Travaux Pratiques 5

1 Introduction à la programmation orientée objet

Dans ce TP, nous allons faire de la *programmation orientée objet*. Pour illustrer ce concept extrêmement puissant, nous allons suivre pas à pas un exemple qui permet d'implanter un agenda très simplifié. Le code ci-dessous est disponible dans le fichier `TP5_Example.py`.

```
from datetime import datetime

class Event:
    def __init__(self, title, when):
        self.title = title
        self.when = when

    def change_time(self, new_time):
        self.when = new_time

lesson = Event("Swimming␣lesson", datetime(2016, 12, 15, 17, 00))
print(lesson.title)
lesson.change_time(datetime(2016, 12, 16, 17, 30))
```

Commençons par regarder les trois dernières lignes ci-dessus. `lesson` est défini comme un *objet* (en l'occurrence, une leçon de natation) appartenant à la *classe* `Event` (qui sert à représenter des événements). À cet objet sont attachées des variables qu'on appelle des *attributs*. Par exemple, quand on exécute `print(lesson.title)`, on constate que son attribut `lesson.title` contient la valeur `"Swimming␣lesson"`. En outre, cet objet bénéficie d'une fonction qu'on appelle aussi une *méthode*, permettant de changer le moment de l'événement : on l'appelle par la syntaxe `lesson.change_time(...)`.

Regardons, quelques lignes plus haut, la définition de la méthode `change_time`. Son premier argument, nommé traditionnellement `self`, est l'objet lui-même : quand on appellera cette méthode avec la syntaxe `lesson.change_time(new_time)`, il s'agira de l'argument placé avant le point, c'est-à-dire `lesson`. Le ou les arguments suivants (ici, `new_time`) seront placés, comme pour une fonction habituelle, dans la parenthèse.

Pour terminer, examinons la définition de la méthode `__init__`. Ce nom, qui est réservé en Python, sert à définir un *constructeur* permettant de créer un nouvel objet de la classe. Quand on utilise la syntaxe `lesson = Event(title, when)`, cela appelle automatiquement la fonction `__init__`. Que fait alors celle-ci ?

1. Elle crée un objet nommé `lesson` (qui est, à cet instant, une coquille vide).
2. Elle affecte à l'attribut `lesson.title` la valeur de l'argument `title`.
3. Elle affecte à l'attribut `lesson.when` la valeur de l'argument `when`.

On peut également définir une *sous-classe*, comme ci-dessous.

```
class Appointment(Event):
    def __init__(self, title, when, with_whom):
        super().__init__(title, when)
        self.with_whom = with_whom

lunch = Appointment(title="Restaurant", with_whom="Donald",
                    when=datetime(2015, 12, 25, 12, 0))
lunch.change_time(datetime(2015, 12, 25, 12, 30))
```

La première ligne définit `Appointment` comme une sous-classe de `Event`. Cela signifie qu'un objet de la classe `Appointment` appartient aussi à la classe plus générale `Event` : il *hérite* donc de toutes les méthodes qui sont définies dans `Event`. Par exemple, l'objet `lunch` peut utiliser la méthode `change_time`, qui est définie dans la classe mère `Event`. On pourrait aussi définir de nouvelles méthodes spécialement pour la classe `Appointment`, par exemple pour changer l'attribut `with_whom`.

Examinons la méthode `__init__` de la classe `Appointment`. La syntaxe `super().__init__(title, when)` signifie qu'on commence par appliquer la méthode `__init__` de la classe mère `Event` : on crée donc un objet et on lui attribue un titre et un temps. À la ligne suivante, on fait une action spécifique à la classe `Appointment` : on affecte à l'attribut `with_whom` de l'objet la valeur de l'argument `with_whom`.

On peut maintenant définir une classe `Calendar`, dont l'attribut qui nous intéresse le plus est une liste d'événements.

```
class Calendar:
    def __init__(self, event_list=None, owner=""):
        self.owner = owner
        if not event_list:
            self.event_list = []
        else:
            self.event_list = event_list

calendar = Calendar(owner="Daisy",
                    event_list=[lesson, lunch])
print(calendar.event_list[1].title)
```

Quelques remarques de syntaxe pour terminer.

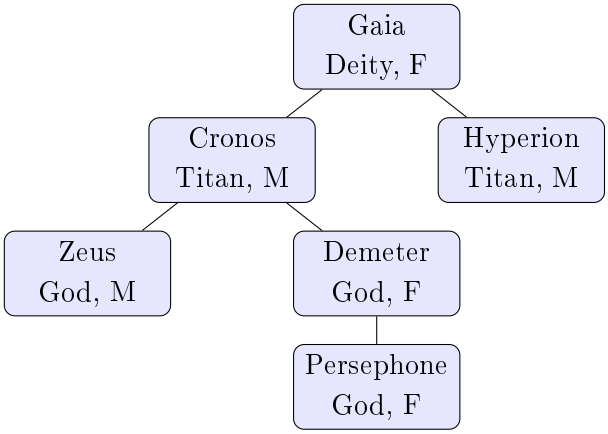
Soit une fonction définie par `def f(x, t)`. Dans la plupart des langages de programmation, il serait nécessaire d'appeler `f` en respectant scrupuleusement l'ordre des arguments. En Python, on peut s'en affranchir si on utilise les étiquettes des arguments quand on appelle `f` : ainsi, `f(x=42, t=51)` et `f(t=51, x=42)` sont équivalents et font la même chose que `f(42, 51)`. Dès qu'on manipule des fonctions possédant plus d'un ou deux arguments, il est recommandé d'employer la syntaxe avec étiquettes pour ne pas s'embêter avec l'ordre des arguments. C'est ce que nous avons fait ci-dessus pour définir `calendar`.

On peut aussi utiliser le symbole `=` dans la définition d'une fonction pour spécifier des valeurs par défaut. Ainsi, si on définit une fonction par `def f(x=42, t=0)`, cela autorise l'utilisateur à ne pas préciser la valeur de `x` et/ou de `t`. Le cas échéant, la fonction considérera que `x` vaut 42 et/ou que `t` vaut 0. Nous avons exploité cette possibilité pour définir le constructeur `__init__` de la classe `Calendar`.

Une ligne enfin peut vous sembler mystérieuse : `if not event_list`. Implicitement, une telle syntaxe convertit `event_list` en booléen : si c'est une liste non vide, elle est convertie en **True** ; si elle est égale à `None` ou à une liste vide, alors elle est convertie en **False**.

2 Opérations sur un arbre

Le but de ce TP est de définir une structure d'arbre et de la doter de quelques méthodes, en prenant comme exemple l'arbre ci-dessous. Chaque nœud possède un nom (par exemple Gaia), un statut (Deity, Titan ou God) et un genre (F ou M).



Exercice 1 Définition des classes Tree et MythologyTree

Question 1.1 Définissez une classe `Tree` et son constructeur. Un objet de cette classe aura deux attributs : `name` (par défaut, la chaîne de caractères vide) et une liste `children` d'enfants, qui sont eux-mêmes des arbres (par défaut, la liste vide).

Question 1.2 Définissez une sous-classe `MythologyTree` et son constructeur. Un objet de cette classe aura deux attributs supplémentaires : `status` et `gender` (les deux valant `None` par défaut).

Question 1.3 Définissez le graphe de l'exemple comme un objet noté `gaia`, appartenant à la classe `MythologyTree`.

Exercice 2 Hauteur de l'arbre

Dans la classe `Tree`, définissez une méthode `height` qui renvoie la hauteur de l'arbre, définie de la façon suivante :

— Si l'arbre est une feuille (c'est-à-dire s'il n'a pas d'enfant), alors sa hauteur est 0 ;
— Dans les autres cas, il s'agit de la hauteur maximale de ses enfants, à laquelle on ajoute 1.

Par exemple, `gaia.height()` doit renvoyer 3.

Exercice 3 Affichage de l'arbre

Dans la classe `Tree`, définissez une méthode `display` qui permet d'afficher l'arbre de l'exemple sous la forme suivante.

```
Gaia
    Cronos
        Zeus
        Demeter
            Persephone
    Hyperion
```

Indice : La méthode `display` prendra un argument optionnel `indent` (par défaut `""`).

Exercice 4 Taille de l'arbre (exercice bonus)

Dans la classe `Tree`, définissez une méthode `size` qui renvoie la taille de l'arbre, c'est-à-dire son nombre de nœuds.

Exercice 5 Nombre de feuilles (exercice bonus)

Dans la classe `Tree`, définissez une méthode `nb_leaves` qui renvoie le nombre de feuilles, c'est-à-dire de nœuds qui n'ont aucun enfant.

Exercice 6 Recherche par un parcours en profondeur (exercice bonus)

Dans la classe `Tree`, définissez une méthode `search_dfs` (Depth First Search) qui, étant donné un nom (donné sous forme de chaîne de caractères), renvoie le sous-arbre dont la racine porte ce nom. Si le nom demandé n'est pas présent dans l'arbre, cette méthode doit renvoyer `None`.

Testez les instructions suivantes.

```
search_result = gaia.search("Demeter")
print(search_result.status)
print(search_result.gender)
```

Exercice 7 Affichage par un parcours en largeur (exercice bonus)

Dans la classe `Tree`, définissez une méthode `display_bfs` (Breadth First Search) qui affiche le nom de la racine, puis le nom de tous les nœuds de profondeur 1, puis ceux de profondeur 2, etc.

Indice : définissez une liste nommée `fifo` que vous utiliserez comme file d'attente (First In First Out). Pour extraire le plus ancien élément de la file, utilisez `fifo.pop(0)`.