Langages de programmation

Cours 8

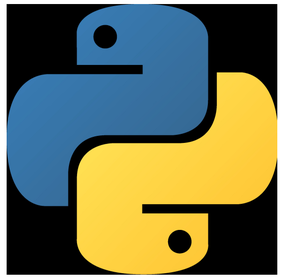
Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/lp-prog

Plan

- programmation objet en C++
- programmation objet en Python
- programmation objet en Java
- principes de la programmation objets
- erzazt de programmation fonctionnelle



A Lire: tutorial sur le langage C++ `http://www.programiz.com/c-programming`

Classes et Objets

[rappel]

- le style de programmation procédurale de C n'est pas le style de C++

- en C++, on utilise des classes et des objets

```
class Personne {
   public:
    string nom;
    int age;

    Personne (string n, int a) { nom = n; age = a; };
};
```

```
int main () {
    Personne JJ = Personne ("Jean-Jacques", 19);
    Personne Wei = Personne ("Wei", 28);

    printPersonne (&JJ);
    printPersonne (&Wei);
    return 0;
}
```

JJ et Wei sont des **objets** de la classe Personne

```
void printPersonne (Personne *p) {
    cout << p->nom << ", " << p->age << " ans\n";
}
```

Classes et Objets

[rappel]

- une classe contient des données (*data*) et des fonctions publiques ou privées (par défaut).

- ces fonctions sont souvent appelés les méthodes de l'objet

```
class Personne {
   public:
    string nom;
    int age;

    Personne (string n, int a) { nom = n; age = a; };

    string str () {
      return nom + string(", ") + to_string(age) + string(" ans");
    }
};
```

données (publiques)

constructeur

méthode pour transformer en `string`

```
int main () {
    Personne JJ = Personne ("Jean-Jacques", 19);
    Personne Wei = Personne ("Wei", 28);
    cout << JJ.str() << endl;
    cout << Wei.str() << endl;
    return 0;
}
```

Classes et Objets

[rappel]

- une sous-classe spécifie les données et méthodes de la classe dont elle dérive

```
class Enseignant : Personne {
   public:
    int nbEleves;
    string matiere;

    Enseignant (string n, int a, string s, int ne) :
        Personne(n, a) { matiere = s; nbEleves = ne; }

    string str () {
      return Personne::str()
          + string(", enseignant de ") + matiere + string(", ")
          + to_string(nbEleves) + string(" élèves");
    }
};
```

nouvelles données

constructeur

redéfinition de la méthode `str`

```
int main () {
    Enseignant Wei = Enseignant ("Wei", 28, "chinois", 3);
    cout << Wei.str() << '\n';
    return 0;
}
```

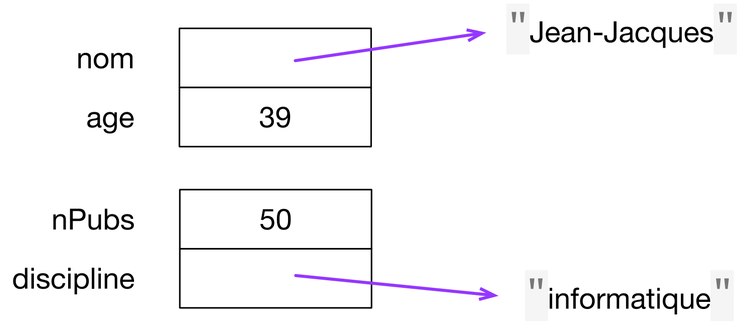
Classes et sous-classes en Python

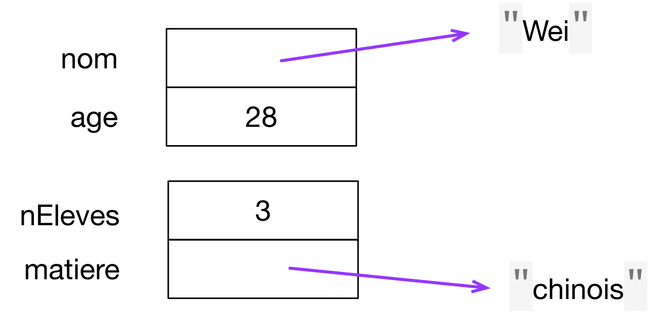
- le programme C++ du cours 7 peut se réécrire en Python très simplement

```
class Personne:
    def __init__ (self, n, a) :
        self.nom = n
        self.age = a

    def __str__ (self) :
        return "{}, {} ans".format (self.nom, self.age)
```

```
class Chercheur (Personne):
    def __init__ (self, n, a, s, np) :
        Personne.__init__(self, n, a)
        self.discipline = s
        self.nPubs = np

    def __str__ (self) :
        return Personne.__str__(self) + \
           ", chercheur en {}, {} publications".format \
           (self.discipline, self.nPubs)
```

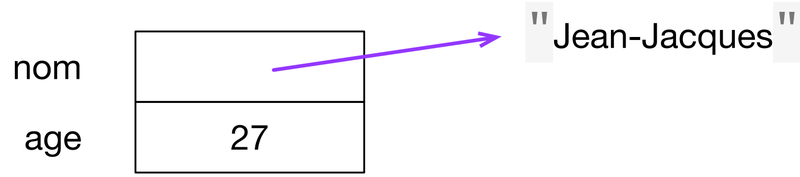
```
JJ = Chercheur ("Jean-Jacques", 39, "informatique", 50)
Wei = Enseignant ("Wei", 28, "chinois", 3)
print (JJ)
print (Wei)
```





Classes et sous-classes en Java

- le programme C++ du cours 7 peut se réécrire aussi en Java

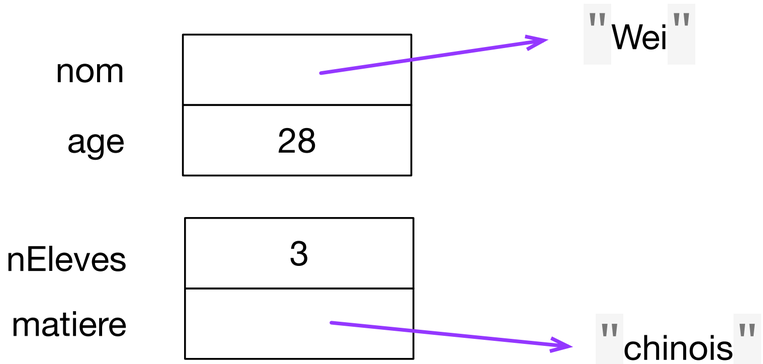



```
class Personne {
  String nom;
  int age;

  Personne (String n, int a) {
    nom = n;
    age = a;
  }

  public String toString () {
    return nom + ", " + age + " ans";
  }
}
```

```
class Test {
   public static void main(String[] args) {
   Personne JJ = new Personne ("Jean-Jacques", 27);
   System.out.println (JJ);
  }
}
```

Classes et sous-classes en Java

- le programme C++ du cours 7 peut se réécrire aussi en Java




```
class Enseignant extends Personne {
  String matiere;
  int nEleves;

  Enseignant (String n, int a, String s, int ne) {
    super (n, a);
    matiere = s;
    nEleves = ne;
  }

  public String toString () {
    return super.toString() + ", matiere " + matiere + ", " + nEleves + " élèves";
  }
}
```

```
class Test {
   public static void main(String[] args) {
   Enseignant Wei = new Enseignant ("Wei", 28, "chinois", 3);
   System.out.println (Wei);
  }
}
```

Programmation objet

- la programmation objet favorise la programmation incrémentale à partir des données

- beaucoup disent que c'est une programmation modulaire. C'est exact à partir des données, mais certainement pas à partir des instructions du programme

- si on veut changer ou rajouter une fonction, il faudra le faire à **plusieurs** endroits du programme

	ajout de données	ajout de fonctions
programmation objet	+	-
programmation procédurale	-	+

programmation objet == programmation à partir des données

Programmation modulaire

- certains langages ont des modules en séparant leur signature du code d'implémentation
- on retrouve cela un peu dans les fichiers `#include` de C ou C++
- en Java, on peut utiliser la commande `interface`
- en Java ou C++, il y a aussi la commande `virtual` pour déclarer la signature des fonctions

programmation objet == programmation par les données

Types inductifs

- type inductif est un mot savant pour désigner un type défini par récurrence
- on définit des arbres ou des listes chaînées en C++, Python, Java dans 2 styles: procédural ou objet
- 1 style procédural

```
typedef struct arbre {
    int val;
    struct arbre *filsG, *filsD;
} Arbre;

Arbre *Noeud (int x, Arbre *gauche, Arbre *droit) {
  Arbre *r = malloc (sizeof (Arbre));
  r->val = x;
  r->filsG = gauche;
  r->filsD = droit;
  return r;
}

Arbre *Feuille (int x) {
  return Noeud (x, NULL, NULL);
}
```

Types inductifs

- 2 style programmation par objets

```
class Arbre {
  int val;
}

class Noeud extends Arbre {
  Arbre filsG;
  Arbre filsD;

  Noeud (int x, Arbre a, Arbre b) {
    val = x; filsG = a; filsD = b;
  }
}

class Feuille extends Arbre {
  Feuille (int x) { val = x; }
}

class Test {
   public static void main(String[] args) {
     Noeud a = new Noeud (3, new Feuille (2), new Feuille(5));
  }
}
```

Types inductifs

- 2 style programmation par objets (avec des types plus simples)

```
class Arbre {
  int val;
  Arbre filsG;
  Arbre filsD;

  Arbre (int x, Arbre g, Arbre d) { // noeud
    val = x; filsG = g; filsD = d;
  }
}

class Test {
   public static void main(String[] args) {
    Arbre a11 = new Arbre (11, null, null);
    Arbre a22 = new Arbre (22, null, new Arbre (33, null, null));
    Arbre a = new Arbre (3, a11, a22);
     System.out.println (a.val);
     System.out.println (a.filsG.val);
  }
}
```

Exercices

Exercice Dans l'exemple précédent, écrire la méthode `toString` et faire `System.out.println`

Exercice Ecrire l'exemple précédent en Python

Exercice Ecrire l'exemple en C++

Un peu de programmation fonctionnelle

- pas d'état mémoire, pas de variables modifiables, on n'utilise que des fonctions et des constantes !

- l'ordre de l'exécution n'a pas d'importance

- le shell de Unix est un début de programmation fonctionnelle

```
cat verlaine | sort

cat verlaine | rev | head

cat verlaine | tr a-z A-Z | sed -e 's/$/!!!/'
```

`verlaine`

```
Les sanglots longs
des violons
de l'automne
blessent mon coeur
d'une langueur
monotone.
```

- Haskell ou Ocaml sont des langages de programmation fonctionnelle

Haskell Curry

Robin Milner

Un peu de programmation fonctionnelle

- pas d'état mémoire, pas de variables modifiables, on n'utilise que des fonctions et des constantes !

```
import Data.List

process t = unlines (sort (lines t))
main = readFile "verlaine" >>= putStr . process
```

- et en utilisant la composition des fonctions

```
import Data.List

process = unlines . sort . lines
main = readFile "verlaine" >>= putStr . process
```

- ou en faisant une abstraction sur la fonction `sort`

```
import Data.List

process f = unlines . f . lines
main = readFile "verlaine" >>= putStr . (process sort)
```

`verlaine`

```
Les sanglots longs
des violons
de l'automne
blessent mon coeur
d'une langueur
monotone.
```

Haskell amuse-bouche de Mark Lentczner (Google)

Un peu de programmation fonctionnelle

• en utilisant un nom de fonction plus parlant

```
import Data.List

byLines f = unlines . f . lines
main = readFile "verlaine" >>= putStr . (byLines sort)
```

• ou peut-être plus expressif

```
sortLines = byLines sort
main = readFile "verlaine" >>= putStr . sortLines
```

• et encore

```
reverseLines = byLines reverse
main = readFile "verlaine" >>= putStr . reverseLines
```

`verlaine`

```
Les sanglots longs
des violons
de l'automne
blessent mon coeur
d'une langueur
monotone.
```

Un peu de programmation fonctionnelle

- essayons d'indenter les lignes du poème avec :

```
indent s = "    " ++ s
indentLines = byLines indent
```

attention aux types !!

- Haskell est un langage fortement typé (les types sont **synthétisés**)

```
Prelude> :t lines
lines :: String -> [String]
Prelude> :t unlines
unlines :: [String] -> String
Prelude> :t byLines
byLines :: ([String] -> [String]) -> String -> String

Prelude> :t indent
indent :: [Char] -> [Char]
```

Haskell est typé statiquement

- les types ne correspondent pas

Un peu de programmation fonctionnelle

- la fonction map

```
map reverse ["j'aime", "beaucoup", "l'informatique"]
["emia'j","puocuaeb","euqitamrofni'l"]

map sort ["j'aime", "beaucoup", « l'informatique"]
["'aeijm","abceopuu","'aefiilmnoqrtu"]
```

- la fonction `map` applique une fonction sur tous les éléments d'une liste

```
map f [x1, x2, … , xn]  == [f x1, f x2, … , f xn ]
```

- et donc pour indenter le poème:

```
onEachLine f = unlines . (map f). lines
indent s = "    " ++ s

main = readFile "verlaine" >>= putStr . (onEachLine indent)
```

Un peu de programmation fonctionnelle

- définition des fonctions avec filtrage (comme en mathématiques)

```
fact 0 = 1
fact n = n * fact (n-1)
```

définition par filtrage

- et imprimer le résultat

```
main = print (fact 50)
```

programme principal

admirer le style succinct

Un peu de programmation fonctionnelle

- définition des types inductifs (arbres, listes, …)

```
data Arbre a = Nil | Noeud a (Arbre a) (Arbre a)
```

arbre de type polymorphe

- et définir des fonctions sur ces structures

```
hauteur Nil = 0
hauteur (Noeud _ a b) = 1 + max (hauteur a)(hauteur b)
```

définition par filtrage

```
main = do
    let a = Noeud 3 (Noeud 5 Nil Nil) Nil
    print (hauteur a)
```

programme principal

admirer le style succinct

Un peu de programmation fonctionnelle

- ML est un langage un peu moins fonctionnel qui a introduit les types polymorphes synthétisés

- Ocaml est la version développée à Inria (en France)

- très bon environnement de programmation

```
let rec fact n = match n with
  | 0 -> 1
  | n -> n * fact (n-1) ;;

print_int (fact 10) ;;
```

Conclusion

vive la programmation !