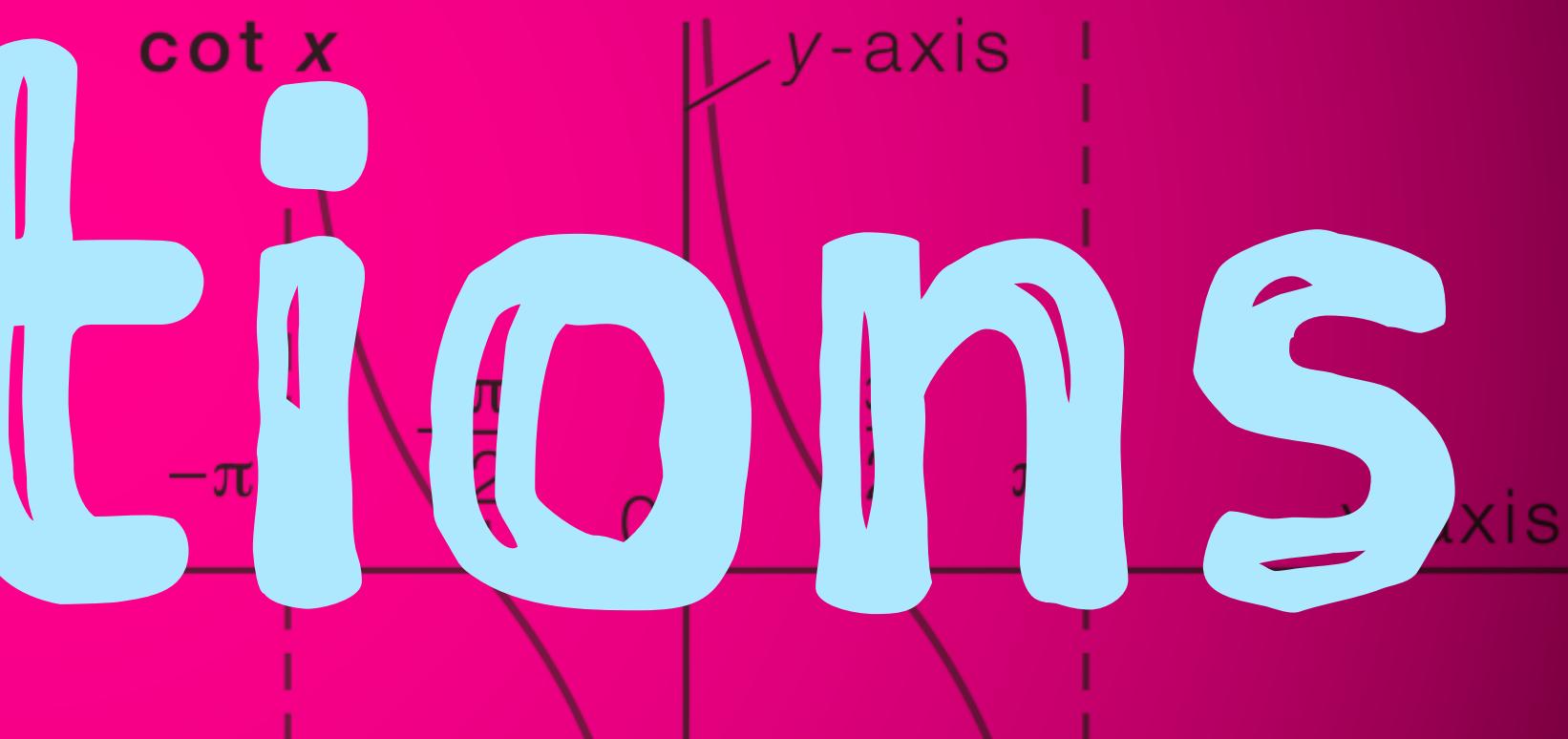


Les

fonctions



Plan

- les (bonnes) bases
 - 1. les arguments
 - 2. les "closures"
 - 3. les générateurs
 - 4. les outils de la librairies Python
- des fonctions partout (brain teaser 

A close-up photograph of a person's face in profile, facing right. The person has short, light-colored hair and is wearing a dark-colored shirt. They are looking down at a laptop computer, which is open and positioned in front of them. The background is slightly blurred, showing what appears to be an indoor setting with warm lighting.

les (bonnes) bases

I. les arguments

Bases

On a vu dans le *primer* comment créer et appeler une fonction python:

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

Cette fonction prends deux arguments: x et y et retourne leur somme.

Bases (suite)

on doit passer **exactement** le nombre d'arguments attendus, sous peine de lever un `TypeError`:

```
>>> add(1,2)
3
>>> add(1)
TypeError: f() missing 1 required positional argument: 'y'
>>> f(1,2,3)
TypeError: f() takes 2 positional arguments but 3 were given
```



Arguments nommés (aussi dans le premier !)

on peut préciser le nom de chaque argument pour les fournir dans le désordre par exemple:

```
# rappel: def f(x, y): return x + y
>>> f(y=2, x=1)
3
```

Arguments nommés (suite)

on peut mélanger les deux styles d'arguments:

```
# ici y est explicitement nommé, mais pas x  
>>> f(1, y=2)  
3
```

mais attention: les arguments nommés doivent être placés à la fin de l'appel à la fonction:

```
>>> f(y=2, 1)  
SyntaxError: positional argument follows keyword argument
```



Stay focused

Nombre variable d'arguments

Valeurs par défaut

```
def g(x, y=10):  
    return y-x
```

Les valeurs par défaut permettent d'éviter de préciser des paramètres qui sont souvent identiques.

Valeurs par défaut (suite)

on peut invoquer la fonction sans préciser y:

```
>>> g(1)
9 # x=1, y=10 (défaut)
```

ou

```
>>> g(1, 2)
1 # x=1, y=2
>>> g(y=3, x=1)
2
```

Nombre variable d'arguments

argument "wildcard" *

```
def h(x, *args):  
    result = x  
    for a in args:  
        result += a  
    return result
```

l'argument `*args` va captuer tous les arguments passés
à `h` après `x`, dans un tuple.

Argument "wildcard" * (suite)

```
>>> h(1)
1 # args = ()
>>> h(1, 2, 3)
6 # args = (2, 3)
```

les autres arguments sont toujours vérifiés (!)

```
>>> h()
TypeError: h() missing 1 required positional argument: 'x'
```

Argument "wildcard" * (suite)

tout argument déclaré après le wildcard doit être nommé explicitement:

```
>>> def h(*args, x): pass
>>> h(1) # args = (1,), x non défini !
TypeError: h() missing 1 required keyword-only argument: 'x'
>>> h(x=1)
# OK
```

Argument "wildcard" * (avancé)

depuis Python 3.7, on peut même déclarer un wildcard qui ne "capture rien":

```
def h(x, *, y, z=3):
    ...

>>> h(1) # x=1, y non défini, z=3
TypeError: h() missing 1 required keyword-only argument: 'y'
>>> h(1, 2) # x=1, le second argument n'a pas de sens
TypeError: h() takes 1 positional argument but 2 were given
>>> h(1, y=2) # x=1, y=2, z=3
# OK
>>> h(1, y=2, z=4) # x=1, y=2, z=4
# OK
```

A black and white line drawing of a skeleton dressed in a formal tuxedo and bow tie. The skeleton is leaning forward over a large top hat, with its hands positioned as if it has just pulled a rabbit out of the hat. The background features decorative elements like a starburst, a small banner with the letter 'C', and a martini glass on the right.

SPOILER: magic is coming

Passage dynamique d'arguments

soit à nouveau notre fonction f:

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

on peut utiliser une liste ou un tuple pour lui passer des arguments "par lot":

```
>>> l = [1, 2]  
>>> f(*l) # équivalent à f(1, 2)  
3
```

c'est l'inverse du wildcard !

Capture des arguments nommés

l'argument wildcard `*args` a un défaut: il ne capture pas les arguments nommés !

exemple:

```
def p(*args):
    print(args)

>>> p(1, 2, 3)
(1, 2, 3)
>>> p(1, 2, 3, z=4)
TypeError: p() got an unexpected keyword argument 'z'
```

Capture des arguments nommés (suite)

il faut donc soit préciser l'argument en question:

```
def p(*args, z):  
    print("args = ", args)  
    print("z = ", z)  
  
>>> p(1, 2, 3, z=4)  
args = (1, 2, 3)  
z = 4 # argument nommé !
```

Capture des arguments nommés (suite)

soit utiliser un autre *wildcard* ** spécialement conçu pour ces arguments nommés:

```
def p(*args, **kwds):
    print("args = ", args)
    print("kwds = ", kwds)

>>> p(1, 2, 3, z=4)
args = (1, 2, 3)
kwds = {'z': 4} # argument nommé capturé dans un dict
```

Passage dynamique d'arguments

on peut à nouveau passer dynamiquement les arguments nommés (ou pas) en "inversant" la syntaxe wildcard:

```
args = (1, 2)
kwds = {'z': 3}
```

```
>>> p(*args, **kwds)
args = (1, 2)
kwds = {'z': 3}
```

(attention à la "**double** astérisque" avant le dict)

Exercices (échauffement)

- écrire une fonction qui calcule le produit de ses arguments, et force le passage d'au moins deux arguments
- écrire une fonction qui prend un argument x (obligatoire) et un argument factor valant 2 par défaut et calcule le produit des deux. On ne peut passer prod que sous forme d'un argument nommé

Exercices (solutions)

```
def p(x, y, *args):  
    result = x*y  
    for a in args:  
        result *= a  
    return result
```

```
def pp(x, *, prod=2):  
    return x*prod
```

2. les "closures" et les lambdas

Un exemple un peu étrange

```
def sum_and_prod(x, y, z):  
    def prod():  
        return y * z  
    return x + prod()
```

- `prod()` est définie à l'intérieur de `sum_and_prod()`
- `prod()` “voit” les arguments `y` et `z` par closure

Deux catégories de fonctions

1. les fonctions dites "pures" qui ne reçoivent leurs valeurs que de leurs arguments explicites
2. les fonctions utilisant les "closures" qui vont récupérer des valeurs de leur environnement

en règle générale, **on préférera les fonctions pures**, mais en faisant attention les closures sont très pratiques

Encore mieux

```
def product_with(x):  
    def _inner(y):  
        return x * y  
    return _inner
```

la fonction `product_with()` retourne une fonction:

```
>>> times2 = product_with(2)  
>>> times2(4)  
8
```



En détail

```
def product_with(x):  
    def _inner(y):  
        return x * y  
    return _inner
```

- on définit une fonction interne
- on retourne la fonction

et on pourra invoquer la fonction plus tard avec (et)

En détail

```
def product_with(x):  
    def _inner(y):  
        return x * y  
    return _inner
```

- on définit une fonction interne
- on retourne la fonction

et on pourra invoquer la fonction plus tard avec (et)

Autre syntaxe: lambda

On peut utiliser le mot-clé `lambda` pour créer des fonctions *anonymes simples*:

```
def product_with(x):  
    return lambda y: x * y
```

la syntaxe est `lambda argument(s) : <EXPRESSION>`

```
>>> times2 = product_with(2)  
>>> times2(4)
```

Exercice

Écrire une fonction `f` que l'on peut invoquer de la manière suivante:

`f(x)(y)(z)`

et qui calcule $x \times (y + z)$:

```
>>> f(2)(3)(4)
```

14

Solution

```
def f(x):  
    def _ff(y):  
        def _fff(z):  
            return x * (y + z)  
        return _fff  
    return _ff
```

or

```
f = lambda x: lambda y: lambda z: x * (y+z)
```



Haskell Curry

1900-1982

3. les générateurs

Bases

Ce sont des fonctions qui "retournent" plusieurs valeurs:

```
def count():
    for i in range(1, 4):
        yield i

>>> for c in count():
...     print(c)
1
2
3
```

appeler la fonction génératrice retourne un objet générateur:

```
>>> count()  
<generator object count at 0x7fa8681eecd0>
```

la boucle for..in va appeler next autant que fois que possible:

```
>>> g = count()  
>>> next(g)  
1  
>>> next(g)  
2  
>>> next(g)  
3  
>>> next(g)  
StopIteration
```

Exercices

- écrire une fonction génératrice $f(n)$ qui énumère les n premiers entiers pairs
- écrire une fonction génératrice $g()$ qui énumère les carrés, dans l'ordre, à l'infini

Solutions

```
def f(n):
    for i in range(0, 2*n, 2):
        yield i

def g():
    x = 0
    while True:
        yield x * x
        x += 1
```

Rappel: les expressions génératrices

C'est un autre façon de créer un générateur:

```
>>> odds = (2*x+1 for x in range(4))
<generator object <genexpr> at 0x7fd0e81eebd0>
>>> for o in odds:
...     print(o)
1
3
5
7
```

4. les outils standard de la lib Python

Bases: map, filter & co.

L'idée est toujours la même:

- on dispose d'une énumérable (aka une liste)
- on a une fonction à appliquer à tous les éléments de la liste

Bases: map

map(FONCTION, _ENUMERABLE)

```
>>> l = [1, 2, 3, 4]
>>> doubles = map(lambda x: x*2, l)
>>> list(doubles)
[2, 4, 6, 8]
```

*ici on applique lambda x: x*2 à chacun de éléments de l*

Bases: map

map(FONCTION, _ENUMERABLE)

```
>>> l = [1, 2, 3, 4]
>>> doubles = map(lambda x: x*2, l)
>>> list(doubles)
[2, 4, 6, 8]
```

*ici on applique lambda x: x*2 à chacun de éléments de l*

Bases: filter

filter(FONCTION, _ENUMERABLE)

```
>>> l = [1, 2, 3, 4]
>>> evens = filter(lambda x: x%2==0, l)
>>> list(evens)
[2, 4]
```

- on applique lambda $x: x \% 2 == 0$ à chaque élément de l
- si le résultat est True on garde l'élément

Bases: filter

filter(FONCTION, _ENUMERABLE)

```
>>> l = [1, 2, 3, 4]
>>> evens = filter(lambda x: x%2==0, l)
>>> list(evens)
[2, 4]
```

- on applique lambda $x: x \% 2 == 0$ à chaque élément de l
- si le résultat est True on garde l'élément

Bases: all et any

Ils opèrent sur des générateurs, on les utilise souvent avec les expressions génératrices:

```
>>> l = [1, 4, 8]
>>> all(x < 5 for x in l)
False # l contient des éléments  $\geq 5$ 
>>> all(x < 10 for x in l)
True # tous les éléments de l sont < 10
>>> any(x < 5 for x in l)
True # certains (ou moins 1) éléments de l sont < 5
>>> any(x > 8 for x in l)
False # aucun élément de l n'est > 8
```



Easy

Évaluation partielle avec partial

permet de "figer" à l'avance certains arguments:

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

```
>>> from functools import partial  
>>> plus2 = partial(add, 2)  
>>> plus2(5)
```

Énumération diverses avec itertools

product

```
>>> from itertools import product
>>> l = ['a', 'b']; g = [ 1, 2]; h = [True, False]
>>> for x in product(l, g, h):
    print(x)
('a', 1, True)
('a', 1, False)
('a', 2, True)
('a', 2, False)
('b', 1, True)
('b', 1, False)
('b', 2, True)
('b', 2, False)
```

Énumération diverses avec itertools

permutations

```
>>> from itertools import permutations
>>> l = ['a', 'b', 'c']
>>> list(permutations(l, 2))
[
    ('a', 'b'), ('a', 'c'), ('b', 'a'),
    ('b', 'c'), ('c', 'a'), ('c', 'b')
]
>>> list(permutations(l, 3))
[
    ('a', 'b', 'c'), ('a', 'c', 'b'), ('b', 'a', 'c'),
    ('b', 'c', 'a'), ('c', 'a', 'b'), ('c', 'b', 'a')
]
```

Énumération diverses avec itertools

combinations

```
>>> from itertools import combinations  
>>> l = ['a', 'b', 'c']  
>>> list(combinations(l, 2))  
[('a', 'b'), ('a', 'c'), ('b', 'c')]  
>>> list(permutations(l, 3))  
[('a', 'b', 'c')] 
```

Exercice

soit une liste de couleurs et de valeurs de cartes:

```
colors = [ '♦', '♠', '♥', '♣']  
values = [ '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']
```

- générer la liste de toutes les cartes possibles (ex. '♥7', '♣A' etc)
- générer toutes les mains de 5 cartes possibles

Solutions

```
cards = map(''.join, product(colors, values))  
hands = combinations(cards, 5)
```



Changeons de sujet...



des fonctions partout

ou comment TOUT remplacer par des fonctions

C'est à dire ?

Nous allons explorer l'expressivité des fonctions:

- ce qu'on peut exprimer avec des fonctions "simples"
- leur rôle central tant pratique que "théorique"

La règle du jeu

Nous allons donc considérer des fonctions (et seulement des fonctions 😊) ayant les propriétés suivantes:

- prennent **un argument**
- retournent une valeur (pas de None)

& that's all folks

Exemple(s)

OUI 

f = lambda x: x+1

f = lambda x: lambda y: x + y

NON 

f = 1 # pas une fonction...

f = lambda: 2 # pas d'argument

f = lambda x,y: x+y # trop d'arguments

Quelques astuces

on peut ignorer un argument:

```
f = lambda _: 1 # j'ignore mon argument
```

ou traiter plusieurs arguments par currying:

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

```
f = lambda x: lambda y: lambda z: x + y - z
```

Attention ✕

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

Attention ✕

f = lambda x: x+1

→ j'ai triché...

Attention ✕

f = lambda x: x+1

- j'ai triché...
- 1 n'est pas une fonction

Attention ✕

f = lambda x: x+1

- j'ai triché...
- 1 n'est pas une fonction
- + n'est pas une fonction

Attention ✕

f = lambda x: x+1

- j'ai triché...
- 1 n'est pas une fonction
- + n'est pas une fonction
- or on ne veut que des fonctions !



Church Numerals

Commençons par remplacer les nombres par des fonctions.

Nous allons définir **un nombre** "n: comme **une fonction**:

- qui "prend 2 arguments" f et x
- applique n fois f à x
- et retourne le résultat

Church Numerals: exemple

on a donc le nombre un:

```
# applique 1 fois f à x et retourne ne résultat  
un = lambda f: lambda x: f(x)
```

et le nombre deux:

```
# applique 1 fois f à x et retourne ne résultat  
deux = lambda f: lambda x: f(f(x))
```

Exercices

- définir le nombre trois
- définir le nombre zero
- écrire un code python qui, étant donné un nombre de Church, affiche sa valeur numérique (1, 2, ...)

Solutions

```
trois = lambda f: lambda x: f(f(f(x)))
```

```
zero = lambda f: lambda x: x
```

```
def print_church(n):
    print(n(lambda x: x+1)(0))
```



Attention ça va se
compliquer

Exercice: le successeur

on cherche à définir une fonction `succ` qui retourne le successeur d'un nombre ($x \mapsto x + 1$).

Par étape:

- définir deux en fonction de un
- définir trois en fonction de deux
- en déduire la définition de `succ`

```
deux = lambda f: lambda x: f(un(f)(x))
```

```
trois = lambda f: lambda x: f(deux(f))(x)
```

le successeur prend un de nos nombres n et retourne un nouveau nombre:

```
succ = lambda n: lambda f: lambda x: f(n(f))(x)
```

Exercice: d'addition

créer une fonction somme qui réalise l'addition de deux nombres de Church $x, y \mapsto x + y$:

```
>>> print_church(somme(un)(deux))
```

3

Solution

Rappel: un nombre de Church N prend une f et x et applique N fois f à x.

Du coup pour ajouter x et y, il suffit de faire x fois succ sur y (ou y fois succ sur x c'est pareil):

```
somme = lambda x: lambda y: x(succ)(y)
```

```
>>> print_church(somme(un)(deux))
```

3

```
>>> print_church(somme(deux)(deux))
```

4

Exercice: la multiplication

créer une fonction mult qui réalise la multiplication de deux nombres de Church $x, y \mapsto x * y$:

```
>>> print_church(mult(un)(deux))  
2
```

si vous avez compris somme vous êtes en bonne voie

Solution

pour multiplier n par m:

- on part de zero
- on ajoute +m, n fois

```
mult = lambda n: lambda m: n(somme(m))(zero)
```

```
>>> print_church(mult(deux)(deux))
```

4

```
>>> print_church(mult(deux)(zero))
```

0

Cadeau: la soustraction

(plus de détails "en bonus" à la fin)

```
pred = lambda n: lambda f: lambda x: \
    n(lambda g: lambda h: h(g(f)))(lambda _: x)(lambda z: z)
```

```
>>> print_church(pred(deux))
```

1

```
>>> print_church(pred(un))
```

0



on a remplacé le + et le 1
par des fonctions !

NOT BAD

Mais c'est encore très insuffisant:

- l'informatique ce n'est pas que des nombres !
- c'est aussi des tests
- et des boucles

peut-on faire ça avec des fonctions ?

(incide: oui)

Posons-nous un problème classique

```
def fact(n):
    if n == 0:
        return 1
    else
        return fact(n-1)
```

Les booléens

soit les expressions “booléennes” suivantes:

```
true = lambda x: lambda y: x
```

```
false = lambda x: lambda y: y
```

Exercice: print_boolean

créer une fonction print_boolean qui prend un boolean "lambda" et retourne True ou False

```
>>> print_boolean(true)
```

True

```
>>> print_boolean(false)
```

False

Solution

```
def print_boolean(b):  
    print(b(True)(False))
```

Exercice: le test à zéro

créer une fonction `is_zero` qui test si un nombre (de Church) est zero:

```
>>> is_zero(zero)
```

True

```
>>> is_zero(zero)
```

True

Solution

```
is_zero = lambda n: n(lambda _: False)(True)
```

- si f n'est jamais appelé renvoie x = true
- sinon renvoie false

Exercice: le if

créer un opérateur si qui prend un booléen b, et deux valeurs x et y: si le b est vrai si renvoie x, sinon si renvoie y

```
>>> print_church(si(true)(un)(deux))
```

1

```
>>> print_church(si(false)(un)(deux))
```

2

Solution

la logique de "branchement" est directement dans le booléens eux-mêmes:

```
si = lambda b: lambda x: lambda y: b(x)(y)
```

Ce qui reste à faire

on se rend compte qu'on peut ré-écrire une bonne partie de notre factorielle:

```
fact = lambda n: si(is_zero(n))(un)(mult(n)(fact(pred(n))))
```

mais ça bug:

```
>>> print_church(fact(quatre))
RecursionError: maximum recursion depth exceeded
```

Question: pourquoi ???

Problème

Python évalue les arguments avant d'appeler la fonction

```
fact = lambda n: si(is_zero(n))(un)(mult(n)(fact(pred(n))))
```

pour fixer ce problème on va rendre les branches "lazy":

```
fact = lambda n: si(is_zero(n))(lambda _: un)(lambda _: mult(n)(fact(pred(n))))(zero)
```

- on wrap les deux branches du *si* dans une fonction
- et on appelle la fonction (ici avec zero)

A black and white photograph of a group of police officers standing in a line. They are all wearing dark uniforms with "SHERIFF" printed across the chest. They have serious, stern expressions on their faces. The background is dark and out of focus.

Problème n°2: on a triché !



on a jamais autorisé la récursion

"U-combinator"

pour éliminer fact de notre expression: on en fait un argument !

```
fact = lambda f: lambda n: \
    si(is_zero(n)) \
        (lambda _: un) \
        (lambda _: mult(n)(f(f)(pred(n)))) \
    (zero) # <- uniquement là pour "extraire" la valeur
```

```
>>> print_church(fact(fact)(quatre))
```

24

"Y-combinator"

passer la fonction en argument d'elle-même c'est inélégant, introduisons un nouvel opérateur:

$$Y = \lambda f. (\lambda x. (f(xx)))\lambda x. (f(xx)))$$

soit:

```
Y = lambda f: \
    (lambda x: x(x))(lambda y: f(lambda z: y(y)(z)))
```

"Y-combinator" (en pratique)

```
fact = Y(lambda f: lambda n: \
    si(is_zero(n)) \
        (lambda _: un) \
        (lambda _: mult(n)(f(pred(n)))) ) \
    #                                     ^ on ne passe plus f à lui-même !
    (zero))
```

```
>>> print_church(fact(quatre))
```

24

Bonus (TODO...)

- comprendre la fonction "prédescesseur" et la "wisdom's torch" ?
- les structures de données: paires (nécessaire pour le précédent), les listes "à la lisp" (avec cons, car, cdr) ?
- avec le pred, écrire la soustraction ?
- à partir du is_zero, on peut définir les opérateur de comparaison (plus grand que etc.) ?
- très similaire mais plus complexe: la suite de Fibonacci ?
- dériver le "Y-combinator" de façon "pratique" ?
- mini poly avec un peu de théorie (lambda calcul non typé) ?