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_entière(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.

```
def noter():
    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.

```
def noter():
    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.

```
def noter_gentil(n):
    meilleur_note = noter()
    i = 1
    while i<n:
        note = noter()
        if note > meilleur_note:
            meilleur_note = note
        i = i+1
    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`.

```
def moyenne(k,n):
    somme = 0
    i = 0
    while i<k:
        somme = somme + noter_gentil(n)
        i = i+1
    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

```
étoile()
dièse()
nouvelle_ligne()
dièse()
étoile()
nouvelle_ligne()
```

affichera :

```
*#
#*
```

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.

```
>>> frise(0)

>>> frise(1)
#
>>> frise(6)
#*#*#*
```

```
def frise(n):
    i=0
    while i<n:
        if i%2==0:
            dièse()
        else:
            étoile()
        i=i+1
    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`.

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

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

```
def le(n):
    if n==0 or n==1:
        return 0
    else:
        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 :

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

```
def affiche_calcul_binaire(n) :
    while n >= 2:
        q = n//2
        r = n%2
        print(r , 'car' , n , '=' , '2 ×' , q ,'+', r )
        n = q
    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 :

```
>>> affiche_binaire(13)
1101
```

```
def affiche_binaire_récursive(n) :
    if n>1 :
        affiche_binaire_récursive(n//2)
    print(n%2 , end='')

# Pour le retour à la ligne final qui ne peut pas être fait
# au cours des appels récursifs
def affiche_binaire(n):
    affiche_binaire_récursive(n)
    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 suffit d'écrire : `binaire = str(bit) + binaire`. Dans ce cas comment faut-il initialiser la variable `binaire` ?

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

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

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

```python
# On suppose n!=0 et d>1
# On regarde combien de fois d divise n
def multiplicité(n,d):
    mult = 0
    while n%d==0:
        mult = mult+1
        n = n//d
    return mult


def affiche_facteurs(n):
    if n==0 or n==1:
        produit=str(n)
    else:
        produit=''
        d=2
        while n>1:
            p = multiplicité(n,d)
            if p!=0:
                produit = produit  + str(d) + "**" + str(p) + " "
            n=n//(d**p) # Les parenthèses sont ici facultatives
            d=d+1
    print(produit)
```

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

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

```python
def affiche_facteurs(n):
    plus='' # Pour le premier facteur, on n'affiche pas le symbole +
    if n==0 or n==1:
        produit=str(n)
    else:
        produit=''
        d=2
        while n>1:
            p = multiplicité(n,d)
            if p!=0:
                produit = produit + plus + str(d) + "**" + str(p)
                plus=" + " # à partir de maintenant, on affichera les +
            n=n//(d**p) # Les parenthèses sont ici facultatives
            d=d+1
    print(produit)
```