Master : Méthodes Numériques et Statistiques Appliquées (MNSA)
Pr. Youssef OUASSIT

FSAC
FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

TD 1 : Syntaxe de base et fonctions

Exercice 1 :

1. Ecrire une fonction « **somme_div(n)** » qui retourne la somme de tous les diviseurs d'un nombre passé en paramètre. (n doit être positif)

Un nombre entier est parfait s'il est égal à la somme de ses diviseurs (sauf lui-même).
Ex : 6 = 1 + 2 + 3 est parfait.

2. Ecrire une fonction « **parfait (m)** qui teste si un nombre passé en paramètre **m** est parfait et qui retourne **True** s'il l'est et **False** sinon.
3. Ecrire une fonction « **premier (x)** qui teste si un nombre passé en paramètre **x** est premier et qui retourne **True** s'il l'est et **False** sinon.

Correction :

```
def somme_div(n):
    s = 0
    for i in range(1, n + 1):
        if n % i == 0:
            s += i
    return s

def parfait(m):
    if m <= 1:
        return False
    return somme_div(m) - m == m

def premier(x):
    if x < 2:
        return False

    for i in range(2, x):
        if x % i == 0:
            return False

    return True

def premierV2(x):
    if x < 2:
        return False
    return somme_div(x)==x+1
```

Exercice 2 :

Un **nombre narcissique** (ou nombre d'Armstrong) est un entier naturel non nul égal à la somme de ses chiffres élevés à la puissance (p), où (p) est le nombre de chiffres du nombre.

Par exemple : $153 = 1^3 + 5^3 + 3^3$ et $548834 = 5^6+4^6+8^6+8^6+3^6+4^6$

Travail demandé :

Écrire un programme Python permettant de vérifier si un entier n saisi par l'utilisateur est narcissique.

Le programme devra être décomposé en fonctions :

nombre_chiffres(n) # Retourne le nombre de chiffres de n supposé positif
somme_puissances_chiffres(n, p) # Retourne la somme des chiffres de n élevés à la puissance p
est_narcissique(n) # Retourne True si n est narcissique et False sinon

Correction :

```
def nombre_chiffres(n):
    compteur = 0
    while n > 0:
        compteur += 1
        n = n // 10
    return compteur

def somme_puissances_chiffres(n, p):
    if n < 0 or p < 1:
        raise ValueError("n doit être positif ou nul et p >= 1")

    somme = 0
    while n > 0:
        chiffre = n % 10
        somme += chiffre ** p
        n //= 10
    return somme


def est_narcissique(n):
    if n <= 0:
        return False
    p = nombre_chiffres(n)
    return somme_puissances_chiffres(n, p) == n
```

Exercice 3 :

Un joueur lance un certain nombre de fléchettes. Chaque fléchette peut rapporter 0, 20 ou 50 points selon la zone touchée.

Par exemple, pour 10 fléchettes et un score total de 300 points, une possibilité est :
3x0 + 5x20 + 2x50 = 300 avec 3 + 5 + 2 = 10

Écrire une fonction : possibilites(nb_flechettes, score) qui affiche toutes les possibilités permettant d'obtenir exactement score points avec nb_flechettes lancées.

Correction :

```python
def possibilites(nb_flechettes, score):
    for n50 in range(nb_flechettes + 1):
        for n20 in range(nb_flechettes - n50 + 1):
            n0 = nb_flechettes - n50 - n20
            if 20 * n20 + 50 * n50 == score:
                print(f"{n0} x 0 + {n20} x 20 + {n50} x 50 = {score}")
```

Exercice 4 :

Implémentez une fonction **fibo(n)** qui renvoie le n ème terme de la suite de Fibonacci en utilisant à la fois une approche **itérative** et une approche **récursive**.

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_n = U_{n-2} + U_{n-1} \end{cases}$$

Correction :

```python
def fibo(n):
    assert n > 0 , "n doit être positif ou nul"
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a


def fibo_recursif(n):
    assert n > 0 , "n doit être positif ou nul"
    if n <= 1:
        return n
    return fibo_recursif(n - 1) + fibo_recursif(n - 2)
```

Exercice 5 : Approximation de π

On utilise la formule de Leibniz :

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

Écrire une fonction **approx_pi(n)** qui calcule une approximation de **π** à l'aide des n premiers termes. Tester pour n = 10, 100, 1000, 10000 et comparer avec math.pi.

Correction :

```python
def approx_pi(n):
    assert n > 0, "n doit être strictement positif"

    somme = 0.0
    for k in range(n):
```

```
    somme += (-1) ** k / (2 * k + 1)
return 4 * somme
```