

Beste de savoir

Notions de Python avancées

15 janvier 2019

Table des matières

I. Introduction	6
II. Mise en bouche	8
0.1. À table !	9
1. Conteneurs	10
1.1. Les conteneurs, c'est in	10
1.2. C'est pas la taille qui compte	11
1.3. Objets indexables	12
1.4. Les slices	13
1.5. Module collections	15
1.5.1. <code>namedtuple</code>	15
1.5.2. <code>deque</code>	16
1.5.3. <code>ChainMap</code>	17
1.5.4. <code>Counter</code>	18
1.5.5. <code>OrderedDict</code>	19
1.5.6. <code>defaultdict</code>	20
1.5.7. <i>Wrappers</i>	20
1.6. TP : Une liste chaînée en Python	21
1.6.1. Présentation	21
1.6.2. Bases	21
1.6.3. <code>append</code> et <code>insert</code>	22
1.6.4. <code>__contains__</code>	23
1.6.5. <code>__len__</code>	23
1.6.6. <code>__getitem__</code> et <code>__setitem__</code>	24
1.6.7. Aller plus loin	25
2. Itérables	26
2.1. <code>for</code> lointain	26
2.1.1. Le cas des indexables	27
2.2. Utilisation des itérables	28
2.2.1. Python et les itérables	28
2.2.2. Retour sur <code>iter</code>	30
2.3. Utilisation avancée : le module <code>itertools</code>	30
2.4. L'unpacking	31
2.4.1. Structures imbriquées	32
2.4.2. Opérateur <i>splat</i>	33
2.4.3. Encore du <i>splat</i>	34
2.5. TP : Itérateur sur listes chaînées	34

3. Objets mutables et hashables	36
3.1. Mutables	36
3.2. Égalité et identité	38
3.2.1. Quel opérateur utiliser ?	39
3.3. Hashables	40
3.3.1. Le condensat	40
3.3.2. La fonction <code>hash</code>	40
3.3.3. À quoi servent-ils ?	41
3.3.4. Implémentation	42
3.4. TP : Égalité entre listes	43
III. Ainsi font fonctions	45
3.5. Trois petits tours et puis s'en vont	46
4. Callables	47
4.1. Fonctions, classes et lambdas	47
4.2. Paramètres de fonctions	48
4.2.1. Paramètres et arguments	48
4.2.2. Opérateur <i>splat</i> , le retour	49
4.2.3. Le double- <i>splat</i>	51
4.2.4. L'appel du <i>splat</i>	51
4.3. Call-me maybe	53
4.4. Utilisation des callables	54
4.5. Modules <code>operator</code> et <code>functools</code>	55
4.5.1. <code>operator</code>	56
4.5.2. <code>functools</code>	57
4.6. TP : <code>itemgetter</code>	58
5. Annotations et signatures	60
5.1. Annotations	60
5.1.1. Utilité des annotations	61
5.1.2. Des types plus complexes (module <code>typing</code>)	61
5.1.3. Annotations de variables	62
5.2. Inspecteur Gadget	63
5.2.1. Module <code>inspect</code>	63
5.3. Signatures	65
5.3.1. Arguments préparés	66
5.3.2. Méthode <code>replace</code>	68
5.4. TP : Vérification de signature	69
5.4.1. Prototype	69
5.4.2. Paramètres	70
5.4.3. Résultat	72
6. Décorateurs	75
6.1. D&CO, une semaine pour tout changer	75
6.2. Décorateurs paramétrés	77
6.3. Envelopper une fonction	78
6.3.1. Envelopper des fonctions	79

6.4. TP : Arguments positionnels	80
6.4.1. Réécriture de la signature	80
6.4.2. Décorateur paramétré	81
IV. Plus loin, un peu plus loin	84
6.5. Au-delà des mers	85
7. Générateurs	86
7.1. Dessine-moi un générateur	86
7.1.1. Le mot-clef <code>yield</code>	86
7.2. Altérer un générateur avec <code>send</code>	88
7.3. Méthodes des générateurs	91
7.3.1. <code>throw</code>	91
7.3.2. <code>close</code>	93
7.4. Déléguer à un autre générateur avec <code>yield from</code>	94
7.5. Listes et générateurs en intension	96
7.5.1. Listes en intension	96
7.5.2. Générateurs en intension	97
7.6. Liste ou générateur ?	98
7.7. TP : <code>map</code>	99
8. Gestionnaires de contexte	104
8.1. <code>with</code> or without you	104
8.2. La fonction <code>open</code>	105
8.3. Fonctionnement interne	106
8.4. Simplifions-nous la vie avec <code>contextlib</code>	106
8.5. Réutilisabilité et réentrance	109
8.5.1. Réutilisabilité	109
8.5.2. Réentrance	110
8.6. TP : Redirection de sortie (<code>redirectstdout</code>)	111
9. Accesseurs et descripteurs	114
9.1. L'attribut de Dana	114
9.1.1. <code>dict</code> et <code>slots</code>	115
9.1.2. MRO	116
9.2. Les descripteurs	118
9.2.1. La méthode <code>__set_name__</code>	119
9.3. Les propriétés	120
9.4. Les méthodes	121
9.5. TP : Méthodes	123
V. La rentrée des classes	125
9.6. Et de leurs parents	126
10. Types	127
10.1. Instance, classe et métaclasse	127
10.1.1. Caractéristiques des classes	128

10.2. Le vrai constructeur	128
10.2.1. Le cas des immutables	129
10.3. Paramètres d'héritage	130
10.4. TP : Liste immutable	132
11. Métaclases	135
11.1. Quel est donc ce type ?	135
11.2. Les métaclases	136
11.2.1. À quoi sert une métaclasse ?	136
11.2.2. Notre première métaclasse	136
11.2.3. Préparation de la classe	138
11.2.4. Une métaclasse utile	138
11.3. Utiliser une fonction comme métaclasse	140
11.4. TP : Types immutables	141
11.4.1. Hériter de <code>tuple</code>	142
11.4.2. Les slots à la rescousse	142
11.4.3. Le problème des méthodes de <code>tuple</code>	143
12. Classes abstraites	145
12.1. Module <code>abc</code>	145
12.2. <code>isinstance</code>	146
12.3. <code>issubclass</code>	148
12.3.1. Le cas des classes <code>ABC</code>	150
12.4. Collections abstraites	151
12.5. TP : Reconnaissance d'interfaces	153
VI. Pour quelques exercices de plus	156
12.6. L'histoire sans fin	157
13. Décorateurs	158
13.1. Vérification de types	158
13.2. Mémoïsation	160
13.2.1. Signatures	162
13.3. Fonctions génériques	163
13.4. Récursivité terminale	164
14. Gestionnaires de contexte	168
14.1. Changement de répertoire	168
14.2. Transformer un générateur en gestionnaire de contexte	169
14.3. Suppression d'erreurs	175
15. Accesseurs et descripteurs	177
15.1. Propriétés	177
16. Métaclases	182
16.1. Évaluation paresseuse	182
16.1.1. L'évaluation paresseuse, c'est quoi ?	182
16.1.2. Objectif du TP	182

Table des matières

16.1.3. Opérateurs et méthodes spéciales	183
16.2. Types immutables	186
16.2.1. Objets <code>super</code>	187
16.2.2. <code>ImmutableMeta</code>	188
16.2.3. Améliorations	191

VII. Conclusion **192**

% NOTIONS DE PYTHON AVANCÉES % entwanne % 20 février 2018

Première partie

Introduction

I. Introduction

Python est simple.

C'est probablement ce que l'on vous a dit de nombreuses fois, et ce que vous avez constaté en apprenant et pratiquant ce langage. Mais derrière cette simplicité apparente existent un certain nombre de concepts plus complexes qui forment la puissance de ce langage.

En Python, on s'intéresse plus au comportement des objets qu'à leur nature. Ainsi, l'interface des objets (c'est-à-dire l'ensemble de leurs attributs et méthodes) est quelque chose de très important, c'est entre autres ce qui les définit.

En effet, une grande partie des outils du langage sont génériques — tels les appels de fonctions ou les boucles `for` — c'est-à-dire qu'ils peuvent s'appliquer à des types différents. Python demande simplement à ces types de respecter une interface en implémentant un certain nombre de méthodes spéciales. Ces interfaces et méthodes seront décrites dans le cours.

Le pré-requis pour suivre ce tutoriel est de connaître Python, même à un niveau intermédiaire. Il est simplement nécessaire de savoir manipuler les structures du langage (conditions, boucles, fonctions), les types de base (nombres, chaînes de caractères, listes, dictionnaires), et d'avoir des notions de [programmation objet en Python](#) [↗](#). Connaître le mécanisme des exceptions est un plus.

Ce cours se divise en chapitres consacrés chacun à une spécificité du langage. Ces dernières ne devraient plus avoir de secret pour vous une fois la lecture terminée.

Je tiens aussi à préciser que ce cours s'adresse principalement aux utilisateurs de Python 3, et n'est pas valable pour les versions de Python inférieures.

Deuxième partie

Mise en bouche

II. Mise en bouche

0.1. À table !

Quoi de mieux pour commencer qu'une petite mise en bouche ?

Nous allons nous intéresser dans cette partie aux plus simples interfaces du langage. Celles qui nous permettent de confectionner des objets tels que les listes ou les dictionnaires.

1. Conteneurs

En Python, on appelle conteneur (*container*) un objet ayant vocation à en contenir d'autres, comme les chaînes de caractères, les listes, les ensembles ou les dictionnaires.

Il existe plusieurs catégories de conteneurs, notamment celle des *subscriptables*. Ce nom barbare regroupe tous les objets sur lesquels l'opérateur `[]` peut être utilisé. L'ensemble des types cités dans le premier paragraphe sont *subscriptables*, à l'exception de l'ensemble (*set*), qui n'implémente pas l'opération `[]`.

Les *subscriptables* se divisent en deux nouvelles catégories non exclusives : les *indexables* et les *sliceables*. Les premiers sont ceux pouvant être indexés avec des nombres entiers, les seconds pouvant l'être avec des `slice` (voir plus loin).

On parle plus généralement de séquence quand un conteneur est *indexable* et *sliceable*.

Une autre catégorie importante de conteneurs est formée par les *mappings* : il s'agit des objets qui associent des valeurs à des clefs, comme le font les dictionnaires.

Une séquence et un *mapping* se caractérisent aussi par le fait qu'ils possèdent une taille, comme nous le verrons par la suite dans ce chapitre.

1.1. Les conteneurs, c'est in

Comme indiqué, les conteneurs sont donc des objets qui contiennent d'autres objets. Ils se caractérisent par l'opérateur `in` : `(0, 1, 2, 3)` étant un conteneur, il est possible de tester s'il contient telle ou telle valeur à l'aide de cet opérateur.

```
1 >>> 3 in (0, 1, 2, 3)
2 True
3 >>> 4 in (0, 1, 2, 3)
4 False
```

Comment cela fonctionne ? C'est très simple. Comme pour de nombreux comportements, Python se base sur des méthodes spéciales des objets. Vous en connaissez déjà probablement, ce sont les méthodes dont les noms débutent et s'achèvent par `__`.

Ici, l'opérateur `in` fait simplement appel à la méthode `__contains__` de l'objet, qui prend en paramètre l'opérande gauche, et retourne un booléen.

II. Mise en bouche

```
1 >>> 'o' in 'toto'
2 True
3 >>> 'toto'.__contains__('o')
4 True
```

Il nous suffit ainsi d'implémenter cette méthode pour faire de notre objet un conteneur.

```
1 >>> class MyContainer:
2 ...     def __contains__(self, value):
3 ...         return value is not None # contient tout sauf None
4 ...
5 >>> 'salut' in MyContainer()
6 True
7 >>> 1.5 in MyContainer()
8 True
9 >>> None in MyContainer()
10 False
```

1.2. C'est pas la taille qui compte

Un autre point commun partagé par de nombreux conteneurs est qu'ils possèdent une taille. C'est-à-dire qu'ils contiennent un nombre fini et connu d'éléments, et peuvent être passés en paramètre à la fonction `len` par exemple.

```
1 >>> len([1, 2, 3])
2 3
3 >>> len(MyContainer())
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6 TypeError: object of type 'MyContainer' has no len()
```

Comme pour l'opérateur `in`, la fonction `len` fait appel à une méthode spéciale de l'objet, `__len__`, qui ne prend ici aucun paramètre et doit retourner un nombre entier positif.

```
1 >>> len('toto')
2 4
3 >>> 'toto'.__len__()
4 4
```

Nous pouvons donc aisément donner une taille à nos objets :

```
1 >>> class MySizeable:
2 ...     def __len__(self):
3 ...         return 18
4 ...
5 >>> len(MySizeable())
6 18
```

Je vous invite à faire des essais en retournant d'autres valeurs (nombres négatifs, flottants, chaînes de caractères) pour observer le comportement.

1.3. Objets indexables

Nous voilà bien avancés : nous savons mesurer la taille d'un objet, mais pas voir les éléments qu'il contient.

L'accès aux éléments se fait via l'opérateur []. De même que la modification et la suppression, quand celles-ci sont possibles (c'est-à-dire que l'objet est mutable).

```
1 >>> numbers = [4, 7, 6]
2 >>> numbers[2]
3 6
4 >>> numbers[1] = 5
5 >>> numbers
6 [4, 5, 6]
7 >>> del numbers[0]
8 >>> numbers
9 [5, 6]
```

Le comportement interne est ici régi par 3 méthodes : `__getitem__`, `__setitem__`, et `__delitem__`.

```
1 >>> numbers = [4, 7, 6]
2 >>> numbers.__getitem__(2)
3 6
4 >>> numbers.__setitem__(1, 5)
5 >>> numbers
6 [4, 5, 6]
7 >>> numbers.__delitem__(0)
8 >>> numbers
9 [5, 6]
```

Comme précédemment, nous pouvons donc implémenter ces méthodes dans un nouveau type. Nous allons ici nous contenter de faire un proxy autour d'une liste existante.

II. Mise en bouche

Un proxy¹ est un objet prévu pour se substituer à un autre, il doit donc répondre aux mêmes méthodes, de façon transparente.

```
1 class MyList:
2     def __init__(self, value=()): # Émulation du constructeur de
      list
3         self.internal = list(value)
4
5     def __len__(self): # Sera utile pour les tests
6         return len(self.internal)
7
8     def __getitem__(self, key):
9         return self.internal[key] # Équivalent à return
      self.internal.__getitem__(key)
10
11    def __setitem__(self, key, value):
12        self.internal[key] = value
13
14    def __delitem__(self, key):
15        del self.internal[key]
```

Nous pouvons tester notre objet, celui-ci a bien le comportement voulu :

```
1 >>> numbers = MyList('123456')
2 >>> len(numbers)
3 6
4 >>> numbers[1]
5 '2'
6 >>> numbers[1] = '0'
7 >>> numbers[1]
8 '0'
9 >>> del numbers[1]
10 >>> len(numbers)
11 5
12 >>> numbers[1]
13 '3'
```

1.4. Les slices

Les « slices » se traduisent en français par « tranches ». Cela signifie que l'on va prendre notre objet et le découper en morceaux. Par exemple, récupérer la première moitié d'une liste, ou cette même liste en ne conservant qu'un élément sur deux.

1. [https://fr.wikipedia.org/wiki/Proxy_\(patron_de_conception\)](https://fr.wikipedia.org/wiki/Proxy_(patron_de_conception)) ↗

II. Mise en bouche

Les *slices* sont une syntaxe particulière pour l'indexation, à l'aide du caractère `:` lors des appels à `[]`.

```
1 >>> letters = ('a', 'b', 'c', 'd', 'e', 'f')
2 >>> letters[0:4]
3 ('a', 'b', 'c', 'd')
4 >>> letters[1:-2]
5 ('b', 'c', 'd')
6 >>> letters[:2]
7 ('a', 'c', 'e')
```

Je pense que vous êtes déjà familier avec cette syntaxe. Le *slice* peut prendre jusqu'à 3 nombres :

- Le premier est l'indice de départ (0 si omis) ;
- Le second est l'indice de fin (fin de la liste si omis), l'élément correspondant à cet indice étant exclu ;
- Le dernier est le pas, le nombre d'éléments passés à chaque itération (1 par défaut) ;

Notons que les *slices* peuvent aussi servir pour la modification et la suppression :

```
1 >>> letters = ['a', 'b', 'c', 'd', 'e', 'f']
2 >>> letters[:2] = 'x', 'y', 'z'
3 >>> letters
4 ['x', 'b', 'y', 'd', 'z', 'f']
5 >>> del letters[0:3]
6 >>> letters
7 ['d', 'z', 'f']
```

Une bonne chose pour nous est que, même avec les *slices*, ce sont les 3 mêmes méthodes `__getitem__`, `__setitem__` et `__delitem__` qui sont appelées lors des accès. Cela signifie que la classe `MyList` que nous venons d'implémenter est déjà compatible avec les *slices*.

En fait, c'est simplement que le paramètre `key` passé ne représente pas un nombre, mais est un objet de type `slice` :

```
1 >>> s = slice(1, -1)
2 >>> 'abcdef'[s]
3 'bcde'
4 >>> 'abcdef'[slice(None, None, 2)]
5 'ace'
```

Comme vous le voyez, le *slice* se construit toujours de la même manière, avec 3 nombres pouvant être omis, ou précisés à `None` pour prendre leur valeur par défaut.

L'objet ainsi construit contient 3 attributs : `start`, `stop`, et `step`.


```
1 >>> s = slice(1, 2, 3)
2 >>> s.start
3 1
4 >>> s.stop
5 2
6 >>> s.step
7 3
```

Je vous conseille ce tutoriel de [pascal.ortiz](https://zestedesavoir.com/tutoriels/582/les-slices-en-python/) pour en savoir plus sur les slices : <https://zestedesavoir.com/tutoriels/582/les-slices-en-python/>

1.5. Module collections

Avant d'en terminer avec les conteneurs, je voulais vous présenter le `module collections`. Ce module, parfois méconnu, comprend différents conteneurs utiles de la bibliothèque standard, que je vais essayer de vous présenter brièvement.

1.5.1. `namedtuple`

Les *tuples* nommés (ou *named tuples*), sont très semblables aux *tuples*, dont ils héritent. Ils ajoutent simplement la possibilité de référencer les champs du *tuple* par un nom plutôt que par un index, pour gagner en lisibilité.

On instancie `namedtuple` pour créer un nouveau type de *tuples* nommés. `namedtuple` prend en paramètre le nom du type et la liste des noms de champs.

```
1 >>> from collections import namedtuple
2 >>> Point2D = namedtuple('Point2D', ['x', 'y'])
3 >>> p = Point2D(3, 5)
4 >>> p.x
5 3
6 >>> p.y
7 5
8 >>> p[0]
9 3
10 >>> p
11 Point2D(x=3, y=5)
12 >>> Point2D(x=1, y=2)
13 Point2D(x=1, y=2)
```

II. Mise en bouche

1.5.2. deque

Les *queues* à deux extrémités (ou *double-ended queues* contracté en *deques*), sont des objets proches des listes de Python, mais avec une structure interne différente.

Plutôt qu'avoir un tableau d'éléments, les éléments sont vus comme des maillons liés les uns aux autres.

L'intérêt des *deques* par rapport aux listes est d'offrir de meilleures performances pour l'insertion/suppression d'éléments en tête et queue de liste, mais moins bonnes pour l'accès à un élément en milieu de liste. Elles peuvent donc être indiquées pour gérer des piles ou des files.

Pour bénéficier de ces optimisations, les *deques* sont pourvues de méthodes `appendleft`, `extendleft` et `popleft` qui travaillent sur l'extrémité gauche de la séquence, en plus des habituelles `append`/`extend`/`pop` qui travaillent sur celle de droite.

```
1 >>> from collections import deque
2 >>> d = deque([1, 2, 3])
3 >>> d
4 deque([1, 2, 3])
5 >>> d.append(4)
6 >>> d.appendleft(0)
7 >>> d
8 deque([0, 1, 2, 3, 4])
9 >>> d.popleft()
10 0
11 >>> d.popleft()
12 1
13 >>> d.extendleft([1, 0])
14 >>> d
15 deque([0, 1, 2, 3, 4])
16 >>> d[0], d[1], d[-1]
17 (0, 1, 4)
18 >>> d[-1] = 5
19 >>> d
20 deque([0, 1, 2, 3, 5])
```

Les *deques* ont aussi la possibilité d'être limitées en taille, en supprimant les éléments les plus à droite lors d'une insertion à gauche et inversement. Cela permet la réalisation de tampons circulaires.

```
1 >>> d = deque([], 2)
2 >>> d
3 deque([], maxlen=2)
4 >>> d.append(1)
5 >>> d.append(2)
6 >>> d
7 deque([1, 2], maxlen=2)
```

II. Mise en bouche

```
8 >>> d.append(3)
9 >>> d
10 deque([2, 3], maxlen=2)
11 >>> d.appendleft(1)
12 >>> d
13 deque([1, 2], maxlen=2)
14 >>> d.maxlen
15 2
```

1.5.3. ChainMap

Les `ChainMap` sont des structures de données permettant de grouper (chaîner) plusieurs dictionnaires (ou *mappings*) sans les fusionner, ce qui leur permet de se tenir à jour. Elles se comportent comme des dictionnaires et s'occupent de rechercher les éléments dans les *mappings* qui lui ont été donnés à la construction. Lors de l'insertion de nouveaux éléments, ceux-ci sont ajoutés au premier *mapping*.

```
1 >>> from collections import ChainMap
2 >>> d1 = {'a': 0, 'b': 1}
3 >>> d2 = {'b': 2, 'c': 3}
4 >>> d3 = {'d': 4}
5 >>> c = ChainMap(d1, d2, d3)
6 >>> c
7 ChainMap({'b': 1, 'a': 0}, {'c': 3, 'b': 2}, {'d': 4})
8 >>> c['a'], c['b'], c['c'], c['d']
9 (0, 1, 3, 4)
10 >>> c['e'] = 5
11 >>> c
12 ChainMap({'e': 5, 'b': 1, 'a': 0}, {'c': 3, 'b': 2}, {'d': 4})
13 >>> d1
14 {'e': 5, 'b': 1, 'a': 0}
15 >>> d2['f'] = 6
16 >>> c['f']
17 6
```

Les `ChainMap` ajoutent quelques fonctionnalités pratiques :

- L'attribut `maps` pour obtenir la liste des *mappings* ;
- La méthode `new_child` pour créer un nouveau `ChainMap` à partir de l'actuel, ajoutant un dictionnaire à gauche ;
- L'attribut `parents` pour obtenir les « parents » de l'actuel, c'est-à-dire le `ChainMap` composé des mêmes *mappings* excepté le plus à gauche.

II. Mise en bouche

```
1 >>> c.maps
2 [{ 'e': 5, 'b': 1, 'a': 0 }, { 'c': 3, 'b': 2, 'f': 6 }, { 'd': 4 }]
3 >>> c.new_child()
4 ChainMap({}, { 'e': 5, 'b': 1, 'a': 0 }, { 'c': 3, 'b': 2, 'f': 6 },
5           { 'd': 4 })
6 >>> c.new_child({ 'a': 7 })
7 ChainMap({ 'a': 7 }, { 'e': 5, 'b': 1, 'a': 0 }, { 'c': 3, 'b': 2, 'f':
8           6 }, { 'd': 4 })
9 >>> c.parents
10 ChainMap({ 'c': 3, 'b': 2, 'f': 6 }, { 'd': 4 })
```

Les `ChainMap` se révèlent alors très utiles pour gérer des contextes / espaces de noms imbriqués, comme [présenté dans la documentation](#) .

1.5.4. Counter

Les compteurs (`Counter`) sont des dictionnaires un peu spéciaux qui servent à compter les éléments.

```
1 >>> from collections import Counter
2 >>> c = Counter([1, 2, 3, 1, 3, 1, 5])
3 >>> c
4 Counter({1: 3, 3: 2, 2: 1, 5: 1})
5 >>> c[3]
6 2
7 >>> c[4]
8 0
9 >>> c[5] += 1
10 >>> c
11 Counter({1: 3, 3: 2, 5: 2, 2: 1})
12 >>> c + Counter({1: 2, 2: 3})
13 Counter({1: 5, 2: 4, 3: 2, 5: 2})
14 >>> c - Counter({1: 2, 2: 3})
15 Counter({3: 2, 5: 2, 1: 1})
```

On retrouve quelques méthodes utiles pour manipuler les compteurs.

```
1 >>> list(c.elements())
2 [1, 1, 1, 2, 3, 3, 5, 5]
3 >>> c.most_common()
4 [(1, 3), (3, 2), (5, 2), (2, 1)]
5 >>> c.most_common(2)
6 [(1, 3), (3, 2)]
7 >>> c.update({5: 1})
```

```
8 >>> c
9 Counter({1: 3, 5: 3, 3: 2, 2: 1})
10 >>> c.subtract({2: 4})
11 >>> c
12 Counter({1: 3, 5: 3, 3: 2, 2: -3})
13 >>> +c # éléments positifs
14 Counter({1: 3, 5: 3, 3: 2})
15 >>> -c # éléments négatifs
16 Counter({2: 3})
```

1.5.5. OrderedDict

Les dictionnaires ordonnés (`OrderedDict`) sont comme leur nom l'indique des dictionnaires où l'ordre d'insertion des éléments est conservé, et qui sont itérés dans cet ordre.

Certaines implémentations (pypy, CPython 3.6) disposent de dictionnaires ordonnés par défaut, mais il est préférable de passer par `OrderedDict` pour s'en assurer.

Depuis Python 3.6², on peut construire un dictionnaire ordonné de la manière suivante :

```
1 >>> from collections import OrderedDict
2 >>> d = OrderedDict(b=0, a=1, c=2)
```

Dans les versions précédentes du langage, l'ordre des paramètres nommés n'étant pas assuré, il faut plutôt procéder avec un conteneur ordonné en paramètre.

```
1 >>> d = OrderedDict([('b', 0), ('c', 2), ('a', 1)])
```

Le dictionnaire ordonné est sinon semblable à tout autre dictionnaire.

```
1 >>> d['d'] = 4
2 >>> d['c'] = 5
3 >>> for key, value in d.items():
4 ...     print(key, value)
5 ...
6 b 0
7 c 5
8 a 1
9 d 4
```

pour en savoir plus sur les nouveautés apportées par cette version : <https://zestedesavoir.com/articles/1540/sortie-de-python-3-6/> ↗

2. Je vous invite à consulter cet article sur la sortie de Python 3.6

II. Mise en bouche

1.5.6. defaultdict

Voyons maintenant un dernier type de dictionnaires, les `defaultdict` (dictionnaires à valeurs par défaut). Les compteurs décrits plus haut sont un exemple de dictionnaires à valeurs par défaut : quand un élément n'existe pas, c'est `0` qui est retourné.

Les `defaultdict` sont plus génériques que cela. Lors de leur construction, ils prennent en premier paramètre une fonction qui servira à initialiser les éléments manquants.

```
1 >>> from collections import defaultdict
2 >>> d = defaultdict(lambda: 'x')
3 >>> d[0] = 'a'
4 >>> d[1] = 'b'
5 >>> d[2]
6 'x'
7 >>> d
8 defaultdict(<function <lambda> at 0x7ffa55be42f0>, {0: 'a', 1: 'b',
   2: 'x'})
```

1.5.7. Wrappers

Enfin, 3 classes (`UserDict`, `UserList` et `UserString`) sont présentes dans ce module. Elles permettent par héritage de créer vos propres types de dictionnaires, listes ou chaînes de caractères, pour leur ajouter des méthodes par exemple. Elles gardent une référence vers l'objet qu'elles étendent dans leur attribut `data`.

```
1 >>> from collections import UserList
2 >>> class MyList(UserList):
3 ...     def head(self):
4 ...         return self.data[0]
5 ...     def queue(self):
6 ...         return self.data[1:]
7 ...
8 >>> l = MyList([1, 2, 3])
9 >>> l.append(4)
10 >>> l
11 [1, 2, 3, 4]
12 >>> l.head()
13 1
14 >>> l.queue()
15 [2, 3, 4]
```

Ces classes datent cependant d'un temps ~~que les moins de 20 ans ne peuvent pas connaître~~ où il était impossible d'hériter des types `dict`, `list` ou `str`. Elles ont depuis perdu de leur intérêt.

1.6. TP : Une liste chaînée en Python

1.6.1. Présentation

Nous avons, dans les paragraphes précédents, créé un proxy autour d'une liste pour découvrir le fonctionnement des méthodes décrites.

Dans ce TP, pas à pas, nous créerons notre propre type de liste, à savoir une liste chaînée, c'est-à-dire composée de maillons reliés entre eux. Très courantes dans des langages bas-niveau tels que le C, elles le sont beaucoup moins en Python.

La différence avec le type `deque` du module `collections` est que nos listes seront simplement chaînées : un maillon n'aura connaissance que du maillon suivant, pas du précédent.

En plus de nos méthodes d'accès aux éléments, nous implémenterons les méthodes `insert` et `append` afin d'ajouter facilement des éléments à notre liste.

1.6.2. Bases

Nous appellerons donc notre classe `Deque` et, à la manière de `list`, le constructeur pourra prendre un objet pour pré-remplir notre liste.

Notre liste sera composée de maillons, et nous aurons donc une seconde classe, très simple, pour représenter un maillon : `Node`. un maillon possède une valeur (`value`), un maillon suivant (`next`), et... c'est tout. `next` pouvant être à `None` si l'on est en fin de liste.

La liste ne gardera qu'une référence vers le premier maillon, et vers le dernier (deux extrémités, *double-ended*).

Une seule chose à laquelle penser : quand nousinstancions notre maillon, nous voulons que celui-ci référence le maillon suivant. Mais aussi, et surtout, que le maillon précédent le référence. Ainsi, le `next` du maillon précédent devra pointer vers notre nouveau maillon.

```
1 class Node:
2     def __init__(self, value, next=None):
3         self.value = value
4         self.next = next
```

Et notre classe `Deque`, qui contiendra une référence vers le premier et le dernier maillon, tout en itérant sur le potentiel objet passé au constructeur pour initialiser la liste.

```
1 class Deque:
2     def __init__(self, iterable=()):
3         self.first = None # Premier maillon
4         self.last = None # Dernier maillon
5         for element in iterable:
```

6

```
self.append(element)
```

Notre classe `Node` étant achevée, toutes les méthodes qui seront données par la suite seront à ajouter à la classe `Deque`.

1.6.3. `append` et `insert`

Si vous avez tenté d'instancier notre liste pour le moment (en lui précisant un paramètre), vous avez remarqué que celle-ci levait une erreur : en effet, nous appelons une méthode `append` qui n'est actuellement pas définie.

C'est par celle-ci que nous allons commencer, car son comportement est très simple : nous créons un nouveau noeud que nous ajoutons à la fin de la liste. Cela peut se résumer en :

- Créer un `Node` avec la valeur spécifiée en paramètre, et `None` comme maillon suivant ;
- Lier le nouveau maillon à l'actuel dernier maillon ;
- Faire pointer la fin de liste sur ce nouveau maillon ;
- Et, ne pas oublier, dans le cas où notre liste est actuellement vide, de faire pointer le début de liste sur ce maillon.

```
1 def append(self, value):
2     node = Node(value, None)
3     if self.last is not None:
4         self.last.next = node
5     self.last = node
6     if self.first is None:
7         self.first = node
```

Vient maintenant la méthode `insert`, qui permet d'insérer une nouvelle valeur à n'importe quel endroit de la liste. Nous allons pour cela nous aider d'une première méthode `get_node` pour récupérer un maillon dans la liste (un objet de type `Node`, donc, pas sa valeur), qui nous servira encore beaucoup par la suite.

Cette méthode prendra un nombre en paramètre, correspondant à l'indice du maillon que nous voulons extraire, et itérera sur les maillons de la liste jusqu'à arriver à celui-ci. Nous nous contenterons pour le moment de gérer les nombres positifs. Nous leverons de plus une erreur si l'indice ne correspond à aucun maillon.

```
1 def get_node(self, n):
2     node = self.first
3     while n > 0 and node is not None:
4         node = node.next
5         n -= 1
6     if node is None:
7         raise IndexError("list index out of range")
```


II. Mise en bouche

```
8     return node
```

Notre méthode `insert` prend deux paramètres : la position et la valeur à insérer. Cette méthode aura trois comportements, suivant que l'on cherche à insérer la valeur en tête de liste, en fin, ou au milieu.

- Dans le premier cas, il nous faudra créer un nouveau maillon, avec `self.first` comme maillon suivant, puis faire pointer `self.first` sur ce nouveau maillon ;
- Dans les deux autres, il faudra repérer le maillon précédent à l'aide de `get_node`, puis insérer notre maillon à la suite de celui-ci ;
- Dans tous les cas, il faudra faire pointer `self.last` vers notre maillon si celui-ci est en fin de liste.

```
1 def insert(self, i, value):
2     if not i:
3         node = Node(value, next=self.first)
4         self.first = node
5     else:
6         prev = self.get_node(i - 1)
7         node = Node(value, prev.next)
8         prev.next = node
9     if node.next is None:
10        self.last = node
```

1.6.4. `__contains__`

Pour faire de notre liste un conteneur, il nous faut implémenter l'opération `in` et donc la méthode `__contains__`. Celle-ci, à la manière de `get_node`, itérera sur tous les maillons jusqu'à en trouver un correspondant à la valeur passée en paramètre et ainsi retourner `True`. Si la valeur n'est pas trouvée après avoir itéré sur toute la liste, il convient alors de retourner `False`.

```
1 def __contains__(self, value):
2     node = self.first
3     while node is not None:
4         if node.value == value:
5             return True
6         node = node.next
7     return False
```

1.6.5. `__len__`

Nous souhaitons maintenant pouvoir calculer la taille de notre liste. La méthode `__len__` sera aussi très similaire aux précédentes, elle itère simplement du début à la fin en comptant le

II. Mise en bouche

nombre de noeuds.

```
1 def __len__(self):
2     node = self.first
3     size = 0
4     while node is not None:
5         node = node.next
6         size += 1
7     return size
```

1.6.6. `__getitem__` et `__setitem__`

Dans un premier temps, implémentons ces deux méthodes sans tenir compte des `slice`. Une grande part du travail est déjà réalisé par `get_node` :

```
1 def __getitem__(self, key):
2     return self.get_node(key).value
3
4 def __setitem__(self, key, value):
5     self.get_node(key).value = value
```

Puis vient la gestion des `slice`. Rappelons-nous ce que contient un `slice`, à savoir un indice de début, un indice de fin et un pas. Cela ne vous rappelle pas quelque chose ? Si, c'est exactement ce à quoi correspond `range`. À ceci près que les valeurs `None` n'ont aucune signification pour les `range`.

Mais l'objet `slice` possède une méthode `indices`, qui, en lui donnant la taille de notre ensemble, retourne les paramètres à passer à `range` (en gérant de plus pour nous les indices négatifs).

Nous pouvons ainsi obtenir un `range` à l'aide de l'expression `range(*key.indices(len(self)))` (en considérant que `key` est un objet de type `slice`). Il ne nous reste donc plus qu'à itérer sur le `range`, et :

- Créer une nouvelle liste contenant les éléments extraits dans le cas d'un `__getitem__` ;
- Modifier un à un les éléments dans le cas d'un `__setitem__`.

```
1 def __getitem__(self, key):
2     if isinstance(key, slice):
3         new_list = Deque()
4         indices = range(*key.indices(len(self)))
5         for i in indices:
6             new_list.append(self[i])
7         return new_list
8     else:
```

```
9         return self.get_node(key).value
10
11     def __setitem__(self, key, value):
12         if isinstance(key, slice):
13             indices = range(*key.indices(len(self)))
14             for i, v in zip(indices, value):
15                 self[i] = v
16         else:
17             self.get_node(key).value = value
```

Si vous ne comprenez pas bien ce que fait la fonction `zip`, celle-ci assemble deux listes (nous verrons dans le chapitre suivant que c'est un peu plus large que cela), de la manière suivante :

```
1 >>> list(zip([1, 2, 3], ['a', 'b', 'c']))
2 [(1, 'a'), (2, 'b'), (3, 'c')]
```

1.6.7. Aller plus loin

Ce TP touche à sa fin, mais pour aller plus loin, voici une liste non exhaustive de fonctionnalités qu'il nous reste à implémenter :

- Gérer des nombres négatifs pour l'indexation ;
- Gérer les cas de `__setitem__` où la liste de valeurs a une taille différente de la taille du `slice` ;
- Implémenter la méthode `__delitem__` pour gérer la suppression de maillons, attention toutefois pour la gestion des `slice`, les indices seront invalidés après chaque suppression ;
- Implémenter les méthodes spéciales `__str__` et `__repr__` pour afficher facilement notre liste.

Nous sommes maintenant à la fin du premier chapitre. En guise de conclusion, je vais vous fournir une liste de sources tirées de la documentation officielle permettant d'aller plus loin sur ces sujets.

- La définition du terme séquence : <https://docs.python.org/3/glossary.html#term-sequence> ↗
- Celle du *mapping* : <https://docs.python.org/3/glossary.html#term-mapping> ↗
- Types de séquences : <https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range> ↗
- Types standards : <https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy> ↗
- Émuler les conteneurs : <https://docs.python.org/3/reference/datamodel.html#emulating-container-types> ↗
- Module `collections` : <https://docs.python.org/3/library/collections.html> ↗

2. Itérables

Un itérable est un objet dont on peut parcourir les valeurs, à l'aide d'un `for` par exemple. La liste que nous venons d'implémenter est un exemple d'itérable.

Les types `str`, `tuple`, `list`, `dict` et `set` sont d'autres itérables bien connus. Un grand nombre d'outils Python que nous verrons par la suite travaillent avec des itérables, il est donc intéressant d'en tirer profit.

L'objectif de ce chapitre va être de comprendre ce qu'est un itérable, et comment en implémenter un.

2.1. for for lointain

Les itérables et le mot-clef `for` sont intimement liés. C'est à partir de ce dernier que nous itérons sur les objets.

Mais comment cela fonctionne en interne ? Je vous propose de regarder ça pas à pas, en nous aidant d'un objet de type `list`.

```
1 >>> numbers = [1, 2, 3, 4, 5]
```

La première opération réalisée par le `for` est d'appeler la fonction `iter` avec notre objet. `iter` retourne un itérateur. L'itérateur est l'objet qui va se déplacer le long de l'itérable.

```
1 >>> iter(numbers)
2 <list_iterator object at 0x7f26896c0940>
```

Puis, pas à pas, le `for` appelle `next` en lui précisant l'itérateur. `next` fait avancer l'itérateur et retourne la nouvelle valeur découverte à chaque pas.

```
1 >>> iterator = iter(numbers)
2 >>> next(iterator)
3 1
4 >>> next(iterator)
5 2
6 >>> next(iterator)
```

II. Mise en bouche

```
7 3
8 >>> next(iterator)
9 4
10 >>> next(iterator)
11 5
12 >>> next(iterator)
13 Traceback (most recent call last):
14   File "<stdin>", line 1, in <module>
15 StopIteration
```

Qu'est-ce que ce `StopIteration` ? Il s'agit d'une exception, levée par l'itérateur quand il arrive à sa fin, qui signifie que nous en sommes arrivés au bout, et donc que la boucle doit cesser. `for` attrape cette exception pour nous, ce qui explique que nous ne la voyons pas survenir dans une boucle habituelle.

Ainsi, le code suivant :

```
1 for number in numbers:
2     print(number)
```

Peut se remplacer par celui-ci :

```
1 iterator = iter(numbers)
2 while True:
3     try:
4         number = next(iterator)
5     except StopIteration:
6         break
7     print(number)
```

En interne, `iter` fait habituellement appel à la méthode `__iter__` de l'itérable, et `next` à la méthode `__next__` de l'itérateur. Ces deux méthodes ne prennent aucun paramètre. Ainsi :

- Un itérable est un objet possédant une méthode `__iter__`³ retournant un itérateur ;
- Un itérateur est un objet possédant une méthode `__next__` retournant la valeur suivante à chaque appel, et levant une exception de type `StopIteration` en fin de course.

La [documentation Python](#) indique aussi qu'un itérateur doit avoir une méthode `__iter__` où il se retourne lui-même, les itérateurs étant ainsi des itérables à part entière.

2.1.1. Le cas des indexables

En début du chapitre, j'ai indiqué que notre liste `Deque` était aussi un itérable. Pourtant, nous ne lui avons pas implémenté de méthode `__iter__` permettant de la parcourir.

3. À une approximation près, comme détaillé dans « Le cas des indexables ».

II. Mise en bouche

Il s'agit en fait d'une particularité des indexables, et de la fonction `iter` qui est capable de créer un itérateur à partir de ces derniers. Cet itérateur se contentera d'appeler `__getitem__` sur notre objet avec des indices successifs, partant de 0 et continuant jusqu'à ce que la méthode lève une `IndexError`.

Dans notre cas, ça nous évite donc d'implémenter nous-même `__iter__`, mais ça complexifie aussi les traitements. Souvenez-vous de notre méthode `__getitem__` : elle parcourt la liste jusqu'à l'élément voulu.

Ainsi, pour accéder au premier maillon, on parcourt un élément, on en parcourt deux pour accéder au second, etc. Donc pour itérer sur une liste de 5 éléments, on va devoir parcourir $1 + 2 + 3 + 4 + 5$ soit 15 maillons, là où 5 seraient suffisants. C'est pourquoi nous reviendrons sur `Deque` en fin de chapitre pour lui intégrer sa propre méthode `__iter__`.

2.2. Utilisation des itérables

2.2.1. Python et les itérables

Ce concept d'itérateurs est utilisé par Python dans une grande partie des ses [builtins](#) [↗](#). Plutôt que de vous forcer à utiliser une liste, Python vous permet de fournir un objet itérable, pour `sum`, `max` ou `map` par exemple.

Je vous propose de tester cela avec un itérable basique, qui nous permettra de réaliser un `range` simplifié.

```
1 class MyRange:
2     def __init__(self, size):
3         self.size = size
4
5     def __iter__(self):
6         return MyRangeIterator(self)
7
8 class MyRangeIterator:
9     def __init__(self, my_range):
10        self.current = 0
11        self.max = my_range.size
12
13    def __iter__(self):
14        return self
15
16    def __next__(self):
17        if self.current >= self.max:
18            raise StopIteration
19        ret = self.current
20        self.current += 1
21        return ret
```

II. Mise en bouche

Maintenant, testons notre objet, en essayant d'itérer dessus à l'aide d'un `for`.

```
1 >>> MyRange(5)
2 <__main__.MyRange object at 0x7fcf3b0e8f28>
3 >>> for i in MyRange(5):
4 ...     print(i)
5 ...
6 0
7 1
8 2
9 3
10 4
```

Voilà pour l'itération, mais testons ensuite quelques autres *builtins* dont je parlais plus haut.

```
1 >>> sum(MyRange(5)) # sum réalise la somme de tous les éléments,
   soit 0 + 1 + 2 + 3 + 4
2 10
3 >>> max(MyRange(5)) # max retourne la plus grande valeur
4 4
5 >>> map(str, MyRange(5)) # Ici, map retournera chaque valeur
   convertie en str
6 <map object at 0x7f8b81226cf8>
```

Mmmh, que s'est-il passé ? En fait, `map` ne retourne pas une liste, mais un nouvel itérateur. Si nous voulons en voir le contenu, nous pouvons itérer dessus... ou plus simplement, convertir le résultat en liste :

```
1 >>> list(map(str, MyRange(5)))
2 ['0', '1', '2', '3', '4']
```

Vous l'aurez compris, `list` prend aussi n'importe quel itérable en argument, tout comme `zip` ou `str.join` par exemple.

```
1 >>> list(MyRange(5))
2 [0, 1, 2, 3, 4]
3 >>> list(zip(MyRange(5), 'abcde')) # Les chaînes sont aussi des
   itérables
4 [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
5 >>> ','.join(map(str, MyRange(5)))
6 '0, 1, 2, 3, 4'
```

II. Mise en bouche


J'en resterai là pour les exemples, sachez seulement que beaucoup de fonctions sont compatibles. Seules celles nécessitant des propriétés spécifiques de l'objet ne le seront pas par défaut, comme la fonction `reversed`.

2.2.2. Retour sur `iter`

Je voudrais ici revenir sur la fonction `iter`, qui crée un itérateur à partir d'un itérable. Sachez que ce n'est pas sa seule utilité. Elle peut aussi créer un itérateur à partir d'une fonction et d'une valeur de fin. C'est-à-dire que la fonction sera appelée tant que la valeur de fin n'a pas été retournée, par exemple :

```
1 >>> n = 0
2 >>> def iter_func():
3 ...     global n
4 ...     n += 1
5 ...     return n
6 ...
7 >>> for i in iter(iter_func, 10):
8 ...     print(i)
9 ...
10 1
11 2
12 3
13 4
14 5
15 6
16 7
17 8
18 9
```

2.3. Utilisation avancée : le module `itertools`

Nous l'avons vu, les itérables sont au cœur des fonctions élémentaires de Python. Je voudrais maintenant vous présenter un module qui vous sera probablement très utile : [itertools](#) .

Ce module met à disposition de nombreux itérables plutôt variés, dont :

- `chain(p, q, ...)` — Met bout à bout plusieurs itérables ;
- `islice(p, start, stop, step)` — Fait un travail semblable aux slices, mais en travaillant avec des itérables (nul besoin de pouvoir indexer notre objet) ;
- `combinations(p, r)` — Retourne toutes les combinaisons de `r` éléments possibles dans `p` ;
- `zip_longest(p, q, ...)` — Similaire à `zip`, mais s'aligne sur l'itérable le plus grand plutôt que le plus petit (en permettant de spécifier une valeur de remplissage).


```
1 >>> import itertools
2 >>> itertools.chain('abcd', [1, 2, 3])
3 <itertools.chain object at 0x7f757b508c88>
4 >>> list(itertools.chain('abcd', [1, 2, 3]))
5 ['a', 'b', 'c', 'd', 1, 2, 3]
6 >>> list(itertools.islice(itertools.chain('abcd', [1, 2, 3]), 1,
7     None, 2))
8 ['b', 'd', 2]
9 >>> list(itertools.combinations('abc', 2))
10 [(('a', 'b'), ('a', 'c'), ('b', 'c'))]
11 >>> list(itertools.zip_longest('abcd', [1, 2, 3]))
12 [(('a', 1), ('b', 2), ('c', 3), ('d', None))]
```

Je tiens enfin à attirer votre attention sur les [recettes \(recipes\)](#) [↗](#), un ensemble d'exemples qui vous sont proposés mettant à profit les outils présents dans `itertools`.

2.4. L'unpacking

Une fonctionnalité courante de Python, liée aux itérables, est celle de l'*unpacking*. Il s'agit de l'opération qui permet de décomposer un itérable en plusieurs variables.

Prenons `values` une liste de 3 valeurs, il est possible en une ligne d'assigner chaque valeur à une variable différente.

```
1 >>> values = [1, 3, 5]
2 >>> a, b, c = values
3 >>> a
4 1
5 >>> b
6 3
7 >>> c
8 5
```

J'utilise ici une liste `values`, mais tout type d'itérable est accepté, on pourrait avoir un `range` ou un `set` par exemple. L'itérable n'a pas besoin d'être une séquence comme la liste.

```
1 >>> a, b, c = range(1, 6, 2)
2 >>> a, b, c = {1, 3, 5} # l'ordre n'est pas assuré dans ce dernier
   cas
```

C'est aussi cette fonctionnalité qui est à l'origine de l'assignement multiple et de l'échange de variables.

II. Mise en bouche

```
1 >>> x, y = 10, 20
2 >>> x
3 10
4 >>> y
5 20
6 >>> x, y = y, x
7 >>> x
8 20
9 >>> y
10 10
```

En effet, nous avons dans ces deux cas, à gauche comme à droite du signe `=`, des *tuples*. Et celui de droite est décomposé pour correspondre aux variables de gauche.

Je parle de *tuples*, mais on retrouve la même chose avec des listes. Les assignations suivantes sont d'ailleurs équivalentes.

```
1 >>> x, y = 10, 20
2 >>> (x, y) = (10, 20)
3 >>> [x, y] = [10, 20]
```

2.4.1. Structures imbriquées

Ces cas d'*unpacking* sont les plus simples : nous avons un itérable à droite et un ensemble « plat » de variables à gauche. Je dis « plat » parce qu'il n'y a qu'un niveau, aucune imbrication.

Mais il est possible de faire bien plus que cela, en décomposant aussi des itérables imbriqués les uns dans les autres.

```
1 >>> a, ((b, c, d), e), (f, g) = [0, (range(1, 4), 5), '67']
2 >>> a
3 0
4 >>> b
5 1
6 >>> c
7 2
8 >>> d
9 3
10 >>> e
11 5
12 >>> f
13 '6'
14 >>> g
15 '7'
```

2.4.2. Opérateur *splat*

Mais on peut aller encore plus loin avec l'opérateur *splat*. Cet opérateur est représenté par le caractère `*`.

À ne pas confondre avec la multiplication, opérateur binaire entre deux objets, il s'agit ici d'un opérateur unaire : c'est-à-dire qu'il n'opère que sur un objet, en se plaçant devant.

Utilisé à gauche lors d'une assignation, il permet de récupérer plusieurs éléments lors d'une décomposition.

```
1 >>> head, *tail = range(10)
2 >>> head
3 0
4 >>> tail
5 [1, 2, 3, 4, 5, 6, 7, 8, 9]
6 >>> head, *middle, last = range(10)
7 >>> head
8 0
9 >>> middle
10 [1, 2, 3, 4, 5, 6, 7, 8]
11 >>> last
12 9
13 >>> head, second, *middle, last = range(10)
14 >>> head
15 0
16 >>> second
17 1
18 >>> middle
19 [2, 3, 4, 5, 6, 7, 8]
20 >>> last
21 9
```

Vous l'avez compris, la variable précédée du *splat* devient une liste, dont la taille s'ajuste en fonction du nombre d'éléments.

Il est donc impossible d'avoir deux variables précédées d'un *splat*, cela mènerait à une ambiguïté. Ou plutôt, devrais-je préciser, une seule par niveau d'imbrication.

```
1 >>> *a, (b, *c) = (0, 1, 2, (3, 4, 5))
2 >>> a
3 [0, 1, 2]
4 >>> b
5 3
6 >>> c
7 [4, 5]
```

2.4.3. Encore du *splat*

Nous avons vu l'opérateur *splat* utilisé à gauche de l'assignation, mais il est aussi possible depuis Python 3.5⁴ de l'utiliser à droite. Il aura simplement l'effet inverse, et décomposera un itérable comme si ses valeurs avaient été entrées une à une.

```
1 >>> values = *[0, 1, 2], 3, 4, *[5, 6], 7
2 >>> values
3 (0, 1, 2, 3, 4, 5, 6, 7)
```

Il est bien sûr possible de combiner les deux.

```
1 >>> first, *middle, last = *[0, 1, 2], 3, 4, *[5, 6], 7
2 >>> first
3 0
4 >>> middle
5 [1, 2, 3, 4, 5, 6]
6 >>> last
7 7
```

2.5. TP : Itérateur sur listes chaînées

Revenons sur nos listes chaînées afin d'y implémenter le protocole des itérables. Notre classe `Deque` a donc besoin d'une méthode `__iter__` retournant un itérateur, que nous appellerons simplement `DequeIterator`.

```
1 def __iter__(self):
2     return DequeIterator(self)
```

Cet itérateur contiendra une référence vers un maillon, puis, à chaque appel à `__next__`, renverra la valeur du maillon courant, tout en prenant soin de passer au maillon suivant pour le prochain appel. `StopIteration` sera levée si le maillon courant vaut `None`.

On ajoutera aussi `__iter__` dans l'itérateur, comme vu plus tôt, dans le cas où cet itérateur serait utilisé comme itérable.

```
1 class DequeIterator:
2     def __init__(self, deque):
3         self.current = deque.first
```

4. Pour plus d'informations sur les possibilités étendues de l'opérateur *splat* offertes par Python 3.5 : <https://zestedesavoir.com/articles/175/sortie-de-python-3-5/#2-principales-nouveautes> ↗

II. Mise en bouche

```
4
5     def __next__(self):
6         if self.current is None:
7             raise StopIteration
8         value = self.current.value
9         self.current = self.current.next
10        return value
11
12    def __iter__(self):
13        return self
```

Testons maintenant notre implémentation...

```
1 >>> for i in Deque([1, 2, 3, 4, 5]):
2     ...     print(i)
3     ...
4     1
5     2
6     3
7     4
8     5
```

... Ça marche !

Passons enfin aux ressources de la documentation concernant les itérables et itérateurs.

- Définition du terme itérable : <https://docs.python.org/3/glossary.html#term-iterable> ↗
- Du terme itérateur : <https://docs.python.org/3/glossary.html#term-iterator> ↗
- Type itérateur : <https://docs.python.org/3/library/stdtypes.html#iterator-types> ↗
- Module `itertools` : <https://docs.python.org/3/library/itertools.html> ↗

Les PEP, propositions et descriptions de nouvelles fonctionnalités, sont aussi des sources d'informations intéressantes.

- *Unpacking* généralisé : <https://www.python.org/dev/peps/pep-0448> ↗

Et pour finir, quelques ressources annexes sur ces sujets :

- Nouvelles fonctionnalités de Python 3.5 : <https://zestedesavoir.com/articles/175/sortie-de-python-3-5/> ↗
- Article Sam&Max sur l'opérateur *splat* : <http://sametmax.com/operateur-splat-ou-etoile-en-python/> ↗

3. Objets mutables et hashables

Nous allons ici étudier deux propriétés fondamentales des objets en Python : leur mutabilité et leur hashabilité.

La première correspond à la capacité des objets à être altérés, modifiés.

La seconde est la possibilité pour un objet d'être hashé, c'est-à-dire d'en calculer un condensat qui permet entre-autres à l'objet d'être utilisé comme clef dans un dictionnaire.

3.1. Mutables

Un objet mutable est ainsi un objet qui peut être modifié, dont on peut changer les propriétés une fois qu'il a été défini. Une erreur courante est de confondre modification et réassignation.

La différence est facile à comprendre avec les listes. Les listes sont des objets mutables : une fois la liste instanciée, il est par exemple possible d'y insérer de nouveaux éléments.

```
1 >>> values = [0, 1, 2]
2 >>> values.append(3)
3 >>> values
4 [0, 1, 2, 3]
```

Le fonctionnement des variables en Python fait qu'il est possible d'avoir plusieurs noms (étiquettes) sur une même valeur. Le principe de mutabilité s'observe alors très bien.

```
1 >>> values = othervalues = [0, 1, 2]
2 >>> values.append(3)
3 >>> values
4 [0, 1, 2, 3]
5 >>> othervalues
6 [0, 1, 2, 3]
```

Sans que nous n'ayons explicitement touché à `othervalues`, sa valeur a changé. En effet, `values` et `othervalues` référencent un même objet. Un même objet mutable.

En revanche, la réassignation fait correspondre le nom de la variable à un nouvel objet, il n s'agit pas d'une modification de la valeur initiale.

II. Mise en bouche

```
1 >>> values = othervalues = [0, 1, 2]
2 >>> values = [0, 1, 2, 3] # réassignation de values
3 >>> values
4 [0, 1, 2, 3]
5 >>> othervalues
6 [0, 1, 2]
```

En Python, les objets de type `bool`, `int`, `str`, `bytes`, `tuple`, `range` et `frozenset` sont immutables. Tous les autres types, les listes, les dictionnaires, ou les instances de vos propres classes sont des objets mutables.

Comme nous l'avons vu, les objets mutables sont à prendre avec des pincettes, car leur valeur peut changer sans que nous ne l'ayons explicitement demandé. Cela peut arriver lorsqu'une valeur mutable est passée en argument à une fonction.

```
1 >>> def append_42(values):
2 ...     values.append(42)
3 ...     return values
4 ...
5 >>> v = [1, 2, 3, 4]
6 >>> append_42(v)
7 [1, 2, 3, 4, 42]
8 >>> v
9 [1, 2, 3, 4, 42]
```

Cela ne pourra jamais arriver avec un *tuple* par exemple, qui est immuable et ne possède aucune méthode pour être altéré.

```
1 >>> def append_42(values):
2 ...     return values + (42,)
3 ...
4 >>> v = (1, 2, 3, 4)
5 >>> append_42(v)
6 (1, 2, 3, 4, 42)
7 >>> v
8 (1, 2, 3, 4)
```

Il n'est pas vraiment possible en Python de créer un nouveau type immuable. Cela peut être simulé en rendant les méthodes de modification/suppression inefficaces. Mais il est toujours possible de passer outre en appelant directement les méthodes d'une classe parente.

```
1 >>> class ImmutableDeque(Deque):
2 ...     def append(self, value):
```

```
3 ...     raise TypeError('Object is read-only')
4 ...     def insert(self, index, value):
5 ...         raise TypeError('Object is read-only')
6 ...     def __setitem__(self, key, value):
7 ...         raise TypeError('Object is read-only')
8 ...
9 >>> deque = ImmutableDeque()
10 >>> deque.append('foo')
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13   File "<stdin>", line 3, in append
14 TypeError: Object is read-only
15 >>> Deque.append(deque, 'foo')
16 >>> deque[0]
17 'foo'
```

La seule manière sûre est d'hériter d'un autre type immuable, comme les `namedtuple` qui héritent de `tuple`. Nous verrons plus loin dans ce cours comment cela est réalisable.

3.2. Égalité et identité

L'égalité et l'identité sont deux concepts dont la distinction est parfois confuse. Deux valeurs sont égales lorsqu'elles partagent un même état : par exemple, deux chaînes qui contiennent les mêmes caractères sont égales. Deux valeurs sont identiques lorsqu'elles sont une même instance, c'est-à-dire un même objet en mémoire.

En Python, on retrouve ces concepts sous les opérateurs `==` (égalité) et `is` (identité).

```
1 >>> [1, 2, 3] == [1, 2, 3]
2 True
3 >>> [1, 2, 3] is [1, 2, 3]
4 False
5 >>> values = [1, 2, 3]
6 >>> values is values
7 True
```

Leur différence est fondamentale pour les types mutables, puisque deux valeurs distinctes peuvent être égales à un moment et ne plus l'être par la suite (si l'une d'elles est modifiée). Deux valeurs identiques resteront à l'inverse égales, puisque les modifications seront perçues sur les deux variables.

```
1 >>> values1, values2 = [1, 2, 3], [1, 2, 3]
2 >>> values1 == values2
3 True
```


II. Mise en bouche

```
4 >>> values1 is values2
5 False
6 >>> values1.append(4)
7 >>> values1 == values2
8 False
```

```
1 >>> values1 = values2 = [1, 2, 3]
2 >>> values1 == values2
3 True
4 >>> values1 is values2
5 True
6 >>> values1.append(4)
7 >>> values1 == values2
8 True
```

L'opérateur d'égalité est surchargeable en Python, *via* la méthode spéciale `__eq__` des objets. Il est en effet de la responsabilité du développeur de gérer la comparaison entre ses objets, et donc de déterminer quand ils sont égaux. Cette méthode reçoit en paramètre la valeur à laquelle l'objet est comparé, et retourne un booléen.

On peut imaginer une valeur qui sera égale à toute les autres, grâce à une méthode `__eq__` retournant toujours `True`.

```
1 >>> class AlwaysEqual:
2 ...     def __eq__(self, value):
3 ...         return True
4 ...
5 >>> val = AlwaysEqual()
6 >>> val == 0
7 True
8 >>> 1 == val
9 True
```

L'opérateur d'identité testant si deux objets sont une même instance, il n'est bien sûr pas possible de le surcharger. En absence de surcharge, l'opérateur d'égalité donnera la même résultat que l'identité.

Vous pouvez vous référer à [ce chapitre du cours sur la POO en Python](#) pour davantage d'informations sur la surcharge d'opérateurs.

3.2.1. Quel opérateur utiliser ?

Une question légitime à se poser suite à ces lignes est de savoir quel opérateur utiliser pour comparer nos valeurs. La réponse est que cela dépend des valeurs et des cas d'utilisation.

II. Mise en bouche

En règle générale, c'est l'opérateur d'égalité (`==`) qui est à utiliser. Quand nous comparons un nombre entré par l'utilisateur avec un nombre à deviner, nous ne cherchons pas à savoir s'ils sont un même objet, mais s'ils représentent la même chose.

L'opérateur `is` s'utilise principalement avec `None`. `None` est une valeur unique (*singleton*), il n'en existe qu'une instance. Quand on compare une valeur avec `None`, on vérifie qu'elle est `None` et non qu'elle vaut `None`.

Globalement, `is` s'utilise pour la comparaison avec des *singletons*, et `==` s'utilise pour le reste.

3.3. Hashables

Comme je le disais plus tôt, les objets hashables vont notamment servir pour les clefs des dictionnaires. Mais voyons tout d'abord à quoi correspond cette capacité.

3.3.1. Le condensat

En informatique, et plus particulièrement en cryptographie, on appelle condensat (*hash*) un nombre calculé depuis une valeur quelconque, unique et invariable pour cette valeur. Deux valeurs égales partageront un même *hash*, deux valeurs différentes auront dans la mesure du possible des *hash* différents.

En effet, le condensat est généralement un nombre de taille fixe (64 bits par exemple), il existe donc un nombre limité de *hashs* pour un nombre infini de valeurs. Deux valeurs différentes pourront alors avoir un même condensat, c'est ce que l'on appelle une collision. Les collisions sont plus ou moins fréquentes selon les algorithmes de *hashage*. En cela, l'égalité entre *hashs* ne doit jamais remplacer l'égalité entre les valeurs, elle n'est qu'une étape préliminaire qui peut servir à optimiser des calculs.

3.3.2. La fonction `hash`

En Python, on peut calculer le condensat d'un objet à l'aide de la fonction `hash`.

```
1 >>> hash(10)
2 10
3 >>> hash(2**61 + 9) # collision
4 10
5 >>> hash('toto')
6 -7475273891964572862
7 >>> hash((1, 2, 3))
8 2528502973977326415
9 >>> hash([1, 2, 3])
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12 TypeError: unhashable type: 'list'
```

II. Mise en bouche

Ce dernier exemple nous montre que les listes ne sont pas hashables. Pourquoi ? On a vu que le *hash* était invariable, mais il doit pourtant correspondre à la valeur.

Or, en modifiant une liste, le condensat calculé auparavant deviendrait invalide. Il est donc impossible de hasher les listes. Il en est de même pour les dictionnaires et les ensembles (`set`). Tous les autres types d'objets sont par défaut hashables.

On remarque une certaine corrélation entre types mutables et hashables. En effet, il est plus facile d'assurer l'invariabilité du condensat quand l'objet est lui-même immuable. Pour les objets mutables, le *hash* n'est possible que si la modification n'altère pas l'égalité entre deux objets, c'est-à-dire que deux objets égaux le resteront même si l'un est modifié.

Il faut aussi garder à l'esprit que des types immutables peuvent contenir des mutables. Par exemple une liste dans un *tuple*. Dans ce genre de cas, la non-hashabilité des valeurs contenues rend non-hashable le conteneur.

```
1 >>> t = ((0, 1), (2, 3))
2 >>> hash(t)
3 8323144716662114087
4 >>> t = ((0, 1), [2, 3])
5 >>> hash(t)
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   TypeError: unhashable type: 'list'
```

3.3.3. À quoi servent-ils ?

Je parle depuis le début de clefs de dictionnaires, nous allons maintenant voir pourquoi les dictionnaires utilisent des condensats.

Les dictionnaires, d'ailleurs appelés tables de hashage dans certains langages, sont des structures qui doivent permettre un accès rapide aux éléments. Ainsi, ils ne peuvent pas être une simple liste de couples clef/valeur, qui serait parcourue chaque fois que l'on demande un élément.

À l'aide des *hash*, les dictionnaires disposent les éléments tels que dans un tableau et offrent un accès direct à la majorité d'entre eux.

Outre les dictionnaires, ils sont aussi utilisés dans les `set`, ensembles non ordonnés de valeurs uniques.

On remarque facilement que les objets non-hashables ne peuvent être utilisés en tant que clefs de dictionnaires ou dans un ensemble.

```
1 >>> {[0]: 'foo'}
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: unhashable type: 'list'
5 >>> {'foo': 'bar'}
```

```
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   TypeError: unhashable type: 'dict'
```

Plus généralement, les *hash* peuvent être utilisés pour optimiser le test d'égalité entre deux objets. Le *hash* étant invariable, il est possible de ne le calculer qu'une fois par objet (en stockant sa valeur).

Ainsi, lors d'un test d'égalité, on peut facilement dire que les objets sont différents, si les *hash* le sont. L'inverse n'est pas vrai à cause des collisions : deux objets différents peuvent avoir un même *hash*. Le test d'égalité proprement dit (appel à la méthode `__eq__`) doit donc toujours être réalisé si les *hash* sont égaux.

3.3.4. Implémentation

Les types de votre création sont par défaut hashables, puisque l'égalité entre objets vaut l'identité. La question de la *hashabilité* ne se pose donc que si vous surchargez l'opérateur `__eq__`.

Dans ce cas, il convient normalement de vous occuper aussi de la méthode spéciale `__hash__`. C'est cette méthode qui est appelée par la fonction `hash` pour calculer le condensat d'un objet.

Il est aussi possible d'assigner `None` à `__hash__` afin de rendre l'objet non-*hashable*. Python le fait par défaut lorsque nous surchargeons l'opérateur `__eq__`.

Pour reprendre la classe `AlwaysEqual` définie précédemment :

```
1 >>> val = AlwaysEqual()
2 >>> hash(val)
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   TypeError: unhashable type: 'AlwaysEqual'
6 >>> print(val.__hash__)
7 None
```

Si toutefois vous souhaitez redéfinir la méthode `__hash__`, il vous faut respecter les quelques règles énoncées plus haut.

- L'invariabilité du *hash* ;
- L'égalité entre deux *hashs* de valeurs égales.

Ces conditions état plus faciles à respecter pour des valeurs immutables.

Notons enfin que le résultat de la méthode `__hash__` est tronqué par la fonction `hash`, afin de tenir sur un nombre fixe de bits.

Pour plus d'informations sur cette méthode `__hash__` : https://docs.python.org/3/reference/datamodel.html#object.__hash__ ↗.

3.4. TP : Égalité entre listes

Nous allons maintenant nous intéresser à l'implémentation de l'opérateur d'égalité entre listes. Nos listes, mutables, deviendront par conséquent non-hashables. Nous reviendrons vers la fin de ce cours sur l'implémentation de listes immutables.

L'opérateur d'égalité correspond donc à la méthode spéciale `__eq__`, recevant en paramètre l'objet auquel `self` est comparé. La méthode retourne ensuite `True` si les objets sont égaux, `False` s'ils sont différents, et `NotImplemented` si la comparaison ne peut être faite.

`NotImplemented` est une facilité de Python pour gérer les opérateurs binaires. En effet, dans une égalité `a == b` par exemple, on ne peut pas savoir lequel de `a` ou `b` redéfinit la méthode `__eq__`. L'interpréteur va alors tester en premier d'appeler la méthode de `a` :

- Si la méthode retourne `True`, les objets sont égaux ;
- Si elle retourne `False`, ils sont différents ;
- Si elle retourne `NotImplemented`, alors l'interpréteur appellera la méthode `__eq__` de `b` pour déterminer le résultat ;
- Si les deux méthodes retournent `NotImplemented`, les objets sont différents.

Dans notre méthode, nous allons donc premièrement vérifier le type du paramètre. S'il n'est pas du type attendu (`Deque`), nous retournerons `NotImplemented`.

Nous comparerons ensuite la taille des listes, si la taille diffère, les listes sont nécessairement différentes. Dans l'idéal, nous devrions éviter cette comparaison car elle est coûteuse (elle nécessite de parcourir entièrement chacune des listes), mais nous pouvons la conserver dans le cadre de l'exercice.

Enfin, nous itérerons simultanément sur nos deux listes pour vérifier que tous les éléments sont égaux.

```
1 def __eq__(self, other):
2     if not isinstance(other, Deque):
3         return NotImplemented
4     if len(self) != len(other):
5         return False
6     for elem1, elem2 in zip(self, other):
7         if elem1 != elem2:
8             return False
9     return True
```

C'est l'heure du test !

```
1 >>> d = Deque([0, 1])
2 >>> d == Deque([0, 1])
3 True
4 >>> d == Deque([0, 1, 2])
5 False
```

II. Mise en bouche

```
6 >>> d == Deque([1, 2])
7 False
8 >>> d == 0
9 False
10 >>> d.append(2)
11 >>> d == Deque([0, 1])
12 False
13 >>> d == Deque([0, 1, 2])
14 True
```

Et comme nous pouvons le constater, notre `Deque` a perdu son pouvoir d'hashabilité.

```
1 >>> hash(d)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   TypeError: unhashable type: 'Deque'
```

Nous retrouvons ici les traditionnelles références vers la documentation officielle.

- Définition du terme mutable : <https://docs.python.org/3/glossary.html#term-mutable> ↗
- Du terme hashable : <https://docs.python.org/3/glossary.html#term-hashable> ↗
- Fonction `hash` : <https://docs.python.org/3/library/functions.html#hash> ↗
- Méthode `__hash__` et protocole : https://docs.python.org/3/reference/datamodel.html#object.__hash__ ↗

Troisième partie
Ainsi font fonctions

3.5. Trois petits tours et puis s'en vont

Nous allons dans ces chapitres traiter le concept de fonction, dans sa définition large (c'est-à-dire englobant tous les objets se comportant comme des fonctions).

4. Callables

Nous allons maintenant nous intéresser à un « nouveau » type d'objets : les *callables*. Je place des guillemets autour de nouveau car vous les fréquentez en réalité depuis que vous faites du Python, les fonctions étant des *callables*.

Qu'est-ce qu'un *callable* me demanderez-vous ? C'est un objet que l'on peut appeler. Appeler un objet consiste à utiliser l'opérateur `()`, en lui précisant un certain nombre d'arguments, de façon à recevoir une valeur de retour.

```
1 >>> print('Hello', 'world', end='!\n') # Appel d'une fonction avec
    différents arguments
2 Hello world!
3 >>> x = pow(2, 3) # Valeur de retour
4 >>> x
5 8
```

4.1. Fonctions, classes et lambdas

L'ensemble des *callables* contient donc les fonctions, mais pas seulement. Les classes en sont, les méthodes, les lambdas, etc. Sont callables tous les objets derrière lesquels on peut placer une paire de parenthèses, pour les appeler.

En Python, on peut vérifier qu'un objet est callable à l'aide de la fonction `callable`.

```
1 >>> callable(print)
2 True
3 >>> callable(lambda: None)
4 True
5 >>> callable(callable)
6 True
7 >>> callable('')
8 False
9 >>> callable('').join)
10 True
11 >>> callable(str)
12 True
13 >>> class A: pass
14 ...
```

```
15 >>> callable(A)
16 True
17 >>> callable(A())
18 False
```

4.2. Paramètres de fonctions

4.2.1. Paramètres et arguments

Parlons un peu des paramètres de fonctions (et plus généralement de *callable*). Les paramètres sont décrits lors de la définition de la fonction, ils possèdent un nom et potentiellement une valeur par défaut.

Il faut les distinguer des arguments : les arguments sont les valeurs passées lors de l'appel.

4.2.1.1. Paramètres

```
1 def function(a, b, c, d=1, e=2):
2     return
```

`a`, `b`, `c`, `d` et `e` sont les paramètres de la fonction `function`. `a`, `b` et `c` n'ont pas de valeur par défaut, il faut donc préciser explicitement une valeur lors de l'appel, pour que celui-ci soit valide. Les paramètres avec valeur par défaut se placent obligatoirement après les autres.

4.2.1.2. Arguments

```
1 function(3, 4, d=5, c=3)
```

Nous sommes là dans un appel de fonction, donc les valeurs sont des arguments. `3` et `4` sont des arguments positionnels, car ils sont repérés par leur position, et seront donc associés aux deux premiers paramètres de la fonction (`a` et `b`). `d=5` et `c=3` sont des arguments nommés, car la valeur est précédée du nom du paramètre associé. Ils peuvent ainsi être placés dans n'importe quel ordre (pour peu qu'ils soient placés après les arguments positionnels).

Des paramètres avec valeur par défaut peuvent recevoir des arguments positionnels, et des paramètres sans valeur par défaut peuvent recevoir des arguments nommés. Les deux notions, même si elles partagent une notation commune, sont distinctes.

Voici enfin différents cas d'appels posant problème :

III. Ainsi font fonctions

```
1 >>> function(1) # Pas assez d'arguments
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 TypeError: function() missing 2 required positional arguments: 'b'
   and 'c'
5 >>> function(1, 2, 3, 4, 5, 6) # Trop d'arguments
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 TypeError: function() takes from 3 to 5 positional arguments but 6
   were given
9 >>> function(1, b=2, 3) # Mélange d'arguments positionnels et
   nommés
10  File "<stdin>", line 1
11 SyntaxError: non-keyword arg after keyword arg
12 >>> function(1, 2, b=3, c=4) # b est à la fois positionnel et nommé
13 Traceback (most recent call last):
14  File "<stdin>", line 1, in <module>
15 TypeError: function() got multiple values for argument 'b'
```

4.2.2. Opérateur *splat*, le retour

On retrouve l'opérateur *splat* que nous avons vu lors des assignations dans le chapitre sur les Itérables. Il nous permet ici de récupérer la liste (ou plus précisément le `tuple`) des arguments positionnels passés lors d'un appel, on appelle cela le *packing*.

```
1 >>> def func(*args): # Il est conventionnel d'appeler args la liste
   ainsi récupérée
2 ...     print(args)
3 ...
4 >>> func(1, 2, 3, 'a', 'b', None)
5 (1, 2, 3, 'a', 'b', None)
```

La présence d'`*args` n'est pas incompatible avec celle d'autres paramètres.

```
1 >>> def func(foo, bar, *args):
2 ...     print(foo)
3 ...     print(bar)
4 ...     print(args)
5 ...
6 >>> func(1, 2, 3, 'a', 'b', None)
7 1
8 2
9 (3, 'a', 'b', None)
```

III. Ainsi font fonctions

Les paramètres placés avant `*args` pourront toujours recevoir des arguments positionnels comme nommés. Mais ceux placés après ne seront éligibles qu'aux arguments nommés (puisque `*args` aura récupéré le reste des positionnels).

```
1 >>> def func(foo, *args, bar):
2     ...     print(foo)
3     ...     print(args)
4     ...     print(bar)
5     ...
6 >>> func(1, 2, 3, 'a', 'b', None)
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9 TypeError: func() missing 1 required keyword-only argument: 'bar'
10 >>> func(1, 2, 3, 'a', 'b', None, bar='c')
11 1
12 (2, 3, 'a', 'b', None)
13 c
```

On notera aussi que *splat* peut s'utiliser sans nom de paramètre, pour marquer une distinction entre les types d'arguments.

```
1 def func(foo, *, bar):
2     print(foo)
3     print(bar)
```

Ici, aucune récupération de la liste des arguments nommés n'est opérée. Mais `foo`, placé à gauche de `*`, peut prendre un argument positionnel ou nommé, alors que `bar` placé à droite ne peut recevoir qu'un argument nommé.

```
1 >>> func(1, bar=2)
2 1
3 2
4 >>> func(foo=1, bar=2)
5 1
6 2
7 >>> func(bar=2, foo=1)
8 1
9 2
```

Je disais plus haut que les paramètres avec valeur par défaut se plaçaient obligatoirement après ceux sans. Ceci est à nuancer avec l'opérateur *splat*, qui sépare en deux parties la liste des paramètres : cela n'est vrai qu'à gauche du *splat*. À droite, cela n'importe pas puisque tous les paramètres recevront des arguments nommés (donc sans notion d'ordre).

```
1 >>> def f(a, b=1, *, d): pass
2 ...
3 >>> def f(a, b=1, *, d=2): pass
4 ...
5 >>> def f(a, b=1, *, d=2, e): pass
6 ...
7 >>> def f(a, b=1, *, d=2, e, f=3): pass
8 ...
9 >>> def f(a, b=1, *, d, e=2, f, g=3): pass
10 ...
```

4.2.3. Le double-splat

Enfin, outre l'opérateur *splat* que nous connaissions déjà, on découvre ici le double-*splat* (**). Cet opérateur sert à récupérer le dictionnaire des arguments nommés. Celui-ci doit se placer après tous les paramètres, et se nomme habituellement `kwargs`.

```
1 >>> def func(a, b=1, **kwargs):
2 ...     print(kwargs)
3 ...
4 >>> func(0)
5 {}
6 >>> func(0, b=2)
7 {}
8 >>> func(0, c=3)
9 {'c': 3}
```

En combinant ces deux opérateurs, une fonction est donc en mesure de récupérer l'ensemble de ses arguments (positionnels et nommés).

```
1 def func(*args, **kwargs):
2     pass
```

On notera que contrairement au simple *splat* qui servait aussi pour les assignations, le double n'a aucune signification en dehors des arguments de fonctions.

4.2.4. L'appel du *splat*

Mais ces opérateurs servent aussi lors de l'appel à une fonction, via l'*unpacking*. Comme pour les assignations étudiées dans le chapitre des itérables, il est possible de transformer un itérable en arguments positionnels avec l'opérateur *splat*.

III. Ainsi font fonctions

Le double-*splat* nous permet aussi ici de transformer un dictionnaire (ou autre *mapping*) en arguments nommés.

```
1 >>> def addition_3(a, b, c):
2 ...     return a + b + c
3 ...
4 >>> addition_3(*[1, 2, 3])
5 6
6 >>> addition_3(1, *[2, 3])
7 6
8 >>> addition_3(**{'b': 2, 'a': 1, 'c': 3})
9 6
10 >>> addition_3(1, **{'b': 2, 'c': 3})
11 6
12 >>> addition_3(1, *[2], **{'c': 3})
13 6
14 >>> addition_3(*range(3)) # Splat est valable avec tous les
    itérables
15 3
```

Ainsi, il est possible de relayer les paramètres reçus par une fonction à une autre fonction, sans les préciser explicitement.

```
1 >>> def proxy_addition_3(*args, **kwargs):
2 ...     return addition_3(*args, **kwargs)
3 ...
4 >>> proxy_addition_3(1, 2, 3)
5 6
6 >>> proxy_addition_3(1, c=3, b=2)
7 6
```

Avant Python 3.5, chaque opérateur *splat* ne pouvait être utilisé qu'une fois dans un appel, et * devait être placé après tous les arguments positionnels.

Ces règles ont depuis disparu, comme relaté dans [cet article sur la sortie de Python 3.5](#) .

```
1 >>> addition_3(*[1, 2], 3)
2 6
3 >>> addition_3(*[1], 2, *[3])
4 6
5 >>> addition_3(*[1], **{'b': 2}, **{'c': 3})
6 6
```

4.3. Call-me maybe

Je vous le disais, plusieurs types d'objets peuvent être appelés. Que cache donc un *callable* ? Encore une fois, il s'agit d'un objet qui possède une méthode spéciale. La méthode est ici `__call__`, dont les paramètres seront les arguments passés lors de l'appel. La valeur renvoyée par `__call__` sera le retour de l'appel.

Ainsi, testons avec divers objets :

```
1 >>> def func(arg): return arg
2 ...
3 >>> func(1)
4 1
5 >>> func.__call__(1)
6 1
7 >>> (lambda x: x + 1)(1)
8 2
9 >>> (lambda x: x + 1).__call__(1)
10 2
```

Mais, vous devez vous dire, si on peut appeler `func.__call__`, c'est que `func.__call__` est un *callable*, qui possède donc sa propre méthode `__call__` ? C'est le cas, et l'on peut continuer ainsi indéfiniment.

```
1 >>> func.__call__.__call__(1)
2 1
3 >>> func.__call__.__call__.__call__.__call__.__call__.__call__(1)
4 1
```

Cela s'explique par le fait que `__call__` est une méthode, donc un *callable*. En interne, Python est capable d'identifier qu'il s'agit d'une fonction et d'en exécuter le code, pour ne pas avoir à appeler indéfiniment des `__call__`.

Maintenant, implémentons `__call__` dans un objet de notre création :

```
1 class MyCallable:
2     def __init__(self, a):
3         self.a = a
4
5     def __call__(self, b):
6         return self.a + b
```

Nous avons là une classe `MyCallable`, dont les instances sont des *callables*, réalisant la somme du paramètre reçu à la construction avec celui reçu lors de l'appel.

III. Ainsi font fonctions

```
1 >>> add_3 = MyCallable(3)
2 >>> add_3(5)
3 8
4 >>> add_3(10)
5 13
6 >>> add_100 = MyCallable(100)
7 >>> add_100(2)
8 102
9 >>> MyCallable(6)(3)
10 9
```

Il y a différents intérêts à créer un type *callable*. Le premier serait simplement de rendre compatible notre objet à l'interface utilisée par de nombreuses fonctions Python que nous verrons dans la section suivante. Aussi, utiliser une classe pour cela est un moyen simple de sauvegarder un état, permettant d'avoir un comportement différent à chaque appel.

```
1 >>> class Increment:
2 ...     def __init__(self):
3 ...         self.n = 0
4 ...     def __call__(self):
5 ...         self.n += 1
6 ...         return self.n
7 ...
8 >>> incr = Increment()
9 >>> incr()
10 1
11 >>> incr()
12 2
13 >>> Increment()() # Les deux objets sont bien indépendants
14 1
15 >>> incr()
16 3
```

4.4. Utilisation des callables

De même que pour les itérables, les *callables* sont au cœur de Python en pouvant être utilisés avec un grand nombre de *builtins*.

Par exemple, la fonction `max` évoquée dans un précédent chapitre : en plus de prendre un itérable sur lequel trouver le maximum, elle peut aussi prendre un paramètre `key`. Ce paramètre est un *callable* expliquant comment extraire le maximum depuis les arguments passés à `max`.

Si l'itérable ne contient que des entiers, il est plutôt simple de déterminer le maximum, mais si nous avons une liste de points 2D par exemple ? Le maximum pourrait être le point avec la

III. Ainsi font fonctions

plus grande abscisse, la plus grande ordonnée, le point le plus éloigné de l'origine du repère, ou encore bien d'autres choses.

Le *callable* `key` est donc chargé de calculer une valeur numérique pour chacun des paramètres, et pouvoir ainsi les comparer entre-eux. Le paramètre ayant la plus grande valeur sera le maximum.

Nous représenterons ici nos points par des tuples de deux valeurs.

```
1 >>> points = [(0, 0), (1, 4), (3, 3), (4, 0)]
2 >>> max(points) # par défaut, Python sélectionne suivant le premier
   élément, soit l'abscisse
3 (4, 0)
4 >>> max(points, key=lambda p: p[0]) # Nous précisons ici
   explicitement la sélection par l'abscisse
5 (4, 0)
6 >>> max(points, key=lambda p: p[1]) # Par ordonnée
7 (1, 4)
8 >>> max(points, key=lambda p: p[0]**2 + p[1]**2) # Par distance de
   l'origine
9 (3, 3)
```

En dehors de `max`, d'autres fonctions Python prennent un tel paramètre `key`, comme `min` ou encore `sorted` :

```
1 >>> sorted(points, key=lambda p: p[1])
2 [(0, 0), (4, 0), (3, 3), (1, 4)]
3 >>> sorted(points, key=lambda p: p[0]**2 + p[1]**2)
4 [(0, 0), (4, 0), (1, 4), (3, 3)]
```

`map`, que nous avons déjà vu, prend lui aussi un *callable* de n'importe quel type.

```
1 >>> list(map(lambda p: p[0], points))
2 [0, 1, 3, 4]
3 >>> list(map(lambda p: p[1], points))
4 [0, 4, 3, 0]
```

Je vous invite une nouvelle fois à jeter un œil aux [builtins Python](#) , ainsi qu'au [module itertools](#) , et de voir lesquels peuvent vous faire tirer profit des *callables*.

4.5. Modules operator et functools

Passons maintenant à la présentation de deux modules, contenant deux collections de *callables*.

4.5.1. operator

Ce premier module, `operator` [↗](#) regroupe l'ensemble des opérateurs Python sous forme de fonctions. Ainsi, une addition pourrait se formuler par :

```
1 >>> import operator
2 >>> operator.add(3, 4)
3 7
```

Outre les opérateurs habituels, nous en trouvons d'autres sur lesquels nous allons nous intéresser plus longuement, dont la particularité est de retourner des *callable*s.

4.5.1.1. itemgetter

`itemgetter` permet de récupérer un élément précis depuis un indexable, à la manière de l'opérateur `[]`.

```
1 >>> get_second = operator.itemgetter(1) # Récupère le 2ème élément
   de l'indexable donné en argument
2 >>> get_second([5, 8, 0, 3, 1])
3 8
4 >>> get_second('abcdef')
5 b
6 >>> get_second(range(3,10))
7 4
8 >>> get_x = operator.itemgetter('x')
9 >>> get_x({'x': 5, 'y': 1})
10 5
```

4.5.1.2. methodcaller

`methodcaller` permet d'appeler une méthode prédéterminée d'un objet, avec ses arguments.

```
1 >>> dash_splitter = operator.methodcaller('split', '-')
2 >>> dash_splitter('a-b-c-d')
3 ['a', 'b', 'c', 'd']
4 >>> append_b = operator.methodcaller('append', 'b')
5 >>> l = [0, 'a', 4]
6 >>> append_b(l)
7 >>> l
8 [0, 'a', 4, 'b']
```

4.5.2. `functools`

Je tenais ensuite à évoquer le module `functools` [↗](#), et plus particulièrement la fonction `partial` : celle-ci permet de réaliser un appel partiel de fonction.

Imaginons que nous ayons une fonction prenant divers paramètres, mais que nous voudrions fixer le premier : l'application partielle de la fonction nous créera un nouveau *callable* qui, quand il sera appelé avec de nouveaux arguments, nous renverra le résultat de la première fonction avec l'ensemble des arguments.

Par exemple, prenons une fonction de journalisation `log` prenant quatre paramètres, un niveau de gravité, un type, un objet, et un message descriptif :

```
1 def log(level, type, item, message):
2     print('[{}]<{}>({}): {}'.format(level.upper(), type, item,
      message))
```

Une application partielle reviendrait à avoir une fonction `warning` tel que chaque appel `warning('foo', 'bar', 'baz')` équivaldrait à `log('warning', 'foo', 'bar', 'baz')`. Ou encore une fonction `warning_foo` avec `warning_foo('bar', 'baz')` équivalent à l'appel précédent.

Nous allons la tester avec une fonction du module `operator` : la fonction de multiplication. En appliquant partiellement `5` à la fonction `operator.mul`, `partial` nous retourne une fonction réalisant la multiplication par `5` de l'objet passé en paramètre.

```
1 >>> from functools import partial
2 >>> mul_5 = partial(operator.mul, 5)
3 >>> mul_5(3)
4 15
5 >>> mul_5('z')
6 'zzzzz'
7 >>> warning = partial(log, 'warning')
8 >>> warning('overflow', 100, 'number is too large')
9 [WARNING]<overflow>(100): number is too large
10 >>> overflow = partial(log, 'warning', 'overflow')
11 >>> overflow(-1, 'number is too low')
12 [WARNING]<overflow>(-1): number is too low
```

Le module `functools` comprend aussi la fonction `reduce`, un outil tiré du fonctionnel permettant de transformer un itérable en une valeur unique. Pour cela, elle itère sur l'ensemble et applique une fonction à chaque valeur, en lui précisant la valeur courante et le résultat précédent.

Imaginons par exemple que nous disposions d'une liste de nombres `[5, 8, 0, 3, 1]` et que nous voulions en calculer la somme. Nous savons faire la somme de deux nombres, il s'agit d'une addition, donc de la fonction `operator.add`.

III. Ainsi font fonctions

`reduce` va nous permettre d'appliquer `operator.add` sur les deux premiers éléments ($5 + 8 = 13$), réappliquer la fonction avec ce premier résultat et le prochain élément de la liste ($13 + 0 = 13$), puis avec ce résultat et l'élément suivant ($13 + 3 = 16$), et enfin, sur ce résultat et le dernier élément ($16 + 1 = 17$).

Ce processus se résume en :

```
1 >>> from functools import reduce
2 >>> reduce(operator.add, [5, 8, 0, 3, 1])
3 17
```

qui revient donc à faire

```
1 >>> operator.add(operator.add(operator.add(operator.add(5, 8), 0),
2 3), 1)
2 17
```

Bien sûr, en pratique, `sum` est déjà là pour répondre à ce problème.

4.6. TP : `itemgetter`

Dans ce nouveau TP, nous allons réaliser `itemgetter` à l'aide d'une classe formant des objets *callable*s.

La clef à récupérer est passée à l'instanciation de l'objet `itemgetter`, et donc à son constructeur. L'objet depuis lequel nous voulons récupérer la clef sera passé lors de l'appel (dans la méthode `__call__`). Il nous suffit alors, dans cette méthode, d'appeler l'opérateur `[]` sur l'objet avec la clef enregistrée au moment de la construction.

```
1 class itemgetter:
2     def __init__(self, key):
3         self.key = key
4
5     def __call__(self, obj):
6         return obj[self.key]
```

C'est aussi simple que cela, et nous pouvons le tester :

```
1 >>> points = [(0, 0), (1, 4), (3, 3), (4, 0)]
2 >>> sorted(points, key=itemgetter(0))
3 [(0, 0), (1, 4), (3, 3), (4, 0)]
4 >>> sorted(points, key=itemgetter(1))
```

III. Ainsi font fonctions

```
5 [(0, 0), (4, 0), (3, 3), (1, 4)]
```

Nous aurions aussi pu profiter des fermetures (*closures*) de Python pour réaliser `itemgetter` sous la forme d'une fonction retournant une fonction.

```
1 def itemgetter(key):
2     def function(obj):
3         return obj[key]
4     return function
```

Si vous vous êtes intéressés de plus près à `operator.itemgetter`, vous avez aussi pu remarquer que celle-ci pouvait prendre plus d'un paramètre :

```
1 >>> from operator import itemgetter
2 >>> get_x = itemgetter('x')
3 >>> get_x_y = itemgetter('x', 'y')
4 >>> get_x({'x': 9, 'y': 6})
5 9
6 >>> get_x_y({'x': 9, 'y': 6})
7 (9, 6)
```

Je vous propose, pour aller un peu plus loin, d'ajouter cette fonctionnalité à notre classe, et donc d'utiliser les listes d'arguments positionnels. Vous trouverez la solution dans la [documentation du module operator](#) [↗](#).

Voici donc quelques liens relatifs aux *callable*s.

- Définition du terme paramètre : <https://docs.python.org/3/glossary.html#term-parameter> [↗](#)
- Et du terme argument : <https://docs.python.org/3/glossary.html#term-argument> [↗](#)
- Types callable : <https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy> [↗](#)
- Émuler les callable : <https://docs.python.org/3/reference/datamodel.html#emulating-callable-objects> [↗](#)
- Syntaxe des appels : <https://docs.python.org/3/reference/expressions.html#calls> [↗](#)
- Module `operator` : <https://docs.python.org/3/library/operator.html> [↗](#)
- Module `functools` : <https://docs.python.org/3/library/functools.html> [↗](#)
- Description des *trucmunchables* : <http://sametmax.com/les-trucmunchables-en-python/> [↗](#)

5. Annotations et signatures

Intéressons-nous maintenant à ce qui entoure les fonctions. Une fonction n'est pas seulement un nom ou un bloc de code à exécuter. Elle possède aussi des informations complémentaires comme des noms de paramètres, des annotations, ou une *docstring*.

Ce chapitre s'intéresse justement à ces informations : comment les définir, mais surtout comment y accéder et ce que l'on peut en faire.

5.1. Annotations

Commençons par les annotations. Il se peut que vous ne les ayez jamais rencontrées, il s'agit d'une fonctionnalité relativement nouvelle du langage.

Les annotations sont des informations de types que l'on peut ajouter sur les paramètres et le retour d'une fonction.

Prenons une fonction d'addition entre deux nombres.

```
1 def addition(a, b):  
2     return a + b
```

Nous avons destiné cette fonction à des calculs numériques, mais nous pourrions aussi l'appeler avec des chaînes de caractères. Les annotations vont nous permettre de préciser le type des paramètres attendus, et le type de la valeur de retour.

```
1 def addition(a: int, b: int) -> int:  
2     return a + b
```

Leur définition est assez simple : pour annoter un paramètre, on le fait suivre d'un `:` et on ajoute le type attendu. Pour annoter le retour de la fonction, on ajoute `->` puis le type derrière la liste des paramètres.

Attention cependant, les annotations ne sont là qu'à titre indicatif. Rien n'empêche de continuer à appeler notre fonction avec des chaînes de caractères.

À ce titre, on notera aussi que donner des types comme annotations n'est qu'une convention. Annoter des paramètres avec des chaînes de caractères ne provoquera pas d'erreur par exemple.

5.1.1. Utilité des annotations

Si elles ne sont là qu'à titre indicatif, les annotations sont surtout utiles dans un but de documentation. Elles sont le meilleur moyen en Python de documenter les types des paramètres et de retour d'une fonction.

Elles apparaissent d'ailleurs dans l'aide de la fonction fournie par `help`.

```
1 >>> help(addition)
2 Help on function addition in module __main__:
3
4 addition(a:int, b:int) -> int
```

Les annotations ne sont pas censées avoir d'autre utilité lors de l'exécution d'un programme Python. Elles ne sont pas destinées à vérifier au *runtime* le type des paramètres par exemple. La définition d'annotations ne doit normalement rien changer sur le déroulement du programme.

Toutefois, les annotations ont l'utilité que l'on veut bien leur donner. Il existe des outils d'analyse statique tels que `mypy` qui peuvent en tirer partie. Ces outils n'exécutent pas le code, mais se contentent de vérifier que les types utilisés n'entrent pas en conflit avec les annotations.

5.1.2. Des types plus complexes (module `typing`)

Nous avons défini une fonction `addition` opérant sur deux nombres, mais l'avons annotée comme ne pouvant recevoir que des nombres entiers (`int`).

En effet, les annotations utilisées jusqu'ici étaient plutôt simples. Mais elles peuvent accueillir des expressions plus complexes.

Le [module `typing`](#) nous présente une collection de classes pour composer des types. Ce module a été introduit dans Python 3.5, et n'est donc pas disponible dans les versions précédentes du langage.

Dans notre fonction `addition`, nous voudrions en fait que les `int`, `float` et `complex` soient admis. Nous pouvons pour cela utiliser le type `Union` du module `typing`. Il nous permet de définir un ensemble de types valides pour nos paramètres, et s'utilise comme suit.

```
1 Number = Union[int, float, complex]
2
3 def addition(a: Number, b: Number) -> Number:
4     return a + b
```

Nous définissons premièrement un type `Number` comme l'ensemble des types `int`, `float` et `complex` *via* la syntaxe `Union[...]`. Puis nous utilisons notre nouveau type `Number` au sein de nos annotations.

III. Ainsi font fonctions

Outre `Union`, le module `typing` présente d'autres types génériques pour avoir des annotations plus précises. Nous pourrions avoir, par exemple :

- `List[str]` – Une liste de chaînes de caractères ;
- `Sequence[str]` – Une séquence (liste/tuple/etc.) de chaînes de caractères ;
- `Callable[[str, int], str]` – Un *callable* prenant deux paramètres de types `str` et `int` respectivement, et retournant un objet `str` ;
- Et bien d'autres à découvrir dans la [documentation du module](#) `typing`.

Attention encore, le module `typing` ne doit servir que dans le cadre des annotations. Les types fournis par ce module ne doivent pas être utilisés au sein d'expressions avec `isinstance` ou `issubclass`.

Dans le cas précis de notre fonction `addition`, nous aurions aussi pu utiliser le type `Number` du module `numbers`. Nous y reviendrons plus tard dans ce cours, mais il s'agit d'un type qui regroupe et hiérarchise tous les types numériques.

5.1.3. Annotations de variables

Les annotations sont ici abordées sous l'angle des fonctions et de leurs paramètres. Mais il est à noter que depuis Python 3.6, il est aussi possible d'annoter les variables et attributs.

La syntaxe est la même que pour les paramètres de fonction. Encore une fois, les annotations sont là à titre indicatif, et pour les analyseurs statiques. Elles sont toutefois stockées dans le dictionnaire `__annotations__` du module ou de la classe qui les contient.

```
1 max_value : int = 10 # Définition d'une variable max_value annotée
   comme int
2 min_value : int      # Annotation seule, la variable n'est pas
   définie dans ce cas
```

Un petit coup d'œil à la variable `__annotations__` et aux variables annotées.

```
1 >>> __annotations__
2 {'max_value': <class 'int'>, 'min_value': <class 'int'>}
3 >>> max_value
4 10
5 >>> min_value
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   NameError: name 'min_value' is not defined
```


5.2. Inspecteur Gadget

Nous savons maintenant renseigner des informations complémentaires sur nos fonctions. En plus des annotations vues précédemment, vous avez probablement déjà rencontré les *docstrings*.

Les *docstrings* sont des chaînes de caractères, à placer en tête d'une fonction, d'une classe ou d'un module. Elles servent à décrire l'usage de ces objets, et sont accessibles dans l'aide fournie par `help`, au même titre que les annotations.

```
1 def addition(a:int, b:int) -> int:
2     "Return the sum of the two numbers `a` and `b`"
3     return a + b
```

```
1 >>> help(addition)
2 Help on function addition in module __main__:
3
4 addition(a:int, b:int) -> int
5     Return the sum of the two numbers `a` and `b`
```

Elles deviennent aussi accessibles par l'attribut spécial `__doc__` de l'objet.

```
1 >>> addition.__doc__
2 'Return the sum of the two numbers `a` and `b`'
```

5.2.1. Module inspect

`inspect` [↗](#) est un module de la bibliothèque standard qui permet d'extraire des informations complémentaires sur les objets Python. Il est notamment dédié aux modules, classes et fonctions.

Il comporte en effet des fonctions pour vérifier le type d'un objet : `ismodule`, `isclass`, `isfunction`, etc.

```
1 >>> import inspect
2 >>> inspect.ismodule(inspect)
3 True
4 >>> inspect.isclass(int)
5 True
6 >>> inspect.isfunction(addition)
7 True
8 >>> inspect.isbuiltin(len)
```

III. Ainsi font fonctions

```
9 True
```

D'autres fonctions vont s'intéresser plus particulièrement aux documentations (`getdoc`) et à la gestion du code source (`getsource`, `getsourcefile`, etc.).

Imaginons un fichier `operations.py` contenant le code suivant :

```
1 "Mathematical operations"
2
3 def addition(a:int, b:int) -> int:
4     """
5     Return the sum of the two numbers `a` and `b`
6
7     ex: addition(3, 5) -> 8
8     """
9     return a + b
```

Depuis la fonction `addition` importée du module, nous pourrons grâce à `inspect` récupérer toutes les informations nécessaires au débogage.

```
1 >>> import inspect
2 >>> from operations import addition
3 >>> inspect.getdoc(addition)
4 'Return the sum of the two numbers `a` and `b`\n\nex: addition(3, 5) -> 8'
5 >>> inspect.getsource(addition)
6 'def addition(a:int, b:int) -> int:\n    [...]\n    return a + b\n'
7 >>> inspect.getsourcefile(addition)
8 '/home/entwanne/operations.py'
9 >>> inspect.getmodule(addition)
10 <module 'operations' from '/home/entwanne/operations.py'>
11 >>> inspect.getsourcelines(addition)
12 (['def addition(a:int, b:int) -> int:\n', ...,
    '    return a + b\n'], 3)
```

L'intérêt de `getdoc` par rapport à l'attribut `__doc__` étant que la documentation est nettoyée (suppression des espaces en début de ligne) par la fonction `cleandoc`.

```
1 >>> addition.__doc__
2 '\n    Return the sum of the two numbers `a` and `b`\n\n    ex: addition(3, 5)'
3 >>> inspect.cleandoc(addition.__doc__)
4 'Return the sum of the two numbers `a` and `b`\n\nex: addition(3, 5) -> 8'
```

5.3. Signatures

Le module `inspect` ne nous a pas encore révélé toutes ses surprises. Il contient aussi une méthode `signature`, retournant la signature d'une fonction.

La signature est l'ensemble des paramètres (avec leurs noms, positions, valeurs par défaut et annotations), ainsi que l'annotation de retour d'une fonction. C'est-à-dire toutes les informations décrites à droite du nom de fonction lors d'une définition.

```
1 >>> sig = inspect.signature(addition)
2 >>> print(sig)
3 (a:int, b:int) -> int
```

Les objets retournés par `signature` sont de type `Signature`. Ces objets comportent notamment un dictionnaire ordonné des paramètres de la fonction (attribut `parameters`), et l'annotation de retour (`return_annotation`).

Les paramètres sont un encore un nouveau type d'objets, `Parameter`. Les objets `Parameter` possèdent un nom (`name`), une valeur par défaut (`default`), une annotation (`annotation`), et un type de positionnement (`kind`).

```
1 >>> sig.return_annotation
2 <class 'int'>
3 >>> for param in sig.parameters.values():
4 ...     print(param.name, param.default, param.annotation,
5 ...           param.kind)
6 a <class 'inspect._empty'> <class 'int'> POSITIONAL_OR_KEYWORD
7 b <class 'inspect._empty'> <class 'int'> POSITIONAL_OR_KEYWORD
```

On retrouve bien les noms et annotations de nos paramètres. `inspect._empty` indique que nos paramètres ne prennent pas de valeurs par défaut.

Le type de positionnement correspond à la différence entre arguments positionnels et arguments nommés, ils sont au nombre de 5 :

- `POSITIONAL_ONLY` – Le paramètre ne peut recevoir qu'un argument positionnel ;
- `POSITIONAL_OR_KEYWORD` – Le paramètre peut indifféremment recevoir un argument positionnel ou nommé ;
- `VAR_POSITIONAL` – Correspond au paramètre spécial `*args` ;
- `KEYWORD_ONLY` – Le paramètre ne peut recevoir qu'un argument nommé ;
- `VAR_KEYWORD` – Correspond au paramètre spécial `**kwargs`.

Il n'existe aucune syntaxe en Python pour définir des paramètres *positional-only*, ils existent cependant dans certaines *builtins* (`range` par exemple).

Les *positional-or-keyword* sont les plus courants. Ils sont en fait tous les paramètres définis à gauche d'`*args`.

III. Ainsi font fonctions

Les *keyword-only* sont ceux définis à sa droite.

Pour prendre une signature de fonction plus complète :

```
1 >>> def function(a, b:int, c=None, d:int=0, *args, g, h:int,
2     ...         pass
3     ...
4 >>> for param in inspect.signature(function).parameters.values():
5     ...     print(param.name, param.default, param.annotation,
6     ...           param.kind)
7 a <class
8     'inspect._empty'> <class 'inspect._empty'> POSITIONAL_OR_KEYWORD
9 b <class 'inspect._empty'> <class 'int'> POSITIONAL_OR_KEYWORD
10 c None <class 'inspect._empty'> POSITIONAL_OR_KEYWORD
11 d 0 <class 'int'> POSITIONAL_OR_KEYWORD
12 args <class
13     'inspect._empty'> <class 'inspect._empty'> VAR_POSITIONAL
14 g <class 'inspect._empty'> <class 'inspect._empty'> KEYWORD_ONLY
15 h <class 'inspect._empty'> <class 'int'> KEYWORD_ONLY
16 i foo <class 'inspect._empty'> KEYWORD_ONLY
17 j 5.0 <class 'float'> KEYWORD_ONLY
18 kwargs <class
19     'inspect._empty'> <class 'inspect._empty'> VAR_KEYWORD
```

Rappelons qu'il est possible d'avoir des paramètres *keyword-only* sans pour autant définir `*args`. Il faut pour cela avoir un simple `*` dans la liste des paramètres, juste à gauche des *keyword-only*.

```
1 >>> def function(a, *, b, c):
2     ...     pass
3     ...
4 >>> for param in inspect.signature(function).parameters.values():
5     ...     print(param.name, param.kind)
6     ...
7 a POSITIONAL_OR_KEYWORD
8 b KEYWORD_ONLY
9 c KEYWORD_ONLY
```

5.3.1. Arguments préparés

Nous venons de voir quelles informations nous pouvons tirer des signatures. Mais celles-ci ne servent pas qu'à la documentation.

III. Ainsi font fonctions

Les objets `Signature` sont aussi pourvus d'une méthode `bind`. Cette méthode reçoit les mêmes arguments que la fonction cible et retourne un objet de type `BoundArguments`, qui fait la correspondance entre les paramètres et les arguments, après vérification que ces derniers respectent la signature.

L'objet `BoundArguments`, avec ses attributs `args` et `kwargs`, pourra ensuite être utilisé pour appeler la fonction cible.

```
1 >>> sig = inspect.signature(addition)
2 >>> bound = sig.bind(3, b=5)
3 >>> bound.args
4 (3, 5)
5 >>> bound.kwargs
6 {}
7 >>> addition(*bound.args, **bound.kwargs)
8 8
```

Comme on peut le voir, cela permet de résoudre les paramètres pour n'avoir dans `kwargs` que les *keyword-only*, et les autres (ceux qui peuvent être positionnels) dans `args`. Cet objet `BoundArguments` sera donc identique quelle que soit la manière dont seront passés les arguments.

```
1 >>> sig.bind(3, 5) == sig.bind(b=5, a=3)
2 True
```

Nous pouvons ainsi avoir une représentation unique des arguments de l'appel, pouvant servir pour une mise en cache des résultats par exemple.

```
1 cache_addition = {}
2 sig_addition = inspect.signature(addition)
3
4 def addition_with_cache(*args, **kwargs):
5     bound = sig_addition.bind(*args, **kwargs)
6     # addition ne reçoit aucun paramètre keyword-only
7     # nous pouvons donc nous contenter de bound.args
8     if bound.args in cache_addition:
9         print('Retrieving from cache')
10        return cache_addition[bound.args]
11    print('Computing result')
12    result = cache_addition[bound.args] = addition(*bound.args)
13    return result
```

```
1 >>> addition_with_cache(3, 5)
2 Computing result
```

III. Ainsi font fonctions

```
3 8
4 >>> addition_with_cache(3, b=5)
5 Retrieving from cache
6 8
7 >>> addition_with_cache(b=5, a=3)
8 Retrieving from cache
9 8
10 >>> addition_with_cache(5, 3) # Le résultat de (a=5, b=3) n'est pas
    en cache
11 Computing result
12 8
```

Depuis Python 3.5, une autre fonctionnalité des `BoundArguments` est de pouvoir appliquer les valeurs par défaut aux paramètres. Ils ont pour cela une méthode `apply_defaults`.

```
1 >>> def addition_with_default(a, b=3):
2 ...     return a + b
3 ...
4 >>> sig = inspect.signature(addition_with_default)
5 >>> bound = sig.bind(5)
6 >>> bound.args
7 (5,)
8 >>> bound.apply_defaults()
9 >>> bound.args
10 (5, 3)
```

5.3.2. Méthode `replace`

Je voudrais enfin vous parler des méthodes `replace` des objets `Signature` et `Parameter`. Ces méthodes permettent de retourner une copie de l'objet en en modifiant un ou plusieurs attributs.

Nous pourrions sur un paramètre modifier les valeurs `name`, `kind`, `default` et `annotation`.

```
1 >>> param = sig.parameters['b']
2 >>> print(param)
3 b=3
4 >>> new_param = param.replace(name='c', default=4, annotation=int)
5 >>> print(new_param)
6 c:int=4
7 >>> print(param)
8 b=3
```

III. Ainsi font fonctions

La méthode `replace` des signatures est similaire, et permet de modifier `parameters` et `return_annotation`.

```
1 >>> print(sig)
2 (a, b=3)
3 >>> new_params = [sig.parameters['a'], new_param]
4 >>> new_sig = sig.replace(parameters=new_params)
5 >>> print(new_sig)
6 (a, c:int=4)
```

5.4. TP : Vérification de signature

Afin de nous exercer un peu sur les signatures, nous allons créer une fonction vérifiant la validité d'un ensemble d'arguments. Cette fonction, `signature_check`, recevra en paramètres la signature à tester, et tous les arguments positionnels et nommés de l'appel.

5.4.1. Prototype

On pourrait à première vue la prototyper ainsi.

```
1 def signature_check(signature, *args, **kwargs):
2     ...
```

Mais ce serait oublier que `signature` peut aussi posséder un paramètre nommé `signature`, qui ne pourrait alors être testé. Nous allons donc plutôt récupérer la signature à tester depuis `*args`.

```
1 def signature_check(*args, **kwargs):
2     signature, *args = args # Soit signature = args[0] et args =
3     list(args[1:])
4     ...
```

Passons maintenant au contenu de notre fonction. Celle-ci va devoir itérer sur les paramètres, pour leur faire correspondre les arguments. On lèvera des `TypeError` en cas de non-correspondance ou de manque/surplus d'arguments.

Le plus simple est de procéder en consommant la liste `args` et le dictionnaire `kwargs`. Les arguments seront supprimés chaque fois qu'une correspondance sera faite avec un paramètre.

S'il reste des arguments après itération de tous les paramètres, c'est qu'il y a surplus. On lèvera donc une erreur pour l'indiquer.

III. Ainsi font fonctions

On obtient alors un premier squelette.

```
1 def signature_check(*args, **kwargs):
2     sig, *args = args
3     for param in sig.parameters.values():
4         ...
5     if args:
6         raise TypeError('too many positional arguments')
7     if kwargs:
8         raise TypeError('too many keyword arguments')
```

5.4.2. Paramètres

Attachons-nous maintenant à remplacer ces points de suspension par le traitement des paramètres. Pour rappel, on retrouve 5 sortes de paramètres, qui seront traitées différemment.

5.4.2.1. VAR_POSITIONAL

Si un paramètre `VAR_POSITIONAL` est présent, il aura pour effet de consommer tous les arguments restant, c'est-à-dire de vider `args`.

```
1 if param.kind == param.VAR_POSITIONAL:
2     args.clear()
```

5.4.2.2. VAR_KEYWORD

De même ici, mais en vidant `kwargs`, puisque tous les arguments nommés sont consommés.

```
1 elif param.kind == param.VAR_KEYWORD:
2     kwargs.clear()
```

5.4.2.3. POSITIONAL_ONLY

Passons aux choses sérieuses avec les paramètres `POSITIONAL_ONLY`. Les arguments correspondant à ce type de paramètres se trouveront nécessairement dans `args`.

Deux cas sont cependant à distinguer :

- Si la liste `args` n'est pas vide, nous en consommons le premier élément ;
- Si elle est vide, il faut alors vérifier que le paramètre possède une valeur par défaut.

III. Ainsi font fonctions

En effet, si aucun argument n'est reçu, le paramètre prend sa valeur par défaut. Mais s'il ne possède aucune valeur par défaut, il faut alors lever une erreur : le paramètre est manquant.

```
1 elif param.kind == param.POSITIONAL_ONLY:
2     if args:
3         del args[0]
4     elif param.default == param.empty: # Ni argument ni valeur par
        défaut
5         raise
            TypeError("Missing '{}' positional argument".format(param.name))
```

5.4.2.4. KEYWORD_ONLY

On retrouve ici un code semblable au *positional-only*, mais en s'intéressant à `kwargs` plutôt qu'à `args`. Le même cas d'erreur est à traiter si le paramètre n'est associé à aucune valeur.

```
1 elif param.kind == param.KEYWORD_ONLY:
2     if param.name in kwargs:
3         del kwargs[param.name]
4     elif param.default == param.empty:
5         raise
            TypeError("Missing '{}' keyword argument".format(param.name))
```

5.4.2.5. POSITIONAL_OR_KEYWORD

Le dernier cas, le plus complexe, est celui des paramètres pouvant recevoir arguments positionnels ou nommés. Nous devons alors tester la présence d'un argument dans chacun des deux conteneurs.

Nous serons confrontés à un nouveau cas d'erreur : si un paramètre est associé à la fois à un argument positionnel et à un nommé.

```
1 else: # POSITIONAL_OR_KEYWORD
2     positional = keyword = False
3     if args: # Prend valeur dans args
4         del args[0]
5         positional = True
6     if param.name in kwargs: # Prend valeur dans kwargs
7         del kwargs[param.name]
8         keyword = True
9     if positional and keyword:
10        raise
            TypeError("Multiple arguments for parameter '{}'.format(param.name))
```

III. Ainsi font fonctions

```
11     if not positional and not keyword and param.default ==
12         param.empty:
13         raise
14             TypeError("Missing argument for '{}' parameter".format(param.name))
```

5.4.3. Résultat

En mettant bout à bout nos morceaux de code, nous obtenons notre fonction `signature_check` pour tester si des arguments correspondent à une signature. La fonction ne retourne rien, mais lève une erreur en cas de non-correspondance.

```
1 def signature_check(*args, **kwargs):
2     sig, *args = args
3     for param in sig.parameters.values():
4         if param.kind == param.VAR_POSITIONAL: # Consomme tous les
5             positionnels
6             args.clear()
7         elif param.kind == param.VAR_KEYWORD: # Consomme tous les
8             nommés
9             kwargs.clear()
10        elif param.kind == param.POSITIONAL_ONLY: # Prend valeur
11            dans args
12            if args:
13                del args[0]
14            elif param.default == param.empty: # Ni argument ni
15                valeur par défaut
16                raise
17                    TypeError("Missing '{}' positional argument".format(param.name))
18        elif param.kind == param.KEYWORD_ONLY: # Prend valeur dans
19            kwargs
20            if param.name in kwargs:
21                del kwargs[param.name]
22            elif param.default == param.empty:
23                raise
24                    TypeError("Missing '{}' keyword argument".format(param.name))
25        else: # POSITIONAL_OR_KEYWORD
26            positional = keyword = False
27            if args: # Prend valeur dans args
28                del args[0]
29                positional = True
30            if param.name in kwargs: # Prend valeur dans kwargs
31                del kwargs[param.name]
32                keyword = True
33            if positional and keyword:
34                raise
35                    TypeError("Multiple arguments for parameter '{}'".format(p
```

III. Ainsi font fonctions

```
28         if not positional and not keyword and param.default ==
29             param.empty:
30             raise
31             TypeError("Missing argument for '{}' parameter".format(param.name))
32     if args:
33         raise TypeError('too many positional arguments')
34     if kwargs:
35         raise TypeError('too many keyword arguments')
```

Et pour voir `signature_check` en action :

```
1 >>> sig = inspect.signature(lambda a, b: None)
2 >>> signature_check(sig, 3, 5)
3 >>> signature_check(sig, 3, b=5)
4 >>> signature_check(sig, 3)
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7   File "signature_check.py", line 37, in signature_check
8     raise
9     TypeError("Missing argument for '{}' parameter".format(param.name))
10 TypeError: Missing argument for 'b' parameter
11 >>> signature_check(sig, 3, 5, 7)
12 Traceback (most recent call last):
13   File "<stdin>", line 1, in <module>
14   File "signature_check.py", line 39, in signature_check
15     raise TypeError('too many positional arguments')
16 TypeError: too many positional arguments
17 >>> signature_check(sig, 3, 5, b=5)
18 Traceback (most recent call last):
19   File "<stdin>", line 1, in <module>
20   File "signature_check.py", line 35, in signature_check
21     raise
22     TypeError("Multiple arguments for parameter '{}'.format(param.name))
23 TypeError: Multiple arguments for parameter 'b'
```

```
1 >>> sig = inspect.signature(lambda a, *, b: None)
2 >>> signature_check(sig, 3, 5)
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   File "signature_check.py", line 25, in signature_check
6     raise
7     TypeError("Missing '{}' keyword argument".format(param.name))
8 TypeError: Missing 'b' keyword argument
9 >>> signature_check(sig, 3, b=5)
10 >>> signature_check(sig, 3, b=5, c=7)
11 Traceback (most recent call last):
```

III. Ainsi font fonctions

```
11 File "<stdin>", line 1, in <module>
12 File "signature_check.py", line 41, in signature_check
13     raise TypeError('too many keyword arguments')
14 TypeError: too many keyword arguments
```

```
1 >>> sig = inspect.signature(lambda *args, file=sys.stdin: None)
2 >>> signature_check(sig, 1, 2, 3, 4)
3 >>> signature_check(sig)
4 >>> signature_check(sig, 1, 2, 3, 4, file=None)
5 >>> signature_check(sig, 1, 2, 3, 4, file=None, foo=0)
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   File "signature_check.py", line 41, in signature_check
9     raise TypeError('too many keyword arguments')
10 TypeError: too many keyword arguments
```

Retrouvons les quelques ressources complémentaires sur ces sujets.

- Module `inspect` : <https://docs.python.org/3/library/inspect.html> ↗
- Annotations de fonctions : <https://www.python.org/dev/peps/pep-3107> ↗
- `mypy`, analyseur statique de code à partir des annotations : <http://mypy-lang.org/> ↗

6. Décorateurs

Dans ce chapitre nous allons découvrir les décorateurs et leur utilisation.

Le nom de décorateur provient du patron de conception [décorateur](#) ⁵ (*pattern decorator*). Un décorateur permet de se soustraire à un objet pour en modifier le comportement.

6.1. D&CO, une semaine pour tout changer

En Python, vous en avez peut-être déjà croisé, les décorateurs se repèrent au caractère @.

Le principe de la décoration en Python est d'appliquer un décorateur à une fonction, afin de retourner un nouvel objet (généralement une fonction). On peut donc voir le décorateur comme une fonction prenant une fonction en paramètre, et retournant une nouvelle fonction⁵

Le décorateur est un *callable* prenant un *callable* en paramètre, et pouvant retourner tout type d'objet.

```
1 def decorator(f): # decorator est un décorateur
2     print(f.__name__)
3     return f
```

Pour appliquer un décorateur, on précède la ligne de définition de la fonction à décorer par une ligne comportant un @ puis le nom du décorateur à appliquer, par exemple :

```
1 >>> @decorator
2 ... def addition(a, b):
3 ...     return a + b
4 ...
5 addition
```

Cela a pour effet de remplacer `addition` par le retour de la fonction `decorator` appelée avec `addition` en paramètre, ce qui est strictement équivalent à :

```
1 def addition(a, b):
2     return a + b
```

5. Bien que la définition soit plus large que cela.

III. Ainsi font fonctions

```
3
4 addition = decorator(addition)
```

On voit donc bien que le décorateur est appliqué au moment de la définition de la fonction, et non lors de ses appels. Nous utilisons ici un décorateur très simple qui retourne la même fonction, mais il se pourrait très bien qu'il en retourne une autre, qui serait par exemple créée à la volée.

Disons que nous aimerions modifier notre fonction `addition` pour afficher les opérandes puis le résultat, sans toucher au corps de notre fonction. Nous pouvons réaliser un décorateur qui retournera une nouvelle fonction se chargeant d'afficher les paramètres, d'appeler notre fonction originale, puis d'afficher le retour et de le retourner (afin de conserver le comportement original).

Ainsi, notre décorateur devient :

```
1 def print_decorator(function):
2     def new_function(a, b): # Nouvelle fonction se comportant comme
        la fonction à décorer
3         print('Addition des nombres {} et {}'.format(a, b))
4         ret = function(a, b) # Appel de la fonction originale
5         print('Retour: {}'.format(ret))
6         return ret
7     return new_function # Ne pas oublier de retourner notre
        nouvelle fonction
```

Si on applique maintenant ce décorateur à notre fonction d'addition :

```
1 >>> @print_decorator
2 ... def addition(a, b):
3 ...     return a + b
4 ...
5 >>> addition(1, 2)
6 Addition des nombres 1 et 2
7 Retour: 3
8 3
```

Mais notre décorateur est ici très spécialisé, il ne fonctionne qu'avec les fonctions prenant deux paramètres, et affichera « Addition » dans tous les cas. Nous pouvons le modifier pour le rendre plus générique (souvenez vous d'`*args` et `**kwargs`).

```
1 def print_decorator(function):
2     def new_function(*args, **kwargs):
```

III. Ainsi font fonctions

```
3         print('Appel de la fonction {} avec args={} et kwargs={}'.format(
4             function.__name__, args, kwargs))
5     ret = function(*args, **kwargs)
6     print('Retour: {}'.format(ret))
7     return ret
8     return new_function
```

Je vous laisse l'essayer sur des fonctions différentes pour vous rendre compte de sa généralité.

Les définitions de fonctions ne sont pas limitées à un seul décorateur : il est possible d'en spécifier autant que vous le souhaitez, en les plaçant les uns à la suite des autres.

```
1 @decorator
2 @print_decorator
3 def useless():
4     pass
```

L'ordre dans lequel ils sont spécifiés est important, le code précédent équivaut à :

```
1 def useless():
2     pass
3 useless = decorator(print_decorator(useless))
```

On voit donc que les décorateurs spécifiés en premiers sont ceux qui seront appliqués en derniers.

J'ai dit plus haut que les décorateurs s'appliquaient aux fonctions. C'est aussi valable pour les fonctions définies à l'intérieur de classes (les méthodes, donc). Mais sachez enfin que les décorateurs s'étendent aux déclarations de classes.

```
1 @print_decorator
2 class MyClass:
3     @decorator
4     def method(self):
5         pass
```

6.2. Décorateurs paramétrés

Nous avons vu comment appliquer un décorateur à une fonction, nous pourrions cependant vouloir paramétrer le comportement de ce décorateur. Dans notre exemple précédent (`print_decora`

III. Ainsi font fonctions

`tor`), nous affichons du texte avant et après l'appel de fonction. Mais que faire si nous souhaitons modifier ce texte (pour en changer la langue, utiliser un autre terme que « fonction ») ?

Nous ne voulons pas avoir à créer un décorateur différent pour chaque phrase possible et imaginable. Nous aimerions pouvoir passer nos chaînes de caractères à notre décorateur pour qu'il s'occupe de les afficher au moment opportun.

En fait, `@` ne doit pas nécessairement être suivi d'un nom d'objet, des arguments peuvent aussi s'y ajouter à l'aide de parenthèses (comme on le ferait pour un appel). Mais le comportement peut vous sembler étrange au premier abord.

Par exemple, pour utiliser un tel décorateur paramétré :

```
1 @param_print_decorator('call {} with args({}) and kwargs({})',
2   'ret={}')
3 def test_func(x):
4     return x
```

Il nous faudra posséder un `callable param_print_decorator` qui, quand il sera appelé, retournera un décorateur qui pourra ensuite être appliqué à notre fonction. Un décorateur paramétré n'est ainsi qu'un `callable` retournant un décorateur simple.

Le code de `param_print_decorator` ressemblerait ainsi à :

```
1 def param_print_decorator(before, after): # Décorateur paramétré
2     def decorator(function): # Décorateur
3         def new_function(*args, **kwargs): # Fonction qui
4             remplacera notre fonction décorée
5             print(before.format(function.__name__, args, kwargs))
6             ret = function(*args, **kwargs)
7             print(after.format(ret))
8             return ret
9         return new_function
10    return decorator
```

6.3. Envelopper une fonction

Une fonction n'est pas seulement un bout de code avec des paramètres. C'est aussi un nom (des noms, avec ceux des paramètres), une documentation (*docstring*), des annotations, etc. Quand nous décorons une fonction à l'heure actuelle (dans les cas où nous en retournons une nouvelle), nous perdons toutes ces informations annexes.

Un exemple pour nous en rendre compte :

III. Ainsi font fonctions

```
1 >>> def decorator(f):
2 ...     def decorated(*args, **kwargs):
3 ...         return f(*args, **kwargs)
4 ...     return decorated
5 ...
6 >>> @decorator
7 ... def addition(a: int, b: int) -> int:
8 ...
9     "Cette fonction réalise l'addition des paramètres `a` et `b`"
10 ...     return a + b
11 >>> help(addition)
12 Help on function decorated in module __main__:
13
14 decorated(*args, **kwargs)
```

Alors, que voit-on ? Pas grand chose. Le nom qui apparaît est celui de `decorated`, les paramètres sont `*args` et `**kwargs` (sans annotations), et nous avons aussi perdu notre *docstring*. Autant dire qu'il ne reste rien pour comprendre ce que fait la fonction.

6.3.1. Envelopper des fonctions

Plus tôt dans ce cours, je vous parlais du module [functools](#). Il ne nous a pas encore révélé tous ses mystères.

Nous allons ici nous intéresser aux fonctions `update_wrapper` et `wraps`. Ces fonctions vont nous permettre de copier les informations d'une fonction vers une nouvelle.

`update_wrapper` prend en premier paramètre la fonction à laquelle ajouter les informations et celle dans laquelle les puiser en second. Pour reprendre notre exemple précédent, il nous faudrait faire :

```
1 import functools
2
3 def decorator(f):
4     def decorated(*args, **kwargs):
5         return f(*args, **kwargs)
6     functools.update_wrappers(decorated, f) # Nous copions les
7     informations de `f` dans `decorated`
8     return decorated
```

Mais une autre fonction nous sera bien plus utile car plus concise, et recommandée par la documentation Python pour ce cas. Il s'agit de `wraps`, qui retourne un décorateur lorsqu'appelé avec une fonction.

III. Ainsi font fonctions

La fonction décorée par `wraps` prendra les informations de la fonction passée à l'appel de `wraps`. Ainsi, nous n'aurons qu'à précéder toutes nos fonctions décorées par `@functools.wraps(fonction_a_decorer)`. Dans notre exemple :

```
1 import functools
2
3 def decorator(f):
4     @functools.wraps(f)
5     def decorated(*args, **kwargs):
6         return f(*args, **kwargs)
7     return decorated
```

Vous pouvez maintenant redéfinir la fonction `addition`, et tester à nouveau l'appel à `help` pour constater les différences.

6.4. TP : Arguments positionnels

Nous avons vu avec les signatures qu'il existait en Python des paramètres *positional-only*, c'est-à-dire qui ne peuvent recevoir que des arguments positionnels.

Mais il n'existe à ce jour aucune syntaxe pour écrire en Python une fonction avec des paramètres *positional-only*. Il est seulement possible, comme nous l'avons fait au TP précédent, de récupérer les arguments positionnels avec `*args` et d'en extraire les valeurs qui nous intéressent.

Nous allons alors développer un décorateur pour pallier à ce manque. Ce décorateur modifiera la signature de la fonction reçue pour transformer en *positional-only* ses `n` premiers paramètres. `n` sera un paramètre du décorateur.

Python nous permet de redéfinir la signature d'une fonction, en assignant la nouvelle signature à son attribut `__signature__`. Mais cette redéfinition n'est que cosmétique (elle apparaît par exemple dans l'aide de la fonction). Ici, nous voulons que la modification ait un effet.

Nous créerons donc une fonction *wrapper*, qui se chargera de vérifier la conformité des arguments avec la nouvelle signature.

Nous allons diviser le travail en deux parties :

- Dans un premier temps, nous réaliserons une fonction pour réécrire une signature ;
- Puis dans un second temps, nous écrirons le code du décorateur.

6.4.1. Réécriture de la signature

La première fonction, que nous appellerons `signature_set_positional`, recevra en paramètres une signature et un nombre `n` de paramètres à passer en *positional-only*. La fonction retournera une signature réécrite.

Nous utiliserons donc les méthodes `replace` de la signature et des paramètres, pour changer le positionnement des paramètres ciblés, et mettre à jour la liste des paramètres de la signature.

III. Ainsi font fonctions

La fonction itérera sur les `n` premiers paramètres, pour les convertir en *positional-only*.

On distinguera trois cas :

- Le paramètre est déjà *positional-only*, il n'y a alors rien à faire ;
- Le paramètre est *positional-or-keyword*, il peut être transformé ;
- Le paramètre est d'un autre type, il ne peut pas être transformé, on lèvera alors une erreur.

Puis une nouvelle signature sera créée et retournée avec cette nouvelle liste de paramètres.

```
1 def signature_set_positional(sig, n):
2     params = list(sig.parameters.values()) # Liste des paramètres
3     if len(params) < n:
4         raise
5         TypeError('Signature does not have enough parameters')
6     for i, param in zip(range(n), params): # Itère sur les n
7         premiers paramètres
8         if param.kind == param.POSITIONAL_ONLY:
9             continue
10        elif param.kind == param.POSITIONAL_OR_KEYWORD:
11            params[i] = param.replace(kind=param.POSITIONAL_ONLY)
12        else:
13            raise
14            TypeError('{} parameter cannot be converted to POSITIONAL_ONLY')
15    return sig.replace(parameters=params)
```

6.4.2. Décorateur paramétré

Passons maintenant à `positional_only`, notre décorateur paramétré. Pour rappel, un décorateur paramétré est une fonction qui retourne un décorateur. Et un décorateur est une fonction qui reçoit une fonction et retourne une fonction.

Le décorateur proprement dit se chargera de calculer la nouvelle signature et de l'appliquer à la fonction décorée. Il créera aussi un *wrapper* à la fonction, lequel se chargera de vérifier la correspondance des arguments avec la signature.

Nous n'oublierons pas d'appliquer `functools.wraps` à notre *wrapper* pour récupérer les informations de la fonction initiale.

```
1 import functools
2 import inspect
3
4 def positional_only(n):
5     def decorator(f):
6         sig = signature_set_positional(inspect.signature(f), n)
7         @functools.wraps(f)
```

III. Ainsi font fonctions

```
8     def decorated(*args, **kwargs):
9         bound = sig.bind(*args, **kwargs)
10        return f(*bound.args, **bound.kwargs)
11    decorated.__signature__ = sig
12    return decorated
13    return decorator
```

Voyons maintenant l'utilisation.

```
1 >>> @positional_only(2)
2 ... def addition(a, b):
3 ...     return a + b
4 ...
5 >>> print(inspect.signature(addition))
6 (a, b, /)
7 >>> addition(3, 5)
8 8
9 >>> addition(3, b=5)
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12 TypeError: 'b' parameter is positional only, but was passed as a
    keyword
```

```
1 >>> @positional_only(1)
2 ... def addition(a, b):
3 ...     return a + b
4 ...
5 >>> addition(3, 5)
6 8
7 >>> addition(3, b=5)
8 8
9 >>> addition(a=3, b=5)
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12 TypeError: 'a' parameter is positional only, but was passed as a
    keyword
```

Les ressources sur les décorateurs sont peu nombreuses dans la documentation, car il ne s'agit au final que d'un sucre syntaxique. Je vous indique tout de même les deux principales pages qui les évoquent.

— Définition du terme décorateur : <https://docs.python.org/3/glossary.html#term-decorator> ↗

III. Ainsi font fonctions

- Syntaxe de déclaration d'une fonction : https://docs.python.org/3/reference/compound_stmts.html#function ↗

Quatrième partie
Plus loin, un peu plus loin

6.5. Au-delà des mers

Continuons notre voyage, et voguons vers de nouveaux horizons. Aventurons-nous maintenant au sein d'objets plus complexes du langage.

7. Générateurs

Nous étudierons dans ce chapitre les générateurs, un nouveau type d'itérables.

7.1. Dessine-moi un générateur

Les générateurs sont donc des itérables, mais aussi des itérateurs, ce qui implique qu'ils se consomment quand on les parcourt (et que nous ne pouvons donc les parcourir qu'une seule fois).

Ils sont généralement créés par des fonctions construites à l'aide du mot clef `yield`. Par abus de langage ces fonctions génératrices sont parfois elles-mêmes appelées générateurs.

7.1.1. Le mot-clef `yield`

Un générateur est donc construit à partir d'une fonction. Pour être génératrice, une fonction doit contenir un ou plusieurs `yield`.

Lors de l'appel, la fonction retournera un générateur, et à chaque appel à la fonction *builtin* `next` sur ce générateur, le code jusqu'au prochain `yield` sera exécuté. Comme pour tout itérateur, la fonction `next` appelle la méthode spéciale `__next__` du générateur.

`yield` peut-être ou non suivi d'une expression. La valeur retournée par `next` sera celle apposée au `yield`, ou `None` dans le cas où aucune valeur n'est spécifiée.

Un exemple pour mieux comprendre cela :

```
1 >>> def function():
2 ...     yield 4
3 ...     yield
4 ...     yield 'hej'
5 ...
6 >>> gen = function()
7 >>> next(gen)
8 4
9 >>> next(gen)
10 None
11 >>> next(gen)
12 'hej'
13 >>> next(gen)
14 Traceback (most recent call last):
```


IV. Plus loin, un peu plus loin

```
15 File "<stdin>", line 1, in <module>
16 StopIteration
17 >>> list(gen) # Tout le générateur a été parcouru
18 []
```

Ou avec un `for` :

```
1 >>> for i in function():
2 ...     print(i)
3 ...
4 4
5 None
6 hej
```

Il est aussi possible pour une fonction génératrice d'utiliser `return`, qui aura pour effet de le stopper (`StopIteration`). La valeur passée au `return` sera contenue dans l'exception levée.

```
1 >>> def function():
2 ...     yield 4
3 ...     yield
4 ...     return 2
5 ...     yield 'hej'
6 ...
7 >>> gen = function()
8 >>> next(gen)
9 4
10 >>> next(gen)
11 None
12 >>> next(gen)
13 Traceback (most recent call last):
14   File "<stdin>", line 1, in <module>
15 StopIteration: 2
```

Le générateur en lui-même ne retourne rien (il n'est pas *callable*), il produit des valeurs à l'aide de `yield`. Pour bien faire la différence entre notre générateur et sa fonction génératrice, on peut regarder ce qu'en dit Python.

```
1 >>> gen
2 <generator object function at 0x7f008dfb0570>
3 >>> function
4 <function function at 0x7f008dce2ea0>
```

Bien sûr, notre générateur est très simpliste dans l'exemple, mais toutes les structures de contrôle du Python peuvent y être utilisées. De plus, le générateur peut aussi être paramétré

IV. Plus loin, un peu plus loin

via les arguments passés à la fonction. Voici un exemple un peu plus poussé avec un générateur produisant les n premiers termes d'une [suite de Fibonacci](#) débutant par a et b .

```
1 >>> def fibonacci(n, a=0, b=1):
2 ...     for _ in range(n):
3 ...         yield a
4 ...         a, b = b, a + b
5 ...
6 >>> list(fibonacci(10))
7 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
8 >>> list(fibonacci(5, 6, 7))
9 [6, 7, 13, 20, 33]
```

7.2. Altérer un générateur avec send

Maintenant que nous savons créer des générateurs, nous allons voir qu'il est possible de communiquer avec eux après leur création.

Pour cela, les générateurs sont dotés d'une méthode `send`. Le paramètre reçu par cette méthode sera transmis au générateur. Mais comment le reçoit-il ?

Au moment où il arrive sur une instruction `yield`, le générateur se met en pause. Lors de l'itération suivante, l'exécution reprend au niveau de ce même `yield`. Quand ensuite vous appelez la méthode `send` du générateur, en lui précisant un argument, l'exécution reprend ; et `yield` retourne la valeur passée à `send`.

Attention donc, un appel à `send` produit une itération supplémentaire dans le générateur. `send` retourne alors la valeur de la prochaine itération comme le ferait `next`.

```
1 >>> gen = fibonacci(10)
2 >>> next(gen)
3 0
4 >>> next(gen)
5 1
6 >>> gen.send('test') # Le send consomme une itération
7 1
8 >>> next(gen)
9 2
```

Comme je le disais, une valeur peut-être retournée par l'instruction `yield`, c'est-à-dire dans le corps même de la fonction génératrice. Modifions quelque peu notre générateur `fibonacci` pour nous en apercevoir.

IV. Plus loin, un peu plus loin

```
1 >>> def fibonacci(n, a=0, b=1):
2     ...     for _ in range(n):
3     ...         ret = yield a
4     ...         print('retour:', ret)
5     ...         a, b = b, a + b
6     ...
7 >>> gen = fibonacci(10)
8 >>> next(gen)
9 0
10 >>> next(gen)
11 retour: None
12 1
13 >>> gen.send('test')
14 retour: test
15 1
16 >>> next(gen)
17 retour: None
18 2
```

Nous pouvons voir que lors de notre premier `next`, aucun retour n'est imprimé : c'est normal, le générateur n'étant encore jamais passé dans un `yield`, nous ne sommes pas encore arrivés jusqu'au premier appel à `print` (qui se trouve après le premier `yield`).

Cela signifie aussi qu'il est impossible d'utiliser `send` avant le premier `yield` (puisque'il n'existe aucun précédent `yield` qui retournerait la valeur).

```
1 >>> gen = fibonacci(10)
2 >>> gen.send('test')
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   TypeError: can't send non-None value to a just-started generator
```

Pour comprendre un peu mieux l'intérêt de `send`, nous allons implémenter une file (*queue*) par l'intermédiaire d'un générateur. Celle-ci sera construite à l'aide des arguments donnés à l'appel, retournera le premier élément à chaque itération (en le retirant de la file). On pourra aussi ajouter de nouveaux éléments à cette *queue* via `send`.

```
1 def queue(*args):
2     elems = list(args)
3     while elems:
4         new = yield elems.pop(0)
5         if new is not None:
6             elems.append(new)
```

Testons un peu pour voir.

IV. Plus loin, un peu plus loin

```
1 >>> q = queue('a', 'b', 'c')
2 >>> next(q)
3 'a'
4 >>> q.send('d')
5 'b'
6 >>> next(q)
7 'c'
8 >>> next(q)
9 'd'
```

Et si nous souhaitons itérer sur notre file :

```
1 >>> q = queue('a', 'b', 'c')
2 >>> for letter in q:
3 ...     if letter == 'a':
4 ...         q.send('d')
5 ...         print(letter)
6 ...
7 'b'
8 a
9 c
10 d
```

Que se passe-t-il ? En fait, `send` consommant une itération, le `b` n'est pas obtenu via le `for`, mais en retour de `send`, et directement imprimé sur la sortie de l'interpréteur (avant même le `print` de `a` puisque le `send` est fait avant).

Nous pouvons assigner le retour de `q.send` afin d'éviter que l'interpréteur n'en imprime le résultat, mais cela ne changerait pas tellement le problème : nous ne tomberons jamais sur `'b'` dans nos itérations du `for`.

Pour obtenir le comportement attendu, nous pourrions avancer dans les itérations uniquement si le dernier `yield` a renvoyé `None` (un `yield` renvoyant `None` étant considéré comme un `next`). Comment faire cela ? Par une boucle qui exécute `yield` jusqu'à ce qu'il retourne `None`. Ce `yield` n'aura pas de paramètre spécifique, cette valeur étant celle retournée ensuite par `send`, elle ne nous intéresse pas ici.

Ainsi, les `send` ne consommeront qu'une itération de la sous-boucle, tandis que les « vraies » itérations seront réservées aux `next`.

```
1 def queue(*args):
2     elems = list(args)
3     while elems:
4         new = yield elems.pop(0)
5         while new is not None:
6             elems.append(new)
```

```
7 new = yield
```

```
1 >>> q = queue('a', 'b', 'c')
2 >>> for letter in q:
3 ...     if letter == 'a':
4 ...         q.send('d')
5 ...         print(letter)
6 ...
7 a
8 b
9 c
10 d
```

7.3. Méthodes des générateurs

Nous avons vu que les générateurs possédaient des méthodes `__next__` et `send`. Nous allons maintenant nous intéresser aux deux autres méthodes de ces objets : `throw` et `close`.

7.3.1. `throw`

La méthode `throw` permet de lever une exception depuis le générateur, à l'endroit où ce dernier s'est arrêté. Elle a pour effet de réveiller le générateur pour lui faire lever une exception du type indiqué.

Il s'agit alors d'une autre manière d'influer sur l'exécution d'un générateur, par des événements qui lui sont extérieurs.

`throw` possède 3 paramètres dont deux facultatifs :

- Le premier, `type`, est le type d'exception à lever ;
- Le second, `value`, est la valeur à passer en instanciant cette exception ;
- Et le troisième, `traceback`, permet de passer un pile d'appel (*traceback object*) particulière à l'exception.

L'exception survient donc au niveau du `yield`, et peut tout à fait être attrapée par le générateur. Si c'est le cas, `throw` retournera la prochaine valeur produite par le générateur, ou lèvera une exception `StopIteration` indiquant que le générateur a été entièrement parcouru, à la manière de `next`.

```
1 >>> def generator_function():
2 ...     for i in range(5):
3 ...         try:
4 ...             yield i
```

IV. Plus loin, un peu plus loin

```
5 ...         except ValueError:
6 ...             print('Something goes wrong')
7 ...
8 >>> gen = generator_function()
9 >>> next(gen)
10 0
11 >>> next(gen)
12 1
13 >>> gen.throw(ValueError)
14 Something goes wrong
15 2
16 >>> next(gen)
17 3
18 >>> next(gen)
19 4
20 >>> gen.throw(ValueError)
21 Something goes wrong
22 Traceback (most recent call last):
23   File "<stdin>", line 1, in <module>
24 StopIteration
```

Si l'exception n'est pas attrapée par le générateur, elle provoque alors sa fermeture, et remonte logiquement jusqu'à l'objet ayant fait appel à lui.

```
1 >>> def generator_function():
2 ...     for i in range(5):
3 ...         yield i
4 ...
5 >>> gen = generator_function()
6 >>> next(gen)
7 0
8 >>> next(gen)
9 1
10 >>> gen.throw(ValueError)
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13   File "<stdin>", line 3, in generator_function
14 ValueError
15 >>> next(gen)
16 Traceback (most recent call last):
17   File "<stdin>", line 1, in <module>
18 StopIteration
```

7.3.2. close

Un cas particulier d'appel à la méthode `throw` est de demander au générateur de s'arrêter en lui faisant lever une exception `GeneratorExit`.

À la réception de cette dernière, il est attendu que le générateur se termine (`StopIteration`) ou lève à son tour une `GeneratorExit` (par exemple en n'attrapant pas l'exception survenue).

La méthode `close` du générateur permet de réaliser cet appel et d'attraper les `StopIteration/GeneratorExit` en retour.

```
1 >>> def generator_function():
2 ...     for i in range(5):
3 ...         yield i
4 ...
5 >>> gen = generator_function()
6 >>> next(gen)
7 0
8 >>> next(gen)
9 1
10 >>> gen.close()
11 >>> next(gen)
12 Traceback (most recent call last):
13   File "<stdin>", line 1, in <module>
14 StopIteration
```

```
1 >>> def generator_function():
2 ...     for i in range(5):
3 ...         try:
4 ...             yield i
5 ...         except GeneratorExit:
6 ...             break
7 ...
8 >>> gen = generator_function()
9 >>> next(gen)
10 0
11 >>> next(gen)
12 1
13 >>> gen.close()
14 >>> next(gen)
15 Traceback (most recent call last):
16   File "<stdin>", line 1, in <module>
17 StopIteration
```

Puisqu'elle attrape les `StopIteration`, il est possible d'appeler plusieurs fois la méthode `close` sur un même générateur.

IV. Plus loin, un peu plus loin

```
1 >>> gen.close()
2 >>> gen.close()
```

`close` s'occupe aussi de lever une `RuntimeError` dans le cas où le générateur ne s'arrêterait pas et continuerait à produire des valeurs.

```
1 >>> def generator_function():
2 ...     for i in range(5):
3 ...         try:
4 ...             yield i
5 ...         except GeneratorExit:
6 ...             print('Ignoring')
7 ...
8 >>> gen = generator_function()
9 >>> next(gen)
10 0
11 >>> next(gen)
12 1
13 >>> gen.close()
14 Ignoring
15 Traceback (most recent call last):
16   File "<stdin>", line 1, in <module>
17 RuntimeError: generator ignored GeneratorExit
18 >>> next(gen)
19 3
```

7.4. Déléguer à un autre générateur avec `yield from`

Nous savons produire les valeurs d'un générateur à l'aide du mot clef `yield`. Voyons maintenant quelque chose d'un peu plus complexe avec `yield from`. Ce nouveau mot clef permet de déléguer l'itération à un sous-générateur pris en paramètre. La rencontre du `yield from` provoque une interruption du générateur courant, le temps d'itérer et produire les valeurs du générateur délégué.

```
1 def big_queue():
2     yield 0
3     yield from queue(1, 2, 3)
4     yield 4
```

Celui-ci agit comme si nous itérions sur `queue(1, 2, 3)` depuis `big_queue`, tout en *yieldant* toutes ses valeurs.

IV. Plus loin, un peu plus loin

```
1 def big_queue():
2     yield 0
3     for value in queue(1, 2, 3):
4         yield value
5     yield 4
```

À la différence près qu'avec `yield from`, les paramètres passés lors d'un `send` sont aussi relégués aux sous-générateurs. Tout comme sont relayés aux sous-générateurs les appels aux méthodes `throw` et `close`.

Dans notre première version, nous pouvons nous permettre ceci :

```
1 >>> q = big_queue()
2 >>> next(q)
3 0
4 >>> next(q)
5 1
6 >>> q.send('foo')
7 >>> next(q)
8 2
9 >>> next(q)
10 3
11 >>> next(q)
12 'foo'
13 >>> next(q)
14 4
15 >>> next(q)
16 Traceback (most recent call last):
17   File "<stdin>", line 1, in <module>
18 StopIteration
```

Où l'on voit bien que le `send` est pris en compte par la sous-*queue*, et la valeur ajoutée à la file. Je vous invite à essayer avec la seconde implémentation de `big_queue` (celle sans `yield from`), pour bien observer l'effet de la délégation du `send`.

On peut aussi noter que `yield from` n'attend pas nécessairement un générateur en paramètre, mais n'importe quel type d'itérable.

```
1 def gen_from_iterables():
2     yield from [1, 2, 3]
3     yield from 'abcdef'
4     yield from {'x': 1, 'y': -1}
```

Et par exemple, si nous voulions réécrire la fonction `chain` du module `itertools`, nous pourrions procéder ainsi :

```
1 def chain(*iterables):
2     for iterable in iterables:
3         yield from iterable
```

7.5. Listes et générateurs en intension

Intéressons-nous maintenant aux sucres syntaxiques que sont les listes et générateurs en intension.

7.5.1. Listes en intension

Vous connaissez probablement déjà les listes en intension (*comprehension lists*), mais je vais me permettre un petit rappel.

Les listes en intension sont une syntaxe courte pour définir des listes à partir d'un itérable et d'une expression à appliquer sur chaque élément (un peu à la manière de `map`, qui lui ne permet que d'appeler un *callable* pour chaque l'élément).

Une expression étant en Python tout ce qui possède une valeur (même `None`), c'est-à-dire ce qui n'est pas une instruction (`if`, `while`, etc.). Il faut noter que les ternaires sont des expressions (`a if predicat() else b`), et peuvent donc y être utilisés.

Ainsi,

```
1 >>> squares = [x**2 for x in range(10)]
2 >>> squares
3 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

sera l'équivalent de

```
1 squares = []
2 for x in range(10):
3     squares.append(x**2)
```

ou encore de

```
1 squares = list(map(lambda x: x**2, range(10)))
```

Le gain en lisibilité est net.

IV. Plus loin, un peu plus loin

Cette syntaxe permet aussi d'appliquer des filtres, à la manière de `filter`. Par exemple si nous ne voulions que les carrés supérieurs à 10 :

```
1 >>> [x**2 for x in range(10) if x**2 >= 10]
2 [16, 25, 36, 49, 64, 81]
```

Une liste en intension peut directement être passée en paramètre à une fonction.

```
1 >>> sum([x**2 for x in range(10)])
2 285
```

Étant des expressions, elles peuvent être imbriquées dans d'autres listes en intension :

```
1 >>> [[x + y for x in range(5)] for y in range(3)]
2 [[0, 1, 2, 3, 4], [1, 2, 3, 4, 5], [2, 3, 4, 5, 6]]
```

Enfin, il est possible de parcourir plusieurs niveaux de listes dans une seule liste en intension :

```
1 >>> matrix = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
2 >>> [elem + 1 for line in matrix for elem in line]
3 [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Vous noterez que les `for` se placent dans l'ordre des dimensions que nous voulons explorer : d'abord les lignes, puis les éléments qu'elles contiennent.

7.5.2. Générateurs en intension

De la même manière que pour les listes, nous pouvons définir des générateurs en intension (*generator expressions*). La syntaxe est très similaire, il suffit de remplacer les crochets par des parenthèses pour passer d'une liste à un générateur.

```
1 >>> squares = (x**2 for x in range(10))
2 >>> squares
3 <generator object <genexpr> at 0x7f8b8a9a7090>
```

Et nous pouvons les utiliser tels que les autres itérables.

IV. Plus loin, un peu plus loin

```
1 >>> list(squares)
2 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
3 >>> list(squares) # En gardant à l'esprit qu'ils se consomment
4 []
```

Et pour finir, il est possible de simplifier encore la syntaxe quand un générateur en intension est seul paramètre d'une fonction, en supprimant les parenthèses redondantes.

```
1 >>> sum((x**2 for x in range(10)))
2 285
3 >>> sum(x**2 for x in range(10)) # Une paire de parenthèses est
  retirée
4 285
```

7.6. Liste ou générateur ?

Une question qui revient souvent est celle de savoir quand choisir une liste et quand choisir un générateur, voici donc un petit comparatif :

- Les listes prennent généralement plus de place en mémoire : tous les éléments existent en même temps. Dans le cas d'un générateur, l'élément n'existe que quand l'itérateur l'atteint, et n'existe plus après (sauf s'il est référencé autre part) ;
- Les générateurs peuvent être infinis (contrairement aux listes qui occupent un espace qui se doit d'être fini).

```
1 >>> def infinite():
2 ...     n = 0
3 ...     while True:
4 ...         yield n
5 ...         n += 1
```

- Les générateurs ne sont pas indexables : on ne peut accéder à un élément particulier (il faut itérer jusque cet élément) ;
- Les générateurs ont une durée de vie plus courte (ils ne contiennent plus rien une fois qu'ils ont été itérés en entier) ;
- Du fait que les générateurs n'occupent que peu de place en mémoire, on peut les enchaîner sans crainte.

```
1 numbers = (x**2 for x in range(100))
2 numbers = zip(infinite(), numbers)
```

IV. Plus loin, un peu plus loin

```
3 numbers = (a + b for (a, b) in numbers)
```

Ainsi, les éléments du premier générateur ne seront calculés qu'au parcours de `numbers`. Il est aussi possible de profiter des avantages de l'un et de l'autre en récupérant une liste en fin de chaîne, par exemple en remplaçant la dernière ligne par :

```
1 numbers = [a + b for (a, b) in numbers]
```

Ou en ajoutant à la fin :

```
1 numbers = list(numbers)
```

7.7. TP : map

Dans ce TP, nous allons réaliser un équivalent de la fonction `map`, que nous appellerons `my_map`.

Pour rappel, `map` permet d'appliquer une fonction sur tous les éléments d'un itérable. Elle reçoit en paramètres la fonction à appliquer et l'itérable, et retourne un nouvel itérable correspondant à l'application de la fonction sur chacune des valeurs de l'itérable d'entrée.

Un générateur se prête donc très bien à cet exercice : nous ferons de `my_map` une fonction génératrice.

Elle se contentera dans un premier temps d'itérer sur l'entrée, et de *yielder* le résultat obtenu pour chaque élément.

```
1 def my_map(f, iterable):
2     for item in iterable:
3         yield f(item)
```

Cette implémentation répond très bien à la problématique initiale.

```
1 >>> for x in my_map(lambda x: x*2, range(10)):
2     ...     print(x)
3     ...
4     0
5     2
6     4
7     6
8     8
9     10
```

IV. Plus loin, un peu plus loin

```
10 12
11 14
12 16
13 18
```

Seulement, pour aller un peu plus loin et mettre en pratique ce que nous avons vu dans le chapitre, nous allons faire en sorte de pouvoir changer la fonction `f` en cours de route.

Pour ce faire, nous communiquerons avec notre générateur à l'aide de sa méthode `send`. Nous voulons donc que chaque fois que `yield` retourne autre chose que `None`, cette valeur soit utilisée comme nouvelle fonction.

Une approche simpliste serait la suivante :

```
1 def my_map(f, iterable):
2     for item in iterable:
3         ret = yield f(item)
4         if ret is not None:
5             f = ret
```

Mais comme `send` provoque une itération supplémentaire du générateur, elle a l'inconvénient de faire perdre des valeurs. (les valeurs ne sont pas vraiment perdues, mais sont les valeurs de retour de `send`)

```
1 >>> gen = my_map(lambda x: x+1, range(10))
2 >>> for x in gen:
3     ...     print('Got', x)
4     ...     if x == 5:
5     ...         gen.send(lambda x: x+2)
6     ...
7 Got 1
8 Got 2
9 Got 3
10 Got 4
11 Got 5
12 7
13 Got 8
14 Got 9
15 Got 10
16 Got 11
```

L'affichage du 7 étant dû à l'interpréteur interactif qui affiche la valeur de retour du `send`.

Pour palier à ce petit problème, nous pouvons dans notre générateur `yielder None` quand une fonction a été reçue. Ainsi, le retour du `send` sera le `None` transmis par `yield`, et la valeur ne sera pas perdue.

IV. Plus loin, un peu plus loin

Nous réécrivons alors notre générateur en conséquence.

```
1 def my_map(f, iterable):
2     for item in iterable:
3         ret = yield f(item)
4         if ret is not None:
5             f = ret
6         yield None
```

Et observons la correction.

```
1 >>> gen = my_map(lambda x: x+1, range(10))
2 >>> for x in gen:
3     ...     print('Got', x)
4     ...     if x == 5:
5     ...         gen.send(lambda x: x+2)
6     ...
7 Got 1
8 Got 2
9 Got 3
10 Got 4
11 Got 5
12 Got 7
13 Got 8
14 Got 9
15 Got 10
16 Got 11
```

Mais notre générateur reste sujet à un problème plus subtil : des pertes se produisent si nous appelons `yield` plusieurs fois d'affilée.

```
1 >>> gen = my_map(lambda x: x+1, range(10))
2 >>> for x in gen:
3     ...     print('Got', x)
4     ...     if x == 5:
5     ...         gen.send(lambda x: x+2)
6     ...         gen.send(lambda x: x*2)
7     ...
8 Got 1
9 Got 2
10 Got 3
11 Got 4
12 Got 5
13 7
14 Got 8
```

IV. Plus loin, un peu plus loin

```
15 Got 9
16 Got 10
17 Got 11
```

En effet, lors du premier `send`, nous retons dans la condition du générateur pour arriver sur le `yield None`, valeur retournée par `send`. Mais nous ne nous inquiétons pas de savoir ce que retourne ce second `yield` (en l'occurrence, la fonction envoyée par le second `send`, qui n'est jamais prise en compte).

Vous l'aurez compris, il nous suffira donc de mettre en place une boucle autour du `yield None` et n'en sortir qu'une fois qu'il aura retourné `None`. Chaque valeur différente sera enregistrée comme nouvelle fonction `f`.

```
1 def my_map(f, iterable):
2     for value in iterable:
3         newf = yield f(value)
4         while newf is not None:
5             f = newf
6             newf = yield None
```

Cette fois-ci, plus de problèmes !

```
1 >>> gen = my_map(lambda x: x+1, range(10))
2 >>> for x in gen:
3     ...     print('Got', x)
4     ...     if x == 5:
5     ...         gen.send(lambda x: x+2)
6     ...         gen.send(lambda x: x*2)
7     ...
8 Got 1
9 Got 2
10 Got 3
11 Got 4
12 Got 5
13 Got 10
14 Got 12
15 Got 14
16 Got 16
17 Got 18
```

Quelques pages habituelles tirées de la documentation :

— Définition du terme générateur : <https://docs.python.org/3/glossary.html#term-generator> ↗

IV. Plus loin, un peu plus loin

— Expressions `yield` : <https://docs.python.org/3/reference/expressions.html#yieldexpr> ↗

Je profite aussi de la fin de ce chapitre pour vous conseiller cet article de [nohar](#) ↗ sur les coroutines, une utilisation possible et courante des générateurs : <https://zestedesavoir.com/articles/152/la-puissance-cachee-des-coroutines> ↗

J'aurais aimé abordé le sujet des coroutines et de la programmation asynchrone en Python, mais un cours entier sur le sujet n'est pas de trop !

8. Gestionnaires de contexte

Avant de parler de cette spécificité du langage, je voudrais expliciter la notion de contexte. Un contexte est une portion de code cohérente, avec des garanties en entrée et en sortie.

Par exemple, pour la lecture d'un fichier, on garantit que celui-ci soit ouvert et accessible en écriture en entrée (donc à l'intérieur du contexte), et l'on garantit sa fermeture en sortie (à l'extérieur).

De multiples utilisations peuvent être faites des contextes, comme l'allocation et la libération de ressources (fichiers, verrous, etc.), ou encore des modifications temporaires sur l'environnement courant (répertoire de travail, redirection d'entrées/sorties).

Python met à notre disposition des gestionnaires de contexte, c'est-à-dire une structure de contrôle pour les mettre en place, à l'aide du mot-clef `with`.

8.1. `with` or without you

Un contexte est ainsi un scope particulier, avec des opérations exécutées en entrée et en sortie.

Un bloc d'instructions `with` se présente comme suit.

```
1 with expr as x: # avec expr étant un gestionnaire de contexte
2     ... # opérations sur x
```

La syntaxe est assez simple à appréhender, `x` permettra ici de contenir des données propres au contexte (`x` vaudra `expr` dans la plupart des cas). Si par exemple `expr` correspondait à une ressource, la libération de cette ressource (fermeture du fichier, déblocage du verrou, etc.) serait gérée pour nous en sortie du scope, dans tous les cas.

Il est aussi possible de gérer plusieurs contextes dans un même bloc :

```
1 with expr1 as x, expr2 as y:
2     ... # traitements sur x et y
```

équivalent à

```
1 with expr1 as x:
2     with expr2 as y:
3         ... # traitements sur x et y
```

8.2. La fonction open

L'un des gestionnaires de contexte les plus connus est probablement le fichier, tel que retourné par la fonction `open`. Jusque là, vous avez pu l'utiliser de la manière suivante :

```
1 f = open('filename', 'w')
2 ##### traitement sur le fichier
3 ...
4 f.close()
```

Mais sachez que ça n'est pas la meilleure façon de procéder. En effet, si une exception survient pendant le traitement, la méthode `close` ne sera par exemple jamais appelée, et les dernières données écrites pourraient être perdues.

Il est donc conseillé de plutôt procéder de la sorte, avec `with` :

```
1 with open('filename', 'w') as f:
2     # traitement sur le fichier
3     ...
```

Ici, la fermeture du fichier est implicite, nous verrons plus loin comment cela fonctionne en interne.

Nous pourrions reproduire un comportement similaire sans gestionnaire de contexte, mais le code serait un peu plus complexe.

```
1 try:
2     f = open('filename', 'w')
3     # traitement sur le fichier
4     ...
5 finally:
6     f.close()
```

8.3. Fonctionnement interne

Ça, c'est pour le cas d'utilisation, nous étudierons ici le fonctionnement interne.

Les gestionnaires de contexte sont en fait des objets disposant de deux méthodes spéciales : `__enter__` et `__exit__`, qui seront respectivement appelées à l'entrée et à la sortie du bloc `with`.

Le retour de la méthode `__enter__` sera attribué à la variable spécifiée derrière le `as`.

Le bloc `with` est donc un bloc d'instructions très simple, offrant juste un sucre syntaxique autour d'un `try/except/finally`.

`__enter__` ne prend aucun paramètre, contrairement à `__exit__` qui en prend 3 : `exc_type`, `exc_value`, et `traceback`. Ces paramètres interviennent quand une exception survient dans le bloc `with`, et correspondent au type de l'exception levée, à sa valeur, et à son *traceback*. Dans le cas où aucune exception n'est survenue pendant le traitement de la ressource, ces 3 paramètres valent `None`.

`__exit__` retourne un booléen, intervenant dans la propagation des exceptions. En effet, si `True` est retourné, l'exception survenue dans le contexte sera attrapée.

Nous pouvons maintenant créer notre propre type de gestionnaire, contentons-nous pour le moment de quelque chose d'assez simple qui afficherait un message à l'entrée et à la sortie.

```
1 class MyContext:
2     def __enter__(self):
3         print('enter')
4         return self
5     def __exit__(self, exc_type, exc_value, traceback):
6         print('exit')
```

Et à l'utilisation :

```
1 >>> with MyContext() as ctx:
2     ...     print(ctx)
3     ...
4 enter
5 <__main__.MyContext object at 0x7f23cc446cf8>
6 exit
```

8.4. Simplifions-nous la vie avec la contextlib

La [contextlib](#) est un module de la bibliothèque standard comportant divers outils ou gestionnaires de contexte bien utiles.

IV. Plus loin, un peu plus loin

Par exemple, une classe, `ContextDecorator`, permet de transformer un gestionnaire de contexte en décorateur, et donc de pouvoir l'utiliser comme l'un ou comme l'autre. Cela peut s'avérer utile pour créer un module qui mesurerait le temps d'exécution d'un ensemble d'instructions : on peut vouloir s'en servir via `with`, ou via un décorateur autour de notre fonction à mesurer.

Cet outil s'utilise très facilement, il suffit que notre gestionnaire de contexte hérite de `ContextDecorator`.

```
1 from contextlib import ContextDecorator
2 import time
3
4 class spent_time(ContextDecorator):
5     def __enter__(self):
6         self.start = time.time()
7     def __exit__(self, *_):
8         print('Elapsed {:.3}s'.format(time.time() - self.start))
```

Et à l'utilisation :

```
1 >>> with spent_time():
2 ...     print('x')
3 ...
4 x
5 Elapsed 0.000106s
6 >>> @spent_time()
7 ... def func():
8 ...     print('x')
9 ...
10 >>> func()
11 x
12 Elapsed 0.000108s
```

Intéressons-nous maintenant à `contextmanager`. Il s'agit d'un décorateur capable de transformer une fonction génératrice en *context manager*. Cette fonction génératrice devra disposer d'un seul et unique `yield`. Tout ce qui est présent avant le `yield` sera exécuté en entrée, et ce qui se situe ensuite s'exécutera en sortie.

```
1 >>> from contextlib import contextmanager
2 >>> @contextmanager
3 ... def context():
4 ...     print('enter')
5 ...     yield
6 ...     print('exit')
7 ...
8 >>> with context():
```

IV. Plus loin, un peu plus loin

```
9 ...     print('during')
10 ...
11 enter
12 during
13 exit
```

Attention tout de même, une exception levée dans le bloc d'instructions du `with` remonterait jusqu'au générateur, et empêcherait donc l'exécution du `__exit__`.

```
1 >>> with context():
2 ...     raise Exception
3 ...
4 enter
5 Traceback (most recent call last):
6   File "<stdin>", line 2, in <module>
7 Exception
```

Il convient donc d'utiliser un `try/finally` si vous souhaitez vous assurer que la fin du générateur sera toujours exécutée.

```
1 >>> @contextmanager
2 ... def context():
3 ...     try:
4 ...         print('enter')
5 ...         yield
6 ...     finally:
7 ...         print('exit')
8 ...
9 >>> with context():
10 ...     raise Exception
11 ...
12 enter
13 exit
14 Traceback (most recent call last):
15   File "<stdin>", line 2, in <module>
16 Exception
```

Enfin, le module contient divers gestionnaires de contexte, qui sont :

- `closing` [↗](#) qui permet de fermer automatiquement un objet (par sa méthode `close`) ;
- `suppress` [↗](#) afin de supprimer certaines exceptions survenues dans un contexte ;
- `redirect_stdout` [↗](#) pour rediriger temporairement la sortie standard du programme.

8.5. Réutilisabilité et réentrance

8.5.1. Réutilisabilité

Nous avons vu que la syntaxe du bloc `with` était `with expr as var`. Dans les exemples précédents, nous avons toujours une expression `expr` à usage unique, qui était évaluée pour le `with`.

Mais un même gestionnaire de contexte pourrait être utilisé à plusieurs reprises si l'expression est chaque fois une même variable. En reprenant la classe `MyContext` définie plus tôt :

```
1 >>> ctx = MyContext()
2 >>> with ctx:
3 ...     pass
4 ...
5 enter
6 exit
7 >>> with ctx:
8 ...     pass
9 ...
10 enter
11 exit
```

`MyContext` est un gestionnaire de contexte réutilisable : on peut utiliser ses instances à plusieurs reprises dans des blocs `with` successifs.

Mais les fichiers tels que retournés par `open` ne sont par exemple pas réutilisables : une fois sortis du bloc `with`, le fichier est fermé, il est donc impossible d'ouvrir un nouveau contexte.

```
1 >>> f = open('filename', 'r')
2 >>> with f:
3 ...     pass
4 ...
5 >>> with f:
6 ...     pass
7 ...
8 Traceback (most recent call last):
9   File "<stdin>", line 1, in <module>
10  ValueError: I/O operation on closed file.
```

Notre gestionnaire `context` créé grâce au décorateur `contextmanager` n'est pas non plus réutilisable : il dépend d'un générateur qui ne peut être itéré qu'une fois.

8.5.2. Réentrance

Un cas particulier de la réutilisabilité est celui de la réentrance. Un gestionnaire de contexte est réentrant quand il peut être utilisé dans des `with` imbriqués.

```
1 >>> ctx = MyContext()
2 >>> with ctx:
3 ...     with ctx:
4 ...         pass
5 ...
6 enter
7 enter
8 exit
9 exit
```

On peut alors prendre l'exemple des classes `Lock` et `RLock` du module `threading`, qui servent à poser des verrous sur des ressources. Le premier est un gestionnaire réutilisable (seulement) et le second est réentrant.

Pour bien distinguer la différences entre les deux, je vous propose les codes suivant.

```
1 >>> from threading import Lock
2 >>> lock = Lock()
3 >>> with lock:
4 ...     with lock:
5 ...         pass
6 ...
```

Python bloque à l'exécution de ces instructions. En effet, le bloc intérieur demande l'accès à une ressource (`lock`) déjà occupée par le bloc extérieur. Python met en pause l'exécution en attendant que la ressource se libère. Mais celle-ci ne se libérera qu'en sortie du bloc extérieur, qui attend la fin de l'exécution du bloc intérieur.

Les deux blocs s'attendent mutuellement, l'exécution ne se terminera donc jamais. On est ici dans un cas de blocage, appelé *dead lock*. Dans notre cas, nous pouvons sortir à l'aide d'un `Ctrl+C` ou en fermant l'interpréteur.

Passons à `RLock` maintenant.

```
1 >>> from threading import RLock
2 >>> lock = RLock()
3 >>> with lock:
4 ...     with lock:
5 ...         pass
6 ...
```


Celui-ci supporte les `with` imbriqués, il est réentrant.

8.6. TP : Redirection de sortie (`redirectstdout`)

Nous allons ici mettre en place un gestionnaire de contexte équivalent à `redirect_stdout` pour rediriger la sortie standard vers un autre fichier. Il sera aussi utilisable en tant que décorateur pour rediriger la sortie standard de fonctions.

La redirection de sortie est une opération assez simple en Python. La sortie standard est identifiée par l'attribut/fichier `stdout` du module `sys`. Pour rediriger la sortie standard, il suffit alors de faire pointer `sys.stdout` vers un autre fichier.

Notre gestionnaire de contexte sera construit avec un fichier dans lequel rediriger la sortie. Nous enregistrerons donc ce fichier dans un attribut de l'objet.

À l'entrée du contexte, on gardera une trace de la sortie courante (`sys.stdout`) avant de la remplacer par notre cible. Et en sortie, il suffira de faire à nouveau pointer `sys.stdout` vers la précédente sortie standard, préalablement enregistrée.

Nous pouvons faire hériter notre classe de `ContextDecorator` afin de pouvoir l'utiliser comme décorateur.

```
1 import sys
2 from contextlib import ContextDecorator
3
4 class redirect_stdout(ContextDecorator):
5     def __init__(self, file):
6         self.file = file
7
8     def __enter__(self):
9         self.old_output = sys.stdout
10        sys.stdout = self.file
11
12    def __exit__(self, exc_type, exc_value, traceback):
13        sys.stdout = self.old_output
```

Pour tester notre gestionnaire de contexte, nous allons nous appuyer sur les `StringIO` du module `io`. Il s'agit d'objets se comportant comme des fichiers, mais dont tout le contenu est stocké en mémoire, et accessible à l'aide d'une méthode `getvalue`.

```
1 >>> from io import StringIO
2 >>> output = StringIO()
3 >>> with redirect_stdout(output):
4 ...     print('ceci est écrit dans output')
5 ...
6 >>> print('ceci est écrit sur la console')
```

IV. Plus loin, un peu plus loin

```
7 ceci est écrit sur la console
8 >>> output.getvalue()
9 'ceci est écrit dans output\n'
```

```
1 >>> output = StringIO()
2 >>> @redirect_stdout(output)
3 ... def addition(a, b):
4 ...     print('result =', a + b)
5 ...
6 >>> addition(3, 5)
7 >>> output.getvalue()
8 'result = 8\n'
```

Notre gestionnaire de contexte se comporte comme nous le souhaitions, mais possède cependant une lacune : il n'est pas réentrant.

```
1 >>> output = StringIO()
2 >>> redir = redirect_stdout(output)
3 >>> with redir:
4 ...     with redir:
5 ...         print('ceci est écrit dans output')
6 ...
7 >>> print('ceci est écrit sur la console')
```

Comme on le voit, ou plutôt comme on ne le voit pas, le dernier affichage n'est pas imprimé sur la console, mais toujours dans `output`. En effet, lors de la deuxième entrée dans `redir`, `sys.stdout` ne pointait plus vers la console mais déjà vers notre `StringIO`, et la trace sauvegardée (`self.old_output`) est alors perdue puisqu'assignée à `sys.stdout`.

Pour avoir un gestionnaire de contexte réentrant, il nous faudrait gérer une pile de fichiers de sortie. Ainsi, en entrée, la sortie actuelle serait ajoutée à la pile avant d'être remplacée par le fichier cible. Et en sortie, il suffirait de retirer le dernier élément de la pile et de l'assigner à `sys.stdout`.

```
1 import sys
2
3 class redirect_stdout(ContextDecorator):
4     def __init__(self, file):
5         self.file = file
6         self.stack = []
7
8     def __enter__(self):
9         self.stack.append(sys.stdout)
10        sys.stdout = self.file
```

```
11
12     def __exit__(self, exc_type, exc_value, traceback):
13         sys.stdout = self.stack.pop()
```

Vous pouvez constater en reprenant les tests précédent que cette version est parfaitement fonctionnelle (pensez juste à réinitialiser votre interpréteur suite aux tests qui ont définitivement redirigé `sys.stdout` vers une `StringIO`).

Ne changeons pas les bonnes habitudes, ces quelques pages de documentation vous régaleront autant que les précédentes.

- Définition du terme gestionnaire de contexte : <https://docs.python.org/3/glossary.html#term-context-manager> ↗
- Gestionnaires de contexte : <https://docs.python.org/3/library/stdtypes.html#context-manager-types> ↗
- Blocs `with` : <https://docs.python.org/3/reference/datamodel.html#with-statement-context-managers> ↗
- Module `contextlib` : <https://docs.python.org/3/library/contextlib.html> ↗
- PEP liée au bloc `with` : <https://www.python.org/dev/peps/pep-0343/> ↗

9. Accesseurs et descripteurs

L'expression `foo.bar` est en apparence très simple : on accède à l'attribut `bar` d'un objet `foo`. Cependant, divers mécanismes entrent en jeu pour nous retourner cette valeur, nous permettant d'accéder à des attributs définis à la volée.

Nous allons découvrir dans ce chapitre quels sont ces mécanismes, et comment les manipuler.

Sachez premièrement que `foo.bar` revient à exécuter

- `getattr(foo, 'bar')`

Il s'agit là de la lecture, deux fonctions sont équivalentes pour la modification et la suppression :

- `setattr(foo, 'bar', value)` pour `foo.bar = value`
- `delattr(foo, 'bar')` pour `del foo.bar`

9.1. L'attribut de Dana

Que font réellement `getattr`, `setattr` et `delattr` ? Elles appellent des méthodes spéciales de l'objet.

`setattr` et `delattr` sont les cas les plus simples, la correspondance est faite avec les méthodes `__setattr__` et `__delattr__`. Ces deux méthodes prennent les mêmes paramètres (en plus de `self`) que les fonctions auxquelles elles correspondent. `__setattr__` prendra donc le nom de l'attribut et sa nouvelle valeur, et `__delattr__` le nom de l'attribut.

Quant à `getattr`, la chose est un peu plus complexe, car deux méthodes spéciales lui correspondent : `__getattr__` et `__getattribute__`. Ces deux méthodes prennent en paramètre le nom de l'attribut. La première est appelée lors de la récupération de tout attribut. La seconde est réservée aux cas où l'attribut n'a pas été trouvé (si `__getattribute__` lève une `AttributeError`).

Ces méthodes sont chargées de retourner la valeur de l'attribut demandé. Il est en cela possible d'implémenter des attributs dynamiquement, en modifiant le comportement des méthodes : par exemple une condition sur le nom de l'attribut pour retourner une valeur particulière.

Par défaut, `__getattribute__` retourne les attributs définis dans l'objet (contenus dans son dictionnaire `__dict__` que nous verrons plus loin), et lève une `AttributeError` si l'attribut ne l'est pas. `__getattr__` n'est pas présente de base dans l'objet, et n'a donc pas de comportement par défaut. Il est plutôt conseillé de passer par cette dernière pour implémenter nos attributs dynamiques.

Ainsi, il nous suffit de coupler les méthodes de lecture, d'écriture, et/ou de suppression pour disposer d'attributs dynamiques. Il faut aussi penser à relayer les appels aux méthodes parentes

IV. Plus loin, un peu plus loin

via `super` pour utiliser le comportement par défaut quand on ne sait pas gérer l'attribut en question.

Le cas de `__getattr__` est un peu plus délicat : n'étant pas implémentée dans la classe `object`, il n'est pas toujours possible de relayer l'appel. Il convient alors de travailler au cas par cas, en utilisant `super` si la classe parente implémente `__getattr__`, ou en levant une `AttributeError` sinon.

L'exemple suivant présente une classe `Temperature` dont les instances possèdent deux attributs `celsius` et `fahrenheit` qui sont générés à la volée, et non stockés dans l'objet.

```
1 class Temperature:
2     def __init__(self):
3         self.value = 0
4
5     def __getattr__(self, name):
6         if name == 'celsius':
7             return self.value
8         if name == 'fahrenheit':
9             return self.value * 1.8 + 32
10        raise AttributeError(name)
11
12    def __setattr__(self, name, value):
13        if name == 'celsius':
14            self.value = value
15        elif name == 'fahrenheit':
16            self.value = (value - 32) / 1.8
17        else:
18            super().__setattr__(name, value)
```

Et à l'utilisation :

```
1 >>> t = Temperature()
2 >>> t.celsius = 37
3 >>> t.celsius
4 37
5 >>> t.fahrenheit
6 98.6 # Ou valeur approximative
7 >>> t.fahrenheit = 212
8 >>> t.celsius
9 100.0
```

9.1.1. dict et slots

Le `__dict__` dont je parle plus haut est le dictionnaire contenant les attributs d'un objet Python. Par défaut, il contient tous les attributs que vous définissez sur un objet (si vous ne modifiez pas

IV. Plus loin, un peu plus loin

le fonctionnement de `setattr`). En effet, chaque fois que vous créez un attribut (`foo.bar = value`), celui-ci est enregistré dans le dictionnaire des attributs de l'objet (`foo.__dict__['bar'] = value`). La méthode `__getattr__` de l'objet se contente donc de rechercher l'attribut dans le dictionnaire de l'objet et de ses parents (type de l'objet et classes dont ce type hérite).

Les slots sont une seconde manière de procéder, en vue de pouvoir optimiser le stockage de l'objet. Par défaut, lors de la création d'un objet, le dictionnaire `__dict__` est créé afin de pouvoir y stocker l'ensemble des attributs. Si la classe définit un itérable `__slots__` contenant les noms des attributs possibles de l'objet, une structure dynamique telle que le dictionnaire n'est plus nécessaire, `__dict__` ne sera donc pas instancié lors de la création d'un nouvel objet. Notez tout de même que si votre classe définit un `__slots__`, vous ne pourrez plus définir d'autres attributs sur l'objet que ceux décrits dans les slots.

Je vous invite à consulter la section de la documentation consacrée aux slots pour plus d'informations : <https://docs.python.org/3/reference/datamodel.html#slots> ↗

9.1.2. MRO

J'évoquais précédemment le comportement de `__getattr__`, qui consiste à consulter le dictionnaire de l'objet puis de ses parents. Ce mécanisme est appelé *method resolution order* ou plus généralement *MRO*.

Chaque classe que vous définissez possède une méthode `mro`. Elle retourne un *tuple* contenant l'ordre des classes à interroger lors de la résolution d'un appel sur l'objet. C'est ce *MRO* qui définit la priorité des classes parentes lors d'un héritage multiple (quelle classe interroger en priorité), c'est encore lui qui est utilisé lors d'un appel à `super`, afin de savoir à quelle classe `super` fait référence. En interne, la méthode `mro` fait appel à l'attribut `__mro__` de la classe.

Le comportement par défaut de `foo.__getattr__('bar')` est donc assez simple :

1. On recherche dans `foo.__dict__` la présence d'une clef 'bar', dont on retourne la valeur si la clef existe ;
2. On recherche dans les `__dict__` de toutes les classes référencées par `type(foo).mro()`, en s'arrêtant à la première valeur trouvée ;
3. On lève une exception `AttributeError` si l'attribut n'a pu être trouvé.

Pour bien comprendre le fonctionnement du *MRO*, je vous propose de regarder quelques exemples d'héritage.

Premièrement, définissons plusieurs classes :

```
1 class A: pass
2 class B(A): pass
3 class C: pass
4 class D(A, C): pass
5 class E(B, C): pass
6 class F(D, E): pass
7 class G(E, D): pass
```

IV. Plus loin, un peu plus loin

Puis observons.

```
1 >>> object.mro()
2 (<class 'object'>,)
3 >>> A.mro()
4 (<class '__main__.A'>, <class 'object'>)
5 >>> B.mro()
6 (<class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
7 >>> C.mro()
8 (<class '__main__.C'>, <class 'object'>)
9 >>> D.mro()
10 (<class
11     '__main__.D'>, <class '__main__.A'>, <class '__main__.C'>, <class 'object'>)
12 >>> E.mro()
13 (<class
14     '__main__.E'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
15 >>> F.mro()
16 (<class
17     '__main__.F'>, <class '__main__.D'>, <class '__main__.E'>, <class '__main__.A'>,
18     <class '__main__.C'>, <class 'object'>)
19 >>> G.mro()
20 (<class
21     '__main__.G'>, <class '__main__.E'>, <class '__main__.B'>, <class '__main__.A'>,
22     <class '__main__.C'>, <class 'object'>)
```

On constate bien que les classes les plus à gauche sont prioritaires lors d'un héritage, mais aussi que le mécanisme de *MRO* évite la présence de doublons dans la hiérarchie.

On remarque qu'en cas de doublon, les classes sont placées le plus loin possible du début de la liste : par exemple, **A** est placée après **B** et non après **D** dans le *MRO* de **F**.

Cela peut nous poser problème dans certains cas.

```
1 >>> class H(A, B): pass
2 ...
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   TypeError: Cannot create a consistent method resolution
6   order (MRO) for bases B, A
```

En effet, nous cherchons à hériter d'abord de **A** en la plaçant à gauche, mais **A** étant aussi la mère de **B**, le *MRO* souhaiterait la placer à la fin, ce qui provoque le conflit.

Tout fonctionne très bien dans l'autre sens :

```

1 >>> class H(B, A): pass
2 ...

```

9.2. Les descripteurs

Les descripteurs sont une manière de placer des comportements plus évolués derrière des attributs. En effet, plutôt que toujours recourir à `__getattr__` et consorts, ils sont un autre moyen d'avoir des attributs dynamiques. Les propriétés (*properties*) sont des exemples de descripteurs.

Un descripteur se définit comme attribut d'une classe, et devient accessible en tant qu'attribut de ses instances. Pour cela, le descripteur peut implémenter des méthodes spéciales `__get__`/`__set__`/`__delete__` qui seront respectivement appelées lors de la lecture/écriture/suppression de l'attribut sur une instance de la classe.

Par exemple, si une classe `Foo` définit un descripteur de type `Descriptor` sous son attribut `attr`, alors, avec `foo` instance de `Foo` :

- `foo.attr` fera appel à `Descriptor.__get__` ;
- `foo.attr = valeur` à `Descriptor.__set__` ;
- et `del foo.attr` à `Descriptor.__delete__`.

La méthode `__get__` du descripteur prend deux paramètres : `instance` et `owner`. `instance` correspond à l'objet depuis lequel on accède à l'attribut. Dans le cas où l'attribut est récupéré depuis la classe (`Foo.attr` plutôt que `foo.attr`), `instance` vaudra `None`. C'est alors que `owner` intervient, ce paramètre contient toujours la classe définissant le descripteur (`Foo`).

`__set__` prend simplement l'instance et la nouvelle valeur, et `__delete__` se contente de l'instance. Contrairement à `__get__`, ces deux dernières méthodes ne peuvent s'utiliser que sur les instances, et non sur la classe⁶, d'où l'absence du paramètre `owner`.

Pour reprendre notre exemple précédent sur les températures, nous pourrions avoir deux descripteurs `Celsius` et `Fahrenheit`, qui modifieraient à leur manière la valeur de notre objet `Temperature`.

```

1 class Celsius:
2     def __get__(self, instance, owner):
3         # Dans le cas où on appellerait `Temperature.celsius`
4         # On préfère retourner le descripteur lui-même
5         if instance is None:
6             return self
7         return instance.value
8     def __set__(self, instance, value):
9         instance.value = value

```

6. En effet, redéfinir `A.attr` ou le supprimer ne doit déclencher aucune méthode spéciale du descripteur, ça revient juste à redéfinir/supprimer l'attribut de classe.


```
10
11 class Fahrenheit:
12     def __get__(self, instance, owner):
13         if instance is None:
14             return self
15         return instance.value * 1.8 + 32
16     def __set__(self, instance, value):
17         instance.value = (value - 32) / 1.8
18
19 class Temperature:
20     # On instancie les deux attributs de la classe
21     celsius = Celsius()
22     fahrenheit = Fahrenheit()
23
24     def __init__(self):
25         self.value = 0
```

Je vous laisse exécuter à nouveau les exemples précédents pour constater que le comportement est le même.

9.2.1. La méthode `__set_name__`

Depuis Python 3.6⁷, les descripteurs peuvent aussi être pourvus d'une méthode `__set_name__`. Cette méthode est appelée pour chaque assignation d'un descripteur à un attribut dans le corps de la classe. La méthode reçoit en paramètres la classe et le nom de l'attribut auquel le descripteur est assigné.

```
1 >>> class Descriptor:
2 ...     def __set_name__(self, owner, name):
3 ...         print(name)
4 ...
5 >>> class A:
6 ...     value = Descriptor()
7 ...
8 value
```

Le descripteur peut ainsi agir dynamiquement sur la classe en fonction du nom de son attribut.

Nous pouvons imaginer un descripteur `PositiveValue`, qui assurera qu'un attribut sera toujours positif. Le descripteur stockera ici sa valeur dans un attribut de l'instance, en utilisant pour cela son nom préfixé d'un *underscore*.

7. <https://zestedesavoir.com/articles/1540/sortie-de-python-3-6/#principales-nouveautes> ↗

```
1 class PositiveValue:
2     def __get__(self, instance, owner):
3         return getattr(instance, self.attr)
4
5     def __set__(self, instance, value):
6         setattr(instance, self.attr, max(0, value))
7
8     def __set_name__(self, owner, name):
9         self.attr = '_' + name
10
11 class A:
12     x = PositiveValue()
13     y = PositiveValue()
```

```
1 >>> a = A()
2 >>> a.x = 15
3 >>> a.x
4 15
5 >>> a._x
6 15
7 >>> a.x -= 20
8 >>> a.x
9 0
10 >>> a.y = -1
11 >>> a.y
12 0
```

9.3. Les propriétés

Les propriétés (ou *properties*) sont un moyen de simplifier l'écriture de descripteurs et de leurs 3 méthodes spéciales.

En effet, `property` est une classe qui, à la création d'un objet, prend en paramètre les fonctions `fget`, `fset` et `fdel` qui seront respectivement appelées par `__get__`, `__set__` et `__delete__`.

On pourrait ainsi définir une version simplifiée de `property` comme ceci :

```
1 class my_property:
2     def __init__(self, fget, fset, fdel):
3         self.fget = fget
4         self.fset = fset
5         self.fdel = fdel
```

IV. Plus loin, un peu plus loin

```
6     def __get__(self, instance, owner):
7         return self.fget(instance)
8     def __set__(self, instance, value):
9         return self.fset(instance, value)
10    def __delete__(self, instance):
11        return self.fdel(instance)
```

Pour faire de `my_property` un clone parfait de `property`, il nous faudrait gérer le cas où `instance` vaut `None` dans la méthode `__get__` ; et permettre à `my_property` d'être utilisé en tant que décorateur autour du `getter`. Nous verrons dans la section exercices comment compléter notre classe à cet effet.

Les propriétés disposent aussi de décorateurs `getter`, `setter` et `deleter` pour redéfinir les fonctions `fget/fset/fdel`.

À l'utilisation, les propriétés nous offrent donc un moyen simple et élégant de réécrire notre classe `Temperature`.

```
1  class Temperature:
2      def __init__(self):
3          self.value = 0
4
5      @property
6      def celsius(self): # le nom de la méthode devient le nom de la
7                          propriété
8                          return self.value
9      @celsius.setter
10     def celsius(self, value): # le setter doit porter le même nom
11                               self.value = value
12
13     @property
14     def fahrenheit(self):
15         return self.value * 1.8 + 32
16     @fahrenheit.setter
17     def fahrenheit(self, value):
18         self.value = (value - 32) / 1.8
```

Pour plus d'informations sur l'utilisation des propriétés, je vous renvoie [ici](#) .

9.4. Les méthodes

Les méthodes en Python vous réservent aussi bien des surprises. Si vous avez déjà rencontré les termes de méthodes de classe (*class methods*), méthodes statiques (*static methods*), ou méthodes préparées (*bound methods*), vous avez pu vous demander comment cela fonctionnait.

IV. Plus loin, un peu plus loin

En fait, les méthodes sont des descripteurs vers les fonctions que vous définissez à l'intérieur de votre classe. Elles sont même ce qu'on appelle des *non-data descriptors*, c'est-à-dire des descripteurs qui ne définissent ni *setter*, ni *deleter*.

Définissons une simple classe A possédant différents types de méthodes.

```
1 class A:
2     def method(self):
3         return self
4     @staticmethod
5     def staticmeth():
6         pass
7     @classmethod
8     def clsmeth(cls):
9         return cls
```

Puis observons à quoi correspondent les différents accès à ces méthodes.

```
1 >>> a = A() # on crée une instance `a` de `A`
2 >>> A.method # méthode depuis la classe
3 <function A.method at 0x7fd412ad5f28>
4 >>> a.method # méthode depuis l'instance
5 <bound method A.method of <__main__.A object at 0x7fd412a3ad68>>
6 >>> A.staticmeth # méthode statique depuis la classe
7 <function A.staticmeth at 0x7fd412a41048>
8 >>> a.staticmeth # depuis l'instance
9 <function A.staticmeth at 0x7fd412a41048>
10 >>> A.clsmeth # méthode de classe depuis la classe
11 <bound method type.clsmeth of <class '__main__.A'>>
12 >>> a.clsmeth # depuis l'instance
13 <bound method type.clsmeth of <class '__main__.A'>>
```

On remarque que certains accès retournent des fonctions, et d'autres des *bound methods*, mais quelle différence ? En fait, la différence survient lors de l'appel, pour le passage du premier paramètre.

Ne vous êtes-vous jamais demandé comment l'objet courant arrivait dans `self` lors de l'appel d'une méthode ? C'est justement parce qu'il s'agit d'une *bound method*. C'est en fait une méthode dont le premier paramètre est déjà préparé, et qu'il n'y aura donc pas besoin de spécifier à l'appel. C'est le descripteur qui joue ce rôle, il est le seul à savoir si vous utilisez la méthode depuis une instance ou depuis la classe (`instance` valant `None` dans ce second cas), et connaît toujours le premier paramètre à passer (`instance`, `owner`, ou rien). Il peut ainsi construire un nouvel objet (*bound method*), qui lorsqu'il sera appelé se chargera de relayer l'appel à la vraie méthode en lui ajoutant ce paramètre.

Le même comportement est utilisé pour les méthodes de classes, où la classe de l'objet doit être passée en premier paramètre (`cls`). Le cas des méthodes statiques est en fait le plus simple, il

IV. Plus loin, un peu plus loin

ne s'agit que de fonctions qui ne prennent pas de paramètres spéciaux, donc qui ne nécessitent pas d'être décorées par le descripteur.

On remarque aussi que, `A.method` retournant une fonction et non une méthode préparée, il nous faudra indiquer une instance lors de l'appel.

Pour rappel, voici comment s'utilisent ces différentes méthodes :

```
1 >>> A.method(a)
2 <__main__.A object at 0x7fd412a3ad68>
3 >>> a.method()
4 <__main__.A object at 0x7fd412a3ad68>
5 >>> A.staticmeth()
6 >>> a.staticmeth()
7 >>> A.clsmeth()
8 <class '__main__.A'>
9 >>> a.clsmeth()
10 <class '__main__.A'>
```

9.5. TP : Méthodes

Pour clore ce chapitre, je vous propose d'implémenter les descripteurs `staticmethod` et `classmethod`. J'ajouterai à cela un descripteur `method` qui reproduirait le comportement par défaut des méthodes en Python.

Pour résumer :

- Ces trois descripteurs sont de type *non-data* (n'implémentent que `__get__`) ;
- `my_staticmethod`
 - Retourne la fonction cible, qu'elle soit utilisée depuis la classe ou depuis l'instance ;
- `my_classmethod`
 - Retourne une méthode préparée avec la classe en premier paramètre ;
 - Même comportement que l'on utilise la méthode de classe depuis la classe ou l'instance ;
- `my_method`
 - Si utilisée depuis la classe, retourne la fonction ;
 - Sinon, retourne une méthode préparée avec l'instance en premier paramètre.

Notez que vous pouvez vous aider du type `MethodType` (`from types import MethodType`) pour créer vos *bound methods*. Il s'utilise très facilement, prenant en paramètres la fonction cible et le premier paramètre de cette fonction.

```
1 class my_staticmethod:
2     def __init__(self, func):
3         self.func = func
4     def __get__(self, instance, owner):
```

IV. Plus loin, un peu plus loin

```
5     return self.func
```

```
1 from types import MethodType
2
3 class my_classmethod:
4     def __init__(self, func):
5         self.func = func
6     def __get__(self, instance, owner):
7         return MethodType(self.func, owner)
```

```
1 from types import MethodType
2
3 class my_method:
4     def __init__(self, func):
5         self.func = func
6     def __get__(self, instance, owner):
7         if instance is None:
8             return self.func
9         return MethodType(self.func, instance)
```

La documentation est cette fois bien plus fournie, je vous souhaite donc une bonne lecture. Les liens de ce chapitre sont particulièrement intéressants, notamment concernant le protocole des descripteurs et le *MRO*.

- Définition du terme descripteur : <https://docs.python.org/3/glossary.html#term-descriptor> ↗
- Personnaliser l'accès aux attributs : <https://docs.python.org/3/reference/datamodel.html#customizing-attribute-access> ↗
- Mise en œuvre des descripteurs : <https://docs.python.org/3/howto/descriptor.html> ↗
- Protocole des descripteurs : <https://docs.python.org/3/reference/datamodel.html#implementing-descriptors> ↗
- *Fonction property* : <https://docs.python.org/3/library/functions.html#property> ↗
- Description du *MRO* : <https://www.python.org/download/releases/2.3/mro/> ↗

Cinquième partie

La rentrée des classes

9.6. Et de leurs parents

Les classes renferment encore bien des secrets, que nous allons tenter de percer dans les présents chapitres.

10. Types

Il vous est peut-être arrivé de lire qu'en Python tout était objet. Il faut cependant nuancer quelque peu : tout ne l'est pas, une instruction n'est pas un objet par exemple. Mais toutes les valeurs que l'on peut manipuler sont des objets.

À quoi peut-on alors reconnaître un objet ? Cela correspond à tout ce qui peut être assigné à une variable. Ainsi, les nombres, les chaînes de caractère, les fonctions ou même les classes sont des objets. Et ce sont ici ces dernières qui nous intéressent.

10.1. Instance, classe et métaclass

On sait que tout objet est instance d'une classe. On dit aussi que la classe est le type de l'objet. Et donc, tout objet a un type. Le type d'un objet peut être récupéré grâce à la *fonction* `type`.

```
1 >>> type(5)
2 <class 'int'>
3 >>> type('foo')
4 <class 'str'>
5 >>> type(lambda: None)
6 <class 'function'>
```

Mais si les classes sont des objets, quel est alors leur type ?

```
1 >>> type(object)
2 <class 'type'>
```

Le type d'`object` est `type`. En effet, `type` est un peu plus complexe que ce que l'on pourrait penser, nous y reviendrons dans le prochain chapitre.

On notera simplement qu'une classe est alors une instance de la classe `type`. Et qu'une classe telle que `type`, qui permet d'instancier d'autres classes, est appelée une métaclass.

Instancier une classe pour en créer une nouvelle n'est pas forcément évident. Nous avons plutôt l'habitude d'hériter d'une classe existante. Mais dans les cas où nous créons une classe par héritage, c'est aussi une instanciation de `type` qui est réalisée en interne.

10.1.1. Caractéristiques des classes

Les classes (ou *type objects*) sont un ensemble d'objets qui possèdent quelques caractéristiques communes :

- Elles héritent d'`object` (mise à part `object` elle-même) ;
- Elles sont des instances plus ou moins directes de `type` (de `type` ou de classes héritant de `type`) ;
- On peut en hériter ;
- Elles peuvent être instanciées (elles sont des *callable*s qui retournent des objets de ce `type`).

```
1 >>> int.__bases__ # int hérite d'object
2 (<class 'object'>,)
3 >>> type(int) # int est une instance de type
4 <class 'type'>
5 >>> class A(int): pass # on peut hériter de la classe int
6 >>> int() # on peut instancier int
7 0
8 >>> type(int()) # ce qui retourne un objet du type int
9 <class 'int'>
```

Et on observe que notre classe `A` est elle aussi instance de `type`.

```
1 >>> type(A)
2 <class 'type'>
```

10.2. Le vrai constructeur

En Python, la méthode spéciale `__init__` est souvent appelée constructeur de l'objet. Il s'agit en fait d'un abus de langage : `__init__` ne construit pas l'objet, elle intervient après la création de ce dernier pour l'initialiser.

Le vrai constructeur d'une classe est `__new__`. Cette méthode prend la classe en premier paramètre (le paramètre `self` n'existe pas encore puisque l'objet n'est pas créé), et doit retourner l'objet nouvellement créé (contrairement à `__init__`). Les autres paramètres sont identiques à ceux reçus par `__init__`.

C'est aussi `__new__` qui est chargée d'appeler l'initialiseur `__init__` (ce que fait `object.__new__` par défaut, en lui passant aussi la liste d'arguments).

```
1 >>> class A:
2 ...     def __new__(cls):
```

```
3 ...     print('création')
4 ...     return super().__new__(cls)
5 ...     def __init__(self):
6 ...         print('initialisation')
7 ...
8 >>> A()
9 création
10 initialisation
11 <__main__.A object at 0x7ffb15ef9048>
```

Nous choisissons ici de faire appel à `object.__new__` dans notre constructeur (via `super`), mais nous n'y sommes pas obligés. Rien ne nous oblige non plus — mise à part la logique — à retourner un objet du bon type.

10.2.1. Le cas des immutables

La méthode `__new__` est particulièrement utile pour les objets immutables. En effet, il est impossible d'agir sur les objets dans la méthode `__init__`, puisque celle-ci intervient après la création, et que l'objet n'est pas modifiable.

Si l'on souhaite hériter d'un type immuable (`int`, `str`, `tuple`), et agir sur l'initialisation de l'objet, il est donc nécessaire de redéfinir `__new__`. Par exemple une classe `Point2D` immuable, qui hériterait de `tuple`.

```
1 class Point2D(tuple):
2     def __new__(cls, x, y):
3         return super().__new__(cls, (x, y))
4
5     @property
6     def x(self):
7         return self[0]
8
9     @property
10    def y(self):
11        return self[1]
```

```
1 >> p = Point2D(3, 5)
2 >>> p.x
3 3
4 >>> p.y
5 5
6 >>> p.x = 10
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
```

```
9 AttributeError: can't set attribute
10 >>> p[0] = 10
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13 TypeError: 'Point2D' object does not support item assignment
```

10.3. Paramètres d'héritage

Quand nous créons une classe, nous savons que nous pouvons spécifier entre parenthèses les classes à hériter.

```
1 class A(B, C): # A hérite de B et C
2     pass
```

Les classes parentes sont ici comme les arguments positionnels d'un appel de fonction. Vous vous en doutez peut-être maintenant, mais il est aussi possible de préciser des arguments nommés.

```
1 class A(B, C, foo='bar', x=3):
2     pass
```

Cette fonctionnalité existait déjà en Python 3.5, mais était assez étrange et se gérait au niveau de la métaclasse. Le comportement est simplifié avec [Python 3.6 qui ajoute une méthode spéciale pour gérer ces arguments](#) [↗](#).

Il est donc maintenant possible d'implémenter la méthode de classe `__init_subclass__`, qui recevra tous les arguments nommés. La méthode ne sera pas appelée pour la classe courante, mais le sera pour toutes ses classes filles.

Pour reprendre notre class `Deque`, nous pourrions imaginer une classe `TypedDeque` qui générerait des listes d'éléments d'un type prédéfini. Nous lèverions alors une exception pour toute insertion de valeur d'un type inadéquat.

```
1 class TypedDeque(Deque):
2     elem_type = None
3
4     def __init_subclass__(cls, type, **kwargs):
5         super().__init_subclass__(**kwargs)
6         cls.elem_type = type
7
8     @classmethod
9     def check_type(cls, value):
```

V. La rentrée des classes

```
10     if cls.elem_type is not None and not isinstance(value,
11         cls.elem_type):
12         raise TypeError('Cannot insert element of type '
13             f'{type(value).__name__} in {cls.__name__}')
14
15     def append(self, value):
16         self.check_type(value)
17         super().append(value)
18
19     def insert(self, i, value):
20         self.check_type(value)
21         super().insert(i, value)
22
23     def __setitem__(self, key, value):
24         self.check_type(value)
25         super().__setitem__(key, value)
26
27 class IntDeque(TypedDeque, type=int):
28     pass
29
30 class StrDeque(TypedDeque, type=str):
31     pass
```

```
1 >>> d = IntDeque([0, 1, 2])
2 >>> d.append(3)
3 >>> list(d)
4 [0, 1, 2, 3]
5 >>> d.append('foo')
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   File "init_subclass.py", line 91, in append
9     self.check_type(value)
10  File "init_subclass.py", line 87, in check_type
11    raise TypeError('Cannot insert element of type '
12  TypeError: Cannot insert element of type str in IntDeque
13 >>>
14 >>> d = StrDeque()
15 >>> d.append('foo')
16 >>> list(d)
17 ['foo']
18 >>> d.insert(0, 5)
19 Traceback (most recent call last):
20   File "<stdin>", line 1, in <module>
21   File "init_subclass.py", line 95, in insert
22     self.check_type(value)
23   File "init_subclass.py", line 87, in check_type
24    raise TypeError('Cannot insert element of type '
```

```
25 TypeError: Cannot insert element of type int in StrDeque
```

Le paramètre `cls` de la méthode `__init_subclass__` correspond bien ici à la classe fille. Il convient d'utiliser `super` pour faire appel aux `__init_subclass__` des autres parents, en leur donnant le reste des arguments nommés. On note aussi qu'`__init_subclass__` étant obligatoirement une méthode de classe, l'utilisation du décorateur `@classmethod` est facultative.

10.4. TP : Liste immutable

Nous en étions resté sur notre liste chaînée à l'opérateur d'égalité, qui rendait la liste non-hashable. Je vous disais alors que l'on s'intéresserait à un type de liste immutable (et donc hashable) : chose promise, chose due.

Nous savons que pour créer un type immutable en Python, il faut hériter d'un autre type immutable. Par commodité, nous choisissons `tuple` puisqu'il nous permet en tant que conteneur de stocker des données.

Pour rappel, notre liste se compose de deux classes : `Node` et `Deque`. Nos nouvelles classes se nommeront `ImmutableNode` et `ImmutableDeque`.

`ImmutableNode` est un ensemble de deux éléments : le contenu du maillon dans `value`, et le maillon suivant (ou `None`) dans `next`. On peut aisément représenter cette classe par un `tuple` de deux éléments. On ajoutera juste deux propriétés, `value` et `next`, pour faciliter l'accès à ces valeurs.

```
1 class ImmutableNode(tuple):
2     def __new__(cls, value, next=None):
3         return super().__new__(cls, (value, next))
4
5     @property
6     def value(self):
7         return self[0]
8
9     @property
10    def next(self):
11        return self[1]
```

Passons maintenant à `ImmutableDeque`. Au final, il s'agit aussi d'un ensemble de deux éléments : `first` et `last`, les deux extrêmités de la liste.

Mais `ImmutableDeque` présente un autre défi, c'est cette classe qui est chargée de créer les maillons, qui sont ici immutables. Cela signifie que le `next` de chaque maillon doit être connu lors de sa création.

Pour rappel, la classe sera instanciée avec un itérable en paramètre, celui-ci servant à créer les maillons. Il nous faudra donc itérer sur cet ensemble en connaissant l'élément suivant.

V. La rentrée des classes

Je vous propose pour cela une méthode de classe réursive, `create_node`. Cette méthode recevra un itérateur en paramètre, récupérera la valeur actuelle avec la fonction `next`, puis appellera la méthode sur le reste de l'itérateur. `create_node` retournera un objet `ImmutableNode`, qui sera donc utilisé comme maillon `next` dans l'appel parent. En cas de `StopIteration` (fin de l'itérateur), `create_node` renverra simplement `None`.

```
1 class ImmutableDeque(tuple):
2     def __new__(cls, iterable=()):
3         first = cls.create_node(iter(iterable))
4         last = first
5         while last and last.next:
6             last = last.next
7         return super().__new__(cls, (first, last))
8
9     @classmethod
10    def create_node(cls, iterator):
11        try:
12            value = next(iterator)
13        except StopIteration:
14            return None
15        next_node = cls.create_node(iterator)
16        return ImmutableNode(value, next_node)
```

À la manière de notre classe `ImmutableNode`, nous ajoutons des propriétés `first` et `last`. Celles-ci diffèrent un peu tout de même : puisque `__getitem__` sera surchargé dans la classe, nous devons faire appel au `__getitem__` parent, *via* `super`.

```
1 @property
2 def first(self):
3     return super().__getitem__(0)
4
5 @property
6 def last(self):
7     return super().__getitem__(1)
```

Les autres méthodes (`__contains__`, `__len__`, `__getitem__`, `__iter__` et `__eq__`) seront identiques à celles de la classe `Deque`. On prendra seulement soin, dans `__getitem__`, de remplacer les occurrence de `Deque` par `ImmutableDeque` en cas de *slicing*, ou de faire appel au type de `self` pour construire la nouvelle liste.

Les méthodes de modification (`append`, `insert`, `__setitem__`) ne sont bien sûr pas à implémenter. On remarque d'ailleurs que l'attribut `last` de nos listes n'a pas vraiment d'intérêt ici, puisqu'il n'est pas utilisé pour faciliter l'ajout d'éléments en fin de liste.

Enfin, on peut maintenant ajouter une méthode `__hash__`, pour rendre nos objets *hashables*. Pour cela, nous ferons appel à la méthode `__hash__` du parent, qui retournera le condensat du *tuple*.

V. La rentrée des classes

```
1 def __hash__(self):
2     return super().__hash__()
```

Nous pouvons maintenant passer au test de notre nouvelle classe.

```
1 >>> d = ImmutableDeque(range(10))
2 >>> d
3 ((0, (1, (2, (3, (4, (5, (6, (7, (8, (9, None)))))))))), (9, None))
4 >>> list(d)
5 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
6 >>> d[1:-1:2]
7 ((1, (3, (5, (7, None))))), (7, None))
8 >>> list(d[1:-1:2])
9 [1, 3, 5, 7]
10 >>> 5 in d
11 True
12 >>> 11 in d
13 False
14 >>> len(d)
15 10
16 >>> d[0], d[1], d[5], d[9]
17 (0, 1, 5, 9)
18 >>> d == ImmutableDeque(range(10))
19 True
20 >>> d == ImmutableDeque(range(9))
21 False
22 >>> hash(d)
23 -9219024882206086640
24 >>> {d: 0}
25 {((0, (1, (2, (3, (4, (5, (6, (7, (8, (9, None)))))))))), (9,
26     None)): 0}
27 >>> {d: 0}[d]
28 0
```

Revenons maintenant avec la documentation sur les concepts étudiés dans ce chapitre.

- Définition du terme type : <https://docs.python.org/3/glossary.html#term-type> ↗
- Méthode `__new__` : https://docs.python.org/3/reference/datamodel.html#object.__new__ ↗
- Paramètres d'héritage : https://docs.python.org/3/reference/datamodel.html#object.__init_subclass__ ↗

Un autre très bon article, qui revient sur les concepts de classe, d'instance, de métaclasse et d'héritage : http://www.cafepython.com/article/python_types_and_objects ↗

11. Métaclasses

Nous connaissons maintenant une première métaclasse, `type`. Une métaclasse est une classe dont les instances sont des classes.

Ce chapitre a pour but de présenter comment créer nos propres métaclasses, et les mécanismes mis en œuvre par cela.

11.1. Quel est donc ce type ?

Ainsi, vous l'aurez compris, `type` n'est pas utile que pour connaître le type d'un objet. Dans l'utilisation que vous connaissiez, `type` prend un unique paramètre, et en retourne le type.

Pour notre autre utilisation, ses paramètres sont au nombre de 3 :

- `name` – une chaîne de caractères représentant le nom de la classe à créer ;
- `bases` – un tuple contenant les classes dont nous héritons (`object` est implicite) ;
- `dict` – le dictionnaire des attributs et méthodes de la nouvelle classe.

```
1 >>> type('A', (), {})  
2 <class '__main__.A'>  
3 >>> A = type('A', (), {'x': 4})  
4 >>> A.x  
5 4  
6 >>> A().x  
7 4  
8 >>> type(A)  
9 <class 'type'>  
10 >>> type(A())  
11 <class '__main__.A'>
```

Nous avons ici une classe `A`, strictement équivalente à la suivante :

```
1 class A:  
2     x = 4
```

Voici maintenant un exemple plus complet, avec héritage et méthodes.

```
1 >>> B = type('B', (int,), {})
2 >>> B()
3 0
4 >>> B = type('B', (int,), {'next': lambda self: self + 1})
5 >>> B(5).next()
6 6
7 >>> def C_prev(self):
8 ...     return self - 1
9 ...
10 >>> C = type('C', (B,), {'prev': C_prev})
11 >>> C(5).prev()
12 4
13 >>> C(5).next()
14 6
```

11.2. Les métaclasses

11.2.1. À quoi sert une métaclassse ?

Lorsqu'on découvre les métaclasses, il est courant de commencer à les utiliser à tort et à travers. Les métaclasses sont un mécanisme complexe, et rendent plus difficile la compréhension du code. Il est alors préférable de s'en passer dans la limite du possible : les chapitres précédents présentent ce qu'il est possible de réaliser sans métaclasses.

L'intérêt principal des métaclasses est d'agir sur les classes lors de leur création, dans la méthode `__new__` de la métaclassse. Par exemple pour ajouter à la classe de nouvelles méthodes ou des attributs supplémentaires. Ou encore pour transformer les attributs définis dans le corps de la classe.

Je vous propose plus loin dans ce chapitre l'exemple d'`Enum`, une implémentation du [type énuméré](#) en Python, pour illustrer l'utilité des métaclasses. Un autre exemple est celui des *ORM*⁸, où les classes représentent des tables d'une base de données. Les attributs de classe y sont transformés pour réaliser le schéma de la table, et de nouvelles méthodes sont ajoutées pour manipuler les entrées.

11.2.2. Notre première métaclassse

Pour mieux saisir le concept de métaclassse, je vous propose maintenant de créer notre première métaclassse. Nous savons que `type` est une classe, et possède donc les mêmes caractéristiques que les autres classes, énoncées plus tôt.

8. *Object-Relational mapping*, ou *Mapping objet-relationnel*, technique fournissant une interface orientée objet aux bases de données.

V. La rentrée des classes

```
1 >>> type.__bases__ # type hérite d'object
2 (<class 'object'>,)
3 >>> type(type) # type est une instance de type
4 <class 'type'>
5 >>> type('A', (), {}) # on peut instancier type
6 <class '__main__.A'>
7 >>> class M(type): pass # on peut hériter de type
```

Toutes les classes étant des instances de `type`, on en déduit qu'il faut passer par `type` pour toute construction de classe. Une métaclasse est donc une classe héritant de `type`. La classe `M` du précédent exemple est une nouvelle métaclasse.

Une métaclasse opérera plus souvent lors de la création d'une classe que lors de son initialisation. C'est donc dans le constructeur (méthode `__new__`) que le tout va s'opérer. Avec une métaclasse `M`, la méthode `M.__new__` sera appelée chaque fois que nous créerons une nouvelle classe de métaclasse `M`.

Le constructeur d'une métaclasse devra donc prendre les mêmes paramètres que `type`, et faire appel à ce dernier pour créer notre objet.

```
1 >>> class M(type):
2 ...     def __new__(cls, name, bases, dict):
3 ...         return super().__new__(cls, name, bases, dict)
4 ...
5 >>> A = M('A', (), {})
6 >>> A
7 <class '__main__.A'>
8 >>> type(A)
9 <class '__main__.M'>
```

Nous avons ainsi créé notre propre métaclasse, et l'avons utilisée pour instancier une nouvelle classe.

Une autre syntaxe pour instancier notre métaclasse est possible, à l'aide du mot clef `class` : la métaclasse à utiliser peut être spécifiée à l'aide du paramètre `metaclass` entre les parenthèses derrière le nom de la classe.

```
1 >>> class B(metaclass=M):
2 ...     pass
3 ...
4 >>> type(B)
5 <class '__main__.M'>
```

11.2.3. Préparation de la classe

Nous avons étudié dans le chapitre sur les accesseurs l'attribut `__dict__` des classes. Celui-ci est un dictionnaire, mais à quel moment est-il créé ?

Lors de la définition d'une classe, avant même de s'attaquer à ce que contient son corps, celle-ci est préparée. C'est-à-dire que le dictionnaire `__dict__` est instancié, afin d'y stocker tout ce qui sera défini dans le corps.

Par défaut, la préparation d'une classe est donc un appel à `dict`, qui retourne un dictionnaire vide. Mais si la métaclasse est dotée d'une méthode de classe `__prepare__`, celle-ci sera appelée en lieu et place de `dict`. Cette méthode doit toutefois retourner un dictionnaire ou objet similaire. Elle peut par exemple initialiser ce dictionnaire avec des valeurs par défaut.

```
1 >>> class M(type):
2 ...     @classmethod
3 ...     def __prepare__(cls, name, bases):
4 ...         return {'test': lambda self: print(self)}
5 ...
6 >>> class A(metaclass=M): pass
7 ...
8 >>> A().test()
9 <__main__.A object at 0x7f886cfd4e10>
```

11.2.4. Une métaclasse utile

Maintenant que nous savons créer et utiliser des métaclasses, servons-nous-en à bon escient. Il faut bien noter que les métaclasses répondent à des problèmes bien spécifiques, leur utilisation pourrait ne pas vous sembler évidente.

Les énumérations en Python sont implémentées à l'aide de métaclasses.

```
1 >>> from enum import Enum
2 >>> class Color(Enum):
3 ...     red = 1
4 ...     green = 2
5 ...     blue = 3
6 ...
7 >>> Color.red
8 <Color.red: 1>
9 >>> Color(1) is Color.red
10 True
```

En héritant d'`Enum`, on hérite aussi de sa métaclasse (`EnumMeta`)

V. La rentrée des classes

```
1 >>> type(Color)
2 <class 'enum.EnumMeta'>
```

Attention d'ailleurs, lorsque vous héritez de plusieurs classes, assurez-vous toujours que leurs métaclasses soient compatibles (la hiérarchie entre les différentes métaclasses doit être linéaire).

Une implémentation simplifiée possible d'Enum est la suivante :

```
1 class EnumMeta(type):
2     def __new__(cls, name, bases, dict):
3         # Cache dans lequel les instances seront stockées
4         dict['__mapping__'] = {}
5         # Membres de l'énumération (tous les attributs qui ne sont
6         # pas du type __foo__)
7         members = {k: v for (k, v) in dict.items() if not
8                     (k.startswith('__') and k.endswith('__'))}
9         enum = super().__new__(cls, name, bases, dict)
10        # On instancie toutes les valeurs possibles et on les
11        # intègre à la classe
12        for key, value in members.items():
13            value = enum(value)
14            value.name = key # On spécifie le nom du membre
15            setattr(enum, key, value) # On le définit comme attribut
16            # de classe
17        return enum
18
19 class Enum(metaclass=EnumMeta):
20     def __new__(cls, value):
21         # On retourne la valeur depuis le cache si elle existe
22         if value in cls.__mapping__:
23             return cls.__mapping__[value]
24         v = super().__new__(cls)
25         v.value = value
26         v.name = ''
27         # On l'ajoute au cache
28         cls.__mapping__[value] = v
29         return v
30     def __repr__(self):
31         return '<{}.{}: {}>'.format(type(self).__name__, self.name,
32                                     self.value)
```

Notre exemple précédent avec les couleurs s'exécute de la même manière.

11.3. Utiliser une fonction comme métaclasse

Par extension, on appelle parfois métaclasse tout *callable* qui renverrait une classe lorsqu'il serait appelé.

Ainsi, une fonction faisant appel à `type` serait considérée comme métaclasse.

```
1 >>> def meta(*args):
2 ...     print('enter metaclass')
3 ...     return type(*args)
4 ...
5 >>> class A(metaclass=meta):
6 ...     pass
7 ...
8 enter metaclass
```

Cependant, on ne peut pas à proprement parler de métaclasse, celle de notre classe A étant toujours `type`.

```
1 >>> type(A)
2 <class 'type'>
```

Ce qui fait qu'à l'héritage, l'appel à la métaclasse serait perdu (cet appel n'étant réalisé qu'une fois).

```
1 >>> class B(A):
2 ...     pass
3 ...
4 >>> type(B)
5 <class 'type'>
```

Pour rappel, le comportement avec une « vraie » métaclasse serait le suivant :

```
1 >>> class meta(type):
2 ...     def __new__(cls, *args):
3 ...         print('enter metaclass')
4 ...         return super().__new__(cls, *args)
5 ...
6 >>> class A(metaclass=meta):
7 ...     pass
8 ...
9 enter metaclass
10 >>> type(A)
```

```

11 <class '__main__.meta'>
12 >>> class B(A):
13     ...     pass
14     ...
15 enter metaclass
16 >>> type(B)
17 <class '__main__.meta'>

```

11.4. TP : Types immutables

Nous avons vu dans le chapitre précédent comment réaliser un type immutable. Nous voulons maintenant aller plus loin, en mettant en place une métaclasse qui nous permettra facilement de créer de nouveaux types immutables.

Déjà, à quoi ressemblerait une classe d'objets immutables ? Il s'agirait d'une classe dont les noms d'attributs seraient fixés à l'avance pour tous les objets. Et les attributs en question ne seraient bien sûr pas modifiables sur les objets. La classe pourrait bien sûr définir des méthodes, mais toutes ces méthodes auraient un accès en lecture seule sur les instances.

On aurait par exemple quelque chose comme :

```

1 class Point(metaclass=ImmutableMeta):
2     __fields__ = ('x', 'y')
3
4     def distance(self):
5         return (self.x**2 + self.y**2)**0.5

```

```

1 >>> p = Point(x=3, y=4)
2 >>> p.x
3 3
4 >>> p.y
5 4
6 >>> p.distance()
7 5.0
8 >>> p.x = 0
9 Traceback (most recent call last):
10   File "<stdin>", line 1, in <module>
11 AttributeError: can't set attribute
12 >>> p.z = 0
13 Traceback (most recent call last):
14   File "<stdin>", line 1, in <module>
15 AttributeError: 'Point' object has no attribute 'z'

```

11.4.1. Hériter de `tuple`

Plusieurs solutions s'offrent à nous pour mener ce travail. Nous pouvons, comme précédemment, faire hériter tous nos immutables de `tuple`. Il faudra alors faire pointer chacun des noms d'attributs sur les éléments du *tuple*, via des propriétés par exemple. On peut simplifier cela avec `namedtuple`, qui réalise cette partie du travail.

Notre métaclasse se chargerait ainsi d'extraire les champs du type immuable, de créer un `namedtuple` correspondant, puis en faire hériter notre classe immuable.

```
1 class ImmutableMeta(type):
2     def __new__(cls, name, bases, dict):
3         fields = dict.pop('__fields__', ())
4         bases += (namedtuple(name, fields),)
5         return super().__new__(cls, name, bases, dict)
```

Si l'on implémente une classe `Point` comme dans l'exemple plus haut, on remarque que la classe se comporte comme convenu jusqu'au `p.z = 0`. En effet, il nous est ici possible d'ajouter de nouveaux attributs à nos objets, pourtant voulus immutables.

```
1 >>> p = Point(x=3, y=4)
2 >>> p.z = 5
3 >>> p.z
4 5
```

11.4.2. Les slots à la rescousse

Comme nous l'avons vu avec les accesseurs, il est possible de définir un ensemble `__slots__` des attributs possibles des instances de cette classe. Celui-ci a entre autres pour effet d'empêcher de définir d'autres attributs à nos objets.

C'est donc dans ce sens que nous allons maintenant l'utiliser. Nos types immutables n'ont besoin d'aucun attribut : tout ce qu'ils stockent est contenu dans un *tuple*, et les accesseurs sont des propriétés. Ainsi, notre métaclasse `ImmutableMeta` peut simplement définir un attribut `__slots__ = ()` à nos classes.

```
1 class ImmutableMeta(type):
2     def __new__(cls, name, bases, dict):
3         fields = dict.pop('__fields__', ())
4         bases += (namedtuple(name, fields),)
5         dict['__slots__'] = ()
6         return super().__new__(cls, name, bases, dict)
```


11.4.3. Le problème des méthodes de `tuple`

Nous avons maintenant entre les mains une classe de types immutables répondant aux critères décrits plus haut. Mais si on y regarde de plus près, on remarque un léger problème : nos classes possèdent des méthodes incongrues héritées de `tuple` et `namedtuple`. On voit par exemple des méthodes `__getitem__`, `count` ou `index` qui ne nous sont d'aucune utilité et polluent les classes. `__getitem__` est d'autant plus problématique qu'il s'agit d'un opérateur du langage, qui se retrouve automatiquement surchargé.

```

1 >>> dir(Point)
2 ['__add__', '__class__', '__contains__', '__delattr__', '__dict__',
3  '__dir__',
4  '__doc__', '__eq__', '__format__', '__ge__', '__getattr__',
5  '__getitem__',
6  '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__',
7  '__iter__',
8  '__le__', '__len__', '__lt__', '__module__', '__mul__', '__ne__',
9  '__new__',
10 '__reduce__', '__reduce_ex__', '__repr__', '__rmul__',
11 '__setattr__',
12 '__sizeof__', '__slots__', '__str__', '__subclasshook__',
13  '__asdict__', '_fields',
14  '_make', '_replace', '_source', 'count', 'index', 'x', 'y']

```

Alors on peut dans un premier temps choisir d'hériter de `tuple` plutôt que d'un `namedtuple` pour faire un premier tri, mais ça ne règle pas le soucis. Et il nous est impossible de supprimer ces méthodes, puisqu'elles ne sont pas définies dans nos classes mais dans une classe parente.

Il faut alors bidouiller, en remplaçant les méthodes par des attributs levant des `AttributeError` pour faire croire à leur absence, en redéfinissant `__dir__` pour les en faire disparaître, etc. Mais nos objets continueront à être des *tuples* et ces méthodes resteront accessibles d'une manière ou d'une autre (en appelant directement `tuple.__getitem__`, par exemple).

Nous verrons dans les exercices complémentaires une autre piste pour créer nos propres types immutables.

Et pour terminer ce chapitre, un nouveau rappel vers la documentation Python. Je vous encourage vraiment à la lire le plus possible, elle est très complète et très instructive, bien que parfois un peu bordélique.

- Définition du terme métaclasse : <https://docs.python.org/3/glossary.html#term-metaclass> ↗
- Personnalisation de la création de classes : <https://docs.python.org/3/reference/data-model.html#customizing-class-creation> ↗
- Classe `type` : <https://docs.python.org/3/library/functions.html#type> ↗
- PEP relative aux métaclasses : <https://www.python.org/dev/peps/pep-3115/> ↗

V. La rentrée des classes

Je tenais aussi à présenter ici [un tutoriel/guide en 8 parties de Sam&Max dédié au modèle objet, à `type` et aux métaclasses](#) [↗](#) .

Enfin, je ne peux que vous conseiller de vous pencher sur [les sources de CPython](#) [↗](#) pour comprendre les mécanismes internes. Notamment [le fichier `Objects/typeobject.c`](#) [↗](#) qui définit les classes `type` et `object`.

12. Classes abstraites

Nous avons durant ce cours étudié diverses interfaces (conteneurs, itérables, hashables, etc.). Nous allons maintenant voir comment les classes abstraites peuvent nous permettre de reconnaître ces interfaces.

Je ne reviendrai pas ici sur la notion même de classe abstraite, [présentée dans cet autre cours](#) ⁸, dont il est conseillé de prendre connaissance avant de passer à la suite.

12.1. Module abc

Pour rappel, le module `abc` ⁸ donne accès à la classe `ABC` qui permet par héritage de construire une classe abstraite, et au décorateur `abstractmethod` pour définir des méthodes abstraites.

Une autre classe importante réside dans ce module, `ABCMeta`. `ABCMeta` est la métaclasse de la classe `ABC`, et donc le type de toutes les classes abstraites. C'est `ABCMeta` qui s'occupe de référencer dans l'ensemble `__abstractmethods__` ⁹ les méthodes abstraites définies dans la classe.

Mais outre le fait de pouvoir spécifier les méthodes à implémenter, les classes abstraites de Python ont un autre but : définir une interface. Vous connaissez probablement `isinstance`, qui permet de vérifier qu'un objet est du bon type ; peut-être moins `issubclass`, pour vérifier qu'une classe *hérite* d'une autre.

```
1 >>> isinstance(4, int) # 4 est un int
2 True
3 >>> isinstance(4, str) # 4 n'est pas une str
4 False
5 >>> issubclass(int, object) # int hérite d'object
6 True
7 >>> issubclass(int, str) # int n'hérite pas de str
8 False
```

Ces deux fonctions sont en fait des opérateurs, qui font appel à des méthodes spéciales, et sont à ce titre surchargeables, comme nous le verrons par la suite.

J'ai utilisé plus haut le terme « hérite » pour décrire l'opérateur `issubclass`. C'est en fait légèrement différent, `issubclass` permet de vérifier qu'une classe est une sous-classe (ou sous-type) d'une autre.

Quand une classe hérite d'une autre, elle en devient un sous-type (sauf cas exceptionnels¹⁰). Mais elle peut aussi être sous-classe de classes dont elle n'hérite pas.

V. La rentrée des classes

C'est le but de la méthode `register` des classes `ABC`. Elle sert à enregistrer une classe comme sous-type de la classe abstraite.

Imaginons une classe abstraite `Sequence` correspondant aux types de séquences connus (`str`, `list`, `tuple`)¹¹. Ces types sont des *builtins* du langage, il ne nous est pas possible de les redéfinir pour les faire hériter de `Sequence`. Mais la méthode `register` de notre classe abstraite `Sequence` va nous permettre de les enregistrer comme sous-classes.

```
1 >>> import abc
2 >>> class Sequence(abc.ABC):
3     ...     pass
4     ...
5 >>> Sequence.register(str)
6 <class 'str'>
7 >>> Sequence.register(list)
8 <class 'list'>
9 >>> Sequence.register(tuple)
10 <class 'tuple'>
11 >>> isinstance('foo', Sequence)
12 True
13 >>> isinstance(42, Sequence)
14 False
15 >>> issubclass(list, Sequence)
16 True
17 >>> issubclass(dict, Sequence)
18 False
```

12.2. isinstance

Nous venons de voir que `isinstance` était un opérateur, et qu'il était surchargeable. Nous allons ici nous intéresser à la mise en œuvre de cette surcharge.

Pour rappel, la surcharge d'opérateur se fait par la définition d'une méthode spéciale dans le type de l'objet. Par exemple, il est possible d'utiliser `+` sur le nombre `4` parce que `4` est de type `int`, et qu'`int` implémente la méthode `__add__`.

`isinstance` est un opérateur qui s'applique à une classe (la classe dont on cherche à savoir si tel objet en est l'instance). La surcharge se fera donc dans le type de cette classe, c'est-à-dire dans la métaclass.

La méthode spéciale correspondant à l'opérateur est `__instancecheck__`, qui reçoit en paramètre l'objet à tester, et retourne un booléen (`True` si l'objet est du type en question, `False` sinon).

9. L'ensemble `__abstractmethods__` est ensuite analysé pour savoir si une classe peut être instanciée, le constructeur d'`object` levant une erreur dans le cas échéant.

10. Voir à ce propos la section « `issubclass` ».

11. Pour rappel, une séquence est un objet *indexable* et *sliceable*.

V. La rentrée des classes

On peut par exemple imaginer une classe `ABCIterable`, qui cherchera à savoir si un objet donné est itérable (possède une méthode `__iter__`). On teste pour cela si cet objet a un attribut `__iter__`, et si cet attribut est *callable*.

```
1 class ABCIterableMeta(type):
2     def __instancecheck__(self, obj):
3         return hasattr(obj, '__iter__') and callable(obj.__iter__)
4
5 class ABCIterable(metaclass=ABCIterableMeta):
6     pass
```

```
1 >>> isinstance([], ABCIterable)
2 True
3 >>> isinstance((), ABCIterable)
4 True
5 >>> isinstance('foo', ABCIterable)
6 True
7 >>> isinstance({'a': 'b'}, ABCIterable)
8 True
9 >>> isinstance(18, ABCIterable)
10 False
11 >>> isinstance(object(), ABCIterable)
12 False
```

Quelques dernières précisions sur `isinstance` : l'opérateur est un peu plus complexe que ce qui a été montré.

Premièrement, `isinstance` peut recevoir en deuxième paramètre un *tuple* de types plutôt qu'un type simple. Il regardera alors si l'objet donné en premier paramètre est une instance de l'un de ces types.

```
1 >>> isinstance(4, (int, str))
2 True
3 >>> isinstance('foo', (int, str))
4 True
5 >>> isinstance(['bar'], (int, str))
6 False
```

Ensuite, la méthode `__instancecheck__` n'est pas toujours appelée. Lors d'un appel `isinstance(obj, cls)`, la méthode `__instancecheck__` est appelée que si `type(obj)` n'est pas `cls`.

On peut s'en rendre compte avec une classe dont `__instancecheck__` renverrait `False` pour tout objet testé.

```
1 >>> class NoInstancesMeta(type):
2 ...     def __instancecheck__(self, obj):
3 ...         return False
4 ...
5 >>> class NoInstances(metaclass=NoInstancesMeta):
6 ...     pass
7 ...
8 >>> isinstance(NoInstances(), NoInstances)
9 True
```

En revanche, si nous héritons de notre classe `NoInstances` :

```
1 >>> class A(NoInstances):
2 ...     pass
3 ...
4 >>> isinstance(A(), NoInstances)
5 False
```

Pour comprendre le fonction d'`isinstance`, on pourrait grossièrement réécrire l'opérateur avec la fonction suivante.¹²

```
1 def isinstance(obj, cls):
2     if type(obj) is cls:
3         return True
4     if isinstance(type(cls), tuple):
5         return any(isinstance(obj, c) for c in cls)
6     return type(cls).__instancecheck__(cls, obj)
```

12.3. `issubclass`

Dans la même veine qu'`isinstance`, nous avons donc l'opérateur `issubclass`, qui vérifie qu'une classe est sous-classe d'une autre.

La surcharge se fait là aussi sur la métaclasse, à l'aide de la méthode spéciale `__subclasscheck__`. Cette méthode est très semblable à `__instancecheck__` : en plus de `self` (la classe courante), elle reçoit en paramètre la classe à tester. Elle retourne elle aussi un booléen (`True` si la classe donnée est une sous-classe de l'actuelle, `False` sinon).

Reprenons ici l'exemple précédent des itérables : notre classe `ABCIterable` permet de tester si une classe est un type d'objets itérables.

12. Voir à ce propos la fonction `PyObject_IsInstance` du fichier [Objects/abstract.c](#) des sources de CPython.

V. La rentrée des classes

```
1 class ABCIterableMeta(type):
2     def __subclasscheck__(self, cls):
3         return hasattr(cls, '__iter__') and callable(cls.__iter__)
4
5 class ABCIterable(metaclass=ABCIterableMeta):
6     pass
```

```
1 >>> isinstance(list, ABCIterable)
2 True
3 >>> isinstance(tuple, ABCIterable)
4 True
5 >>> isinstance(str, ABCIterable)
6 True
7 >>> isinstance(dict, ABCIterable)
8 True
9 >>> isinstance(int, ABCIterable)
10 False
11 >>> isinstance(object, ABCIterable)
12 False
```

Cet exemple est d'ailleurs meilleur que le précédent, puisque comme Python il vérifie que la méthode `__iter__` est présente au niveau de la classe et pas au niveau de l'instance.

Comme `isinstance`, `issubclass` peut recevoir en deuxième paramètre un *tuple* de différentes classes à tester.

```
1 >>> isinstance(int, (int, str))
2 True
3 >>> isinstance(str, (int, str))
4 True
5 >>> class Integer(int):
6     ...     pass
7     ...
8 >>> isinstance(Integer, (int, str))
9 True
10 >>> isinstance(list, (int, str))
11 False
```

En revanche, pas de raccourci pour éviter l'appel à `__subclasscheck__`, même quand on cherche à vérifier qu'une classe est sa propre sous-classe.

```
1 >>> class NoSubclassesMeta(type):
2     ...     def __subclasscheck__(self, cls):
```

```

3 ...         return False
4 ...
5 >>> class NoSubclasses(metaclass=NoSubclassesMeta):
6 ...     pass
7 ...
8 >>> isinstance(NoSubclasses, NoSubclasses)
9 False

```

12.3.1. Le cas des classes ABC

Pour les classes abstraites ABC, c'est-à-dire qui ont `abc.ABCMeta` comme métaclasse, une facilité est mise en place. En effet, `ABCMeta` définit une méthode `__subclasscheck__` (qui s'occupe entre autres de gérer les classes enregistrées *via* `register`).

Pour éviter de recourir à une nouvelle métaclasse et redéfinir `__subclasscheck__`, la méthode d'`ABCMeta` relaie l'appel à la méthode de classe `__subclasshook__`, si elle existe. Ainsi, une classe abstraite n'a qu'à définir `__subclasshook__` si elle veut étendre le comportement d'`isinstance`.

```

1 >>> import abc
2 >>> class ABCIterable(abc.ABC):
3 ...     @classmethod
4 ...     def __subclasshook__(cls, subcls):
5 ...         return hasattr(subcls, '__iter__') and
6 ...             callable(subcls.__iter__)
7 >>> isinstance(list, ABCIterable)
8 True
9 >>> isinstance(int, ABCIterable)
10 False

```

On notera que la méthode `__subclasshook__` sert aussi à l'opérateur `isinstance`.

```

1 >>> isinstance([1, 2, 3], ABCIterable)
2 True

```

Contrairement à `__subclasscheck__`, `__subclasshook__` ne retourne pas forcément un booléen. Elle peut en effet retourner `True`, `False`, ou `NotImplemented`.

Dans le cas où elle retourne un booléen, il sera la valeur de retour de `isinstance/isinstance`. Mais dans le cas de `NotImplemented`, la main est rendue à la méthode `__subclasscheck__` d'`ABC`, qui s'occupe de vérifier si les classes sont parentes, ou si la classe est enregistrée (`register`).

V. La rentrée des classes

Nous allons donc réécrire notre classe `ABCIterable` de façon à retourner `True` si la classe implémente `__iter__`, et `NotImplemented` sinon. Ainsi, si la classe hérite d'`ABCIterable` mais n'implémente pas `__iter__`, elle sera tout de même considérée comme une sous-classe, ce qui n'est pas le cas actuellement.

```
1 >>> class ABCIterable(abc.ABC):
2 ...     @classmethod
3 ...     def __subclasshook__(cls, subcls):
4 ...         if hasattr(subcls, '__iter__') and
5 ...             callable(subcls.__iter__):
6 ...             return True
7 ...             return NotImplemented
8 >>> isinstance(list, ABCIterable)
9 True
10 >>> isinstance(int, ABCIterable)
11 False
12 >>> class X(ABCIterable): pass
13 ...
14 >>> isinstance(X, ABCIterable)
15 True
```

12.4. Collections abstraites

Nous connaissons le module `collections`, spécialisé dans les conteneurs ; et `abc`, dédié aux classes abstraites. Que donnerait le mélange des deux ? `collections.abc` !

Ce module fournit des classes abstraites toutes prêtes pour reconnaître les différentes interfaces du langage (`Container`, `Sequence`, `Mapping`, `Iterable`, `Iterator`, `Hashable`, `Callable`, etc.).

Assez simples à appréhender, ces classes abstraites testent la présence de méthodes essentielles au respect de l'interface.

```
1 >>> import collections.abc
2 >>> isinstance(10, collections.abc.Hashable)
3 True
4 >>> isinstance([10], collections.abc.Hashable)
5 False
6 >>> isinstance(list, collections.abc.Sequence)
7 True
8 >>> isinstance(dict, collections.abc.Sequence)
9 False
10 >>> isinstance(list, collections.abc.Mapping)
11 False
```

V. La rentrée des classes

```
12 >>> isinstance(dict, collections.abc.Mapping)
13 True
```

Outre la vérification d'interfaces, certaines de ces classes servent aussi de *mixins*, en apportant des méthodes abstraites et des méthodes prêtes à l'emploi.

La classe `MutableMapping`, par exemple, a pour méthodes abstraites `__getitem__`, `__setitem__`, `__delitem__`, `__iter__` et `__len__`. Mais la classe fournit en plus l'implémentation d'autres méthodes utiles aux *mappings* : `__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, `__ne__`, `pop`, `popitem`, `clear`, `update`, et `setdefault`.

C'est-à-dire qu'il suffit de redéfinir les 5 méthodes abstraites pour avoir un type de dictionnaires parfaitement utilisable.

```
1 class MyMapping(collections.abc.MutableMapping):
2     def __init__(self, *args, **kwargs):
3         super().__init__()
4         self._subdict = dict(*args, **kwargs)
5
6     def __getitem__(self, key):
7         return self._subdict[key]
8
9     def __setitem__(self, key, value):
10        self._subdict[key] = value
11
12    def __delitem__(self, key):
13        del self._subdict[key]
14
15    def __iter__(self):
16        return iter(self._subdict)
17
18    def __len__(self):
19        return len(self._subdict)
```

```
1 >>> m = MyMapping()
2 >>> m['a'] = 0
3 >>> m['b'] = m['a'] + 1
4 >>> len(m)
5 2
6 >>> list(m.keys())
7 ['b', 'a']
8 >>> list(m.values())
9 [1, 0]
10 >>> dict(m)
11 {'b': 1, 'a': 0}
12 >>> m.get('b')
```

```
13 1
14 >>> 'a' in m
15 True
16 >>> m.pop('a')
17 0
18 >>> 'a' in m
19 False
```

Dans un genre similaire, on notera aussi les classes du module `numbers` : `Number`, `Complex`, `Real`, `Rational`, `Integral`. Ces classes abstraites, en plus de reconnaître l'ensemble des types numériques, permettent par héritage de créer nos propres types de nombres.

12.5. TP : Reconnaissance d'interfaces

Je vous propose dans ce TP de nous intéresser à la reconnaissance d'interfaces comme nous l'avons fait dans ce chapitre. Nous allons premièrement écrire une classe `Interface`, héritant de `abc.ABC`, qui permettra de vérifier qu'un type implémente un certain nombre de méthodes. Cette classe sera destinée à être héritée pour spécifier quelles méthodes doivent être implémentées par quels types. On trouvera par exemple une classe `Container` héritant d'`Interface` pour vérifier la présence d'une méthode `__contains__`.

Les méthodes nécessaires pour se conformer au type seront inscrites dans un attribut de classe `__methods__`. Notre classe `Interface` définira la méthode `__subclasshook__` pour s'assurer que toutes les méthodes de la séquence `__methods__` sont présentes dans la classe.

La méthode `__subclasshook__` se déroulera en 3 temps :

- Premièrement, appeler l'implémentation parente *via* `super`, et retourner `False` si elle a retourné `False`. En effet, si la classe parente dit que le type n'est pas un sous-type, on est sûr qu'il n'en est pas un. Mais si la méthode parente retourne `True` ou `NotImplemented`, le doute peut persister ;
- Dans un second temps, nous récupérerons la liste de toutes les méthodes à vérifier. Il ne s'agit pas seulement de l'attribut `__methods__`, mais de cet attribut ainsi que celui de toutes les classes parentes ;
- Et finalement, nous testerons que chacune des méthodes est présente dans la classe, afin de retourner `True` si elle le sont toutes, et `NotImplemented` sinon.

Le deuxième point va nous amener à explorer le *MRO*, à l'aide de la méthode de classe `mro`, et de concaténer les attributs `__methods__` de toutes les classes (*via* la fonction `sum`). Afin de toujours récupérer une séquence, nous utiliserons `getattr(cls, '__methods__', ())`, qui nous retournera un *tuple* vide si l'attribut `__methods__` n'est pas présent.

Quant au 3ème point, la *builtin* `all` va nous permettre de vérifier que chaque nom de méthode est présent dans la classe, et qu'il s'agit d'un *callable* et donc d'une méthode.

Notre classe `Interface` peut alors se présenter comme suit.

V. La rentrée des classes

```
1 import abc
2
3 class Interface(abc.ABC):
4     # Attribut `__methods__` vide pour montrer sa structure
5     __methods__ = ()
6
7     @classmethod
8     def __subclasshook__(cls, subcls):
9         # Appel au __subclasshook__ parent
10        ret = super().__subclasshook__(cls, subcls)
11        if not ret:
12            return ret
13        # Récupération de toutes les méthodes
14        all_methods = sum((getattr(c, '__methods__', ()) for c in
15                           cls.mro()), ())
16        # Vérification de la présence des méthodes dans la classe
17        if all(callable(getattr(subcls, meth, None)) for meth in
18               all_methods):
19            return True
20        return NotImplemented
```

Nous pouvons dès lors créer nos nouvelles classes hérités d'`Interface` avec leurs propres attributs `__methods__`.

```
1 class Container(Interface):
2     __methods__ = ('__contains__',)
3
4 class Sized(Interface):
5     __methods__ = ('__len__',)
6
7 class SizedContainer(Sized, Container):
8     pass
9
10 class Subscriptable(Interface):
11     __methods__ = ('__getitem__',)
12
13 class Iterable(Interface):
14     __methods__ = ('__iter__',)
```

Et qui fonctionnent comme prévu.

```
1 >>> isinstance([], Iterable)
2 True
3 >>> isinstance([], Subscriptable)
4 True
```

V. La rentrée des classes

```
5 >>> isinstance([], SizedContainer)
6 True
7 >>> gen = (x for x in range(10))
8 >>> isinstance(gen, Iterable)
9 True
10 >>> isinstance(gen, Subscriptable)
11 False
12 >>> isinstance(gen, SizedContainer)
13 False
```

Pour terminer, un dernier tour par la documentation et ses pages intéressantes.

- Définition du terme classe abstraite : <https://docs.python.org/3/glossary.html#term-abstract-base-class> ↗
- Personnaliser la vérification de types : <https://docs.python.org/3/reference/datamodel.html#customizing-instance-and-subclass-checks> ↗
- Module `abc` : <https://docs.python.org/3/library/abc.html> ↗
- Module `collections.abc` : <https://docs.python.org/3/library/collections.abc.html> ↗
- Module `numbers` : <https://docs.python.org/3/library/numbers.html> ↗

Sixième partie

Pour quelques exercices de plus

12.6. L'histoire sans fin

Vous pensiez en avoir fini ? Malheureux !

Dans cette dernière partie, retrouvez des exercices plus complets pour mettre en œuvre les notions décrites dans ce cours. Car le Python vient en pythonant.

Tous les exercices sont rangés selon le chapitre auquel ils se rapportent principalement. Mais ces exercices pouvant mêler plusieurs chapitres, les pré-requis à connaître sont chaque fois décrits en en-tête.

13. Décorateurs

13.1. Vérification de types

Pré-requis : Annotations, Signatures, Décorateurs

L'intérêt de ce nouvel exercice va être de vérifier dynamiquement que les types des arguments passés à notre fonction sont les bons, à l'aide d'annotations sur les paramètres de la fonction. La vérification ne pourra ainsi se faire que sur les paramètres possédant un nom.

Notre décorateur va donc se charger d'analyser les paramètres lors de chaque appel à la fonction, et de les comparer un à un avec les annotations de notre fonction. Pour accéder aux annotations, nous allons à nouveau utiliser la fonction `signature`. La signature retournée comportant un attribut `parameters` contenant la liste des paramètres.

Ces paramètres peuvent être de différentes natures, suivant leur emplacement dans la ligne de définition de la fonction. Par exemple, si notre fonction se définit par :

```
1 def f(a, b, c='', *args, d=(), e=[], **kwargs): pass
```

- `*args` est un `VAR_POSITIONAL`, nous ne nous y intéresserons pas ici.
- `**kwargs` est un `VAR_KEYWORD`, de même, il ne nous intéresse pas dans notre cas.
- `a`, `b` et `c` sont des `POSITIONAL_OR_KEYWORD` (on peut les utiliser *via* des arguments positionnels ou *via* des arguments nommés).
- Enfin, `d` et `e` sont des `KEYWORD_ONLY` (on ne peut les utiliser que *via* des arguments nommés).
- Il existe aussi `POSITIONAL_ONLY`, mais celui-ci n'est pas représentable en Python ; il peut cependant exister dans des builtins ou des extensions. Ce type ne nous intéressera pas non plus ici.

Ainsi, nous voudrions récupérer les annotations des paramètres `POSITIONAL_OR_KEYWORD` et `KEYWORD_ONLY`. Reprenons notre fonction donnée plus haut et ajoutons lui des annotations pour certains paramètres :

```
1 def f(a:int, b, c:str='', *args, d=(), e:list=[], **kwargs): pass
```

Analysons maintenant les paramètres tels que définis dans la signature de la fonction.

VI. Pour quelques exercices de plus

```
1 >>> for p in signature(f).parameters.values():
2     ...     print(p.name, p.kind, p.annotation)
3 a POSITIONAL_OR_KEYWORD <class 'int'>
4 b POSITIONAL_OR_KEYWORD <class 'inspect._empty'>
5 c POSITIONAL_OR_KEYWORD <class 'str'>
6 args VAR_POSITIONAL <class 'inspect._empty'>
7 d KEYWORD_ONLY <class 'inspect._empty'>
8 e KEYWORD_ONLY <class 'list'>
9 kwargs VAR_KEYWORD <class 'inspect._empty'>
```

Nous retrouvons donc les types énoncés plus haut, ainsi que nos annotations (ou `empty` quand aucune annotation n'est donnée). Nous stockerons donc d'un côté les annotations des `POSITIONAL_OR_KEYWORD` dans une liste, et de l'autre celles des `KEYWORD_ONLY` dans un dictionnaire. Par la suite, `bind` rangera pour nous les paramètres de ce premier type dans `args`, et ceux du second dans `kwargs`.

```
1 from functools import wraps
2 from inspect import signature
3
4 def check_types(f):
5     sig = signature(f)
6     # Nous récupérons les types des paramètres positionnels
7     # (en remplaçant les annotations vides par None pour conserver
8     # l'ordre)
9     args_types = [(p.annotation if p.annotation != sig.empty else
10                   None)
11                   for p in sig.parameters.values()
12                   if p.kind == p.POSITIONAL_OR_KEYWORD]
13     # Puis ceux des paramètres nommés
14     # (il n'est pas nécessaire de conserver les empty ici)
15     kwargs_types = {p.name: p.annotation for p in
16                     sig.parameters.values()
17                     if p.kind == p.KEYWORD_ONLY and p.annotation !=
18                     p.empty}
19
20 @wraps(f)
21 def decorated(*args, **kwargs):
22     # On range correctement les paramètres des deux types
23     bind = sig.bind(*args, **kwargs)
24     # Vérification des paramètres positionnels
25     for value, typ in zip(bind.args, args_types):
26         if typ and not isinstance(value, typ):
27             raise
28             TypeError('{} must be of type {}'.format(value,
29                                                       typ))
30     # Et des paramètres nommés
31     for name, value in bind.kwargs.items():
```

VI. Pour quelques exercices de plus

```
25         # Si le type n'est pas précisé par l'annotation, on
           # considère object
26         # (toutes les valeurs sont de type object)
27         typ = kwargs_types.get(name, object)
28         if not isinstance(value, typ):
29             raise
           TypeError('{} must be of type {}'.format(value,
           typ))
30     return f(*args, **kwargs)
31     return decorated
```

Voyons maintenant ce que donne notre décorateur à l'utilisation.

```
1 >>> @check_types
2 ... def addition(a:int, b:int):
3 ...     return a + b
4 ...
5 >>> @check_types
6 ... def concat(a:str, b:str):
7 ...     return a + b
8 ...
9 >>> addition(1, 2)
10 3
11 >>> concat('x', 'y')
12 'xy'
13 >>> addition(1, 'y')
14 Traceback (most recent call last):
15   File "<stdin>", line 1, in <module>
16   File "<stdin>", line 19, in decorated
17   TypeError: y must be of type <class 'int'>
18 >>> concat(1, 2)
19 Traceback (most recent call last):
20   File "<stdin>", line 1, in <module>
21   File "<stdin>", line 19, in decorated
22   TypeError: 1 must be of type <class 'str'>
```

13.2. Mémoïsation

Pré-requis : Signatures, Décorateurs

Un des exemples les plus courants de mise en pratique des décorateurs est la réalisation d'un système de mise en cache (mémoïsation) : sauvegarder les résultats d'un calcul pour éviter de le refaire à chaque appel.

VI. Pour quelques exercices de plus

Nous allons débiter par une version simple : pour chaque appel, nous enregistrerons la valeur de retour associée au couple `(args, kwargs)` si celle-ci n'existe pas déjà. Dans le cas contraire, il nous suffira de retourner la valeur existante.

Seul bémol, nous ne pouvons pas stocker directement `(args, kwargs)` comme clef de notre dictionnaire, car certains objets n'y sont pas hashables (car modifiables, tel que le dictionnaire). Nous procéderons donc à l'aide d'une sérialisation via `repr` pour obtenir la représentation de nos paramètres sous forme d'une chaîne de caractères.

```
1 import functools
2
3 def memoize(f):
4     cache = {}
5     @functools.wraps(f)
6     def decorated(*args, **kwargs):
7         key = repr((args, kwargs))
8         if key not in cache:
9             cache[key] = f(*args, **kwargs)
10        return cache[key]
11    return decorated
```

Je vous conseille de le tester sur une fonction procédant à des affichages, pour bien constater la mise en cache.

```
1 >>> @memoize
2 ... def addition(a, b):
3 ...     print('Computing addition of {} and {}'.format(a, b))
4 ...     return a + b
5 ...
6 >>> addition(3, 5)
7 Computing addition of 3 and 5
8 8
9 >>> addition(3, 5)
10 8
11 >>> addition(3, 6)
12 Computing addition of 3 and 6
13 9
```

Comme je le disais, c'est une version simple, dans le sens où s'il nous venait à l'esprit d'utiliser une fonction tantôt avec des arguments positionnels, tantôt avec des arguments nommés, nous ne bénéficierions pas des capacités du cache.

```
1 >>> addition(3, b=5)
2 Computing addition of 3 and 5
3 8
```

VI. Pour quelques exercices de plus

```
4 >>> addition(a=3, b=5)
5 Computing addition of 3 and 5
6 8
```

13.2.1. Signatures

Afin d'avoir une représentation unique de nos arguments, nous allons alors utiliser les signatures de fonctions, et leur méthode `bind`. Nous obtiendrons ainsi un couple unique `(args, kwargs)`, où tous les arguments nommés qui peuvent l'être seront transformés en positionnels.

Il ne s'agit que de peu de modifications dans notre code.

```
1 import functools
2 from inspect import signature
3
4 def memoize(f):
5     cache = {}
6     sig = signature(f)
7     @functools.wraps(f)
8     def decorated(*args, **kwargs):
9         bind = sig.bind(*args, **kwargs)
10        key = repr((bind.args, bind.kwargs))
11        if not key in cache:
12            cache[key] = f(*args, **kwargs)
13        return cache[key]
14    return decorated
```

À l'utilisation, nous obtenons donc :

```
1 >>> @memoize
2 ... def addition(a, b):
3 ...     print('Computing addition of {} and {}'.format(a, b))
4 ...     return a + b
5 ...
6 >>> addition(3, 5)
7 Computing addition of 3 and 5
8 8
9 >>> addition(3, 5)
10 8
11 >>> addition(3, b=5)
12 8
13 >>> addition(b=5, a=3)
14 8
15 >>> addition(5, 3)
16 Computing addition of 5 and 3
```

Nous avons déjà parlé de `functools` à plusieurs reprises dans ce cours. Si vous y avez prêté attention, vous avez remarqué que le décorateur que nous venons d'implémenter ressemble beaucoup à `lru_cache` (à l'exception près que notre version gère les types non-hashables, mais avec une perte de performances et une moins bonne fiabilité).

13.3. Fonctions génériques

Pré-requis : Callables, Décorateurs

Nous allons ici nous intéresser à `singledispatch`, une fonction du module `functools`. Il s'agit d'une implémentation de fonctions génériques en Python, permettant de *dispatcher* l'appel en fonction du type du premier paramètre.

La généricité est un concept qui permet d'appeler une fonction avec des arguments de types variables. C'est le cas par défaut en Python : les variables étant typées dynamiquement, il est possible d'appeler les fonctions quels que soient les types des arguments envoyés.

Mais, de pair avec la généricité vient le concept de spécialisation, qui est plus subtil en Python. Spécialiser une fonction générique, c'est fournir une implémentation différente de la fonction pour certains types de ses paramètres.

En Python, `singledispatch` permet de spécialiser une fonction selon le type de son premier paramètre. Il est ainsi possible de définir plusieurs fois une même fonction, en spécifiant le type sur lequel on souhaite la spécialiser.

`singledispatch` est un décorateur, prenant donc une fonction en paramètre (la fonction qui sera appelée si aucune spécialisation n'est trouvée), et retournant un nouveau *callable*. Ce *callable* possède une méthode `register`, qui s'utilisera comme un décorateur paramétré par le type pour lequel nous voulons spécialiser notre fonction.

Lors de chaque appel au *callable* retourné par `singledispatch`, la fonction à appeler sera déterminée selon le type du premier paramètre. Nos appels devront donc posséder au minimum un argument positionnel.

```

1 >>> @singledispatch
2 ... def print_type(arg):
3 ...     print('Je ne connais pas le type de ce paramètre')
4 ...
5 >>> @print_type.register(int)
6 ... def _(arg): # Le nom doit être différent de print_type
7 ...     print(arg, 'est un entier')
8 ...
9 >>> @print_type.register(str)
10 ... def _(arg):
11 ...     print(arg, 'est une chaîne')
```

VI. Pour quelques exercices de plus

```
12 ...
13 >>> print_type(15)
14 15 est un entier
15 >>> print_type('foo')
16 foo est une chaîne
17 >>> print_type([])
18 Je ne connais pas le type de ce paramètre
```

Pour notre implémentation, je vous propose pour cette fois de réaliser le décorateur à l'aide d'une classe. Cela nous permettra d'avoir facilement un attribut `registry` à disposition.

```
1 import functools
2
3 class singledispatch:
4     def __init__(self, func):
5         self.default = func
6         self.registry = {}
7         functools.update_wrapper(self, func)
8
9     def __call__(self, *args, **kwargs):
10        func = self.registry.get(type(args[0]), self.default)
11        return func(*args, **kwargs)
12
13    def register(self, type_):
14        def decorator(func):
15            self.registry[type_] = func
16            return func
17        return decorator
```

Il faut donc bien comprendre que c'est `__init__` qui sera appelée lors de la décoration de la première fonction, puisque cela revient à instancier un objet `singledispatch`. Cet objet contient alors une méthode `register` pour enregistrer des spécialisations de la fonction. Et enfin, il est callable, *via* sa méthode `__call__`, qui déterminera laquelle des fonctions enregistrées appeler.

Pour aller plus loin, nous pourrions aussi permettre de *dispatcher* en fonction du type de tous les paramètres, ou encore utiliser les annotations pour préciser les types.

13.4. Récursivité terminale

Pré-requis : Callables, Décorateurs

La récursivité terminale est un concept issu du [paradigme fonctionnel](#) [↗](#), permettant d'optimiser les appels récursifs.

VI. Pour quelques exercices de plus

Chaque fois que vous réalisez un appel de fonction, un contexte doit se mettre en place afin de contenir les variables locales à la fonction (dont les paramètres). Il doit être conservé jusqu'à la fin de l'exécution de la fonction.

Ces contextes sont stockés dans une zone mémoire appelée la pile, de taille limitée. Lors d'appels récursifs, les fonctions parentes restent présentes dans la pile, car n'ont pas terminé leur exécution. Donc plus on s'enfonce dans les niveaux de récursivité, plus la pile se remplit, jusqu'à parfois être pleine. Une fois pleine, il n'est alors plus possible d'appeler de nouvelles fonctions, cela est représenté par l'exception `RecursionError` en Python.

Si vous avez déjà tenté d'écrire des fonctions récursives en Python, vous vous êtes rapidement confronté à l'impossibilité de descendre au-delà d'un certain niveau de récursion, à cause de la taille limitée de la pile d'appels.

```
1 >>> def factorial(n):
2 ...     if not n:
3 ...         return 1
4 ...     return n * factorial(n - 1)
5 ...
6 >>> factorial(5)
7 120
8 >>> factorial(1000)
9 Traceback (most recent call last):
10   File "<stdin>", line 1, in <module>
11   File "<stdin>", line 4, in factorial
12   File "<stdin>", line 4, in factorial
13   [...]
14   File "<stdin>", line 4, in factorial
15 RecursionError: maximum recursion depth exceeded in comparison
```

Certains langages, notamment les langages fonctionnels, ont réussi à résoudre ce problème, à l'aide de la récursivité terminale. Il s'agit en fait d'identifier les appels terminaux dans la fonction (c'est-à-dire quand aucune autre opération n'est effectuée après l'appel récursif). Si l'appel est terminal, cela signifie que l'on ne fera plus rien d'autre dans la fonction, et il est alors possible de supprimer son contexte de la pile, et ainsi économiser de l'espace à chaque appel récursif.

Prenons la fonction `factorial` codée plus haut : elle ne peut pas être optimisée par récursivité terminale. En effet, une multiplication est encore effectuée entre l'appel récursif et le `return`.

Prenons maintenant cette seconde implémentation :

```
1 def factorial(n, acc=1):
2     if not n:
3         return acc
4     return factorial(n - 1, acc * n)
```

VI. Pour quelques exercices de plus

Le problème est ici résolu : la multiplication est effectuée avant l'appel puisque dans les arguments. Cette deuxième fonction peut donc être optimisée.

Cependant, [la récursivité terminale n'existe pas en Python](#) [↗](#). Guido von Rossum le dit lui-même. Mais il nous est possible de la simuler, en reproduisant le comportement voulu, avec un décorateur dédié.

En fait, nous allons nous contenter d'ajouter une méthode `call` à nos fonctions. Lorsque nous ferons `function.call(...)`, nous n'appellerons pas réellement la fonction, mais enregistrerons l'appel. Le *wrapper* de notre fonction sera ensuite chargé d'exécuter en boucle tous ces appels enregistrés.

Il faut bien noter que le retour de la méthode `call` ne sera pas le retour de notre fonction. Il s'agira d'un objet temporaire qui servira à réaliser plus tard le réel appel de fonction, dans le *wrapper*.

Nous nous appuyerons sur une classe `tail_rec_exec`, qui n'est autre qu'un *tuple* comportant la fonction à appeler et ses arguments (`args` et `kwargs`).

```
1 class tail_rec_exec(tuple):
2     pass
```

Maintenant nous allons réaliser notre décorateur `tail_rec`, j'ai opté pour une classe :

```
1 import functools
2
3 class tail_rec:
4     def __init__(self, func):
5         self.func = func
6         functools.update_wrapper(self, func)
7
8     def call(self, *args, **kwargs):
9         return tail_rec_exec((self.func, args, kwargs))
10
11    def __call__(self, *args, **kwargs):
12        r = self.func(*args, **kwargs)
13        # Nous exécutons les appels "récursifs" tant que le retour
14        # est de type tail_rec_exec
15        while isinstance(r, tail_rec_exec):
16            func, args, kwargs = r
17            r = func(*args, **kwargs)
18        return r
```

La méthode `__call__` sera celle utilisée lorsque nous appellerons notre fonction, et la méthode `call` utilisée pour temporiser appel.

À l'utilisation, cela donne :

VI. Pour quelques exercices de plus

```
1 @tail_rec
2 def my_sum(values, acc=0):
3     if not values:
4         return acc
5     return my_sum.call(values[1:], acc + values[0])
6
7 @tail_rec
8 def factorial(n, acc=1):
9     if not n:
10        return acc
11    return factorial.call(n - 1, acc * n)
12
13 @tail_rec
14 def even(n):
15     if not n:
16         return True
17     return odd.call(n - 1)
18
19 @tail_rec
20 def odd(n):
21     if not n:
22         return False
23     return even.call(n - 1)
```

```
1 >>> my_sum(range(5000))
2 12497500
3 >>> factorial(5)
4 120
5 >>> factorial(1000)
6 4023872600770937735437024...
7 >>> even(5000)
8 True
9 >>> odd(5000)
10 False
11 >>> even(5001)
12 False
13 >>> odd(5001)
14 True
```

14. Gestionnaires de contexte

14.1. Changement de répertoire

Pré-requis : Gestionnaires de contexte

Dans cet exercice, je vous propose d'implémenter un gestionnaire de contexte pour gérer le répertoire courant. En effet, on voudrait pouvoir changer temporairement de dossier courant, sans effet de bord sur la suite du programme.

Nous voudrions aussi notre gestionnaire de contexte réutilisable et réentrant, à la manière du TP `redirect_stdout`, afin de ne pas perdre le répertoire de départ en cas de contextes imbriqués.

Ainsi, à la construction de l'objet, on enregistrerait le dossier cible, qui serait passé en paramètre. Puis, à l'entrée du contexte, on garderait une trace du dossier courant (`os.getcwd()`) dans une pile de dossiers, avant de se déplacer vers la cible (`os.chdir()`). En sortie, il nous suffirait de nous déplacer à nouveau vers le précédent dossier courant (le dernier élément de la pile).

```
1 import os
2
3 class changedir:
4     def __init__(self, target):
5         self.target = target
6         self.stack = []
7
8     def __enter__(self):
9         current = os.getcwd()
10        self.stack.append(current)
11        os.chdir(self.target)
12
13    def __exit__(self, exc_type, exc_value, traceback):
14        old = self.stack.pop()
15        os.chdir(old)
```

On constate que ce gestionnaire répond bien aux utilisations simples...

```
1 >>> os.getcwd()
2 '/home/entwanne'
3 >>> with changedir('/tmp'):
```

VI. Pour quelques exercices de plus

```
4 ...     os.getcwd()
5 ...
6 '/tmp'
7 >>> os.getcwd()
8 '/home/entwanne'
```

... comme aux complexes.

```
1 >>> cd = changedir('/tmp')
2 >>> os.getcwd()
3 '/home/entwanne'
4 >>> with cd:
5 ...     with cd:
6 ...         os.getcwd()
7 ...
8 '/tmp'
9 >>> os.getcwd()
10 '/home/entwanne'
```

14.2. Transformer un générateur en gestionnaire de contexte

Pré-requis : Décorateurs, Générateurs, Gestionnaires de contexte

Nous avons vu dans ce cours le module `contextlib` et son décorateur `contextmanager` qui permet de créer un gestionnaire de contexte à partir d'une fonction génératrice. Vous l'aurez deviné, le but de cet exercice sera de recoder ce décorateur, ou au moins une version basique.

Le code du décorateur en lui-même sera assez court. Celui-ci se chargera de *wrapper* la fonction, afin de retourner un gestionnaire de contexte plutôt qu'un générateur lors des appels à celle-ci.

```
1 import functools
2
3 def contextmanager(gen_func):
4     @functools.wraps(gen_func)
5     def wrapper(*args, **kwargs):
6         gen = gen_func(*args, **kwargs)
7         return GeneratorContextManager(gen)
8     return wrapper
```

Le type `GeneratorContextManager`, qu'il nous reste à définir, est un gestionnaire de contexte. Il fera appel au générateur qui lui est passé en paramètre pour ses méthodes `__enter__` et `__exit__`.

Nous pouvons en imaginer une première version sous la forme suivante :

VI. Pour quelques exercices de plus

```
1 class GeneratorContextManager:
2     def __init__(self, generator):
3         self.generator = generator
4
5     def __enter__(self):
6         return next(self.generator)
7
8     def __exit__(self, exc_type, exc_value, traceback):
9         next(self.generator)
```

Le code est ici plutôt simple : `__enter__` provoque une première itération du générateur, afin d'aller jusqu'au `yield`. Et `__exit__` en provoque une seconde, pour exécuter la code qui suit ce `yield`.

Mais nous pouvons déjà constater un premier problème.

```
1 >>> @contextmanager
2 ... def context():
3 ...     print('before')
4 ...     yield 'foo'
5 ...     print('after')
6 ...
7 >>> with context() as val:
8 ...     print('during', val)
9 ...
10 before
11 during foo
12 after
13 Traceback (most recent call last):
14   File "<stdin>", line 2, in <module>
15   File "<stdin>", line 9, in __exit__
16 StopIteration
```

Tout se déroule bien... jusqu'à la fermeture. En effet, `__exit__` fait un `next` sur le générateur, mais ce dernier ne contient qu'un `yield`, il est donc normal qu'il lève une exception `StopIteration`. Il nous faudra alors l'attraper afin de la masquer.

Et d'ailleurs, nous voulons empêcher que le générateur produise plus d'une valeur. Pour cela, nous lèverons notre propre exception si aucune `StopIteration` n'est attrapée.

Nous réécrivons alors comme suit notre méthode `__exit__`.

```
1 def __exit__(self, exc_type, exc_value, traceback):
2     try:
3         next(self.generator)
4     except StopIteration:
```

VI. Pour quelques exercices de plus

```
5         return
6     else:
7         raise RuntimeError("generator didn't stop")
```

Et notre gestionnaire de contexte se comporte maintenant correctement avec l'exemple précédent.

```
1 >>> with context() as val:
2 ...     print('during', val)
3 ...
4 before
5 during foo
6 after
```

Mais, de la même manière que nous nous assurons que notre générateur ne produise pas plus d'une valeur, il faudrait aussi vérifier qu'il en produit au moins une.

Nous allons donc ajouter un `try/except` à notre méthode `__enter__` et lever une `RuntimeError` en cas de `StopIteration` lors du `next`.

```
1 def __enter__(self):
2     try:
3         return next(self.generator)
4     except StopIteration:
5         raise RuntimeError("generator didn't yield")
```

Notre gestionnaire est maintenant plus complet, mais il lui manque une fonctionnalité cruciale : il ne gère pas les exceptions levées dans le bloc `with`.

Avec `@contextmanager`, il est normalement possible d'écrire le gestionnaire suivant, qui attraperait toutes les `TypeError`.

```
1 @contextmanager
2 def catch_typeerror():
3     try:
4         yield
5     except TypeError:
6         pass
```

Avec notre décorateur, l'exception n'est jamais attrapée.

VI. Pour quelques exercices de plus

```
1 >>> with catch_typerror():
2 ...     print(1 + '2')
3 ...
4 Traceback (most recent call last):
5   File "<stdin>", line 2, in <module>
6 TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

En effet, en cas d'exception attrapée, la méthode `__exit__` est censée retourner `True`. Mais l'exception étant levée dans le contexte, et gérée par le gestionnaire, elle n'atteint jamais le générateur.

Comment alors savoir si l'exception aurait bien été attrapée par le générateur ? Il suffit de la lui faire lever avec sa méthode `throw`, et de regarder si cette exception remonte jusqu'au gestionnaire de contexte. On distingue alors 4 cas :

- Le générateur lève une `StopIteration`, et s'est donc terminé normalement, `__exit__` retourne `True` ;
- Le générateur lève l'exception qui lui a été fournie lors du `throw`, il ne l'a donc pas attrapée, `__exit__` retourne `False` ;
- Le générateur lève une autre exception, on la laisse remonter ;
- Le générateur ne lève aucune exception, il ne s'est donc pas terminé, on lève une `RuntimeError`.

Ces cas n'interviennent bien sûr que si une exception s'est produite dans le bloc `with`, et donc que le paramètre `exc_type` fourni à `__exit__` ne vaut pas `None`. Dans le cas où il vaut `None`, nous gardons le comportement actuel de notre méthode.

Nous complétons donc le code de notre méthode `__exit__` pour ajouter cette gestion d'erreurs.

```
1 def __exit__(self, exc_type, exc_value, traceback):
2     if exc_type is None:
3         try:
4             next(self.generator)
5         except StopIteration:
6             return
7         else:
8             raise RuntimeError("generator didn't stop")
9     try:
10        self.generator.throw(exc_type, exc_value, traceback)
11    except StopIteration:
12        return True
13    except exc_type:
14        return False
15    else:
16        raise RuntimeError("generator didn't stop after throw")
```

À l'utilisation, on constate bien qu'il permet d'attraper certaines exceptions et d'en laisser remonter d'autres.

VI. Pour quelques exercices de plus

```
1 >>> @contextmanager
2 ... def catch_typeerror():
3 ...     try:
4 ...         yield
5 ...     except TypeError:
6 ...         pass
7 ...
8 >>> with catch_typeerror():
9 ...     print(1 + 'a')
10 ...
11 >>> with catch_typeerror():
12 ...     raise ValueError
13 ...
14 Traceback (most recent call last):
15   File "<stdin>", line 2, in <module>
16 ValueError
```

Nous approchons de la fin de nos déboires, mais il y a encore deux cas que nous ne gérons pas.

- Que faire si une `StopIteration` survient dans le bloc `with` ?
- Que faire si le générateur lève une `exc_type`, qui n'est pas la même exception que celle survenue dans le bloc `with` ?

Dans le premier cas, l'exception est automatiquement attrapée sans que nous ne l'ayons demandé.

```
1 >>> with catch_typeerror():
2 ...     raise StopIteration
3 ...
```

En effet, dans le code d'`__exit__`, nous attrapons toutes les `StopIteration` sans vérifier où elles se sont produites. Comment faire cela ? En plus du paramètre `exc_type`, la méthode reçoit aussi `exc_value` qui est l'instance de l'exception levée dans le `with`.

Nous pouvons alors vérifier que la `StopIteration` attrapée est la même instance que l'exception survenue dans le `with`. Si tel est le cas, `__exit__` doit retourner `False` (pour laisser remonter l'exception). Dans le cas contraire elle continuera de retourner `True`.

Quant au fait que le générateur puisse lever une `exc_type` différente de celle du bloc `with`, il s'agit en fait du même problème.

On peut l'illustrer avec l'exemple suivant.

```
1 >>> @contextmanager
2 ... def catch_typeerror():
3 ...     try:
4 ...         yield
```

VI. Pour quelques exercices de plus

```
5 ...     except TypeError:
6 ...         print('error n°' + 4)
7 ...
8 >>> with catch_typeerror():
9 ...     raise TypeError
10 ...
11 Traceback (most recent call last):
12   File "<stdin>", line 2, in <module>
13 TypeError
```

L'exception qui nous remonte est celle du bloc `with`, normalement attrapée par le générateur. À l'inverse, l'exception qui survient dans le générateur (au niveau du `print`) est totalement ignorée.

Le problème est que `__exit__` retourne `False` dans tous les cas où une `exc_type` survient. Elle devrait d'abord s'assurer qu'il s'agit de la même instance (en comparant l'exception avec `exc_value`). Si les instances sont différentes, `__exit__` devrait relayer l'exception (avec `raise`) plutôt que de retourner `False`.

Nous ajoutons donc ces améliorations à `__exit__` pour obtenir notre code final.

```
1 def __exit__(self, exc_type, exc_value, traceback):
2     if exc_type is None:
3         try:
4             next(self.generator)
5         except StopIteration:
6             return
7         else:
8             raise RuntimeError("generator didn't stop")
9     try:
10        self.generator.throw(exc_type, exc_value, traceback)
11    except StopIteration as e:
12        return e is not exc_value
13    except exc_type as e:
14        if e is exc_value:
15            return False
16        raise
17    else:
18        raise RuntimeError("generator didn't stop after throw")
```

Code final ? Oui, dans le sens où nous en resterons là pour notre exercice. Mais d'autres cas sont normalement gérés par `contextmanager`, notamment celui où `exc_value` vaudrait `None` ou le cas d'exceptions qui en auraient causé d'autres. Pour une version vraiment complète, je vous invite à consulter [les sources de la classe `_GeneratorContextManager` du module `contextlib`](#) [↗](#).

14.3. Suppression d'erreurs

Pré-requis : Gestionnaires de contexte

Vous pensiez en avoir terminé avec `contextlib` ? Que nenni ! Ce module présente beaucoup de gestionnaires de contexte assez simples à réimplémenter, il est donc idéal pour les exercices.

Intéressons-nous maintenant à `suppress`, qui permet d'ignorer des exceptions, comme pourrait le faire un `try/except`. En effet, les deux exemples de codes qui suivent sont équivalents.

```
1 from contextlib import suppress
2
3 with suppress(TypeError):
4     print(1 + '2')
```

```
1 try:
2     print(1 + '2')
3 except TypeError:
4     pass
```

Nous l'avons vu, les erreurs qui surviennent dans un contexte sont transmises à la méthode `__exit__` du gestionnaire, qui peut choisir d'annuler l'exception en retournant `True`. Tout ce que nous avons à faire est donc de vérifier si une exception est survenue et si cette dernière est du bon type, puis retourner `True` si ces deux conditions sont respectées.

Pour vérifier que l'exception est du bon type, il nous suffira de faire appel à `issubclass` et tester que le paramètre `exc_type` est un sous-type de celui passé à la construction du gestionnaire.

Le code de notre gestionnaire de contexte se présente alors comme suit.

```
1 class suppress:
2     def __init__(self, exc_type):
3         self.exc_type = exc_type
4
5     def __enter__(self):
6         pass
7
8     def __exit__(self, exc_type, exc_value, traceback):
9         return exc_type is not None and issubclass(exc_type,
10            self.exc_type)
```

Je vous laisse expérimenter et vérifier que notre classe répond bien aux attentes. Une petite subtilité tout de même : `suppress` peut normalement s'utiliser en spécifiant plusieurs types d'exception à annuler.

VI. Pour quelques exercices de plus

```
1 with suppress(TypeError, ValueError, IndexError):  
2     print(1 + '2')
```

Étant donné qu'`issubclass` peut prendre un *tuple* en second paramètre, la modification du code de notre gestionnaire de contexte sera très rapide.

```
1 class suppress:  
2     def __init__(self, *exc_types):  
3         self.exc_types = exc_types  
4  
5     def __enter__(self):  
6         pass  
7  
8     def __exit__(self, exc_type, exc_value, traceback):  
9         return exc_type is not None and issubclass(exc_type,  
             self.exc_types)
```

15. Accesseurs et descripteurs

15.1. Propriétés

Pré-requis : Décorateurs, Attributs, Descripteurs

Durant le chapitre sur les descripteurs, nous avons utilisé `my_property`, une copie minimaliste de la classe `property`. Je vous propose ici d'en terminer l'implémentation, afin de rendre ces deux classes similaires.

Pour rappel, `property` est une classe de descripteurs, utilisable en tant que décorateur sur les méthodes à transformer en propriétés. Les propriétés étant des descripteurs faisant appel à des fonctions particulières pour l'accès en lecture/écriture ou la suppression de l'attribut. En plus des méthodes `__get__`, `__set__` et `__delete__` des descripteurs, `property` possède aussi des méthodes/décorateurs `getter`, `setter` et `deleter` pour redéfinir ces fonctions.

Notre implémentation utilisée dans le chapitre sur les accesseurs était la suivante :

```
1 class my_property:
2     def __init__(self, fget, fset, fdel):
3         self.fget = fget
4         self.fset = fset
5         self.fdel = fdel
6
7     def __get__(self, instance, owner):
8         return self.fget(instance)
9
10    def __set__(self, instance, value):
11        return self.fset(instance, value)
12
13    def __delete__(self, instance):
14        return self.fdel(instance)
```

Nous implémentons bien les 3 méthodes propres aux descripteurs, mais il faut pour chaque propriété que ces 3 fonctions soient définies, et notre classe n'est pas utilisable comme décorateur. Il manque aussi la redéfinition des fonctions.

Un autre point qui n'a pas été abordé est celui de la documentation. En plus des `fget`, `fset` et `fdel`, la propriété peut-être initialisée avec un paramètre `doc`, qui sera la documentation de l'attribut. En l'absence de `doc`, l'attribut prendra pour documentation celle de la fonction `fget`, si existante.

VI. Pour quelques exercices de plus

Ces 4 paramètres sont tous facultatifs, nous les initialiserons alors à `None`. Nous pouvons alors écrire le nouvel initialiseur de notre classe `my_property`.

```
1     def __init__(self, fget=None, fset=None, fdel=None, doc=None):
2         if doc is None and fget is not None:
3             doc = getattr(fget, '__doc__', None)
4         self.fget, self.fset, self.fdel = fget, fset, fdel
5         self.__doc__ = doc
```

On récupère ainsi la documentation de `fget` si aucune documentation n'a été fournie à la construction (`doc is None`), et si `fget` est présente (`fget is not None`). En utilisant `getattr`, on permet à `fget` de ne pas avoir de documentation, auquel cas `doc` vaudra toujours `None`.

Les méthodes `__get__`, `__set__` et `__delete__` seront un peu plus complexes que précédemment. Il nous faudra maintenant tester la présence des fonctions cibles (`fget`, `fset` et `fdel`), et lever une exception `AttributeError` le cas échéant. Cette exception indiquera que l'attribut n'est pas lisible, redéfinissable, ou supprimable.

Nous ferons aussi en sorte que `__get__` appelée sans instance (`instance` valant `None`) retourne la propriété elle-même.

```
1     def __get__(self, instance, owner):
2         'Return an attribute of instance, which is of type owner.'
3         if instance is None:
4             return self
5         if self.fget is None:
6             raise AttributeError('unreadable attribute')
7         return self.fget(instance)
8
9     def __set__(self, instance, value):
10        'Set an attribute of instance to value.'
11        if self.fset is None:
12            raise AttributeError("can't set attribute")
13        return self.fset(instance, value)
14
15    def __delete__(self, instance):
16        'Delete an attribute of instance.'
17        if self.fdel is None:
18            raise AttributeError("can't delete attribute")
19        return self.fdel(instance)
```

Enfin, les méthodes `getter`, `setter` et `deleter` copieront celles de `property`. Plutôt que de modifier la propriété avec la fonction reçue en paramètre, elles retourneront une nouvelle propriété. Aucun changement ne sera effectué si le paramètre vaut `None`.

VI. Pour quelques exercices de plus

```
1 def getter(self, fget):
2     'Descriptor to change the getter on a property.'
3     if fget is None:
4         return self
5     return type(self)(fget, self.fset, self.fdel, self.__doc__)
6
7 def setter(self, fset):
8     'Descriptor to change the setter on a property.'
9     if fset is None:
10        return self
11    return type(self)(self.fget, fset, self.fdel, self.__doc__)
12
13 def deleter(self, fdel):
14    'Descriptor to change the deleter on a property.'
15    if fdel is None:
16        return self
17    return type(self)(self.fget, self.fset, fdel, self.__doc__)
```

Où nous utilisons `type(self)(...)` afin de faire appel au constructeur de notre propriété. Ce qui restera valable avec une nouvelle classe qui hériterait de `my_property`.

Une fois toutes ces méthodes réunies dans notre classe `my_property`, à laquelle on ajouterait encore une petite dose de documentation, on retrouve un équivalent complet de `property`.

```
1 class my_property:
2     '''
3     my_property(fget=None, fset=None, fdel=None, doc=None) -> property attribute
4
5     fget is a function to be used for getting an attribute value, and likewise
6     fset is a function for setting, and fdel a function for del'ing, an
7     attribute. Typical use is to define a managed attribute x:
8
9     class C(object):
10        def getx(self): return self._x
11        def setx(self, value): self._x = value
12        def delx(self): del self._x
13        x = my_property(getx, setx, delx, "I'm the 'x' property.")
14
15    Decorators make defining new properties or modifying existing ones easy:
16
17    class C(object):
18        @my_property
19        def x(self):
20            "I am the 'x' property."
21            return self._x
22        @x.setter
23        def x(self, value):
```

VI. Pour quelques exercices de plus

```
24         self._x = value
25     @x.deleter
26     def x(self):
27         del self._x
28     '''
29
30     def __init__(self, fget=None, fset=None, fdel=None, doc=None):
31         if doc is None and fget is not None:
32             doc = getattr(fget, '__doc__', None)
33         self.fget, self.fset, self.fdel = fget, fset, fdel
34         self.__doc__ = doc
35
36     def __get__(self, instance, owner):
37         'Return an attribute of instance, which is of type owner.'
38         if instance is None:
39             return self
40         if self.fget is None:
41             raise AttributeError('unreadable attribute')
42         return self.fget(instance)
43
44     def __set__(self, instance, value):
45         'Set an attribute of instance to value.'
46         if self.fset is None:
47             raise AttributeError("can't set attribute")
48         return self.fset(instance, value)
49
50     def __delete__(self, instance):
51         'Delete an attribute of instance.'
52         if self.fdel is None:
53             raise AttributeError("can't delete attribute")
54         return self.fdel(instance)
55
56     def getter(self, fget):
57         'Descriptor to change the getter on a property.'
58         if fget is None:
59             return self
60         return type(self)(fget, self.fset, self.fdel, self.__doc__)
61
62     def setter(self, fset):
63         'Descriptor to change the setter on a property.'
64         if fset is None:
65             return self
66         return type(self)(self.fget, fset, self.fdel, self.__doc__)
67
68     def deleter(self, fdel):
69         'Descriptor to change the deleter on a property.'
70         if fdel is None:
71             return self
72         return type(self)(self.fget, self.fset, fdel, self.__doc__)
```

VI. Pour quelques exercices de plus

Que nous pouvons alors utiliser dans notre classe `Temperature` par exemple.

```
1 class Temperature:
2     def __init__(self):
3         self.value = 0
4
5     @my_property
6     def celsius(self): # le nom de la méthode devient le nom de la
7         # propriété
8         return self.value
9     @celsius.setter
10    def celsius(self, value): # le setter doit porter le même nom
11        self.value = value
12
13    @my_property
14    def fahrenheit(self):
15        return self.value * 1.8 + 32
16    @fahrenheit.setter
17    def fahrenheit(self, value):
18        self.value = (value - 32) / 1.8
```

```
1 >>> t = Temperature()
2 >>> t.celsius
3 0
4 >>> t.fahrenheit
5 32.0
6 >>> t.celsius = 100
7 >>> t.fahrenheit
8 212.0
```

16. Métaclasses

16.1. Évaluation paresseuse

Pré-requis : Attributs, Métaclasses

Dans ce TP, nous nous intéresserons à l'évaluation paresseuse (*lazy evaluation*).

16.1.1. L'évaluation paresseuse, c'est quoi ?

Lorsque vous entrez une expression Python dans votre interpréteur et que celui-ci vous retourne une valeur, on dit que cette expression est évaluée. Évaluer une expression correspond donc à en calculer le résultat.

L'évaluation paresseuse se différencie de l'évaluation standard par rapport au moment où le calcul a lieu. Lors d'une évaluation traditionnelle, le résultat est tout de suite retourné, et peut être manipulé. Dans le cas d'une évaluation paresseuse, celui-ci n'est calculé que lorsqu'il est réellement nécessaire (quand on commence à manipuler l'objet), d'où le terme de paresseux.

En Python par exemple, nous avons étudié plus tôt le concept de générateurs, ils correspondent à de l'évaluation paresseuse : ils ne sont pas évalués avant que l'on ne commence à itérer dessus.

16.1.2. Objectif du TP

Ici, nous voulons réaliser un appel paresseux à une fonction. C'est-à-dire embarquer la fonction à appeler et ses paramètres, mais ne réaliser l'appel qu'au moment où nous avons besoin du résultat.

Par exemple :

```
1 >>> def square(x):
2 ...     return x ** 2
3 ...
4 >>> a = square(3)
5 >>> b = square(4)
6 >>> c = square(5)
7 >>> a + b == c
8 True
```

Nous n'avons réellement besoin des valeurs `a`, `b` et `c` qu'en ligne 5.

VI. Pour quelques exercices de plus

Si nous ne voulons pas calculer tout de suite le résultat, il faudra tout de même que notre fonction `square` retourne quelque chose. Et dans notre exemple, l'objet retourné devra posséder les méthodes `__add__` et `__eq__`. Méthodes qui se chargeront d'effectuer le calcul du carré.

Ce ne sont ici que deux opérateurs, mais il en existe beaucoup d'autres, dont l'énumération serait inutile et fastidieuse, et il va nous falloir tous les gérer.

16.1.3. Opérateurs et méthodes spéciales

Le problème des opérateur en Python, c'est que les appels aux méthodes spéciales sont optimisés et ne passent pas par `__getattr__`.

```
1 >>> class A:
2 ...     def __add__(self, rhs):
3 ...         return 0
4 ...     def __getattr__(self, name):
5 ...         print('getattr')
6 ...         return super().__getattr__(name)
7 ...
8 >>> A() + A()
9 0
10 >>> import operator
11 >>> operator.add(A(), A())
12 0
13 >>> A().__add__(A())
14 getattr
15 0
```

Il va donc nous falloir intégrer toutes les méthodes spéciales à nos objets, et les métaclasses nous seront alors d'une grande aide pour toutes les générer.

Une liste de méthodes spéciales nous est fournie dans la documentation Python : <https://docs.python.org/3/reference/datamodel.html#specialnames> ↗

```
— __new__, __init__, __del__
— __repr__, __str__, __bytes__, __format__
— __lt__, __le__, __eq__, __ne__, __gt__, __ge__
— __hash__, __bool__
— __getattr__, __getitem__, __setattr__, __delattr__, __dir__
— __get__, __set__, __delete__
— __instancecheck__, __subclasscheck__
— __call__
— __len__, __length_hint__, __getitem__, __missing__, __setitem__, __deli
tem__, __iter__, __reversed__, __contains__
— __add__, __sub__, __mul__, __matmul__, __truediv__, __floordiv__, __mod__,
__divmod__, __pow__, __lshift__, __rshift__, __and__, __xor__, __or__
— __radd__, __rsub__, __rmul__, __rmatmul__, __rtruediv__, __rfloordiv__, __rmod__,
__rdivmod__, __rpow__, __rlshift__, __rrshift__, __rand__, __rxor__, __ror__
```

VI. Pour quelques exercices de plus

```
— __iadd__, __isub__, __imul__, __imatmul__, __itruediv__, __ifloordiv__, __imod__,  
  __ipow__, __ilshift__, __irshift__, __iand__, __ixor__, __ior__  
— __neg__, __pos__, __abs__, __invert__, __complex__, __int__, __float__, __round__,  
  __index__  
— __enter__, __exit__  
— __await__, __aiter__, __anext__, __aenter__, __aexit__
```

Mais celle-ci n'est pas complète, `__next__` n'y figure par exemple pas. Je n'ai pas trouvé de liste exhaustive, et c'est donc celle-ci que nous utiliserons. Nous omettrons cependant la première ligne (constructeur, initialiseur et destructeur), car les objets que nous recevrons seront déjà construits.

Il nous faut aussi différencier les opérateurs des autres méthodes spéciales. En effet, les méthodes spéciales associées aux opérateurs peuvent dans certains cas retourner `NotImplemented` et laisser l'opérateur décider d'un comportement (comme appeler une méthode de l'autre opérande dans le cas d'un opérateur binaire). Pour nous faciliter la tâche et ne pas avoir à gérer nous-même ces comportements, nous ferons donc appel à l'opérateur plutôt qu'à la méthode spéciale. Le module `operator` nous permettra facilement de savoir si la méthode spéciale est un opérateur, et donc d'agir en conséquence (en vérifiant que la méthode est présente dans `operator.__dict__` par exemple).

La solution que je propose est la suivante.

```
1 import operator  
2  
3 class LazyMeta(type):  
4     # Référencement de toutes les méthodes spéciales, ou presque  
5     specials = [  
6         '__repr__', '__str__', '__bytes__', '__format__',  
7         '__lt__', '__le__', '__eq__', '__ne__', '__gt__', '__ge__',  
8         '__hash__', '__bool__',  
9         '__getattr__', '__getattribute__', '__setattr__',  
10        '__delattr__', '__dir__',  
11        '__get__', '__set__', '__delete__',  
12        '__instancecheck__', '__subclasscheck__',  
13        '__call__',  
14        '__len__', '__length_hint__', '__getitem__', '__missing__',  
15        '__setitem__', '__delitem__',  
16        '__iter__', '__reversed__', '__contains__',  
17        '__add__', '__sub__', '__mul__', '__matmul__',  
18        '__truediv__', '__floordiv__', '__mod__',  
19        '__divmod__', '__pow__', '__lshift__', '__rshift__',  
        '__and__', '__xor__', '__or__',  
        '__radd__', '__rsub__', '__rmul__', '__rmatmul__',  
        '__rtruediv__', '__rfloordiv__', '__rmod__',  
        '__rdivmod__', '__rpow__', '__rlshift__', '__rrshift__',  
        '__rand__', '__rxor__', '__ror__',  
        '__iadd__', '__isub__', '__imul__', '__imatmul__',  
        '__itruediv__', '__ifloordiv__', '__imod__',
```

VI. Pour quelques exercices de plus

```
20     '__ipow__', '__ilshift__', '__irshift__', '__iand__',
21         '__ixor__', '__ior__',
22     '__neg__', '__pos__', '__abs__', '__invert__',
23         '__complex__', '__int__', '__float__',
24     '__round__', '__index__',
25     '__enter__', '__exit__',
26     '__await__', '__aiter__', '__anext__', '__aenter__',
27         '__aexit__',
28     '__next__'
29 ]
30
31 def get_meth(methname):
32     "Fonction utilisée pour créer une méthode dynamiquement"
33     def meth(self, *args, **kwargs):
34         # On tente d'accéder à l'objet évalué (value)
35         try:
36             value = object.__getattr__(self, 'value')
37         # S'il n'existe pas, il nous faut alors le calculer
38         # puis le stocker
39         except AttributeError:
40             value = object.__getattr__(self, 'expr')()
41             object.__setattr__(self, 'value', value)
42         # Appel à l'opérateur si la méthode est un opérateur
43         if methname in operator.__dict__:
44             return getattr(operator, methname)(value, *args,
45                 **kwargs)
46         # Sinon, appel à la méthode de l'objet
47         return getattr(value, methname)(*args, **kwargs)
48     return meth
49
50 @classmethod
51 def __prepare__(cls, name, bases):
52     # On prépare la classe en lui ajoutant toutes les méthodes
53     # référencées
54     methods = {}
55     for methname in cls.specials:
56         methods[methname] = cls.get_meth(methname)
57     return methods
58
59 ##### Le type Lazy est celui que nous utiliserons pour l'évaluation
60 # paresseuse
61 class Lazy(metaclass=LazyMeta):
62     def __init__(self, expr):
63         # Il possède une expression (un callable qui retournera
64         # l'objet évalué)
65         # Les autres méthodes de Lazy sont ajoutées par la
66         # métaclass
67         object.__setattr__(self, 'expr', expr)
```

Nous sommes obligés d'utiliser `object.__getattr__` et `object.__setattr__` pour

VI. Pour quelques exercices de plus

accéder aux attributs, afin de ne pas interférer avec les méthodes redéfinies dans la classe courante.

À l'utilisation, cela donne :

```
1 >>> def _eval():
2 ...     print('evaluated')
3 ...     return 4
4 ...
5 >>> l = Lazy(_eval)
6 >>> l + 5
7 evaluated
8 9
9 >>> l + 5
10 9
11 >>> l = Lazy(lambda: [])
12 >>> l
13 []
14 >>> l.append(5)
15 >>> l
16 [5]
17 >>> type(l)
18 <class '__main__.Lazy'>
19 >>> class A:
20 ...     pass
21 ...
22 >>> l = Lazy(lambda: A())
23 >>> l
24 <__main__.A object at 0x7f7e3d7376a0>
25 >>> type(l)
26 <class '__main__.Lazy'>
27 >>> l.x = 0
28 >>> l.x
29 0
30 >>> l.y # AttributeError
31 >>> l + 1 # TypeError
32 >>> abs(l) # TypeError
```

Aussi, pour en revenir au TP sur la récursivité terminale, il nous suffirait de faire retourner à notre fonction un objet de type `Lazy` pour ne plus avoir à différencier `call` et `__call__`. Les appels ne seraient alors exécutés, itérativement, qu'à l'utilisation du retour (quand on chercherait à itérer dessus, à l'afficher, ou autre). Il n'y aurait ainsi plus besoin de se soucier de savoir si nous sommes dans un appel récursif ou dans le premier appel.

16.2. Types immutables

Pré-requis : Descripteurs, Métaclasses

VI. Pour quelques exercices de plus

Je vous avais promis une autre approche pour créer des types immutables, la voici ! Pour rappel, notre première version créait des classes héritant de `tuple/namedtuple`. Les attributs, ainsi stockés dans une structure immuable, n'étaient alors pas réassignables. Mais nous étions gênés par la présence de trop nombreuses méthodes qui polluaient nos objets.

La seconde version que je vous propose ici est plus étonnante, car elle ne repose pas sur un type immuable, mais sur `super`. Vous pourriez vous demander ce que `super` vient faire dans cette histoire. Pour vous l'expliquer, intéressons-nous au fonctionnement des objets de ce type.

16.2.1. Objets `super`

Dans sa forme complète, un objet `super` s'initialise avec une classe et une instance (plus ou moins directe) de cette classe.

```
1 >>> obj = super(tuple, (1, 2, 3))
```

L'objet ainsi créé se comportera comme une instance de la classe suivante dans le *MRO*. Dans l'exemple précédent, `obj` se comportera comme une instance d'`object` (classe parente de `tuple`).

Les deux arguments fournis à `super` se retrouvent stockés dans l'objet.

```
1 >>> obj.__thisclass__
2 <class 'tuple'>
3 >>> obj.__self__
4 (1, 2, 3)
```

On trouve aussi un 3ème attribut, `__self_class__`, le type de `__self__`. Ce type pouvant être différent de `__thisclass__` si l'instance passée à `super` est l'instance d'une de ses classes filles.

```
1 >>> class T(tuple): pass
2 ...
3 >>> obj = super(tuple, T((0, 1, 2)))
4 >>> obj.__thisclass__
5 <class 'tuple'>
6 >>> obj.__self_class__
7 <class '__main__.T'>
```

L'intérêt est que ces 3 seuls attributs ne sont pas redéfinissables.

VI. Pour quelques exercices de plus

```
1 >>> obj.__self__ = (2, 3, 4)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 AttributeError: readonly attribute
```

Aussi, les objets `super` disposent de peu de méthodes, contrairement aux *tuples*.

```
1 >>> dir(obj)
2 ['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
3  '__format__', '__ge__',
4  '__get__', '__getattr__', '__gt__', '__hash__', '__init__',
5  '__le__', '__lt__',
6  '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
7  '__self__',
8  '__self_class__', '__setattr__', '__sizeof__', '__str__',
9  '__subclasshook__', '__thisclass__']
```

Sur ces 26 éléments, seuls 7 sont définis dans `super`, les autres appartiennent à `object`.

```
1 >>> [meth for meth in dir(obj) if getattr(getattr(super, meth,
2  None), '__objclass__', None) is super]
3 ['__get__', '__getattr__', '__init__', '__repr__', '__self__',
4  '__self_class__', '__thisclass__']
```

Ainsi, notre nouvelle version de types immutables se basera sur `super`, en incluant un objet `tuple` qui stockera les données.

16.2.2. ImmutableMeta

Mais revenons-en à nos métaclasse. Ne nous appuyant ici pas sur `namedtuple`, il va falloir gérer nous-même la liaison entre les noms d'attributs et les éléments du *tuple*.

Pour cela, depuis le champ `__fields__` définit dans la classe `immutable`, on générera des propriétés pour chaque nom de champ, en l'associant à un index du *tuple*.

On utilisera pour cela une méthode `field_property`, qui sera définie comme méthode statique dans la métaclasse, et recevra un unique paramètre : l'index dans le *tuple* du champ à indexer.

```
1 @staticmethod
2 def field_property(i):
3     def get_field(self):
```

VI. Pour quelques exercices de plus

```
4         return self.__self__[i] # On accède au tuple pointé par
           __self__ et à l'élément d'index i
5     return property(get_field) # On enveloppe la fonction get_field
           dans une property
```

Il nous faudra aussi gérer l'initialiseur de notre classe, associant les arguments positionnels et nommés aux champs de nos objets.

Vient alors la méthode `__new__` de la métaclasse, qui se déroulera en plusieurs temps :

- D'abord, récupérer le champ `__fields__` du `dict` de la classe, et instancier un `field_property` pour chaque, ajouté au `dict` ;
- Ensuite, générer une signature de la méthode `__init__`, utilisée pour vérifier que les arguments correspondent bien aux champs lors de la construction ;
- Enfin, définir dans le `dict` un attribut `__slots__` à `()` pour éviter de pouvoir ajouter d'autres attributs aux instances de nos classes ;

Notre classe `ImmutableMeta` complète est donc la suivante.

```
1 class ImmutableMeta(type):
2     def __new__(cls, name, bases, dict):
3         fields = dict.pop('__fields__', ())
4         # Création des propriétés associées aux champs
5         dict.update({field: cls.field_property(i) for i, field in
6                       enumerate(fields)})
7         # Création d'une signature artificielle composée des noms
           de champs
8         dict['__signature__'] = inspect.Signature(
           inspect.Parameter(field,
9                             inspect.Parameter.POSITIONAL_OR_KEYWORD) for field in
           fields)
10        dict['__slots__'] = ()
11        return super().__new__(cls, name, bases, dict)
12
13    @staticmethod
14    def field_property(i):
15        def get_field(self):
16            return self.__self__[i]
17        return property(get_field)
```

Et pour l'utiliser, nous créons une classe `Immutable`, héritant de `super` et définissant une méthode `__init__` pour initialiser nos objets immutables. Cette méthode devra vérifier les arguments conformément à la signature, puis créer un *tuple* des attributs, à passer à l'initialiseur parent (celui de `super`).

VI. Pour quelques exercices de plus

```
1 class Immutable(super, metaclass=ImmutableMeta):
2     def __init__(self, *args, **kwargs):
3         t = self.__signature__.bind(*args, **kwargs).args
4         # Rappel : l'initialiseur prend un type et une instance de
           ce type
5         super().__init__(tuple, t)
```

Il nous suffit alors d'hériter d'Immutable, et de définir un champ `__fields__` pour avoir un nouveau type d'objets immutables.

```
1 class Point(Immutable):
2     __fields__ = ('x', 'y')
3
4     def distance(self):
5         return (self.x**2 + self.y**2)**0.5
```

```
1 >>> p = Point(3, 4)
2 >>> p.x
3 3
4 >>> p.y
5 4
6 >>> p.distance()
7 5.0
8 >>> p.x = 0
9 Traceback (most recent call last):
10   File "<stdin>", line 1, in <module>
11 AttributeError: can't set attribute
12 >>> p.z = 0
13 Traceback (most recent call last):
14   File "<stdin>", line 1, in <module>
15 AttributeError: 'Point' object has no attribute 'z'
```

Grâce à `super`, nous n'avons plus ici accès à `[]` ou `__getitem__` sur nos objets.

```
1 >>> p[0]
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 TypeError: 'Point' object does not support indexing
5 >>> p.__getitem__(0)
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 AttributeError: 'Point' object has no attribute '__getitem__'
```


16.2.3. Améliorations

Notre classe est terminée, mais nous pouvons encore lui apporter quelques améliorations.

Premièrement, redéfinir la méthode `__repr__` de la classe `Immutable`. Elle pointe ici sur celle de `super`, qui ne correspond pas vraiment à notre objet.

On peut choisir de la faire pointer sur `object.__repr__` :

```
1 >>> Immutable.__repr__ = object.__repr__
2 >>> Point(1, 2)
3 <__main__.Point object at 0x7f012d454448>
```

Ou encore donner un résultat similaire aux *named tuples*, en écrivant le nom de la classe suivi de ses attributs entre paramètres.

```
1 def __repr__(self):
2     return '{}{}'.format(type(self).__name__, repr(self.__self__))
```

```
1 >>> Point(1, 2)
2 Point(1, 2)
3 >>> Point(x=1, y=2)
4 Point(1, 2)
```

Bonus Python 3.6 :

```
1 def __repr__(self):
2     return f'{type(self).__name__}{self.__self__!r}'
```

Seconde amélioration, masquer les méthodes inutiles. Pour cela, on on peut définir une méthode `__dir__` dans `Immutable`. Cette méthode spéciale est celle appelée par la fonction `dir`. Nous pouvons alors en filtrer les méthodes pour supprimer celles définies par `super` (comme nous l'avons fait plus tôt en regardant l'attribut `__objclass__` des méthodes).

```
1 def __dir__(self):
2     d = super().__dir__()
3     d = [meth for meth in d
4         if getattr(getattr(type(self), meth, None), '__objclass__',
5                     None) is not super]
5     return d
```

Septième partie

Conclusion

VII. Conclusion

Ce cours touche maintenant à sa fin. Puisse-t-il vous avoir fait découvrir de nouveaux concepts du Python, ou l'envie de voir encore plus loin.

Ce cours ne couvre en effet qu'un nombre restreint de domaines, qui pourraient chacun être plus approfondis. Il y aurait encore tant à dire, sur les coroutines, sur les utilisations de l'interpréteur, sur les outils, les bibliothèques, *les frameworks*. À elle seule, la bibliothèque standard regorge encore de nombreuses choses, et je vous invite à voguer dans les pages de sa documentation.

Nous nous sommes ici surtout intéressés aux pages de documentation sur les types et le modèle de données. Mais elle comporte bien d'autres sections comme des tutoriaux, des *recettes*, la description de l'API C, etc.

Cependant, bien que cette documentation soit assez complète, elle ne l'est pas autant que le code source de l'interpréteur CPython. Écrit en C, son code reste très accessible et permet de mieux comprendre les mécanismes internes du langage.

Si vous souhaitez en apprendre plus sur la philosophie du langage, ou sur la bonne utilisation des concepts décrits dans le cours, je vous renvoie à [cet article sur le code pythonique](#) .

Je tiens aussi à remercier les différents contributeurs ayant aidé à l'élaboration de ce projet :

- [Vayel](#) , pour ses longues et intensives relectures ;
- [nohar](#) , pour les précisions apportées tout le long de l'écriture du tutoriel ;
- [Bibibye](#) , pour les nombreuses typographies corrigées ;
- [yoch](#) , pour ses diverses corrections.

Notez enfin que ce cours est diffusé sous licence *Creative Commons Attribution-ShareAlike 4.0* et que toute contribution est bienvenue. Les sources sont disponibles à l'adresse suivante : https://github.com/entwanne/cours_python_avance