

# Paradigme fonctionnel

Structurer le programme avec des fonctions et leur structure de contrôle principale : la composition de fonction

La fonction s'applique à un élément d'un ensemble (domaine) de départ et donne un unique résultat dans un autre ensemble (domaine) d'arriver.

$$f : D1 \rightarrow D2, g : D2 \rightarrow D3 \text{ alors } g \circ f : D1 \rightarrow D3 \text{ tel que } g \circ f(x) = g(f(x))$$

Dans un langage de programmation fonctionnel les domaines (ensembles) sont les types

## Association lexicale

Association lexicale: Quand on écrit une définition en utilisant des symbole et des opérateurs; on suppose que ces symboles ont une sémantique et on été défini auparavant

$$\text{successeur} : \mathbb{N} \rightarrow \mathbb{N}$$

$$x \in \mathbb{N} \rightarrow (x + 1) \in \mathbb{N}$$

Par exemple, on utilise le symbole (le caractère) + ce symbole a une sémantique (que fait-il?) qui est une opération d'addition qui doit être déjà définie

Remarque en Python on dirait (duck typing): successeur s'applique à tout type possédant l'opérateur + (`__add__`)

## Abstraction fonctionnelle

Un programme fonctionnel consiste en une expression E (représentant l'algorithme et les entrées). Cette expression E est sujette à des règles de réécriture : la réduction consiste en un remplacement d'une partie de programme fonctionnel par une autre partie de programme selon une règle de réécriture bien définie. Ce processus de réduction sera répété jusqu'à l'obtention d'une expression irréductible (aucune partie ne peut être réécrite). L'expression E\* ainsi obtenue est appelée forme normale (fn) de E et constitue la sortie du programme.

par exemple dans  $f(x) = 2 * x$  quand on fait  $f(2)$  on a  $f(2) \rightarrow 2 * 2 \rightarrow 4$  uniquement par règles de substitution

## Les fondements : Le lambda Calcul

Le  $\lambda$ -calcul a été inventé par le logicien américain Alonzo Church dans les années 1930 Fournir un fondement au mathématiques plus simple que la théorie des ensembles, et fondé sur la notion de fonction => résultat en calculabilité, étant équivalent aux machine de turing et donc aux algorithme impératif

# Lambda Calculus

Les expressions lambda en Python et dans d'autres langages de programmation ont leurs racines dans le calcul lambda, un modèle de calcul inventé par Alonzo Church. Vous découvrirez quand le lambda calcul a été introduit et pourquoi c'est un concept fondamental qui s'est retrouvé dans l'écosystème Python.

## Histoire, impératif fonctionnel les bases

Alonzo Church a officialisé lambda calculus, une langue basée sur l'abstraction pure, dans les années 1930. Les fonctions lambda sont également appelées abstractions lambda, une référence directe au modèle d'abstraction de la création originale d'Alonzo Church.

Le calcul lambda peut coder n'importe quel calcul. Il s'agit de [Turing complete](#), mais contrairement au concept d'une Turing machine, il est pur et ne garde aucun état.

Les langages fonctionnels trouvent leur origine dans la logique mathématique et le calcul lambda, tandis que les langages de programmation impératifs embrassent le modèle de calcul basé sur l'état inventé par Alan Turing. Les deux modèles de calcul, lambda calcul et Turing machines, peuvent être traduits l'un dans l'autre. Cette équivalence est connue sous le nom de "Church-Turing hypothesis".

Les langages fonctionnels héritent directement de la philosophie du calcul lambda, adoptant une approche **déclarative** de la programmation qui met l'accent sur l'abstraction, la **transformation** des données, la **composition** et la **pureté** (aucun état et aucun effet secondaire). Des exemples de langages fonctionnels incluent Haskell, Lisp ou Erlang.

En revanche, la machine de Turing a conduit à une programmation impérative trouvée dans des langages comme Fortran, C, Java, C# ou Python.

Le style impératif consiste à programmer avec des instructions, conduisant le déroulement du programme étape par étape avec des instructions détaillées. Cette approche favorise la mutation et nécessite la gestion de l'état.

La séparation dans les deux familles présente quelques nuances, car certains langages fonctionnels incorporent des caractéristiques impératives, comme OCaml, tandis que les caractéristiques fonctionnelles imprègnent la famille impérative des langues en particulier avec l'introduction de lambda fonctions dans Java, ou Python.

**Python n'est pas intrinsèquement un langage fonctionnel, mais il a adopté très tôt certains concepts fonctionnels.** En particulier, `map`, `filter`, `Reduce` et l'opérateur `lambda`, sont présents.

## la syntaxe : Les trois constructions principales du $\lambda$ -calcul sont :

- **la variable** (il faut bien commencer quelque part) : nous les noterons  $x, y, z$ , etc.
- **l'application** : si  $u$  et  $v$  sont deux programmes, on peut considérer  $u$  comme une fonction et  $v$  comme un argument possible, et former l'application  $u \ v$ . Ceci correspond à la notation mathématique usuelle  $u(v)$  tout en nous économisant quelques parenthèses. On remarquera qu'il n'y a pas de contraintes de typage ici, et qu'il est donc tout à fait légal de former des auto-applications  $x \ x$ , par exemple.
- **l'abstraction** : si  $u$  est un programme dépendant (ou non) de la variable  $x$ , alors on peut former un nouveau programme  $\lambda x: u$ , qui représente la fonction qui à  $x$  associe  $u$ . Par exemple,  $\lambda x: x + 1$  est intuitivement la fonction qui à  $x$  associe  $x + 1$

sauf que  $+$  et  $1$  ne font pas partie du vocabulaire décrit ci-dessus, faudra donc les définir avec du  $\lambda$ -calcul. Un point sur lequel nous reviendrons.

## Calculs et réduction : Les règles essentiel du $\lambda$ -calcul

- **$\alpha$ -renommage: ( $\alpha$ )**  $\lambda x: u \ \lambda y: (u[x \mapsto y])$ ,  $[x \mapsto y]$  exprime remplacer  $x$  par  $y$  ceci exprime par exemple que  $\lambda x: x + 1$  et  $\lambda y: y + 1$  représente la même expression
- **$\beta$ -réduction: ( $\beta$ )**  $(\lambda x: u) v \rightarrow u[x \mapsto v]$  qui exprime que si on applique la fonction qui à  $x$  associe  $u$  à l'argument  $v$ , on obtient la même chose que si on calcule directement  $u$  avec  $x$  remplacé par  $v$ .

Par exemple,  $(\lambda x: x + 1)4 \rightarrow 4 + 1$

## Python possède une définition lambda

`lambda v : expression`

exemple:

In [ ]: `lambda x: x+1`

Out[ ]: <function \_\_main\_\_.<lambda>>

pour appliquer la  $\beta$ -réduction  $(\lambda x: x+1)(2) \rightarrow$  python répondra 3

In [ ]: `(lambda x: x+1)(2)`

Out[ ]: 3

$(\lambda x: x+1)(2) \Rightarrow (2+1) \rightarrow 3$

on peut nommer les lambdas, tout simplement `inc = lambda x: x + 1`

In [ ]: `inc = lambda x: x+1`

In [ ]: `inc(2)`

Out[ ]: 3

```
In [ ]: e1 if c else e2
```

Dans lambda, pas d'état: pas d'affectation, pas d'instructions tout est expression

```
In [ ]: fact = lambda n: n*fact(n-1) if n else 1
```

```
In [ ]: fact(3)
```

```
Out[ ]: 6
```

En lambda calcul tout est fonction, pas de différence entre la fonction et le paramètre dans l'application u v

```
In [ ]: #Autre exemple, un parametre peux être aussi une fonction  
applique10 = lambda f: f(10)  
  
applique10(inc)
```

```
Out[ ]: 11
```

```
In [ ]: (lambda f: f(10))(inc)
```

```
Out[ ]: 11
```

applique10(inc) => inc(10) => 10+1 => 11

# Ce texte est au format code

```
In [ ]: applique10(2)
```

applique10(2) => 2(10)

## La programmation fonctionnelle : quelques caractéristiques

- Les fonctions sont de première classe (objets). Autrement dit, tout ce que vous pouvez faire avec les «données» peut être fait avec les fonctions elles-mêmes (telles que le passage d'une fonction à une autre fonction).
- La récursivité est utilisée comme structure de contrôle principale. Dans certaines langues, aucune autre construction de «boucle» n'existe.
- L'accent est mis sur le traitement des listes (par exemple, c'est la source du nom Lisp). Les listes sont souvent utilisées avec la récursion sur les sous-listes en remplacement des boucles.
- Les langages fonctionnels «purs» évitent les effets de bords. Cela exclut le modèle presque omniprésent dans les langages impératifs d'affectation d'une première, puis d'une autre valeur à la même variable pour suivre l'état du programme.

- En programmation fonctionnelle on transforme les données on ne les modifie pas, nous utiliserons et nous nous forceons à manipuler des données immutables
- La programmation fonctionnelle décourage ou interdit carrément les déclarations, et fonctionne à la place avec l'évaluation des expressions (fonctions plus arguments). Dans le cas pur, un programme est expression.
- La programmation fonctionnelle se préoccupe de ce qui doit être calculé plutôt que de la manière dont il doit être calculé.
- Une grande partie de la programmation fonctionnelle utilise des fonctions «d'ordre supérieur» (en d'autres termes, des fonctions qui agissent sur des fonctions qui fonctionnent sur des fonctions).

```
In [ ]: plus = lambda x: lambda y: x+y
```

```
In [ ]: plus10 = plus(10)
```

plus(10) => lambda y: 10+y

```
In [ ]: plus10(100)
```

```
Out[ ]: 110
```

lambda y: 10+y => 10+100 => 110

L'idée de la programmation fonctionnelle est de se concentrer sur l'écriture de petites fonctions expressives qui effectuent les **transformations** de données requises. Les combinaisons de fonctions peuvent souvent créer du code plus succinct et plus expressif que de longues chaînes d'instructions procédurales ou les méthodes d'objets complexes avec état. Ce partie se concentre sur les fonctionnalités de programmation fonctionnelle de Python.

Cela offre une voie pour la conception de logiciels distincte de l'approche strictement orientée objet utilisée ailleurs. La combinaison d'objets avec des fonctions permet une flexibilité dans l'assemblage d'une collection optimale de composants.

## Fonctions et collections de données

Étant donné que la programmation fonctionne souvent avec des collections de données, nous appliquerons souvent une fonction à tous les éléments d'une collection. Cela se produit lors de l'extraction et de la transformation de données. Cela se produit également lors de la synthèse des données. Cela correspond parfaitement à l'idée mathématique d'un constructeur d'ensemble ou d'une compréhension d'ensemble.

Il existe trois modèles courants pour appliquer une fonction à un ensemble de données:

- **Mappage:** Ceci applique une fonction à tous les éléments d'une collection,. Nous appliquons une fonction,  $m$ , à chaque élément,  $x$ , d'une plus grande collection,  $S$ .  
 $\{m(x)/x \in S\}$
- **Filtrage:** Cela utilise une fonction pour sélectionner des éléments d'une collection,. Nous utilisons un prédictat,  $f$ , pour déterminer s'il faut passer ou rejeter chaque élément,  $x$ , de la

plus grande collection, S.  $\{x/x \in S \text{ si } f(x)\}$

- **Réduire:** cela résume les éléments d'une collection. L'une des réductions les plus courantes consiste à créer une somme de tous les éléments d'une collection, S, écrite comme.  $\sum_{x \in S} x$  D'autres réductions courantes incluent la recherche du plus petit, du plus grand, du produit de tous les éléments, moyenne, écart type, etc...

$$\begin{cases} \text{map} & : \{ \text{map}(x) | x \in S \} \\ \text{filter} & : \{ x | x \in S \text{ si } \text{filter}(x) \} \\ \text{reduce} & : \text{reduce}(S) \end{cases}$$

## Python comme langage fonctionnel

La programmation fonctionnelle étant un style de programmation, elle pourrait être mise en œuvre avec tout langage de programmation.

Par contre pour être suffisamment puissante il est préférable de l'utiliser avec un langage

- permettant l'ordre supérieur
- et le polymorphisme. ou mieux le typage dynamique ou encore le duck typing

Python permet les 2, il possède aussi les fonctions anonymes lambda

```
In [ ]: lambda x,y: x+y
Out[ ]: <function __main__.lambda>
```

## Ordre supérieur et typage dynamique : Fonctionnel en Python

Python permet les 2, Les fonctions sont des entités comme toutes les autres, et le duck typing permet la souplesse au niveau des "Abstraction fonctionnelle"

quand on définit par exemple EV if Cond else EF , on a une expression qui modélise:

si Cond est évalué à vraie, alors le résultat est celle de l'expression EV sinon l'évaluation est celle de l'expression EF .

Dans ce cas peu importe les types de EV et EF. Le langage que l'on utilise devrait le permettre.

Je vous laisse réfléchir à cette définition:

```
In [ ]: LunOuLautrePure = lambda cond: lambda ev: lambda ef: ev if cond else ef
         LunOuLautre = lambda cond, ev, ef: ev if cond else ef
In [ ]: LunOuLautrePure(True)('sasf')(100)
Out[ ]: 'sasf'
```

```
In [ ]: LunOuLautrePure(True)
Out[ ]: <function __main__.<lambda>.<locals>.<lambda>>

In [ ]: LunOuLautre(True)

-----
TypeError Traceback (most recent call last)
<ipython-input-25-950016937bba> in <module>()
----> 1 LunOuLautre(True)

TypeError: <lambda>() missing 2 required positional arguments: 'ev' and 'ef'

In [ ]: LunOuLautre(False, 'saf', 10)
Out[ ]: 10

In [ ]: LunOuLautrePure(False)('expression vaie')(100)
Out[ ]: 100
```

## Écrivons nos premières fonctions en paradigme fonctionnel pur

- factoriel
- fibonacci
- taille d'une liste
- plus petit/plus grand élément d'une liste

### Factoriel : fact(n)

$$\begin{cases} \text{si } n > 0 & : n * fact(n - 1) \\ \text{si } n == 0 & : 1 \end{cases}$$

```
In [ ]: fact = lambda n: n*fact(n-1) if n else 1
In [ ]: fact(5)
Out[ ]: 120

In [ ]: print(fact(10))
%timeit fact(10)

3628800
1000000 loops, best of 3: 1.3 µs per loop
```

### Fibonacci : fib(n)

$$\begin{cases} \text{si } n > 1 & : \text{ } fibo(n - 1) + fibo(n - 2) \\ \text{si } n \leq 1 & : \text{ } 1 \end{cases}$$

```
In [ ]: fibo = lambda n: fibo(n-1)+fibo(n-2) if n>1 else 1
```

```
In [ ]: print(fibo(20))
%timeit fibo(20)
```

```
10946
100 loops, best of 3: 2.38 ms per loop
```

## Le prix de la récursivité non terminale

Comparons à la version impérative

```
In [ ]: def fiboimp(n):
    f_2, f_1=1,1
    for c in range(n-1):
        f_2, f_1 = f_1, f_2+f_1
    return f_1
```

```
In [ ]: print(fiboimp(20))
%timeit fiboimp(20)
```

```
10946
The slowest run took 4.74 times longer than the fastest. This could mean that an intermediate result is being cached.
1000000 loops, best of 3: 1.11 µs per loop
```

Algorithme plus complexe car moins naturel que la définition:

Trouver entre autre : initialisation, la condition d'arrêt de la boucle et l'invariant de boucle

## Récursivité, récursivité terminale, itération

L'équivalence entre une fonction purement récursive et une fonction itérative issue de sa version récursive terminale.

**Une fonction récursive terminale:** Aucun traitement n'est effectué à la remontée d'un appel récursif (sauf le retour d'une valeur).

**Une fonction récursive non terminale:** Le résultat de l'appel récursif est utilisé pour réaliser un traitement (en plus du retour d'une valeur).

Une fonction récursive terminale est en théorie plus efficace (mais souvent moins facile à écrire) que son équivalent non terminale : il n'y a qu'une phase de descente et pas de phase de remontée.

En récursivité terminale, les appels récursifs n'ont pas besoin d'être empilés dans la pile d'exécution, car l'appel suivant remplace simplement l'appel précédent dans le contexte d'exécution.

Il est possible de transformer, de façon simple, une fonction récursive terminale en une fonction itérative : c'est la dérécursivation.

```
In [ ]: # redéfinissons fact et fobo en récursivité terminale  
fact_t = lambda n, acc: fact_t(n-1, acc*n) if n else acc
```

```
In [ ]: print(fact_t(10, 1))  
%timeit fact_t(10,1)
```

3628800  
1000000 loops, best of 3: 1.34 µs per loop

```
In [ ]: fibo_t = lambda n , acc: fibo_t(n-1, [acc[1], acc[0]+acc[1]]) if n>1 else acc[1]
```

```
In [ ]: print(fibo_t(20, [1,1]))  
%timeit fibo_t(20, [1,1])
```

10946  
100000 loops, best of 3: 3.95 µs per loop

## manipulation d'ensemble et listes

Pour nous entraîner à écrire des fonctions récursives en fonctionnel pure, nous allons faire quelques fonctions sur les listes.

Commençons par définir les 3 opérateurs définissant une liste (à la Lisp):

- créer une liste. new\_l
- estVide. vide\_l
- tête,
- reste
- cons

nous avons ces propriétés

- cons(tête(l),reste(l)) = l
- vide\_l(newl()) = True

## Définissons ces fonctions en Python et fonctionnel pure

```
In [ ]: new_l = lambda : []
```

```
In [ ]: vide_l = lambda l: not l
```

```
In [ ]: vide_l(new_l())
```

True

```

Out[ ]:

In [ ]: tete = lambda l: l[0]

In [ ]: tete([1,2,3])

Out[ ]: 1

In [ ]: reste = lambda l: l[1:]

In [ ]: reste([1,2,3])

Out[ ]: [2, 3]

In [ ]: cons = lambda t, r: [t, *r]

In [ ]: cons(1,[2,3])

Out[ ]: [1, 2, 3]

In [ ]: cons(1, cons(2, new_l()))

Out[ ]: [1, 2]

```

**Suivre ce [lien](#) pour les exemples de récursivité avec les listes, et définition d'unschémas général de récursivité**

<https://colab.research.google.com/drive/1R3F3gOYoihrsOSgQnZWHZSgQXOO-d0tD>

## Transformer une récursivité en recursivité terminale

### La récursivité

Un algorithme est récursif lorsqu'il intervient dans sa propre description. Il est défini en fonction de lui-même. En général un algorithme récursif est lié à une relation de récurrence.

Par exemple on peut définir  $x^n$  par la récurrence suivante:

$$\begin{cases} x^0 = 1 \\ x^n = x * x^{n-1} \quad \text{si } n \geq 1 \end{cases}$$

```

In [ ]: # ce qui donne naturellement en Python
xpn = lambda x: lambda n: 1 if n==0 else x*xpn(x)(n-1)

```

```

In [ ]:

```

```
xpn(10)(3)
Out[ ]: 1000

In [ ]:
# ce qui donne naturellement en Python
xpn = lambda x, n: 1 if n==0 else x*xpn(x,n-1)

In [ ]:
xpn(10,3)
Out[ ]: 1000
```

## Récursivité double

soit les combinaison  $C_n^p$  on a

$$\begin{cases} C_n^0 = 1 & \text{si } p = 0 \\ C_i^i = 1 & \text{si } n = p \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{si } n > 0, p > 0, n \neq p \end{cases}$$

```
In [ ]:
c_n_p = lambda n,p: 1 if p==0 else 1 if n==p else c_n_p(n-1,p)+c_n_p(n-1,p-1)

In [ ]:
c_n_p(10,0)
Out[ ]: 1

In [ ]:
c_n_p(10,10)
Out[ ]: 1

In [ ]:
c_n_p(5,1)
Out[ ]: 5

In [ ]:
c_n_p(5,2)
Out[ ]: 10
```

## Point d'arrêt

Toute les définitions récuses possèdent un point d'arrêt. Il faut un cas d'arrêt où l'on ne fait pas d'appel récursif.

<reponseDirecte> if <test\_arret> else <appelRecuratif>

## récusivité et récusivité terminale

## Schemas général d'une récursivité

$$\begin{cases} f(\dots, 0, \dots) &= \text{quelque chose sans recursivité} \\ f(\dots, n, \dots) &= g(n, f(\dots, n-1, \dots), \dots) \text{ forme générale} \end{cases}$$

## Schemas général d'une récursivité terminale

$$\begin{cases} f(0) &= \text{quelque chose sans recursivité} \\ f(\dots, n, \dots) &= f(\dots, (n-1), \dots) \text{ forme générale} \end{cases}$$

## La pile d'execution

La Pile d'exécution (call stack) du programme en cours est un emplacement mémoire destiné à mémoriser les paramètres, les variables locales ainsi que l'adresse de retour de chaque fonction en cours d'exécution.

Elle fonctionne selon le principe LIFO (Last-In-First-Out) : dernier entré premier sorti.

Attention ! La pile à une taille fixée, une mauvaise utilisation de la récursivité peut entraîner un débordement de pile (stack overflow).

```
In [ ]: fact = lambda n: 1 if n ==0 else fact(n-1)*n
In [ ]: fact(1000)
-----
RecursionError                                     Traceback (most recent call last)
<ipython-input-14-a3d790031eab> in <module>()
----> 1 fact(1000)

<ipython-input-13-3d66625b130e> in <lambda>(n)
----> 1 fact = lambda n: 1 if n ==0 else fact(n-1)*n

... last 1 frames repeated, from the frame below ...

<ipython-input-13-3d66625b130e> in <lambda>(n)
----> 1 fact = lambda n: 1 if n ==0 else fact(n-1)*n

RecursionError: maximum recursion depth exceeded in comparison
```

## Transformation

- Utilisation d'un accumulateur `acc` : correspond à la valeur jusqu'à l'étape de récursivité précédente `n-1`.
- On démarre avec `acc` initialisé à la valeur du point d'arrêt

exemple cas de factorielle:

$$fact(n) = \begin{cases} \text{si } n > 0 & : n * fact(n-1) \\ \text{si } n == 0 & : 1 \end{cases}$$

devient: acc représente  $\text{fact}(n-1)$

$$\text{fact}_t(\text{acc}, n) = \begin{cases} \text{si } n > 0 & : \text{ fact}_t(\text{acc} * n, n - 1) \\ \text{si } n == 0 & : \text{ acc} \end{cases}$$

puis

$$\text{fact}(n) = \text{fact}_t(1, n)$$

```
In [ ]: # en definition Lambda
fact_t = lambda acc: lambda n: acc if n==0 else fact_t(acc*n)(n-1)
fact = fact_t(1)
```

```
In [ ]: print(fact(3), fact(5), fact(10))
```

6 120 3628800

```
In [ ]: # en definition normale
```

```
In [ ]: # fact = Lambda n: 1 if n==0 else n*fatcvt(n-1)
def fact(n):
    if (n==0):
        return 1
    else:
        return n*fact(n-1)
```

```
In [ ]: def fact_t(acc, n):
    if n==0:
        return acc
    else:
        return fact_t(acc*n,n-1)

def fact(n):
    return fact_t(1,n)
```

```
In [ ]: print(fact(3), fact(5), fact(10))
```

6 120 3628800

Avons nous résolu le problème de la pile? En recursivité terminale plus besoin d'empiler, le dernier `return` est le résultat

```
In [ ]: fact_t(1,1000)
```

```
-----  
RecursionError                                     Traceback (most recent call last)  
<ipython-input-21-ddce5fd6c117> in <module>()  
----> 1 fact_t(1,1000)  
  
<ipython-input-19-14e11e0a208d> in fact_t(acc, n)  
      3     return acc  
      4     else:  
----> 5     return fact_t(acc*n,n-1)  
      6
```

```

7 def fact(n):
... last 1 frames repeated, from the frame below ...

<ipython-input-19-14e11e0a208d> in fact_t(acc, n)
    3     return acc
    4 else:
----> 5     return fact_t(acc*n,n-1)
    6
    7 def fact(n):

```

`RecursionError: maximum recursion depth exceeded in comparison`

Non! car python étant plutot orienté impératif, ou plutot préfère l'explicite à l'implicite, ne tante pas d'obtimier les récusions alors? quel solution?

Traduire en boucle....

```

def fact_t(acc, n):
    if n==0:
        return acc
    else:
        return fact_t(acc*n,n-1)

def fact_t(acc, n):
    debut_fact_t
    if n==0:
        return acc
    else:
        acc = acc*n
        n = n-1
        aller à debut_fact_t:

```

```
In [ ]: #ce qui donnerait
def fact_t(acc,n):
    while True:
        if n==0:
            return acc
        else:
            acc = acc*n
            n = n-1
```

```
In [ ]: fact_t(1,100000)
```

```
In [ ]: #dernière optimisation
def fact_t(acc,n):
    while(n!=0):
        acc = acc*n
        n = n-1
    # On sort de la boucle quand not (n!=0)

def fact(n):
    return fact_t(1,n)
```

```
In [ ]: #puis finalement tout simplement
def fact(n):
    acc = 1
    while(n!=0):
```

```
acc = acc*n  
n = n-1
```

Qu'aurait donner la fonction fact en raisonnement impératif avec boucle while

on cherche un invariant de boucle on à fait le calcul jusqu'aun certaine étape:

fact\_i = 12...\*i

i=0, fact\_0 = 1

```
In [ ]:  
def fact(n):  
    i=0  
    acc=1  
    while (i<n):  
        #avant acc = 1*2*...*i  
        i=i+1  
        #après acc = 1*2*...*(i+1)  
        acc = acc * i  
    #quand on sort i = n donc acc = 1*2*...*n
```

## Quelques problèmes facile en récursif plus complexe en itératif

### Tour de hanoi



Nous allons voir qu'il est très simple de créer un programme récursif qui nous dit quoi faire pour résoudre le problème. Tout d'abord on appelle les tours A, B et C. On appelle n le nombre de disque présents au départ dans la tour A. Pour déplacer tous les disques de la tour A vers la tour C, on peut raisonner comme suit :

- On déplace n-1 disques de A vers la tour B
- On déplace le dernier disque de A vers C
- On déplace les n-1 disques de B vers C

L'astuce ici est de créer une fonction hanoi qui prend 4 paramètres :

hanoi(n,debut,inter,fin) où n est le nombre de disques à déplacer, debut est la tour de départ de nos n disques, inter est la tour intermédiaire que l'on peut utiliser pour déplacer et fin est la tour ou doivent se trouver les n disques au final.

Ainsi, au début on va lancer `hanoi(n,"A","B","C")` mais quand on va vouloir déplacer les n-1 disques de A vers B, on écrira `hanoi(n-1,"A","C","B")` .

Exercice pour Ecrire cette fonction recursive `hanoi(n,debut,inter,fin)` de manière à afficher (avec print) à chaque étape le déplacement à effectuer sous la forme "A B" pour un

déplacement de la tour "A" vers la tour "B" par exemple.

Entrée :  $(n, "A", "B", "C")$  où  $n$  est un entier.

Sortie : Les instructions à suivre pour déplacer les  $n$  disques de la tour "A" à la tour "C" donnée sous la forme "A B" pour signifier un déplacement de A vers B et affiché avec print.

## Ackerman

$\text{Ack}(m, n)$

- vaut :  $n+1$  si  $m=0$
- $\text{Ack}(m-1, 1)$  sinon et si  $n=0$
- $\text{Ack}(m-1, \text{Ack}(m, n-1))$  autrement