Fonction

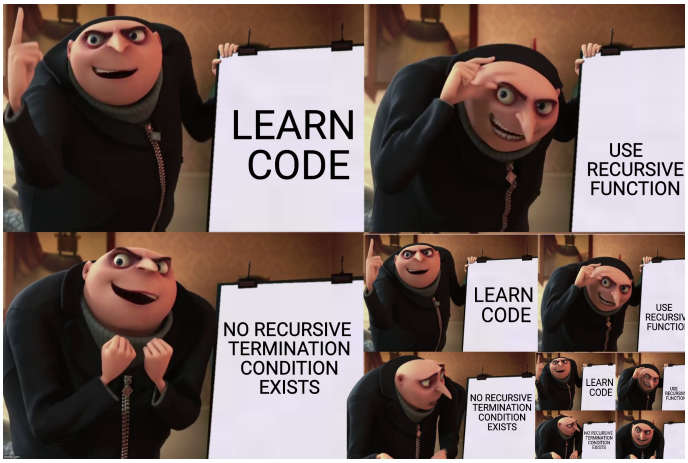


Table des matières

1 Fonction **1**
1.1 Fonction . . . 1
1.2 Les fonctions comme valeurs . . . 2
1.3 Assertion, test unitaire . . . 3
1.4 Sortie anticipée . . . 4

2 Variable locale et globale **5**
2.1 Variable locale . . . 5
2.2 Variable globale . . . 6
2.3 Composition de fonctions . . . 6

3 Programmation récursive **7**
3.1 Fonction récursive pure . . . 7
3.2 Fonction récursive impérative . . . 9
3.3 Fonctions mutuellement récursives . . . 11

1 Fonction

1.1 Fonction

La syntaxe générale d'une fonction en Python est

```
def nom_fonction(arg1, ..., argn):
    bloc...................
    .........d'instructions
```

Le bloc d'instruction a pour vocation soit :
— de calculer une nouvelle valeur qui est renvoyée à l'aide de l'instruction `return`.
— d'avoir un effet de bord comme afficher du texte sur la sortie standard.
Les arguments `arg1`, ..., `argn` sont appelés *arguments formels*. On appelle une fonction à l'aide de la syntaxe `nom_fonction(arg1, ..., argn)` ; les valeurs des expressions `arg1`, ..., `argn` sont appelées *arguments effectifs*.

Une fonction renvoie toujours *une unique* valeur. On utilise pour cela l'instruction `return`. Lorsqu'on souhaite seulement effectuer un effet de bord, on renvoie `None`, ce que Python fait automatiquement s'il ne rencontre pas de `return`.

Python étant un langage de programmation à *typage dynamique*, nous n'avons pas besoin de préciser, ni les types des arguments, ni le type de la valeur de retour. Cette caractéristique du langage nous permet d'avoir des fonctions acceptant des arguments effectifs de types différents :

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

```
In [1]: f(2)
Out[1]: 3

In [2]: f(2.0)
Out[2]: 3.0
```

L'idée est que f peut accepter comme argument n'importe quel type que l'on peut ajouter à 1. En Python, les types `int` et `float` sont de bons candidats. On peut dire que la fonction f accepte un nombre et renvoie un nombre. On dit que Python fonctionne avec le principe du « duck typing » dont la devise est : « If it walks like a duck and it quacks like a duck, then it must be a duck ». Autrement dit, dans notre cas, si $f(x)$ a un sens, c'est que x est un nombre.

Cependant, le plus souvent, nous définirons des fonctions qui ont vocation à être utilisées avec des paramètres d'un type donné. Dans ce cas, la valeur de retour est généralement aussi d'un type déterminé. Par exemple, la fonction

```
def est_pair(n):
    return n % 2 == 0
```

est pensée pour prendre en entrée un entier ; elle renvoie alors un booléen. On dit que la signature de cette fonction est `est_pair(n: int) -> bool`. En Python, si le bloc d'instructions commence par une chaine de caractères, elle est utilisée comme documentation. L'utilisation des triples `"` permet de rentrer des chaines de caractères qui s'étirent sur plusieurs lignes. Il est coutume d'utiliser de telles chaines pour la documentation ; on les appelle *docstrings*.

```
def est_pair(n):
    """est_pair(n: int) -> bool"""
    ans = (n % 2 == 0)
    return ans
```

Cette signature est présente uniquement à titre de documentation et n'est pas lue par Python. Par exemple, rien n'empêche la fonction

```
def suivant(n):
    """suivant(n: int) -> int"""
    return n + 1
```

d'être appelée avec un nombre flottant. Cependant, les fonctions que nous écrirons ne s'utiliseront qu'avec les types suggérés par la docstring.

Une fonction ne peut renvoyer qu'une seule valeur, ce qui est parfois problématique. Supposons par exemple que l'on souhaite écrire une fonction qui nous donne l'heure en fonction du nombre de secondes qui se sont écoulées depuis minuit. On doit pour cela renvoyer trois entiers : h, m et s. Pour cela, on choisit de renvoyer un tuple formé de 3 entiers. Une affectation simultanée permet de déconstruire le tuple lors de l'appel d'une telle fonction.

```
def heure(n):
    """heure(n: int) -> tuple[int, int, int]"""
    s = n % 60
    n = n // 60
    m = n % 60
    h = n // 60
    return h, m, s
```

```
In [3]: h, m, s = heure(42000)
```

1.2 Les fonctions comme valeurs

Les fonctions sont des valeurs comme les autres. Leur type est `function`.

```
def next(n):
    """next(n: int) -> int"""
    return n + 1
```

```
In [1]: type(next)
Out[1]: function
```

En particulier, il est possible de passer une fonction en argument d'une autre fonction. On peut par exemple définir la fonction suivante qui calcule une approximation de la dérivée d'une fonction.

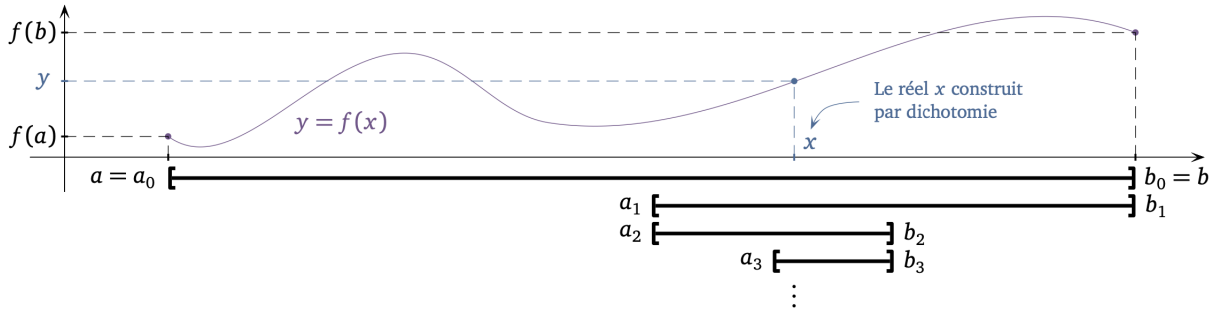
```
def f(x):
    """f(x: float) -> float"""
    return x**2 - 2

def derive(f, x, eps):
    """derive(f: function, x: float, eps: float) -> float"""
    return (f(x + eps) - f(x)) / eps
```

```
In [2]: derive(f, 1.0, 1.0e-6)
Out[2]: 2.0000009999243673
```

Exercice 1

⇨ Écrire une fonction `dichotomie(f: function, a: float, b: float, y: float, eps: float) -> tuple[float, flo` qui prend en argument une fonction continue f telle que $f(a)$ et $f(b)$ sont de part et d'autre de y et renvoyant un couple (u, v) tel que $0 \leqslant v - u \leqslant \varepsilon$ et tel qu'il existe $x \in [u, v]$ tel que $f(x) = y$.



On utilisera pour cela l'algorithme de *dichotomie* qui consiste à définir deux suites (a_n) et (b_n) en commençant par poser $a_0 \coloneqq a$, $b_0 \coloneqq b$. Pour tout $n \in \mathbb{N}$, une fois a_n et b_n définis, on définit a_{n+1} et b_{n+1} de la manière suivante : on commence par calculer $f(c_n)$ où $c_n \coloneqq (a_n + b_n)/2$ et
— si $f(a_n) - y$ et $f(c_n) - y$ sont de signes distincts, on pose $a_{n+1} \coloneqq a_n$ et $b_{n+1} \coloneqq c_n$.
— sinon, on pose $a_{n+1} \coloneqq c_n$ et $b_{n+1} \coloneqq b_n$.
Alors les suites (a_n) et (b_n) sont telles que pour tout $n \in \mathbb{N}$, il existe un $x \in [a_n, b_n]$ tel que $f(x) = y$. De plus $b_n - a_n$ tend vers 0 lorsque n tend vers $+\infty$.

1.3 Assertion, test unitaire

Afin de déceler au plus tôt la présence de bugs dans nos programmes, il est important d'écrire des jeux de tests unitaires. Ces tests exécutent une fonction pure avec des arguments dont la valeur de retour est connue ; ils vérifient ainsi leur conformité. Par exemple, pour vérifier que la fonction

```
def factorielle(n):
    """factorielle(n: int) -> int"""
    fac = 1
    for k in range(1, n + 1):
        fac = fac * k
    return fac
```

nous renvoie bien la factorielle de n pour tout $n \geqslant 0$, on pourra vérifier que `factorielle(0)` renvoie bien 1, `factorielle(2)` renvoie bien 2 et `factorielle(5)` renvoie bien 120. Pour cela, on écrira

```
assert factorielle(0) == 1
assert factorielle(2) == 2
assert factorielle(5) == 120
```

en dessous de la définition de notre fonction. De manière générale, le mot clé `assert` est suivi d'une expression qui doit s'évaluer en un booléen : si ce booléen s'évalue en `True`, l'instruction `assert` ne fait rien ; sinon, elle lève une exception, ce qui se traduit par une erreur.

1.4 Sortie anticipée

Une fonction peut posséder plusieurs `return` dans son corps. Dans ce cas, le premier `return` rencontré fait sortir de la fonction. Il est courant d'utiliser cette propriété afin d'exprimer simplement certaines boucles. Par exemple, la fonction suivante teste si une chaine de caractères possède un e.

```
1 def possede_un_e(s):
2     """possede_un_e(s: str) -> bool"""
3     for k in range(len(s)):
4         if s[k] == 'e':
5             return True
6     return False
```

Dans cette fonction, la boucle a pour tâche de vérifier les caractères un à un. Si la chaines de caractères s possède un « e », il existe un indice $k \in [\![0, n-1]\!]$ pour lequel `s[k]` est égal à `'e'` ; la fonction va exécuter la ligne 5, renvoyer `True` et sortir immédiatement. Si la chaine de caractères ne contient pas la lettre « e », la condition ligne 4 n'est jamais satisfaite ; on sort alors de la boucle et la dernière instruction renvoie `False`.

```
In [1]: s = "Puis, à la fin, nous saisirons pourquoi tout fut bati à partir d'un
    carcan si dur, d'un canon si tyrannisant. Tout naquit d'un souhait fou, d'un
    souhait nul : assouvir jusqu'au bout la fascination du cri vain, sortir du
    parcours rassurant du mot trop subit, trop confiant, trop commun, n'offrir au
    signifiant qu'un goulot, qu'un boyau, qu'un chas, si aminci, si fin, si aigu
    qu'on y voit aussitot sa justification."

In [2]: possede_un_e(s)
Out[2]: False
```

Ce style de progammation, dans lequel il existe plusieurs points de sortie d'une fonction, peut devenir plus difficile à comprendre. Il est donc découragé d'en abuser, car c'est une source de bugs. Cependant, dans un exemple comme celui-ci, son usage est pleinement justifié.

Remarquons qu'une sortie anticipée casse le caractère inconditionnel d'une boucle `for`, puisqu'on ne sait plus avant de rentrer dans la boucle quel va être le nombre d'itérations. Cependant, elle garde son caractère borné et est toujours assurée, soit de terminer totalement, soit d'être interrompue par un `return`.

Exercice 2

⇨ 1. Écrire une fonction `est_sous_mot_position(sm: str, s: str, k: int) -> bool` prenant deux chaines de caractères `sm` et `m` et renvoyant `True` si `sm` est un sous-mot de `s` commençant à la position k, et `False` sinon. Par exemple `est_sous_mot_position("th", "python", 2)` doit renvoyer `True`. On supposera que le mot `m` est assez grand pour contenir le sous-mot `sm` à partir de la position k.

2. En déduire une fonction `est_sous_mot(sm: str, s: str) -> bool` renvoyant `True` si `sm` est un sous-mot de `m` et `False` sinon.

Notons que l'instruction `break` permet de sortir d'une boucle `for/while` (la plus intérieure s'il y en a plusieurs imbriquées), sans sortir de la fonction. Son utilisation peut se justifier dans quelques rares cas, par exemple si l'on souhaite écrire la fonction `est_sous_mot(sm: str, s: str) -> bool` de l'exercice précédent sans utiliser une fonction auxiliaire :

```
1  def est_sous_mot(sm, s):
2      """est_sous_mot(sm: str, s: str) -> bool"""
3      m = len(sm)
4      n = len(s)
5      for k in range(n - m + 1):
6          found = True
7          for i in range(m):
8              if sm[i] != s[k + i]:
9                  found = False
10                 break
11         if found:
12             return True
13     return False
```

```
In [3]: est_sous_mot("thon", "python")
Out[3]: True
```

On l'utilisera cependant avec parcimonie, car il rend la compréhension d'un algorithme plus délicate. Il sera en général beaucoup plus simple d'utiliser une fonction auxiliaire utilisant une sortie anticipée.

2 Variable locale et globale

2.1 Variable locale

Il est possible de créer des variables à l'intérieur d'une fonction. Ces variables sont *locales* à la fonction et sont détruites une fois sorti de cette dernière.

```
1 def puissance_quatre(n):
2     """puissance_quatre(n: int) -> int"""
3     c = n * n
4     return c * c
```

```
In [1]: puissance_quatre(2)
Out[1]: 16

In [2]: c
NameError: name 'c' is not defined
```

Si la variable c est définie avant l'appel de la fonction, sa valeur est masquée lors de l'appel et on la retrouve une fois sorti de la fonction :

```
In [3]: c = 5

In [4]: puissance_quatre(2)
Out[4]: 16

In [5]: c
Out[5]: 5
```

Pour comprendre ce phénomène, il est important de comprendre la notion de masquage des variables : une fois dans la fonction, ligne 4, juste avant d'exécuter l'instruction `return`, l'état du système est le suivant :

```
# sous-état local fonction  {c: 4, n: 2}
# sous-état global          {c: 5}
```

À ce moment précis, l'état du système est la superposition de deux sous-états : le sous-état du dessus a été créé lors de l'appel de notre fonction et celui du dessous correspond au sous-état au moment de l'appel. Cette superposition de sous-états est rendue possible par *la pile d'appels*. La variable à laquelle on accède est par défaut celle qui se situe dans le sous-état *actif*, celui qui est le plus haut sur la pile. Par exemple, une fois à l'intérieur de notre fonction, la variable globale c contenant 5 est masquée par la variable locale de même nom, contenant 4 : c'est cette dernière à laquelle on accède. Une fois l'appel terminé, le sous-état local à l'appel est supprimé et on retrouve notre état initial.

```
# sous-état global          {c: 5}
```

Ces sous-états sont agencés comme une pile d'assiettes : lorsqu'on appelle une fonction, une nouvelle « assiette » est empilée au sommet de la pile ; lorsque cet appel se termine, cette assiette est supprimée.

Ce mécanisme permet à la fonction `puissance_quatre` de n'avoir aucun effet sur l'état au niveau de l'appel. Notons d'ailleurs que la variable n est locale à la fonction : elle est initialisée avec l'argument effectif passé lors de son appel. Comme cette variable est locale, on peut la modifier sans craindre d'effet de bord.

```
1 def puissance_quatre(n):
2     """puissance_quatre(n: int) -> int"""
3     n = n * n
4     return n * n
```

```
In [6]: n = 2

In [7]: puissance_quatre(n)
Out[7]: 16

In [8]: n
Out[8]: 2
```

2.2 Variable globale

On appelle *variable globale* toute variable définie dans le niveau le plus bas de la pile. Les *variables locales* sont celles définies dans les niveaux supérieurs. Si elles ne sont pas masquées, il est toujours possible d'accéder en lecture aux variables globales.

```
1 b = 1
2
3 def f(a):
4     """f(a: int) -> int"""
5     return a + b
```

```
In [1]: f(2)
Out[1]: 3
```

Lorsque l'on est dans la fonction f pour calculer `f(2)`, ligne 5, juste avant le `return`, l'état du système est donné par

```
# sous-état local fonction  {a: 2}
# sous-état global                {b: 1}
```

et l'accès à la variable globale b est possible.

Cependant, par défaut, il n'est pas possible de changer la valeur d'une variable qui n'est pas locale. Afin de pouvoir modifier une telle variable, il convient de la déclarer à l'intérieur de la fonction comme *globale* à l'aide du mot clé `global`.

```
1 compteur = 0
2
3 def carre_mission_impossible(n):
4     """carre_mission_impossible(n: int) -> int"""
5     global compteur
6     compteur = compteur + 1
7     if compteur <= 2:
8         return n * n
9     else:
10        return 0
```

```
In [2]: carre_mission_impossible(2)
Out[2]: 4

In [3]: carre_mission_impossible(2)
Out[3]: 4

In [4]: carre_mission_impossible(2)
Out[4]: 0
```

Cette fonction, si vous l'utilisez, s'autodétruira après 2 appels. Ce genre d'effet est très difficilement compréhensible pour son utilisateur. On dit qu'elle est *impure* car elle a un effet de bord. Comme elle ne renvoie pas `None`, ce comportement est surprenant. C'est pourquoi, la modification de variables globales est un style de programmation à proscrire.

2.3 Composition de fonctions

Une fonction peut elle-même appeler une autre fonction. On peut ainsi définir une fonction calculant les coefficients binomiaux à l'aide d'une fonction calculant la factorielle d'un entier.

```
1 def factorielle(n):
2     """factorielle(n: int) -> int"""
3     fac = 1
4     for k in range(1, n + 1):
5         fac = fac * k
6     return fac
7
8 def binome(k, n):
9     """binome(k: int, n: int) -> int"""
10    return factorielle(n) // (factorielle(k) * factorielle(n - k))
```

Lors de l'exécution, la composition de fonctions fait apparaitre une succession de sous-états sur plusieurs niveaux. Voyons cela sur un exemple simple :

```
1  def puissance_deux(n):
2      """puissance_deux(n: int) -> int"""
3      u = n * n
4      return u
5
6  def puissance_quatre(n):
7      """puissance_quatre(n: int) -> int"""
8      u = puissance_deux(n)
9      v = puissance_deux(u)
10     return v
```

```
In [1]: n = 2

In [2]: puissance_quatre(n)
Out[2]: 16
```

Lors du calcul de `puissance_quatre(n)`, ligne 9, on appelle `puissance_deux(u)` et à l'intérieur de cette fonction, ligne 4, juste avant le `return u`, le système est dans l'état suivant :

```
# sous-état local puissance_deux    {n: 4, u: 16}
# sous-état local puissance_quatre  {n: 2, u: 4}
# sous-état global                  {n: 2}
```

La superposition de ces sous-états forme la pile d'appels.

3 Programmation récursive

Une fonction *récursive* est une fonction qui s'appelle elle-même. Cette possibilité donne naissance à un style d'algorithmes qu'on appelle programmation récursive. L'idée essentielle derrière ce style est de *réduire* la résolution d'un problème à la résolution de problèmes similaires de tailles strictement inférieures. Pour que ce principe fonctionne, il faut d'une part spécifier des problèmes de tailles élémentaires, qu'on appelle *cas de base*, et donner leurs solutions ; il faut s'assurer d'autre part que les réductions précédentes finissent toujours par rencontrer de tels cas.

3.1 Fonction récursive pure

Commençons par un classique de la programmation récursive : le calcul de $n!$.

— *réduction* : Si $n \geqslant 1$, alors $n! = n \times (n-1)!$.

— *cas de base* : Sinon $n = 0$ et $0! = 1$.

La traduction en Python est immédiate.

```
1  def factorielle(n):
2      """factorielle(n: int) -> int"""
3      if n >= 1:
4          return n * factorielle(n - 1)
5      else:
6          return 1
```

```
In [1]: factorielle(5)
Out[1]: 120
```

Sur le même principe, nous allons programmer la division euclidienne d'un entier $a \in \mathbb{N}$ par $b \in \mathbb{N}^*$ en n'utilisant que des comparaisons, des additions et des soustractions.

— *réduction* : Si $a \geqslant b$, on note q et r le quotient et le reste de la division euclidienne de $a - b$ par b. Alors

$$a - b = qb + r, \quad \text{donc} \quad a = (q+1)b + r.$$

On en déduit que $q+1$ et r sont le quotient et le reste de la division euclidienne de a par b.

— *cas de base* : Sinon $0 \leqslant a < b$ et le quotient de la division euclidienne de a par b est 0 et son reste est a.

On obtient ainsi la fonction :

```python
def division(a, b):
    """division(a: int, b:int) -> tuple[int, int]"""
    if a >= b:
        q, r = division(a - b, b)
        return (q + 1, r)
    else:
        return (0, a)
```

```
In [2]: division(23, 7)
Out[2]: (3, 2)
```

Exercice 3

⇨ Définir de manière récursive la fonction `puissance(x: int, n: int) -> int` calculant x^n pour $n \in \mathbb{N}$. On utilisera le fait que $x^0 = 1$ et que si $n \geqslant 1$, alors $x^n = x^{n-1}x$.

Tout comme une boucle `while` peut être infinie, une fonction récursive peut ne jamais terminer. Prenons l'exemple de la fonction

```python
def est_pair(n):
    """est_pair(n: int) -> bool"""
    if n == 0:
        return True
    elif n == 1:
        return False
    else:
        return est_pair(n - 2)
```

qui détermine si un entier n est pair ou non. Elle fonctionne parfaitement pour un entier $n \geqslant 0$, mais si on appelle `est_pair(-1)`, la fonction va appeler successivement `est_pair` avec les valeurs $-3, -5, -7$, etc. Elle ne terminera jamais et on obtiendra l'erreur

```
In [3]: est_pair(-1)
RecursionError: maximum recursion depth exceeded in comparison
```

Il faudra donc être attentif, lorsqu'on définit une fonction récursive, à ce que tous les cas se réduisent à un cas de base en un nombre fini d'appels.

L'algorithme d'*exponentiation rapide* permet de calculer efficacement x^n pour $n \in \mathbb{N}$. Cet algorithme se base sur les deux remarques suivantes :

- *réduction* : Si $n > 0$, pour calculer x^n, on effectue la division euclidienne de n par 2. Il existe donc $p \in \mathbb{N}$ et $r \in \{0, 1\}$ tel que $n = 2p + r$. On remarque ensuite que
  - si $r = 0$, c'est-à-dire si n est pair, on a $x^n = (x^p)^2$.
  - si $r = 1$, c'est-à-dire si n est impair, on a $x^n = x(x^p)^2$.
- *cas de base* : On a $x^0 = 1$.

Ces deux remarques conduisent à l'algorithme suivant :

```python
def expo_rapide(x, n):
    """expo_rapide(x: int, n: int) -> int"""
    if n == 0:
        return 1
    else:
        p = n // 2
        y = expo_rapide(x, p)
        if n % 2 == 0:
            return y * y
        else:
            return x * y * y
```

L'algorithme d'exponentiation rapide permet de calculer x^n de manière plus efficace que l'algorithme naïf. Le calcul de x^n se fait de manière naïve en $n-1$ multiplications. Cependant, une récurrence immédiate montre qu'avec l'algorithme d'exponentiation rapide, le calcul de x^n pour $n := 2^p$ nécessite seulement $2 + p$ multiplications. Ainsi, l'algorithme naïf a besoin de 1023 multiplications pour calculer x^{1024} alors que l'algorithme d'exponentiation rapide en a besoin seulement de 12, car $1024 = 2^{10}$.

Cet exemple nous permet de réaliser qu'une fonction récursive va le plus souvent créer une pile d'appels conséquente. Par exemple, lors de l'appel de `expo_rapide(3, 4)`, on va finir par appeler récursivement `expo_rapide(3, 0)`. Lors de cet appel, une fois à la ligne 4, juste avant le `return 1`, la pile d'appels est dans l'état suivant :

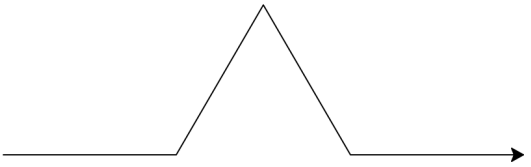
```
# sous-état local expo_rapide(3, 0)  {x: 3, n: 0}
# sous-état local expo_rapide(3, 1)  {x: 3, n: 1, p: 0}
# sous-état local expo_rapide(3, 2)  {x: 3, n: 2, p: 1}
# sous-état local expo_rapide(3, 4)  {x: 3, n: 4, p: 2}
# sous-état global                   {}
```

Lors de l'appel précédent `est_pair(-1)` qui ne terminait pas, l'erreur renvoyée était d'ailleurs liée à cette pile d'appels qui était devenue trop grande. On parle de *débordement de la pile d'appels*, ou de *stackoverflow* en anglais.

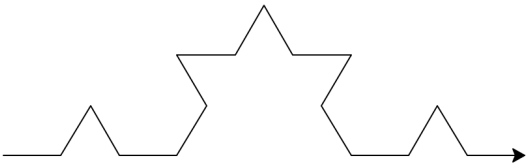
3.2 Fonction récursive impérative

Nous allons continuer en programmant le flocon de Von Koch. La génération 0 de ce flocon est un segment de longueur a,

la première génération est la figure suivante, le « triangle » central étant équilatéral,



puis la seconde génération est :



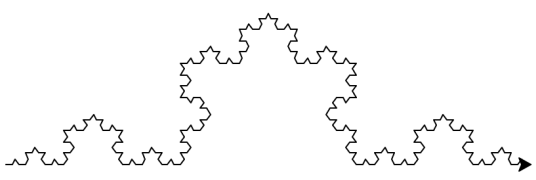
Le but est d'écrire un programme dessinant la n-ième génération du flocon de Von Koch de longueur a. On remarque évidemment le caractère récursif de sa définition.

- *réduction* : Si $n \geqslant 1$, on dessine un flocon de Von Koch de longueur $a/3$ et de génération $n-1$, puis on tourne à gauche de 60 degrés, on dessine de nouveau le même flocon de Von Koch, on tourne à droite de 120 degrés, on dessine à nouveau un flocon de Von Koch, on tourne à gauche de 60 degrés et on dessine un dernier flocon.
- *cas de base* : Si $n = 0$, on trace un segment de longueur a.

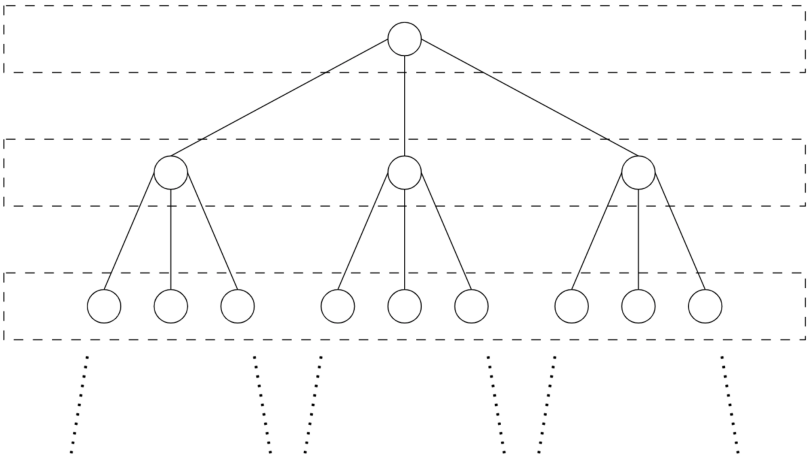
On obtient ainsi le programme suivant :

```
1  import turtle as lg
2
3  def koch(a, n):
4      if n == 0:
5          lg.forward(a)
6      else:
7          koch(a / 3, n - 1)
8          lg.left(60)
9          koch(a / 3, n - 1)
10         lg.right(120)
11         koch(a / 3, n - 1)
12         lg.left(60)
13         koch(a / 3, n - 1)
```

Le tracé de la 4^e génération nous donne



Ce programme récursif est l'occasion de présenter l'arbre d'appels d'une fonction récursive. Au sommet, nous avons la racine qui représente l'appel initial à notre fonction. Cette fonction va elle-même s'appeler plusieurs fois et donc générer des appels représentés dans l'arbre avec une profondeur de 1.

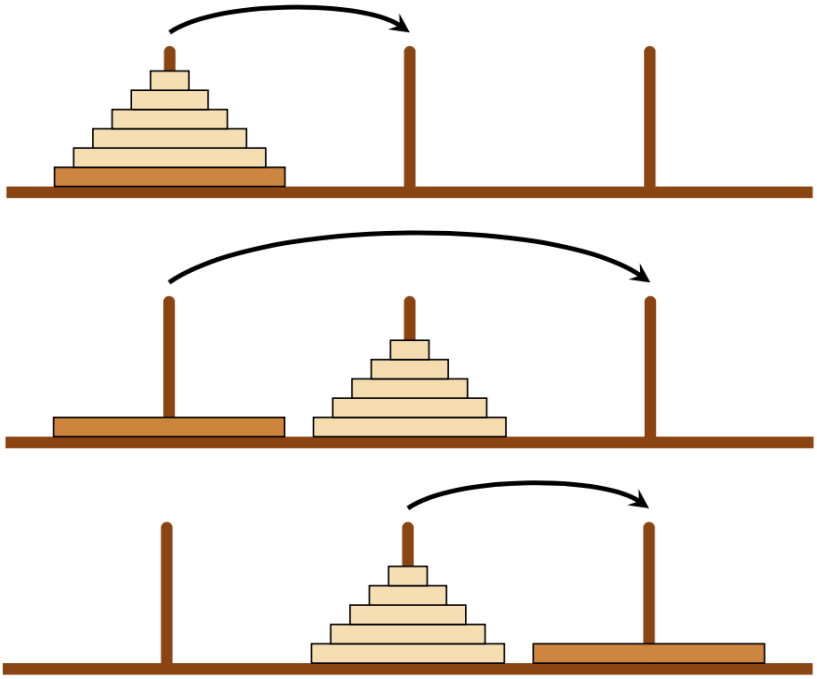


Ces fonctions s'appellent elles-mêmes plusieurs fois, et ainsi de suite.

Nous allons maintenant nous atteler à la résolution d'un grand classique des jeux mathématiques : le jeu des tours de Hanoï, inventé par le mathématicien Edouard Lucas. Ce jeu est constitué de trois tiges sur lesquelles sont enfilés n disques de diamètres différents. Au début du jeu, ces disques sont tous positionnés sur la première tige, du plus grand qui est en dessous, au plus petit. L'objectif est de déplacer tous ces disques sur la troisième tige en respectant les règles suivantes :

— On ne peut déplacer qu'un disque à la fois.
— On ne peut pas poser un disque sur un disque de diamètre inférieur.

Raisonnons par récurrence : pour pouvoir déplacer le dernier disque, on déplace les $n-1$ disques qui le couvrent sur la tige centrale. Une fois ces déplacements effectués, nous pouvons déplacer le dernier disque sur la troisième tige. Il reste alors à déplacer les $n-1$ autres disques vers la troisième tige.



Tout est dit : pour pouvoir déplacer n disques de la tige 1 vers la tige 3, il suffit de savoir déplacer $n-1$ disques de la tige 1 vers la tige 2 puis de la tige 2 vers la tige 3. Autrement dit, il suffit de généraliser le problème de manière à décrire le déplacement de n disques de la tige i à la tige k en utilisant la tige j comme pivot. Ceci conduit à la fonction suivante :

```python
def hanoi(n, i, j, k):
    """hanoi(n: int, i: int, j: int, k: int) -> NoneType"""
    if n == 0:
        return None
    else:
        hanoi(n - 1, i, k, j)
        print("Déplacer le disque au sommet de la tige", i, "vers la tige", k)
        hanoi(n - 1, j, i, k)
```

```
In [1]: hanoi(3, 1, 2, 3)
Déplacer le disque au sommet de la tige 1 vers la tige 3
Déplacer le disque au sommet de la tige 1 vers la tige 2
Déplacer le disque au sommet de la tige 3 vers la tige 2
Déplacer le disque au sommet de la tige 1 vers la tige 3
Déplacer le disque au sommet de la tige 2 vers la tige 1
Déplacer le disque au sommet de la tige 2 vers la tige 3
Déplacer le disque au sommet de la tige 1 vers la tige 3
```

Exercice 4

⇨ Déterminer le nombre de mouvements utilisés par l'algorithme précédent pour résoudre le problème des tours de Hanoï à n disques.

3.3 Fonctions mutuellement récursives

Il est possible de définir des fonctions *mutuellement récursives* : l'exemple le plus simple serait

$$\begin{cases} f(0) \coloneqq 1 \\ g(0) \coloneqq 0 \\ \forall n \in \mathbb{N}, \quad f(n+1) \coloneqq g(n) \\ \forall n \in \mathbb{N}, \quad g(n+1) \coloneqq f(n) \end{cases}$$

Voici le code Python mettant en oeuvre ces fonctions :

```python
def f(n):
    """f(n: int) -> int"""
    if n == 0:
        return 1
    else:
        return g(n - 1)

def g(n):
    """g(n: int) -> int"""
    if n == 0:
        return 0
    else:
        return f(n - 1)
```

Exercice 5

⇨ De quelles fonctions élémentaires f et g sont-elles des implémentations tordues et inefficaces ?

Nous n'aurons pas très souvent besoin de définir des fonctions mutuellement récursives, mais c'est quand même nécessaire de temps en temps.