Séq. 18 – Paradigmes de programmation

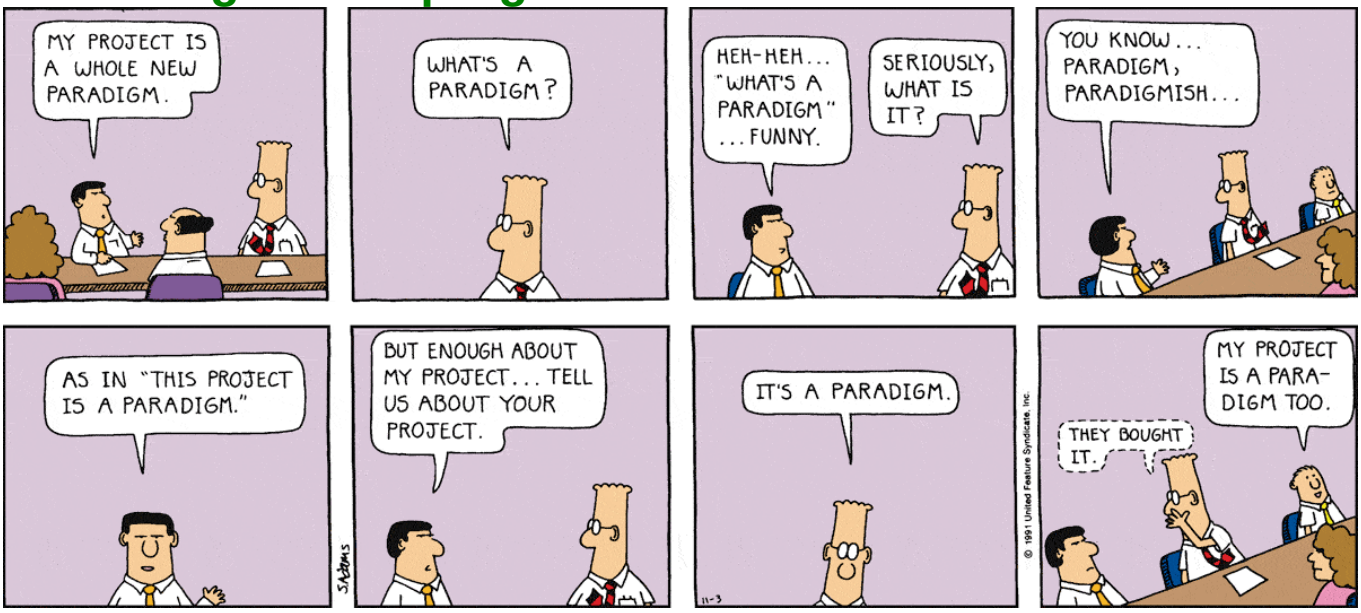
Objectifs

1. Distinguer sur des exemples les paradigmes impératif, fonctionnel et objet.
2. Choisir le paradigme de programmation selon le champ d'application d'un programme.

Cette séquence s'appuie sur :

- https://www.lecluse.fr/nsi/NSI_T/langages/paradigmes/

1 Paradigmes de programmation



Le nombre de langages de programmation est gigantesque : on en dénombre plus de 2000. Pourquoi autant de langages ? Comment les choisir ?
Il y a bien sûr des langages plus populaires que d'autres - reste bien sûr à définir la notion de *populaire*. Si on prend comme critère les recherches des développeurs sur les forums d'entraide, on obtient ce classement qui fait de Python le langage le plus populaire.

http://pypl.github.io/PYPL.html

Python rentre dans la catégorie des langages généralistes : il peut s'adapter à beaucoup de situations et de *paradigmes* différents.

Dans certaines situations, on va choisir un langage particulier parce qu'il est plus adapté qu'un langage généraliste pour effectuer certaines tâches. Pour prendre un exemple un peu extrême : pour gérer efficacement l'accès à un grand nombre de données, on va privilégier un langage *orienté requêtes* comme **SQL** plutôt que python.
Voici quelques exemples de paradigmes : Impératif, Orienté Objets, Fonctionnel, Logique, événementiel, Concurrent, Orienté requêtes etc... Cette liste n'est pas exhaustive.

1.1 Programmation Impérative

Le paradigme Impératif est celui avec lequel vous avez probablement découvert la programmation : on y trouve toutes les structures comme les boucles (*for*, *while*), les conditionnelles (*if*, ...), les variables, les tableaux...
Python est bien sûr un langage impératif.
Le langage **BASIC** (Beginner's All-purpose Symbolic Instruction Code)) est un langage développé en 1963 et qui, comme son nom l'indique, était plutôt dédié aux débutants. Dans ce langage, les fonctions n'existent pas (il faut utiliser à la place des **GOSUB... RETURN**, et les lignes d'un programme doivent être numérotés. par ailleurs, l'absence de la notion de bloc d'instruction oblige à utiliser les **GOTO** à haute dose ce qui rend toute programmation rigoureuse et structurée extrêmement difficile.
Voici un exemple qui calcule les termes de la suite de Fibonacci (https://fr.wikipedia.org/wiki/Suite_de_Fibonacci) :

```
10 CLS
20 PRINT "PJ DEMO"
30 PRINT
40 LET I = 10
50 GOSUB 80
60 PRINT "RESULTAT : "  + R
70 END
80 IF I > 1 THEN GOTO 110
90 LET R = 1
95 PRINT "R=" + R
100 RETURN
110 LET A = 1
120 LET B = 1
```

```
130 FOR J = 2 TO I+1
140 LET T = A
150 LET A = B
160 LET B = A + T
170 PRINT T + "-" + A + "=>" + B
180 NEXT J
190 LET R = B
200 RETURN
```

A faire vous même 1.

Testez le programme ci-dessus sur le simulateur de langage Basic :

http://www.quitebasic.com/

- Étudiez l' aide : www.quitebasic.com/help/
- Créez un 1er programme qui imprime `DEBUT DE PROGRAMME`, puis `HELLO WORLD !` puis `FIN DE PROGRAMME`
- Créez un 2e programme de résolution d' équation du 2nd degré
  - Qui initialise trois valeurs `a`, `b`, et `c` (pour ax^2+bx+c)
  - Qui calcule le discriminant
  - En fonction de la valeur du discriminant, qui affiche 0, 1 ou les 2 solutions

1.2 Paradigme orienté Objets

Nous avons abordé la programmation orientée objet. Les langages qui permettent de manipuler les classes, propriétés et méthodes rentrent dans cette catégorie des langages objets. Le langage **C** est un langage impératif, mais pas orienté objet. Le langage **C++** est une évolution du langage **C** pour prendre en charge le paradigme objet.
Les idées sous-tendant le paradigme objet datent des années 60. Mais il faudra attendre le début des années 70 et la mise au point du langage Smalltalk pour que le paradigme objet gagne en popularité chez les informaticiens. Ajourd' hui de nombreux langages permettent d'utiliser le paradigme objet : C++, Java,...

1.3 Programmation événementielle

Vous avez étudié l'an passé le langage Javascript, spécifiquement conçu pour rendre les pages webs plus interactives. Il est donc optimisé pour répondre aux événements qui peuvent survenir sur une page web. Javascript obéit - entre autres - au paradigme de programmation événementielle.
Le langage **Scratch** est au autre exemple de langage événementiel dans lequel on associe des blocs de code à des actions de l'utilisateur.
Le développement d'interfaces graphiques (par exemple TKinter en Python) fait largement appel à la programmation événementielle.

1.4 Programmation concurrentielle ou parallèle

Un aperçu de la programmation parallèle en Python est donné ici : illustration de l'interblocage. La grosse problématique avec la programmation parallèle est le partage des ressources entre plusieurs processus. Certains langages sont mieux armés pour traiter efficacement cette problématique. Nous ne nous étendrons pas davantage sur ce sujet délicat.

1.5 Programmation Logique

Certains langages sont spécialement conçus pour traiter des problèmes logiques, voire faire des preuves mathématiques. On peut citer **Prolog** ou **Coq** pour les preuves.
Développons le langage **Prolog** : ce langage a été créé en 1972 par un Français : *Alain Colmerauer*. Il est surtout utilisé dans le domaine de l'intelligence artificielle, mais aussi dans le traitement du langage. Dans Prolog, un programme est en fait une base de faits et règles, et le programme est en quelque sorte exécuté en posant une question dans l'interpréteur.
Sans faire un tutoriel, regardons juste un exemple qui montre le fonctionnement totalement différent de ce langage très spécial :

1.5.1 Le problème à résoudre

Les données du problème sont :

- Max a un chat.
- Eric n'est pas en pavillon.
- Luc habite un studio mais le cheval n'y est pas.
- Chacun habite une maison différente et possède un animal distinct.

La problématique à résoudre est : Qui habite le château et qui a le poisson ?

1.5.2 Le programme en Prolog

```
Source : http://www.gecif.net/articles/linux/prolog.html
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Réalisé par Jean-Christophe MICHEL
    % Juillet 2011
```

```
% www.gecif.net
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%-----------------------------------------------------------
% les faits :

% les 3 maisons :
maison(chateau).
maison(studio).
maison(pavillon).

% les 3 animaux :
animal(chat).
animal(poisson).
animal(cheval).

%-----------------------------------------------------------
% les règles :

% le prédicat relation constitue la relation entre une personne, son
animal et sa maison :
relation(max,M,chat):-maison(M).
relation(luc,studio,A):-animal(A),A\==cheval.
relation(eric,M,A):-maison(M),M\==pavillon,animal(A).

% le prédicat different est vraie seulement si ses 3 paramètres sont
différents :
different(X,X,_):-!,fail.
different(X,_,X):-!,fail.
different(_,X,X):-!,fail.
different(_,_,_).

% le prédicat "resoudre" indique les 4 inconnues à retrouver :
resoudre(MM,ME,AE,AL):-

        relation(max,MM,chat),
        relation(eric,ME,AE),
        different(MM,ME,studio),
        relation(luc,studio,AL),
        different(AE,AL,chat).
```

1.5.3 La solution

Pour résoudre le problème, on tape `resoudre(MM,ME,AE,AL).` ce qui donne :

```
MM = pavillon,
ME = chateau,
AE = cheval,
AL = poisson .
```

qui s'interprète ainsi :

- la maison de Max (MM) est un pavillon
- la maison d'Eric (ME) est un chateau
- l'animal d'Eric (AE) est le cheval
- l'animal de Luc (AL) est le poisson

Sans chercher à rentrer dans les détails, un survol de cet exemple vous montre comment à partir de faits et de règles, le langage trouve une solution à un problème. Il utilise pour cela un **moteur d'inférence** qui simule des raisonnements déductifs.

A faire vous même 2.

- P. 77 ex 5

☐

1.6 Programmation fonctionnelle

☐

1.6.1 Le principe du λ-calcul

Les langages fonctionnels sont des langages qui reposent sur le λ-calcul, créé par Church en 1925. Le principe du λ-calcul consiste à considérer les fonctions comme des données comme les autres. Ce concept ne vous est pas étranger.

Exemple : en λ-calcul, la fonction mathématique $x \mapsto x+1$ s'écrirait $\lambda x \cdot (x+1)$

Python peut faire du λ-calcul comme le montre cet exemple :

```
>>> ma_fonction = lambda x:x+1
>>> ma_fonction(4)
```

Cet appel renvoie 5.

`ma_fonction` est une variable, déclarée comme telle, mais c'est aussi une fonction. Vous remarquerez au passage à quel point une fois de plus la syntaxe python est proche de la syntaxe mathématique !

1.6.2 Les avantages de la programmation fonctionnelle

Les langages fonctionnels présentent en général les particularités suivantes :

1. Les données sont **immuables** : leur valeur n'est jamais modifiée ; les fonctions peuvent créer de nouvelles données, mais pas en modifier. On obtient ainsi un code beaucoup plus sûr ;
2. Les **effets de bord** sont clairement identifiés et en général séparés du cœur du langage : On sait ce qui rentre dans une fonction, ce qui en sort et ce que fait la fonction. En effet, ce sont eux qui peuvent entraîner un comportement erratique ;
3. Il est possible de mettre en place un mécanisme d'**évaluation paresseuse** : les nouvelles données ne sont calculées que lorsque leur valeur devient nécessaire. Cela évite de calculer des données qui, au final, ne serviraient pas ;
4. Un mécanisme, appelé **inférence de type** permet à l'interpréteur du langage de déduire le type des données et fonction sans qu'il soit nécessaire de le spécifier.

1.6.3 quelques langages fonctionnels populaires

Un des premiers langages fonctionnel est **Lisp** (1958). Sa syntaxe particulière à base de parenthèses est très reconnaissable :)

```
(de inter (L M)
    (cond
        ((null L) nil)
        ((member (car L) M) (cons (car L) (inter (cdr L) M)))
        (t (inter (cdr L) M)) ))
```

Les poids lourds de la catégorie sont **CAML** (1985) et le plus récent **Haskell**.
La encore ce sont des langages à la syntaxe très particulière comme le montre cet exemple en **Haskell** qui calcule la somme des termes d'une liste :

```
somme liste = if (length liste == 0) then 0 else head liste + somme
(tail liste)
```

Si vous trouvez que Python a une syntaxe difficile, essayez les langages fonctionnels :)

A faire vous même 3.

- P. 77 ex 5

1.7 Conclusion

Des langages généralistes comme python permettent au développeur d'utiliser beaucoup de paradigmes. Cela peut paraître être la panacée : Si Python fait tout avec une syntaxe simple, pourquoi créer des langages spécialisés comme Haskell à la syntaxe obscure ? Avec Python On peut faire de la programmation fonctionnelle mais on peut aussi comme on vient de le voir faire des entorses à ce paradigme fonctionnel ! Laisser au développeur la possibilité de créer des variables globales et de faire des effets de bords peut le mener à des bugs parfois difficiles à déceler.
Un langage purement fonctionnel comme Haskell ne tolérera pas ces entorses et oblige donc à une rigueur absolue. Le développeur y gagne à terme car son code est plus fiable, moins susceptible d'avoir des bugs donc plus fiable.
Un langage généraliste permet de tout faire, y compris des choses pas propres ! En fonction des projets le développeur fera des choix de paradigmes pertinents ce qui pourra le conduire naturellement vers tel ou tel langage.