Merde , on tourne en rond,
merde, on tourne en rond,
merde, on tourne en rond,
merde , on tourne en rond.

Bernard Blier [a], Le Grand Blond avec une chaussure noire (1972), écrit par Francis Veber

a. `https://www.youtube.com/watch?time_continue=4&v=ej_IKoF325k&feature=emb_logo`

3 La récursivité

Extrait du programme

Thème : Langages et programmation

Contenus :

Récursivité.

Capacités attendus :

Écrire un programme récursif.

Analyser le fonctionnement d'un programme récursif.

Commentaires :

Des exemples relevant de domaines variés sont à privilégier.

Vous pouvez retrouver le cours complet à l'adresse suivante :
`https://github.com/NaturelEtChaud/NSI-Terminale/tree/main/3%20R%C3%A9cursivit%C3%A9`

I. Fonction récursive

a) Une définition

La définition d'une **fonction récursive** est assez simple. C'est une fonction qui s'appelle elle-même à l'intérieur de sa propre définition !
Vous avez ci-contre un exemple d'une fonction récursive. On peut aisément deviné ce qu'elle produira.

```
def bernard():
    print("On tourne en rond, merde")
    bernard()
```

Mais finalement cet exemple n'est pas un bon exemple puisque *en général*, on préfère que le programme s'arrête.
Par défaut, Python limite le nombre d'appels récursifs à 1000. Au-delà, on obtient le message d'erreur suivant `RecursionError : maximum recursion depth exceed.`
Cette limite peut se modifier. Par exemple, si on souhaite la passer à 2000 :

```
import sys
sys.setrecursionlimit(2000)
```

b) Un exemple

Cet exemple est inspiré du livre ***Numérique et Sciences Informatiques - Terminale*** *de Balabonski, Conchon, Filliâtre et Nguyen.*

Imaginons que l'on veuille effectuer la somme n premiers entiers :

$$S = 1 + 2 + 3 + 4 + ... + (n-1) + n$$

Une première solution consiste à ajouter étape après étape tous les entiers jusqu'à n. C'est ce que nous pourrions facilement faire avec une boucle **POUR**.

```
def somme(n):
    calcul = 0
    for i in range(1,n+1):
        calcul = calcul + i
    return calcul
```

On pourrait aussi imaginer le calcul d'une autre manière :
- *Grand Sage, je dois effectuer le calcul suivant* $1 + 2 + 3 + 4 + ... + (n-1) + n$. *Pourrais-tu m'aider ?*
- *Bien sûr Petit Scarabée, fais moi-d'abord le calcul* $1 + 2 + 3 + 4 + ... + (n-1)$ *je rajouterai ensuite* n *!*

Autrement dit le calcul pourrait se faire ainsi :

$$S = \underbrace{1 + 2 + 3 + 4 + ... + (n-1)}_{\text{Pour le Petit Scarabée}} \quad \underbrace{+ n}_{\text{Pour le Grand Sage}}$$

Lorsque l'on effectue un programme récursif, on va procéder comme le Grand Sage, on ne va pas s'embarrasser de savoir comment on effectue le calcul $1 + 2 + 3 + 4 + ... + (n-1)$, on va le laisser au Petit Scarabée. On va juste programmer la dernière étape en considérant que le Petit Scarabée a bien effectué son travail [1].

Notre programme pourrait alors s'écrire ainsi :

```
def somme(n):
    return n + somme(n-1)
```

1. Je me demande si ce n'est pas un peu trop risqué d'avoir autant confiance dans le travail de notre Petit Scarabée...

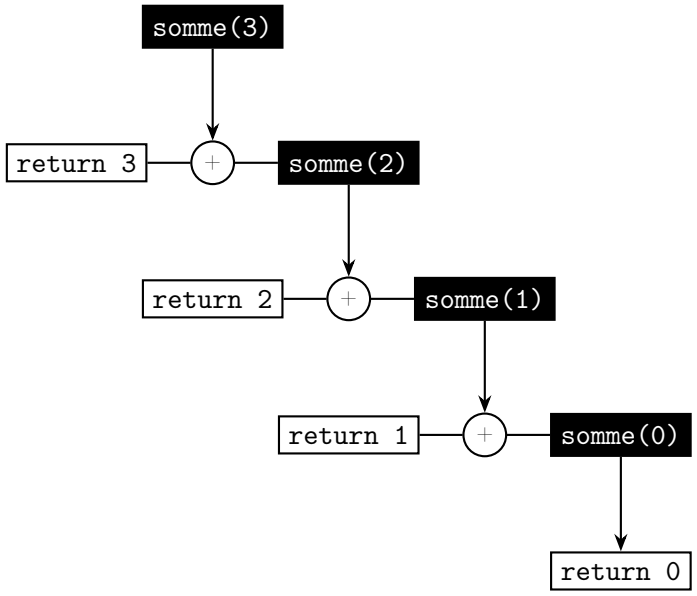
Le problème c'est que ce programme ne va pas donner le résultat attendu !
En effet, si je tape la commande `somme(3)`, le programme va appeler `somme(2)` qui va appeler `somme(1)` qui va lui-même appeler `somme(0)` puis ensuite `somme(-1)` puis encore `somme(-2)`, ...
Les appels récursifs ne vont jamais s'arrêter, comme notre brave fonction `bernard()`.
Il nous faut obligatoirement rajouter un cas de base.

Notre programme devient alors :

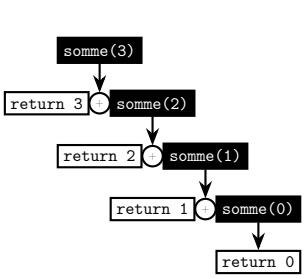
```
def somme(n):
    if n == 0 :
        #cas de base
        return 0
    else :
        return n + somme(n-1)
```

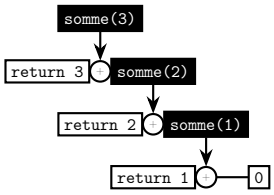
Que se passe-t-il si on tape la commande `somme(3)` ?
Pour calculer la valeur renvoyé par `somme(3)`, il faut d'abord appeler `somme(2)`. Cet appel va lui-même déclencher un appel à `somme(1)` qui à son tour va effectuer un appel à `somme(0)`. Ce dernier appel se termine directement en renvoyant la valeur 0.

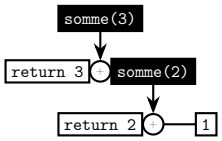
Une fois que la valeur 0 a été renvoyé, l'appel à `somme(0)` est remplacé par 0 dans l'expression `return 1 + somme(0)`.
A cet instant, l'appel à `somme(1)` peut alors se terminer et renvoyer le résultat sous la forme `1+0`.
Enfin, l'appel à `somme(2)` peut lui-même renvoyer la valeur `2+1` comme résultat, ce qui permet à `somme(3)` de se terminer en renvoyant le résultat `3+3`.

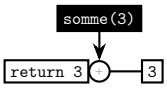


On parle alors de **pile d'appels**.









On peut retrouver ce phénomène, en ligne, sur Pythontutor : http://pythontutor.com/

c) Définitions récursives bien formées

Il est important de respecter les quelques règles élémentaires suivantes lorsque l'on écrit une fonction récursive :

- s'assurer que les valeurs utilisées pour appeler la fonction soient **strictement décroissantes** au fur et à mesure des appels.
- s'assurer que l'on a bien défini un **cas de base**.
- s'assurer que les valeurs utilisées pour appeler la fonction restent dans le **domaine de définition** de la fonction.

Voici quelques exemples problématiques.

Exemple (1)
Ligne 5, les valeurs utilisées ne sont pas décroissantes. Le cas de base ne sera alors jamais atteint.
En théorie, on obtient une pile d'appels infinie.

```
1 def somme(n):
2     if n==0:
3         return 0
4     else :
5         return n + somme(n+1)
```

Exemple (2)
Les valeurs utilisées sont bien strictement décroissantes et nous avons bien un cas de base.
Malheureusement, ce cas de base n'est pas toujours atteint. En effet pour les nombres impairs nous avons un problème. Par exemple, pour `somme(1)`, on va faire un appel à `somme(-1)` puis à `somme(-3)`, ...
En théorie, on obtient à nouveau une pile d'appels infinie.

```
1 def somme(n):
2     if n==0:
3         return 1
4     else :
5         return n + somme(n-2)
```

Exemple (3)
La fonction récursive est bien définie mais l'appel de la fonction ne respecte pas le domaine de définition de la fonction. Il n'y aura pas d'arrêts des appels car on passera à côté du cas de base.

```
1 def somme(n):
2     if n==0:
3         return 0
4     else :
5         return n + somme(n-1)
6 somme(3.5)
```

Revenons sur ce dernier exemple. Une solution serait de s'assurer que l'on utilise bien la fonction `somme()` avec un entier positif. On pense donc naturellement à utiliser un `assert`.
Cette solution est parfaitement correcte, mais elle pose un petit souci. Une fois le premier appel vérifié, les autres appels se feront bien évidemment sur des nombres entiers positifs. Il n'est plus nécessaire d'effectuer à chaque fois ces tests.

```
1 def somme(n):
2     assert (type(n)==int) and (n>=0)
3     if n==0:
4         return 0
5     else :
6         return n + somme(n-1)
```

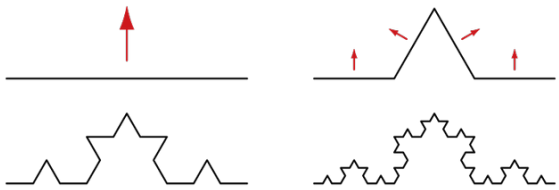
Une solution consiste à effectuer l'assertion dans une fonction principale `somme()` qui lancera, si le teste est réussi, la fonction récursive `somme_rec()`.

```
1 def somme_rec(n):
2     if n==0:
3         return 0
4     else :
5         return n + somme_rec(n-1)
6
7 def somme(n)
8     assert (type(n)==int) and (n>=0)
9     return somme_rec(n)
```

II. Exercices

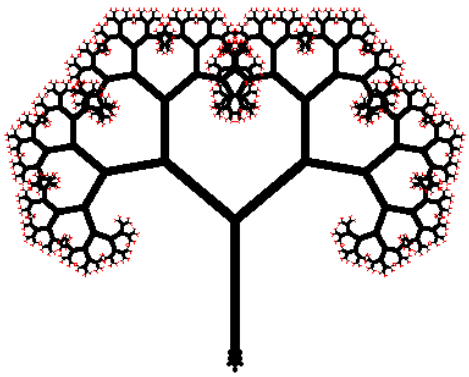
Exercice 1

Programmer à l'aide du module `turtle` la courbe de Koch [2].
Vous pouvez trouvez quelques variantes de la courbe de Koch sur le lien suivant :
`https://natureletchaud.github.io/recursivite/`.



Exercice 2

Programmer à l'aide du module `turtle` un arbre récursif.



Exercice 3 (Travail à la main)

On considère la fonction ci-contre. Construire la pile d'appels correspondant à la commande `puis(3,5)` puis calculer le résultat obtenu.

```
def puiss(x,n) :
    if n == 0 :
        return 1
    elif n%2 == 0 :
        return puiss(x,n//2)**2
    else :
     return x*puiss(x,(n-1)//2)**2
```

Exercice 4 (Travail à la main)

On considère la fonction ci-contre. Construire la pile d'appels correspondant à la commande `fib(5)` puis calculer le résultat obtenu.

```
def fib(n):
    if n == 0 :
        return 0
    elif n == 1 :
        return 1
    else :
     return fib(n-1) + fib(n-2)
```

Exercice 5

Écrire une fonction récursive qui inverse l'ordre des caractères dans un texte.

2. `https://fr.wikipedia.org/wiki/Flocon_de_Koch`

Exercice 6
Écrire une fonction récursive qui vérifie qu'un mot ou un texte est bien un palindrome.
On pourra tester le programme avec les palindromes suivants :
`https://www.topito.com/top-10-des-palyndromes-bien-nazes-que-tout-le-monde-connait-mais-qui-font-classe-q`
On pourra aussi vérifier le palindrome écrit par Georges Perec en 1969, « Au moulin d'Andé » comportant 1247 mots (c'est le record !) :
`https://jeretiens.net/palindrome-de-georges-perec-au-moulin-dande/`

Exercice 7
Écrire une fonction récursive qui indique tous les chemins accessible depuis un répertoire [3].

- On utilisera le module `os` avec la commande classique `import os`.
- `os.path.isdir(chemin)` permet de vérifier si `chemin` est un répertoire valide ou non, autrement dit si c'est un répertoire ou un fichier. La réponse est un booléen.
- `os.listdir(chemin)` donne la liste de tous les sous-répertoires et fichiers contenus dans `chemin`.

Exercice 8
Écrire une fonction récursive qui inverse l'ordre d'une pile.
On utilisera l'implémentation ci-dessous.

```
class Pile:
    def __init__(self):
        self.pile = []

    def empiler(self, e):
        self.pile.append(e)

    def sommet(self):
        assert len(self.pile) > 0, "Pile vide!"
        return self.pile[-1]

    def depiler(self):
        assert len(self.pile) > 0, "Pile vide!"
        s = self.pile.pop()
        return s

    def estVide(self):
        return len(self.pile) == 0

    def __str__(self):
        retour = ""
        for e in range(len(self.pile)-1, -1,-1):
                retour = retour + str(self.pile[e]) + '\n'
        retour = retour + "====\n"
        return retour
```

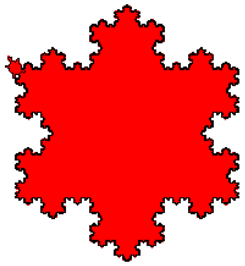
3. Sur une idée de Guillmaume Connan, professeur et formateur de l'académie de Nantes

III. Corrections

Correction de l'exercice 1

```
from turtle import *
shape("turtle")

#longueur de la courbe
l = 243
#nombre d'appels récursif
n = 5

#la fonction définie par récursivité
def koch(n,l):
    if n == 0:
        forward(l)
    else:
        koch(n-1, l/3)
        left(60)
        koch(n-1, l/3)
        right(120)
        koch(n-1, l/3)
        left(60)
        koch(n-1, l/3)

#on positionne la tortue
up()
right(180)
forward(l/2)
right(180)
down()

#c'est parti pour faire un beau dessin
koch(n,l)

#pour fermer proprement
exitonclick()
mainloop()
```

On peut aussi appeler 3 fois la fonction pour obtenir le flocon de Koch. Ci-dessous le flocon obtenu après 6 appels.



Correction de l'exercice 2

Une solution possible. Plus les appels sont importants et plus les branches anciennes sont épaisses.

```
from turtle import *

#longueur du tronc
l = 150
#angle de l'inclinaison de la branche
o = 50
#rapport entre branche et tronc
r = 2/3
#nombre d'appels récursif
appel = 11

#la fonction définie par récursivité
def arbre(n,l):
    if n == 0:
        width(1)
        color('red')
        forward(l)
        right(180)
        forward(l)
```

```
        color('black')
    else:
        #le tronc
        width(n)
        forward(l)
        #la branche à droite
        right(o)
        width(n-1)
        arbre(n-1, l*r)
        #la branche à gauche
        right(180-2*o)
        arbre(n-1, l*r)
        #retour à la base du tronc
        right(o)
        width(n)
        forward(l)

#construction de l'image
shape("turtle")
fillcolor("red")
#on positionne la tortue
up()
right(90)
forward(l)
down()
left(180)

#c'est parti pour faire un beau dessin
arbre(n,l)

#pour fermer proprement
exitonclick()
mainloop()
```

Correction de l'exercice 3 (Travail à la main)

Voici la pile d'appels

- `puiss(3,5)`
- `puiss(3,5)` = `3*puiss(3,2)**2` car 5 est impair
- `puiss(3,2)` = `puiss(3,1)**2` car 2 est pair
- `puiss(3,1)` = `3*puiss(3,0)**2` car 1 est impair
- `puiss(3,0)` = 1

On peut ensuite remonter la pile d'appels.

- `puiss(3,0)` = 1
- `puiss(3,1)` = `3*puiss(3,0)**2` = `3*1**2` = 3
- `puiss(3,2)` = `puiss(3,1)**2` = `3**2` = 9
- `puiss(3,5)` = `3*puiss(3,2)**2` = `3*9**2` = 243

```
def puiss(x,n) :
    if n == 0 :
        return 1
    elif n%2 == 0 :
        return puiss(x,n//2)**2
    else :
     return x*puiss(x,(n-1)//2)**2
```

On vient simplement de calcule 3^5 avec une méthode très efficace qui s'appelle la méthode d'exponentiation rapide [4].

4. https://fr.wikipedia.org/wiki/Exponentiation_rapide

Correction de l'exercice 4 (Travail à la main)

Dans ce cas, il ne s'agit pas tout à fait d'une pile, mais plutôt d'un arbre d'appels.

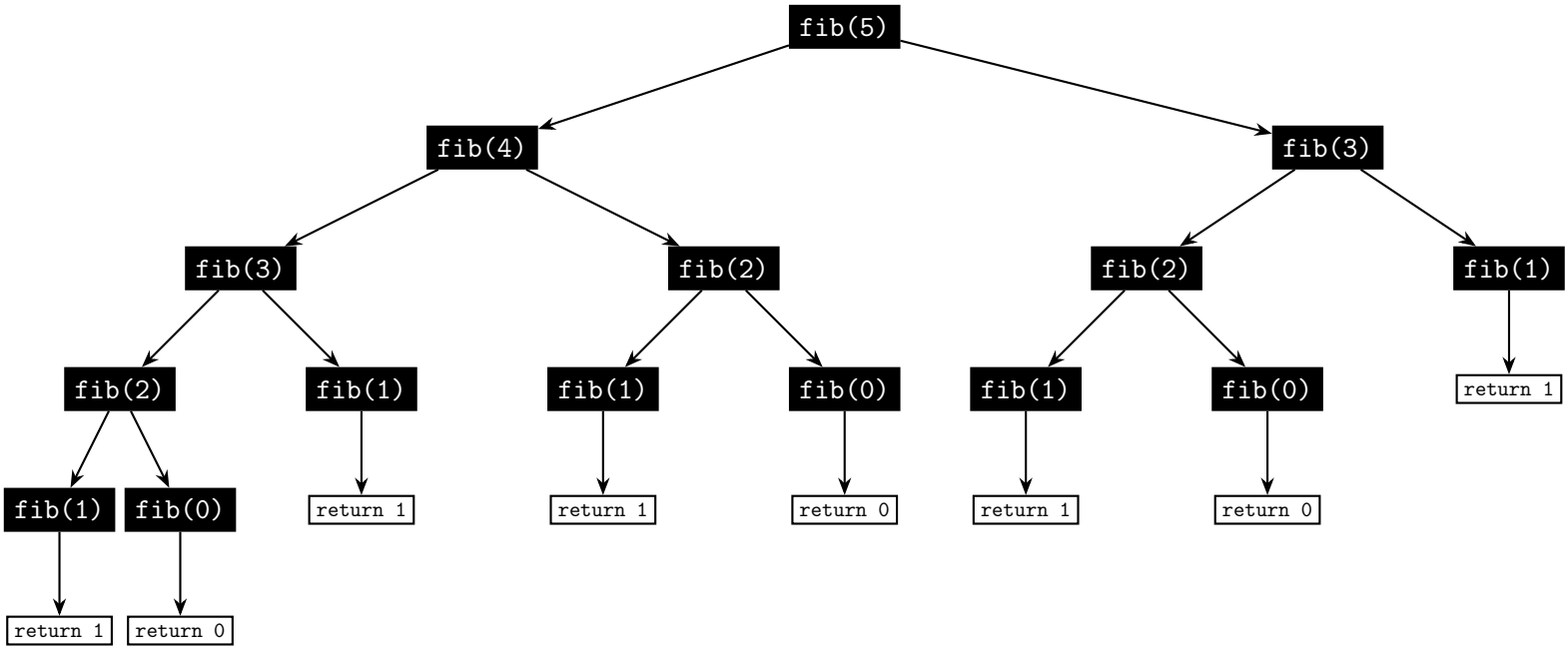
En remontant *tranquillement* chaque branche de l'arbre, on en déduit les résultats intermédiaires suivants :

- `fib(2)` renvoie 1
- `fib(3)` renvoie 2
- `fib(4)` renvoie 3
- `fib(5)` renvoie 5

```python
def fib(n):
    if n == 0 :
        return 0
    elif n == 1 :
        return 1
    else :
     return fib(n-1) + fib(n-2)
```

Certains auront très sûrement reconnu la fameuse suite de Fibonacci [5].

Un Petit Scarabée averti se rendrait également compte du manque d'efficacité de cet algorithme car `fib(2)` est calculé à 3 reprises. De même `fib(3)` est calculé à 2 reprises. On pourrait essayer de gagner en efficacité en stockant au fur et à mesure certains calculs. C'est entres autres, ce qu'on appelle de la **programmation dynamique**. Nous en reparlerons plus tard dans l'année... ou pas !



Correction de l'exercice 5

On utilise dans ce programme quelques petites feintes sur les chaînes de caractères :

- `texte[0]` renvoie le premier caractère `'v'`
- `texte[1:]` renvoie les caractères de la chaîne en partant de celui indicé 1 jusqu'à la fin `'oici un texte'`
- `inverse(chaine[1:]) + car` le symbole `+` est le symbole de la concaténation

```python
texte = "voici un texte"

def inverse(chaine):
    long = len(chaine)
    if long == 0:
        return chaine
    elif long == 1:
        return chaine
    else:
        #on prend le premier caractère
        car = chaine[0]
        #on le place à fin de la chaine sans le premier caractère
        return inverse(chaine[1:]) + car

print(inverse(texte))
```

5. https://fr.wikipedia.org/wiki/Suite_de_Fibonacci

Correction de l'exercice 6

```
1  # -*- coding: utf-8 -*-
2
3  texte = "Éric notre valet alla te laver ton ciré"
4
5  def preTraitement(chaine):
6      """
7      la chaîne de caractères est convertie en minuscule sans les espaces
8      """
9      modif = chaine.lower()
10     modif = modif.replace(' ','')
11     return modif
12
13 def palindrome_rec(chaine):
14     long = len(chaine)
15     if long <= 1:
16         return True
17     else:
18         #on prend le premier et le dernier caractères
19         pre, der = chaine[0], chaine[-1]
20         return (pre == der) and palindrome_rec(chaine[1:long-1])
21
22
23 def palindrome(chaine):
24     modif = preTraitement(chaine)
25     return palindrome_rec(modif)
26
27 print(palindrome(texte))
```

Correction de l'exercice 7

Voici la solution proposée par Guillaume Connan.

La ligne 9, `def printFichiers(chemin: str) -> None:` peut paraître assez surprenante. On aurait pu simplement écrire `def printFichiers(chemin):`

C'est une juste une précision pour les lecteurs du code, la fonction `printFichiers()` prend en argument `chemin` qui doit être de type `str`. La fonction renvoie `None` autrement dit, rien.

Par contre rien ne vous empêche de ne pas respecter cette précision. Ce n'est pas une assertion.

```
1  # -*- coding: utf-8 -*-
2  """
3  Created 18/09/20
4  @author: Guillaume Connan
5  """
6
7  import os
8
9  def printFichiers(chemin: str) -> None:
10     if not os.path.isdir(chemin):
11         #ce n'est pas un répertoire donc c'est un fichier
12         print(chemin)
13     else:
14         for sous_chemin in os.listdir(chemin):
15             printFichiers(chemin + "/" + sous_chemin)
```

Correction de l'exercice 8

```
1  # -*- coding: utf-8 -*-
2  class Pile:
3      def __init__(self):
4          self.pile = []
5
6      def empiler(self, e):
7          self.pile.append(e)
8
9      def sommet(self):
10         assert len(self.pile) > 0, "Pile vide!"
11         return self.pile[-1]
12
13     def depiler(self):
14         assert len(self.pile) > 0, "Pile vide!"
15         s = self.pile.pop()
16         return s
17
18     def estVide(self):
19         return len(self.pile) == 0
20
21     def __str__(self):
22         retour = ""
23         for e in range(len(self.pile)-1, -1,-1):
24                 retour = retour + str(self.pile[e]) + '\n'
25         retour = retour + "====\n"
26         return retour
27
28
29 def inverser(pile, nvpile=Pile()):
30         """
31         fonction récursive qui inverse l'ordre d'une pile
32         """
33     if pile.estVide() :
34         return nvpile
35     else :
36         #on supprime le sommet de la pile et on le récupère
37         top = pile.depiler()
38         #on l'empiler dans la nouvelle pile
39         nvpile.empiler(top)
40         return inverser(pile, nvpile)
41
42
43 a = Pile()
44 #création d'une pile contenant 0, 1, 2, 3, 4, 5, 6
45 for i in range(7):
46     a.empiler(i)
47 #on imprime la pile pour le plaisir
48 print(a)
49
50 a = inverser(a)
51 print(a)
```

La ligne 29 `def inverser(pile, nvpile=Pile()):` peut surprendre. On attend 2 arguments `pile` et `nvpile=Pile()`.
Si on ne met qu'un seul argument, comme à la ligne 50, `a = inverser(a)`, le deuxième argument `nvpile=Pile()` est automatiquement créé comme étant une pile vide.
Par contre si on met un deuxième argument, comme à la ligne 40, `return inverser(pile, nvpile)`, ce deuxième argument est bien pris en compte.