Séance 2 : Itérations et récursions

L1 – Université Côte d'Azur

Création du répertoire de travail

Avant de commencer ce TP : sur votre ordinateur, si ce n'a pas été déjà fait au TP 1, créez un répertoire `Python` dans lequel vous rangerez tous vos fichiers de l'UE. Dans votre répertoire `Python` créez un répertoire `TP2`. Durant cette séance, tous vos fichiers devront être sauvegardés dans ce repertoire `TP2`.

Les prochaines semaines, vous devrez créer les répertoires `TP3`, puis `TP4`. Ainsi, il sera facile de vous retrouver dans tous les fichiers.

Exercice 1 — Récursion versus boucle `while`

On appelle factorielle de n, que l'on note $n!$, le produit $n! = 1 \times \cdots \times n$.

1. Écrire *par récurrence* la fonction `fac_rec(n)` prenant un entier $n \geq 0$ et retournant la factorielle de n. Testez sur `fac_rec(5)` qui vaut 120.
2. Écrire en utilisant une boucle `while`, sous la forme d'une fonction `fac(n)`. Testez sur `fac(5)`.
3. Python est-il capable de calculer 3000! avec chacune de ces définitions ? Pour bien comprendre ce qui pose problème, observez l'exécution de `fact_rec(5)` et `fact(5)` en mode débuggage rapide (⇧+F5 puis en appuyant sur la touche F7).

Exercice 2 — Suite de Syracuse

On définit la suite de Syracuse de la manière suivante. On part d'un entier u_0 positif auquel on applique les règles de récurrence suivantes jusqu'à ce que l'on tombe sur 1.

$$u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{si } u_n \text{ est impair.} \end{cases}$$

Par exemple en partant de 12 on obtient la suite suivante :

$$12 \to 6 \to 3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

1. Écrire une fonction `suivant(u)` qui **renvoie** l'élément suivant une valeur `u`. Par exemple `suivant(12)` renvoie `6` et `suivant(3)` renvoie `10`. Écrire au moins quatre tests.
2. En utilisant la fonction `suivant(u)`, écrivez une fonction `syracuse(u)` avec une boucle `while` qui affiche les différentes valeurs de la suite.
3. En vous inspirant du programme précédant, écrivez une fonction `nombre_syracuse` qui calcule le nombre d'étapes nécessaires avant de terminer.
4. *(bonus : pour les plus rapides ou à faire chez soi)* Parmi tous les nombres inférieurs à 100, lequel nécessite-t-il le plus d'étapes pour terminer ?

Exercice 3 — Dessiner un tapis

Ajoutez à votre programme les définitions de fonctions ci-dessous.

```
def étoile():
    print('*',end='')

def dièse() :
    print('#',end='')

def nouvelle_ligne() :
    print()
```

Sans utiliser la fonction `print`, mais uniquement les fonctions `étoile`, `dièse`, et `nouvelle_ligne` ci-dessus, écrivez des fonctions `tapis_a(l,h)`, `tapis_b(l,h)`, `tapis_c(l,h)` et `tapis_d(l,h)` qui affichent des tapis de largeur `l` et de hauteur `h` avec les motifs ci-dessous.

a)
```
*********
*********
*********
*********
*********
*********
*********
*********
*********
```

b)
```
*#*#*#*#*
*#*#*#*#*
*#*#*#*#*
*#*#*#*#*
*#*#*#*#*
*#*#*#*#*
*#*#*#*#*
*#*#*#*#*
*#*#*#*#*
```

c)
```
*#*#*#*#*
#*#*#*#*#
*#*#*#*#*
#*#*#*#*#
*#*#*#*#*
#*#*#*#*#
*#*#*#*#*
#*#*#*#*#
*#*#*#*#*
```

d)
```
*#*#*#*#*
#*#*#*#*#
*********
*#*#*#*#*
#*#*#*#*#
*********
*#*#*#*#*
#*#*#*#*#
*********
```

On doit faire deux boucles `for`. La première pour les lignes, et la seconde (à l'intérieur de la première) pour les colonnes.

```
def tapis_a(largeur,hauteur):
    i=0
    while i<hauteur: # Pour chaque ligne
        j=0
        while j<largeur:  # Pour chaque colonnes
            dièse()
            j=j+1
        nouvelle_ligne()
        i=i+1
```

```
def tapis_b(largeur,hauteur):
    i=0
    while i<hauteur: # Pour chaque ligne
        j=0
        while j<largeur:  # Pour chaque colonnes
            if j%2==0: # Si j est pair
                étoile()
            else: # sinon j est forcèment impaire
                dièse()
            j=j+1
        nouvelle_ligne()
        i=i+1
```

Remarquez comme le code devient beaucoup plus simple et plus lisible grâce à l'emploi de fonctions auxiliaires. Ceci étant c'est un bonne exercice de tout écrire en utlisant à chaque fois deux boucles `while`, une pour les lignes et une pour les colonnes.

```
def ligne_alternée(n,symbole1,symbole2): #  pour afficher les lignes -> *#*#*#*
    j=0
    while j<n:
        if j%2==0:
            symbole1()
        else:
            symbole2()
        j=j+1
    nouvelle_ligne()

def tapis_b(largeur,hauteur):
    i=0
    while i<hauteur:
        ligne_alternée(largeur,étoile,dièse)
        i=i+1
```

```
def tapis_c(largeur,hauteur):
    i=0
    while i<hauteur:
        if i%2==0:
            ligne_alternée(largeur,étoile,dièse)
        else:
            ligne_alternée(largeur,dièse,étoile)
        i=i+1
```

```
def tapis_d(largeur,hauteur):
    i=0
    while i<hauteur:
        if i%3==0:
            ligne_alternée(largeur,étoile,dièse)
        elif i%3==1:
            ligne_alternée(largeur,dièse,étoile)
        else:
            ligne_alternée(largeur,étoile,étoile)
        i=i+1
```

Exercice 4 — Fermat, Grothendieck, et les nombres premiers

Le n-ième nombre de Fermat est $2^{2^n} + 1$. Fermat avait conjecturé que tous ces nombres étaient des nombres premiers. Votre mission : montrer à l'aide de Python que cette conjecture est fausse !

1. Écrivez une fonction `fermat(n)` qui renvoie le n-ième nombre de Fermat (avec $n \geq 0$). Par exemple, `fermat(3)` renvoie 257.

```
def fermat(n):
    return 2 ** (2 ** n) + 1
```

2. Écrivez une fonction `premier_facteur(n)` qui renvoie le plus petit nombre supérieur ou égal à 2 qui divise n. Par exemple, `premier_facteur(35)` renvoie 5 et `premier_facteur(31)` renvoie 31.

```
def premier_facteur(n):
    d = 2
    while n%d != 0 :
        d = d+1
    return d
```

3. Que pensez-vous de 57, appelé nombre premier de Grothendieck ? Testez de même la primalité de `fermat(3)` et

`fermat(4)` en utilisant la fonction `premier_facteur`. [1]

```
>>> print('Le premier facteur de',57,'est',premier_facteur(57))
Le premier facteur de 57 est 3
```

4. Écrivez un programme qui affiche le premier n tel que `fermat(n)` n'est pas premier. Vous afficherez aussi un diviseur premier de `fermat(n)`.

Une première méthode naïve :

```
def est_premier(n):
    return n == premier_facteur(n)

def dernière_question() :
    n = 0
    # Heureusement que Fermat s'est trompé, sinon notre boucle ne terminerai jamais !
    while est_premier(fermat(n)) :
        n = n + 1
    print('fermat(',n,')=',fermat(n),' n'est pas premier',sep='',end=' ')
    print('car il a pour diviseur ', premier_facteur(fermat(n)),sep='')
```

La solution ci-dessus n'est pas optimale, car on calcule 2 fois `fermat(n)` et on cherche 2 fois son diviseur pour le `n` qu'on affiche. On peut faire plus efficace en sauvant les calculs intermédiaires dans des variables comme ci-dessous

```
def dernière_question_v2() :
    n = 0
    f = fermat(n)
    p = premier_facteur(f)
    # Pour que le test du while ait un sens la première fois qu'on
    # rentre dans la boucle, il faut initialiser f et p avant.
    while f == p :
        n = n + 1
        f = fermat(n)
        p = premier_facteur(f)
    print('fermat(',n,')=',f,' n'est pas premier',sep='',end=' ')
    print('car il a pour diviseur ',p,sep='')
```

On peut faire encore plus efficace ! En effet, le calcul de `fermat(n)` peut se faire de manière incrémentale : on passe de `fermat(n)`$-1 = 2^{2^n}$ à `fermat(n+1)`$-1 = 2^{2^{n+1}}$ en élevant au carré. C'est la propriété qu'on utilise ci-dessous

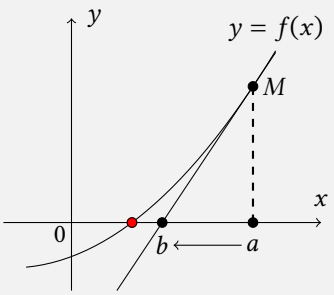
```
def dernière_question_v3() :
    n = 0
    f = fermat(0)
    p = premier_facteur(f)
    while f == p :
        n = n + 1
        f = (f-1)**2 + 1
        p = premier_facteur(f)
    print('fermat(',n,')=',f,' n'est pas premier',sep='',end=' ')
    print('car il a pour diviseur ',p,sep='')
```

Exercice 5 — Méthode de Newton

S'il existe des formules pour résoudre les équations du second degré, de nombreuses équations ne possèdent pas de telles formules. Il existe cependant des algorithmes pour trouver des valeurs approchées de ces solutions. Nous allons programmer nous même une méthode générale (la méthode de Newton) qui résout de manière approchée les équations de la forme $f(x) = 0$ (en tous cas elle trouvera une solution si elle existe et si f vérifie certaines propriétés).

1. Mais alors pourquoi parler de premier de Grothendieck ? Et qui était Grothendieck ? Lire https://images-archive.math.cnrs.fr/Alexandre-Grothendieck.html si cela vous intéresse.

Nous allons supposer que la fonction f est une fonction dérivable et à dérivée non nulle presque partout (de sorte que la tangente à la courbe existe avec une probabilité quasi-nulle d'être horizontale) pour appliquer la méthode des tangentes de Newton mentionnée en cours.



On suppose que l'approximation courante est un nombre strictement positif a. L'équation de la droite tangente à la courbe de f au point $M(a, f(a))$ s'écrit $y - f(a) = f'(a)(x - a)$. En $y = 0$, la droite coupe l'axe des abscisses en un point d'abcisse b donnée par la formule :

$$b = a - \frac{f(a)}{f'(a)}$$

Ainsi, pour $f(x) = x^2 - r$, on a $f'(x) = 2x$ et $b = a - \frac{a^2 - r}{2a} = \frac{1}{2}\left(a + \frac{r}{a}\right)$ comme vu en cours.

1. Écrivez une fonction `dériver(f,a,h)` qui renvoie $\frac{f(a+h)-f(a)}{h}$: pour un h petit, c'est une bonne approximation de $f'(a)$.

```
def dériver(f,a,h) :
    return (f(a+h) - f(a)) / h
```

2. Écrivez une fonction `résoudre(f,a0,h)` qui renvoie un nombre a tel que $|f(a)| < h$ en prenant pour approximation initiale a_0. *Indice* : on va calculer une suite d'approximation de a en utilisant la formule de récurrence :

$$a = a - \frac{f(a)}{f'(a)}$$

```
def résoudre(f,a,h) :
    while abs(f(a)) >= h :
        a = a - f(a) / dériver(f,a,h)
    return a
```

3. Testez votre fonction : calculez les cinq premières décimales de $\sqrt{2}$; vous devez trouver $1,41421\ldots$

```
def f(x) :
    return x*x - 2

print("racine(2) =",résoudre(f, 1, 0.000001))
```

Exercice 6 — Dessiner des tapis, bonus

Même consigne que pour l'exercice 3. Vous utiliserez lorsque ce sera nécessaire les deux fonctions suivantes.

```
def barre_1():
    print('/',end='')

def barre_2():
    print('\\',end='') # Il faut échapper le symbole de backslash
```

a)

```
*********
*********
*********
*********
*********
*********
*********
*********
*********
*********
```

b)

```
*#########
*#########
*##########
*##########
*##########
*##########
*##########
*##########
*##########
*#########
```

c)

```
####/\####
###/##\###
##/####\##
#/######\#
/########\
\########/
#\######/#
##\####/##
###\##/###
####\/####
```

```python
# Pour afficher n fois le même symbole
def ligne_simple(n,symbole):
    i=0
    while i<n:
        symbole()
        i=i+1

def tapis_A(largeur,hauteur):
    ligne_simple(largeur,étoile) # première ligne
    nouvelle_ligne()
    i=0
    while i<hauteur-2:
        étoile()
        ligne_simple(largeur-2,dièse)
        étoile()
        nouvelle_ligne()
        i=i+1
    ligne_simple(largeur,étoile) # dernière ligne
    nouvelle_ligne()

# Pour afficher ######*##### où l'* est à la position p
def ligne_position(n,p,symbole1,symbole2):
    i=0
    while i<n:
        if i==p:
            symbole2()
        else:
            symbole1()
        i=i+1


def tapis_B(largeur,hauteur):
    i=0
    while i<hauteur:
        ligne_position(largeur,largeur-i-1,dièse,étoile)
        nouvelle_ligne()
        i=i+1

def barre_1():
    print('/',end='')

def barre_2():
    print('\\',end='') # Il faut échapper le symbole de backslash

# Ne marche que si largeur est pair
def tapis_C(largeur,hauteur):
    i=0
    m=hauteur//2
    n=largeur//2
    while i<m:
        ligne_position(n,n-i-1,dièse,barre_1)
        ligne_position(n,i,dièse,barre_2)
        nouvelle_ligne()
        i=i+1
    i=0
    while i<m:
        ligne_position(n,i,dièse,barre_2)
        ligne_position(n,n-i-1,dièse,barre_1)
        nouvelle_ligne()
        i=i+1
```

Exercice 7 — Syracuse récursif

Refaire les questions 2 et 3 de l'exercice 2, mais en utilisant une récursion plutôt qu'une boucle `while`.

```python
def syracuse_récursive(un):
    if un == 1:
        print(1)
    else:
        print(un, " -> ", end='')
        syracuse_récursive(suivant(un))

def nombre_syracuse_récursive(un):
    if un == 1:
        return 0
    else:
        return 1+nombre_syracuse_récursive(suivant(un))
```