Labo d'introduction à l'informatique

pour les mathématiques

Yann Thorimbert

UNIVERSITÉ DE GENÈVE
CENTRE UNIVERSITAIRE D'INFORMATIQUE

Semaine 13

Les classes

Rappel : exemples de structures de données pertinentes

- Stocker les coefficients d'un polynôme : liste de nombres.
  `coeffs = [3, 0, -6, 5]`

- Stocker différents mots : liste de chaînes de caractères.
  `mots = ["Ballon", "Balle", "Frisbee"]`

- Stocker l'altitude de différentes villes : dictionnaire.
  `altitudes = {"Genève":400, "Barcelone":0, "La Paz":3650}`

Rappel : utilité de définir ses propres structures

- Stocker l'altitude, population, pays, indicatif téléphonique de différents villes ?
  ⇒ On peut imaginer toutes sortes de solutions.

- Exemple de solution possible :

```
nom = ['Genève', 'Barcelone', 'New York', 'Tokyo']
pop = [5e5, 2e6, 8e6, 14e6]
pays = ['Suisse', 'Espagne', 'USA', 'Japon']
etc…
```

  ⇒ mais dans tous les cas, cela constitue une solution *ad hoc*.
  Il est désirable de tout **regrouper** au sein d'une **variable unique** !

- Dans cette situation, déclarer une nouvelle **structure de données** peut être utile.

Rappel : Structures/Classes

- Structure : permet d’associer un nombre arbitraire de données à une variable unique.
- Certains langages (comme Python) permettent de définir des classes, qui sont des structures enrichies de fonctionnalités/comportements supplémentaires.
- Les différents champs de données associés à une classe se nomment les **attributs**.
- Les différentes fonctions associées à une classe se nomme les **méthodes**.
- Une classe est un peu un "patron" ou "chablon" permettant de définir un **type** de donnée. Lorsqu'un objet d'une certaine classe est effectivement créé, on dit que c'est une "**instance**" de ladite classe.

Structures/Classes

- L'étude des langages **orientés-objets** est une matière à part entière de l'informatique (pas le but de ce cours).
- En Python, on retrouve les objets un peu partout, mais le langage ne colle pas entièrement à la philosophie de l'orienté-objet pur et dur (e.g. comme Java).
- On peut très bien utiliser les objets d'une classe sans jamais définir de nouvelle classes soi-même. C'est ce que nous faisons depuis le début avec tous les objets Python (flagrant surtout avec `str`, `list` et `dict`).

Philosophie de l'orienté objet

- Ici on se restreint à des objets pour agréger des données et regrouper des comportements.
- On ne traite pas du mécanisme d'héritage
- On ne traite pas l'architecture globale des codes orientés objet.

Premier exemple

Dans le script qui sert à la mise à jour des bonus, chaque étudiant est un objet :

```
class Etudiant:
    def __init__(self, prenom, nom, email):
        self.prenom = prenom
        self.nom = nom
        self.email = email
        self.bonus = {i:0 for i in range(2,15)} #séries 1 à 14

#exemple de création d'une instance d'Etudiant:
a = Etudiant("Jean", "Dupont", "jean.dupont@etu.unige.ch")
a.bonus[3] = 1
a.bonus[12] = 1
```

Premier exemple

Dans le script qui sert à la mise à jour des bonus, chaque étudiant est un objet :

```
class Etudiant:
    def __init__(self, prenom, n
        self.prenom = prenom
        self.nom = nom
        self.email = email
        self.bonus = {i:0 for i in range(2,15)} #séries 1 à 14

#exemple de création d'une
a = Etudiant("Jean", "Dupon
a.bonus[3] = 1
a.bonus[12] = 1
```

Fonction d'initialisation spéciale dont le premier argument est une référence à l'étudiant créé.

Appel implicite à la méthode `__init__` attachée aux objets de la classe `Etudiant`.

Premier exemple

- `self` est une référence vers l'objet que l'on est en train de décrire.
- Certaines méthodes spéciales commencent par un double underscore (`__`) et sont utilisées par Python pour définir le comportement par défaut de notre objet. Exemple : `__init__(self, …)` est relatif à l'initialisation d'un objet.
- On peut tout à fait créer des méthodes non spéciales. Nous allons maintenant en définir une qui retourne un booléen disant si l'étudiant atteint les critères d'obtention du bonus.

Premier exemple

```
class Etudiant:
     def __init__(self, prenom, nom, email):
          self.prenom = prenom
          self.nom = nom
          self.email = email
          self.bonus = {i:0 for i in range(2,15)} #séries 1 à 14

     def has_bonus(self):
        if sum(self.bonus) >= 8:
            if sum(self.bonus[len(self.bonus)-6:]) >= 4:
                return True
        return False

#exemple de création d'une instance d'Etudiant:
a = Etudiant("Jean", "Dupont", "jean.dupont@etu.unige.ch")
a.bonus[3] = 1
a.bonus[12] = 1
print(a.has_bonus()) #affiche False
```

Second exemple

```
class Cube:
    def __init__(self, c):
        self.c = c

    def calcul_volume(self):
        return self.c ** 3

class Sphere:
    def __init__(self, r):
        self.r = r

    def calcul_volume(self):
        return (4/3) * pi * (self.r ** 3)

#Attention, on définit maintenant une fonction liée à aucune classe en particulier :
def calcul_poussee_archimede(mon_objet):
    rho_eau = 1000
    return rho_eau * 9.81 * mon_objet.calcul_volume()

sphere = Sphere(0.5)  # instance d'une sphère de rayon 0.5 m
print("Poussée d'Archimède sur la sphère immergée :", calcul_poussee_archimede(sphere), "N")
```

Second exemple

```
class Cube:
    def __init__(self, c):
        self.c = c

    def calcul_volume(self):
        return self.c ** 3

class Sphere:
    def __init__(self, r):
        self.r = r

    def calcul_volume(self):
        return (4/3) * pi * (self.r ** 3)

#Attention, on définit maintenant une fonction liée à au
def calcul_poussee_archimede(mon_objet):
    rho_eau = 1000
    return rho_eau * 9.81 * mon_objet.calcul_volume()

sphere = Sphere(0.5)  # instance d'une sphère de rayon 0.5 m
print("Poussée d'Archimède sur la sphère immergée :", calcul_poussee_archimede(sphere), "N")
```

Deux types d'objets différents mais respectant une certaine **interface** peuvent être utilisés dans un même contexte ⇒ On parle de "polymorphisme".

Ici, mon_objet peut être n'importe quoi qui implémente une méthode calcul_volume