Introduction à Python

Cours 2 : classes, librairies et gestion de fichiers.

Justin Cano

Polytechnique Montréal
Département de Génie Électrique.

21 mai 2021

1. Retour sur le devoir 1
2. Programmation orientée objet
   - Définition d'une classe
   - Méthodes Standards
   - Hiérarchie de classes
   - Exemple pratique
3. Gestion de bibliothèques
   - Invocation de ressources externes
   - L'outil d'installation de bibliothèques `PIP`
   - Comment créer ses propres bibliothèques et modules
4. Gestion automatisée de fichiers et de dates
   - Lecture et écriture de base (noyau)
   - Gestion de fichiers au format `.csv`
   - Parler à votre système d'exploitation

Plan de la section 1

Devoir 1 I

- Le devoir un consistait à compter les poules suivant une loi donnée au bout d'une période de N ans ;
- Une façon de voir le problème était de raisonner sur une liste comme suit :

$$P(t) = [p_0(t), p_1(t), p_2(t), p_3(t), p_4(t)]$$

avec t l'année en cours et $p_i(t)$ les poules ayant au mois i ans durant l'année t.

- Les poules reproductrices $r(t)$, c'est-à-dire qui ont au moins deux ans, créent 3 poussins pour chaque couple :

$$r(t) = \sum_{i=2}^{5} p_i(t) \Rightarrow p_0(t+1) = 3 \times \text{int}(r(t)/2)$$

Devoir 1 II

- Chaque année les poules prennent un an d'âge et les poules qui atteignent cinq ans meurent :

$$p_{i+1}(t+1) = p_i(t), \forall i \in [0,4],$$

ce qui **écrase** les poules de cinq ans ($p_5(t)$ n'intervient pas dans l'équation et sa valeur est remplacée par $p_4(t)$ au temps $t+1$) et ne touche pas aux poussins nouveaux nés.

- La résolution du problème se fait en itérant $N = 20$ fois les deux précédentes étapes en fournissant $p_0(0) = p_0 = 10$ le nombre de poussins donné dans l'énoncé et ainsi, obtenant la solution S voulue :

$$S = \sum_{i=0}^{5} p_i(20)$$

soit le nombres de poules obtenues au bout de vingt ans.

Plan de la section 2

Définition du concept

Definition

Un **objet** est un ensemble informatique d'**attributs** (variables) et de **méthodes** (ou fonctions) qui s'appliquent exclusivement aux premières. Les objets d'un même types sont dits membres d'une **classe**.

Exemple

Un gestionnaire de magasin crée la classe `Client`, comprenant les **attributs** «`nom`,`prénom`,`adresse`,`date-de-naissance`».

- Une méthode serait de calculer l'âge du client **à partir du dernier attribut** pour savoir si ce dernier a droit à une réduction ;
- Une autre serait de calculer en **fonction de son adresse** un coût de livraison ;
- Une autre, bien plus simple, serait de mettre à jour son adresse.

Classes en Python I

En python on définit une classe comme suit :

```
class NomDeLaClasse():
  #Constructeur
  def __init__(self,attributPublic_,attributPrive_):
    self.attributPublic = attributPublic_
    self.__attributPrive =  attributPrive_
  #Methodes
  def ma_methode(self,argument1,...)
    #code donnant resultat
    self.attribut_Public = attribut_modifie
    return eventuelRetour
```

Classes en Python II

Analyse de ce qui est écrit :

1. Le premier `def` est une méthode particulière `__init__(self)` : il s'agit d'un **constructeur** qui viendra créer un objet de type `NomDeLaClasse`, cette méthode peut être **surchargée** c'est-à-dire qu'on peut en avoir plusieurs au sein de la même classe. En effet, on peut, par exemple, créer un objet par défaut avec des attributs vides et ou déjà en préremplir à sa création.
2. L'appel à `self` dans chacune des méthodes signifie que **les méthodes invoquent l'objet en cours d'édition**. C'est-à-dire qu'elles peuvent modifier ou lire ses attributs en les invoquant comme suit : `self.attributs`.

Classes en Python III

3. La notation «.» est donc très importante en langage objet. On pourra traduire ceci sémantiquement comme «du» et marque une possession. A contrario les «()» sont donné comme «de», sans possession d'un objet ou d'appel implicite à ses propriété.

Exemple

`self.age` dans la classe `Client` est comprise comme l'«âge **du client**».

`age(annee,naissance)` retourne un laps de temps entre *annee* et `naissance`, c'est la durée **de** l'existence entre le temps de création x et la date y.
Remarquer qu'on a précisé tout les arguments nécessaires au calcul dans cette déclaration : elle est donc plus redondante que la notation condensée objet.

Classes en Python IV

4. Un **attribut** peut être privé ou public. On met deux *underscore* __ devant ce dernier pour préciser son caractère privé ; le mode public étant par défaut. L'attribut privé permet de réserver la modification de l'attribut *seulement* à l'objet. Seule une méthode de ce dernier pourra modifier ledit attribut.
5. Les méthodes invoquent l'objet `self`, elles peuvent (bien que se soit une mauvaise pratique de de ne pas l'initialiser à la création de l'objet) créer un nouvel attribut. Il faut alors les déclarer avec `def` en les subordonnant à la classe.

Les *setters*

Comme on vient de le voir, les attributs peuvent être **privés**, ce qui est une bonne pratique de code (impossibilité de modifier l'objet sans son «accord»). On peut définir alors les ***setters*** comme des **méthodes** qui permettent de mettre à jour un attribut.

```
def setAttribut(self,nouvelle_valeur):
  # Possibles tests de compatibilite ici
  self.attribut = nouvelle_valeur
```

Remarquer que ces dernières peuvent intégrer des tests qui évitent par exemple une erreur d'exécution.

Exemple

Rentrer un âge négatif mène à l'exécution une ligne de code `error(2)`, qui interrompra l'exécution programme, prévenant des bugs ultérieurs.

Les *getters*

Les ***getters*** sont le symétrique des **setters** : ils retournent la valeur de l'attribut.

```
def getAttribut(self):
  return self.attribut
```

Invocation de setter/getter : comment les utiliser ?

Exemple

On suppose une classe `King` avec un attribut `nom` qui dispose d'un jeu de *setter* et *getter* qui lui est propre.

```
Loulou = King() #creation d'un roi
Loulou.setNom("Louis XIV") # on ecrase la variable "nom"
nom_du_roi = Loulou.getNom() # on retourne la variable "nom"

>> print(nom_du_roi)
Louis XIV
```

Les méthodes d'impression (*print*)

Parfois, on veut définir une **méthode d'impression** automatisée des valeurs d'un objet pour rendre l'usage et le déverminage du code plus aisés.

```
def print(self):
  string = "Ma classe"+"Attribut 1 :" + str(self.attribut1)
  string += "Attribut 2 :" + str(self.attribut2)
  ...
  string += "Attribut N :" + str(self.attributN)
  print(string)
```

Exemple

Supposons une classe `EtatPiscine` qui stocke les variables `temperature` et `PH` d'une piscine. Une telle méthode retournerait par exemple :

```
maPiscine = EtatPiscine(25,7.75)
>> maPiscine.print()
Piscine | temperature = 25 degres | pH = 7.75
```

Destructeurs

Un **destructeur** permet de supprimer un objet et de libérer son espace mémoire alloué. Il est invoqué par la méthode `__del__` qui n'est d'ailleurs pas obligatoire de définir, le mot clé `del` servira dans l'invocation du destructeur.

Exemple

```
class DestroyObject():
  def __init__(self):
    print("Bonjour j'ai construit un objet")
  def __del__(self):
    print("Bonjour je l'ai supprime")
o = DestroyObject() #Creation
del o # Destruction
```

Composition

Une **composition** est un lien entre deux objets dont l'un est encapsulé dans l'autre. De plus, si on détruit l'objet encapsulant, alors on détruit l'encapsulé.

Exemple

```
class Classe1:
  def __init__(self)
    self.attribut1 = "blabla"
class Classe2:
  def __init__(self):
    self.attribut2 = self.Classe1()

c = Classe2() # c contient un objet Classe 1
```

Agrégation

Une **agrégation** est une relation d'agréagation dans laquelle la destruction de l'encapsulant n'implique pas la destruction de l'objet encapsulé.

Exemple

```
class Classe1:
  def __init__(self)
    self.attribut1 = "blabla"
class Classe2:
  def __init__(self,a):
    self.attribut2 = a

c1 = Classe1()
c2 = Classe2(c1) # c1 contient un objet Classe 1, le
                                    supprimer ne supprimera pas c1
```

Héritage I

L'**héritage** est le fait de pouvoir étendre des objets préexistants avec des fonctions spécifique.

Exemple

Une classe `Meuble` qui partagerait certaines propriétés d'une clase `Chaise`, mais pas toutes pourrait être coder sous forme d'héritage. `Chaise` hériterait de fonctionnalité de `Meuble`.

Syntaxe en `Python` :

```
class NomDeLaClasseHeritiere(NomClasseMere):
  #Definition des methodes et des attributs specifiques ici
```

On appelle **classe mère** la classe dont **l'héritière** hérite.

Héritage II

Exemple

Usage pour le cas précédent des meubles.

```
class Meuble:
  def __init__(self,prix_):
    self.prix_
class Chaise(Meuble):
  def print(self):
    print("Le prix de la chaise est de "+str(self.prix)+"
                                    euros.")
>> ma_chaise = Chaise(500)
>> ma_chaise.print()
Le prix de la chaise est de 500 euros.
```

On remarque dans l'exemple précédent que le prix a été initialisé dans la classe `Meuble` tandis que l'invocation de la méthode `print()` est partie inhérente de la classe `Chaise`.

Héritage III

Pour voir d'autres exemples, vous pouvez consulter les exemples bien construits de **Mouna Hamin** sur le site francophone `cours-gratuit.com` puisque nos projets et exercices n'invoquent que peu cette notion, qui est une notion efficace de programmer en langage objet.
`https://www.cours-gratuit.com/tutoriel-python/tutoriel-python-matriser-la-notion-dhritage`

Plan de la section 3

Invocation de bibliothèques

En Python, pour une grande variété de projets, il est préférable de ne pas réinventer la roue. Cela signifie en concret de réutiliser du code déjà préparé que le cœur de Python ne dispose pas par **défaut**.

Syntaxe d'import :

```
import librairie as alias
```

On importe une librairie à l'aide d'un alias plus sympthatique à écrire (mais non obligatoire).

Example

On veut importer la (sous)-libraire Pyplot de Matplotlib pour tracer un graphique :

```
import matplotlib.pyplot as plt
plt.draw() #Exemple d'invocation de la bibliotheque
```

C'est quand même plus commode d'invoquer plt !

Import de fonctionnalité spécifique

Parfois, seule une fonction spécifique est requise (sans potentiellement un alias) dans un code. Dans ce cas on écrit :

```
from librairie import fonctionnalite as alias
```

Example

Supposons que j'aie besoin de la fonction mathématique racine carrée (*square root*, `sqrt`) pour une application, elle n'est pas présente dans le cœur de `Python`, on peut écrire :

```
from math import sqrt as racine
>> racine(2)
1.4142135623730951
```

PIP (Python Package Index)

Souvent, la librairie n'est pas installée *a priori* et cela peut mener à l'énervement du codeur. Heureusement, un logiciel plutôt efficace de gestion de paquet existe pour Python : PIP.

Systèmes Unix[1] (ligne de commande)

```
sudo pip install LibrairieSouhaitee
```

Remarque : pip3 est la commande pour Python 3 (paquet géré par aptitude sur Ubuntu).

Systèmes Windows (ligne de commande)

```
py -m pip install LibrairieSouhaitee
```

Se référer à la documentation de la page de PIP https://pypi.org/project/pip/.

1. Linux et Mac

Créer vos propres modules

Ceci est utile si le projet de code est conséquent car cela permet de scinder en plusieurs fichiers l'exécutable. Supposons que vous ayez déclaré des fonctions et ou des classes au sein d'un fichier `monModule.py`. Alors on importera ce module comme suit :

```
import monModule
```

si on se trouve dans le même répertoire. La fonction `fonction(argument)` définie dans `monModule.py` s'invoquera ainsi :

```
monModule.fonction(argument)
```

On peut aussi importer une seule fonction du module et utiliser les alias comme vu précédemment.
Évidemment, ceci limite à un seul dossier l'invocation du module...

Créer et installer une bibliothèque I

La solution est simple : créer et installer une bibliothèque. **Création :** On va créer une arborescence de fichiers comme suit :

```
maBibliotheque/
    package/
        __init__.py
        monCode.py
        monAutreCode.py
    setup.py
```

`__init__.py` est un fichier vide qui est une convention pour l'installateur qui va comprendre qu'il s'agit d'un module ;

`monCode.py` est un fichier qui contient le code `Python` du module `package`. Remarquer qu'on peut scinder le code en plusieurs fichiers ;

Créer et installer une bibliothèque II

`setup.py` est un fichier d'installation qui servira pour l'installateur in doit contenir les renseignements necessaires :

```
from setuptools import setup
setup(name='maBibliotheque',
  version='0.1',
  description='ma bibliotheque cool',
  url='http://leSiteDeLaBiblio.com/',
  author='Mon Nom',
  author_email='your.name@example.com',
  license='MIT',
  packages=['package'],
  zip_safe=False)
```

Une fois ces étapes «administratives» remplies on peut utiliser `pip` pour installer notre librairie.

Créer et installer une bibliothèque III

Installation : Dans l'invite de commande de votre système d'exploitation, il faut premièrement se placer à l'aide de `cd` (*change directory*) à l'intérieur du bon répertoire, c'est à dire `MaBibliotheque/`. Ensuite, il faut dire à `pip` de l'installer.

`Systèmes Unix` :

```
sudo pip install -e .
```

`Systèmes Windows` :

```
py -m pip install -e .
```

l'option `-e` de `pip` permet d'incorporer les dernières modifications apportées à la bibliothèque dans l'installateur (version de *debug*). L'argument « . » signifie «le dossier actuel».

Plan de la section 4

Ouverture et création de fichiers

Le noyau de `Python` permet de gérer les fichiers avec la fonction `open()`. En premier lieu, on crée un objet fichier (*file*) :

```
f=open("monFichier.extension",option)
```

Le terme `option` est à remplacer par :

- `"x"` si on veut créer un nouveau fichier répondant au nom de `monFichier.extension` ;
- `"a"` si on veut ajouter des lignes à un fichier préexistant du même nom ;
- `"w"` si on veut écraser un fichier préexistant du même nom ;
- `"r"` si on veut lire un fichier préexistant.

Note : on peut combiner certaines opérations, comme "rw" (lecture et écriture) par exemple.

Écriture de fichiers

Pour écrire dans un fichier, on va utiliser la méthode `write()` de la classe `file` :

```
f.write("La ligne que je veux ecrire dans le fichier.")
```

Elle prend en argument un *string*.
On fermera le fichier à la fin des opérations sur ce dernier avec la méthode `close()` (ceci détruit l'objet par la même occasion).

```
f.close()
```

Lecture automatisée de fichiers

On peut utiliser pour ce faire la méthode `read()` :

```
string = f.read()
```

La méthode retourne un string contenant tout le contenu du fichier.
Le souci est que souvent, on n'est pas intéressé par le fichier en entier mais le lire ligne par ligne. Pour ce faire, on peut utiliser la fonction `readlines()`

```
tab = f.readlines()
```

Cela permet de lire les fichiers lignes par lignes en les ajoutant dans un tableau de string de taille N, nombre de lignes du fichier. La ligne i (**rappel** $0 \leq i \leq N-1$: on commence les indices par zéro en `Python`.) est stockée dans la case i du tableau.
Mais cela peut ne pas être suffisant...

Le module csv I

Les fichiers .csv (*Comma Separated Values*, *i.e.*, valeurs séparées par des virgules) sont une classe de fichier qui sont la base du stockage de toute matrice de données. On peut donc utiliser le module csv de Python pour les gérer efficacement.

```
import csv
with open("monFicher.csv", "r") as fichierCSV:
  lecteur_csv = csv.reader(fichierCSV,delimiter=",")
  for ligne in lecteur_csv:
    ligne = [entry.decode("utf8") for entry in ligne]
```

Explication :

- L'import du module se fait au début du fichier ;
- La structure with permet de ne pas exécuter le code si le fichier n'a pas été trouvé et ainsi éviter potentiellement une erreur ;
- On ouvre le fichier ainsi ;

Le module csv II

- `csv.reader` renvoie un tableau. Il s'agit du *string* du fichier traité ligne par ligne en cherchant un **délimiteur** de champ. Traditionnellement le délimiteur est une virgule « , » comme dans le code ci-dessus mais on peut en choisir un autre. Il est important de le préciser car le «décryptage» de la structure de donnée dépend du choix de ce dernier ;
- la boucle `for` est une précaution : elle précise que chaque ligne doit être comprise comme étant un `string` codé en `utf8`, ce qui peut être salvateur pour les accents de notre belle langue française.

Commandes du paquet OS (en bref)

Ce paquet permet de parler au système d'exploitation indépendamment de ce dernier. Par exemple, il permet de gérer les fichiers : très utile pour mener des sauvegardes automatisées donc. On l'importe ainsi :

```
import os
```

`os.chdir(path)` change le répertoire dans lequel on opère ;

`os.listdir()` renvoie dans un tableau de `string` tous les noms des fichiers ;

`os.getcwd()` renvoie dans un *string* le chemin du répertoire où on est ;

os.mkdir() créé un répertoire automatiquement ;

os.remove(path) supprime un fichier donné en argument ;

os.rmdir(path) supprime un dossier donné en argument ;

os.system(command) exécute la commande `command` dans votre invite de commande.

Date et heure du système

Afin d'obtenir la date et l'heure actuelle de l'ordinateur, on peut faire intervenir le paquet `datetime.date`.

```
from datetime import date
date_actuelle = date.today()
jour = date_actuelle.day
mois = date_actuelle.month
annee = date_actuelle.year
```

Modifier les formats : pour l'export de dates dans des *string*, on peut modifier l'ordre de la chaîne de caractère à notre guise. Par exemple pour obtenir le format français, il suffit de faire :

```
date_actuelle = date_actuelle.strftime("%d-%m-%y")
```

Application I

Exemple

On veut créer un générateur de verlan, qui vient inverser les mots d'une phrase et stocker ces derniers dans un fichier texte avec la date du jour comme nom dans le répertoire `verlan`.

Le fichier comportant la phrase à l'endroit est `sentence.txt` il contient l'unique ligne :

```
Agathe sac de patate
```

le délimiteur est donc l'espace entre deux mots, soit le *string* `" "`, on peut utiliser le paquet `csv` de manière détournée ici.
Ainsi on peut commencer à programmer avec les trois bibliothèques vues précédemment.

Application II

```
import csv
import os
from datetime import date

# Operation de lecture
with open("sentence.txt","r") as f:
  lecteur = csv.reader(f,delimiter=" ")
  ligne = next(lecteur) # La ligne est seule dans le fichier
  ligne.reverse() # On inverse la liste
  f.close() # On ferme le fichier

# Operation sur le repertoire et creation du nom de fichier
if not os.path.exists("verlan"): # Test d'existence
  os.mkdir("verlan") #creation du repertoire
os.chdir("verlan") #on va dans le repertoire
nom_fichier = str(date.today()) #nom du fichier
```

Application III

```
# Operation d'ecriture
f = open(nom_fichier+".txt","w") #Creation du nouveau
                                 fichier
sw = "" # String vide
for s in ligne:
  sw+=" "+s # Compression du tableau dans un seul string
f.write(sw) # On ecrit le texte en entier dans le fichier
f.close() # On ferme le fichier
```

Et on obtient dans le fichier 2021-05-17.txt (date à laquelle j'ai créé ce tuto) :

patate de sac Agathe