Programmation Orientée Objet en Python

DIU Enseigner l'Informatique au Lycée

B. Wack

UFR IM2AG, Université Grenoble Alpes

août 2020

Remerciements : Guillaume Huard

Plan

Notions de langages à objets

Meilleur contrôle des attributs

Utilisation pour le chaînage

Héritage, sous-classes

Plan

Notions de langages à objets

Meilleur contrôle des attributs

Utilisation pour le chaînage

Héritage, sous-classes

Notion de classe

Classe = modèle permettant de créer des objets, regroupe des définitions

- d'attributs : données stockées dans les objets de la classe (accès sous la forme `objet.attribut`)
- de méthodes : opérations applicables sur les objets de la classe

Notion de classe

Classe = modèle permettant de créer des objets, regroupe des définitions

- d'attributs : données stockées dans les objets de la classe (accès sous la forme `objet.attribut`)
- de méthodes : opérations applicables sur les objets de la classe

Un exemple : nombres complexes

```
import math

class Complexe:
    def module(self):
        return math.sqrt( self.re**2 + self.im**2 )

    def argument(self):
        return math.atan( self.im / self.re )

    def translater(self, u, v):
        self.re += u
        self.im += v
```

En Python, une classe ne déclare pas explicitement ses attributs.

Instanciation d'une classe

Un **appel** de la classe (comme si c'était une fonction) crée et renvoie un objet qui est une instance de cette classe.

```
>>> c = Complexe()
```

Instanciation d'une classe

Un **appel** de la classe (comme si c'était une fonction) crée et renvoie un objet qui est une instance de cette classe.

```
>>> c = Complexe()
```

Le nouvel objet ainsi créé

- est une référence à l'aide de laquelle on peut appeler les méthodes de la classe
- mais ne contient en général pas encore d'attribut
- on peut lui en ajouter à volonté

```
>>> c.re
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Complexe instance has no attribute 're'

>>> c.re = 1
```

Remarque : c'est en fait la seule vraie notion de type de Python !

Utilisation dégénérée

Par rapport aux langages non orienté objet, les **attributs** sont similaires aux *champs* d'un type enregistrement.

```
class mon_enregistrement:
    pass


e = mon_enregistrement()
e.x = ...
e.y = ...
e.z = ...
```

(Ici `pass` est nécessaire pour des raisons syntaxiques.)

Les méthodes

```
class Complexe:
    def module(self):
        return math.sqrt( self.re**2 + self.im**2 )
    ...
```

À première vue une méthode n'est qu'une fonction mais :

- son premier argument représente obligatoirement l'objet à partir duquel elle est appelée
- par convention, ce premier argument est appelé `self`

Les méthodes

```
class Complexe:
    def module(self):
        return math.sqrt( self.re**2 + self.im**2 )
    ...
```

À première vue une méthode n'est qu'une fonction mais :

- son premier argument représente obligatoirement l'objet à partir duquel elle est appelée
- par convention, ce premier argument est appelé `self`
- une méthode doit être accédée comme un attribut (`objet.methode`), et l'objet est passé *implicitement* comme premier argument

```
>>> c = Complexe()
>>> c.module()
AttributeError: Complexe instance has no attribute 're'

>>> c.re = 1
>>> c.im = 1
>>> c.module()
1.414
```

Initialisation des objets

Par défaut un objet est créé sans attributs mais :

- si elle est définie dans la classe, la fonction `__init__` est immédiatement appelée sur l'objet
- elle joue donc le rôle de *constructeur* de la classe
- le cas échéant on peut lui passer des arguments supplémentaires

Initialisation des objets

Par défaut un objet est créé sans attributs mais :

- si elle est définie dans la classe, la fonction `__init__` est immédiatement appelée sur l'objet
- elle joue donc le rôle de *constructeur* de la classe
- le cas échéant on peut lui passer des arguments supplémentaires

```
class Complexe:
    def __init__(self, re, im):
        self.re = re
        self.im = im

    def module(self):
        ...

>>> c = Complexe(1,1)
>>> c.module()
1.414
```

Initialisation des objets

Par défaut un objet est créé sans attributs mais :

- si elle est définie dans la classe, la fonction `__init__` est immédiatement appelée sur l'objet
- elle joue donc le rôle de *constructeur* de la classe
- le cas échéant on peut lui passer des arguments supplémentaires

```
class Complexe:
    def __init__(self, re, im):
        self.re = re
        self.im = im

    def module(self):
        ...

>>> c = Complexe(1,1)
>>> c.module()
1.414
```

Remarque : Un objet n'a pas besoin d'être détruit (*garbage collector*).

Surcharge du constructeur

Il peut être pratique de définir plusieurs variantes de constructeur :

En Java :

```
File() {
    buffer = new int[10];
    taille = 0;
}

File(int capacite) {
    buffer = new int[capacite];
    taille = 0;
}
```

Surcharge du constructeur

Il peut être pratique de définir plusieurs variantes de constructeur :

Pas de vraie notion de surcharge en Python (méthodes ayant le même nom mais des signatures différentes) car pas de typage suffisamment fort pour les distinguer

En revanche, on peut définir des valeurs par défaut pour les arguments des fonctions.

En Python :

```
class File:
    def __init__(self, capacite=10):
        self.buffer = [None]*capacite
        self.taille = 0
```

Plan

Notions de langages à objets

Meilleur contrôle des attributs

Utilisation pour le chaînage

Héritage, sous-classes

Visibilité des attributs

Principe d'encapsulation : on aimerait choisir quels attributs et méthodes sont visibles de l'utilisateur (publics) ou à l'intérieur de la classe seulement (privés).
Pas de telle notion en Python mais...

Visibilité des attributs

Principe d'encapsulation : on aimerait choisir quels attributs et méthodes sont visibles de l'utilisateur (publics) ou à l'intérieur de la classe seulement (privés).
Pas de telle notion en Python mais...

Par convention les noms commençant par _ :

- doivent être considérés comme non-publics (mais ne sont pas strictement privés)
- ne doivent donc jamais être utilisés directement ni redéfinis (on risque de casser les propriétés de la classe, et leur implémentation est susceptible de changer)

Visibilité des attributs

Principe d'encapsulation : on aimerait choisir quels attributs et méthodes sont visibles de l'utilisateur (publics) ou à l'intérieur de la classe seulement (privés).
Pas de telle notion en Python mais...

Par convention les noms commençant par _ :

- doivent être considérés comme non-publics (mais ne sont pas strictement privés)
- ne doivent donc jamais être utilisés directement ni redéfinis (on risque de casser les propriétés de la classe, et leur implémentation est susceptible de changer)

Et les noms commençant par __ :

- sont automatiquement préfixés par le nom de leur classe
- leur utilisation est donc compliquée (mais pas impossible)
- ils ne polluent pas l'espace de nom

Exemple

```
import math

class Complexe:
    def __init__(self, re, im):
        self._re = re
        self._im = im

    def reel(self):
        return self._re

    def imag(self):
        return self._im

    def module(self):
        return math.sqrt( self._re**2 + self._im**2 )

    def argument(self):
        return math.atan( self._im / self._re )
```

Exemple

```
import math

class Complexe:
    def __init__(self, re, im):
        self._mod = re**2 + im**2
        self._arg = math.atan( im / re )

    def reel(self):
        return self._mod * math.cos(self._arg)

    def imag(self):
        return self._mod * math.sin(self._arg)

    def module(self):
        return self._mod

    def argument(self):
        return self._arg
```

Attribut de classe, attribut d'instance

Une classe peut en fait définir des *attributs de classe* (hors de ses méthodes) :

```
class Toto:
    truc = []
```

- ceux-ci seront **partagés** entre les instances de la classe
- lorsqu'un attribut n'est pas trouvé dans une instance, alors on va chercher cet attribut en second recours dans la classe correspondante

Attribut de classe, attribut d'instance

Une classe peut en fait définir des *attributs de classe* (hors de ses méthodes) :

```
class Toto:
    truc = []
```

- ceux-ci seront **partagés** entre les instances de la classe
- lorsqu'un attribut n'est pas trouvé dans une instance, alors on va chercher cet attribut en second recours dans la classe correspondante

Lorsqu'une instance **modifie** un attribut de classe (mutable), alors elle le fait pour **toutes les instances**.

```
>>> t1=Toto()
>>> t2=Toto()
>>> t1.truc.append(1)
>>> t2.truc.append(2)
>>> t1.truc
[1, 2]
```

Plan

Notions de langages à objets

Meilleur contrôle des attributs

Utilisation pour le chaînage

Héritage, sous-classes

Un exemple de liste chaînée en Python

```
class Maillon:
    def __init__(self, e, suiv):
        self.element = e
        self.suivant = suiv
```

Un exemple de liste chaînée en Python

```
class Maillon:
    def __init__(self, e, suiv):
        self.element = e
        self.suivant = suiv


class ListeChainee:
    # Constructeur (on mémorise une référence vers un maillon
    # pour pouvoir modifier la liste, meme vide, par effet de bord)
    def __init__(self):
        self.sommet = None
```

Un exemple de liste chaînée en Python

```
class Maillon:
    def __init__(self, e, suiv):
        self.element = e
        self.suivant = suiv


class ListeChainee:
    # Constructeur (on mémorise une référence vers un maillon
    # pour pouvoir modifier la liste, meme vide, par effet de bord)
    def __init__(self):
        self.sommet = None

    # Operateurs
    def ajout_tete(self, element):
        m = Maillon(element, self.sommet)
        self.sommet = m
    def suppression_tete(self):
        e = self.sommet.element
        self.sommet = self.sommet.suivant
        return e
    def est_liste_vide(self):
        return self.sommet==None
```

Comparaison avec Java

```
class Maillon {
    int element;
    Maillon suivant;

    Maillon(int e, Maillon s) { ...
}

class ListeChainee {
    Maillon sommet;

    // Constructeur (vide par défaut)

    // Operateurs
    void ajout_tete(int element) { ...
    int suppression_tete() { ...
    boolean est_liste_vide() { ...

    // Pas de destructeur
}
```

Exemple

```
p = ListeChainee()
p.ajout_tete(1)
p.ajout_tete(2)
print(p.suppression_tete())
p.suppression_tete()
print(p.est_liste_vide())
```

Plan

Notions de langages à objets

Meilleur contrôle des attributs

Utilisation pour le chaînage

Héritage, sous-classes

Héritage

Une classe peut être définie comme **héritant d'une autre classe**, elle

- se souvient de la classe dont elle dérive
- fait appel à ses méthodes et attributs **sauf si elle les redéfinit**
- et ainsi de suite récursivement en cas d'héritages successifs

Conséquences :

- les instances de la sous-classe sont construites comme pour toute classe via `SousClasse(...)`
- si celle-ci ne redéfinit pas `__init__`, alors c'est le constructeur de la superclasse qui est utilisé

Héritage (suite)

La sous-classe peut **spécialiser** la superclasse en

- redéfinissant ses méthodes et ses attributs
- définissant de nouvelles méthodes et attributs
- on peut retrouver les méthodes de la classe parente à l'aide de `ClasseParente.methode(self, args)` ou plutôt `super().methode(args)`

Attention, même une méthode d'une classe parente peut finir par appeler une méthode redéfinie

Mise en œuvre

```
class Polygone():
    def __init__(self):
        self.nb_cotes = 0
        self.cotes = []

    def perimetre(self):
        p = 0
        for i in range(self.nb_cotes):
            p += self.cotes[i]
        return p

class Rectangle(Polygone):
    def __init__(self, l, h):
        self.nb_cotes = 4
        self.cotes = [l,h,l,h]
```

La classe parente doit être définie dans le même fichier.

Héritage multiple (spécifique à Python)

```
class ClasseFille(Parent1, Parent2, Parent3):
    ...
```

Les attributs non définis sont alors recherchés :

- dans la classe parente la plus à gauche (`Parent1`) et ses ascendants
- puis dans la suivante de gauche à droite et ses ascendants
- etc.

(parcours en **profondeur** de l'arborescence des classes parentes)

Conclusion

En Python, les objets sont partout ! Même là où on ne les voit pas...

- Ils permettent de manipuler agréablement des références, des enregistrements...
- Le type d'un objet est aussi la classe avec laquelle il a été construit
- Même certains types primitifs définissent des méthodes (`append`...)

Conclusion

En Python, les objets sont partout ! Même là où on ne les voit pas...

- Ils permettent de manipuler agréablement des références, des enregistrements...
- Le type d'un objet est aussi la classe avec laquelle il a été construit
- Même certains types primitifs définissent des méthodes (`append`...)

Mais les concepts utilisés ne sont pas (tous) spécifiques à Python :

- Une classe décrit un schéma commun à plusieurs objets :
  - leur structure interne (attributs)
  - ainsi que leur interface avec le reste du monde (méthodes)
- Chaque instance d'une classe est un objet indépendant, qui peut modifier ses attributs mais bénéficie de méthodes communes
- Le mécanisme d'héritage permet :
  - de factoriser proprement beaucoup de code
  - tout en gardant une certaine encapsulation
  - de hiérarchiser les concepts implémentés par les sous-classes

`https://docs.python.org/3/tutorial/classes.html`