Chapitre 6

Conception des classes

6.1 Introduction aux classes

6.1.1 Complément - niveau basique

On définit une classe lorsqu'on a besoin de créer un type spécifique au contexte de l'application. Il faut donc voir une classe au même niveau qu'un type *built-in* comme `list` ou `dict`.

Un exemple simpliste

Par exemple, imaginons qu'on a besoin de manipuler des matrices 2×2

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

Et en guise d'illustration, nous allons utiliser le déterminant ; c'est juste un prétexte pour implémenter une méthode sur cette classe, ne vous inquiétez pas si le terme ne vous dit rien, ou vous rappelle de mauvais souvenirs. Tout ce qu'on a besoin de savoir c'est que, sur une matrice de ce type, le déterminant vaut :

$$det(A) = a_{11}.a_{22} - a_{12}.a_{21}$$

Dans la pratique, on utiliserait la classe `matrix` de `numpy` qui est une bibliothèque de calcul scientifique très populaire et largement utilisée. Mais comme premier exemple de classe, nous allons écrire **notre propre classe** `Matrix2` pour mettre en action les mécanismes de base des classes de python. Naturellement, il s'agit d'une implémentation jouet.

```
In [1]: class Matrix2:
            "Une implémentation sommaire de matrice carrée 2x2"

            def __init__(self, a11, a12, a21, a22):
                "construit une matrice à partir des 4 coefficients"
                self.a11 = a11
                self.a12 = a12
                self.a21 = a21
                self.a22 = a22

            def determinant(self):
                "renvoie le déterminant de la matrice"
                return self.a11 * self.a22 - self.a12 * self.a21
```

La première version de `Matrix2`

Une classe peut avoir un *docstring*

Pour commencer, vous remarquez qu'on peut attacher à cette classe un *docstring* comme pour les fonctions

```
In [2]: help(Matrix2)

Help on class Matrix2 in module __main__:

class Matrix2(builtins.object)
 |  Une implémentation sommaire de matrice carrée 2x2
 |
```

```
|  Methods defined here:
|
|  __init__(self, a11, a12, a21, a22)
|      construit une matrice à partir des 4 coefficients
|
|  determinant(self)
|      renvoie le déterminant de la matrice
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

La classe définit donc deux méthodes, nommées `__init__` et `determinant`.

La méthode `__init__`

La méthode `__init__`, comme toutes celles qui ont un nom en *__nom__*, est une **méthode spéciale**. En l'occurrence, il s'agit de ce qu'on appelle le **constructeur** de la classe, c'est-à-dire le code qui va être appelé lorsqu'on crée une instance. Voyons cela tout de suite sur un exemple.

```
In [3]: matrice = Matrix2(1, 2, 2, 1)
        print(matrice)
```

```
<__main__.Matrix2 object at 0x04CC57B0>
```

Vous remarquez tout d'abord que `__init__` s'attend à recevoir *5 arguments*, mais que nous appelons `Matrix2` avec seulement *4 arguments*.

L'argument surnuméraire, le **premier** de ceux qui sont déclarés dans la méthode, correspond à l'**instance qui vient d'être créée** et qui est automatiquement passée par l'interpréteur python à la méthode `__init__`. En ce sens, le terme constructeur est impropre puisque la méthode `__init__` ne crée pas l'instance, elle ne fait que l'initialiser, mais c'est un abus de langage très répandu. Nous reviendrons sur le processus de création des objets lorsque nous parlerons des métaclasses en dernière semaine.

La **convention** est de nommer le premier argument de ce constructeur `self`, nous y reviendrons un peu plus loin.

On voit également que le constructeur se contente de mémoriser, à l'intérieur de l'instance, les arguments qu'on lui passe, sous la forme d'**attributs** de l'**instance** `self`.

C'est un cas extrêmement fréquent ; de manière générale, il est recommandé d'écrire des constructeurs passifs de ce genre ; dit autrement, on évite de faire trop de traitements dans le constructeur.

La méthode `determinant`

La classe définit aussi la méthode `determinant`, qu'on utiliserait comme ceci :

```
In [4]: matrice.determinant()
```

```
Out[4]: -3
```

Vous voyez que la **syntaxe** pour appeler une méthode sur un objet est **identique** à celle que nous avons utilisée jusqu'ici avec **les types de base**. Nous verrons très bientôt comment on peut pousser beaucoup plus loin la similitude, pour pouvoir par exemple calculer la **somme** de deux objets de la classe `Matrix2` avec l'opérateur `+`, mais n'anticipons pas.

Vous voyez aussi que, ici encore, la méthode définie dans la classe attend ***1 argument*** `self`, alors qu'apparemment nous ne lui en passons **aucun**. Comme tout à l'heure avec le constructeur, le premier argument passé automatiquement par l'interpréteur python à `determinant` est l'objet `matrice` lui-même.

En fait on aurait pu aussi bien écrire, de manière parfaitement équivalente :

```
In [5]: Matrix2.determinant(matrice)
```

```
Out[5]: -3
```

qui n'est presque jamais utilisé en pratique, mais qui illustre bien ce qui se passe lorsqu'on invoque une méthode sur un objet. En réalité, lorsque l'on écrit `matrice.determinant()` l'interpréteur python va essentiellement convertir cette expression en `Matrix2.determinant(matrice)`.

6.1.2 Complément - niveau intermédiaire

À quoi ça sert ?

Ce cours n'est pas consacré à la Programmation Orientée Objet (OOP) en tant que telle. Voici toutefois quelques-uns des avantages qui sont généralement mis en avant :

— encapsulation ;
— résolution dynamique de méthode ;
— héritage.

Encapsulation L'idée de la notion d'encapsulation consiste à ce que :

— une classe définit son **interface**, c'est-à-dire les méthodes par lesquelles on peut utiliser ce code,
— mais reste tout à fait libre de modifier son **implémentation**, et tant que cela n'impacte pas l'interface, **aucun changement** n'est requis dans les **codes utilisateurs**.

Nous verrons plus bas une deuxième implémentation de `Matrix2` qui est plus générale que notre première version, mais qui utilise la même interface, donc qui fonctionne exactement de la même manière pour le code utilisateur.

La notion d'encapsulation peut paraître à première vue banale ; il ne faut pas s'y fier, c'est de cette manière qu'on peut efficacement découper un gros logiciel en petits morceaux indépendants, et réellement découplés les uns des autres, et ainsi casser, réduire la complexité.

La programmation objet est une des techniques permettant d'atteindre cette bonne propriété d'encapsulation. Il faut reconnaître que certains langages comme Java et C++ ont des mécanismes plus sophistiqués, mais aussi plus complexes, pour garantir une bonne étanchéité entre l'interface publique et les détails d'implémentation. Les choix faits en la matière en Python reviennent, une fois encore, à privilégier la simplicité.

Aussi, il n'existe pas en Python l'équivalent des notions d'interface `public`, `private` et `protected` qu'on trouve en C++ et en Java. Il existe tout au plus une convention, selon laquelle les attributs commençant par un underscore (le tiret bas _) sont privés et ne *devraient* pas être utilisés par un code tiers, mais le langage ne fait rien pour garantir le bon usage de cette convention.

Si vous désirez creuser ce point nous vous conseillons de lire :

— *Reserved classes of identifiers* où l'on décrit également les noms privés à une classe (les noms de variables en *__nom*) ;
— *Private Variables and Class-local References*, qui en donne une illustration.

Malgré cette simplicité revendiquée, les classes de python permettent d'implémenter en pratique une encapsulation tout à fait acceptable, on peut en juger rien que par le nombre de bibliothèques tierces existantes dans l'écosystème python.

Résolution dynamique de méthode

Le deuxième atout de OOP, c'est le fait que l'envoi de méthode est résolu lors de l'exécution (*run-time*) et non pas lors de la compilation (*compile-time*). Ceci signifie que l'on peut écrire du code générique, qui pourra fonctionner avec des objets non connus *a priori*. Nous allons en voir un exemple tout de suite, en redéfinissant le comportement de `print` dans la deuxième implémentation de `Matrix2`.

Héritage

L'héritage est le concept qui permet de :

— dupliquer une classe presque à l'identique, mais en redéfinissant une ou quelques méthodes seulement (héritage simple) ;
— composer plusieurs classes en une seule, pour réaliser en quelque sorte l'union des propriétés de ces classes (héritage multiple).

Illustration

Nous revenons sur l'héritage dans une prochaine vidéo. Dans l'immédiat, nous allons voir une seconde implémentation de la classe `Matrix2`, qui illustre l'encapsulation et l'envoi dynamique de méthodes.

Pour une raison ou pour une autre, disons que l'on décide de remplacer les 4 attributs nommés `self.a11`, `self.a12`, etc., qui n'étaient pas très extensibles, par un seul attribut `a` qui regroupe tous les coefficients de la matrice dans un seul tuple.

```
In [6]: class Matrix2:
            """Une deuxième implémentation, tout aussi
            sommaire, mais différente, de matrice carrée 2x2"""
```

```
            def __init__(self, a11, a12, a21, a22):
                "construit une matrice à partir des 4 coefficients"
                # on décide d'utiliser un tuple plutôt que de ranger
                # les coefficients individuellement
                self.a = (a11, a12, a21, a22)

            def determinant(self):
                "le déterminant de la matrice"
                return self.a[0] * self.a[3] - self.a[1] * self.a[2]

            def __repr__(self):
                "comment présenter une matrice dans un print()"
                return f"<<mat-2x2 {self.a}>>"
```

Grâce à l'**encapsulation**, on peut continuer à utiliser la classe exactement de la même manière :

```
In [7]: matrice = Matrix2(1, 2, 2, 1)
        print("Determinant =", matrice.determinant())

Determinant = -3
```

Et en prime, grâce à la **résolution dynamique de méthode**, et parce que dans cette seconde implémentation on a défini une autre méthode spéciale `__repr__`, nous avons maintenant une impression beaucoup plus lisible de l'objet `matrice` :

```
In [8]: print(matrice)

<<mat-2x2 (1, 2, 2, 1)>>
```

Ce format d'impression reste d'ailleurs valable dans l'impression d'objets plus compliqués, comme par exemple :

```
In [9]: # on profite de ce nouveau format d'impression même si on met
        # par exemple un objet Matrix2 à l'intérieur d'une liste
        composite = [matrice, None, Matrix2(1, 0, 0, 1)]
        print(f"composite={composite}")

composite=[<<mat-2x2 (1, 2, 2, 1)>>, None, <<mat-2x2 (1, 0, 0, 1)>>]
```

Cela est possible parce que le code de `print` envoie la méthode `__repr__` sur les objets qu'elle parcourt. Le langage fournit une façon de faire par défaut, comme on l'a vu plus haut avec la première implémentation de `Matrix2`; et en définissant notre propre méthode `__repr__` nous pouvons surcharger ce comportement, et définir notre format d'impression.

Nous reviendrons sur les notions de surcharge et d'héritage dans les prochaines séquences vidéos.

La convention d'utiliser `self`

Avant de conclure, revenons rapidement sur le nom `self` qui est utilisé comme nom pour le premier argument des méthodes habituelles (nous verrons en semaine 9 d'autres sortes de méthodes, les méthodes statiques et de classe, qui ne reçoivent pas l'instance comme premier argument).

Comme nous l'avons dit plus haut, le premier argument d'une méthode s'appelle `self` **par convention**. Cette pratique est particulièrement bien suivie, mais ce n'est qu'une convention, en ce sens qu'on aurait pu utiliser n'importe quel identificateur ; pour le langage `self` n'a aucun sens particulier, ce n'est pas un mot clé ni une variable *built-in*.

Ceci est à mettre en contraste avec le choix fait dans d'autres langages, comme par exemple en C++ où l'instance est référencée par le mot-clé `this`, qui n'est pas mentionné dans la signature de la méthode. En Python, selon le manifeste, *explicit is better than implicit*, c'est pourquoi on mentionne l'instance dans la signature, sous le nom `self`.

6.2 Enregistrements et instances

6.2.1 Complément - niveau basique

Un enregistrement implémenté comme une instance de classe

Nous reprenons ici la discussion commencée en semaine 3, où nous avions vu comment implémenter un enregistrement comme un dictionnaire. Un enregistrement est l'équivalent, selon les langages, de *struct* ou *record*.

Notre exemple était celui des personnes, et nous avions alors écrit quelque chose comme :

```
In [1]: pierre = {'nom': 'pierre', 'age': 25, 'email': 'pierre@foo.com'}
        print(pierre)
```

```
{'nom': 'pierre', 'age': 25, 'email': 'pierre@foo.com'}
```

Cette fois-ci nous allons implémenter la même abstraction, mais avec une classe `Personne` comme ceci :

```
In [2]: class Personne:
            """Une personne possède un nom, un âge et une adresse e-mail"""

            def __init__(self, nom, age, email):
                self.nom = nom
                self.age = age
                self.email = email

            def __repr__(self):
                # comme nous avons la chance de disposer de python-3.6
                # utilisons un f-string
                return f"<<{self.nom}, {self.age} ans, email:{self.email}>>"
```

Le code de cette classe devrait être limpide à présent ; voyons comment on l'utiliserait - en guise de rappel sur le passage d'arguments aux fonctions :

```
In [3]: personnes = [

            # on se fie à l'ordre des arguments dans le créateur
            Personne('pierre', 25, 'pierre@foo.com'),

            # ou bien on peut être explicite
            Personne(nom='paul', age=18, email='paul@bar.com'),

            # ou bien on mélange
            Personne('jacques', 52, email='jacques@cool.com'),
        ]
        for personne in personnes:
            print(personne)
```

```
<<pierre, 25 ans, email:pierre@foo.com>>
<<paul, 18 ans, email:paul@bar.com>>
<<jacques, 52 ans, email:jacques@cool.com>>
```

Un dictionnaire pour indexer les enregistrements

Nous pouvons appliquer exactement la même technique d'indexation qu'avec les dictionnaires :

```
In [4]: # on crée un index pour pouvoir rechercher efficacement
        # une personne par son nom
        index_par_nom = {personne.nom: personne for personne in personnes}
```

De façon à pouvoir facilement localiser une personne :

```
In [5]: pierre = index_par_nom['pierre']
        print(pierre)
```

```
<<pierre, 25 ans, email:pierre@foo.com>>
```

Encapsulation

Pour marquer l'anniversaire d'une personne, nous pourrions faire :

```
In [6]: pierre.age += 1
        pierre
```

```
Out[6]: <<pierre, 26 ans, email:pierre@foo.com>>
```

À ce stade, surtout si vous venez de C++ ou de Java, vous devriez vous dire que ça ne va pas du tout !

En effet, on a parlé dans le complément précédent des mérites de l'encapsulation, et vous vous dites que là, la classe n'est pas du tout encapsulée car le code utilisateur a besoin de connaître l'implémentation.

En réalité, avec les classes python on a la possibilité, grâce aux *properties*, de conserver ce style de programmation qui a l'avantage d'être très simple, tout en préservant une bonne encapsulation, comme on va le voir dans le prochain complément.

6.2.2 Complément - niveau intermédiaire

Illustrons maintenant qu'en Python on peut ajouter des méthodes à une classe *à la volée* - c'est-à-dire en dehors de l'instruction `class`.

Pour cela on tire simplement profit du fait que **les méthodes sont implémentées comme des attributs de l'objet classe**.

Ainsi, on peut étendre l'objet `classe` lui-même dynamiquement :

```
In [7]: # pour une implémentation réelle voyez la bibliothèque smtplib
        # https://docs.python.org/3/library/smtplib.html

        def sendmail(self, subject, body):
            "Envoie un mail à la personne"
            print(f"To: {self.email}")
            print(f"Subject: {subject}")
```

```
            print(f"Body: {body}")

        Personne.sendmail = sendmail
```

Ce code commence par définir une fonction en utilisant `def` et la signature de la méthode. La fonction accepte un premier argument `self`; exactement comme si on avait défini la méthode dans l'instruction `class`.

Ensuite, il suffit d'affecter la fonction ainsi définie à **l'attribut** `sendmail` de l'objet classe.

Vous voyez que c'est très simple, et à présent la classe a connaissance de cette méthode exactement comme si on l'avait définie dans la clause `class`, comme le montre l'aide :

```
In [8]: help(Personne)

Help on class Personne in module __main__:

class Personne(builtins.object)
 |  Une personne possède un nom, un âge et une adresse e-mail
 |
 |  Methods defined here:
 |
 |  __init__(self, nom, age, email)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  sendmail(self, subject, body)
 |      Envoie un mail à la personne
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Et on peut à présent utiliser cette méthode :

```
In [9]: pierre.sendmail("Coucou", "Salut ça va ?")

To: pierre@foo.com
Subject: Coucou
Body: Salut ça va ?
```

6.3 Les *property*

Note : nous reviendrons largement sur cette notion de property lorsque nous parlerons des *property et descripteurs* en semaine 9. Cependant, cette notion est suffisamment importante pour que nous vous proposions un complément dès maintenant dessus.

6.3.1 Complément - niveau intermédiaire

Comme on l'a vu dans le complément précédent, il est fréquent en Python qu'une classe expose dans sa documentation un ou plusieurs attributs ; c'est une pratique qui, en apparence seulement, paraît casser l'idée d'une bonne encapsulation.

En réalité, grâce au mécanisme de *property*, il n'en est rien. Nous allons voir dans ce complément comment une classe peut en quelque sorte intercepter les accès à ses attributs, et par là fournir une encapsulation forte.

Pour être concret, on va parler d'une classe `Temperature`. Au lieu de proposer, comme ce serait l'usage dans d'autres langages, une interface avec `get_kelvin()` et `set_kelvin()`, on va se contenter d'exposer l'attribut `kelvin`, et malgré cela on va pouvoir faire diverses vérifications et autres.

Implémentation naïve

Je vais commencer par une implémentation naïve, qui ne tire pas profit des *properties* :

```
In [1]: # dans sa version la plus épurée, une classe
        # température pourrait ressembler à ça :

        class Temperature1:
            def __init__(self, kelvin):
                self.kelvin = kelvin

            def __repr__(self):
                return f"{self.kelvin}K"
```

```
In [2]: # créons une instance
        t1 = Temperature1(20)
        t1
```

```
Out[2]: 20K
```

```
In [3]: # et pour accéder à la valeur numérique je peux faire
        t1.kelvin
```

```
Out[3]: 20
```

Avec cette implémentation il est très facile de créer une température négative, qui n'a bien sûr pas de sens physique, ce n'est pas bon.

Interface *getter/setter*

Si vous avez été déjà exposés à des langages orientés objet comme C++, Java ou autre, vous avez peut-être l'habitude d'accéder aux données internes des instances par des **méthodes** de type *getter* ***ou*** *setter*, de façon à contrôler les accès et, dans une optique d'encapsulation, de préserver des invariants, comme ici le fait que la température doit être positive.

C'est-à-dire que vous vous dites peut-être, ça ne devrait pas être fait comme ça, on devrait plutôt proposer une interface pour accéder à l'implémentation interne ; quelque chose comme :

```
In [4]: class Temperature2:
            def __init__(self, kelvin):
                # au lieu d'écrire l'attribut il est plus sûr
                # d'utiliser le setter
                self.set_kelvin(kelvin)

            def set_kelvin(self, kelvin):
                # je m'assure que _kelvin est toujours positif
                # et j'utilise un nom d'attribut avec un _ pour signifier
                # que l'attribut est privé et qu'il ne faut pas y toucher directement
                # on pourrait aussi bien sûr lever une exception
                # mais ce n'est pas mon sujet ici
                self._kelvin = max(0, kelvin)

            def get_kelvin(self):
                return self._kelvin

            def __repr__(self):
                return f"{self._kelvin}K"
```

Bon c'est vrai que d'un coté, c'est mieux parce que je garantis un invariant, la température est toujours positive :

```
In [5]: t2 = Temperature2(-30)
        t2

Out[5]: 0K
```

Mais par contre, d'un autre coté, c'est très lourd, parce que chaque fois que je veux utiliser mon objet, je dois faire pour y accéder :

```
In [6]: t2.get_kelvin()

Out[6]: 0
```

et aussi, si j'avais déjà du code qui utilisait `t.kelvin` il va falloir le modifier entièrement.

Implémentation pythonique

La façon de s'en sortir ici consiste à définir une property. Comme on va le voir ce mécanisme permet d'écrire du code qui fait référence à l'attribut `kelvin` de l'instance, mais qui passe tout de même par une couche de logique.

Ça ressemblerait à ceci :

```
In [7]: class Temperature3:
            def __init__(self, kelvin):
                self.kelvin = kelvin

            # je définis bel et bien mes accesseurs de type getter et setter
            # mais _get_kelvin commence avec un _
            # car il n'est pas censé être appelé par l'extérieur
            def _get_kelvin(self):
                return self._kelvin

            # idem
            def _set_kelvin(self, kelvin):
                self._kelvin = max(0, kelvin)

            # une fois que j'ai ces deux éléments je peux créer une property
            kelvin = property(_get_kelvin, _set_kelvin)

            # et toujours la façon d'imprimer
            def __repr__(self):
                return f"{self._kelvin}K"

In [8]: t3 = Temperature3(200)
        t3

Out[8]: 200K

In [9]: # par contre ici on va le mettre à zéro
        # à nouveau, une exception serait préférable sans doute
        t3.kelvin = -30
        t3

Out[9]: 0K
```

Comme vous pouvez le voir, cette technique a plusieurs avantages :

— on a ce qu'on cherchait, c'est-à-dire une façon d'ajouter une couche de logique lors des accès en lecture et en écriture à l'intérieur de l'objet,
— mais **sans toutefois** demander à l'utilisateur de passer son temps à envoyer des méthodes `get_` et `set()` sur l'objet, ce qui a tendance à alourdir considérablement le code.

C'est pour cette raison que vous ne rencontrerez presque jamais en Python une bibliothèque qui offre une interface à base de méthodes `get_something` et `set_something`, mais au contraire les API vous exposeront directement des attributs que vous devez utiliser directement.

6.3.2 Complément - niveau avancé

À titre d'exemple d'utilisation, voici une dernière implémentation de `Temperature` qui donne l'illusion d'avoir 3 attributs (`kelvin`, `celsius` et `fahrenheit`), alors qu'en réalité le seul attribut de donnée est `_kelvin`.

```
In [10]: class Temperature:

             ## les constantes de conversion
```

```
    # kelvin / celsius
    K = 273.16
    # fahrenheit / celsius
    RF = 5 / 9
    KF = (K / RF) - 32

    def __init__(self, kelvin=None, celsius=None, fahrenheit=None):
        """
        Création à partir de n'importe quelle unité
        Il faut préciser exactement une des trois unités
        """
        # on passe par les properties pour initialiser
        if kelvin is not None:
            self.kelvin = kelvin
        elif celsius is not None:
            self.celsius = celsius
        elif fahrenheit is not None:
            self.fahrenheit = fahrenheit
        else:
            self.kelvin = 0
            raise ValueError("need to specify at least one unit")

    # pour le confort
    def __repr__(self):
        return f"<{self.kelvin:g}K == {self.celsius:g}°C " \
               f"== {self.fahrenheit:g}°F>"

    def __str__(self):
        return f"{self.kelvin:g}K"


    # l'attribut 'kelvin' n'a pas de conversion à faire,
    # mais il vérifie que la valeur est positive
    def _get_kelvin(self):
        return self._kelvin

    def _set_kelvin(self, kelvin):
        if kelvin < 0:
            raise ValueError(f"Kelvin {kelvin} must be positive")
        self._kelvin = kelvin

    # la property qui définit l'attribut `kelvin`
    kelvin = property(_get_kelvin, _set_kelvin)


    # les deux autres properties font la conversion, puis
    # sous-traitent à la property kelvin pour le contrôle de borne
    def _set_celsius(self, celsius):
        # using .kelvin instead of ._kelvin to enforce
        self.kelvin = celsius + self.K
```

```
            def _get_celsius(self):
                return self._kelvin - self.K

            celsius = property(_get_celsius, _set_celsius)

            def _set_fahrenheit(self, fahrenheit):
                # using .kelvin instead of ._kelvin to enforce
                self.kelvin = (fahrenheit + self.KF) * self.RF

            def _get_fahrenheit(self):
                return self._kelvin / self.RF - self.KF

            fahrenheit = property(_get_fahrenheit, _set_fahrenheit)
```

Et voici ce qu'on peut en faire :

```
In [11]: t = Temperature(celsius=0)
         t

Out[11]: <273.16K == 0°C == 32°[F]>

In [12]: t.fahrenheit

Out[12]: 32.0

In [13]: t.celsius += 100
         print(t)

373.16K


In [14]: try:
             t = Temperature(fahrenheit = -1000)
         except Exception as e:
             print(f"OOPS, {type(e)}, {e}")

OOPS, <class 'ValueError'>, Kelvin -300.1733333333333 must be positive
```

Pour en savoir plus

Voir aussi la documentation officielle.

Vous pouvez notamment aussi, en option, ajouter un *deleter* pour intercepter les instructions du type :

```
In [15]: # comme on n'a pas défini de deleter, on ne peut pas faire ceci
         try:
             del t.kelvin
         except Exception as e:
             print(f"OOPS {type(e)} {e}")

OOPS <class 'AttributeError'> can't delete attribute
```

6.4 Un exemple de classes de la bibliothèque standard

Notez que ce complément, bien qu'un peu digressif par rapport au sujet principal qui est les classes et instances, a pour objectif de vous montrer l'intérêt de la programmation objet avec un module de la bibliothèque standard.

6.4.1 Complément - niveau basique

Le module `time`

Pour les accès à l'horloge, python fournit un module `time` - très ancien; il s'agit d'une interface de très bas niveau avec l'OS, qui s'utilise comme ceci :

```
In [1]: import time

        # on obtient l'heure courante sous la forme d'un flottant
        # qui représente le nombre de secondes depuis le 1er Janvier 1970
        t_now = time.time()
        t_now

Out[1]: 1539774899.2159433

In [2]: # et pour calculer l'heure qu'il sera dans trois heures on fait
        t_later = t_now + 3 * 3600
```

Nous sommes donc ici clairement dans une approche non orientée objet; on manipule des types de base, ici le type flottant :

```
In [3]: type(t_later)

Out[3]: float
```

Et comme on le voit, les calculs se font sous une forme pas très lisible. Pour rendre ce nombre de secondes plus lisible, on utilise des conversions, pas vraiment explicites non plus; voici par exemple un appel à `gmtime` qui convertit le flottant obtenu par la méthode `time()` en heure UTC (`gm` est pour Greenwich Meridian) :

```
In [4]: struct_later = time.gmtime(t_later)
        print(struct_later)

time.struct_time(tm_year=2018, tm_mon=10, tm_mday=17, tm_hour=14, tm_min=14,
tm_sec=59, tm_wday=2, tm_yday=290, tm_isdst=0)
```

Et on met en forme ce résultat en utilisant des méthodes comme, par exemple, `strftime()` pour afficher l'heure UTC dans 3 heures :

```
In [5]: print(f'heure UTC dans trois heures {time.strftime("%Y-%m-%d at %H:%M" \
, struct_later)}')

heure UTC dans trois heures 2018-10-17 at 14:14
```

Le module datetime

Voyons à présent, par comparaison, comment ce genre de calculs se présente lorsqu'on utilise la programmation par objets.

Le module `datetime` expose un certain nombre de classes, que nous illustrons brièvement avec les classes `datetime` (qui modélise la date et l'heure d'un instant) et `timedelta` (qui modélise une durée).

La première remarque qu'on peut faire, c'est qu'avec le module `time` on manipulait un flottant pour représenter ces deux sortes d'objets (instant et durée) ; avec deux classes différentes notre code va être plus clair quant à ce qui est réellement représenté.

Le code ci-dessus s'écrirait alors, en utilisant le module `datetime` :

```
In [6]: from datetime import datetime, timedelta

        dt_now = datetime.now()
        dt_later = dt_now + timedelta(hours=3)
```

Vous remarquez que c'est déjà un peu plus expressif.

Voyez aussi qu'on a déjà moins besoin de s'escrimer pour en avoir un aperçu lisible :

```
In [7]: # on peut imprimer simplement un objet date_time
        print(f'maintenant {dt_now}')

maintenant 2018-10-17 13:14:59.309541


In [8]: # et si on veut un autre format, on peut toujours appeler strftime
        print(f'dans trois heures {dt_later.strftime("%Y-%m-%d at %H:%M")}')

dans trois heures 2018-10-17 at 16:14


In [9]: # mais ce n'est même pas nécessaire, on peut passer le format directement
        print(f'dans trois heures {dt_later:%Y-%m-%d at %H:%M}')

dans trois heures 2018-10-17 at 16:14
```

Je vous renvoie à la documentation du module, et notamment ici, pour le détail des options de formatage disponibles.

Conclusion

Une partie des inconvénients du module `time` vient certainement du parti-pris de l'efficacité. De plus, c'est un module très ancien, mais auquel on ne peut guère toucher pour des raisons de compatibilité ascendante.

Par contre, le module `datetime`, tout en vous procurant un premier exemple de classes exposées par la bibliothèque standard, vous montre certains des avantages de la programmation

orientée objet en général, et des classes de Python en particulier.

Si vous devez manipuler des dates ou des heures, le module `datetime` constitue très certainement un bon candidat ; voyez la documentation complète du module pour plus de précisions sur ses possibilités.

6.4.2 Complément - niveau intermédiaire

Fuseaux horaires et temps local

Le temps nous manque pour traiter ce sujet dans toute sa profondeur.

En substance, c'est un sujet assez voisin de celui des accents, en ce sens que lors d'échanges d'informations de type *timestamp* entre deux ordinateurs, il faut échanger d'une part une valeur (l'heure et la date), et d'autre part le référentiel (s'agit-il de temps UTC, ou bien de l'heure dans un fuseau horaire, et si oui lequel).

La complexité est tout de même moindre que dans le cas des accents ; on s'en sort en général en convenant d'échanger systématiquement des heures UTC. Par contre, il existe une réelle diversité quant au format utilisé pour échanger ce type d'information, et cela reste une source d'erreurs assez fréquente.

6.4.3 Complément - niveau avancé

Classes et *marshalling*

Ceci nous procure une transition pour un sujet beaucoup plus général.

Nous avons évoqué en semaine 4 les formats comme JSON pour échanger les données entre applications, au travers de fichiers ou d'un réseau.

On a vu, par exemple, que JSON est un format "proche des langages" en ce sens qu'il est capable d'échanger des objets de base comme des listes ou des dictionnaires entre plusieurs langages comme, par exemple, JavaScript, Python ou Ruby. En XML, on a davantage de flexibilité puisqu'on peut définir une syntaxe sur les données échangées.

Mais il faut être bien lucide sur le fait que, aussi bien pour JSON que pour XML, il n'est **pas possible** d'échanger entre applications des **objets** en tant que tel. Ce que nous voulons dire, c'est que ces technologies de *marshalling* prennent bien en charge le *contenu* en termes de données, mais pas les informations de type, et *a fortiori* pas non plus le code qui appartient à la classe.

Il est important d'être conscient de cette limitation lorsqu'on fait des choix de conception, notamment lorsqu'on est amené à choisir entre classe et dictionnaire pour l'implémentation de telle ou telle abstraction.

Voyons cela sur un exemple inspiré de notre fichier de données liées au trafic maritime. En version simplifiée, un bateau est décrit par trois valeurs, son identité (`id`), son nom et son pays d'attachement.

Nous allons voir comment on peut échanger ces informations entre, disons, deux programmes dont l'un est en Python, via un support réseau ou disque.

Si on choisit de simplement manipuler un dictionnaire standard :

```
In [10]: bateau1 = {'name' : "Toccata", 'id' : 1000, 'country' : "France"}
```

alors on peut utiliser tels quels les mécanismes d'encodage et décodage de, disons, JSON. En effet c'est exactement ce genre d'informations que sait gérer la couche JSON.

Si au contraire on choisit de manipuler les données sous forme d'une classe on pourrait avoir envie d'écrire quelque chose comme ceci :

```
In [11]: class Bateau:
             def __init__(self, id, name, country):
                 self.id = id
                 self.name = name
                 self.country = country

         bateau2 = Bateau(1000, "Toccata", "FRA")
```

Maintenant, si vous avez besoin d'échanger cet objet avec le reste du monde, en utilisant par exemple JSON, tout ce que vous allez pouvoir faire passer par ce médium, c'est la valeur des trois champs, dans un dictionnaire. Vous pouvez facilement obtenir le dictionnaire en question pour le passer à la couche d'encodage :

```
In [12]: vars(bateau2)

Out[12]: {'country': 'FRA', 'id': 1000, 'name': 'Toccata'}
```

Mais à l'autre bout de la communication il va vous falloir :

— déterminer d'une manière ou d'une autre que les données échangées sont en rapport avec la classe `Bateau` ;
— construire vous même un objet de cette classe, par exemple avec un code comme :

```
In [13]: # du côté du récepteur de la donnée
         class Bateau:
             def __init__(self, *args):
                 if len(args) == 1 and isinstance(args[0], dict):
                     self.__dict__ = args[0]
                 elif len(args) == 3:
                     id, name, country = args
                     self.id = id
                     self.name = name
                     self.country = country

         bateau3 = Bateau({'id': 1000, 'name': 'Leon', 'country': 'France'})
         bateau4 = Bateau(1001, 'Maluba', 'SUI' )
```

Conclusion

Pour reformuler ce dernier point, il n'y a pas en Python l'équivalent de jmi (Java Metadata Interface) intégré à la distribution standard.

De plus on peut écrire du code en dehors des classes, et on n'est pas forcément obligé d'écrire une classe pour tout - à l'inverse ici encore de Java. Chaque situation doit être jugée dans son contexte naturellement, mais, de manière générale, la classe n'est pas la solution universelle ; il peut y avoir des mérites dans le fait de manipuler certaines données sous une forme allégée comme un type natif.

6.5 Manipuler des ensembles d'instances

6.5.1 Complément - niveau intermédiaire

Souvenez-vous de ce qu'on avait dit en semaine 3 séquence 4, concernant les clés dans un dictionnaire ou les éléments dans un ensemble. Nous avions vu alors que, pour les types *built-in*, les clés devaient être des objets immuables et même globalement immuables.

Nous allons voir dans ce complément quelles sont les règles qui s'appliquent aux instances de classe.

Instance mutable dans un ensemble

Une instance de classe est par défaut un objet mutable. Malgré cela, le langage vous permet d'insérer une instance dans un ensemble - ou de l'utiliser comme clé dans un dictionnaire. Nous allons voir ce mécanisme en action.

Hachage par défaut : basé sur `id()`

```
In [1]: # une classe Point
        class Point1:
            def __init__(self, x, y):
                self.x = x
                self.y = y

            def __repr__(self):
                return f"Pt[{self.x}, {self.y}]"
```

Avec ce code, les instances de `Point` sont mutables :

```
In [2]: # deux instances
        p1 = Point1(2, 2)
        p2 = Point1(2, 3)
```

```
In [3]: # objets mutables
        p1.y = 3
```

Mais par contre soyez attentifs, car il faut savoir que pour la classe `Point1`, où nous n'avons rien redéfini, la fonction de hachage sur une instance de `Point1` ne dépend que de la valeur de `id()` sur cet objet.

Ce qui, dit autrement, signifie que deux objets qui sont distincts au sens de `id()` sont considérés comme différents, et donc peuvent coexister dans un ensemble (ou dans un dictionnaire) :

```
In [4]: # nos deux objets se ressemblent
        p1, p2
```

```
Out[4]: (Pt[2, 3], Pt[2, 3])
```

```
In [5]: # mais peuvent coexister
        # dans un ensemble
        # qui a alors 2 éléments
        s = { p1, p2 }
        len(s)
```

```
Out[5]: 2
```

Si on recherche un de ces deux objets on le trouve :

```
In [6]: p1 in s

Out[6]: True

In [7]: # mais pas un troisième
        p3 = Point1(2, 4)
        p3 in s

Out[7]: False
```

Cette possibilité de gérer des ensembles d'objets selon cette stratégie est très commode et peut apporter de gros gains de performance, notamment lorsqu'on a souvent besoin de faire des tests d'appartenance.

En pratique, lorsqu'un modèle de données définit une relation de type "1-n", je vous recommande d'envisager d'utiliser un ensemble plutôt qu'une liste.

Par exemple envisagez :

```
class Animal:
    # blabla

class Zoo:
    def __init__(self):
        self.animals = set()
```

Plutôt que :

```
class Animal:
    # blabla

class Zoo:
    def __init__(self):
        self.animals = []
```

6.5.2 Complément - niveau avancé

Ce n'est pas ce que vous voulez ?

Le comportement qu'on vient de voir pour les instances de `Point1` dans les tables de hachage est raisonnable, si on admet que deux points ne sont égaux que s'ils sont **le même objet** au sens de `is`.

Mais imaginons que vous voulez au contraire considérer que deux points son égaux lorsqu'ils coincident sur le plan. Avec ce modèle de données, vous voudriez que :

— un ensemble dans lequel on insère `p1` et `p2` ne contienne qu'un élément,
— et qu'on trouve `p3` quand on le cherche dans cet ensemble.

Le protocole *hashable* : `__hash__` et `__eq__`

Le langage nous permet de faire cela, grâce au protocole *hashable* ; pour cela il nous faut définir deux méthodes :

— `__eq__` qui, sans grande surprise, va servir à évaluer `p == q` ;
— `__hash__` qui va retourner la clé de hachage sur un objet.

La subtilité étant bien entendu que ces deux méthodes doivent être cohérentes, si deux objets sont égaux, il faut que leurs hashs soient égaux ; de bon sens, si l'égalité se base sur nos deux attributs `x` et `y`, il faudra bien entendu que la fonction de hachage utilise elle aussi ces deux attributs. Voir la documentation de `__hash__`.

Voyons cela sur une sous-classe de `Point1`, dans laquelle nous définissons ces deux méthodes :

```
In [8]: class Point2(Point1):

            # l'égalité va se baser naturellement sur x et y
            def __eq__(self, other):
                return self.x == other.x and self.y == other.y

            # du coup la fonction de hachage
            # dépend aussi de x et de y
            def __hash__(self):
                return (11 * self.x + self.y) // 16
```

On peut vérifier que cette fois les choses fonctionnent correctement :

```
In [9]: q1 = Point2(2, 3)
        q2 = Point2(2, 3)
```

Nos deux objets sont distincts pour `id()`/`is`, mais égaux pour `==` :

```
In [10]: print(f"is → {q1 is q2} \n== → {q1 == q2}")

is → False
== → True
```

Et un ensemble contenant les deux points n'en contient qu'un :

```
In [11]: s = {q1, q2}
         len(s)

Out[11]: 1

In [12]: q3 = Point2(2, 3)
         q3 in s

Out[12]: True
```

Comme les ensembles et les dictionnaires reposent sur le même mécanisme de table de hachage, on peut aussi indifféremment utiliser n'importe lequel de nos 3 points pour indexer un dictionnaire :

```
In [13]: d = {}
         d[q1] = 1
         d[q2]

Out[13]: 1

In [14]: # les clés q1, q2 et q3 sont
         # les mêmes pour le dictionnaire
         d[q3] = 10000
         d

Out[14]: {Pt[2, 3]: 10000}
```

Attention !

Tout ceci semble très bien fonctionner ; sauf qu'en fait, il y a une **grosse faille**, c'est que nos objets `Point2` sont **mutables**. Du coup on peut maintenant imaginer un scénario comme celui-ci :

```
In [15]: t1, t2 = Point2(10, 10), Point2(10, 10)
         s = {t1, t2}
         s

Out[15]: {Pt[10, 10]}

In [16]: t1 in s, t2 in s

Out[16]: (True, True)
```

Mais si maintenant je change un des deux objets :

```
In [17]: t1.x = 100

In [18]: s

Out[18]: {Pt[100, 10]}

In [19]: t1 in s

Out[19]: False

In [20]: t2 in s

Out[20]: False
```

Évidemment cela n'est pas correct. Ce qui se passe ici c'est qu'on a

— d'abord inséré `t1` dans `s`, avec un indice de hachage calculé à partir de `10, 10`
— pas inséré `t2` dans `s` parce qu'on a déterminé qu'il existait déjà.

Après avoir modifié `t1` qui est le seul élément de `s` : À ce stade :

— lorsqu'on cherche `t1` dans `s`, on le fait avec un indice de hachage calculé à partir de `100, 10` et du coup on ne le trouve pas,
— lorsqu'on cherche `t2` dans `s`, on utilise le bon indice de hachage, mais ensuite le seul élément qui pourrait faire l'affaire n'est pas égal à `t2`.

Conclusion

La documentation de Python sur ce sujet indique ceci :

> If a class defines mutable objects and implements an `__eq__`() method, it should not implement `__hash__`(), since the implementation of hashable collections requires that a key's hash value is immutable (if the object's hash value changes, it will be in the wrong hash bucket).

Notre classe `Point2` illustre bien cette limitation. Pour qu'elle soit utilisable en pratique, il faut **rendre ses instances immutables**. Cela peut se faire de plusieurs façons, dont deux que nous aborderons dans la prochaine séquence et qui sont - entre autres :

— le `namedtuple`
— et la `dataclass` (nouveau en 3.7).

6.6 Surcharge d'opérateurs (1)

6.6.1 Complément - niveau intermédiaire

Ce complément vise à illustrer certaines des possibilités de surcharge d'opérateurs, ou plus généralement les mécanismes disponibles pour étendre le langage et donner un sens à des fragments de code comme :

- `objet1 + objet2`
- `item in objet`
- `objet[key]`
- `objet.key`
- `for i in objet:`
- `if objet:`
- `objet(arg1, arg2)` (et non pas `classe(arg1, arg2)`)
- etc..

que jusqu'ici, sauf pour la boucle `for` et pour le hachage, on n'a expliqués que pour des objets de type prédéfini.

Le mécanisme général pour cela consiste à définir des **méthodes spéciales**, avec un nom en `__nom__`. Il existe un total de près de 80 méthodes dans ce système de surcharges, aussi il n'est pas question ici d'être exhaustif. Vous trouverez dans ce document une liste complète de ces possibilités.

Il nous faut également signaler que les mécanismes mis en jeu ici sont **de difficultés assez variables**. Dans le cas le plus simple il suffit de définir une méthode sur la classe pour obtenir le résultat (par exemple, définir `__call__` pour rendre un objet callable). Mais parfois on parle d'un ensemble de méthodes qui doivent être cohérentes, voyez par exemple les descriptors qui mettent en jeu les méthodes `__get__`, `__set__` et `__delete__`, et qui peuvent sembler particulièrement cryptiques. On aura d'ailleurs l'occasion d'approfondir les descriptors en semaine 9 avec les sujets avancés.

Nous vous conseillons de commencer par des choses simples, et surtout de n'utiliser ces techniques que lorsqu'elles apportent vraiment quelque chose. Le constructeur et l'affichage sont pratiquement toujours définis, mais pour tout le reste il convient d'utiliser ces traits avec le plus grand discernement. Dans tous les cas écrivez votre code avec la documentation sous les yeux, c'est plus prudent :)

Nous avons essayé de présenter cette sélection par difficulté croissante. Par ailleurs, et pour alléger la présentation, cet exposé a été coupé en trois notebooks différents.

Rappels

Pour rappel, on a vu dans la vidéo :

- la méthode `__init__` pour définir un **constructeur** ;
- la méthode `__str__` pour définir comment une instance s'imprime avec `print`.

Affichage : `__repr__` et `__str__`

Nous commençons par signaler la méthode `__repr__` qui est assez voisine de `__str__`, et qui donc doit retourner un objet de type chaîne de caractères, sauf que :

— `__str__` est utilisée par `print` (affichage orienté utilisateur du programme, priorité au confort visuel) ;
— alors que `__repr__` est utilisée par la fonction `repr()` (affichage orienté programmeur, aussi peu ambigu que possible) ;
— enfin il faut savoir que `__repr__` est utilisée **aussi** par `print` si `__str__` n'est pas définie.

Pour cette dernière raison, on trouve dans la nature `__repr__` plutôt plus souvent que `__str__` ; voyez ce lien pour davantage de détails.

Quand est utilisée `repr()` ?

La fonction `repr()` est utilisée massivement dans les informations de debugging comme les traces de pile lorsqu'une exception est levée. Elle est aussi utilisée lorsque vous affichez un objet sans passer par `print`, c'est-à-dire par exemple :

```
In [1]: class Foo:
            def __repr__(self):
                return 'custom repr'

        foo = Foo()
        # lorsque vous affichez un objet comme ceci
        foo
        # en fait vous utilisez repr()

Out[1]: custom repr
```

Deux exemples

Voici deux exemples simples de classes ; dans le premier on n'a défini que `__repr__`, dans le second on a redéfini les deux méthodes :

```
In [2]: # une classe qui ne définit que __repr__
        class Point:
            "première version de Point - on ne définit que __repr__"
            def __init__(self, x, y):
                self.x = x
                self.y = y
            def __repr__(self):
                return f"Point({self.x},{self.y})"

        point = Point (0,100)

        print("avec print", point)

        # si vous affichez un objet sans passer par print
        # vous utilisez repr()
        point

avec print Point(0,100)


Out[2]: Point(0,100)
```

```
In [3]: # la même chose mais où on redéfinit __str__ et __repr__
        class Point2:
            "seconde version de Point - on définit __repr__ et __str__"
            def __init__(self, x, y):
                self.x = x
                self.y = y
            def __repr__(self):
                return f"Point2({self.x},{self.y})"
            def __str__(self):
                return f"({self.x},{self.y})"

        point2 = Point2 (0,100)

        print("avec print", point2)

        # les f-strings (ou format) utilisent aussi __str__
        print(f"avec format {point2}")

        # et si enfin vous affichez un objet sans passer par print
        # vous utilisez repr()
        point2
```

```
avec print (0,100)
avec format (0,100)
```

```
Out[3]: Point2(0,100)
```

`__bool__`

Vous vous souvenez que la condition d'un test dans un `if` peut ne pas retourner un booléen (nous avons vu cela en Semaine 4, Séquence "Test if/elif/else et opérateurs booléens"). Nous avions noté que pour les types prédéfinis, sont considérés comme *faux* les objets : `None`, la liste vide, un tuple vide, etc.

Avec `__bool__` on peut redéfinir le comportement des objets d'une classe vis-à-vis des conditions - ou si l'on préfère, quel doit être le résultat de `bool(instance)`.

Attention pour éviter les comportements imprévus, comme on est en train de redéfinir le comportement des conditions, il **faut** renvoyer un **booléen** (ou à la rigueur 0 ou 1), on ne peut pas dans ce contexte retourner d'autres types d'objet.

Nous allons **illustrer** cette méthode dans un petit moment avec une nouvelle implémentation de la classe `Matrix2`.

Remarquez enfin qu'en l'absence de méthode `__bool__`, on cherche aussi la méthode `__len__` pour déterminer le résultat du test ; une instance de longueur nulle est alors considéré comme `False`, en cohérence avec ce qui se passe avec les types *built-in* `list`, `dict`, `tuple`, etc.

Ce genre de *protocole*, qui cherche d'abord une méthode (`__bool__`), puis une autre (`__len__`) en cas d'absence de la première, est relativement fréquent dans la mécanique de surcharge des opérateurs ; c'est entre autres pourquoi la documentation est indispensable lorsqu'on surcharge les opérateurs.

`__add__` et apparentés (`__mul__`, `__sub__`, `__div__`, `__and__`, etc.)

On peut également redéfinir les opérateurs arithmétiques et logiques. Dans l'exemple qui suit, nous allons l'illustrer sur l'addition de matrices. On rappelle pour mémoire que :

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} \\ a_{21}+b_{21} & a_{22}+b_{22} \end{pmatrix}$$

Une nouvelle version de la classe `Matrix2`

Voici (encore) une nouvelle implémentation de la classe de matrices 2x2, qui illustre cette fois :

- la possibilité d'ajouter deux matrices ;
- la possibilité de faire un test sur une matrice - le test sera faux si la matrice a tous ses coefficients nuls ;
- et, bien que ce ne soit pas le sujet immédiat, cette implémentation illustre aussi la possibilité de construire la matrice à partir :
  - soit des 4 coefficients, comme par exemple : `Matrix2(a, b, c, d)`
  - soit d'une séquence, comme par exemple : `Matrix2(range(4))`

Cette dernière possibilité va nous permettre de simplifier le code de l'addition, comme on va le voir.

```
In [4]: # notre classe Matrix2 avec encore une autre implémentation
        class Matrix2:

            def __init__(self, *args):
                """
                le constructeur accepte
                (*) soit les 4 coefficients individuellement
                (*) soit une liste - ou + généralement une séquence - des mêmes
                """
                # on veut pouvoir créer l'objet à partir des 4 coefficients
                # souvenez-vous qu'avec la forme *args, args est toujours un tuple
                if len(args) == 4:
                    self.coefs = args
                # ou bien d'une séquence de 4 coefficients
                elif len(args) == 1:
                    self.coefs = tuple(*args)

            def __repr__(self):
                "l'affichage"
                return "[" + ", ".join([str(c) for c in self.coefs]) + "]"

            def __add__(self, other):
                """
                l'addition de deux matrices retourne un nouvel objet
                la possibilité de créer une matrice à partir
                d'une liste rend ce code beaucoup plus facile à écrire
                """
                return Matrix2([a + b for a, b in zip(self.coefs, other.coefs)])
```

```
    def __bool__(self):
        """
        on considère que la matrice est non nulle
        si un au moins de ses coefficients est non nul
        """
        # ATTENTION le retour doit être un booléen
        # ou à la rigueur 0 ou 1
        for c in self.coefs:
            if c:
                return True
        return False
```

On peut à présent créer deux objets, les ajouter, et vérifier que la matrice nulle se comporte bien comme attendu :

```
In [5]: zero     = Matrix2 ([0,0,0,0])

        matrice1 = Matrix2 (1,2,3,4)
        matrice2 = Matrix2 (list(range(10,50,10)))

        print('avant matrice1', matrice1)
        print('avant matrice2', matrice2)

        print('somme', matrice1 + matrice2)

        print('après matrice1', matrice1)
        print('après matrice2', matrice2)

        if matrice1:
            print(matrice1,"n'est pas nulle")
        if not zero:
            print(zero,"est nulle")
```

```
avant matrice1 [1, 2, 3, 4]
avant matrice2 [10, 20, 30, 40]
somme [11, 22, 33, 44]
après matrice1 [1, 2, 3, 4]
après matrice2 [10, 20, 30, 40]
[1, 2, 3, 4] n'est pas nulle
[0, 0, 0, 0] est nulle
```

Voici en vrac quelques commentaires sur cet exemple.

Utiliser un tuple

Avant de parler de la surcharge des opérateurs *per se*, vous remarquerez que l'on range les coefficients dans un **tuple**, de façon à ce que notre objet `Matrix2` soit indépendant de l'objet qu'on a utilisé pour le créer (et qui peut être ensuite modifié par l'appelant).

Créer un nouvel objet Vous remarquez que l'addition `__add__` renvoie un **nouvel objet**, au lieu de modifier `self` en place. C'est la bonne façon de procéder tout simplement parce que lorsqu'on écrit :

```
print('somme', matrice1 + matrice2)
```

on ne s'attend pas du tout à ce que `matrice1` soit modifiée après cet appel.

Du code qui ne dépend que des 4 opérations

Le fait d'avoir défini l'addition nous permet par exemple de bénéficier de la fonction *built-in* `sum`. En effet le code de `sum` fait lui-même des additions, il n'y a donc aucune raison de ne pas pouvoir l'exécuter avec en entrée une liste de matrices puisque maintenant on sait les additionner, (mais on a dû toutefois passer à `sum` comme élément neutre `zero`) :

```
In [6]: sum([matrice1, matrice2, matrice1] , zero)

Out[6]: [12, 24, 36, 48]
```

C'est un effet de bord du typage dynamique. On ne vérifie pas *a priori* que tous les arguments passés à `sum` savent faire une addition ; *a contrario*, s'ils savent s'additionner on peut exécuter le code de `sum`.

De manière plus générale, si vous écrivez par exemple un morceau de code qui travaille sur les éléments d'un anneau (au sens anneau des entiers $\mathbb{Z}$) - imaginez un code qui factorise des polynômes - vous pouvez espérer utiliser ce code avec n'importe quel anneau, c'est à dire avec une classe qui implémente les 4 opérations (pourvu bien sûr que cet ensemble soit effectivement un anneau).

On peut aussi redéfinir un ordre

La place nous manque pour illustrer la possibilité, avec les opérateurs `__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, et `__ge__`, de redéfinir un ordre sur les instances d'une classe.

Signalons à cet égard qu'il existe un mécanisme "intelligent" qui permet de définir un ordre à partir d'un sous-ensemble seulement de ces méthodes, l'idée étant que si vous savez faire > et =, vous savez sûrement faire tout le reste. Ce mécanisme est documenté ici ; il repose sur **un décorateur** (`@total_ordering`), un mécanisme que nous étudierons en semaine 9, mais que vous pouvez utiliser dès à présent.

De manière analogue à `sum` qui fonctionne sur une liste de matrices, si on avait défini un ordre sur les matrices, on aurait pu alors utiliser les *built-in* `min` et `max` pour calculer une borne supérieure ou inférieure dans une séquence de matrices.

6.6.2 Complément - niveau avancé

Le produit avec un scalaire

On implémenterait la multiplication de deux matrices d'une façon identique (quoique plus fastidieuse naturellement).

La multiplication d'une matrice par un scalaire (un réel ou complexe pour fixer les idées), comme ici :

```
matrice2 = reel * matrice1
```

peut être également réalisée par surcharge de l'opérateur `__rmul__`.

Il s'agit d'une astuce, destinée précisément à ce genre de situations, où on veut étendre la classe de l'opérande de **droite**, sachant que dans ce cas précis l'opérande de gauche est un type de base, qu'on ne peut pas étendre (les classes *built-in* sont non mutables, pour garantir la stabilité de l'interpréteur).

Voici donc comment on s'y prendrait. Pour éviter de reproduire tout le code de la classe, on va l'étendre à la volée.

```
In [7]: # remarquez que les opérandes sont apparemment inversés
        # dans le sens où pour evaluer
        #     reel * matrice
        # on écrit une méthode qui prend en argument
        #   la matrice, puis le réel
        # mais n'oubliez pas qu'on est en fait en train
        # d'écrire une méthode sur la classe `Matrix2`
        def multiplication_scalaire(self, alpha):
            return Matrix2([alpha * coef for coef in self.coefs])

        # on ajoute la méthode spéciale __rmul__
        Matrix2.__rmul__ = multiplication_scalaire

In [8]: matrice1

Out[8]: [1, 2, 3, 4]

In [9]: 12 * matrice1

Out[9]: [12, 24, 36, 48]
```

6.7 Méthodes spéciales (2/3)

6.7.1 Complément - niveau avancé

Nous poursuivons dans ce complément la sélection de méthodes spéciales entreprise en première partie.

`__contains__`, `__len__`, `__getitem__` et apparentés

La méthode `__contains__` permet de donner un sens à :

```
item in objet
```

Sans grande surprise, elle prend en argument un objet et un item, et doit renvoyer un booléen. Nous l'illustrons ci-dessous avec la classe `DualQueue`.

La méthode `__len__` est utilisée par la fonction *built-in* `len` pour retourner la longueur d'un objet.

La classe `DualQueue`

Nous allons illustrer ceci avec un exemple de classe, un peu artificiel, qui implémente une queue de type FIFO. Les objets sont d'abord admis dans la file d'entrée (`add_input`), puis déplacés dans la file de sortie (`move_input_to_output`), et enfin sortis (`emit_output`).

Clairement, cet exemple est à but uniquement pédagogique ; on veut montrer comment une implémentation qui repose sur deux listes séparées peut donner l'illusion d'une continuité, et se présenter comme un container unique. De plus cette implémentation ne fait aucun contrôle pour ne pas obscurcir le code.

```
In [1]: class DualQueue:
            """Une double file d'attente FIFO"""

            def __init__(self):
                "constructeur, sans argument"
                self.inputs = []
                self.outputs = []

            def __repr__ (self):
                "affichage"
                return f"<DualQueue, inputs={self.inputs}, outputs={self.outputs}>"

            # la partie qui nous intéresse ici
            def __contains__(self, item):
                "appartenance d'un objet à la queue"
                return item in self.inputs or item in self.outputs

            def __len__(self):
                "longueur de la queue"
                return len(self.inputs) + len(self.outputs)

            # l'interface publique de la classe
```

```
            # le plus simple possible et sans aucun contrôle
            def add_input(self, item):
                "faire entrer un objet dans la queue d'entrée"
                self.inputs.insert(0, item)

            def move_input_to_output (self):
                "l'objet le plus ancien de la queue d'entrée est promu
                 dans la queue de sortie"
                self.outputs.insert(0, self.inputs.pop())

            def emit_output (self):
                "l'objet le plus ancien de la queue de sortie est émis"
                return self.outputs.pop()
```

```
In [2]: # on construit une instance pour nos essais
        queue = DualQueue()
        queue.add_input('zero')
        queue.add_input('un')
        queue.move_input_to_output()
        queue.move_input_to_output()
        queue.add_input('deux')
        queue.add_input('trois')

        print(queue)
```

```
<DualQueue, inputs=['trois', 'deux'], outputs=['un', 'zero']>
```

Longueur et appartenance

Avec cette première version de la classe `DualQueue` on peut utiliser `len` et le test d'appartenance :

```
In [3]: print(f'len() = {len(queue)}')

        print(f"deux appartient-il ? {'deux' in queue}")
        print(f"1 appartient-il ? {1 in queue}")
```

```
len() = 4
deux appartient-il ? True
1 appartient-il ? False
```

Accès séquentiel (accès par un index entier)

Lorsqu'on a la notion de longueur de l'objet avec `__len__`, il peut être opportun - quoique cela n'est pas imposé par le langage, comme on vient de le voir - de proposer également un accès indexé par un entier pour pouvoir faire :

```
queue[1]
```

Pour ne pas répéter tout le code de la classe, nous allons étendre `DualQueue`; pour cela nous définissons une fonction, que nous affectons ensuite à `DualQueue.__getitem__`, comme nous avons déjà eu l'occasion de le faire :

```
In [4]: # une première version de DualQueue.__getitem__
        # pour uniquement l'accès par index

        # on définit une fonction
        def dual_queue_getitem (self, index):
            "redéfinit l'accès [] séquentiel"

            # on vérifie que l'index a un sens
            if not (0 <= index < len(self)):
                raise IndexError(f"Mauvais indice {index} pour DualQueue")
            # on décide que l'index 0 correspond à l'élément le plus ancien
            # ce qui oblige à une petite gymnastique
            li = len(self.inputs)
            lo = len(self.outputs)
            if index < lo:
                return self.outputs[lo - index - 1]
            else:
                return self.inputs[li - (index-lo) - 1]

        # et on affecte cette fonction à l'intérieur de la classe
        DualQueue.__getitem__ = dual_queue_getitem
```

À présent, on peut **accéder** aux objets de la queue **séquentiellement** :

```
In [5]: print(queue[0])
```

```
zero
```

ce qui lève la même exception qu'avec une vraie liste si on utilise un mauvais index :

```
In [6]: try:
            print(queue[5])
        except IndexError as e:
            print('ERREUR',e)
```

```
ERREUR Mauvais indice 5 pour DualQueue
```

Amélioration : accès par slice

Si on veut aussi supporter l'accès par slice comme ceci :

```
queue[1:3]
```

il nous faut modifier la méthode `__getitem__`.

Le second argument de `__getitem__` correspond naturellement au contenu des crochets `[]`, on utilise donc `isinstance` pour écrire un code qui s'adapte au type d'indexation, comme ceci :

```
In [7]: # une deuxième version de DualQueue.__getitem__
        # pour l'accès par index et/ou par slice

        def dual_queue_getitem (self, key):
            "redéfinit l'accès par [] pour entiers, slices, et autres"

            # l'accès par slice queue[1:3]
            # nous donne pour key un objet de type slice
            if isinstance(key, slice):
                # key.indices donne les indices qui vont bien
                return [self[index] for index in range(*key.indices(len(self)))]

            # queue[3] nous donne pour key un entier
            elif isinstance(key, int):
                index = key
                # on vérifie que l'index a un sens
                if index < 0 or index >= len(self):
                    raise IndexError(f"Mauvais indice {index} pour DualQueue")
                # on décide que l'index 0 correspond à l'élément le plus ancien
                # ce qui oblige à une petite gymnastique
                li = len(self.inputs)
                lo = len(self.outputs)
                if index < lo:
                    return self.outputs[lo-index-1]
                else:
                    return self.inputs[li-(index-lo)-1]
            # queue ['foo'] n'a pas de sens pour nous
            else:
                raise KeyError(f"[] avec type non reconnu {key}")

        # et on affecte cette fonction à l'intérieur de la classe
        DualQueue.__getitem__ = dual_queue_getitem
```

Maintenant on peut accéder par slice :

```
In [8]: queue[1:3]

Out[8]: ['un', 'deux']
```

Et on reçoit bien une exception si on essaie d'accéder par clé :

```
In [9]: try:
            queue['key']
        except KeyError as e:
            print(f"OOPS: {type(e).__name__}: {e}")

OOPS: KeyError: '[] avec type non reconnu key'
```

L'objet est itérable (même sans avoir `__iter__`)

Avec seulement `__getitem__`, on peut **faire une boucle** sur l'objet queue. On l'a mentionné rapidement dans la séquence sur les itérateurs, mais la **méthode `__iter__` n'est pas la seule façon** de rendre un objet itérable :

```
In [10]: # grâce à __getitem__ on a rendu les
         # objets de type DualQueue itérables
         for item in queue:
            print(item)
```

```
zero
un
deux
trois
```

On peut faire un test sur l'objet

De manière similaire, même sans la méthode `__bool__`, cette classe sait **faire des tests de manière correcte** grâce uniquement à la méthode `__len__` :

```
In [11]: # un test fait directement sur la queue
         if queue:
             print(f"La queue {queue} est considérée comme True")
```

```
La queue <DualQueue, inputs=['trois', 'deux'], outputs=['un', 'zero']> est considérée
comme True
```

```
In [12]: # le même test sur une queue vide
         empty = DualQueue()

         # maintenant le test est négatif (notez bien le *not* ici)
         if not empty:
             print(f"La queue {empty} est considérée comme False")
```

```
La queue <DualQueue, inputs=[], outputs=[]> est considérée comme False
```

`__call__` **et les *callables***

Le langage introduit de manière similaire la notion de ***callable*** - littéralement, qui peut être appelé. L'idée est très simple, on cherche à donner un sens à un fragment de code du genre de :

```
# on crée une instance
objet = Classe(arguments)
```

et c'est l'objet (Attention : **l'objet, pas la classe**) qu'on utilise comme une fonction

```
objet(arg1, arg2)
```

Le protocole ici est très simple ; cette dernière ligne a un sens en Python dès lors que :

— `objet` possède une méthode `__call__` ;
— et que celle-ci peut être envoyée à `objet` avec les arguments `arg1, arg2` ;
— et c'est ce résultat qui sera alors retourné par `objet(arg1, arg2)` :

```
objet(arg1, arg2) ⟺ objet.__call__(arg1, arg2)
```

Voyons cela sur un exemple :

```
In [13]: class PlusClosure:
             """Une classe callable qui permet de faire un peu comme la
             fonction built-in sum mais en ajoutant une valeur initiale"""
             def __init__(self, initial):
                 self.initial = initial
             def __call__(self, *args):
                 return self.initial + sum(args)

         # on crée une instance avec une valeur initiale 2 pour la somme
         plus2 = PlusClosure (2)

In [14]: # on peut maintenant utiliser cet objet
         # comme une fonction qui fait sum(*arg)+2
         plus2()

Out[14]: 2

In [15]: plus2(1)

Out[15]: 3

In [16]: plus2(1, 2)

Out[16]: 5
```

Pour ceux qui connaissent, nous avons choisi à dessein un exemple qui s'apparente à une clôture. Nous reviendrons sur cette notion de *callable* lorsque nous verrons les décorateurs en semaine 9.

6.8 Méthodes spéciales (3/3)

6.8.1 Complément - niveau avancé

Ce complément termine la série sur les méthodes spéciales.

`__getattr__` et apparentés

Dans cette dernière partie nous allons voir comment avec la méthode `__getattr__`, on peut redéfinir la façon que le langage a d'évaluer :

```
objet.attribut
```

Avertissement : on a vu dans la séquence consacrée à l'héritage que, pour l'essentiel, le mécanisme d'héritage repose **précisément** sur la façon d'évaluer les attributs d'un objet, aussi nous vous recommandons d'utiliser ce trait avec précaution, car il vous donne la possibilité de "faire muter le langage" comme on dit.

Remarque : on verra en toute dernière semaine que `__getattr__` est *une* façon d'agir sur la façon dont le langage opère les accès aux attributs. Sachez qu'en réalité, le protocole d'accès aux attributs peut être modifié beaucoup plus profondément si nécessaire.

Un exemple : la classe `RPCProxy`

Pour illustrer `__getattr__`, nous allons considérer le problème suivant. Une application utilise un service distant, avec laquelle elle interagit au travers d'une API.

C'est une situation très fréquente : lorsqu'on utilise un service météo, ou de géolocalisation, ou de réservation, le prestataire vous propose une **API** (Application Programming Interface) qui se présente bien souvent comme une **liste de fonctions**, que votre fonction peut appeler à distance au travers d'un mécanisme de **RPC** (Remote Procedure Call).

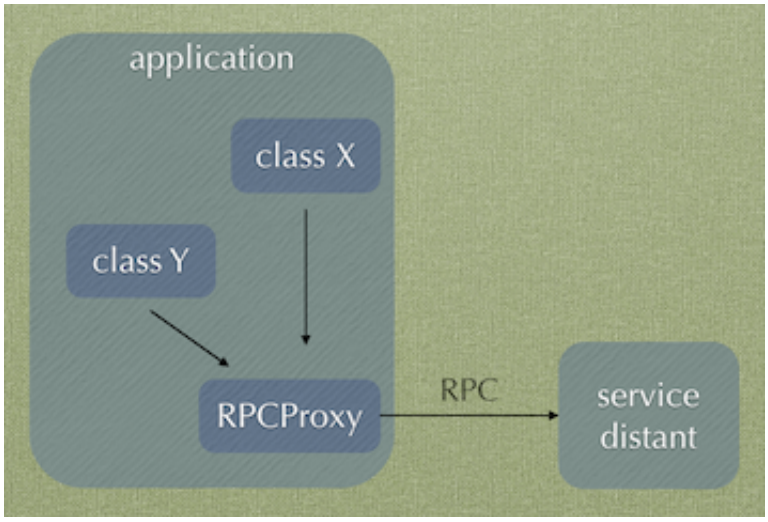
Imaginez pour fixer les idées que vous utilisez un service de réservation de ressources dans un Cloud, qui vous permet d'appeler les fonctions suivantes :

— `GetNodes(...)` pour obtenir des informations sur les noeuds disponibles ;
— `BookNode(...)` pour réserver un noeud ;
— `ReleaseNode(...)` pour abandonner un noeud.

Naturellement ceci est une API extrêmement simplifiée. Le point que nous voulons illustrer ici est que le dialogue avec le service distant :

— requiert ses propres données - comme l'URL où on peut joindre le service, et les identifiants à utiliser pour s'authentifier ;
— et possède sa propre logique - dans le cas d'une authentification par session par exemple, il faut s'authentifier une première fois avec un login/password, pour obtenir une session qu'on peut utiliser dans les appels suivants.

Pour ces raisons il est naturel de concevoir une classe `RPCProxy` dans laquelle on va rassembler à la fois ces données et cette logique, pour soulager toute l'application de ces détails, comme on l'a illustré ci-dessous :



Pour implémenter la plomberie liée à RPC, à l'encodage et décodage des données, et qui sera interne à la classe `RPCProxy`, on pourra en vraie grandeur utiliser des outils comme :

— `xmlrpc.client` qui fait partie de la bibliothèque standard ;
— ou, pour JSON, une des nombreuses implémentations qu'un moteur de recherche vous exposera si vous cherchez `python rpc json`, comme par exemple `json-rpc`.

Cela n'est toutefois pas notre sujet ici, et nous nous contenterons, dans notre code simplifié, d'imprimer un message.

Une approche naïve

Se pose donc la question de savoir quelle interface la classe `RPCProxy` doit offrir au reste du monde. Dans une première version naïve on pourrait écrire quelque chose comme :

```
In [1]: # la version naïve de la classe RPCProxy

        class RPCProxy:

            def __init__(self, url, login, password):
                self.url = url
                self.login = login
                self.password = password

            def _forward_call(self, functionname, *args):
                """
                helper method that marshalls and forwards
                the function and arguments to the remote end
                """
                print(f"""Envoi à {self.url}
        de la fonction {functionname} -- args= {args}""")
                return "retour de la fonction " + functionname

            def GetNodes (self, *args):
                return self._forward_call ('GetNodes', *args)
            def BookNode (self, *args):
                return self._forward_call ('BookNode', *args)
```

```
            def ReleaseNode (self, *args):
                return self._forward_call ('ReleaseNode', *args)
```

Ainsi l'application utilise la classe de cette façon :

```
In [2]: # création d'une instance de RPCProxy

        rpc_proxy = RPCProxy(url='http://cloud.provider.com/JSONAPI',
                             login='dupont',
                             password='***')

        # cette partie du code, en tant qu'utilisateur de l'API,
        # est supposée connaître les détails
        # des arguments à passer
        # et de comment utiliser les valeurs de retour
        nodes_list = rpc_proxy.GetNodes (
            [ ('phy_mem', '>=', '32G') ] )

        # réserver un noeud
        node_lease = rpc_proxy.BookNode (
            { 'id' : 1002, 'phy_mem' : '32G' } )

Envoi à http://cloud.provider.com/JSONAPI
de la fonction GetNodes -- args= ([('phy_mem', '>=', '32G')],)
Envoi à http://cloud.provider.com/JSONAPI
de la fonction BookNode -- args= ({'id': 1002, 'phy_mem': '32G'},)
```

Discussion

Quelques commentaires en vrac au sujet de cette approche :

— l'interface est correcte ; l'objet `rcp_proxy` se comporte bien comme un proxy, on a donné au programmeur l'illusion complète qu'il utilise une classe locale (sauf pour les performances bien entendu...) ;
— la séparation des rôles est raisonnable également, la classe RPCProxy n'a pas à connaître le détail de la signature de chaque méthode, charge à l'appelant d'utiliser l'API correctement ;
— par contre ce qui cloche, c'est que l'implémentation de la classe RPCProxy dépend de la liste des fonctions exposées par l'API ; imaginez une API avec 100 ou 200 méthodes, cela donne une dépendance assez forte et surtout inutile ;
— enfin, nous avons escamoté la nécessité de faire de RPCProxy un singleton, mais c'est une toute autre histoire.

Une approche plus subtile

Pour obtenir une implémentation qui conserve toutes les qualités de la version naïve, mais sans la nécessité de définir une à une toutes les fonctions de l'API, on peut tirer profit de `__getattr__`, comme dans cette deuxième version :

```
In [3]: # une deuxième implémentation de RPCProxy
```

```
class RPCProxy:

    def __init__(self, url, login, password):
        self.url = url
        self.login = login
        self.password = password

    def __getattr__(self, function):
        """
        Crée à la volée une méthode sur RPCProxy qui correspond
        à la fonction distante 'function'
        """
        def forwarder(*args):
            print(f"Envoi à {self.url}\nde la fonction {function} -- args= {args}")
            return "retour de la fonction " + function
        return forwarder
```

Qui est cette fois **totalement découplée** des détails de l'API, et qu'on peut utiliser exactement comme tout à l'heure :

```
In [4]: # création d'une instance de RPCProxy

        rpc_proxy = RPCProxy (url='http://cloud.provider.com/JSONAPI',
                              login='dupont',
                              password='***')

        # cette partie du code, en tant qu'utilisateur de l'API,
        # est supposée connaître les détails
        # des arguments à passer
        # et de comment utiliser les valeurs de retour
        nodes_list = rpc_proxy.GetNodes (
            [ ('phy_mem', '>=', '32G') ] )

        # réserver un noeud
        node_lease = rpc_proxy.BookNode (
            { 'id' : 1002, 'phy_mem' : '32G' } )
```

```
Envoi à http://cloud.provider.com/JSONAPI
de la fonction GetNodes -- args= ([('phy_mem', '>=', '32G')],)
Envoi à http://cloud.provider.com/JSONAPI
de la fonction BookNode -- args= ({'id': 1002, 'phy_mem': '32G'},)
```

6.9 Héritage

6.9.1 Complément - niveau basique

La notion d'héritage, qui fait partie intégrante de la Programmation Orientée Objet, permet principalement de maximiser la **réutilisabilité**.

Nous avons vu dans la vidéo les mécanismes d'héritage *in abstracto*. Pour résumer très brièvement, on recherche un attribut (pour notre propos, disons une méthode) à partir d'une instance en cherchant :

— d'abord dans l'instance elle-même ;
— puis dans la classe de l'instance ;
— puis dans les super-classes de la classe.

L'objet de ce complément est de vous donner, d'un point de vue plus appliqué, des idées de ce que l'on peut faire ou non avec ce mécanisme. Le sujet étant assez rabâché par ailleurs, nous nous concentrerons sur deux points :

— les pratiques de base utilisées pour la conception de classes, et notamment pour bien distinguer **héritage** et **composition** ;
— nous verrons ensuite, sur des **exemples extraits de code réel**, comment ces mécanismes permettent en effet de contribuer à la réutilisabilité du code.

Plusieurs formes d'héritage

Une méthode héritée peut être rangée dans une de ces trois catégories :

— *implicite* : si la classe fille ne définit pas du tout la méthode ;
— *redéfinie* : si on récrit la méthode entièrement ;
— *modifiée* : si on récrit la méthode dans la classe fille, mais en utilisant le code de la classe mère.

Commençons par illustrer tout ceci sur un petit exemple :

```
In [1]: # Une classe mère
        class Fleur:
            def implicite(self):
                print('Fleur.implicite')
            def redefinie(self):
                print('Fleur.redéfinie')
            def modifiee(self):
                print('Fleur.modifiée')

        # Une classe fille
        class Rose(Fleur):
            # on ne définit pas implicite
            # on rédéfinit complètement redefinie
            def redefinie(self):
                print('Rose.redefinie')
            # on change un peu le comportement de modifiee
            def modifiee(self):
                Fleur.modifiee(self)
                print('Rose.modifiee apres Fleur')
```

On peut à présent créer une instance de Rose et appeler sur cette instance les trois méthodes.

```
In [2]: # fille est une instance de Rose
        fille = Rose()

        fille.implicite()

Fleur.implicite


In [3]: fille.redefinie()

Rose.redefinie
```

S'agissant des deux premières méthodes, le comportement qu'on observe est simplement la conséquence de l'algorithme de recherche d'attributs : `implicite` est trouvée dans la classe Fleur et `redefinie` est trouvée dans la classe Rose.

```
In [4]: fille.modifiee()

Fleur.modifiée
Rose.modifiee apres Fleur
```

Pour la troisième méthode, attardons-nous un peu car on voit ici comment *combiner* facilement le code de la classe mère avec du code spécifique à la classe fille, en appelant explicitement la méthode de la classe mère lorsqu'on fait :

```
Fleur.modifiee(self)
```

La fonction *built-in* `super()`

Signalons à ce sujet, pour être exhaustif, l'existence de la fonction *built-in* `super()` qui permet de réaliser la même chose sans nommer explicitement la classe mère, comme on le fait ici :

```
In [5]: # Une version allégée de la classe fille, qui utilise super()
        class Rose(Fleur):
            def modifiee(self):
                super().modifiee()
                print('Rose.modifiee apres Fleur')

In [6]: fille = Rose()

        fille.modifiee()

Fleur.modifiée
Rose.modifiee apres Fleur
```

On peut envisager d'utiliser `super()` dans du code très abstrait où on ne sait pas déterminer statiquement le nom de la classe dont il est question. Mais, c'est une question de goût évidemment, je recommande personnellement la première forme, où on qualifie la méthode avec le nom de la classe mère qu'on souhaite utiliser. En effet, assez souvent :

— on sait déterminer le nom de la classe dont il est question ;
— ou alors on veut mélanger plusieurs méthodes héritées (via l'héritage multiple, dont on va parler dans un prochain complément) et dans ce cas `super()` ne peut rien pour nous.

Héritage *vs* Composition

Dans le domaine de la conception orientée objet, on fait la différence entre deux constructions, l'héritage et la composition, qui à une analyse superficielle peuvent paraître procurer des résultats similaires, mais qu'il est important de bien distinguer.

Voyons d'abord en quoi consiste la composition et pourquoi le résultat est voisin :

```
In [7]: # Une classe avec qui on n'aura pas de relation d'héritage
        class Tige:
            def implicite(self):
                print('Tige.implicite')
            def redefinie(self):
                print('Tige.redefinie')
            def modifiee(self):
                print('Tige.modifiee')

        # on n'hérite pas
        # mais on fait ce qu'on appelle une composition
        # avec la classe Tige
        class Rose:
            # mais pour chaque objet de la classe Rose
            # on va créer un objet de la classe Tige
            # et le conserver dans un champ
            def __init__(self):
                self.externe = Tige()

            # le reste est presque comme tout à l'heure
            # sauf qu'il faut definir implicite
            def implicite(self):
                self.externe.implicite()

            # on redéfinit complètement redefinie
            def redefinie(self):
                print('Rose.redefinie')

            def modifiee(self):
                self.externe.modifiee()
                print('Rose.modifiee apres Tige')
```

```
In [8]: # on obtient ici exactement le même comportement pour les trois sortes de méthodes
        fille = Rose()

        fille.implicite()
```

```
        fille.redefinie()
        fille.modifiee()

Tige.implicite
Rose.redefinie
Tige.modifiee
Rose.modifiee apres Tige
```

Comment choisir ?

Alors, quand faut-il utiliser l'héritage et quand faut-il utiliser la composition ?
On arrive ici à la limite de notre cours, il s'agit plus de conception que de codage à proprement parler, mais pour donner une réponse très courte à cette question :

— on utilise l'héritage lorsqu'un objet de la sous-classe **est aussi un** objet de la super-classe (une rose est aussi une fleur) ;
— on utilise la composition lorsqu'un objet de la sous-classe **a une relation avec** un objet de la super-classe (une rose possède une tige, mais c'est un autre objet).

6.9.2 Complément - niveau intermédiaire

Des exemples de code

Sans transition, dans cette section un peu plus prospective, nous vous avons signalé quelques morceaux de code de la bibliothèque standard qui utilisent l'héritage. Sans aller nécessairement jusqu'à la lecture de ces codes, il nous a semblé intéressant de commenter spécifiquement l'usage qui est fait de l'héritage dans ces bibliothèques.

Le module `calendar`

On trouve dans la bibliothèque standard le module `calendar`. Ce module expose deux classes `TextCalendar` et `HTMLCalendar`. Sans entrer du tout dans le détail, ces deux classes permettent d'imprimer dans des formats différents le même type d'informations.

Le point ici est que les concepteurs ont choisi un graphe d'héritage comme ceci :

```
Calendar
|-- TextCalendar
|-- HTMLCalendar
```

qui permet de grouper le code concernant la logique dans la classe `Calendar`, puis dans les deux sous-classes d'implémenter le type de sortie qui va bien.

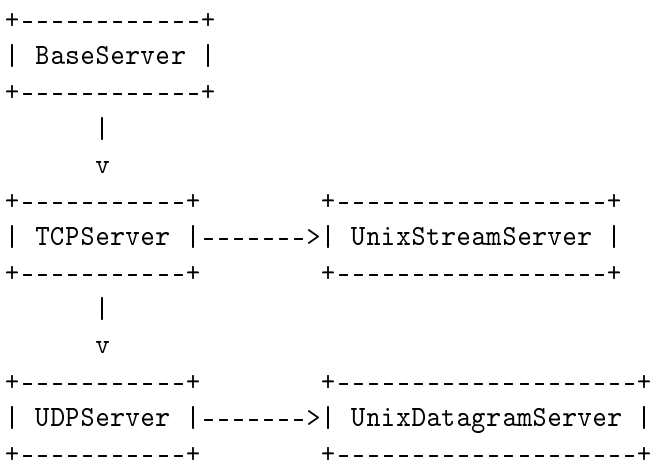
C'est l'utilisateur qui choisit la classe qui lui convient le mieux, et de cette manière, le maximum de code est partagé entre les deux classes ; et de plus si vous avez besoin d'une sortie au format, disons PDF, vous pouvez envisager d'hériter de `Calendar` et de n'implémenter que la partie spécifique au format PDF.

C'est un peu le niveau élémentaire de l'héritage.

Le module `SocketServer`

Toujours dans la bibliothèque standard, le module `SocketServer` fait un usage beaucoup plus sophistiqué de l'héritage.

Le module propose une hiérarchie de classes comme ceci :



Ici encore notre propos n'est pas d'entrer dans les détails, mais d'observer le fait que les différents *niveaux d'intelligence* sont ajoutés les uns aux les autres au fur et à mesure que l'on descend le graphe d'héritage.

Cette hiérarchie est par ailleurs étendue par le module `http.server` et notamment au travers de la classe `HTTPServer` qui hérite de `TCPServer`.

Pour revenir au module `SocketServer`, j'attire votre attention dans la page d'exemples sur cet exemple en particuler, où on crée une classe de serveurs multi-threads (la classe `ThreadedTCPServer`) par simple héritage multiple entre `ThreadingMixIn` et `TCPServer`. La notion de `Mixin` est décrite dans cette page Wikipédia dans laquelle vous pouvez accéder directement à la section consacrée à Python.

6.10 Hériter des types *built-in* ?

6.10.1 Complément - niveau avancé

Vous vous demandez peut-être s'il est possible d'hériter des types *built-in*.

La réponse est oui, et nous allons voir un exemple qui est parfois très utile en pratique, c'est le type - ou plus exactement la famille de types - `namedtuple`

La notion de *record*

On se place dans un contexte voisin de celui de *record* - en français enregistrement - qu'on a déjà rencontré souvent ; pour ce notebook nous allons à nouveau prendre le cas du point à deux coordonnées x et y. Nous avons déjà vu que pour implémenter un point on peut utiliser :

un dictionnaire

```
In [1]: p1 = {'x': 1, 'y': 2}
        # ou de manière équivalente
        p1 = dict(x=1, y=2)
```

ou une classe

```
In [2]: class Point:
            def __init__(self, x, y):
                self.x = x
                self.y = y

        p2 = Point(1, 2)
```

Nous allons voir une troisième façon de s'y prendre, qui présente deux caractéristiques :

— les objets seront non-mutables (en fait ce sont des tuples) ;
— et accessoirement on pourra accéder aux différents champs par leur nom aussi bien que par un index.

Pous faire ça il nous faut donc créer une sous-classe de `tuple` ; pour nous simplifier la vie, le module `collections` nous offre un utilitaire :

`namedtuple`

```
In [3]: from collections import namedtuple
```

Techniquement, il s'agit d'une fonction :

```
In [4]: type(namedtuple)
```

```
Out[4]: function
```

qui **renvoie une classe** - oui les classes sont des objets comme les autres ; par exemple pour créer une classe `TuplePoint`, on ferait :

```
In [5]: # on passe à namedtuple
        #  - le nom du type qu'on veut créer
        #  - la liste ordonnée des composants (champs)
        TuplePoint = namedtuple('TuplePoint', ['x', 'y'])
```

Et maintenant si je crée un objet :

```
In [6]: p3 = TuplePoint(1, 2)

In [7]: # cet objet est un tuple
        isinstance(p3, tuple)

Out[7]: True

In [8]: # auquel je peux accéder par index
        # comme un tuple
        p3[0]

Out[8]: 1

In [9]: # mais aussi par nom via un attribut
        p3.x

Out[9]: 1

In [10]: # et comme c'est un tuple il est immuable
         try:
             p3.x = 10
         except Exception as e:
             print(f"OOPS {type(e)} {e}")

OOPS <class 'AttributeError'> can't set attribute
```

À quoi ça sert

Les `namedtuple` ne sont pas d'un usage fréquent, mais on en a déjà rencontré un exemple dans le notebook sur le module `pathlib`. En effet le type de retour de la méthode `Path.stat` est un `namedtuple` :

```
In [11]: from pathlib import Path
         dot_stat = Path('.').stat()

In [12]: dot_stat

Out[12]: os.stat_result(st_mode=16895, st_ino=844424930203143, st_dev=369767116,
         st_nlink=1, st_uid=0, st_gid=0, st_size=28672, st_atime=1539778506,
         st_mtime=1539778506, st_ctime=1539334282)

In [13]: isinstance(dot_stat, tuple)

Out[13]: True
```

Nom

Quand on crée une classe avec l'instruction `class`, on ne mentionne le nom de la classe qu'une seule fois. Ici vous avez remarqué qu'il faut en pratique le donner deux fois. Pour être précis, le paramètre qu'on a passé à `namedtuple` sert à ranger le nom dans l'attribut `__name__` de la classe créée :

```
In [14]: Foo = namedtuple('Bar', ['spam', 'eggs'])

In [15]: # Foo est le nom de la variable classe
         foo = Foo(1, 2)

In [16]: # mais cette classe a son attribut __name__ mal positionné
         Foo.__name__

Out[16]: 'Bar'
```

Il est donc évidemment préférable d'utiliser deux fois le même nom..

Mémoire

À titre de comparaison voici la place prise par chacun de ces objets ; le `namedtuple` ne semble pas de ce point de vue spécialement attractif par rapport à une instance :

```
In [17]: import sys

         # p1 = dict / p2 = instance / p3 = namedtuple

         for p in p1, p2, p3:
             print(sys.getsizeof(p))

136
32
36
```

Définir des méthodes sur un `namedtuple`

Dans un des compléments de la séquence précédente, intitulé *"Manipuler des ensembles d'instances"*, nous avions vu comment redéfinir le protocole *hashable* sur des instances, en mettant en évidence la nécessité de disposer d'instances non mutables lorsqu'on veut redéfinir `__hash__()`.

Voyons ici comment on pourrait tirer parti d'un `namedtuple` pour refaire proprement notre classe `Point2` - souvenez-vous, il s'agissait de rechercher dans un ensemble de points.

```
In [18]: Point2 = namedtuple('Point2', ['x', 'y'])
```

Sans utiliser le mot-clé `class`, il faudrait se livrer à une petite gymnastique pour redéfinir les méthodes spéciales sur la classe `Point2`. Nous allons utiliser l'héritage pour arriver au même résultat :

```
In [19]: # ce code est très proche du code utilisé dans le précédent complément
         class Point2(namedtuple('Point2', ['x', 'y'])):

             # l'égalité va se baser naturellement sur x et y
             def __eq__(self, other):
                 return self.x == other.x and self.y == other.y

             # du coup la fonction de hachage
             # dépend aussi de x et de y
             def __hash__(self):
                 return (11 * self.x + self.y) // 16
```

Avec ceci en place on peut maintenant faire :

```
In [20]: # trois points égaux au sens de cette classe
         q1, q2, q3 = Point2(10, 10), Point2(10, 10), Point2(10, 10)

In [21]: # deux objets distincts
         q1 is q2

Out[21]: False

In [22]: # mais égaux
         q1 == q2

Out[22]: True

In [23]: # ne font qu'un dans un ensemble
         s = {q1, q2}
         len(s)

Out[23]: 1

In [24]: # et on peut les trouver
         # par le troisiéme
         q3 in s

Out[24]: True

In [25]: # et les instances ne sont pas mutables
         try:
             q1.x = 100
         except Exception as e:
             print(f"OOPS {type(e)}")

OOPS <class 'AttributeError'>
```

6.11 Pour en savoir plus

Vous pouvez vous reporter à la documentation officielle.

Si vous êtes intéressés de savoir comment on peut bien arriver à rendre les objets d'une classe immuable, vous pouvez commencer par regarder le code utilisé par `namedtuple` pour créer son résultat, en l'invoquant avec le mode bavard (cette possibilité a disparu malheureusement dans python-3.7).

Vous y remarquerez notamment :

— une redéfinition de la méthode spéciale `__new__`,

— et aussi un usage des `property` que l'on a rencontrés en début de semaine.

```
In [26]: # exécuter ceci pour voir le détail de ce que fait `namedtuple`
         Point = namedtuple('Point', ['x', 'y'], verbose=True)

from builtins import property as _property, tuple as _tuple
from operator import itemgetter as _itemgetter
from collections import OrderedDict

class Point(tuple):
    'Point(x, y)'

    __slots__ = ()

    _fields = ('x', 'y')

    def __new__(_cls, x, y):
        'Create new instance of Point(x, y)'
        return _tuple.__new__(_cls, (x, y))

    @classmethod
    def _make(cls, iterable, new=tuple.__new__, len=len):
        'Make a new Point object from a sequence or iterable'
        result = new(cls, iterable)
        if len(result) != 2:
            raise TypeError('Expected 2 arguments, got %d' % len(result))
        return result

    def _replace(_self, **kwds):
        'Return a new Point object replacing specified fields with new values'
        result = _self._make(map(kwds.pop, ('x', 'y'), _self))
        if kwds:
            raise ValueError('Got unexpected field names: %r' % list(kwds))
        return result

    def __repr__(self):
        'Return a nicely formatted representation string'
        return self.__class__.__name__ + '(x=%r, y=%r)' % self

    def _asdict(self):
        'Return a new OrderedDict which maps field names to their values.'
        return OrderedDict(zip(self._fields, self))

    def __getnewargs__(self):
```

```
    'Return self as a plain tuple.  Used by copy and pickle.'
    return tuple(self)

x = _property(_itemgetter(0), doc='Alias for field number 0')

y = _property(_itemgetter(1), doc='Alias for field number 1')
```

6.12 dataclasses

Nouveauté de la version 3.7

Python 3.7 apporte un nouveauté pour simplifier la définition de classes dites “de données” ; ce type de classes s’applique pour des objets qui sont essentiellement un assemblage de quelques champs de données.

Comme cette capacité n’est disponible qu’à partir de Python 3.7 et que le cours est basé sur Python 3.6, nous n’aurons pas la possibilité de manipuler directement ce nouveau concept. Voici toutefois quelques exemples pour vous donner un aperçu de ce qu’on peut faire de cette notion.

Aperçu

La raison d’être de `dataclass` est de fournir - encore un - moyen de définir des classes d’enregistrements.

Voici par exemple comment on pourrait définir une classe `Personne` :

```
>>> from dataclasses import dataclass
>>>
>>> @dataclass
... class Personne:
...     nom: str
...     age: int
...     email: str = ""
...
>>> personne = Personne(nom='jean', age=12)
>>>
>>> print(personne)
Personne(nom='jean', age=12, email='')
>>>
```

Instances non mutables

Le décorateur `dataclass` accepte divers arguments pour choisir le comportement de certains aspects de la classe. Reportez-vous à la documentation pour une liste complète, mais voici un exemple qui utilise `frozen=True` et qui illustre la possibilité de créer des instances non mutables. Nous retrouvons ici le même scénario d’ensemble de points que nous avons déjà rencontré plusieurs fois :

```
>>> import sys; print(sys.version)
3.7.0 (default, Jun 29 2018, 20:14:27)
[Clang 9.0.0 (clang-900.0.39.2)]

>>> from dataclasses import dataclass
>>>
>>> @dataclass(frozen=True)
... class Point:
...     x: float
...     y: float
...
```

```
...     def __eq__(self, other):
...         return self.x == other.x and self.y == other.y
...
...     def __hash__(self):
...         return (11 * self.x + self.y) // 16
...
>>> p1, p2, p3 = Point(1, 1), Point(1, 1), Point(1, 1)
>>>
>>> s = {p1, p2}
>>> len(s)
1
>>>
>>> p3 in s
True
>>>
>>> try:
...     p1.x = 10
... except Exception as e:
...     print(f"OOPS {type(e)}")
...
OOPS <class 'dataclasses.FrozenInstanceError'>
```

Pour aller plus loin

Vous pouvez vous rapporter

— au PEP 557 qui a abouti au consensus, et
— à la documentation officielle du module.

6.13 Énumérations

6.13.1 Complément - niveau basique

On trouve dans d'autres langages la notion de types énumérés.

L'usage habituel, c'est typiquement un code d'erreur qui peut prendre certaines valeurs précises. Pensez par exemple aux codes prévus par le protocole HTTP. Le protocole prévoit un code de retour qui peut prendre un ensemble fini de valeurs, comme par exemple 200, 301, 302, 404, 500, mais pas 90 ni 110.

On veut pouvoir utiliser des noms parlants dans les programmes qui gèrent ce type de valeurs, c'est une application typique des types énumérés.

La bibliothèque standard offre depuis Python-3.4 un module qui s'appelle sans grande surprise `enum`, et qui expose entre autres une classe `Enum`. On l'utiliserait comme ceci, dans un cas d'usage plus simple :

```
In [1]: from enum import Enum

In [2]: class Flavour(Enum):
            CHOCOLATE = 1
            VANILLA = 2
            PEAR = 3

In [3]: vanilla = Flavour.VANILLA
```

Un premier avantage est que les représentations textuelles sont plus parlantes :

```
In [4]: str(vanilla)

Out[4]: 'Flavour.VANILLA'

In [5]: repr(vanilla)

Out[5]: '<Flavour.VANILLA: 2>'
```

Vous pouvez aussi retrouver une valeur par son nom :

```
In [6]: chocolate = Flavour['CHOCOLATE']
        chocolate

Out[6]: <Flavour.CHOCOLATE: 1>

In [7]: Flavour.CHOCOLATE

Out[7]: <Flavour.CHOCOLATE: 1>
```

Et réciproquement :

```
In [8]: chocolate.name

Out[8]: 'CHOCOLATE'
```

`IntEnum`

En fait, le plus souvent on préfère utiliser `IntEnum`, une sous-classe de `Enum` qui permet également de faire des comparaisons. Pour reprendre le cas des codes d'erreur HTTP :

```
In [9]: from enum import IntEnum

        class HttpError(IntEnum):

            OK = 200
            REDIRECT = 301
            REDIRECT_TMP = 302
            NOT_FOUND = 404
            INTERNAL_ERROR = 500

            # avec un IntEnum on peut faire des comparaisons
            def is_redirect(self):
                return 300 <= self.value <= 399

In [10]: code = HttpError.REDIRECT_TMP

In [11]: code.is_redirect()

Out[11]: True
```

Pour en savoir plus

Consultez la documentation officielle du module `enum`.

6.14 Héritage, typage

6.14.1 Complément - niveau avancé

Dans ce complément, nous allons revenir sur la notion de *duck typing*, et attirer votre attention sur cette différence assez essentielle entre python et les langages statiquement typés. On s'adresse ici principalement à ceux d'entre vous qui sont habitués à C++ et/ou Java.

Type concret et type abstrait

Revenons sur la notion de type et remarquons que les types peuvent jouer plusieurs rôles, comme on l'a évoqué rapidement en première semaine ; et pour reprendre des notions standard en langages de programmation nous allons distinguer deux types.

1. **type concret :** d'une part, la notion de type a bien entendu à voir avec l'implémentation ; par exemple, un compilateur C a besoin de savoir très précisément quel espace allouer à une variable, et l'interpréteur python sous-traite à la classe le soin d'initialiser un objet ;
2. **type abstrait :** d'autre part, les types sont cruciaux dans les systèmes de vérification statique, au sens large, dont le but est de trouver un maximum de défauts à la seule lecture du code (par opposition aux techniques qui nécessitent de le faire tourner).

Duck typing

En python, ces deux aspects du typage sont relativement décorrélés.

Pour la seconde dimension du typage, le système de types abstraits de python est connu sous le nom de *duck typing*, une appellation qui fait référence à cette phrase :

```
When I see a bird that walks like a duck and swims like a duck and quacks like
a duck, I call that bird a duck.
```

L'exemple des itérables

Pour prendre l'exemple sans doute le plus représentatif, la notion d'*itérable* est un type abstrait, en ce sens que, pour que le fragment :

```
for item in container:
    do_something(item)
```

ait un sens, il faut et il suffit que `container` soit un itérable. Et vous connaissez maintenant plein d'exemples très différents d'objets itérables, a minima parmi les *built-in* `str`, `list`, `tuple`, `range`...

Dans un langage typé statiquement, pour pouvoir donner un type à cette construction, on serait **obligé** de définir un type - qu'on appellerait logiquement une classe abstraite - dont ces trois types seraient des descendants.

En python, et c'est le point que nous voulons souligner dans ce complément, il n'existe pas dans le système python d'objet de type `type` qui matérialise l'ensemble des `iterables`. Si on regarde les superclasses de nos types concrets itérables, on voit que leur seul ancêtre commun est la classe `object` :

```
In [1]: str.__bases__
```

```
Out[1]: (object,)

In [2]: list.__bases__

Out[2]: (object,)

In [3]: tuple.__bases__

Out[3]: (object,)

In [4]: range.__bases__

Out[4]: (object,)
```

Un autre exemple

Pour prendre un exemple plus simple, si je considère :

```
def foo(graphic):
    ...
    graphic.draw()
```

pour que l'expression `graphic.draw()` ait un sens, il faut et il suffit que l'objet `graphic` ait une méthode `draw`.

À nouveau, dans un langage typé statiquement, on serait amené à définir une classe abstraite `Graphic`. En python ce n'est **pas requis** ; vous pouvez utiliser ce code tel quel avec deux classes `Rectangle` et `Texte` qui n'ont pas de rapport entre elles - autres que, à nouveau, d'avoir `object` comme ancêtre commun - pourvu qu'elles aient toutes les deux une méthode `draw`.

Héritage et type abstrait

Pour résumer, en python comme dans les langages typés statiquement, on a bien entendu la bonne propriété que si, par exemple, la classe `Spam` est itérable, alors la classe `Eggs` qui hérite de `Spam` est itérable.

Mais dans l'autre sens, si `Foo` et `Bar` sont itérables, il n'y a pas forcément une superclasse commune qui représente l'ensemble des objets itérables.

`isinstance` sur stéroïdes

D'un autre côté, c'est très utile d'exposer au programmeur un moyen de vérifier si un objet a un *type* donné - dans un sens volontairement vague ici.

On a déjà parlé en Semaine 4 de l'intérêt qu'il peut y avoir à tester le type d'un argument avec `isinstance` dans une fonction, pour parvenir à faire l'équivalent de la surcharge en C++ (la surcharge en C++, c'est quand vous définissez plusieurs fonctions qui ont le même nom mais des types d'arguments différents).

C'est pourquoi, quand on a cherché à exposer au programmeur des propriétés comme "cet objet est-il iterable ?", on a choisi d'étendre *isinstance* au travers de cette initiative. C'est ainsi qu'on peut faire par exemple :

```
In [5]: from collections.abc import Iterable
```

```
In [6]: isinstance('ab', Iterable)

Out[6]: True

In [7]: isinstance([1, 2], Iterable)

Out[7]: True

In [8]: # comme on l'a vu, un objet qui a des méthodes
        # __iter__() et __next__()
        # est considéré comme un itérable
        class Foo:
            def __iter__(self):
                return self
            def __next__(self):
                # ceci naturellement est bidon
                return

In [9]: foo = Foo()
        isinstance(foo, Iterable)

Out[9]: True
```

L'implémentation du module `abc` donne l'**illusion** que `Iterable` est un objet dans la hiérarchie de classes, et que tous ces *classes* `str`, `list`, et `Foo` lui sont asujetties, mais ce n'est pas le cas en réalité ; comme on l'a vu plus tôt, ces trois types ne sont pas comparables dans la hiérarchie de classes, ils n'ont pas de plus petit (ou plus grand) élément à part `object`.

Je signale pour finir, à propos de `isinstance` et du module `collections`, que la définition du symbole `Hashable` est à mon avis beaucoup moins convaincante que `Iterable` ; si vous vous souvenez qu'en Semaine 3, Séquence "les dictionnaires", on avait vu que les clés doivent être globalement immuables. C'est une caractéristique qui est assez difficile à écrire, et en tous cas ceci de mon point de vue ne remplit pas la fonction :

```
In [10]: from collections.abc import Hashable

In [11]: # un tuple qui contient une liste ne convient
         # pas comme clé dans un dictionnaire
         # et pourtant
         isinstance (([1], [2]), Hashable)

Out[11]: True
```

python et les classes abstraites

Les points à retenir de ce complément un peu digressif sont :

— en python, on hérite des **implémentations** et pas des **spécifications** ;
— et le langage n'est pas taillé pour tirer profit de **classes abstraites** - même si rien ne vous interdit d'écrire, pour des raisons documentaires, une classe qui résume l'interface qui est attendue par tel ou tel système de plugin.

Venant de C++ ou de Java, cela peut prendre du temps d'arriver à se débarrasser de l'espèce de réflexe qui fait qu'on pense d'abord classe abstraite, puis implémentations.

Pour aller plus loin

La documentation du module `collections.abc` contient la liste de tous les symboles exposés par ce module, dont par exemple en vrac :

— `Iterable`
— `Iterator`
— `Hashable`
— `Generator`
— `Coroutine` (rendez-vous semaine 8)

et de nombreux autres.

Avertissement

Prenez garde enfin que ces symboles n'ont - à ce stade du moins - pas de relation forte avec ceux du module `typing` dont on a parlé lorsqu'on a vu les *type hints*.

6.15 Héritage multiple

6.15.1 Complément - niveau intermédiaire

La classe `object`

Le symbole `object` est une variable prédéfinie (qui donc fait partie du module `builtins`) :

```
In [1]: object

Out[1]: object

In [2]: import builtins

        builtins.object is object

Out[2]: True
```

La classe `object` est une classe spéciale ; toutes les classes en Python héritent de la classe `object`, même lorsqu'aucun héritage n'est spécifié :

```
In [3]: class Foo:
            pass

        Foo.__bases__

Out[3]: (object,)
```

L'attribut spécial `__bases__`, comme on le devine, nous permet d'accéder aux superclasses directes, ici de la classe `Foo`.

En Python moderne, on n'a **jamais besoin de mentionner** `object` dans le code. La raison de sa présence dans les symboles prédéfinis est liée à l'histoire de Python, et à la distinction que faisait Python 2 entre classes *old-style* et classes *new-style*. Nous le mentionnons seulement car on rencontre encore parfois du code qui fait quelque chose comme :

```
In [4]: class Bar(object):
            pass
```

qui est un reste de Python 2, et que Python 3 accepte uniquement au titre de la compatibilité.

6.15.2 Complément - niveau avancé

Rappels

L'héritage en Python consiste principalement en l'algorithme de recherche d'un attribut d'une instance ; celui-ci regarde :

1. d'abord dans l'instance ;
2. ensuite dans la classe ;
3. ensuite dans les super-classes.

Ordre sur les super-classes

Le problème revient donc, pour le dernier point, à définir un **ordre** sur l'ensemble des **super-classes**. On parle bien, naturellement, de **toutes** les super-classes, pas seulement celles dont on hérite directement - en termes savants on dirait qu'on s'intéresse à la fermeture transitive de la relation d'héritage.

L'algorithme utilisé pour cela depuis la version 2.3 est connu sous le nom de **linéarisation C3**. Cet algorithme n'est pas propre à python, comme vous pourrez le lire dans les références citées dans la dernière section.

Nous ne décrirons pas ici l'algorithme lui-même dans le détail ; par contre nous allons :

— dans un premier temps résumer **les raisons** qui ont guidé ce choix, en décrivant les bonnes propriétés que l'on attend, ainsi que les **limitations** qui en découlent ;
— puis voir l'ordre obtenu sur quelques **exemples** concrets de hiérarchies de classes.

Vous trouverez dans les références (voir ci-dessous la dernière section, "Pour en savoir plus") des liens vers des documents plus techniques si vous souhaitez creuser le sujet.

Les bonnes propriétés attendues

Il y a un certain nombre de bonnes propriétés que l'on attend de cet algorithme.

Priorité au spécifique

Lorsqu'une classe A hérite d'une classe B, on s'attend à ce que les méthodes définies sur A, qui sont en principe plus spécifiques, soient utilisées de préférence à celles définies sur B.

Priorité à gauche

Lorsqu'on utilise l'héritage multiple, on mentionne les classes mères dans un certain ordre, qui n'est pas anodin. Les classes mentionnées en premier sont bien entendu celles desquelles on veut hériter en priorité.

6.16 La Method Resolution Order (MRO)

De manière un peu plus formelle

Pour reformuler les deux points ci-dessus, on s'intéresse à la `mro` d'une classe O, et on veut avoir les deux bonnes propriétés suivantes :

— si O hérite (pas forcément directement) de A qui elle même hérite de B, alors A est avant B dans la `mro` de O ;
— si O hérite (pas forcément directement) de A, qui elle hérite de B, puis (pas forcément immédiatement) de C, alors dans la `mro` A est avant B qui est avant C.

Limitations : toutes les hiérarchies ne peuvent pas être traitées

L'algorithme C3 permet de calculer un ordre sur $\mathcal{S}$ qui respecte toutes ces contraintes, lorsqu'il en existe un.

En effet, dans certains cas on ne peut pas trouver un tel ordre, on le verra plus bas, mais dans la pratique, il est assez rare de tomber sur de tels cas pathologiques ; et lorsque cela se produit c'est en général le signe d'erreurs de conception plus profondes.

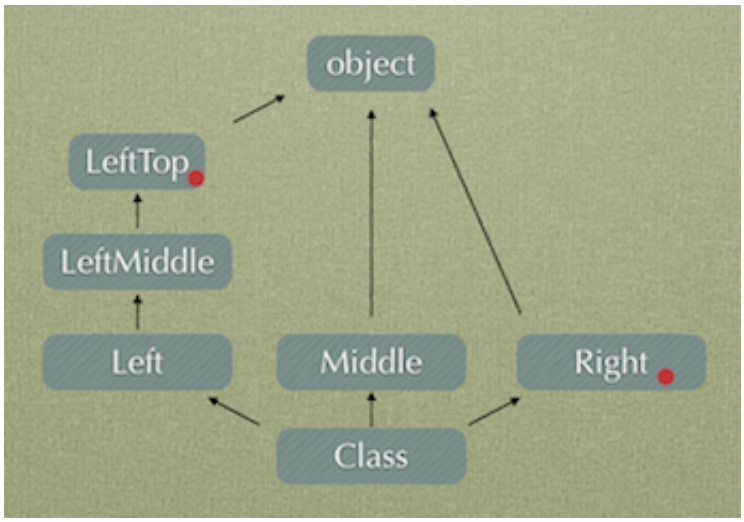
Un exemple très simple

On se donne la hiérarchie suivante :

```
In [5]: class LeftTop(object):
            def attribut(self):
                return "attribut(LeftTop)"

        class LeftMiddle(LeftTop):
            pass

        class Left(LeftMiddle):
            pass

        class Middle(object):
            pass

        class Right(object):
            def attribut(self):
                return "attribut(Right)"

        class Class(Left, Middle, Right):
            pass

        instance = Class()
```

qui donne en version dessinée, avec deux points rouges pour représenter les deux définitions de la méthode `attribut` :

Les deux règles, telles que nous les avons énoncées en premier lieu (priorité à gauche, priorité au spécifique) sont un peu contradictoires ici. En fait, c'est la méthode de `LeftTop` qui est héritée dans `Class`, comme on le voit ici :




```
In [6]: instance.attribut() == 'attribut(LeftTop)'
```

```
Out[6]: True
```

Exercice : Remarquez qu'ici `Right` a elle-même un héritage très simple. À titre d'exercice, modifiez le code ci-dessus pour faire que `Right` hérite de la classe `LeftMiddle`; de quelle classe d'après vous est-ce que `Class` hérite `attribut` dans cette configuration ?

Si cela ne vous convient pas

C'est une évidence, mais cela va peut-être mieux en le rappelant : si la méthode que vous obtenez "gratuitement" avec l'héritage n'est pas celle qui vous convient, vous avez naturellement toujours la possibilité de la redéfinir, et ainsi d'en **choisir** une autre. Dans notre exemple si on préfère la méthode implémentée dans `Right`, on définira plutôt la classe `Class` comme ceci :

```
In [7]: class Class(Left, Middle, Right):
            # en redéfinissant explicitement la méthode
            # attribut ici on court-circuite la mro
            # et on peut appeler explicitement une autre
            # version de attribut()
            def attribut(*args, **kwds):
                return Right.attribut(*args, **kwds)

        instance2 = Class()
        instance2.attribut()
```

```
Out[7]: 'attribut(Right)'
```

Ou encore bien entendu, si dans votre contexte vous devez appelez **les deux** méthodes dont vous pourriez hériter et les combiner, vous pouvez le faire aussi, par exemple comme ceci :

```
In [8]: class Class(Left, Middle, Right):
            # pour faire un composite des deux méthodes
            # trouvées dans les classes mères
            def attribut(*args, **kwds):
                return LeftTop.attribut(*args, **kwds) +
                 " ** " + Right.attribut(*args, **kwds)

        instance3 = Class()
        instance3.attribut()
```

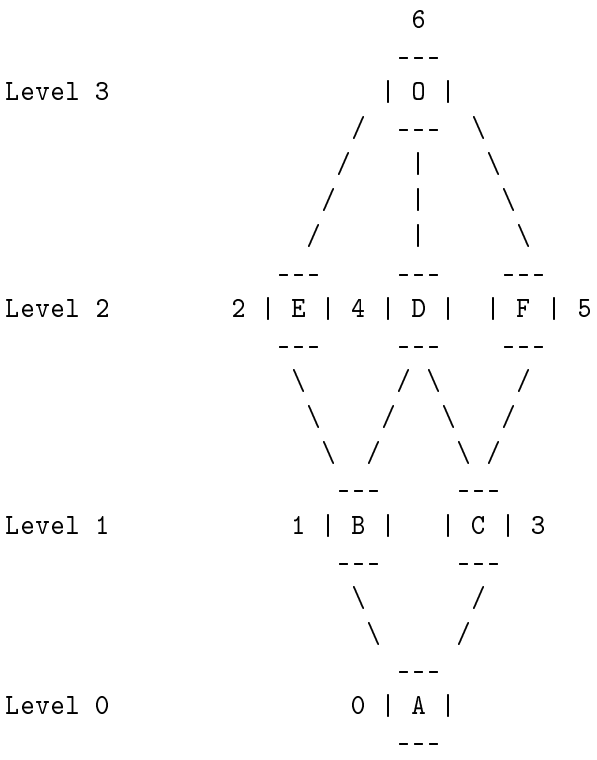
```
Out[8]: 'attribut(LeftTop) ** attribut(Right)'
```

Un exemple un peu plus compliqué

Voici un exemple, assez parlant, tiré de la deuxième référence (voir ci-dessous la dernière section, "Pour en savoir plus").

```
In [9]: O = object
        class F(O): pass
        class E(O): pass
        class D(O): pass
        class C(D, F): pass
        class B(E, D): pass
        class A(B, C): pass
```

Cette hiérarchie nous donne, en partant de A, l'ordre suivant :



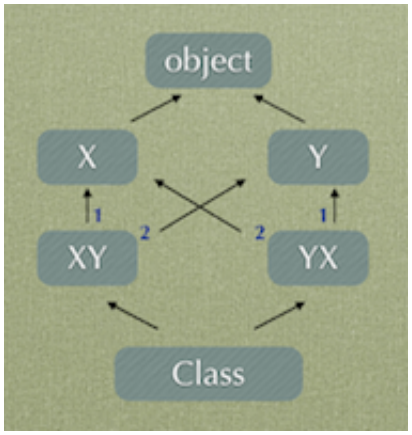
Que l'on peut calculer, sous l'interpréteur python, avec la méthode `mro` sur la classe de départ :

```
In [10]: A.mro()
```

```
Out[10]: [__main__.A,
          __main__.B,
          __main__.E,
          __main__.C,
          __main__.D,
          __main__.F,
          object]
```

Un exemple qui ne peut pas être traité

Voici enfin un exemple de hiérarchie pour laquelle on ne **peut pas trouver d'ordre** qui respecte les bonnes propriétés que l'on a vues tout à l'heure, et qui pour cette raison sera **rejetée par l'interpréteur python**. D'abord en version dessinée :




```
In [11]: # puis en version code
         class X: pass
         class Y: pass
         class XY(X, Y): pass
         class YX(Y, X): pass

         # on essaie de créer une sous-classe de XY et YX
         try:
             class Class(XY, YX): pass
         # mais ce n'est pas possible
         except Exception as e:
             print(f"OOPS, {type(e)}, {e}")
```

```
OOPS, <class 'TypeError'>, Cannot create a consistent method resolution
order (MRO) for bases X, Y
```

Pour en savoir plus

0. Un blog de Guido Van Rossum qui retrace l'historique des différents essais qui ont été faits avant de converger sur le modèle actuel.
1. Un article technique qui décrit le fonctionnement de l'algorithme de calcul de la MRO, et donne des exemples.
2. L'article de Wikipedia sur l'algorithme C3.

6.17 Les attributs

6.17.1 Compléments - niveau basique

La notation . et les attributs

La notation `module.variable` que nous avons vue dans la vidéo est un cas particulier de la notion d'attribut, qui permet d'étendre un objet, ou si on préfère de lui accrocher des données.

Nous avons déjà rencontré ceci de nombreuses fois à présent, c'est exactement le même mécanisme d'attribut qui est utilisé pour les méthodes ; pour le système d'attribut il n'y a pas de différence entre `module.variable`, `module.fonction`, `objet.methode`, etc.

Nous verrons très bientôt que ce mécanisme est massivement utilisé également dans les instances de classe.

Les fonctions de gestion des attributs

Pour accéder programmativement aux attributs d'un objet, on dispose des 3 fonctions *built-in* `getattr`, `setattr`, et `hasattr`, que nous allons illustrer tout de suite.

Lire un attribut

```
In [1]: import math
        # nous savons lire un attribut comme ceci
        # qui lit l'attribut de nom 'pi' dans le module math
        math.pi
```

```
Out[1]: 3.141592653589793
```

La fonction *built-in* `getattr` permet de lire un attribut programmativement :

```
In [2]: # si on part d'une chaîne qui désigne le nom de l'attribut
        # la formule équivalente est alors
        getattr(math, 'pi')
```

```
Out[2]: 3.141592653589793
```

```
In [3]: # on peut utiliser les attributs avec la plupart des objets
        # ici nous allons le faire sur une fonction
        def foo():
            "une fonction vide"
            pass

        # on a déjà vu certains attributs des fonctions
        print(f"nom={foo.__name__}, docstring=`{foo.__doc__}`")
```

```
nom=foo, docstring=`une fonction vide`
```

```
In [4]: # on peut préciser une valeur par défaut pour le cas où l'attribut
        # n'existe pas
        getattr(foo, "attribut_inexistant", 'valeur_par_defaut')
```

```
Out[4]: 'valeur_par_defaut'
```

Écrire un attribut

```
In [5]: # on peut ajouter un attribut arbitraire (toujours sur l'objet fonction)
        foo.hauteur = 100

        foo.hauteur
```

```
Out[5]: 100
```

Comme pour la lecture on peut écrire un attribut programmativement avec la fonction *built-in* `setattr` :

```
In [6]: # écrire un attribut avec setattr
        setattr(foo, "largeur", 200)

        # on peut bien sûr le lire indifféremment
        # directement comme ici, ou avec getattr
        foo.largeur
```

```
Out[6]: 200
```

Liste des attributs

La fonction *built-in* `hasattr` permet de savoir si un objet possède ou pas un attribut :

```
In [7]: # pour savoir si un attribut existe
        hasattr(math, 'pi')
```

```
Out[7]: True
```

Ce qui peut aussi être retrouvé autrement, avec la fonction *built-in* `vars` :

```
In [8]: vars(foo)
```

```
Out[8]: {'hauteur': 100, 'largeur': 200}
```

Sur quels objets

Il n'est pas possible d'ajouter des attributs sur les types de base, car ce sont des classes immuables :

```
In [9]: for builtin_type in (int, str, float, complex, tuple, dict, set, frozenset):
            obj = builtin_type()
            try:
                obj.foo = 'bar'
            except AttributeError as e:
                print(f"{builtin_type.__name__:>10} → exception {type(e)} - {e}")
```

```
       int → exception <class 'AttributeError'> - 'int' object has no attribute 'foo'
       str → exception <class 'AttributeError'> - 'str' object has no attribute 'foo'
     float → exception <class 'AttributeError'> - 'float' object has no attribute 'foo'
   complex → exception <class 'AttributeError'> - 'complex' object has no attribute 'foo'
     tuple → exception <class 'AttributeError'> - 'tuple' object has no attribute 'foo'
      dict → exception <class 'AttributeError'> - 'dict' object has no attribute 'foo'
       set → exception <class 'AttributeError'> - 'set' object has no attribute 'foo'
 frozenset → exception <class 'AttributeError'> - 'frozenset' object has no attribute 'foo'
```

C’est par contre possible sur virtuellement tout le reste, et notamment là où c’est très utile, c’est-à-dire pour ce qui nous concerne sur les :

— modules
— packages
— fonctions
— classes
— instances

6.18 Espaces de nommage

6.18.1 Complément - niveau basique

Nous venons de voir les règles pour l'affectation (ou l'assignation) et le référencement des variables et des attributs ; en particulier, on doit faire une distinction entre les attributs et les variables.

— Les attributs sont résolus de manière **dynamique**, c'est-à-dire au moment de l'exécution (*run-time*) ;
— alors que la liaison des variables est par contre **statique** (*compile-time*) et **lexicale**, en ce sens qu'elle se base uniquement sur les imbrications de code.

Vous voyez donc que la différence entre attributs et variables est fondamentale. Dans ce complément, nous allons reprendre et résumer les différentes règles qui régissent l'affectation et le référencement des attributs et des variables.

Attributs

Un attribut est un symbole `x` utilisé dans la notation `obj.x` où `obj` est l'objet qui définit l'espace de nommage sur lequel `x` existe.

L'**affectation** (explicite ou implicite) d'un attribut `x` sur un objet `obj` va créer (ou altérer) un symbole `x` **directement** dans l'espace de nommage de `obj`, symbole qui va référencer l'objet affecté, typiquement l'objet à droite du signe `=`

```
In [1]: class MaClasse:
            pass

        # affectation explicite
        MaClasse.x = 10

        # le symbole x est défini dans l'espace de nommage de MaClasse
        'x' in MaClasse.__dict__
```

```
Out[1]: True
```

Le **référencement** (la lecture) d'un attribut va chercher cet attribut **le long de l'arbre d'héritage** en commençant par l'instance, puis la classe qui a créé l'instance, puis les super-classes et suivant la MRO (voir le complément sur l'héritage multiple).

Variables

Une variable est un symbole qui n'est pas précédé de la notation `obj.` et l'affectation d'une variable rend cette variable locale au bloc de code dans lequel elle est définie, un bloc de code pouvant être :

— une fonction, dans ce cas la variable est locale à la fonction ;
— une classe, dans ce cas la variable est locale à la classe ;
— un module, dans ce cas la variable est locale au module, on dit également que la variable est globale.

Une variable référencée est toujours cherchée suivant la règle LEGB :

— localement au bloc de code dans lequel elle est référencée ;

— puis dans les blocs de code des **fonctions ou méthodes** englobantes, s'il y en a, de la plus proche à la plus eloignée ;
— puis dans le bloc de code du module.

Si la variable n'est toujours pas trouvée, elle est cherchée dans le module `builtins` et si elle n'est toujours pas trouvée, une exception est levée.

Par exemple :

```
In [2]: var = 'dans le module'

        class A:
            var = 'dans la classe A'
            def f(self):
                var = 'dans la fonction f'
                class B:
                    print(var)
                B()
        A().f()
```

```
dans la fonction f
```

En résumé Dans la vidéo et dans ce complément basique, on a couvert tous les cas standards, et même si python est un langage plutôt mieux fait, avec moins de cas particuliers que d'autres langages, il a également ses cas étranges entre raisons historiques et bugs qui ne seront jamais corrigés (parce que ça casserait plus de choses que ça n'en réparerait). Pour éviter de tomber dans ces cas spéciaux, c'est simple, vous n'avez qu'à suivre ces règles :

— ne jamais affecter dans un bloc de code local une variable de même nom qu'une variable globale ;
— éviter d'utiliser les directives `global` et `nonlocal`, et les réserver pour du code avancé comme les décorateurs et les métaclasses ;
— et lorsque vous devez vraiment les utiliser, toujours mettre les directives `global` et `nonlocal` comme premières instructions du bloc de code où elle s'appliquent.

Si vous ne suivez pas ces règles, vous risquez de tomber dans un cas particulier que nous détaillons ci-dessous dans la partie avancée.

6.18.2 Complément - niveau avancé

La documentation officielle est fausse

Oui, vous avez bien lu, la documentation officielle est fausse sur un point subtil. Regardons le modèle d'exécution, on trouve la phrase suivante "If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block." qui est fausse, il faut lire "If a name binding operation occurs anywhere within a code block **of a function**, all uses of the name within the block are treated as references to the current block."

En effet, les classes se comportent différemment des fonctions :

```
In [3]: x = "x du module"
        class A():
```

```
            print("dans classe A: " + x)
            x = "x dans A"
            print("dans classe A: " + x)
            del x
            print("dans classe A: " + x)

dans classe A: x du module
dans classe A: x dans A
dans classe A: x du module
```

Alors pourquoi si c'est une mauvaise idée de mélanger variables globales et locales de même nom dans une fonction, c'est possible dans une classe ?

Cela vient de la manière dont sont implémentés les espaces de nommage. Normalement, un objet a pour espace de nommage un dictionnaire qui s'appelle `__dict__`. D'un côté un dictionnaire est un objet python qui offre beaucoup de flexibilité, mais d'un autre côté, il induit un petit surcoût pour chaque recherche d'éléments. Comme les fonctions sont des objets qui par définition peuvent être appelés très souvent, il a été décidé de mettre toutes les variables locales à la fonction dans un objet écrit en C qui n'est pas dynamique (on ne peut pas ajouter des éléments à l'exécution), mais qui est un peu plus rapide qu'un dictionnaire lors de l'accès aux variables. Mais pour faire cela, il faut déterminer la portée de la variable dans la phase de précompilation. Donc si le précompilateur trouve une affectation (explicite ou implicite) dans une fonction, il considère la variable comme locale pour tout le bloc de code. Donc si on référence une variable définie comme étant locale avant une affectation dans la fonction, on ne va pas la chercher globalement, on a une erreur `UnboundLocalError`.

Cette optimisation n'a pas été faite pour les classes, parce que dans l'évaluation du compromis souplesse contre efficacité pour les classes, c'est la souplesse, donc le dictionnaire qui a gagné.

6.18.3 Complément - niveau avancé

Implémenter un itérateur de permutations

Dans ce complément nous allons nous amuser à implémenter une fonctionnalité qui est déjà disponible dans le module `itertools`.

C'est quoi déjà les permutations ? En guise de rappel, l'ensemble des permutations d'un ensemble fini correspond à toutes les façons d'ordonner ses éléments ; si l'ensemble est de cardinal n, il possède $n!$ permutations : on a n façons de choisir le premier élément, $n-1$ façons de choisir le second, etc.

Un itérateur sur les permutations est disponible au travers du module standard `itertools`. Cependant il nous a semblé intéressant de vous montrer comment nous pourrions écrire nous-mêmes cette fonctionnalité, de manière relativement simple.

Pour illustrer le concept, voici à quoi ressemblent les 6 permutations d'un ensemble à trois éléments :

```
In [1]: from itertools import permutations

In [2]: set = {1, 2, 3}

        for p in permutations(set):
            print(p)
```

```
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

Une implémentation

Voici une implémentation possible pour un itérateur de permutations :

```
In [3]: class Permutations:
            """
            Un itérateur qui énumère les permutations de n
            sous la forme d'une liste d'indices commençant à 0
            """
            def __init__(self, n):
                # le constructeur bien sûr ne fait (presque) rien
                self.n = n
                # au fur et à mesure des itérations
                # le compteur va aller de 0 à n-1
                # puis retour à 0 et comme ça en boucle sans fin
                self.counter = 0
                # on se contente d'allouer un iterateur de rang n-1
                # si bien qu'une fois qu'on a fini de construire
                # l'objet d'ordre n on a n objets Permutations en tout
```

```
        if n >= 2:
            self.subiterator = Permutations(n-1)

    # pour satisfaire le protocole d'itération
    def __iter__(self):
        return self

    # c'est ici bien sûr que se fait tout le travail
    def __next__(self):
        # pour n == 1
        # le travail est très simple
        if self.n == 1:
            # on doit renvoyer une fois la liste [0]
            # car les indices commencent à 0
            if self.counter == 0:
                self.counter += 1
                return [0]
            # et ensuite c'est terminé
            else:
                raise StopIteration

        # pour n >= 2
        # lorsque counter est nul,
        # on traite la permutation d'ordre n-1 suivante
        # si next() lève StopIteration on n'a qu'à laisser passer
        # car en effet c'est qu'on a terminé
        if self.counter == 0:
            self.subsequence = next(self.subiterator)
        #
        # on insère alors n-1 (car les indices commencent à 0)
        # successivement dans la sous-sequence
        #
        # naivement on écrirait
        # result = self.subsequence[0:self.counter] \
        #     + [self.n - 1] \
        #     + self.subsequence[self.counter:self.n-1]
        # mais c'est mettre le nombre le plus élevé en premier
        # et donc à itérer les permutations dans le mauvais ordre,
        # en commençant par la fin
        #
        # donc on fait plutôt une symétrie
        # pour insérer en commençant par la fin
        cutter = self.n-1 - self.counter
        result = self.subsequence[0:cutter] + [self.n - 1] \
                 + self.subsequence[cutter:self.n-1]
        #
        # on n'oublie pas de maintenir le compteur et de
        # le remettre à zéro tous les n tours
        self.counter = (self.counter+1) % self.n
        return result
```

```
    # la longeur de cet itérateur est connue
    def __len__(self):
        import math
        return math.factorial(self.n)
```

Ce qu'on a essayé d'expliquer dans les commentaires, c'est qu'on procède en fin de compte par récurrence. Un objet `Permutations` de rang `n` possède un sous-itérateur de rang `n-1` qu'on crée dans le constructeur. Ensuite l'objet de rang `n` va faire successivement (c'est-à-dire à chaque appel de `next()`) :

— appel *0* :

  — demander à son sous-itérateur une permutation de rang `n-1` (en lui envoyant `next`),
  — la stocker dans l'objet de rang `n`, ce sera utilisé par les *n* premier appels,
  — et construire une liste de taille `n` en insérant `n-1` à la fin de la séquence de taille `n-1`,

— appel *1* :

  — insérer `n-1` dans la même séquence de rang `n-1` mais cette fois 1 cran avant la fin,

— ...

— appel *n-1* :

  — insérer `n-1` au début de la séquence de rang `n-1`,

— appel *n* :

  — refaire `next()` sur le sous-itérateur pour traiter une nouvelle sous-séquence,
  — la stocker dans l'objet de rang `n`, comme à l'appel *0*, pour ce bloc de n appels,
  — et construire la permutation en insérant *n-1* à la fin, comme à l'appel 0,

— ...

On voit donc le caractère cyclique d'ordre *n* qui est matérialisé par `counter`, que l'on incrémente à chaque boucle mais modulo *n* - notez d'ailleurs que pour ce genre de comportement on dispose aussi de `itertools.cycle` comme on le verra dans une deuxième version, mais pour l'instant j'ai préféré ne pas l'utiliser pour ne pas tout embrouiller ;)

La terminaison se gère très simplement, car une fois que l'on a traité toutes les séquences d'ordre *n-1* eh bien on a fini, on n'a même pas besoin de lever StopIteration explicitement, sauf bien sûr dans le cas *n=1*.

Le seul point un peu délicat, si on veut avoir les permutations dans le "bon" ordre, consiste à commencer à insérer `n-1` par la droite (la fin de la sous-séquence).

Discussion

Il existe certainement des tas d'autres façons de faire bien entendu. Le point important ici, et qui donne toute sa puissance à la notion d'itérateur, c'est **qu'à aucun moment on ne construit** une liste ou une séquence quelconque de **n! termes**.

C'est une erreur fréquente chez les débutants que de calculer une telle liste dans le constructeur, mais procéder de cette façon c'est aller exactement à l'opposé de ce pourquoi les itérateurs ont été conçus ; au contraire, on veut éviter à tout prix le coût d'une telle construction.

On peut le voir sur un code qui n'utiliserait que les 20 premières valeurs de l'itérateur, vous constatez que ce code est immédiat :

```
In [4]: def show_first_items(iterable, nb_items):
            """
            montre les <nb_items> premiers items de iterable
            """
            print(f"Il y a {len(iterable)} items dans l'itérable")
            for i, item in enumerate(iterable):
                print(item)
                if i >= nb_items:
                    print('....')
                    break

In [5]: show_first_items(Permutations(12), 20)

Il y a 479001600 items dans l'itérable
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 11, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 11, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 11, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 11, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 11, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 11, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 11, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 11, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 11, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[11, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 9, 11]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 11, 10, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 11, 8, 10, 9]
[0, 1, 2, 3, 4, 5, 6, 11, 7, 8, 10, 9]
[0, 1, 2, 3, 4, 5, 11, 6, 7, 8, 10, 9]
[0, 1, 2, 3, 4, 11, 5, 6, 7, 8, 10, 9]
[0, 1, 2, 3, 11, 4, 5, 6, 7, 8, 10, 9]
[0, 1, 2, 11, 3, 4, 5, 6, 7, 8, 10, 9]
...
```

Ce tableau vous montre par ailleurs sous un autre angle comment fonctionne l'algorithme, si vous observez le `11` qui balaie en diagonale les 12 premières lignes, puis les 12 suivantes, etc..

Ultimes améliorations

Dernières remarques, sur des améliorations possibles - mais tout à fait optionnelles :

— le lecteur attentif aura remarqué qu'au lieu d'un entier `counter` on aurait pu profitablement utiliser une instance de `itertools.cycle`, ce qui aurait eu l'avantage d'être plus clair sur le propos de ce compteur ;
— aussi dans le même mouvement, au lieu de se livrer à la gymnastique qui calcule `cutter` à partir de `counter`, on pourrait dès le départ créer dans le cycle les bonnes valeurs en commençant à `n-1`.

C'est ce qu'on a fait dans cette deuxième version ; après avoir enlevé la logorrhée de commentaires ça redevient presque lisible ;)

```
In [6]: import itertools

        class Permutations2:
            """
            Un itérateur qui énumère les permutations de n
            sous la forme d'une liste d'indices commençant à 0
            """
            def __init__(self, n):
                self.n = n
                # on commence à insérer à la fin
                self.cycle = itertools.cycle(list(range(n))[::-1])
                if n >= 2:
                    self.subiterator = Permutations2(n-1)
                # pour savoir quand terminer le cas n==1
                if n == 1:
                    self.done = False

            def __iter__(self):
                return self

            def __next__(self):
                cutter = next(self.cycle)

                # quand n==1 on a toujours la même valeur 0
                if self.n == 1:
                    if not self.done:
                        self.done = True
                        return [0]
                    else:
                        raise StopIteration

                # au début de chaque séquence de n appels
                # il faut appeler une nouvelle sous-séquence
                if cutter == self.n-1:
                    self.subsequence = next(self.subiterator)
                # dans laquelle on insére n-1
                return self.subsequence[0:cutter] + [self.n-1] \
                         + self.subsequence[cutter:self.n-1]

            # la longeur de cet itérateur est connue
            def __len__(self):
                import math
                return math.factorial(self.n)

In [7]: show_first_items(Permutations2(5), 20)

Il y a 120 items dans l'itérable
[0, 1, 2, 3, 4]
[0, 1, 2, 4, 3]
```

```
[0, 1, 4, 2, 3]
[0, 4, 1, 2, 3]
[4, 0, 1, 2, 3]
[0, 1, 3, 2, 4]
[0, 1, 3, 4, 2]
[0, 1, 4, 3, 2]
[0, 4, 1, 3, 2]
[4, 0, 1, 3, 2]
[0, 3, 1, 2, 4]
[0, 3, 1, 4, 2]
[0, 3, 4, 1, 2]
[0, 4, 3, 1, 2]
[4, 0, 3, 1, 2]
[3, 0, 1, 2, 4]
[3, 0, 1, 4, 2]
[3, 0, 4, 1, 2]
[3, 4, 0, 1, 2]
[4, 3, 0, 1, 2]
[0, 2, 1, 3, 4]
...
```

Il me semble intéressant de montrer une autre façon, plus simple, d'écrire un itérateur de permutations, à base cette fois de générateurs; c'est un tout petit peu une digression par rapport au cours qui est sur la conception d'itérateurs et d'itérables. Ça va nous permettre surtout de réviser la notion de `yield from`.

On commence par une version très rustique qui fait des impressions :

```
In [8]: # pour simplifier ici on suppose que l'entrée est une vraie liste
        # que l'on va ainsi pouvoir modifier par effets de bord
        def gen_perm1(subject, k=0):
            if k == len(subject):
                # cette version hyper rustique se contente de faire une impression
                print(subject)
            else:
                for i in range(k, len(subject)):
                    # on échange
                    subject[k], subject[i] = subject[i], subject[k]
                    gen_perm1(subject, k+1)
                    # on remet comme c'était pour le prochain échange
                    subject[k], subject[i] = subject[i], subject[k]
```

```
In [9]: gen_perm1(['a', 'b', 'c', 'd'])
```

```
['a', 'b', 'c', 'd']
['a', 'b', 'd', 'c']
['a', 'c', 'b', 'd']
['a', 'c', 'd', 'b']
['a', 'd', 'c', 'b']
['a', 'd', 'b', 'c']
['b', 'a', 'c', 'd']
['b', 'a', 'd', 'c']
```

```
['b', 'c', 'a', 'd']
['b', 'c', 'd', 'a']
['b', 'd', 'c', 'a']
['b', 'd', 'a', 'c']
['c', 'b', 'a', 'd']
['c', 'b', 'd', 'a']
['c', 'a', 'b', 'd']
['c', 'a', 'd', 'b']
['c', 'd', 'a', 'b']
['c', 'd', 'b', 'a']
['d', 'b', 'c', 'a']
['d', 'b', 'a', 'c']
['d', 'c', 'b', 'a']
['d', 'c', 'a', 'b']
['d', 'a', 'c', 'b']
['d', 'a', 'b', 'c']
```

Très bien, mais on ne veut pas imprimer, on veut itérer. On pourrait se dire, il me suffit de remplacer `print` par `yield`. Essayons cela :

```
In [10]: # pour simplifier ici on suppose que l'entrée est une vraie liste
         # que l'on va ainsi pouvoir modifier par effets de bord
         def gen_perm2(subject, k=0):
             if k == len(subject):
                 # cette version hyper rustique se contente de faire une impression
                 yield subject
             else:
                 for i in range(k, len(subject)):
                     # on échange
                     subject[k], subject[i] = subject[i], subject[k]
                     gen_perm2(subject, k+1)
                     # on remet comme c'était pour le prochain échange
                     subject[k], subject[i] = subject[i], subject[k]

In [11]: for perm in gen_perm2(['a', 'b', 'c', 'd']):
             print(perm)
```

On est exactement dans le cas où il nous faut utiliser `yield from`. En effet lorsqu'on appelle `gen_perm(subject, k+1)` ici, ce qu'on obtient en retour c'est maintenant un objet générateur. Pour faire ce qu'on cherche à faire il nous faut bien utiliser cet objet générateur, et pour cela on utilise `yield from`.

```
In [12]: # pour simplifier ici on suppose que l'entrée est une vraie liste
         # que l'on va ainsi pouvoir modifier par effets de bord
         def gen_perm3(subject, k=0):
             if k == len(subject):
                 # cette version hyper rustique se contente de faire une impression
                 yield subject
             else:
                 for i in range(k, len(subject)):
                     # on échange
```

```
                    subject[k], subject[i] = subject[i], subject[k]
                    yield from gen_perm3(subject, k+1)
                    # on remet comme c'était pour le prochain échange
                    subject[k], subject[i] = subject[i], subject[k]
```

```
In [13]: for perm in gen_perm3(['a', 'b', 'c', 'd']):
             print(perm)
```

```
['a', 'b', 'c', 'd']
['a', 'b', 'd', 'c']
['a', 'c', 'b', 'd']
['a', 'c', 'd', 'b']
['a', 'd', 'c', 'b']
['a', 'd', 'b', 'c']
['b', 'a', 'c', 'd']
['b', 'a', 'd', 'c']
['b', 'c', 'a', 'd']
['b', 'c', 'd', 'a']
['b', 'd', 'c', 'a']
['b', 'd', 'a', 'c']
['c', 'b', 'a', 'd']
['c', 'b', 'd', 'a']
['c', 'a', 'b', 'd']
['c', 'a', 'd', 'b']
['c', 'd', 'a', 'b']
['c', 'd', 'b', 'a']
['d', 'b', 'c', 'a']
['d', 'b', 'a', 'c']
['d', 'c', 'b', 'a']
['d', 'c', 'a', 'b']
['d', 'a', 'c', 'b']
['d', 'a', 'b', 'c']
```

6.19 *Context managers* et exceptions

6.19.1 Complément - niveau intermédiaire

On a vu jusqu'ici dans la vidéo comment écrire un context manager ; on a vu notamment qu'il était bon pour la méthode `__exit__()` de retourner `False`, de façon à ce que l'exception soit propagée à l'instruction `with` :

```
In [1]: import time

        class Timer1:
            def __enter__(self):
                print("Entering Timer1")
                self.start = time.time()
                return self

            # en règle générale on se contente de propager l'exception
            # à l'instruction with englobante
            def __exit__(self, *args):
                print(f"Total duration {time.time()-self.start:2f}")

                # et pour cela il suffit que __exit__ retourne False
                return False
```

Ainsi si le corps de l'instruction lève une exception, celle-ci est propagée :

```
In [2]: import time
        try:
            with Timer1():
                time.sleep(0.5)
                1/0
        except Exception as e:
            # on va bien recevoir cette exception
            print(f"OOPS -> {type(e)}")
```

```
Entering Timer1
Total duration 0.514807
OOPS -> <class 'ZeroDivisionError'>
```

À la toute première itération de la boucle, on fait une division par 0 qui lève l'exception `ZeroDivisionError`, qui passe bien à l'appelant.

Il est important, lorsqu'on conçoit un context manager, de bien **propager** les exceptions qui ne sont pas liées au fonctionnement attendu du context manager. Par exemple un objet de type fichier va par exemple devoir attraper les exceptions liées à la fin du fichier, mais doit par contre laisser passer une exception comme `ZeroDivisionError`.

Les paramètres de `__exit__`

Si on a besoin de filtrer entre les exceptions - c'est-à-dire en laisser passer certaines et pas d'autres - il nous faut quelque chose de plus pour pouvoir faire le tri. Comme vous pouvez le retrouver ici, la méthode `__exit__` reçoit trois arguments :

`def __exit__(self, exc_type, exc_value, traceback):`

— si l'on sort du bloc `with` sans qu'une exception soit levée, ces trois arguments valent `None` ;
— par contre si une exception est levée, ils permettent d'accéder respectivement au type, à la valeur de l'exception, et à l'état de la pile lorsque l'exception est levée.

Pour illustrer cela, écrivons une nouvelle version de `Timer` qui filtre, disons, l'exception `ZeroDivisionError` que je choisis au hasard, c'est uniquement pour illustrer le mécanisme.

```
In [3]: # une deuxième version de Timer
        # qui propage toutes les exceptions sauf 'OSError'

        class Timer2:
            def __enter__(self):
                print("Entering Timer1")
                self.start = time.time()
                # rappel : le retour de __enter__ est ce qui est passé
                # à la clause `as` du `with`
                return self

            def __exit__(self, exc_type, exc_value, traceback):
                if exc_type is None:
                    # pas d'exception levée dans le corps du 'with'
                    print(f"Total duration {time.time()-self.start:2f}")
                    # dans ce cas la valeur de retour n'est pas utilisée
                else:
                    # il y a eu une exception de type 'exc_type'
                    if exc_type in (ZeroDivisionError,) :
                        print("on étouffe")
                        # on peut l'étouffer en retournant True
                        return True
                    else:
                        print(f"OOPS : on propage l'exception {exc_type} - {exc_value}")
                        # et pour ça il suffit... de ne rien faire du tout
                        # ce qui renverra None
```

```
In [4]: # commençons avec un code sans souci
        try:
            with Timer2():
                time.sleep(0.5)
        except Exception as e:
            # on va bien recevoir cette exception
            print(f"OOPS -> {type(e)}")
```

```
Entering Timer1
Total duration 0.514807
```

```
In [5]: # avec une exception filtrée
        try:
            with Timer2():
                time.sleep(0.5)
                1/0
```

```
        except Exception as e:
            # on va bien recevoir cette exception
            print(f"OOPS -> {type(e)}")
```

```
Entering Timer1
on étouffe
```

```
In [6]: # avec une autre exception
        try:
            with Timer2():
                time.sleep(0.5)
                raise OSError()
        except Exception as e:
            # on va bien recevoir cette exception
            print(f"OOPS -> {type(e)}")
```

```
Entering Timer1
OOPS : on propage l'exception <class 'OSError'> -
OOPS -> <class 'OSError'>
```

La bibliothèque `contextlib`

Je vous signale aussi la bibliothèque `contextlib` qui offre quelques utilitaires pour se définir un contextmanager.

Notamment, elle permet d'implémenter un context manager sous une forme compacte à l'aide d'une fonction génératrice - et du décorateur `contextmanager` :

```
In [7]: from contextlib import contextmanager
```

```
In [8]: # l'objet compact_timer est un context manager !
        @contextmanager
        def compact_timer(message):
            start = time.time()
            yield
            print(f"{message}: duration = {time.time() - start}")
```

```
In [9]: with compact_timer("Squares sum"):
            print(sum(x**2 for x in range(10**5)))
```

```
333328333350000
Squares sum: duration = 0.031200408935546875
```

Un peu comme on peut implémenter un itérateur à partir d'une fonction génératrice qui fait (n'importe quel nombre de) `yield`, ici on implémente un context manager compact sous la forme d'une fonction génératrice.

Comme vous l'avez sans doute deviné sur la base de cet exemple, il faut que la fonction fasse **exactement un** `yield` : ce qui se passe avant le `yield` est du ressort de `__enter__`, et la fin est du ressort de `__exit__()`.

Bien entendu on n'a pas la même puissance d'expression avec cette méthode par rapport à une vraie classe, mais cela permet de créer des context managers avec le minimum de code.

6.20 Exercice sur l'utilisation des classes

Introduction

Objectifs de l'exercice

Maintenant que vous avez un bagage qui couvre toutes les bases du langage, cette semaine nous ne ferons qu'un seul exercice de taille un peu plus réaliste. Vous devez écrire quelques classes, que vous intégrez ensuite dans un code écrit pas nos soins.

L'exercice comporte donc à la fois une part lecture et une part écriture.

Par ailleurs, cette fois-ci l'exercice n'est plus à faire dans un notebook ; vous êtes donc également incités à améliorer autant que possible l'environnement de travail sur votre propre ordinateur.

Objectifs de l'application

Dans le prolongement des exercices de la semaine 3 sur les données maritimes, l'application dont il est question ici fait principalement ceci :

— en entrée :

  — agréger des données obtenues auprès de marinetraffic ;

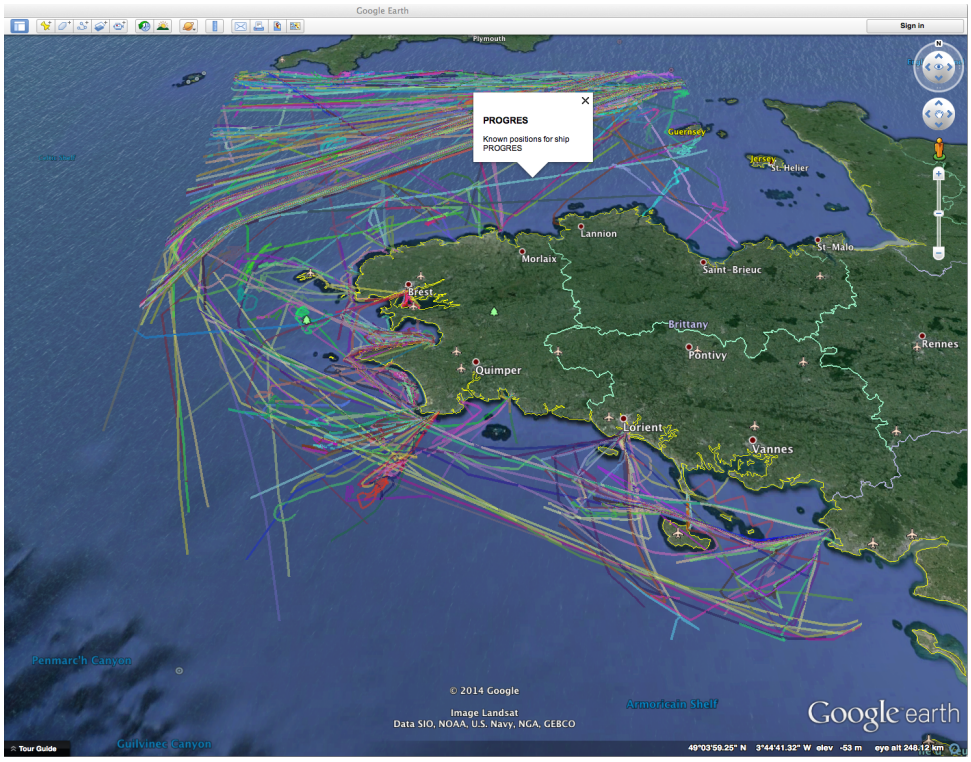
— et produire en sortie :

  — un fichier texte qui liste par ordre alphabétique les bateaux concernés, et le nombre de positions trouvées pour chacun ;
  — et un fichier KML, pour exposer les trajectoires trouvées à Google Earth, Google Maps ou autre outil similaire.

Les données générées dans ces deux fichiers sont triées dans l'ordre alphabétique, de façon à permettre une comparaison des résultats sous forme textuelle. Plus précisément, on trie les bateaux selon le critère suivant :

— ordre alphabétique sur le nom des bateaux ;
— et ordre sur les `id` en cas d'ex-aequo (il y a des bateaux homonymes dans cet échantillon réel).

Voici à quoi ressemble le fichier KML obtenu avec les données que nous fournissons, une fois ouvert sous Google Earth :



Choix d'implémentation

En particulier, dans cet exercice nous allons voir comment on peut gérer des données sous forme d'instances de classes plutôt que de types de base. Cela résonne avec la discussion commencée en Semaine 3, Séquence "Les dictionnaires", dans le complément "record-et-dictionnaire".

Dans les exercices de cette semaine-là nous avions uniquement utilisé des types "standard" comme listes, tuples et dictionnaires pour modéliser les données, cette semaine nous allons faire le choix inverse et utiliser plus souvent des (instances de) classes.

Principe de l'exercice

On a écrit une application complète, constituée de 4 modules; on vous donne le code de trois de ces modules et vous devez écrire le module manquant.

Correction

Tout d'abord nous fournissons un jeu de données d'entrées. De plus, l'application vient avec son propre système de vérification, qui est très rustique. Il consiste à comparer, une fois les sorties produites, leur contenu avec les sorties de référence, qui ont été obtenues avec notre version de l'application.

Du coup, le fait de disposer de Google Earth sur votre ordinateur n'est pas strictement nécessaire, on ne s'en sert pas à proprement parler pour l'exercice.

Mise en place

Partez d'un répertoire vierge

Pour commencer, créez-vous un répertoire pour travailler à cet exercice.

Les données

Commencez par y installer les données que nous publions dans les formats suivants :

— au format tar
— au format tar compressé
— au format zip

Une fois installées, ces données doivent se trouver dans un sous-répertoire `json/` qui contient 133 fichiers `*.json` :

— `json/2013-10-01-00-00--t=10--ext.json`
— `...`
— `json/2013-10-01-23-50--t=10.json`

Comme vous pouvez le deviner, il s'agit de données sur le mouvement des bateaux dans la zone en date du 10 Octobre 2013 ; et comme vous le devinez également, on a quelques exemplaires de données étendues, mais dans la plupart des cas il s'agit de données abrégées.

Les résultats de référence

De même il vous faut installer les résultats de référence que vous trouvez ici :

— au format tar
— au format tar compressé (tgz)
— au format zip

Quel que soit le format choisi, une fois installé ceci doit vous donner trois fichiers :

— `ALL_SHIPS.kml.ref`
— `ALL_SHIPS.txt.ref`
— `ALL_SHIPS-v.txt.ref`

Le code

Vous pouvez à présent aller chercher les 3 modules suivants :

— `merger.py`
— `compare.py`
— `kml.py`

et les sauver dans le même répertoire.

Vous remarquerez que le code est cette fois entièrement rédigé en anglais, ce que nous vous conseillons de faire aussi souvent que possible.

Votre but dans cet exercice est d'écrire le module manquant `shipdict.py` qui permettra à l'application de fonctionner comme attendu.

Fonctionnement de l'application

Comment est structurée l'application

Le point d'entrée s'appelle `merger.py`
Il utilise trois modules annexes, qui sont :

— `shipdict.py`, qui implémente les classes
  — `Position` qui contient une latitude, une longitude, et un timestamp
  — `Ship` qui modélise un bateau à partir de son `id` et optionnellement `name` et `country`
  — `ShipDict`, qui maintient un index des bateaux (essentiellement un dictionnaire)

— `compare.py` qui implémente
  — la classe `Compare` qui se charge de comparer les fichiers résultat avec leur version de référence

— `kml.py` qui implémente
  — la classe `KML` dans laquelle sont concentrées les fonctions liées à la génération de KML ; c'est notamment en fonction de nos objectifs pédagogiques que ce choix a été fait.

Lancement

Lorsque le programme est complet et qu'il fonctionne correctement, on le lance comme ceci :

```
   $ python3 merger.py json/*
Opening ALL_SHIPS.txt for listing all named ships
Opening ALL_SHIPS.kml for ship ALL_SHIPS
Comparing ALL_SHIPS.txt and ALL_SHIPS.txt.ref -> OK
Comparing ALL_SHIPS.kml and ALL_SHIPS.kml.ref -> OK
```

qui comme on le voit produit :

— `ALL_SHIPS.txt` qui résume, par ordre alphabétique les bateaux qu'on a trouvés et le nombre de positions pour chacun, et
— `ALL_SHIPS.kml` qui est le fichier au format `KML` qui contient toutes les trajectoires.

Mode bavard (verbose)

On peut également lancer l'application avec l'option `--verbose` ou simplement `-v` sur la ligne de commande, ce qui donne un résultat plus détaillé. Le code KML généré reste inchangé, mais la sortie sur le terminal et le fichier de résumé sont plus étoffés :

```
$ python merger.py --verbose json/*.json
Opening json/2013-10-01-00-00--t=10--ext.json for parsing JSON
Opening json/2013-10-01-00-10--t=10.json for parsing JSON
...
Opening json/2013-10-01-23-40--t=10.json for parsing JSON
Opening json/2013-10-01-23-50--t=10.json for parsing JSON
Opening ALL_SHIPS-v.txt for listing all named ships
Opening ALL_SHIPS.kml for ship ALL_SHIPS
Comparing ALL_SHIPS-v.txt and ALL_SHIPS-v.txt.ref -> OK
Comparing ALL_SHIPS.kml and ALL_SHIPS.kml.ref -> OK
```

À noter que dans le mode bavard toutes les positions sont listées dans le résumé au format texte, ce qui le rend beaucoup plus bavard comme vous pouvez le voir en inspectant la taille des deux fichiers de référence :

```
$ ls -l ALL_SHIPS*txt.ref v2.0
-rw-r--r--  1 parmentelat  staff  1438681 Dec  4 16:20 ALL_SHIPS-v.txt.ref
-rw-r--r--  1 parmentelat  staff    15331 Dec  4 16:20 ALL_SHIPS.txt.ref
-rw-r--r--  1 parmentelat  staff        0 Dec  4 16:21 v2.0
```

```
   merger.py --help
```

```
$ merger.py --help
usage: merger.py [-h] [-v] [-s SHIP_NAME] [-z] [inputs [inputs ...]]

positional arguments:
  inputs

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Verbose mode
  -s SHIP_NAME, --ship SHIP_NAME
                        Restrict to ships by that name
  -z, --gzip            Store kml output in gzip (KMZ) format
```

Un mot sur les données

Attention, le contenu détaillé des champs `extended` et `abbreviated` peut être légèrement différent de ce qu'on avait pour les exercices de la semaine 3, dans lesquels certaines simplifications ont été apportées.

Voici ce avec quoi on travaille cette fois-ci :

```
>>> extended[0]
[228317000, 48.76829, -4.334262, 75, 333, u'2013-09-30T21:54:00', u'MA GONDOLE',
30, 0, u'FGSA', u'FR', u'', u'', u'', u'CLASS B', u'', 13, 3, 0, u'', u'', u'']
```

c'est-à-dire :

```
[ id, latitude, longitude, _, _, timestamp, name, _, _, _, country, ...]
```

et en ce qui concerne les données abrégées :

```
>>> abbreviated[0]
[232005670, 49.39331, -5.939922, 33, 269, 3, u'2013-10-01T06:08:00']
```

c'est-à-dire :

```
[ id, latitude, longitude, _, _, _, timestamp]
```

Il y a unicité des `id` bien entendu (deux relevés qui portent le même `id` concernent le même bateau).

Note historique Dans une première version de cet exercice, on avait laissé des doublons, c'est-à-dire des bateaux différents mais de même nom. Afin de rendre l'exercice plus facile à corriger (notamment parce que la comparaison des résultats repose sur l'ordre alphabétique), dans la présente version ces doublons ont été enlevés. Sachez toutefois que cette unicité est artificielle, aussi efforcez-vous de ne pas écrire de code qui reposerait sur cette hypothèse.

6.20.1 Niveaux pour l'exercice

Quel que soit le niveau auquel vous choisissez de faire l'exercice, nous vous conseillons de commencer par lire intégralement les 3 modules qui sont à votre disposition, dans l'ordre :

— `merger.py` qui est le chef d'orchestre de toute cette affaire ;
— `compare.py` qui est très simple ;
— `kml.py` qui ne présente pas grand intérêt en soi si ce n'est pour l'utilisation de la classe `string.Template` qui peut être utile dans d'autres contextes également.

En **niveau avancé**, l'énoncé pourrait s'arrêter là ; vous lisez le code qui est fourni et vous en déduisez ce qui manque pour faire fonctionner le tout. En cas de difficulté liée aux arrondis avec le mode bavard, vous pouvez toutefois vous inspirer du code qui est donné dans la toute dernière section de cet énoncé (section "Un dernier indice"), pour traduire un flottant en représentation textuelle.

Vous pouvez considérer que vous avez achevé l'exercice lorsque les deux appels suivants affichent les deux dernières lignes avec OK :

```
$ python merger.py json/*.json
...
Comparing ALL_SHIPS.txt and ALL_SHIPS.txt.ref -> OK
Comparing ALL_SHIPS.kml and ALL_SHIPS.kml.ref -> OK
```

```
$ python merger.py -v json/*.json
...
Comparing ALL_SHIPS-v.txt and ALL_SHIPS-v.txt.ref -> OK
Comparing ALL_SHIPS.kml and ALL_SHIPS.kml.ref -> OK
```

Le cas où on lance `merger.py` avec l'option bavarde est facultatif.

En **niveau intermédiaire**, nous vous donnons ci-dessous un extrait de ce que donne `help` sur les classes manquantes de manière à vous donner une indication de ce que vous devez écrire.

Classe `Position`

```
Help on class Position in module shipdict:

class Position(__builtin__.object)
 |  a position atom with timestamp attached
 |
 |  Methods defined here:
 |
 |  __init__(self, latitude, longitude, timestamp)
 |      constructor
 |
 |  __repr__(self)
 |      only used when merger.py is run in verbose mode
 |
```

Notes

— certaines autres classes comme `KML` sont également susceptibles d'accéder aux champs internes d'une instance de la classe `Position` en faisant simplement `position.latitude`
— La classe `Position` redéfinit `__repr__`, ceci est utilisé uniquement dans la sortie en mode bavard.

Classe `Ship`

```
Help on class Ship in module shipdict:

class Ship(__builtin__.object)
 |  a ship object, that requires a ship id,
 |  and optionally a ship name and country
 |  which can also be set later on
 |
 |  this object also manages a list of known positions
 |
 |  Methods defined here:
 |
 |  __init__(self, id, name=None, country=None)
 |      constructor
 |
 |  add_position(self, position)
 |      insert a position relating to this ship
 |      positions are not kept in order so you need
 |      to call `sort_positions` once you're done
 |
 |  sort_positions(self)
 |      sort list of positions by chronological order
```

Classe `Shipdict`

```
Help on class ShipDict in module shipdict:

class ShipDict(__builtin__.dict)
 |  a repository for storing all ships that we know about
 |  indexed by their id
 |
 |  Method resolution order:
 |      ShipDict
 |      __builtin__.dict
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      constructor
 |
 |  __repr__(self)
 |
 |  add_abbreviated(self, chunk)
 |      adds an abbreviated data chunk to the repository
 |
```

```
 |  add_chunk(self, chunk)
 |      chunk is a plain list coming from the JSON data
 |      and be either extended or abbreviated
 |
 |      based on the result of is_abbreviated(),
 |      gets sent to add_extended or add_abbreviated
 |
 |  add_extended(self, chunk)
 |      adds an extended data chunk to the repository
 |
 |  all_ships(self)
 |      returns a list of all ships known to us
 |
 |  clean_unnamed(self)
 |      Because we enter abbreviated and extended data
 |      in no particular order, and for any time period,
 |      we might have ship instances with no name attached
 |      This method removes such entries from the dict
 |
 |  is_abbreviated(self, chunk)
 |      depending on the size of the incoming data chunk,
 |      guess if it is an abbreviated or extended data
 |
 |  ships_by_name(self, name)
 |      returns a list of all known ships with name <name>
 |
 |  sort(self)
 |      makes sure all the ships have their positions
 |      sorted in chronological order
```

Un dernier indice

Pour éviter de la confusion, voici le code que nous utilisons pour transformer un flottant en coordonnées lisibles dans le résumé généré en mode bavard.

```
def d_m_s(f):
    """
    makes a float readable; e.g. transforms 2.5 into 2.30'00''
    we avoid using ° to keep things simple
    input is assumed positive
    """
    d = int (f)
    m = int((f-d)*60)
    s = int( (f-d)*3600 - 60*m)
    return f"{d:02d}.{m:02d}'{s:02d}''"
```

6.21 Outils périphériques

6.21.1 Compléments - niveau intermédiaire

Pour conclure le tronc commun de ce cours Python, nous allons très rapidement citer quelques outils qui ne sont pas nécessairement dans la bibliothèque standard, mais qui sont très largement utilisés dans l'écosystème python.
[U+25CB]
Il s'agit d'une liste non exhaustive bien entendu.

Distribution et packaging

On l'a rapidement mentionné, il existe une infrastructure globale pour la distribution de librairies écrites en python. Celle-ci repose sur

— un site web https://pypi.org/ où l'on peut consulter les - très nombreuses - bibliothèques diffusées, avec leurs historiques et révisions,
— un outil en ligne de commande `pip`, pour installer et mettre à jour ces bibliothèques (utiliser `pip3` comme `python3` si vous avez python2 installé en parallèle),
— et enfin un système de packaging à destination des éditeurs.

Je vous signale, par rapport à ce dernier point, que la bibliothèque standard vient avec un outil qui s'appelle `distutils`, qui est essentiellement obsolète, et qui n'est conservé que pour des questions de compatibilité. Si vous devez commencer depuis une feuille blanche le packaging d'une nouvelle librairie, je vous recommande d'utiliser plutôt `setuptools` qui est le standard de fait dans le domaine.

Dans une domaine très voisin, l'outil `virtualenv` est très populaire également ; il permet de créer sur une seul machine, plusieurs environnements python avec des versions et contenus différents.

C'est très utile si vous travaillez sur plusieurs projets, dont l'un a besoin de python-3.5 avec numpy et sans pandas, et Django-1.11, et un second avec python-3.6 sans numpy et avec Django-2.0.

Pour finir, on ne peut pas parler de packaging sans citer `conda`, l'outil de référence pour la packaging et les environnements virtuels en data science. Quelques références sur conda :

— une documentation complète de conda https://conda.io/docs/
— une excellente discussion (en anglais) sur le positionnement de `pip` et `conda` http://jakevdp.github.io/blog/2016/08/25/conda-myths-and-misconceptions/

Debugging

Pour le debugging, la bibliothèque standard s'appelle `pdb`. Typiquement pour mettre un *breakpoint* on écrit :

```
def foo(n):
    n = n ** 2
    # pour mettre un point d'arrêt
    import pdb
    pdb.set_trace()
    # la suite de foo()
    return n / 10
```

Je vous signale d'ailleurs qu'à partir de Python 3.7, il est recommandé d'utiliser la nouvelle fonction *built-in breakpoint()* qui rend le même service.

Tests

Le module `unittest` de la bibliothèque standard fournit des fonctionnalités de base pour écrire des tests unitaires.

Je vous signale par ailleurs des outils comme `nosetests` et `pytest`, qui ne sont pas dans la distribution standard, mais qui enrichissent les capacités de `unittest` pour en rendre l'utilisation quotidienne plus fluide.

Documentation

Le standard de fait dans ce domaine est clairement une combinaison basée sur

- l'outil `sphinx`, qui permet de générer la documentation à partir du source, avec
  - des plugins pour divers sous-formats dans les docstrings,
  - un système de templating,
  - et de nombreuses autres possibilités ;
- `readthedocs.io` qui est une plateforme ouverte pour l'hébergement des documentations ; elle-même facilement intégrable avec un repository type `github.io`,

Pour vous donner une idée du résultat, je vous invite à consulter un module de ma facture :

- les sources sur github sur https://github.com/parmentelat/asynciojobs, et notamment le sous-répertoire `sphinx`,
- et la documentation en ligne sur http://asynciojobs.readthedocs.io/.

Linter

Au delà de la vérification automatique de la présentation du code (PEP8), il existe un outil `pylint` qui fait de l'analyse de code source en vue de détecter des erreurs le plus tôt possible dans le cycle de développement.

En quelques mots, ma recommandation à ce sujet est que :

- tout d'abord, et comme dans tous les langages en fait, il est **très utile** de faire passer systématiquement son code dans un linter de ce genre ;
- idéalement on ne devrait commiter que du code qui passe cette étape ;
- cependant, il y a un petit travail de filtrage à faire au démarrage, car pylint détecte plusieurs centaines de sortes d'erreurs, du coup il convient de passer un moment à configurer l'outil pour qu'il en ignore certaines.

Dès que vous commencez à travailler sur des projets sérieux, vous devez utiliser un éditeur qui intègre et exécute automatiquement `pylint`. On peut notamment recommander PyCharm.

Type hints

Je voudrais citer enfin l'outil `mypy` qui est un complément crucial dans la mise en oeuvre des *type hints*.

Comme on l'a vu en Semaine 4 dans la séquence consacrée aux type hints, et en tous cas jusque Python-3.6, les annotations de typage que vous insérez éventuellement dans votre

code sont complètement ignorées de l'interpréteur.

Elles sont par contre analysées par l'outil `mypy` qui fournit une sorte de couche supplémentaire de *linter* et permet de détecter, ici encore, les éventuelles erreurs qui peuvent résulter notamment d'une mauvaise utilisation de telle ou telle librairie.

Conclusion

À nouveau cette liste n'est pas exhaustive, elle s'efforce simplement de guider vos premiers pas dans cet écosystème.

Je vous invite à creuser de votre coté les différents aspects qui, parmi cette liste, vous semblent les plus intéressants pour votre usage.