Exercices - La récursivité

Dans chacun des exercices proposés, on demande un code python et généralement le contenu d'une pile d'exécution pour un exemple type. Le code python est à écrire dans un fichier nommé *recursif.py* qui sera à uploader sur *nsibranly.fr*. Les parties écrites demandées sont à compléter directement sur ce poly.

1- FONCTION QUI CALCULE LE FACTORIEL D'UN NOMBRE :

Le factoriel d'un nombre n est noté $n!$ et est égal au produit des nombres entiers strictement positifs inférieurs ou égaux à n : $n! = n \times (n-1) \times (n-2) \times \ldots\ldots \times 2 \times 1$.

Par exemple $4! = 4 \times 3 \times 2 \times 1 = 24$

La fonction *fact()* a comme paramètre un entier n . Elle retourne le nombre $n!$

1- Ecrire un code iteratif de *fact()*
2- Ecrire un code récursif de *fact()*
3- Détailler l'exécution de *fact(4)* en version récursive, en complétant les 2 tableaux ci-dessous :

Empilement dans la Pile d'exécution	*Dépilement* – Affichage dans la console

2- FONCTION QUI CALCULE LE n^{ieme} TERME D'UNE SUITE DE NOMBRES :

On définit la suite de nombre suivante : 12 ⇨ 14 ⇨ 18 ⇨ 26 ⇨ 42 ⇨ 74 ⇨ ….

Le premier de ces nombres est donc $u_1 = 12$. Le nombre suivant s'obtient en multipliant le précédent par 2 et en soustrayant 10. Ainsi pour le 2nd nombre de cette suite, on a $12 \times 2 - 10 = 14$ et donc $u_2 = 14$. Pour le 3ième, on a : $14 \times 2 - 10 = 18$.

La fonction *u()* a comme paramètre un entier n . Elle retourne le nombre u_n .

1- Ecrire un code iteratif de *u()*
2- Ecrire un code récursif de *u()*
3- Détailler l'exécution de *u(4)* en version récursive, en complétant les 2 tableaux ci-dessous :

Empilement dans la Pile d'exécution	*Dépilement* – Affichage dans la console

3- FONCTION QUI CONSTRUIT UN ARBRE AVEC LE MODULE TURTLE :

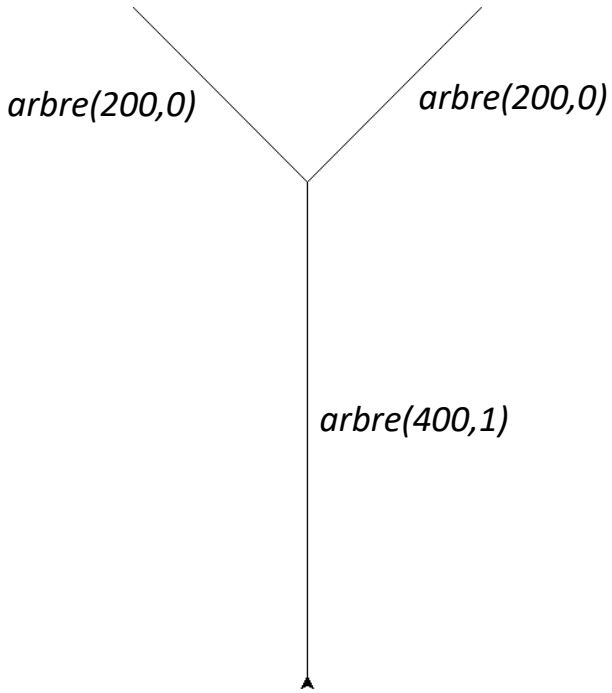
Soit le code récursif de la fonction *arbre()* donné ci-contre.

L'exécution de *arbre(400,1)* permet de tracer l'arbre donné ci-dessous.

On a inscrit à coté de chaque trait, l'appel qui a permis de tracer ce trait.

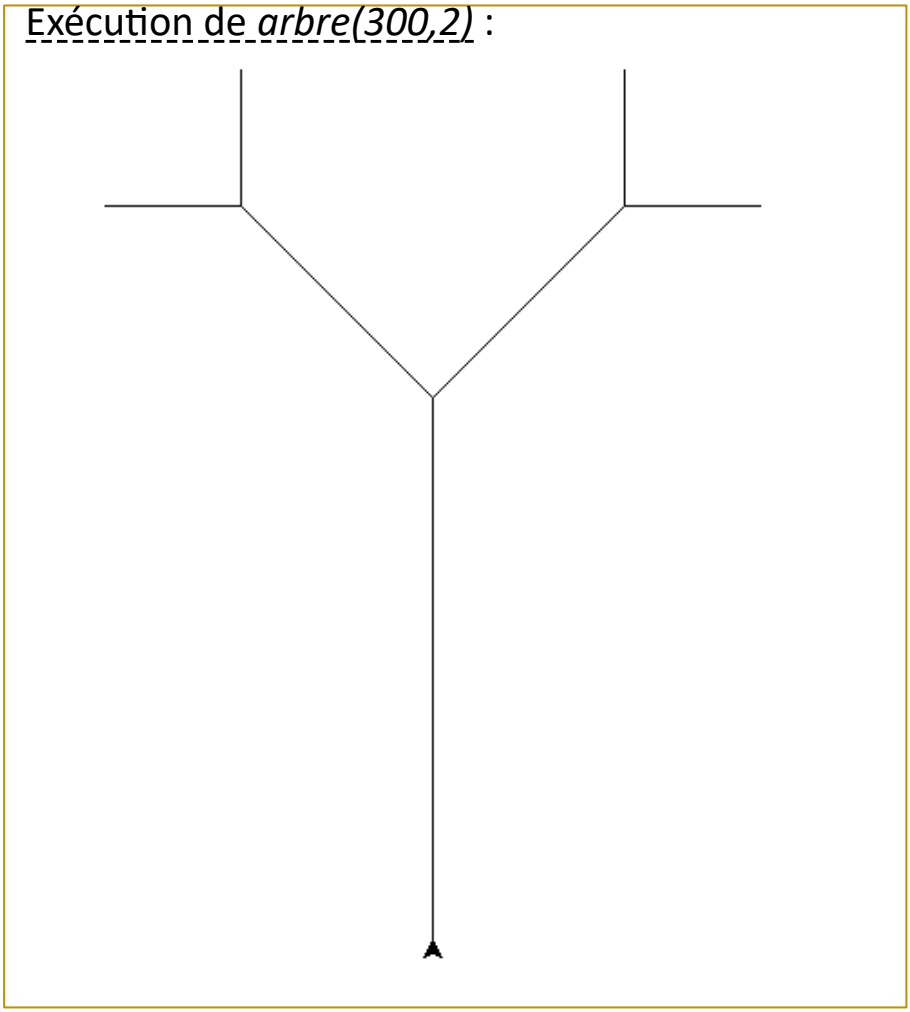
```
from turtle import *

def arbre(hauteur, profondeur) :
        if profondeur == 0 :
                forward(hauteur)
                backward(hauteur)
        else :
                forward(hauteur)
                right(angle)
                arbre(k*hauteur,profondeur-1)
                left(2*angle)
                arbre(k*hauteur,profondeur-1)
                right(angle)
                backward(hauteur)
# Main
angle = 45
k = 0.5
up()
left(90)
goto(0,-300)
down()
arbre(hauteur = 400, profondeur = 1 )
exitonclick() # pour garder ouverte la fenêtre
```

Exécution de *arbre(400,1)* :



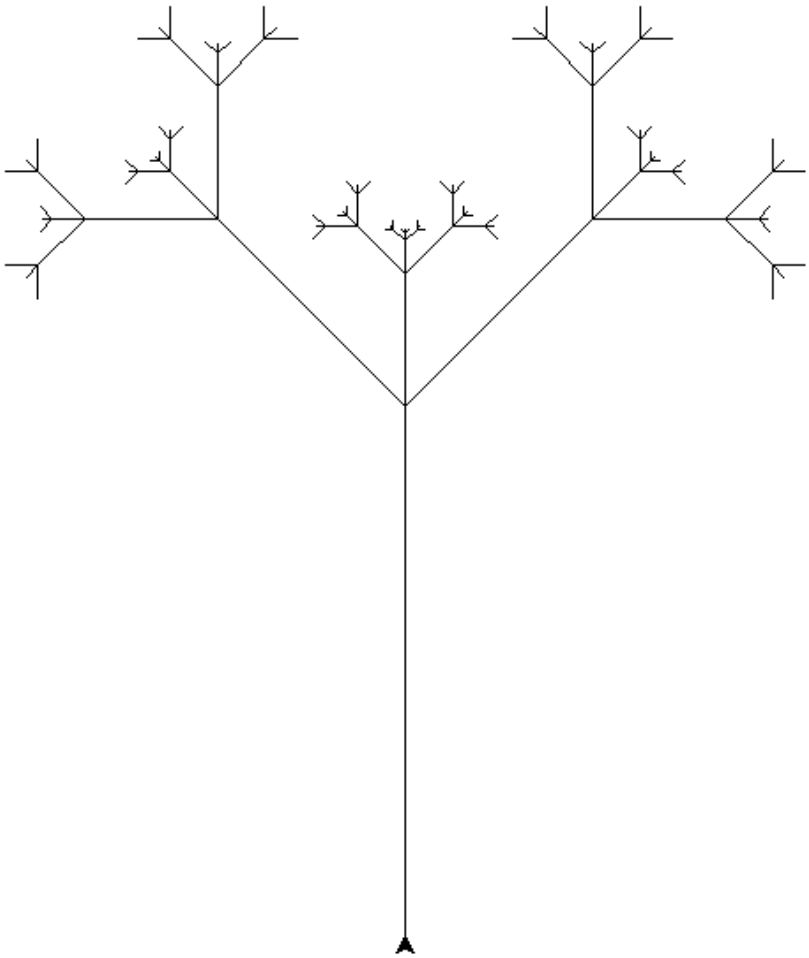
⇨ Copier-coller ce code dans votre fichier et lancer une exécution *arbre(300,2)*. Vous obtenez la figure ci-dessous. Comme précédemment, inscrire à coté de chaque trait, l'appel qui a permis de le tracer.

Exécution de *arbre(300,2)* :



⇨ A partir du script de la fonction arbre() , écrire le script d'une fonction arbreM() qui permet d'obtenir à l'exécution de arbreM(300,4), le tracé qui suit.

Exécution de *arbreM(300,4)* :



4- FONCTION QUI AJOUTE 10 A CHAQUE ELEMENT D'UNE LISTE :

La fonction *ajout()* donnée ci-contre a comme paramètre une liste ℓ de valeurs numériques. Cette fonction retourne une nouvelle liste contenant les éléments de ℓ augmentée de 10.

```
def ajout(l) :
    L = []
    for elt in l :
        elt = elt + 10
        L = L + [elt]
    return L
```

On donne en exemple l'exécution donnée ci-contre, à gauche.

```
>>> l=[1,6,5]

>>> ajout(l)
[11, 16, 15]

>>> print(l)
[1, 6, 5]
```

Pour ajouter un élément à une liste, on utilise ici non pas la méthode *append(),* mais le principe de concaténation de listes.

On donne ci-dessous une version **récursive** de ce même code :

```
def ajout(l) :
    if len(l) == 0 : return []
    else :
        elt = l[-1] + 10
        return ajout(l[:-1])+[elt]
```

On exécute les lignes suivantes dans la console avec cette version récursive de ajout() :

```
>>> l=[1,6,5]

>>> ajout(l)
[11, 16, 15]
```

1- Détailler l'exécution en complétant les 2 tableaux ci-dessous :

Empilement dans la Pile d'exécution	*Dépilement* – Affichage dans la console
ajout([1,6,5)] = ajout([1,6]) + [15]	

Pour bien comprendre le fonctionnement du code dans sa version récursive, on a légèrement modifié le code de ajout() pour y rajouter des affichages intermédiaires de résultats. Ce donne la version ci-dessous :

```
def ajout(l) :
    if len(l) == 0 : return []
    else :
        print(f"                                    ")
        elt = l[-1] + 10
        retour = ajout(l[:-1]) + [elt]
        print(f"                                    ")
        return retour
```

Si on réalise alors la même exécution qu'avant, on obtient :

```
>>> l=[1,6,5]

>>> ajout(l)
appel recursif ajout([1, 6])
appel recursif ajout([1])
appel recursif ajout([])
dépilage de ajout([]) + [11]
dépilage de ajout([1]) + [16]
dépilage de ajout([1, 6]) + [15]
[11, 16, 15]
```

2- Ecrire dans votre fichier *recursif.py* le script de la fonction ajout() en complétant correctement les 2 appels de *print()* afin d'obtenir le résultat donné ici :

5- FONCTION QUI MODIFIE A CHAQUE ELEMENT D'UNE LISTE :

La fonction *affine()* donnée ci-contre a comme paramètre une liste ℓ de valeurs numériques et deux nombres a et b. Cette fonction retourne une nouvelle liste contenant les éléments de ℓ multipliés par a et augmentée de b.

```
def affine(l,a,b) :
    L = []
    for elt in l :
        elt = a*elt + b
        L = L + [elt]
    return L
```

```
>>> l=[1,6,5]

>>> affine(l,10,5)
[15, 65, 55]
```

On donne en exemple l'exécution donnée ci-contre, à gauche.

⇨ Ecrire dans votre fichier *recursif.py* **une version récursive** de la fonction *affine()* .

6- FONCTION QUI COMPTE LE NOMBRE DE CARACTERES :

La fonction *lg()* donnée ci-contre a comme paramètre un string. Elle en retourne le nombre de caractères.

```
>>> lg("Bonne année")
11
```

On donne en exemple l'exécution donnée ci-contre, à gauche.

```
def lg(mot) :
    nb = 0
    for c in mot :
        nb = nb + 1
    return nb
```

⇨ Ecrire dans votre fichier *recursif.py* **une version récursive** de la fonction *lg()* .

7- FONCTION QUI MODIFIE LES CARACTERES D'UN STRING :

La fonction *casse()* donnée ci-contre a comme paramètre un string. Elle retourne le même string en ayant transformé les minuscules en majuscules et vice-versa. Le caractère « *espace* » n'est pas modifié. Ce script utilise un dictionnaire nommé dic, non mis en argument car défini dans le programme principal.

```
def casse(mot) :
    MOT = ""
    for c in mot :
        C = dic[c]
        MOT = MOT + C
    return MOT

# main
dic = {}
minuscules = "abcdefghijklmnopkrstuvwxyz "
majuscules = "ABCDEFGHIJKLMNOPQRSTUVWXYZ "
for i in range(27) :
    dic[minuscules[i]] = majuscules[i]
    dic[majuscules[i]] = minuscules[i]
```

On donne en exemple l'exécution donnée ci-dessous :

```
>>> casse("Bonne annee")
'bONNE ANNEE'
```

⇨ Ecrire dans votre fichier *recursif.py* **une version récursive** de la fonction *casse()* .

8- FONCTION QUI COMPTE LE NOMBRE D'OCCURRENCE D'UNE LISTE :

La fonction *nb()* donnée ci-contre a comme paramètre une liste ℓ et une valeur. Cette fonction retourne le nombre d'éléments de cette liste égaux à cette valeur.

```
def nb(l,val) :
    nb = 0
    for elt in l :
        if val == elt : nb = nb + 1
    return nb
```

On donne en exemple l'exécution donnée ci-contre :

```
>>> l=[1,6,5]

>>> nb(l,6)
1
```

⇨ Ecrire dans votre fichier *recursif.py* **une version récursive** de la fonction *nb()* .

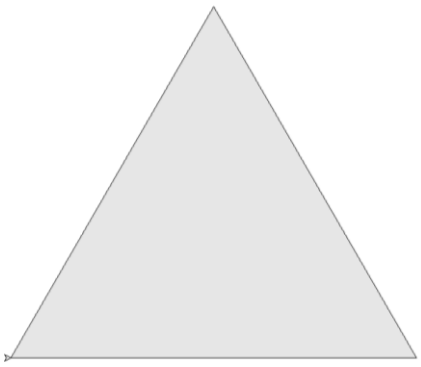
9- FONCTION QUI CONSTRUIT UN FRACTALE AVEC LE MODULE TURTLE :

Soit le code récursif de la fonction *triangle()* donné ci-contre.

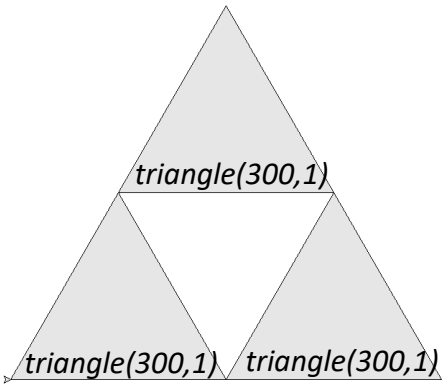
L'exécution de *triangle(600,1)* permet de tracer les figures données ci-dessous.

On a inscrit pour chacun des triangles grisés, l'appel qui a permis de le tracer.

```
from turtle import *

def triangle(cote, n) :
        if n > 0 :
                begin_fill()
                for _ in range(3) :
                        triangle(cote/2, n-1)
                        forward(cote)
                        left(120)
                end_fill()

# Main
fillcolor(.9,.9,.9)
speed(10)
up()
goto(-300,-200)
down()
triangle(600,1)
mainloop()
```

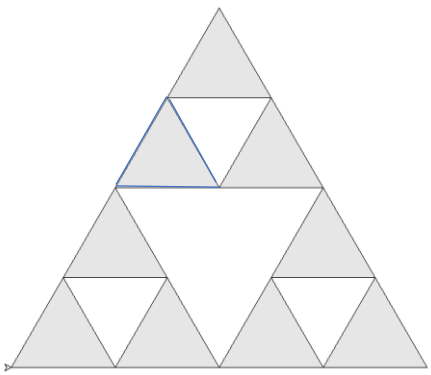


Exécution de *triangle(600,1)*



Exécution de *triangle(600,2)*

⇨ Copier-coller ce code dans votre fichier et lancer une exécution *triangle(600,3)*. Vous obtenez la figure ci-contre. Y colorier en rouge un des triangles construit avec l'appel triangle(300,2). Colorier en bleu un des triangles construit avec l'appel triangle(150,1).



⇨ A partir du script de la fonction *triangle()* , écrire le script d'une fonction *hexa()* qui permet d'obtenir à l'exécution de hexa(300,3), le tracé qui suit.

Exécution de *hexa(300,3)*