Séance 2 : Boucles while et récursions

L1 – Université Côte d'Azur

Exercice 1 — Factorielles (⋆)

Écrivez une fonction `fact(n)` qui renvoie $n!$, la factorielle de n :

1. en utilisant une récurrence ;
2. en utilisant une boucle `while`.

Exercice 2 — Affichage de factorielles (⋆)

1. En utilisant la fonction `fact(n)` de l'exercice précédant, écrivez une fonction `affiche_facts(n)` qui affiche sur n lignes les entiers 1!, 2!, 3!, ..., $n!$.
2. Combien de multiplications sont effectuées par votre code lorsque vous exécutez `affiche_facts(n)` ?
3. Proposez une solution `affiche_facts_efficace(n)` utilisant moins de multiplications.

Exercice 3 — 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, $\texttt{le}(5) = 1 + \texttt{le}(2) = 1 + (1 + \texttt{le}(1)) = 2$.

1. Calculez à la main `le(3)`, `le(5)`, et `le(11)` puis écrivez les tests correspondants avec `assert`.
2. Écrivez la fonction récursive `le(n)` qui renvoie le logarithme entier de `n`.

Exercice 4 — 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`.

Complément sur la fonction print

Par défaut, la fonction `print` affiche un caractère ␣ (espace) entre chacun de ses arguments et termine en allant à la ligne à la fin. Deux paramètres optionnels de la fonction `print` permettent de modifier ce comportement par défaut : `sep` et `end`. Par exemple, le programme :

```
print('Comportement','par','defaut.',sep=' ',end='\n')
print('Sur','une','seule','ligne','avec','des','tirets.',sep='-',end='')
print('Ancienne ligne.\n','Nouvelle ligne','.')
```

affiche

```
Comportement par defaut.
Sur-une-seule-ligne-avec-des-tirets.Ancienne ligne.
 Nouvelle ligne .
```

Le caractère `'\n'` est un caractère de contrôle qui lorsqu'il est affiché provoque un passage à la ligne. Dans la deuxième instruction `print`, on a remplacé le caractère de fin `'\n'` par `''` : `print` ne va pas à la ligne.

Complément sur les chaînes de caractères

On peut convertir un entier en chaîne de caractères : `str(42) == '42'`. Réciproquement, on peut convertir une chaîne de caractères en entier : `int('42') == 42`. Pour concaténer deux chaînes de caractères, on utilise l'opérateur +. On a par exemple `'caram' + 'bolage' == 'carambolage'` ; ce + n'a rien à voir avec l'addition sur les entiers : `'2' + '3'` vaut `'23'`, mais pas `'3'+'2'`.

Exercice 5 — Frise (⋆– ⋆⋆)

1. Écrire une fonction `frise(n)` qui affiche une frise de longueur `n` alternant deux symboles `#` et `*`.

```
>>> frise(0)

>>> frise(1)
#
>>> frise(5)
#*#*#
>>> frise(10)
#*#*#*#*#*
```

```
def frise(n):
    i=0
    while i<n:
        if i%2==0:
            print("#",end='')
        else:
            print("*",end='')
        i=i+1
   print()
```

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 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
```

```python
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.

```
# 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
```

```
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)
```