Enregistrements et instances

Complément - niveau basique

Un record implémenté comme une instance de classe

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

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

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

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

```
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):
        return "[{nom}, {age} ans, email:{email}]".\
            format(**vars(self))
```

Avant de détailler le code de cette classe, voyons comment on l'utiliserait :

```
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'),
]
print personnes
```

Le code de cette classe devrait être limpide à présent, sauf peut-être la dernière ligne de `__repr__`. On utilise ici la fonction *builtin* `vars` (https://docs.python.org/2/library/functions.html#vars) qui

retourne le dictionnaire des attributs d'un objet :

```
vars(personnes[0])
```

Ce qui, combiné avec un passage d'arguments avec la forme `**`, nous permet d'éviter la répétition des arguments à utiliser dans l'appel à `format`.

Un dictionnaire pour indexer les enregistrements

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

```
index_par_nom = {personne.nom: personne for personne in personnes}
```

De façon à pouvoir facilement localiser une personne

```
pierre = index_par_nom['pierre']
print pierre
```

Poursuivons l'analogie

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

```
pierre.age += 1
pierre
```

Cela dit, pour rester en ligne avec l'idée d'encapsulation telle que nous l'avons vue dans un complément récemment, il serait de beaucoup préférable de fournir une méthode `anniversaire` pour cela, de façon à pouvoir faire quelque chose comme

```
# attention cette cellule n'est pas évaluable
pierre.anniversaire()
```

Dans la pratique on définit presque toujours toutes les méthodes en même temps que la classe. Ce qui veut dire que l'on ajouterait tout simplement la méthode `anniversaire` dans le code qui définit la classe ci-dessus.

```
# Cette cellule *n'est pas* évaluable
# elle donne seulement un aperçu de comment
# on aurait pu écrire la classe Personne
# pour lui ajouter une méthode 'anniversaire'
class Personne:
    """Une personne possède un nom, un âge et une adresse e-mail"""

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

    def anniversaire(self):
```

```
        self.age += 1
```

Complément - niveau intermédiaire

Nous allons toutefois profiter de cet exemple pour montrer comment il est également possible d'ajouter cette méthode ***à la volée***, en tirant 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 comme ceci

```
def anniversaire(self):
    "Incrémente l'âge de la personne"
    self.age += 1

Personne.anniversaire = anniversaire
```

Ce code commence par définir une fonction un utilisant `def` et la signature de la méthode. Comme cette méthode ne prend pas d'argument, la fonction accepte un 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 `anniversaire`** 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

```
help(Personne)
```

Et on peut à présent utiliser cette méthode

```
pierre.anniversaire()
pierre
```