Introduction à Python

V. Monbet
(document fortement inspiré du livre de R. Cordeau
et des supports de cours de J.-P. Becirspahic)

UFR de Mathématiques
Université de Rennes 1

13 novembre 2018

Outline

Paradigme de programmation procédurale

- En programmation procédurale, on sépare code et données.
- Concrètement, on utilise
  - des variables (globales) qui contiennent les données et sont utilisées par les fonctions
  - des fonctions qui peuvent modifier les variables et les passer à d'autres fonctions.

Limites de la programmation procédurale

Exemple

- Dans une entreprise située à Biarritz, Robert (num de sécu 10573123456), est employé sur un poste de technicien

```
# Version 1 : variables globales
# Robert
num_secu_robert = 10573123456
nom_robert = "Robert"
qualification_robert = "Technicien"
lieudetravail_robert = "Biarritz"
```

- Pour ajouter un employé, on doit créer de nouvelles variables...

```
# Bernard
num_secu_bernard = 1881111001
nom_bernard = "Bernard"
qualification_bernard = "Ingenieur"
lieudetravail_bernard = "Biarritz"
```

Le code devient vite fastidieux !

Limites de la programmation procédurale

Exemple

- Une solution alternative consiste à utiliser des conténeurs (ou listes)

```
# Version 2 : utilisation de conteneurs
tab_robert = [10573123456 ,"Robert","Technicien","Biarritz"]
tab_bernard = [1881111001, "Bernard","Ingenieur","Biarritz"]
tab =[ ]
tab.append(tab_robert)
tab.append(tab_bernard)
print(tab[0][0])
```

Sans être le concepteur, comment interpréter facilement les données de chaque conteneur ? Cela reste délicat. Le paradigme objet propose des réponses.

Vers la programmation orientée objet

Classe

- En programmation orientée objet, nous pouvons définir une structure de données particulière par la notion de classe.

```
# Version 3 : utilisation de classes
class Employe:
    num_secu
    nom
    qualification
    lieudetravail
```

- On appelle classe un ensemble d'objets partageant certaines propriétés. Il s'agit d'un concept abstrait (comme le plan d'une maison).
- Exemple : la classe *Voiture* possède les propriétés suivantes (les attributs) :
  - couleur
  - puissance
- Les attributs sont les entités qui définissent les propriétés d'objet.

Vers la programmation orientée objet

Objet

- Un objet est une définition de caractéristiques propres à un élément particulier. Il s'agit d'un élément concret qui contient les propriétés de sa classe, comme une maison qui suit les plans définis préalablement.
- Exemple : instance de la classe Voiture
  1. Voiture "Clio 2007 version roland garros"
  - couleur : verte
  - puissance : 70 Ch
  2. Voiture "307 blue lagoon"
  - couleur : bleue
  - puissance : 90 Ch
- On appelle instance d'une classe, un objet avec un comportement et un état, tous deux définis par la classe.

Vers la programmation orientée objet

Objet, exemple

- Mise en place d'objets concrets

```
# Robert
e1 = Employe() # objet e1 : instance de la classe Employe
e1.num_secu = 10573123456
e1.num = "Robert"
e1.qualification = "Technicien"
e1.lieudetravail = "Biarritz"

# Bernard
e2 = Employe() # objet e2 : instance de la classe Employe
e2.num_secu = 1881111001
e2.num = "Bernard"
e2.qualification = "Ingenieur"
e2.lieudetravail = "Biarritz"
```

Vers la programmation orientée objet

Constructeur

- En pratique, on doit initialiser les attributs par une valeur par défaut. Nous faisons alors appelle à une fonction particulière appelée constructeur.

```
class Employe:
    def __init__(self): # constructeur de la classe Employe
        self.num_secu = 000000000000
        self.nom = "no_name"
        self.qualification = "novice"
        self.lieudetravail = "Paris"
e3 = Employe()
print( e3.num_secu,e3.nom)
```

- En PYTHON, le constructeur s'appelle en utilisant la fonction **__init__**.
- Le premier argument est toujours **self**.
- En PYTHON, la fonction **__init__** est appelée implicitement à la création d'un objet.

Vers la programmation orientée objet

Constructeur

- La fonction **__init__** peut prendre plusieurs arguments.

```
class Employe:
    def __init__(self,
                         num_secu,
                         nom,
                         qualification,
                         lieudetravail):
        self.num_secu = num_secu
        self.nom = nom
        self.qualification = qualification
        self.lieudetravail = lieudetravail
```

Vers la programmation orientée objet

Constructeur

- La fonction **__init__** peut prendre plusieurs arguments.
- On peut alors utiliser les arguments du constructeur pour mettre des valeurs par défaut et instancier des objets "plus facilement" en ne précisant que certaines valeurs.

```
class Employe:
    def __init__(self,
                         num_secu = 00000000,
                         nom = "no_name",
                         qualification = "novice",
                         lieudetravail = "Paris"):
        self.num_secu = num_secu
        self.nom = nom
        self.qualificaition = qualification
        self.lieudetravail = lieudetravail
e1 = Employe(lieudetravail="Brest")
```

Vers la programmation orientée objet

Exercice

- Créer une classe `Voiture` possédant les attributs, marque, couleur et puissance.
- Écrire son constructeur avec la possibilité de préciser les valeurs de certains attributs à l'instanciation.
- Instancier 3 objets de la classe Voiture.

Concepts fondateurs

Qu'est-ce qu'une classe ?

- Pour créer un objet, il faut d'abord créer une classe !
- Exemple : pour construire une maison,
  on a besoin d'un plan d'architecte :
  - classe = plan,
  - l'objet = maison.
- "Créer une classe", c'est dessiner les plans de l'objet.
- Une classe est la description d'une famille d'objets qui ont la même structure et les mêmes comportements. Une classe est une sorte de moule à partir duquel sont générés les objets.

Les classes

- Une classe est constituée

  1. d'un ensemble de variables (appelées attributs) qui décrivent la structure des objets ;

  2. d'un ensemble de fonctions (appelées méthodes) qui sont applicables aux objets, et qui décrivent leurs comportements.

Concepts fondateurs

Qu'est-ce qu'un objet ?

- Une classe est une abstraction, elle définit une infinité d'objets.
- Un objet est caractérisé par un état, des comportements et une identité.
  Concrètement, il s'agit :
  - Attributs
  - Méthodes
  - Identité
- On créé des classes pour définir le fonctionnement des objets. On utilise des objets.
- L'état d'un objet
  - regroupe les valeurs instantanées de tous ses attributs
  - évolue au cours du temps
  - est la conséquence de ses comportements passés

Objets

Attributs

- Les attributs sont les caractéristiques de l'objet. Ce sont des variables stockant des informations d'état de l'objet.

```
# definition classe
class Voiture:
    def __init__(self,v1="nomarque",v2="nocolor",v3=0)
        self.marque = v1
        self.couleur = v2
        self.reserveEssence = v3
# Objet
e1 = Voiture()
# Etat
e1.marque = "peugeot"
e1.couleur = "noire"
e1.reserveEssence = 30
```

Objets

Comportement

- Le comportement d'un objet regroupe toutes ses compétences, il décrit les actions et les réactions d'un objet. Il se représente sous la forme de méthodes.

```
class Voiture:
    def __init__(self,v1="nomarque",v2="nocolor",v3=0):
        self.marque = v1
        self.couleur = v2
        self.reserveEssence = v3

    def demarrer(self): # 1ere methode
        print("je demarre")
    def arreter(self): # 2nde methode
        print("je m'arrete")
# Objet
e1 = Voiture()
e1.demarrer()
e1.arreter()
```

Objets

Comportement

- On peut bien sûr complexifier les méthodes, par exemple

```
class Voiture:
    def __init__(self,v1="nomarque",v2="nocolor",v3=0):
        self.marque = v1
        self.couleur = v2
        self.reserveEssence = v3

    def demarrer(self):
        if self.reserveEssence>0:
            print("je demarre")
        else :
            print("je ne peux pas demarrer")
    def arreter(self):
        print("je m'arrete")
    def rouler(self,nbKm):
        self.reserveEssence = self.reserveEssence - 5*nbKm/100
e1 = Voiture()
e1.reserveEssence = 30
e1.demarrer()
e1.rouler(100)
print(e1.reserveEssence)
```

Objets

Identité

- L'identité est propre à l'objet et le caractérise. Elle permet de distinguer tout objet indépendamment de son état
- En pratique on peut distinguer les objets par leur nom, un numéro de référence, ...

Classe et objet

Exercice

Que fait le code ci-dessous ?

```
class Animal:

    "A simple example class Animal with its name, weight and age"

    def __init__(self, name, weight, age):  # constructor
        self.name = name
        self.weight = weight
        self.age = age

    def birthyear(self):  # method
        import datetime
        now = datetime.datetime.now()
        return now.year - self.age


dog = Animal('Dog', 18, 4) # Instance
```

Affichages

Pour afficher les attributs d'une instance, on utilise le dictionnaire :

```
dog.__dict__
Out[3]: {'age': 4, 'name': 'Dog', 'weight': 18}
```

Pour faire un affichage plus élégant

```
print(f' {dog.name}: {dog.weight} Kg, {dog.age} years')
 Dog: 18 Kg, 4 years
```

Objets

Exercice : gestion de stock

Écrire une classe telle que

- Un Article du stock est défini par 4 champs :
  - sa référence (numéro)
  - sa désignation (texte)
  - son prix HT
  - sa quantité (nombre d'articles disponibles)
- Pour manipuler ces champs, les services suivants sont fournis :
  - prix TTC
  - prix Transport (taxe 5% prix HT)
  - retirer
  - ajouter

Classe, objet, attributs

Exercices

1. Différencier les classes des objets (instances de classe) dans les propositions suivantes : Footballeur, Zidane, Basket, Sport, Voiture, Clio, Rouge, Bleu, Homme, Couleur, Mégane, Manadou.

2. Les étudiants sont caractérisés par leur numéro d'étudiant, leur nom, leur prénom, leur date de naissance et leur adresse.
   Identifier les attributs de la classe `Etudiant`.

Encapsulation

Principe

- Exemple introductif : classe `Compte` avec un attribut solde

```
class Compte:
    def __init__(self):
        self.solde = 0
    def affiche(self):
        print("le solde de votre compte est de ", self.solde)

c = Compte()
c.affiche()
c.solde = 100000
```

- N'importe qui ayant accès à une instance de la classe `Compte` peut modifier le solde. Ceci n'est pas acceptable.

Encapsulation

Principe

- Lors de la définition d'une classe il est possible de restreindre voire d'interdire l'accès (aux objets des autres classes) à des attributs ou méthodes des instances de cette classe. Comment ?
- Lors de la déclaration d'attributs ou méthodes, en précisant leur utilisation :
  privée ie accessible uniquement depuis "l'intérieur" de la classe ;
  public ie accessible par tout le monde.

```
class Test:
    def __init__(self,valeurPrivee,valeurPublic):
        self.__valeurPrivee = valeurPrivee
        self.valeurPublic = valeurPublic
    def f1(self):
        ...
    def __f2(self):
        ...
t1 = Test()
print(t1.valeurPublic) # OK
print(t1.__valeurPrivee) # ERREUR
```

Encapsulation

Principe

- Dans l'exemple du compte en banque

```
class Compte:
    def __init__(self):
        self.__solde = 0
    def affiche(self):
        print("le solde de votre compte est de ", self.solde)
    def debit(self,laValeur):
        if self.__solde - laValeur>0:
            self.__solde = self.__solde-laValeur
        else :
            print("Solde insuffisant")
```

Encapsulation

Intérêt

- L'encapsulation est donc l"idée de cacher l'information contenue dans un objet et de ne proposer que des méthodes de manipulation de cet objet. Ainsi, les propriétés et axiomes associés aux informations contenues dans l'objet seront assurés/validés par les méthodes de l'objet et ne seront plus de la responsabilité de l'utilisateur extérieur.

- L'objet est ainsi vu de l'extérieur comme une boîte noire ayant certaines propriétés et ayant un comportement spécifié. La manière dont ces propriétés ont été implémentées est alors cachée aux utilisateurs de la classe.

Encapsulation

Règles à respecter

- Rendre privés les attributs caractérisant l'état de l'objet. Si nous voulons respecter le concept d'encapsulation, nous devons donc conserver la portée de cet attribut et définir des accesseurs pour y avoir accès.

- Fournir des méthodes publiques permettant d'accéder à l'attribut et éventuellement de le modifier.

Exemple

```
class Temperature:
    '''
    La classe Temperature represente une grandeur physique
    liee a la notion de chaud et froid.
    '''
    # Constructeur et destructeur
    def __init__(self,laValeur=0):
        self.__valeur = laValeur
    def __del__(self):
        print("Destruction")

    # Methodes publiques : acces aux attributs
    def getValeur(self): # accesseur
        return self.__valeur
    def setValeur(self,value): # mutateur
        self.__valeur = value
    # Methodes publiques : operations
    def getValeurC(self):
        return self.__valeur - 273.16
    def getValeurF(self):
        return 1.8*self.getValeurC()+32
```

Intérêt d'utiliser des accesseurs

Lorsque vous utilisez des accesseurs, vous n'êtes pas obligé de vous contenter de lire un attribut dans un getter ou de lui affecter une nouvelle valeur dans un setter : en effet, il est tout à fait possible d'ajouter du code supplémentaire, voire de ne pas manipuler d'attribut en particulier !

```
class Voiture:
    '''
    La classe Voiture est definie par sa largeur
    '''
    # Constructeur et destructeur
    def __init__(self,laValeur=0):
        self.__largeur = laValeur
    def __del__(self):
        print("Destruction")
    # Methodes publiques : acces aux attributs
    def getLargeur(self): # accesseur
        return self.__largeur
    def setLargeur(self,value): # mutateur
        self.__largeur = value
    # Methodes publiques : operations
    def getValeurC(self):
        return self.__valeur
    def MettreAJour(self):
        ...
```

Intérêt d'utiliser des accesseurs

Imaginons que dans notre classe, nous disposions d'une méthode `mettreAJour()` qui redessine l'affichage de notre objet en fonction de la valeur de l'attribut.
Maintenant, nous aimerions limiter les valeurs possibles de l'attribut largeur ; disons qu'il doit être supérieur à 100 et inférieur à 200.

```
    def setLargeur(self,value): # mutateur
            self.__largeur = value
            if self.getLargeur()<100 :
                self.__largeur = 100
            else if self.getLargeur()>200:
                self.__largeur = 200
            self.mettreAJour()
```

Denouveau des affichages

La méthode privée `__repr__` permet d'associer une fonction **`print`** spéciale à la classe.

```
class Dog(Animal):  # Parent class is defined here

    " Derived from MyClass with k attribute "

    def __init__(self, name, weight, age):  # constructor
        self.name = name
        self.weight = weight
        self.age = age

    def birthyear(self):  # method
        import datetime
        now = datetime.datetime.now()
        return now.year - self.age

    def __repr__(self):
        return f"{self.__class__.__name__}({self.name}, {self.weight},


beagle = Dog('Jack', 9.0, 1)
print(beagle)
  Dog(Jack, 9.0, 1)
```

Exercice

Décrire ce que fait le code ci-dessous puis le compléter en ajoutant une méthode `diff(n)` qui calcule la dérivée d'ordre n en un point x.

```
class Polynomial:

    " Class representing a polynom P(x) -> c_0+c_1*x+c_2*x^2+..."

    def __init__(self, coeffs):
        self.coeffs = coeffs

    def __call__(self, x):
        return sum([coef*x**exp for exp,coef in enumerate(self.coeffs)])
```