Python – Récursivité

Fractales

Afin de dessiner les fractales, nous allons utiliser le module `turtle`. Voici les premières lignes à mettre au début de votre fichier `fractale.py` :

```
import turtle as tt      # pour utiliser la tortue
tt.setup(640, 480)       # taille de la fenetre
tt.speed(0)              # pour aller encore plus vite
```

Toutes les commandes concernant la tortue devront commencer par le préfixe “`tt.`". Les principales commandes de base de la tortue sont :

Commande Python	Remarques
`forward(N)`, `fd(N)`	Avancer de `N` pixels
`backward(N)`, `back(N)`, `bk(N)`	Reculer de `N` pixels
`right(N)`, `rt(N)`	Tourner à droite de `N` degrés
`left(N)`, `lt(N)`	Tourner à gauche de `N` degrés
`up()`	Lever le crayon
`down()`	Baisser le crayon
`setpos(x, y)`	Aller en `(x, y)`
`seth(direction)`	S’orienter vers `direction`
`reset()`	Tout réinitialiser

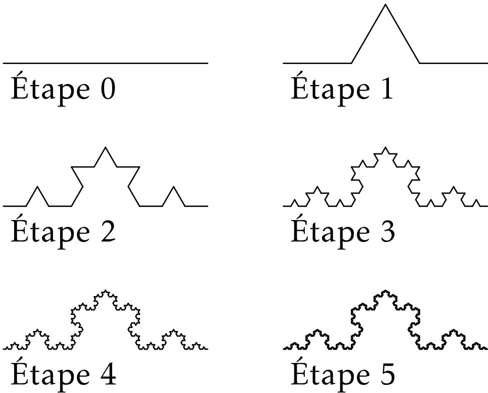
Les coordonnées de la tortue correspondent à celle d’un repère orthonormé dont l’origine est le centre de la fenêtre. Les directions sont exprimées en degrés, 0 correspondant à la droite, 90 le haut, et ainsi de suite. Pour en savoir plus sur ces commandes et sur les autres, vous pouvez aller consulter la page https://docs.python.org/3/library/turtle.html.
La fonction `initialiser(x, y, direction)` fait aller la tortue au point de coordonnées `(x, y)` et l’oriente vers la direction `direction`.

```
def initialiser(x=0,y=0,direction=0):
    tt.up() # lever le stylo
    tt.setposition(x,y) # aller au centre
    tt.down() # baisser le stylo
    tt.setheading(direction) # s'orienter a droite par défaut
```

```
>>> initialiser() # va en (0, 0) et s'oriente vers la droite
>>> initialiser(-100, -100) # va en (-100, -100) et s'oriente vers la droite
>>> initialiser(20, 20, 90) # va en (20, 20) et s'oriente vers le haut
>>> initialiser(direction=90) # va en (0, 0) et s'oriente vers le haut
```

Les valeurs par défaut données aux paramètres permettent de n’en donner que quelques uns lors de l’appel.
Cette fonction permettra de positionner votre tortue à une position permettant d’afficher plus facilement les figures.

Exercice 1 : Compléter la fonction `koch(etape, longueur)` qui trace la courbe de Koch à l'étape `etape` et dont la longueur à l'étape 0 aurait été de `longueur`. À chaque fois qu'on augmente de 1 l'étape, il faut diviser par 3 la longueur.

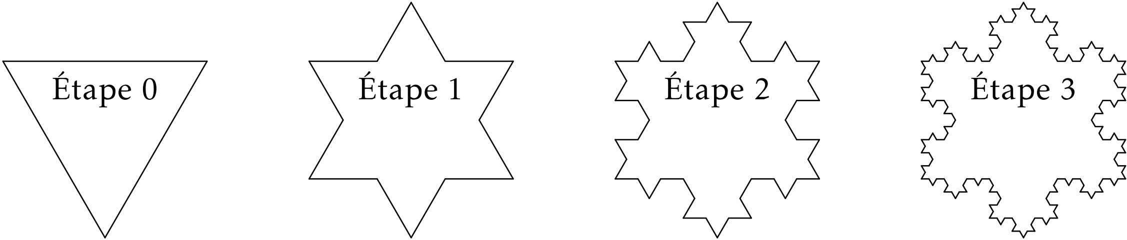



```
def koch(etape, l):
    if ...:
        tt.forward(l)
    else:
        koch(..., ...)
        tt.left(...)
        ... # plusieurs lignes à rajouter
```

Afin d'accélérer l'affichage si le nombre d'étapes un peu grand (évitez tout de même de dépasser 10), vous pouvez utiliser la fonction `tt.tracer(n)` qui permet de ne mettre à jour l'affichage que toutes les `n` opérations. En prenant une valeur comme 1000, vous accélérez grandement l'affichage. Si la fonction se termine avant d'avoir atteint le prochain seuil d'affichage, la figure ne sera pas totalement affichée. Il faut alors utiliser `tt.update()` pour afficher le reste.

```
>>> tt.tracer(1000)
>>> koch(8, 300)
>>> tt.update()
```

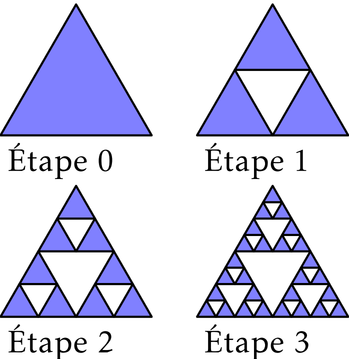
Exercice 2 : Écrire une fonction `flocon(etape, longueur)`, non récursive, qui trace le flocon de Koch à l'étape `etape`. La longueur `longueur` correspond au côté du triangle de l'étape 0. Vous pouvez utiliser la fonction `koch`.



La fonction suivante permet de tracer un triangle coloré en bleu.

```
def triangle(longueur):
    tt.fillcolor('blue') # couleur pour colorier
    tt.begin_fill() # pour colorier la suite
    for i in range(3):
        tt.forward(longueur)
        tt.left(120)
    tt.end_fill() # on finit le coloriage
```

Exercice 3 : Écrire une fonction `sierpinski(etape, longueur)` qui trace le triangle de Sierpinski correspondant, selon le modèle ci-contre. Vous pouvez utiliser la fonction `triangle`.



Fonctions de la feuille de cours

Dans feuille de cours, nous avons vu qu'il était possible de faire des versions fonctionnelles et d'autres itératives (avec des boucles) pour la même définition. Créez un nouveau fichier `recursivite.py`. Pour la plupart des fonctions, il vous sera demandé de faire une version récursive et une autre itérative. Vous pouvez utiliser deux noms différents, comme `fonct_r` et `fonct_i`.

Exercice 4 : Écrire les versions récursives et itératives de `somme(n)` qui calcule la comme de `n` premiers entiers.

```
>>> somme_i(10)
55
>>> somme_r(15)
120
```

Exercice 5 :

1) Comparez le comportement des deux fonctions avec `n` = 1 000 et `n` = 1 000 000.
2) Comparez le comportement des deux fonctions avec `n` = −15.

Il faudra donc faire attention à ne pas utiliser de trop grandes valeurs avec les fonctions récursives.
Il faut également faire attention avec les valeurs qui sont en dehors de la définition de la fonction, comme les valeurs négatives. Vous pourriez rajouter un **`assert`** au début de la fonction récursive, mais ce test serait fait à chaque appel récursif, alors que seul la première valeur de `n` a besoin d'être testé.

Exercice 6 : Écrire une fonction `somme_r2(n)` qui vérifie la positivité de `n` avec un assert avant d'appeler `somme_r(n)`.

Il était également possible de corriger `somme_r` en changeant le cas de base par $n \leq 0$.

Exercice 7 : Compléter les versions récursives et itératives de `fibo(n)` qui renvoie la valeur du `n`-ième terme de la suite de Fibonacci.

```python
def fibo_r(n): # version récursive
    if n == 0:
        return ...
    elif ...:
        return ...
    else:
        return fibo_r(...) + ...
```

```python
def fibo_i(n): # version itérative
    u = ...  #u0
    v = ...  #u1
    for i in range(n):
        w = ... # on calcule le terme suivant
        u = ...  # on avance d'un cran
        v = ...  # on avance d'un cran
    return u
```

```
>>> fibo_r(5)
8
>>> fibo_r(10)
89
```

Exercice 8 : Comparez le comportement des deux fonctions avec `n` = 20, `n` = 30 et `n` = 40.

Lorsqu'il y a plusieurs appels récursifs, il faut d'autant plus attention à ne pas prendre de trop grandes valeurs, comme nous avons pu le voir avec les fractales.

Exercice 9 : Compléter la fonction récursive `f91(n)` qui renvoie le résultat de $f_{91}(n)$.

```
def f91(n):
    #print(n) # pour voir les valeurs de n
    if ...:
        return ...
    else:
        return ...
```

Puisque le résultat est toujours le même, ce qui est intéressant, ce n'est pas le résultat mais plutôt les différentes étapes.

Exercice 10 : Compléter la fonction récursive `f91_p(n, k=1)` qui affiche toutes les valeurs intermédiaires. La valeur `k` correspond au nombre de fois où apparaît f_{91} dans l'expression. Ainsi, `f91_p(70,3)` correspond à $f_{91}(f_{91}(f_{91}(70)))$. Vous pouvez afficher l'expression au tout début de la fonction et ensuite faire l'appel récursif. Il va falloir déterminer quel est le cas de base. De plus, puisque la fonction ne renvoie rien, il n'est pas nécessaire de mettre de **return**, comme c'était déjà le cas pour les fractales.

```
def f91_p(n, k=1):
    print("f91("*k + str(n) + ")"*k) # on affiche l'expression
    if n <= 100: # on rajoute un f91 à l'expression
        f91_p(..., ...)
    elif k == ...: # on va enlever le dernier f91
        print(n-10)   # on affiche la valeur finale
    else: # on enlève un f91
        f91_p(..., ...)
```

```
>>> f91_p(99)
f91(99)
f91(f91(110))
f91(100)
f91(f91(111))
f91(101)
91
```

Exercice 11 : Compléter la fonction récursive `puiss1(x, n)` qui renvoie x^n en utilisant la méthode “naturelle”.

```
def puiss1(x, n):
    if n == 0:
        return ...
    else:
        return x * puiss1(x, ...)
```

```
>>> puiss1(2, 2)
4
>>> puiss1(2, 3)
8
>>> puiss1(2, 8)
256
>>> puiss1(5, 3)
125
```

Exercice 12 : Écrire la fonction récursive `puiss2(x, n)` qui renvoie x^n en utilisant l'exponentiation rapide.

```
def puiss2(x, n):
    if n == 0:
        return ...
    elif n == 1:
        return x
    elif n%2 == ...:
        return x * puiss2(..., ...)
    else:
        return puiss2(..., ...)
```

```
>>> puiss2(2, 2)
4
>>> puiss2(2, 3)
8
>>> puiss2(2, 8)
256
>>> puiss2(5, 3)
125
```

Exercice 13 : Comparez le comportement des deux fonctions pour calculer 2^{100} et 2^{10000}.

Exercice 14 : Écrire les fonctions `a(n)` et `b(n)`.

```
def a(n):
    if ...:
        return 1
    else:
        return ...

def b(n):
    if ...:
        return 0
    else:
        return ...
```

```
>>> a(9)
6
>>> b(9)
6
```

Vous pouvez remarquer que même si une des deux fonctions est écrite avant l'autre, tout en l'appelant, cela ne pose pas de problème à Python. C'est parce que l'évaluation n'est faite que lors de l'appel de la fonction et qu'à ce moment là, les deux fonctions sont en mémoire.

La suite de Syracuse

La **suite de Syracuse** est définie de la manière suivante :

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

Comme nous l'avons vu l'année dernière, il est conjecturé qu'il existe toujours un rang n pour lequel $u_n = 1$, pour n'importe quel entier strictement positif u_0.

Exercice 15 : Compléter la fonction récursive `syracuse(u)` qui affiche tous les termes successifs obtenus pour la suite de Syracuse à partir de $u_0 =$ u et jusqu'à arriver à 1. Puisqu'il faut afficher les valeurs, votre fonction ne doit pas contenir de **return**.

```
def syracuse(u):
    print(u)
    if u > 1:
        if u%2 == 0:
            syracuse(...)
        else:
            syracuse(...)
```

```
>>> syracuse(5)
5
16
8
4
2
1
```

On appelle **durée de vol** de la suite de Syracuse la valeur du plus petit entier k tel que $u_k = 1$. Par exemple, la durée de vol si $u_0 = 5$ est de 5.

Exercice 16 : Compléter la fonction récursive `duree_vol(u)` qui renvoie la durée de vol de la suite de Syracuse avec $u_0 =$ u.

```
def duree_vol(u):
    if u == 1:
        return 0
    elif u%2 == 0:
        return 1 + duree_vol(...)
    else:
        return ...
```

```
>>> duree_vol(548)
92
```

L'**altitude maximum** de la suite de Syracuse est la valeur maximale obtenue avant d'atteindre 1.

Exercice 17 : Compléter la fonction récursive `altitude_max(u)` qui renvoie l'altitude maximum de la suite de Syracuse en partant de $u_0 =$ u.

```
def altitude_max(u):
    if u == 1:
        return 1
    elif u%2 == 0:
        return max(u, altitude_max(...))
    else:
        return max(u, ...)
```

```
>>> altitude_max(674)
9232
```