DIU Enseigner l'informatique au lycée

Notes de cours

Sylvain Conchon

Sylvain.Conchon@lri.fr

Programmation fonctionnelle et paradigmes de programmation

Programmation fonctionnelle

Notions introduites

- Fonctions passées en arguments
- Fonctions renvoyées comme résultats
- Structures de données immuables
- Programmation avec itérateurs

Une question que tout le monde se pose (ou presque)

Qu'est-ce que la programmation fonctionnelle ?

Une question que tout le monde se pose (ou presque)

Qu'est-ce que la programmation fonctionnelle ?

On peut tenter d'y répondre par une autre question.

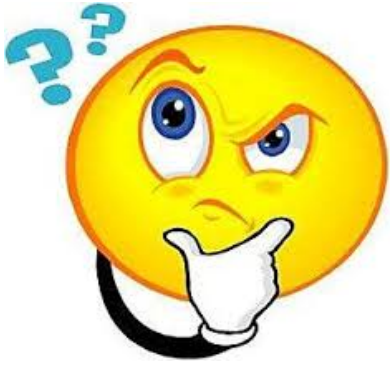
Une question que tout le monde se pose (ou presque)

Qu'est-ce que la programmation fonctionnelle ?

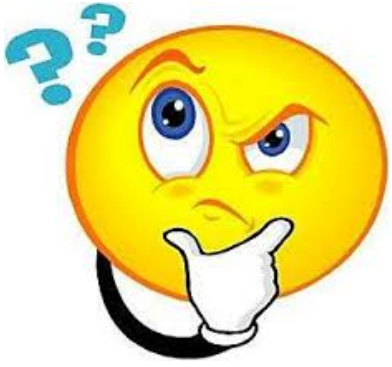
On peut tenter d'y répondre par une autre question.

Que se passerait-il dans le monde (de la programmation) si les variables n'étaient plus modifiables ?

Un monde sans variables modifiables

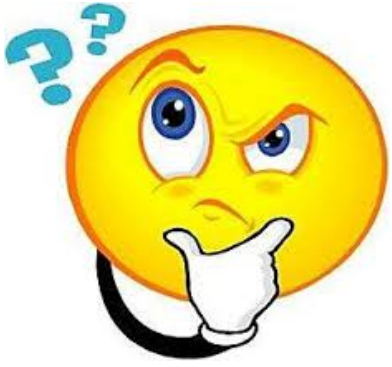


Un monde sans variables modifiables



Les boucles `while` deviennent nécessairement inutiles car on y entre jamais ou en sort jamais !

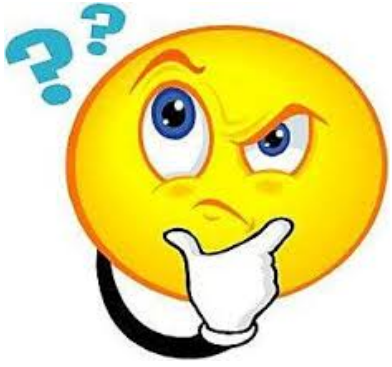
Un monde sans variables modifiables



Les boucles `while` deviennent nécessairement inutiles car on y entre jamais ou en sort jamais !

Mais comment peut-on programmer sans boucles `while` ?

Un monde sans variables modifiables



Les boucles `while` deviennent nécessairement inutiles car on y entre jamais ou en sort jamais !

Mais comment peut-on programmer sans boucles `while` ?

Remarque : les boucles `for` deviennent aussi *presque* inutiles.

On peut tout de même y entrer et en sortir, mais on ne peut pas faire grand chose dans le corps de la boucle.

Réaliser des calculs sans boucles

Pour calculer, on va devoir utiliser uniquement des fonctions.

Réaliser des calculs sans boucles

Pour calculer, on va devoir utiliser uniquement des fonctions.

Exemple :

$$\sum_{i=1}^{10} i$$

Réaliser des calculs sans boucles

Pour calculer, on va devoir utiliser uniquement des fonctions.

Exemple :

$$\sum_{i=1}^{10} i$$

```
x = 0
for i in range(1,11):
  x = x + i
```

```
def somme(i):
  if i == 10:
    return i
  else:
    return i + somme(i+1)
```

Des fonctions récursives locales

Pour réaliser des boucles, on utilise des fonctions récursives locales

Des fonctions récursives locales

Pour réaliser des boucles, on utilise des fonctions récursives locales

```
def somme_tab(t):
  v = t[0]
  for i in range(1, len(t)):
    v = v + t[i]
  return v
```

Des fonctions récursives locales

Pour réaliser des boucles, on utilise des fonctions récursives locales

```
def somme_tab(t):
  v = t[0]
  for i in range(1, len(t)):
    v = v + t[i]
  return v

def somme_tab(t):
  def boucle(i,v):
    if i == len(t): return v
    else:
      return boucle(i+1, v+t[i])
  return boucle(1,t[0])
```

La programmation fonctionnelle

C'est donc un styte de programmation où :

- Aucune variable n'est modifiable
- On *ne modifie pas* les structures de données (cela ne nous empêche pas d'utiliser des structures modifiables)
- On utilise que des fonctions pour réaliser des calculs

Mais c'est plus que ça :

- Fonctions d'ordre supérieur
- Fonctions anonymes
- Fermetures
- Application partielle
- ...

FONCTIONS D’ORDRE SUPÉRIEUR

Les fonctions sont des valeurs à part entière

Les fonctions sont des valeurs comme les autres

Une fonction peut être :

- stockée dans une structure de donnée (n-uplets, listes etc.)
- passée en argument à une autre fonction
- retournée comme résultat d'une fonction

Les fonctions prenant des fonctions en arguments ou rendant des fonctions en résultat sont dites **d'ordre supérieur**

Structures de données contenant des fonctions

On peut stocker des fonctions dans n'importe quelle structure de données.

Par exemple, dans un n-uplet :

```
def f(x) : return x+1
def g(x) : return x+x

>>> p = (f,g)
>>> p[0](p[1](4))
17
```

Fonctions anonymes (1/2)

Les fonctions étant des objets comme les autres, on peut les créer sans avoir à leur donner un nom.

Pour cela, on utilise la notation `lambda` pour créer des fonctions anonymes

Fonctions anonymes (1/2)

Les fonctions étant des objets comme les autres, on peut les créer sans avoir à leur donner un nom.

Pour cela, on utilise la notation `lambda` pour créer des fonctions anonymes

```
lambda <arguments> : <expression>
```

Fonctions anonymes (1/2)

Les fonctions étant des objets comme les autres, on peut les créer sans avoir à leur donner un nom.

Pour cela, on utilise la notation `lambda` pour créer des fonctions anonymes

```
lambda <arguments> : <expression>
```

Par exemple, la fonction $x \mapsto x + 1$ est définie de cette manière :

```
lambda x: x+1
```

Remarquez que l'expression à droite du `:` ne contient pas de return

Fonctions anonymes (2/2)

On pourra donc écrire de manière équivalente :

```
def f(x) : return x + 1
```

et

```
f = lambda x: x+1
```

Fonctions anonymes (2/2)

On pourra donc écrire de manière équivalente :

```
def f(x) : return x + 1
```

et

```
f = lambda x: x+1
```

Cette notation se généralise aux fonctions à plusieurs arguments. Ainsi, la fonction $x, y \mapsto x + y$ s'écrira

```
lambda x,y: x+y
```

Fonctions anonymes (2/2)

On pourra donc écrire de manière équivalente :

```
def f(x) : return x + 1
```

et

```
f = lambda x: x+1
```

Cette notation se généralise aux fonctions à plusieurs arguments. Ainsi, la fonction $x, y \mapsto x + y$ s'écrira

```
lambda x,y: x+y
```

Et on pourra donc écrire de manière équivalente :

```
def g(x,y) : return x+y
```

ou

```
g = lambda x,y: x+y
```

Fonctions comme arguments

- Certaines fonctions prennent naturellement des fonctions en arguments
- Par exemple, les notations mathématiques telles que la sommation $\Sigma_{i=1}^{n} f(i)$ se traduisent facilement si l'on peut utiliser des arguments fonctionnels

```
def somme(f,n):
  if n <= 0: return 0
else:
    return f(n) + somme(f, n-1)

>>> somme(lambda x: x*x*x, 5)
>>> 225
```

Exemple

On souhaite généraliser la fonction pour calculer la somme des éléments d'un tableau `t` en une fonction qui applique un opérateur quelconque op aux éléments du tableau :

$$\texttt{t[0]}\ op\ \texttt{t[1]}\ op\ \cdots\ op\ \texttt{t}[\texttt{len(t)} - 1]$$

Exemple

On souhaite généraliser la fonction pour calculer la somme des éléments d'un tableau `t` en une fonction qui applique un opérateur quelconque op aux éléments du tableau :

$$\texttt{t[0]}\ op\ \texttt{t[1]}\ op\ \cdots\ op\ \texttt{t}[\texttt{len(t)} - 1]$$

```
def calcul(op,t):
  def boucle(i,v):
    if i == len(t): return v
    else:
      return boucle(i+1, op(t[i],v))
  return boucle(1,t[0])

>>> calcul(lambda x,y:x*y, [1,2,3,4])
>>> 24
```

Fonctions comme résultat

Les fonctions peuvent également être renvoyées comme résultat.

Par exemple, pour approcher la dérivée $f'(x)$ d'une fonction f avec un taux d'accroissement

$$\frac{f(x+h) - f(x)}{h}$$

pour un h suffisamment petit, on pourra écrire la fonction suivante en Python :

```
def derive(f):
  h = 1e-7
  return lambda x: (f(x + h) - f(x)) / h

>>> d = derive(math.sin)
>>> d(0)
0.9999999999999983
```

Fonctions à plusieurs arguments (1/3)

La notion de fonction à plusieurs arguments est *plus subtile* qu'il n'y parait.

Fonctions à plusieurs arguments (1/3)

La notion de fonction à plusieurs arguments est *plus subtile* qu'il n'y parait.

Y-a-t'il une différence entre les deux fonctions `f` et `g` définies de la manière suivante ?

```
f = lambda x,y : x+y
```

et

```
g = lambda p : p[0]+p[1]
```

et

```
h = lambda x: lambda y : x+y
```

Fonctions à plusieurs arguments (2/3)

```
>>> f(4,5)
9
```

Fonctions à plusieurs arguments (2/3)

```
>>> f(4,5)
9

>>> g(4,5)
Traceback (most recent call last):
File "...", line 1, in <module>
g(4,5)
TypeError: <lambda>() takes 1 positional argument but 2
were given
```

Fonctions à plusieurs arguments (2/3)

```
>>> f(4,5)
9
```

```
>>> g(4,5)
Traceback (most recent call last):
File "...", line 1, in <module>
g(4,5)
TypeError: <lambda>() takes 1 positional argument but 2
were given
```

La fonction g ne prend donc qu'un unique paramètre, à savoir la paire p.

Pour l'appeler correctement, il faut donc lui donner un seul argument :

```
>>> g( (4,5) )
9
```

Fonctions à plusieurs arguments (3/3)

```
>>> h(4,5)
Traceback (most recent call last):
File "...", line 1, in <module>
h(4,5)
TypeError: <lambda>() takes 1 positional argument but 2
were given
```

Là encore, la fonction `h` ne prend donc qu'un unique paramètre, mais comment faire pour réaliser le calcul `x + y` ?

APPLICATION PARTIELLE

Des fonctions de fonctions

L'expression suivante

```
lambda x: lambda y : x + y
```

est une fonction qui prend une valeur `x` en argument et qui renvoie comme résultat une fonction qui prend une valeur `y` en argument, cette dernière renvoyant `x+y` comme résultat.

```
>>> plus = lambda x: lambda : x + y
>>> plus(4)(5)
9
```

Des fonctions de fonctions

L'expression suivante

```
lambda x: lambda y : x + y
```

est une fonction qui prend une valeur `x` en argument et qui renvoie comme résultat une fonction qui prend une valeur `y` en argument, cette dernière renvoyant `x+y` comme résultat.

```
>>> plus = lambda x: lambda : x + y
>>> plus(4)(5)
9
```

De manière équivalente, on peut définir `plus` de la manière suivante :

```
def plus(x): return lambda y : x + y
```

Application partielle (1/2)

Puisque la fonction `plus` n'attend qu'un seul argument, que vaut `plus(4)` ?

Application partielle (1/2)

Puisque la fonction `plus` n'attend qu'un seul argument, que vaut `plus(4)` ?

```
>>> h = plus(4)
>>> h(5)
9
```

Application partielle (1/2)

Puisque la fonction `plus` n'attend qu'un seul argument, que vaut `plus(4)` ?

```
>>> h = plus(4)
>>> h(5)
9
>>> h(10)
14
```

Nous venons simplement d'appliquer partiellement la fonction `plus`.

Application partielle (2/2)

L'application partielle nous permet par exemple de "spécialiser" des fonctions.

Par exemple,

```
def derive(h):
  return lambda f: lambda x: (f(x + h) - f(x)) / h

>>> der1 = derive(1e-10)
>>> der2 = derive(1e-7)
>>> f = der1(math.sin)
>>> g = der2(math.sin)
```

Application partielle (2/2)

L'application partielle nous permet par exemple de "spécialiser" des fonctions.

Par exemple,

```
def derive(h):
  return lambda f: lambda x: (f(x + h) - f(x)) / h

>>> der1 = derive(1e-10)
>>> der2 = derive(1e-7)
>>> f = der1(math.sin)
>>> g = der2(math.sin)

>>> f(0)
1.0
>>> g(0)
0.9999999999999983
```

Un modèle d'exécution intriguant

```
def plus(x) : return lambda y: x+y
```

```
>>> h = plus(5)
>>> h(4)
9
```

Un modèle d'exécution intriguant

```
def plus(x) : return lambda y: x+y
```

```
>>> h = plus(5)
>>> h(4)
9
```

Où est stockée la variable `x` dans la fonction `h` ?

Un modèle d'exécution intriguant

```
def plus(x) : return lambda y: x+y
```

```
>>> h = plus(5)
>>> h(4)
9
```

Où est stockée la variable `x` dans la fonction `h` ?

Elle devrait avoir disparu de la mémoire d'après le modèle d'exécution avec une pile vu précédemment !

La notion de fermeture

Une fermeture est une fonction avec un état interne. Cet état est constitué de la valeur des variables libres du corps de la fonction.

Python construit une fermeture en capturant la valeur des variables dont la durée de vie échappe à la portée statique des variables.

```
lambda y: x+y
```

Une fermeture est une sorte de paire, avec une composante qui représente l'adresse mémoire du corps de la fonction, et une autre qui pointe sur l'environnement des variables libres.

RÉCURSION SANS RÉCURSION

Définitions récursives

En général, Python n'accepte pas les définitions récursives

```
>>> v = (1,v)
Traceback (most recent call last):
File "...", line 1, in <module>
v = (1,v)
NameError: name 'v' is not defined
```

(Bien que cette définition ne semble pas poser de problème, puisqu'on sait représenter une telle valeur en mémoire)

Les définitions récursives ne semblent être possibles que pour les fonctions.

Qu'est-ce qu'une fonction récursive ?

Une fonction semble être un objet mathématique qui prend un argument et renvoie un résultat.

Par exemple, la fonction $x \mapsto x + 1$ est représentée par la valeur `lambda x: x+1`

Qu'est-ce qu'une fonction récursive ?

Une fonction semble être un objet mathématique qui prend un argument et renvoie un résultat.

Par exemple, la fonction $x \mapsto x + 1$ est représentée par la valeur `lambda x: x+1`

Question : Mais qu'est-ce qu'une fonction récursive ?

Qu'est-ce qu'une fonction récursive ?

Une fonction semble être un objet mathématique qui prend un argument et renvoie un résultat.

Par exemple, la fonction $x \mapsto x + 1$ est représentée par la valeur `lambda x: x+1`

Question : Mais qu'est-ce qu'une fonction récursive ?

Réponse : C'est une fonction qui fait "appel à elle-même" !

Qu'est-ce qu'une fonction récursive ?

Une fonction semble être un objet mathématique qui prend un argument et renvoie un résultat.

Par exemple, la fonction $x \mapsto x + 1$ est représentée par la valeur `lambda x: x+1`

Question : Mais qu'est-ce qu'une fonction récursive ?

Réponse : C'est une fonction qui fait "appel à elle-même" !

Question : Mais comment faire "appel à soi-même" ?

Qu'est-ce qu'une fonction récursive ?

Une fonction semble être un objet mathématique qui prend un argument et renvoie un résultat.

Par exemple, la fonction $x \mapsto x + 1$ est représentée par la valeur `lambda x: x+1`

Question : Mais qu'est-ce qu'une fonction récursive ?

Réponse : C'est une fonction qui fait "appel à elle-même" !

Question : Mais comment faire "appel à soi-même" ?

Réponse : Il faut que la fonction ait un nom !

Qu'est-ce qu'une fonction récursive ?

Une fonction semble être un objet mathématique qui prend un argument et renvoie un résultat.

Par exemple, la fonction $x \mapsto x + 1$ est représentée par la valeur `lambda x: x+1`

Question : Mais qu'est-ce qu'une fonction récursive ?

Réponse : C'est une fonction qui fait "appel à elle-même" !

Question : Mais comment faire "appel à soi-même" ?

Réponse : Il faut que la fonction ait un nom !

Je ne savais pas que cette histoire de nom était si importante pour définir une fonction...

La suite du calcul

Dans la définition de la fonction récursive `f` suivante :

```
def f(x):
  if x == 0 :
    return 0
  else:
    return x + f(x-1)
```

l'expression `f(x-1)` représente la suite du calcul, c'est-à-dire ce qu'il reste encore à faire pour que l'appel à `f` puisse renvoyer un résultat.

Abstraire la suite du calcul

Amusons-nous à abstraire la suite du calcul. Pour cela, on définit la fonction `ff` suivante, qui prend un argument supplémentaire (une fonction) représentant la suite du calcul à effecter :

```
def ff(suite,x):
  if x == 0 :
    return 0
  else:
    return x + suite(      , x-1)
```

Abstraire la suite du calcul

Amusons-nous à abstraire la suite du calcul. Pour cela, on définit la fonction `ff` suivante, qui prend un argument supplémentaire (une fonction) représentant la suite du calcul à effecter :

```
def ff(suite,x):
  if x == 0 :
    return 0
  else:
    return x + suite(suite, x-1)
```

Cette suite peut aussi avoir besoin d'elle-même pour calculer ! !
C'est là que réside toute la subtilité de la récursion !

Abstraire la suite du calcul

Amusons-nous à abstraire la suite du calcul. Pour cela, on définit la fonction `ff` suivante, qui prend un argument supplémentaire (une fonction) représentant la suite du calcul à effecter :

```
def ff(suite,x):
  if x == 0 :
    return 0
  else:
    return x + suite(suite, x-1)
```

Cette suite peut aussi avoir besoin d'elle-même pour calculer ! !
C'est là que réside toute la subtilité de la récursion !

On notera bien que la fonction `ff` n'est pas récursive.

Récursion dans récursion

Finalement, la fonction `f` peut se définir comme une fonction qui appelle `ff` et qui utilise ff comme suite du calcul (ce qui nous rapproche de la définition de la récursivité).

```
>>> f = lambda x : ff(ff,x):
>>> f(5)
15
```

Récursion dans récursion

Finalement, la fonction `f` peut se définir comme une fonction qui appelle `ff` et qui utilise ff comme suite du calcul (ce qui nous rapproche de la définition de la récursivité).

```
>>> f = lambda x : ff(ff,x):
>>> f(5)
15
```

Mais il n'y a plus besoin de nommer une fonction pour calculer de manière récursive.

```
>>> f = lambda x: \
        (lambda f,y: 0 if y==0 else y + f(f,y-1)) \
        (lambda f,y: 0 if y==0 else y + f(f,y-1),x)
>>> f(5)
15
```

PROGRAMMATION AVEC ITÉRATEURS

Itérateurs

Un itérateur est une fonction qui permet de parcourir une structure de données et de réaliser un calcul.

L'utilisation d'un itérateur est nécessaire quand la structure de données est abstraite.

La programmation avec un itérateur évite de définir des fonctions récursives.

L'itérateur `filter`

L'itérateur `filter` est équivalent à l'opérateur de construction de listes pas compréhension.

```
list(filter(f,l)) == [ v for v in l if f(v)]
```

Par exemple,

```
>>> list(filter(lambda x: x%2==0,[10,2,31,42,54,67]))
[10, 2, 42, 54]
```

L'itérateur map

La sémantique de cet itérateur est définie par l'équation suivante :

```
list(map(f,[e1,e2,...,en])) == [f(e1),f(e2),...,f(en)]
```

Par exemple,

```
>>> list(map(lambda x:x*2, [1,2,3]))
[2, 4, 6]
```

L'itérateur reduce

Le schéma de calcul de cet itérateur est défini par les deux équations suivantes :

```
reduce(f,[ ],v) == v

reduce(f,[e1,e2, ...,en],v) ==
    f(...(f(f(v,e1),e2), ...),en)
```

Attention, ajouter la directive suivante pour utiliser `reduce` :

```
from functools import reduce
```

Une deuxième version s'applique à une liste non vide :

```
reduce(f,[e1,e2, ...,en]) ==
    reduce(f,[e2, ...,en],e1)
```

Exemples

La somme des éléments d'une liste d'entiers

```
def somme(l):
  return reduce (lambda s,x: s+x, l, 0)
```

Le nombre d'éléments dans une liste de listes :

```
def nb_elem_list_list(l):
  return reduce (lambda acc,s: \
           reduce (lambda nb,x: 1+nb, s, acc), l, 0)
```

Exemple : les sous-listes

Calcul de la **liste des sous-listes** d'une liste :

```
def sous_listes(l):
  return reduce (lambda p,x: \
        list(map(lambda k:k+[x],p))+p,l,[[]])
```

Exemple : les sous-listes

Calcul de la **liste des sous-listes** d'une liste :

```
def sous_listes(l):
  return reduce (lambda p,x: \
        list(map(lambda k:k+[x],p))+p,l,[[]])
```

Supposons (hypothèse de récurrence) que `p` soit la liste des sous-listes de `[1,2]` et `x == 3`.

```
p == [[1, 2], [2], [1], []]
```

Exemple : les sous-listes

Calcul de la **liste des sous-listes** d'une liste :

```
def sous_listes(l):
  return reduce (lambda p,x: \
        list(map(lambda k:k+[x],p))+p,l,[[]])
```

Supposons (hypothèse de récurrence) que `p` soit la liste des sous-listes de `[1,2]` et `x == 3`.

```
p == [[1, 2], [2], [1], []]
```

Le résultat de `list(map(lambda k:k+[x],p))` est

```
[[1, 2, 3], [2, 3], [1, 3], [3]]
```

Exemple : les sous-listes

Calcul de la **liste des sous-listes** d'une liste :

```
def sous_listes(l):
  return reduce (lambda p,x: \
        list(map(lambda k:k+[x],p))+p,l,[[]])
```

Supposons (hypothèse de récurrence) que `p` soit la liste des sous-listes de `[1,2]` et `x == 3`.

```
p == [[1, 2], [2], [1], []]
```

Le résultat de `list(map(lambda k:k+[x],p))` est

```
[[1, 2, 3], [2, 3], [1, 3], [3]]
```

Par définition, `p` contient aussi des sous-listes de `[1,2,3]` donc on l'ajoute au résultat :

```
[[1, 2, 3], [2, 3], [1, 3], [3]]+p
```

STRUCTURES DE DONNÉES IMMUABLES

Structures de données non modifiables

La programmation fonctionnelle se caractérise aussi par l'utilisation de structures de données immuables.

Il s'agit de structures de données, a priori quelconque (tableau, ensemble, dictionnaire, etc.) que l'on ne peut plus modifier une fois qu'elles sont construites.

Mais on peut néanmoins les utiliser, pour en consulter le contenu ou pour construire d'autres structures de données.

Interfaces

L'aspect immuable d'une structure se voit, non pas dans la manière dont elle est implantée, mais dans la signature des fonctions de son interface.

L'interface d'une structure immuable contiendra donc des fonctions pour :

- créer des structures (vide, à partir d'autres structures)
- ajouter des éléments
- fusionner des structures
- supprimer des éléments dans une stucture

Toutes ces fonctions doivent toujours renvoyer de nouvelles structures, sans jamais modifier les structures passées en arguments.

La transparence référentielle

La programmation fonctionnelle se caractérise par la propriété de transparence référentielle :

Quand on applique deux fois la même fonction aux mêmes arguments, alors on obtient le même résultat.

Partage

On pense souvent à tort que les structures de données immuables sont moins efficaces que les structures modifiables.

En réalité, on peut profiter de l'immuabilité d'une structure pour réaliser un maximum de partage quand on construit une nouvelle structure (pensez par exemple à l'ajout d'un élément dans un arbre)

Par exemple, comment vérifier efficacement (sans copies inefficaces) l'égalité suivante sur des ensembles, autrement qu'avec des structures immuables ?

$$A \cup B = (A \backslash B) \cup (B \backslash A) \cup (A \cap B)$$

Paradigmes de programmation

Un foisonnement de langages de programmation

L'archive de Bill Kinnersley recense plus de 2500 langages de programmation créés depuis les années 50 !

Un foisonnement de langages de programmation

L'archive de Bill Kinnersley recense plus de 2500 langages de programmation créés depuis les années 50 !

En voici une petite liste :

ABC	Ada	Algol	AWK	APL	B
Basic	C	C++	C#	CAML	CLU
Cobol	CPL	CSS	Dart	Delphi	Eiffel
Flow-Matic	Forth	Fortran	Go	Hack	Haskell
HTML	ICon	J	Java	JavaScript	Julia
Kotlin	Lisp	Mainsail	M	ML	Modula
Oberon	Objective-C	OCaml	Pascal	Perl	PHP
PL/I	Postscript	Prolog	Python	R	Ruby
Rust	Scade	Scala	Scheme	Sh	Simula
Smalltalk	SQL	Swift	Tcl/TK	TypeScript	VBA

Pourquoi tant de langages ?

Finalement, tout programme est exécuté par un ordinateur qui se programme en assembleur

Le langage assembleur est très proche de la machine, ses instructions (`mov`, `jne`, `add`, etc.) sont celles du micro-processeur

Pourquoi ne pas *toujours* programmer en assembleur ?

Pourquoi tant de langages ?

Finalement, tout programme est exécuté par un ordinateur qui se programme en assembleur

Le langage assembleur est très proche de la machine, ses instructions (`mov`, `jne`, `add`, etc.) sont celles du micro-processeur

Pourquoi ne pas *toujours* programmer en assembleur ?

- les programmes deviennent vite *très très* volumineux !
- il est très facile d'écrire des programmes avec des erreurs
- les programmes ne sont pas du tout portable d'une machine à l'autre
- il est très difficile de faire évoluer les programmes
- etc.

À quoi servent tous ces langages ?

Les langages de programmation évoluent avec l'informatique

Ils sont conçus pour addresser des besoins et problématiques spécifiques

Certains styles de programmation sont plus adaptés au développement de certaines applications :

- lisibilité
- expressivité
- efficacité
- portabilité
- fiabilité

Les langages les plus influents

Fortran : IBM, 1952

- Compilation
- Premier langage de programmation haut niveau
- Calcul scientifique

Lisp : MIT, 1958

- Manipulation de listes
- Programmation fonctionnelle
- *Garbage Collector*

Smalltalk : Xerox PARC, 1971

- Programmation orientée objets
- Réflexivité, typage dynamique
- Compilation vers du *bytecode*

Les langages d'aujourd'hui

Java
Javascript
C#
C++
Python
Pascal/Delphi
C
Go
OCaml/Haskell/Scheme/F#
Swift
VB .net
Ruby
Rust

Les paradigmes de programmation

Les langages de programmation sont aussi classés par famille (ou paradigme de programmation)

Les paradigmes de programmation

Les langages de programmation sont aussi classés par famille (ou paradigme de programmation)

Mais, là encore, il y en a beaucoup.

En voici une liste, non exhaustive . . .

impératifs	fonctionnels	orientés objets
déclaratifs	applicatifs	dataflow
logique	avec contraintes	concurrents
évènementiels	dynamiques	de requêtes
de spécifications	d'assemblage	intermédiaires

Les paradigmes de programmation

Les langages de programmation sont aussi classés par famille (ou paradigme de programmation)

Mais, là encore, il y en a beaucoup.

En voici une liste, non exhaustive . . .

impératifs	fonctionnels	orientés objets
déclaratifs	applicatifs	dataflow
logique	avec contraintes	concurrents
évènementiels	dynamiques	de requêtes
de spécifications	d'assemblage	intermédiaires

Les paradigmes de programmation

Les langages de programmation sont aussi classés par famille (ou paradigme de programmation)

Mais, là encore, il y en a beaucoup.

En voici une liste, non exhaustive . . .

impératifs	fonctionnels	orientés objets
déclaratifs	applicatifs	dataflow
logique	avec contraintes	concurrents
évènementiels	dynamiques	de requêtes
de spécifications	d'assemblage	intermédiaires

Un langage de programmation appartient souvent à plusieurs familles

Styles de programmation (1/4)

Le paradigme de programmation influence la manière dont les structures de données se présentent aux utilisateurs.

Paradigme impératif :

```
s = creer_ensemble()
ajouter(s, 34)
ajouter(s, 55)
```

Paradigme objet :

```
s = Ensemble()
s.ajouter(34)
s.ajouter(55)
```

Paradigme fonctionnel :

```
s = creer_ensemble()
s = ajouter(s, 34)
s = ajouter(s, 55)
```

Styles de programmation (2/4)

Impératif : `union(a,b)`

Objet : `a.union(b)`

Fonctionnel : `c = union(a,b)`

Styles de programmation (3/4)

$$A \cup B = (A \backslash B) \cup (B \backslash A) \cup (A \cap B)$$

Impératif

```
u1 = copie(a)
union(u1, b)
u2 = copie(a)
difference(u2, b)
b_a = copie(b)
difference(b_a, a)
union(u2, b_a)
i = copie(a)
intersection(i, b)
union(u2, i)
assert egal(u1, u2)
```

Fonctionnel :

```
assert
 egal(union(a, b),
      union(union(difference(a, b), difference(b, a)),intersection(a, b))
```

Styles de programmation (4/4)

```
def somme(l):
  v = 0
  for e in l:
    v = v + e
  return v
```

```
def somme(l):
  return reduce(lambda v,e: v+e, l, 0)
```

Cette différence de style de programmation peut avoir un impact sur la manière d'effectuer le calcul (parallélisation, calcul distribué)

Quel langage utiliser ?

Tout dépend

- du type d'application à programmer
- de la confiance que l'on souhaite avoir dans le code final
- des performances à atteindre
- de la durée de vie du programme
- etc.

Mélange des genres (1/2)

En Python, certaines structures mélangent des opérations impératives et fonctionnelles.

Dans les tableaux, `append` ou `pop` sont impératives, alors que la concaténation `+` est fonctionnelle.

Dans les ensembles, certaines opérations sont proposées à la fois impérative et fonctionnelle. Par exemple, l'union `a|b` est fonctionnelle tandis que `a.update(b)` est impérative.

Mélange des genres (2/2)

On peut mélanger paradigmes impératif et fonctionnel afin, par exemple, de définir des fonctions avec des états modifiables.

```
def compteur():
  cpt = 0
  def incr():
    nonlocal cpt
    cpt += 1
    return cpt
  return incr
```

```
>>> c1 = compteur()
>>> c2 = compteur()
>>> c1()
1
>>> c1()
2
>>> c2()
1
```

Mélange de langages

Il est parfois nécessaire d'utiliser plusieurs paradigmes et langages de programmation dans un projet.

Projects People Organizations Tools Blog

Chromium (Google Chrome)

Settings | Report Duplicate

Analyzed at

Languages

Total Lines :	34,882,169	Code Lines :	25,670,051
Number of Languages :	36	Total Comment Lines :	4,600,875
		Total Blank Lines :	4,611,243

Language	Code Lines	Comment Lines	Comment Ratio	Blank Lines	Total Lines	Total Percentage	
C++	12,188,073	2,248,742	15.6%	2,720,796	17,157,611		49.2%
HTML	3,212,761	95,026	2.9%	352,074	3,659,861		10.5%
C	2,443,530	718,421	22.7%	314,026	3,475,977		10.0%
JavaScript	2,363,222	802,471	25.3%	429,686	3,595,379		10.3%
XML	1,840,387	21,278	1.1%	122,851	1,984,516		5.7%
Python	932,354	274,918	22.8%	242,300	1,449,572		4.2%
Java	892,700	251,566	22.0%	175,738	1,320,004		3.8%