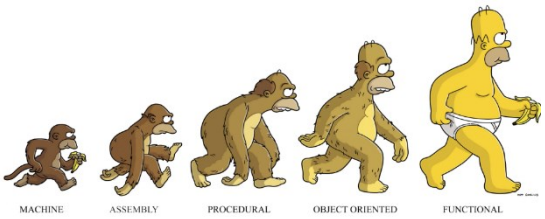
TNSI	Langages et programmation	Cours – **Paradigmes de programmation**
	Paradigmes de programmation	Programmation fonctionnelle

Objectifs :

- ⇨ Revoir ce qu'est un paradigme de programmation
- ⇨ Découvrir un nouveau paradigme de programmation
- ⇨ Savoir distinguer les principaux paradigmes de programmation



I – Paradigmes de programmation

On appelle **paradigme de programmation** une *façon* d'écrire un programme.

Historiquement la première approche et la seule supportée directement par les microprocesseurs est celle du paradigme impératif. C'est aussi la plus utilisée.

Dans le paradigme **impératif**, le programme décrit les opérations en *séquences d'instructions* et en *itérations* exécutées par l'ordinateur pour modifier l'*état* du programme et de la mémoire.

On a depuis développé d'autres approches qui sont plus adaptées à certains problèmes. Nous ne parlerons que des paradigmes **fonctionnel** et **objet** qui sont les seuls au programme.

Dans le **paradigme objet**, le programme est constitué d'un ensemble d'*objets* qui interagissent. Ces objets sont des briques logicielles ayant un *état* (les attributs) et des *fonctions* (méthodes).

Le **paradigme fonctionnel** ramène le programme à l'*évaluation d'expressions* constituées de *fonctions* éventuellement imbriquées.

Il en existe de nombreux autres qui peuvent éventuellement se combiner et qui n'ont pas toujours de frontières bien définies. Citons par exemple la **programmation logique** (ensembles de règles et de faits logiques que l'on définit pour aboutir au résultat), la **programmation évènementielle** (comme pour les GUI ou le javascript), la **programmation parallèle** (plusieurs threads participant à la résolution), …

Il n'y a pas de paradigme *meilleur* qu'un autre, mais chacun possède des avantages et des inconvénients et suivant la tâche que l'on se fixe il sera plus facile d'utiliser tel ou tel paradigme. Il est donc utile de connaître plusieurs paradigmes afin d'être capable de choisir le plus adapté à la situation qu'on rencontre.

De même si certains langages sont clairement orientés vers un paradigme donné (comme prolog pour la programmation logique, Lisp ou Caml pour la programmation fonctionnelle, c++ pour la programmation objet), de nombreux langages sont multi-paradigmes. C'est le cas de python qui permet notamment la programmation objet et la programmation fonctionnelle.

II – Programmation fonctionnelle

1) Historique

La théorie sous-jacente à la programmation fonctionnelle est celle du λ-calcul (prononcer *lambda-calcul*), introduite par Church en 1925. Le premier langage fonctionnel fut le Lisp (1958), qui eut un certain succès, notamment dans le domaine de l'IA. D'autres langages fonctionnels sont à citer :

- ML (1973), plus proche de la théorie du λ-calcul ;
- CAML (1987) puis OCaml (1996), déclinaison française de ML, autorisant la programmation impérative et objet ;
- Haskell (1990), une version plus moderne, très pure, de la notion de langage fonctionnel.

2) Principe

L'idée fondamentale du lambda-calcul est de considérer que les fonctions ..... .
Ainsi, elles peuvent être par exemple passées en paramètre à d'autres fonctions.
Il en découle que :

- les fonctions sont des fonctions au sens mathématique du terme : elles se contentent de renvoyer une valeur en fonction de leurs arguments ;
- il n'y a pas de notion « d'état », ni à l'extérieur des fonctions, ni dans les fonctions. Un programme n'est donc qu'une composition de fonctions.

a. Transparence référentielle

La **transparence référentielle** est la propriété d'un langage (ou d'un programme) qui fait que toute variable, ainsi que tout appel à une fonction, peut être remplacée par sa valeur sans changer le comportement du programme.

On considère la fonction `etend` ci-contre qui ajoute à `L1` les deux derniers éléments de `L2` et renvoie la liste correspondante.

```
1   def etend(L1,L2) :
2       L1.append(L2[-2])
3       L1.append(L2[-1])
4       return L1
5   L = [1,2,3]
```

Application n°1 :
Que vaut `L3` après l'instruction `L3 = etend(L, L)` ? Après `L3 = etend(L, [1,2,3])` ? Cette fonction respecte-t-elle la transparence référentielle ?

En programmation fonctionnelle, la transparence référentielle doit toujours être respectée.

b. Fonctions pures

En **programmation** une fonction est dite pure si .....

. .....

Ainsi en particulier ..... .

Rappel :

Un **effet de bord** est la modification de l'état de la machine (variable, mémoire, entrées/sorties, …) par une fonction autre que celle de sa valeur de retour.

Par exemple la procédure `print()` a l'effet de bord de modifier l'affichage. Une fonction de tri en place d'un tableau a pour effet de bord de modifier le tableau (en le triant).

Application n°2 :
Les fonctions suivantes sont-elles pures ?

Fonction A	Fonction B	Fonction C	Fonction D	Fonction E

`def f() :`
`  return 2*x`
	`def etend(L1,L2) :`
	`  L1.append(L2[-2])`
	`  L1.append(L2[-1])`
	`  return L1`
		`def etend(L1,L2) :`
		`  L = L1.copy()`
		`  L.append(L2[-2])`
		`  L.append(L2[-1])`
		`  return L`
			`def g(x):`
			`  return 3*x*x + 2`
				`def add(n):`
				`  global acc`
				`  acc += n`
				`  return acc`

En programmation fonctionnelle, toute fonction doit donc être une fonction pure.
Pour cela on s'arrange pour qu'une évaluation ne modifie jamais les données, mais plutôt que chaque expression crée une nouvelle valeur.

Par exemple on utilisera `l + [a]` plutôt que `l.append(a)` car la première expression créée une nouvelle liste constituée des valeurs de la liste `l` puis de la valeur `a` sans modifier `l`, tandis que la deuxième ajoute la valeur `a` à la liste `l` (et donc la modifie).

c. Fonction d'ordre supérieur

Les fonctions étant des données comme les autres, on peut alors définir les fonctions d'ordre supérieur :

Une **fonction d'ordre supérieur** est une fonction qui prend en argument une ou plusieurs fonctions et/ou qui renvoie une fonction.

C'est donc en quelque sorte une méta-fonction.

On peut par exemple définir une fonction `incrementeur(n)` :

```
def incrementeur(n):
    def f(x):
        return  n + x
    return f

plusquatre = incrementeur(4)
print(plusquatre(1), plusquatre(5))
```

La fonction `incrementeur(n)` renvoie une fonction qui incrémente le nombre donné en argument de `n`. On peut par la suite utiliser cette fonction en utilisant la notation habituelle (paramètres entre parenthèse).

Application n°3 :
On considère le programme ci-contre.
1) Qu'affiche la ligne 4 ?

2) Qu'affiche la ligne 10 ?

```
1   def f1(x):    return x*x
2   def f2(x):    return 2*x
3   def f3(fn,x): return f2(fn(x))
4   print(f3(f1,3))
5
6   def somme(f, g):
7       def s(x): return f(x) + g(x)
8       return s
9   f4 = somme(f1,f2)
10  print(f4(4))
```

3) Sur le modèle de la fonction `somme`, écrire une fonction `compose(f,g)` qui prend en argument deux fonctions $f(x)$ et $g(x)$ et qui renvoie la fonction composée : $f \circ g$. Vérifier que $f1 \circ f2(3)$ vaut bien 36.

d. Les fonctions lambda

Pour faciliter l'écriture de fonctions intermédiaires (comme la fonction s de l'application n°2), python permet l'écriture simplifiée de *fonctions anonymes*, appelées **fonctions lambda**.

La syntaxe est juste :
`lambda` `[`*`param1`*`[,`*`param2`*`,…]] :` *`expression renvoyée par la fonction`*

Exemple :
```
>>> f = lambda x,y : x + 2*y
>>> f(2,3)
8
```

Application n°4 :
1) Ré-écrire le programme de l'application précédente en utilisant des fonctions lambda.
2) Que feront les lignes suivantes écrites dans la console ?
```
>>> n = lambda : print('nsi')
>>> x = n()
```

e. Boucles et variables

En programmation fonctionnelle stricte, il n'est pas possible de modifier l'état de la mémoire et donc d'une variable. Ainsi l'utilisation de variables n'est autorisée que comme des noms et les variables ne doivent pas être modifiées après leur création.
Il est donc impossible de réaliser des boucles `for` ou des boucles `while` qui utilisent toutes deux des variables ou dépendent d'effets de bords.
Pour réaliser des boucles, en programmation fonctionnelle stricte, il faudra donc utiliser la récursivité.

f. Structures conditionnelles dans une expression

On a vu que la programmation fonctionnelle était basée sur une suite d'évaluations d'expression. Python offre la possibilité d'insérer des structures conditionnelles *dans une expression*.

La syntaxe est : `expr =` *`valeur_si_vrai`* **`if`** *`condition`* **`else`** *`valeur_si_faux`*

Ainsi les codes de gauche et de droite sont équivalents :

```
if char != 'a':
    a = 3
else:
    a = 6
```

```
a = 3 if char != 'a' else 6
```

3) Fonctions d'ordre supérieur de python

Le langage python offre aux programmeurs plusieurs fonctions d'ordre supérieur opérant sur des itérateurs et permettant ainsi de faire des boucles sans avoir recours à la récursivité. Elles permettent d'écrire des programmes dans le paradigme fonctionnel en python en gardant un code simple et lisible.

a. La fonction `map`

La fonction `map()` de Python applique une fonction sur tous les éléments d'une séquence itérable et renvoie un objet map.

La fonction `map()` prend deux arguments positionnels : la fonction à exécuter sur l'itérable et l'itérable lui-même (par exemple une liste).

```
map(fonction_a_appliquer, iterable_sur_lequel_appliquer_la_fonction)
```

Le résultat sera un objet map avec un emplacement en mémoire.

Supposons que l'on veuille multiplier tous les nombres d'une liste par 2 et stocker le résultat dans une nouvelle liste.

De manière classique, on pourrait faire le programme ci-contre :

On remarque au passage que ce programme utilise des variables que l'on modifie (n et doubles) et ne respecte donc pas strictement la programmation fonctionnelle.

```
1   nombres = [2, 3, 4, 5]
2   doubles = []
3   for n in nombres:
4       doubles.append(2 * n)
5   print (doubles)
```

La fonction `map()` nous permet d'avoir le même résultat d'une manière beaucoup plus simple et élégante :

```
nombres = [2, 3, 4, 5]
def f(x): return 2 * x
doubles = list(map(f, nombres))
```

ou encore mieux :

```
nombres = [2, 3, 4, 5]
doubles = list(map(lambda n : 2 * n, nombres))
```

On remarque l'utilisation de la fonction `list()` pour transformer l'objet map en liste python.

b. La fonction `filter`

La fonction `filter()` permet de filtrer un itérable en ne gardant que les éléments pour lesquels une fonction d'évaluation renvoie `True`.

Elle prend en argument la fonction d'évaluation et l'itérable à filtrer et renvoie un objet filter :

```
filter(fonction_d_evaluation, iterable_a_filtrer)
```

```
ages = [15, 23, 7, 55, 13, 48,4]     # Liste des ages
majeurs = list(filter(lambda x : x >= 18, ages))   # Liste des majeurs parmi la liste d'age
```

c. La fonction `zip`

La fonction `zip()` combine les éléments de deux itérables selon les index correspondants en une liste de tuples.

Elle prend en argument les deux itérables et renvoie un objet zip itérable :

```
zip(premier_iterable, deuxieme_iterable)
```

La fonction s'arrête dès qu'un des deux itérables est épuisé.

Exemple :

```
>>> list(zip(['A','B','C'], range(0,10)))
[('A', 0), ('B', 1), ('C', 2)]
```

On remarque que la liste résultante ne contient que 3 tuples (longueur du plus petit des deux itérables).

Application n°5 :

1) En utilisant les fonctions `map` et `zip` écrire une fonction `mini(a, b, c)` qui respecte le paradigme fonctionnel et renvoie une liste contenant le maximum de trois listes a, b et c (voir bout de code ci-contre).

```
a = [7, 9, 3, 7, 2]
b = [3, 4, 1, 8, 3]
c = [3, 8, 0, 8, 7]
assert mini(a,b,c) == [3, 4, 0, 7, 2]
```

2) On dispose d'un dictionnaire du poids de plusieurs animaux et une liste des félins. Ecrire un programme en paradigme fonctionnel qui affiche le nom et le poids du félin le plus lourds du dictionnaire `animaux`.

```
animaux = {'vache': 500, 'baleine': 30000, 'chat': 6,
           'herrison': 0.5, 'tigre': 300, 'ours': 500}
felins = ['chat', 'tigre', 'lion', 'léopard']
```

Aide : On pourra utiliser les fonctions built-ins `max` et `filter` et des fonctions lambda. Ne pas hésiter à calculer et afficher tous les résultats intermédiaires.

4) Avantages et inconvénients

Intérêts :

- une programmation globalement plus sûre, du fait qu'il n'y a pas de données mutables partagées (de ce fait certains programmes ou parties de programme critiques sont écrit avec ce paradigme dans le but de minimiser le risque de bug) ;
- il est bien plus facile de prouver la terminaison ou la correction d'un algorithme dans ce paradigme ;
- de part une réduction forte des structures de contrôle (boucles principalement, mais aussi conditionnelles), un programme plus clair et plus concis ;
- des programmes plus facile à paralléliser (du fait de la transparence référentielle et de l'absence d'effet de bord, dans `z = f1(x) + f2(x)`, `f1` et `f2` peuvent être exécutées simultanément ou dans n'importe quel ordre en donnant toujours le même résultat) ;

Il y a aussi bien sûr quelques inconvénients :

- code moins intuitif pour des non matheux ;
- masquage de la complexité à cause de l'absence de boucles ;
- implantation souvent (mais pas toujours…) plus gourmande en mémoire (problème de la pile d'appel pour la récursivité notamment).

L'essentiel :

.....

.....

.....

.....

.....

.....

.....

.....

Références :

Paradigmes impératif vs déclaratifs : https://www.ionos.fr/digitalguide/sites-internet/developpement-web/paradigmes-de-programmation/
Fonctions zip, filter et map : https://pythonforge.com/les-fonctions-map-filter-et-zip-en-python/

Exercices – Paradigmes de programmation

Exercice 1 : Dérivée

Mathématiquement, la dérivée d'une fonction f en un point d'abscisse x peut être définie comme :

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

On peut approximer la dérivée en prenant une valeur de h suffisamment petite et en calculant $\frac{f(x+h) - f(x)}{h}$.

Ecrire une fonction d'ordre supérieur `derive(f)` qui prend en argument une fonction `f(x)` à un seul argument et qui renvoie une fonction `f_prime(x)` à un argument également qui calcule la dérivée de `f` au point `x`.

Vérifier la fonction en vérifiant que la dérivée de `math.sin` donne bien les mêmes résultats que `math.cos`.

Exercice 2 : Maximum

Proposer un programme Python donnant le maximum d'une liste :

1) dans le paradigme impératif ;
2) dans le paradigme fonctionnel ;
3) dans le paradigme fonctionnel récursif.

Aide : Pour le 2), vous pouvez utiliser la fonction `reduce` (doc en français ou en anglais) du module `functools` pour le faire en fonctionnel strict ou bien la fonction `sorted` (mais évitez `max` !).

Exercice 3 : Table de hash

Réécrire en s'aidant de la fonction `map` le code ci-contre, qui prend en entrée une liste de mots de passe et crée une liste des mots de passes hachés avec la fonction built-in `hash`.

```
passwords = ['abc', '1234', '#eO?5kLp:']
hashes = [None]*len(passwords)
for i in range(len(passwords)):
  hashes[i] = hash(passwords[i])
print(hashes)
```

Exercice 4 : Utilisation de map

Pour chacune des lignes du tableau suivant, écrivez en utilisant `map` et des fonctions lambda une instruction qui convertit l'itérable de la colonne de gauche en sortie de la colonne de droite :

Entrées	Sorties
`['12','-2','0']`	`[12,-2,0]`
`['hello','world']`	`[5,5]`
`['hello','world']`	`['olleh','dlrow']`
`[1,2,3,4,5]`	`[5,4,3,2,1]`

Exercice 5 : Filtrage de texte

1) Écrire une fonction qui retourne vrai si et seulement si il existe au moins un caractère interdit dans un texte. On utilisera une liste de caractères interdits. Aide : on peut s'en passer, mais la fonction `any` peut s'avérer utile ici.
2) Écrire une fonction prenant une chaîne en paramètre et retourne la liste des mots qui ne contiennent pas de 'e'. On utilisera `filter` et une fonction utilitaire qui indique si un mot contient un e. Les mots seront considérés séparés par des espaces, et donc on utilisera la méthode `split` sur le texte.

Exercice 6 : Tri rapide

Le tri rapide consiste à choisir la première valeur de la liste à trier comme valeur pivot puis à séparer le reste de la liste en deux listes : la première liste contient les valeurs inférieures au pivot et la seconde contient les valeurs supérieures au pivot. On trie récursivement les deux listes puis on concatène les listes triées en insérant le pivot entre les deux.

Ecrire la fonction `tri_rapide(tableau)` en utilisant le paradigme fonctionnel.