Récursivité

I. Rappel : fonctions

Comme toujours, vous êtes invité à taper tous les exemples dans la console Python.

Anatomie d'une fonction

Définition La définition d'une fonction se fait à l'aide du *mot-clé* **def** suivi du nom de la fonction puis, *entre parenthèses*, les arguments et enfin deux points :

```
def ma_fonction(argument_1, argument_2):
    ...
```

Ensuite, le *corps* de la fonction est indenté, comme à chaque fois que l'on a deux points en fin de ligne.

Par exemple,

```
>>> def somme(n):
...     s = 0
...     for i in range(n + 1):
...         s = s + i
...     return s
```

Variables et arguments La fonction précédente a un **argument** `n` et utilise deux **variables locales** `s` et `i`.

Fonction et appel de fonction Ici, `somme` est une fonction.

```
>>> somme
<function somme at 0x101bc4670>
```

Pour faire un **appel** de fonction, il faut des **parenthèses** (même si la fonction a 0 argument).

```
>>> somme(10)
55
```

Lors d'un appel de fonction, schématiquement, on affecte les arguments puis on exécute le corps de la fonction. Ainsi, dans un premier temps, l'exécution de `print(somme(10))` peut se voir comme :

```
n = 10 # affectation des arguments
# exécution de la fonction
s = 0
for i in range(n + 1):
    s = s + i
# return s
print(s) # utilisation du résultat
```

Caractère local des arguments et des variables locales Les variables locales et les arguments (qui sont une sorte spéciale de variables locales) sont **locales**, ce qui signifie qu'elles sont définies uniquement durant l'exécution et qu'elles n'écrasent pas une variable de même nom existant auparavant.

Dans l'exemple suivant, `n` est un entier, `s` est défini comme une chaîne de caractères (alors que le `s` de la fonction est un entier) et `i` n'est pas défini.

```
>>> n = 20
>>> s = "Poulet"
>>> i
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'i' is not defined
>>> somme(10) # on exécute la fonction
55
>>> n # la valeur de n n'a pas changé
20
>>> s # la valeur de s n'a pas changé
'Poulet'
>>> i # i n'est toujours pas défini
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'i' is not defined
```

Pour finir, en voici un exemple avec deux appels de fonction imbriqués.

```
>>> def f(n):
...     i = n
...     g(n + 1)
...     print("Dans f, on a n =", n, "et i =", i)
...
>>> def g(n):
...     i = n
...     print("Dans g, on a n =", n, "et i =", i)
...
>>> n = 20
>>> f(10)
Dans g, on a n = 11 et i = 11
Dans f, on a n = 10 et i = 10
>>> n # n est toujours égal à 20
20
>>> i # i n'est toujours pas défini au niveau global
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'i' is not defined
```

II. Fonctions récursives

Nous allons étudier maintenant une nouvelle manière d'écriture de programmes et de résolution de problèmes, basé sur l'idée que parfois, pour résoudre un problème de taille n, on peut se baser sur des solutions de sous-problèmes plus petits (de taille $n-1$ par exemple, mais nous verrons d'autres possibilités plus tard).

1. Présentation

La fonction factorielle peut se programmer simplement en Python de la façon suivante :

```python
def fact(n):
    f = 1
    for i in range(1, n + 1):
        f = f * i
    return f
```

Cela correspond à l'égalité

$$n! = \prod_{k=1}^{n} k$$

Pourtant, on peut aussi définir la factorielle par récurrence :

$$0! = 1 \qquad \forall\, n \geq 1,\ n! = n \times (n-1)!$$

où pour $n \geq 1$, on définit $n!$ en fonction de $(n-1)!$. Une façon de respecter la structure de cette définition est la suivante :

```
def fact_rec(n):
    if n == 0:
        # 0! = 1
        return 1
    else:
        # n! = n × (n−1)! sinon
        return n * fact_rec(n - 1)
```

Dans ce cas, la fonction `fact_rec` s'appelle elle-même. On dit qu'elle est *récursive*.

Définition (Récursivité) Une fonction est *récursive* si son exécution peut impliquer un ou des appels à cette même fonction (directement, ou par le biais d'autres fonctions).

On peut remarquer que dans la fonction récursive `fact_rec`, l'appel récursif est inclus dans un test. C'est nécessaire pour qu'à un moment, la fonction arrête de faire de nouveaux appels récursifs et puisse d'arrêter. On peut faire apparaître cela dans l'exemple précédent.

2. Écriture de fonctions récursives

Pour pouvoir écrire convenablement des fonctions récursives, commençons par quelques règles :

Les impératifs de la récursivité

1. Il doit y avoir des **cas de base** qui ne comportent pas d'appels récursifs.
2. Tous les appels récursifs doivent se faire avec des arguments qui sont « plus proches » d'un cas de base.

Pour l'instant, on aura toujours un entier naturel n dans les arguments, et tous les appels récursifs se font avec des valeurs de n strictement plus petites.[1]

De plus, si vous avez du mal à comprendre comment marche une fonction récursive (ce qui est normal), vous pouvez vous aider du **génie de la récursivité** : pour traiter un cas récursif au rang n, vous pouvez supposer que le *génie de la récursivité* sait traîter tous les cas jusqu'au rang $n-1$ et que vous pouvez faire appel au génie autant que vous voulez.[2]

[1] En Option Informatique, on formalisera et généralisera cela avec la notion d'*ordre bien fondé*.

[2] Pour une étude plus approfondie du fonctionnement des fonctions récursives, il faudra, pour les MPSI, prendre l'Option Informatique.

Exemple – Retour sur la factorielle
Commençons par voir que les deux impératifs sont bien vérifiés (si l'argument n est positif) :

```
def fact_rec(n):
    if n == 0:
        # Cas de base
        return 1
    else:
        # Cas récursif. L'appel récursif se fait avec un
        # plus petit indice, on se rapproche du cas de base.
        return n * fact_rec(n - 1)
```

Concernant l'écriture de la fonction, pour le cas récursif, quand $n \geq 1$, le *génie de la récursivité* permet d'avoir la valeur de `fact_rec(`$n-1$`)` (qui, si tout se passe bien, vaut $(n-1)!$.). La seule question à se poser est donc :

À partir de n et de $(n-1)!$, comment obtenir la valeur de $n!$?

Exercice 1 On veut écrire une fonction récursive `triangle1` qui trace les superbes figures suivantes :

```
>>> triangle1(3)
***
**
*
>>> triangle1(5)
*****
****
***
**
*
>>> triangle1(6)
******
*****
****
***
**
*
```

1. Quel est le cas de base ?
2. Pour le cas récursif, pensez au génie de la récursion. Sinon on sait afficher un triangle de taille $n-1$, que doit-on faire pour afficher un triangle de taille n ?

Pour vous aider à écrire la fonction, l'expression suivante peut vous aider :

```
>>> "*" * 10
'**********'
```

3. Maintenant, vous pouvez écrire la fonction. N'oubliez pas de la tester.

Exercice 2 Modifier cette fonction pour obtenir une fonction `triangle2` qui affiche un triangle pointe vers le bas :

```
>>> triangle2(5)
*
**
***
****
*****
```

Idéalement, cette fonction contiendra une ligne du type `print("*" * n)`, exactement comme pour `triangle1`.

Toutes ces fonctions auraient pû facilement s'écrire à l'aide d'une boucle, c'est un aspect sur lequel nous reviendrons. Notons en attendant que la récursivité vous fournit une nouvelle façon d'écrire des fonctions.

Exercice 3 Testez la fonction suivante :

```
def pas_trop_triangle(n):
    if n > 0:
        pas_trop_triangle(n - 1)
        print("*" * n)
        pas_trop_triangle(n - 1)
```

Pourriez-vous en écrire simplement une version itérative ?

Comme on vient de le voir, rien n'interdit d'avoir *plusieurs* appels récursifs, tant que les règles précédentes sont respectées.

Exercice 4 La suite de Fibonacci est la suite $(F_n)_{n\in\mathbf{N}}$ définie par :

$$F_0 = 0, \quad F_1 = 1, \qquad \forall\, n \geq 2,\, F_n = F_{n-1} + F_{n-2}.$$

Écrire une fonction `fibo` qui, étant donné un entier naturel n, renvoie F_n. On identifiera bien les cas de bases et les cas récursifs, en s'assurant que les règles précédentes sont bien respectées.

Que se passe-t-il lorsque les deux impératifs ne sont pas respectés ? Il y a le risque d'un exécution qui dure infiniment. En pratique, cependant, il y a des mécanismes qui font que l'exécution s'arrête avec une erreur.

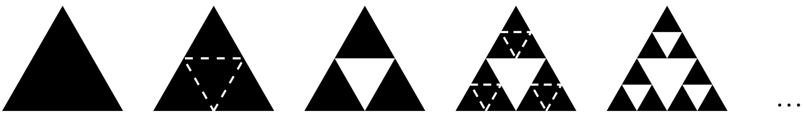
Question 5 (pour ceux qui ont le cœur bien accroché)[3] Écrire une fonction *récursive* la plus simple possible qui ne respecte clairement pas les deux impératifs, et tester cette fonction.

III. Quelques exemples

1. Triangle de Sierpinski

Le triangle de Sierpinski[4] est une fractale qui se construit de la façon suivante : on commence par un triangle équilatéral plein que l'on divise en 4 sous-triangles équilatéraux. On ôte le sous-triangle central et on répète le processus avec les trois sous-triangles restants[5].

La figure obtenue « à la fin » (après une infinité d'étapes) est le triangle de Sierpinski.



Vous avez dans le fichier `recursivite.py` un ensemble de fonction à compléter pour tracer les premières étapes de la construction du triangle de Sierpinski. En particulier, la fonction `sierpinski` prends en entrée deux arguments, la taille `r` du demi-coté du triangle initial et le nombre `r` d'étapes récursives à effectuer.

Cette fonction trace le triangle noir initial, puis appelle la fonction récursive `sierp`. Celle-ci prends quatre arguments : les coordonnées du sommet du bas du triangle blanc à tracer, la mesure de son côté `r` et le nombre `n` d'appels récursifs restant, et trace un triangle blanc puis effectue des appels récursifs à elle-même pour tracer les triangles suivants.

Question 6 Essayer cette fonction en exécutant `sierpinski(200, 5)`.

La flèche qui apparaît et trace la figure est appelée la *tortue*. Vous pourrez explorer un peu plus son utilisation dans la partie sur la courbe de Koch. En attendant, travaillons au tracé du triangle de Sierpinski.

[3] Ne vous inquiétez pas, si vous n'osez pas écrire une telle fonction, vous risquez fortement de le faire de toutes façons par erreur. Et l'ordinateur s'arrêtera tout seul. Au pire, si c'est trop long, appuyez sur la combinaison de touches `contrôle c`.

[4] Ou plutôt Sierpiǹski pour être précis, du nom de son créateur Wracław Sierpiǹski.

[5] Notons qu'il s'agit d'une méthode décrite de manière récursive.

Question 7 On voit que l'on a pas tous les appels récursifs nécessaires, puisque l'étape récursive ne concerne que le sous-triangle du haut. Modifier la fonction `sierp` pour traîter les deux sous-triangles manquant.

Question 8 Essayer la fonction obtenue. Notons que si l'exécution prends trop de temps, si par exemple le nombre d'appels récursifs demandé est trop élevé, on peut interrompre l'exécution en utilisant la combinaison de touches `controle c`.

2. PGCD, Euclide et Bézout

Le calcul du plus grand commun diviseur de deux entiers a et b se calcule facilement à l'aide de l'algorithme d'Euclide (connu, donc, depuis l'antiquité) :

$$\text{pgcd}(a,b) = \begin{cases} a & \text{si } b = 0 \\ \text{pgcd}(b,r) & \text{si } b \neq 0 \end{cases}$$

où, pour le deuxième cas, r désigne le reste de la division euclidienne de a par b.

On rappelle qu'en Python, le quotient de la division euclidienne s'écrit `//` et le reste s'obtient à l'aide de `%` :

```
>>> 2048 // 7
292
>>> 2048 % 7
4
```

C'est clairement un algorithme récursif, et il est historiquement assez important[6].

Question 9 Implémenter cet algorithme en Python. Dans la définition ci-dessous, vous prendrez bien soin de repérer le cas de base et le cas conduisant à un appel récursif.

À nouveau, il est possible d'écrire cela avec une boucle **`while`**. Nous allons maintenant considérér l'*algorithme d'Euclide étendu* relié au théorème de Bézout :

Théorème 1 - Bézout

Étant donné deux entiers $a, b \in \mathbf{Z}$, il existe deux entiers $u, v \in \mathbf{Z}$ tels que

$$au + bv = \text{pgcd}(a,b).$$

Nous allons donc modifier l'algorithme d'Euclide pour calculer, en plus du *pgcd*, un couple d'entiers (u, v) qui convient (notons qu'il n'y a pas unicité).

[6] "[The Euclidean algorithm] is the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day." – Donald Knuth.

Question 10 Pour des entiers a et $b \neq 0$, on note q et r respectivement le quotient et le reste de la division euclidienne de a par b.

1. Donner une expression reliant a, b, q et r.
2. On suppose que, grâce au génie de la récursion, on connaît $\operatorname{pgcd}(b,r)$ ainsi que deux entiers s et t tels que

$$sb + tr = \operatorname{pgcd}(b,r).$$

En déduire la valeur de $\operatorname{pgcd}(a,b)$ (facile si vous avez un peu suivi), puis (plus difficile) déterminer deux entiers u et v tels que

$$ua + bv = \operatorname{pgcd}(a,b).$$

Question 11 Écrire une fonction `bezout` qui prends en argument deux entiers a et b et renvoit le triplet d'entiers (x,u,v) où $x = \operatorname{pgcd}(a,b)$ et u et v sont tels que

$$ua + vb = \operatorname{pgcd}(a,b).$$

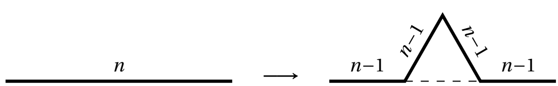
En particulier, il est conseillé de réfléchir convenablement aux cas de bases et aux cas récursifs.

Contrairement à la fonction `euclide` initiale, une écriture itérative de l'algorithme d'Euclide étendu est nettement plus difficile (bien que faisable, nous n'avez qu'à y réfléchir). On voit ici un avantage à la programmation récursive.

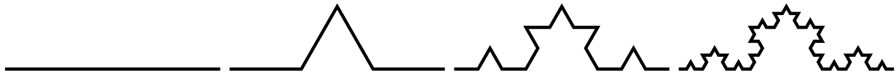
3. Une autre figure fractale, la courbe de Koch

Le courbe de Koch de longueur ℓ d'ordre $n \in \mathbf{N}$ se définit par récurrence de la façon suivante :

- ⋆ À l'ordre 0, il s'agit simplement de tracer un segment de longueur ell.
- ⋆ À un ordre $n \geq 1$, on trace une courbe de Koch de longueur $\ell/3$ d'ordre $n-1$, on tourne vers la gauche de 60°, puis on trace une courbe de Koch de longueur $\ell/3$ d'ordre $n-1$, on tourne vers la droite de 120°, on trace une courbe de Koch de longueur $\ell/3$ d'ordre $n-1$ et enfin on tourne vers la gauche de 60° et on trace une courbe de Koch de longueur $\ell/3$ d'ordre $n-1$. La figure suivante illustre cela plus clairement :



Voici des courbes de Koch d'ordres allant de 0 à 4 :



Question 12 Écrire une fonction `koch(n, l)` qui trace à l'aide de la tortue la courbe de Koch d'ordre `n` de taille `l`.

On pourra pour cela examiner le code de la fonction `sierp_triangleblanc` pour découvrir les fonctions de base d'utilisation de la tortue, ou se référer à la documentation en ligne du module . La page Beginner's Guide to Python Turtle de RealPython contient aussi beaucoup d'informations utilisables.