouf")
>>> t
([1, 2, 3], 'Plouf')
>>> l1[0] = -15
>>> t[0].append(-632)
>>> t
([-15, 2, 3, -632], 'Plouf')
```

On voit que si on modifie un élément de la liste `l1` en ligne 5 ou bien qu'on ajoute un élément à `t[0]` en ligne 6, Python s'exécute et ne renvoie pas de message d'erreur. Or nous avions dit qu'un tuple était non

modifiable... Comment cela-est il possible ? Commençons d'abord par regarder comment les objets sont agencés avec *Python Tutor*.

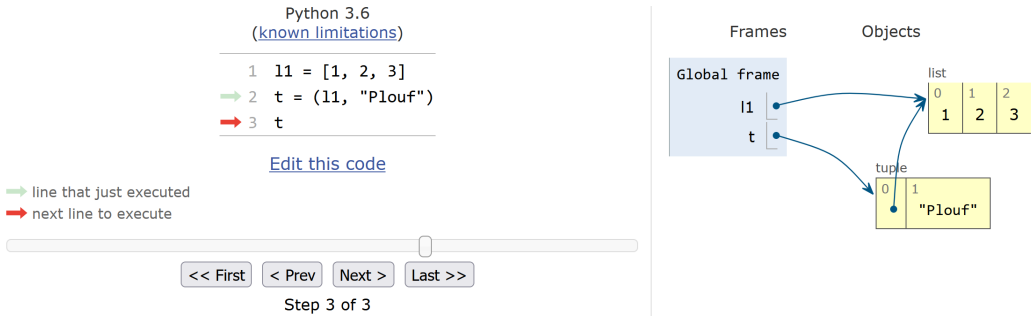

Figure 1. Tuple contenant une liste.

La liste `l1` pointe vers le même objet que l'élément du tuple d'indice 0. Comme pour la copie de liste (par exemple `liste1 = liste2`), ceci est attendu car par défaut Python crée une copie par référence (cf. Chapitre 11 *Plus sur les listes*). Donc, qu'on raisonne en tant que premier élément du tuple ou bien en tant que liste `l1`, on pointe vers **la même liste**. Or, rappelez-vous, au début de ce chapitre nous avons expliqué que lorsqu'on modifiait un élément d'une liste, celle-ci gardait le même identifiant. C'est toujours le cas ici, même si celle-ci se trouve dans un tuple. Regardons cela :

```
>>> l1 = [1, 2, 3]
>>> t = (l1, "Plouf")
>>> t
([1, 2, 3], 'Plouf')
>>> id(l1)
139971081980816
>>> id(t[0])
139971081980816
```

Nous confirmons ici le schéma de *Python Tutor*, c'est bien la même liste que l'on considère `l1` ou `t[0]` puisqu'on a le même identifiant. Maintenant, on modifie cette liste via la variable `l1` ou `t[0]` :

```
>>> l1[2] = -15
>>> t[0].append(-632)
>>> t
([1, 2, -15, -632], 'Plouf')
>>> id(l1)
139971081980816
>>> id(t[0])
139971081980816
```

Malgré la modification de cette liste, l'identifiant n'a toujours pas changé puisque la fonction `id()` nous renvoie toujours le même depuis le début. Ainsi, nous avons l'explication. Même si la liste a été modifiée « de l'intérieur », Python considère que c'est toujours la même liste puisqu'elle n'a pas changé d'identifiant. Si au contraire on essaie de remplacer cette sous-liste par autre chose, Python renvoie une erreur :

```
>>> t[0] = "Plif"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Ceci est dû au fait que le nouvel objet `"Plif"` n'a pas le même identifiant que la sous-liste initiale. En fait, l'immutabilité selon Python signifie qu'un objet créé doit toujours garder le même identifiant. Cela est valable

pour tout objet non modifiable, comme un élément d'un tuple, un caractère dans une chaîne de caractères, etc.

Conseils

Nous avons fait une petite digression ici afin que vous compreniez bien ce qu'il se passe lorsqu'on met une liste dans un tuple. Toutefois, pouvoir modifier une liste en tant qu'élément d'un tuple va à l'encontre de l'intérêt d'un objet non modifiable. Ainsi, dans la mesure du possible, nous vous déconseillons de créer des listes dans des tuples afin d'éviter les déconvenues.

13.3.5 Hachabilité des tuples

Conseil : pour les débutants, vous pouvez passer cette rubrique.

Les tuples sont hachables s'ils ne contiennent que des éléments hachables. Si un tuple contient un ou plusieurs objet(s) non hachable(s) comme une liste, il devient non hachable.

```
>>> t = tuple(range(10))
>>> t
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> hash(t)
-4181190870548101704
>>> t2 = ("Plouf", 2, (1, 3))
>>> t2
('Plouf', 2, (1, 3))
>>> hash(t2)
286288423668065022
>>> t3 = (1, (3, 4), "Plaf", [3, 4, 5])
>>> t3
(1, (3, 4), 'Plaf', [3, 4, 5])
>>> hash(t3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Les tuples `t` et `t2` sont hachables car ils ne contiennent que des éléments hachables. Par contre, `t3` ne l'est pas car un de ses éléments est une liste.

Conseils

Mettre une ou des liste(s) dans un tuple a cette autre conséquence néfaste de le rendre non hachable. Ceci le rend inutilisable comme clé de dictionnaire ou, on le verra ci-après, comme élément d'un *set* ou d'un *frozenset*. Donc, à nouveau, ne mettez pas de listes dans vos tuples !

13.4 *Sets* et *frozensets*

13.4.1 Définition et propriétés

Les objets de type *set* représentent un autre type de containers qui peut se révéler très pratique. Ils ont la particularité d'être modifiables, non hachables, non ordonnés, non indexables et de ne contenir qu'une seule copie maximum de chaque élément. Pour créer un nouveau *set* on peut utiliser les accolades :

```
>>> s = {4, 5, 5, 12}
>>> s
{12, 4, 5}
>>> type(s)
<class 'set'>
```

Remarquez que la répétition du 5 dans la définition du *set* en ligne 1 donne au final un seul 5 car chaque élément ne peut être présent qu'une seule fois. Comme pour les dictionnaires (jusqu'à la version 3.6), les *sets* sont non ordonnés. La manière dont Python les affiche n'a pas de sens en tant que tel et peut être différente de celle utilisée lors de leur création.

Les *sets* ne peuvent contenir que des objets **hachables**. On a déjà eu le cas avec les clés de dictionnaire. Ceci optimise l'accès à chaque élément du *set*. Pour rappel, les objets hachables que nous connaissons sont les chaînes de caractères, les tuples, les entiers, les *floats*, les booléens et les *frozensets* (cf. plus bas) ; les objets non hachables que l'on connait sont les listes, les *sets* et les dictionnaires. Si on essaie tout de même de mettre une liste dans un *set*, Python renvoie une erreur :

```
>>> s = {3, 4, "Plouf", (1, 3)}
>>> s
{(1, 3), 3, 4, 'Plouf'}
>>> s2 = {3.14, [1, 2]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

À quoi différencie-t-on un *set* d'un dictionnaire alors que les deux utilisent des accolades ? Le *set* sera défini seulement par des valeurs `{valeur_1, valeur_2, ...}` alors que le dictionnaire aura toujours des couples clé:valeur `{clé_1: valeur_1, clé_2: valeur_2, ...}`.

La fonction interne à Python `set()` convertit un objet itérable passé en argument en un nouveau *set* (opération de *casting*) :

```
>>> set([1, 2, 4, 1])
{1, 2, 4}
>>> set((2, 2, 2, 1))
{1, 2}
>>> set(range(5))
{0, 1, 2, 3, 4}
>>> set({"clé_1": 1, "clé_2": 2})
{'clé_1', 'clé_2'}
>>> set(["ti", "to", "to"])
{'ti', 'to'}
>>> set("Maître corbeau sur un arbre perché")
{'h', 'u', 'o', 'b', ' ', 'M', 'a', 'p', 'n', 'e', 'é', 'c', 'î', 's', 't', 'r'}
```

Nous avons dit plus haut que les *sets* ne sont pas ordonnés ni indexables, il est donc impossible de récupérer un élément par sa position. Il est également impossible de modifier un de ses éléments par l'indexation.

```
>>> s = set([1, 2, 4, 1])
>>> s[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
>>> s[1] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support item assignment
```

Par contre, les *sets* sont itérables :

```
>>> for element in s:
...     print(element)
...
1
2
4
```

Les *sets* ne peuvent être modifiés que par des méthodes spécifiques.

```
>>> s = set(range(5))
>>> s
{0, 1, 2, 3, 4}
>>> s.add(4)
>>> s
{0, 1, 2, 3, 4}
>>> s.add(472)
>>> s
{0, 1, 2, 3, 4, 472}
>>> s.discard(0)
>>> s
{1, 2, 3, 4, 472}
```

La méthode `.add()` ajoute au *set* l'élément passé en argument. Toutefois, si l'élément est déjà présent dans le *set*, il n'est pas ajouté puisqu'on a au plus une copie de chaque élément. La méthode `.discard()` retire du *set* l'élément passé en argument. Si l'élément n'est pas présent dans le *set*, il ne se passe rien, le *set* reste intact. Comme les *sets* ne sont pas ordonnés ni indexables, il n'y a pas de méthode pour insérer un élément à une position précise contrairement aux listes. Dernier point sur ces méthodes, elles modifient le *set* sur place (*in place* en anglais) et ne renvoient rien à l'instar des méthodes des listes (`.append()`, `.remove()`, etc.).

Enfin, les *sets* ne supportent pas les opérateurs `+` et `*`.

13.4.2 Utilité

Les containers de type *set* sont très utiles pour rechercher les éléments uniques d'une suite d'éléments. Cela revient à éliminer tous les doublons. Par exemple :

```
>>> import random
>>> liste = [random.randint(0, 9) for i in range(10)]
>>> liste
[7, 9, 6, 6, 7, 3, 8, 5, 6, 7]
>>> set(liste)
{3, 5, 6, 7, 8, 9}
```

On peut bien sûr transformer dans l'autre sens un *set* en liste. Cela permet par exemple d'éliminer les doublons de la liste initiale tout en récupérant une liste à la fin :

```
>>> list(set([7, 9, 6, 6, 7, 3, 8, 5, 6, 7]))
[3, 5, 6, 7, 8, 9]
```

On peut faire des choses très puissantes. Par exemple, un compteur de lettres en combinaison avec une liste de compréhension, le tout en une ligne !

```
>>> seq = "atctcgatcgatcgcgctagctagctcgccatacgtacgactacgt"
>>> set(seq)
```

```
{'c', 'g', 't', 'a'}
>>> [(base, seq.count(base)) for base in set(seq)]
[('c', 15), ('g', 10), ('t', 11), ('a', 10)]
```

Les *sets* permettent aussi l'évaluation d'union ou d'intersection mathématiques en conjonction avec les opérateurs respectivement `|` et `&` :

```
>>> liste_1 = [3, 3, 5, 1, 3, 4, 1, 1, 4, 4]
>>> liste_2 = [3, 0, 5, 3, 3, 1, 1, 1, 2, 2]
>>> set(liste_1) | set(liste_2)
{0, 1, 2, 3, 4, 5}
>>> set(liste_1) & set(liste_2)
{1, 3, 5}
```

Notez qu'il existe des méthodes permettant de réaliser ces opérations d'union et d'intersection :

```
>>> s1 = {1, 3, 4, 5}
>>> s2 = {0, 1, 2, 3, 5}
>>> s1.union(s2)
{0, 1, 2, 3, 4, 5}
>>> s1.intersection(s2)
{1, 3, 5}
```

L'instruction `s1.difference(s2)` renvoie sous la forme d'un nouveau *set* les éléments de `s1` qui ne sont pas dans `s2`. Et vice-versa pour `s2.difference(s1)`.

```
>>> s1.difference(s2)
{4}
>>> s2.difference(s1)
{0, 2}
```

Enfin, deux autres méthodes sont très utiles :

```
>>> s1 = set(range(10))
>>> s2 = set(range(3, 7))
>>> s3 = set(range(15, 17))
>>> s1
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> s2
{3, 4, 5, 6}
>>> s3
{16, 15}
>>> s2.issubset(s1)
True
>>> s3.isdisjoint(s1)
True
```

La méthode `.issubset()` indique si un *set* est inclus dans un autre *set*. La méthode `isdisjoint()` indique si un *set* est disjoint d'un autre *set*, c'est-à-dire, s'ils n'ont aucun élément en commun indiquant que leur intersection est nulle.

Il existe de nombreuses autres méthodes que nous n'abordons pas ici mais qui peuvent être consultées sur la documentation officielle de Python.

13.4.3 *Frozensets*

Les *frozensets* sont des *sets* non modifiables et hachables. Ainsi, un *set* peut contenir des *frozensets* mais pas l'inverse. A quoi servent-ils ? Comme la différence entre tuple et liste, l'immutabilité des *frozensets* donne l'assurance de ne pas pouvoir les modifier par erreur. Pour créer un *frozenset* on utilise la fonction interne `frozenset()` qui prend en argument un objet itérable et le convertit (opération de *casting*) :

```
>>> f1 = frozenset([3, 3, 5, 1, 3, 4, 1, 1, 4, 4])
>>> f2 = frozenset([3, 0, 5, 3, 3, 1, 1, 1, 2, 2])
>>> f1
frozenset({1, 3, 4, 5})
>>> f2
frozenset({0, 1, 2, 3, 5})
>>> f1.add(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
>>> f1.union(f2)
frozenset({0, 1, 2, 3, 4, 5})
>>> f1.intersection(f2)
frozenset({1, 3, 5})
```

Les *frozensets* ne possèdent bien sûr pas les méthodes de modification des *sets* (`.add()`, `.discard()`, etc.) puisqu'ils sont non modifiables. Par contre, ils possèdent toutes les méthodes de comparaisons de *sets* (`.union()`, `.intersection()`, etc.).

Conseils

Pour aller plus loin sur les *sets* et les *frozensets*, voici deux articles sur les sites programiz et towardsdatascience.

13.5 Récapitulation des propriétés des containers

Après ce tour d'horizon des différents containers, voici un tableau récapitulant leurs propriétés.

13.5.1 Objets séquentiels

Container	test d'appartenance et fonction `len()`	itérable	ordonné	indexable	modifiable	hacha
liste	oui	oui	oui	oui	oui	nor
chaîne de caractères	oui	oui	oui	oui	non	oui
range	oui	oui	oui	oui	non	oui
tuple	oui	oui	oui	oui	non	oui

* s'il ne contient que des objets hachables

13.5.2 Objects de *mapping*

Container	test d'appartenance et fonction `len()`	itérable	ordonné	indexable	modifiable	hacha
dictionnaire	oui	oui sur les clés	oui*	non	oui	nor

* à partir de Python 3.7 uniquement

13.5.3 Objets *sets*

Container	test d'appartenance et fonction `len()`	itérable	ordonné	indexable	modifiable	hacha
sets	oui	oui	non	non	oui	nor
frozensets	oui	oui	non	non	non	oui

13.5.4 Types de base

Il est aussi intéressant de comparer ces propriétés avec celles des types numériques de base qui ne sont pas des containers.

Objet numérique	test d'appartenance et fonction `len()`	itérable	ordonné	indexable	modifiable	hacha
entier	non	non	non	non	non	oui
float	non	non	non	non	non	oui
booléen	non	non	non	non	non	oui

13.6 Dictionnaires et *sets* de compréhension

Conseil : pour les débutants, vous pouvez passer cette rubrique.

Nous avons vu au chapitre 11 *Plus sur les listes* les listes de compréhension. Il est également possible de générer des dictionnaires de compréhension :

```
>>> dico = {"a": 10, "g": 10, "t": 11, "c": 15}
>>> dico.items()
dict_items([('a', 10), ('g', 10), ('t', 11), ('c', 15)])
>>> {key:val*2 for key, val in dico.items()}
{'a': 20, 'g': 20, 't': 22, 'c': 30}
>>>
>>> animaux = (("singe", 3), ("girafe", 1), ("rhinocéros", 1), ("gazelle", 4))
>>> {ani:nb for ani, nb in animaux}
{'singe': 3, 'girafe': 1, 'rhinocéros': 1, 'gazelle': 4}
```

Avec un dictionnaire de compréhension, on peut rapidement compter le nombre de chaque base dans une séquence d'ADN :

```
>>> seq = "atctcgatcgatcgcgctagctagctcgccatacgtacgactacgt"
>>> {base:seq.count(base) for base in set(seq)}
{'a': 10, 'g': 10, 't': 11, 'c': 15}
```

De manière générale, tout objet sur lequel on peut faire une double itération du type `for var1, var2 in obj` est utilisable pour créer un dictionnaire de compréhension. Si vous souhaitez aller plus loin, vous pouvez consulter cet article sur le site *Datacamp*.

Il est également possible de générer des *sets* de compréhension sur le même modèle que les listes de compréhension :

```
>>> {i for i in range(10)}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> {i**2 for i in range(10)}
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
>>>
>>> animaux = (("singe", 3), ("girafe", 1), ("rhinocéros", 1), ("gazelle", 4))
>>> {ani for ani, _ in animaux}
{'rhinocéros', 'gazelle', 'singe', 'girafe'}
```

13.7 Module *collections*

Conseil : pour les débutants, vous pouvez passer cette rubrique.

Le module *collections* contient d'autres types de *containers* qui peuvent se révéler utiles, c'est une véritable mine d'or ! Nous n'aborderons pas tous ces objets ici, mais nous pouvons citer tout de même certains d'entre eux si vous souhaitez aller un peu plus loin :

- les dictionnaires ordonnés qui se comportent comme les dictionnaires classiques mais qui sont ordonnés ;
- les *defaultdicts* permettant de générer des valeurs par défaut quand on demande une clé qui n'existe pas (cela évite que Python génère une erreur) ;
- les compteurs dont un exemple est montré ci-dessous ;
- les *namedtuples* que nous évoquerons au chapitre 19 *Avoir la classe avec les objets*.

L'objet `collection.Counter()` est particulièrement intéressant et simple à utiliser. Il crée des compteurs à partir d'objets itérables, par exemple :

```
>>> import collections
>>> compo_seq = collections.Counter("aatctccgatcgatcgatcgatgatc")
>>> compo_seq
Counter({'a': 7, 't': 7, 'c': 7, 'g': 5})
>>> type(compo_seq)
<class 'collections.Counter'>
>>> compo_seq["a"]
7
>>> compo_seq["n"]
0
```

Dans cet exemple, Python a automatiquement compté chaque caractère `a`, `t`, `g` et `c` de la chaîne de caractères passée en argument. Cela crée un objet de type `Counter` qui se comporte ensuite comme un dictionnaire, à une exception près : si on appelle une clé qui n'existe pas dans l'itérable initiale (comme le `n` ci-dessus) la valeur renvoyée est 0.

13.8 Exercices

Conseil : pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

13.8.1 Composition en acides aminés

En utilisant un dictionnaire, déterminez le nombre d'occurrences de chaque acide aminé dans la séquence `AGWPSGGASAGLAILWGASAIMPGALW`. Le dictionnaire ne doit contenir que les acides aminés présents dans la séquence.

13.8.2 Mots de 2 et 3 lettres dans une séquence d'ADN

Créez une fonction `compte_mots_2_lettres()` qui prend comme argument une séquence sous la forme d'une chaîne de caractères et qui renvoie tous les mots de 2 lettres qui existent dans la séquence sous la forme d'un dictionnaire. Par exemple pour la séquence `ACCTAGCCCTA`, le dictionnaire renvoyée serait : `{'AC': 1, 'CC': 3, 'CT': 2, 'TA': 2, 'AG': 1, 'GC': 1}`

Créez une nouvelle fonction `compte_mots_3_lettres()` qui a un comportement similaire à `compte_mots_2_lettres()` mais avec des mots de 3 lettres.

Utilisez ces fonctions pour affichez les mots de 2 et 3 lettres et leurs occurrences trouvés dans la séquence d'ADN :
`ACCTAGCCATGTAGAATCGCCTAGGCTTTAGCTAGCTCTAGCTAGCTG`

Voici un exemple de sortie attendue :

```
Mots de 2 lettres
AC : 1
CC : 3
CT : 8
[...]
Mots de 3 lettres
ACC : 1
CCT : 2
```

```
CTA : 5
[...]
```

13.8.3 Mots de 2 lettres dans la séquence du chromosome I de *Saccharomyces cerevisiae*

Créez une fonction `lit_fasta()` qui prend comme argument le nom d'un fichier FASTA sous la forme d'une chaîne de caractères, lit la séquence dans le fichier FASTA et la renvoie sous la forme d'une chaîne de caractères. N'hésitez pas à vous inspirer d'un exercice similaire du chapitre 10 *Plus sur les chaînes de caractères*.

Utilisez cette fonction et la fonction `compte_mots_2_lettres()` de l'exercice précédent pour extraire les mots de 2 lettres et leurs occurrences dans la séquence du chromosome I de la levure du boulanger *Saccharomyces cerevisiae* (fichier `NC_001133.fna`).

Le génome complet est fourni au format FASTA. Vous trouverez des explications sur ce format et des exemples de code dans l'annexe A *Quelques formats de données rencontrés en biologie*.

13.8.4 Mots de *n* lettres dans un fichier FASTA

Créez un script `extract-words.py` qui prend comme arguments le nom d'un fichier FASTA suivi d'un entier compris entre 1 et 4. Ce script doit extraire du fichier FASTA tous les mots et leurs occurrences en fonction du nombre de lettres passé en option.

Utilisez pour ce script la fonction `lit_fasta()` de l'exercice précédent. Créez également la fonction `compte_mots_n_lettres()` qui prend comme argument une séquence sous la forme d'une chaîne de caractères et le nombre de lettres des mots sous la forme d'un entier.

Testez ce script avec :

- la séquence du chromosome I de la levure du boulanger *Saccharomyces cerevisiae* (fichier `NC_001133.fna`)
- le génome de la bactérie *Escherichia coli* (fichier `NC_000913.fna`)

Les deux fichiers sont au format FASTA.

Cette méthode vous paraît-elle efficace sur un génome assez gros comme celui d'*Escherichia coli* ?

13.8.5 Atomes carbone alpha d'un fichier PDB

Téléchargez le fichier `1bta.pdb` qui correspond à la structure tridimensionnelle de la protéine barstar sur le site de la *Protein Data Bank* (PDB).

Créez la fonction `trouve_calpha()` qui prend en argument le nom d'un fichier PDB (sous la forme d'une chaîne de caractères), qui sélectionne uniquement les lignes contenant des carbones alpha et qui les renvoie sous la forme d'une liste de dictionnaires. Chaque dictionnaire contient quatre clés :

- le numéro du résidu (`resid`) avec une valeur entière,
- la coordonnée atomique *x* (`x`) avec une valeur *float*,
- la coordonnée atomique *y* (`y`) avec une valeur *float*,
- la coordonnée atomique *z* (`z`) avec une valeur *float*.

Utilisez la fonction `trouve_calpha()` pour afficher à l'écran le nombre total de carbones alpha de la barstar ainsi que les coordonnées atomiques des carbones alpha des deux premiers résidus (acides aminés).

Conseil : vous trouverez des explications sur le format PDB et des exemples de code pour lire ce type de fichier en Python dans l'annexe A *Quelques formats de données rencontrés en biologie*.

13.8.6 Barycentre d'une protéine (exercice +++)

Téléchargez le fichier `1bta.pdb` qui correspond à la structure tridimensionnelle de la protéine barstar sur le site de la *Protein Data Bank* (PDB).

Un carbone alpha est présent dans chaque résidu (acide aminé) d'une protéine. On peut obtenir une bonne approximation du barycentre d'une protéine en calculant le barycentre de ses carbones alpha.

Le barycentre G de coordonnées (G_x, G_y, G_z) est obtenu à partir des n carbones alpha (CA) de coordonnées $(\mathrm{CA}_x, \mathrm{CA}_y, \mathrm{CA}_z)$ avec :

$$G_x = \frac{1}{n}\sum_{i=1}^{n} \mathrm{CA}_{i,x}$$

$$G_y = \frac{1}{n}\sum_{i=1}^{n} \mathrm{CA}_{i,y}$$

$$G_z = \frac{1}{n}\sum_{i=1}^{n} \mathrm{CA}_{i,z}$$

Créez une fonction `calcule_barycentre()` qui prend comme argument une liste de dictionnaires dont les clés (`resid`, `x`, `y` et `z`) sont celles de l'exercice précédent et qui renvoie les coordonnées du barycentre sous la forme d'une liste de *floats*.

Utilisez la fonction `trouve_calpha()` de l'exercice précédent et la fonction `calcule_barycentre()` pour afficher, avec deux chiffres significatifs, les coordonnées du barycentre des carbones alpha de la barstar.