

Séance 2 : ITÉRATIONS ET RÉCURSIONS

L1 – Université Côte d'Azur

Exercice 1 — Factorielles et suites récurrentes

1. Écrivez une fonction `fact(n)` qui renvoie $n!$, la factorielle de n en utilisant une récurrence ;
2. Même question, mais en utilisant une boucle `while`.

On se donne la suite définie par la formule ci-contre :

$$\begin{cases} u_0 &= 234 \\ u_{n+1} &= -\frac{2}{3}u_n + \frac{1}{2} \end{cases}$$

3. Écrivez une fonction récursive `u(n)` calculant cette suite.
4. Écrivez une fonction `suite(n)` qui calcule cette même suite, mais de manière itérative.
5. On veut tester si ces deux fonctions calculent bien le même résultat pour chaque valeur de n . Écrire une fonction `tester(n)` qui renvoie `True` si les fonctions sont égales pour chaque valeur entière $0 \leq i \leq n$ et `False` sinon.

Exercice 2 — Partie entière de la racine carrée

Soit n un entier positif. La partie entière de la racine carrée de n , notée $\lfloor \sqrt{n} \rfloor$, est le plus grand entier k tel que $k^2 \leq n$. Écrivez une fonction `racine_entiere(n)` qui renvoie la partie entière de la racine carrée de n .

Exercice 3 — Correction de copies

La correction de copies est une activité relativement longue et fastidieuse. L'objectif de l'exercice est de créer une IA capable d'attribuer rapidement une note à un étudiant. On rappelle pour cela l'existence de la fonction `randint(a, b)` qui renvoie un nombre aléatoire entre a et b .

1. Écrivez une fonction `noter()` qui ne prend pas de paramètre et renvoie un entier aléatoire entre 0 et 20 compris.

```
1 def noter():
2     return randint(0,20)
```

2. Modifiez cette fonction pour autoriser les demi points. Les notes possibles seront donc : 0, 0.5, 1, 1.5, ..., 19.5 et 20.

```
1 def noter():
2     return randint(0,40)/2
```

3. Cette notation étant sévère, on obtient beaucoup de plaintes des étudiants. Écrivez une fonction `noter_gentil(n)` qui tire n notes aléatoirement et renvoie la plus grande.

```
1 def noter_gentil(n):
2     meilleur_note = noter()
3     i = 1
4     while i < n:
5         note = noter()
6         if note > meilleur_note:
7             meilleur_note = note
8         i = i+1
9     return meilleur_note
```

4. Écrivez une fonction `moyenne(k, n)` qui calcule la moyenne d'une promo de k étudiants en utilisant la fonction `noter_gentil` avec le paramètre n .

```

1 def moyenne(k,n):
2     somme = 0
3     i = 0
4     while i<k:
5         somme = somme + noter_gentil(n)
6         i = i+1
7     return somme/k

```

Pour l'exercice suivant, on se donne trois fonctions que vous n'avez pas à écrire : 1) étoile() qui affiche le symbole '*' 2) dièse() qui affiche le symbole '#' et 3) nouvelle_ligne() qui retourne à la ligne.

Par exemple

```

1 étoile()
2 dièse()
3 nouvelle_ligne()
4 dièse()
5 étoile()
6 nouvelle_ligne()

```

affichera :

```

1 *#
2 *#

```

Exercice 4 — Frise

1. Écrire une fonction frise(n) qui affiche une frise de longueur n alternant deux symboles # et *. On utilisera obligatoirement les trois fonctions décrites à la fin de la page précédente.

```

1 >>> frise(0)
2
3 >>> frise(1)
4 #
5 >>> frise(6)
6 #####*

```

```

1 def frise(n):
2     i=0
3     while i<n:
4         if i%2==0:
5             dièse()
6         else:
7             étoile()
8         i=i+1
9     nouvelle_ligne()

```

2. Peut-on tester une telle fonction avec assert?

On ne peut donc pas tester le résultat de cette fonction avec assert, car cette fonction n'a pas de résultat. Elle ne renvoie rien et se contente d'afficher

Exercice 5 — Logarithme entier

Soit n un entier positif. On appelle *logarithme entier* de n l'entier $le(n)$ correspondant au nombre de fois où il faut diviser n par deux avant d'atteindre 1 ou 0. Par exemple, $le(5) = 1 + le(2) = 1 + (1 + le(1)) = 2$.

1. Calculez à la main $le(3)$, $le(5)$, et $le(11)$ puis écrivez les tests correspondants avec assert.

```

1 assert le(0) == 0 # par définition
2 assert le(1) == 0 # par définition
3 assert le(3) == 1 # 3 -> 1
4 assert le(5) == 2 # 5 -> 2 -> 1
5 assert le(11) == 3 # 11 -> 5 -> 2 -> 1

```

2. Écrivez la fonction récursive `le(n)` qui renvoie le logarithme entier de n .

```

1 def le(n):
2     if n==0 or n==1:
3         return 0
4     else:
5         return 1+le(n//2)

```

Exercice 6 — Écriture binaire

1. Écrivez une fonction `affiche_calcul_binaire(n)` qui affiche le calcul de la représentation binaire de n , de sorte que l'on peut lire *verticalement* la représentation en binaire de n . Par exemple, pour $13 = (1101)_2$, on obtiendra dans la console :

```

1 >>> affiche_calcul_binaire(13)
2 1 car 13 = 2 × 6 + 1
3 0 car 6 = 2 × 3 + 0
4 1 car 3 = 2 × 1 + 1
5 1

```

```

1 def affiche_calcul_binaire(n) :
2     while n >= 2:
3         q = n//2
4         r = n%2
5         print(r , 'car' , n , '=' , '2 ×' , q , '+', r )
6         n = q
7     print(n)

```

2. Écrivez une fonction récursive `affiche_binaire(n)` qui affiche l'écriture binaire de n dans le sens de lecture **usuel**. Par exemple, on doit obtenir :

```

1 >>> affiche_binaire(13)
2 1101

```

```

1 def affiche_binaire_réursive(n) :
2     if n>1 :
3         affiche_binaire_réursive(n//2)
4     print(n%2 , end='')
5
6 # Pour le retour à la ligne final qui ne peut pas être fait
7 # au cours des appels rékursifs
8 def affiche_binaire(n):
9     affiche_binaire_réursive(n)
10    print()

```

3. Écrivez cette fonction en utilisant une boucle `while`.

Dans le code ci-dessous, on aurait pu construire la variable `binaire` comme une chaîne de caractère et non comme un entier. Pour cela, il aurait suffi d'écrire : `binaire = str(bit) + binaire`. Dans ce cas comment faut-il initialiser la variable `binaire` ?

```

1  # Pour être sûr de bien comprendre comment fonctionne un calcul,
2  # n'hésitez pas à utiliser Thonny et son exécution pas à pas.
3  def affiche_binaire_impérative(n) :
4      binaire = 0
5      i = 0
6      while n > 0 :
7          bit = n%2
8          binaire = binaire + 10**i * bit # Pourquoi ?
9          i = i+1
10         n = n//2
11     print(binaire)

```

Exercice 7 — Décomposition en facteurs premiers

1. Écrivez une fonction `affiche_facteurs(n)` qui affiche la liste des facteurs premiers avec multiplicité de n . Par exemple, `affiche_facteurs(1176)` affichera :

```

1  >>> affiche_facteurs(1176)
2  2**3  3**1  7**2

```

Astuce : on pourra définir des sous-fonctions pour rendre le programme plus lisible.

```

1  # On suppose n!=0 et d>1
2  # On regarde combien de fois d divise n
3  def multiplicité(n,d):
4      mult = 0
5      while n%d==0:
6          mult = mult+1
7          n = n//d
8      return mult
9
10
11 def affiche_facteurs(n):
12     if n==0 or n==1:
13         produit=str(n)
14     else:
15         produit=''
16         d=2
17         while n>1:
18             p = multiplicité(n,d)
19             if p!=0:
20                 produit = produit + str(d) + "**" + str(p) + " "
21                 n=n//(d**p) # Les parenthèses sont ici facultatives
22                 d=d+1
23     print(produit)

```

2. Écrire la même fonction mais en gérant proprement l'affichage des * :

```

1  >>> affiche_facteurs(1176)
2  2**3 * 3**1 * 7**2

```

```
1 def affiche_facteurs(n):
2     plus=' ' # Pour le premier facteur, on n'affiche pas le symbole +
3     if n==0 or n==1:
4         produit=str(n)
5     else:
6         produit=''
7         d=2
8         while n>1:
9             p = multiplicité(n,d)
10            if p!=0:
11                produit = produit + plus + str(d) + "*" + str(p)
12                plus=" + " # à partir de maintenant, on affichera les +
13                n=n//(d*p) # Les parenthèses sont ici facultatives
14                d=d+1
15     print(produit)
```