> D'après 2023, sujet zéro A
> Auteurs : Frédéric Junier, Franck Chambon, Denis Quenton

Exécution de programmes, recherche et corrections de bugs

> D'après 2022, Asie, J2, Ex. 5

1. On considère la fonction `somme` qui reçoit en paramètre un entier `n` strictement positif et renvoie le résultat du calcul $1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$.

```
def somme(n) :
    total = 0
    for i in range(n):
        total = total + 1 / i
    return total
```

   Lors de l'exécution de `somme(10)`, le message d'erreur `#!py ZeroDivisionError: division by zero` apparait. Identifier le problème et corriger la fonction pour qu'elle effectue le calcul demandé.

   > ✓ **"Réponse"**
   >
   > Il s'agit d'un problème d'indice mal parcouru par `#!py range(n)` qui va parcourir les entiers de 0 inclus à n exclu au lieu de 1 inclus à $n+1$ exclu.
   >
   > ```
   > def somme(n) :
   >     total = 0
   >     for i in range(1, n + 1):
   >         total = total + 1 / i
   >     return total
   > ```

2. On considère la fonction `maxi` qui prend comme paramètre une liste `L` **non vide** de nombres et renvoie le plus grand nombre de cette liste :

```
def maxi(L) :
    indice = 0
    maximum = 0
    while indice <= len(L):
        if L[indice] > maximum :
            maximum = L[indice]
        indice = indice + 1
    return maximum
```

a. Lors de l'exécution de `maxi([2, 4, 9, 1])` une erreur est déclenchée. Identifier et corriger le problème.

"Réponse"

Tout d'abord il s'agit d'un problème de dépassement d'indice dans la liste `L` , puisqu'au dernier tour de boucle `indice` est égal à `len(L)` qui est en dehors de la plage de validité des indices (entre `0` et `len(L) - 1`)

On corrige ce premier bug :

```
def maxi(L) :
    indice = 0
    maximum = 0
    while indice < len(L) :
        if L[indice] > maximum :
            maximum = L[indice]
        indice = indice + 1
    return maximum
```

b. Le bug précédent est maintenant corrigé. Que renvoie à présent l'exécution de `maxi([-2, -7, -3])` ? Modifier la fonction pour qu'elle renvoie le bon résultat.

"Réponse"

Ensuite comme on a initialisé `maximum` à `0` , le parcours des éléments tous négatifs de `[-2, -7, -3]` ne peut modifier la valeur de `maximum` .
`maxi([-2, -7, -3])` renvoie donc `0` qui n'est pas le maximum de `[-2, -7, -3]` .

On corrige ce bug en initialisant `maximum` avec le premier élément de la liste `L` non vide.

```
def maxi(L) :
    indice = 0
    maximum = L[0]
    while indice < len(L):
        if L[indice] > maximum :
            maximum = L[indice]
        indice = indice + 1
    return maximum
```

3. On souhaite réaliser une fonction qui génère une liste de `n` joueurs identifiés par leur numéro. Par exemple on souhaite que l'appel `genere(3)` renvoie la liste `['Joueur 1', 'Joueur 2', 'Joueur 3']` .

```
def genere(n) :
    L = []
    for i in range(1, n + 1):
        L.append('Joueur ' + i)
    return L
```

L'appel `genere(3)` déclenche l'erreur suivante

```
#!py TypeError: can only concatenate str (not "int") to str.
```

Expliquer ce message d'erreur et corriger la fonction afin de régler le problème.

> ✓ **"Réponse"**
>
> Le message d'erreur signale que lors de l'exécution du code, Python a essayé de concaténer une chaine de caractères et un entier. La concaténation avec une chaine n'est possible qu'avec une autre chaine de caractères. Pour corriger ce bug il suffit de convertir l'entier en chaine de caractère avec le constructeur `str` .
>
> ```
> def genere(n) :
> L = []
> for i in range(1, n + 1):
>     L.append('Joueur ' + str(i))
> return L
> ```

4. On considère la fonction `suite` qui prend un entier positif `n` en paramètre et renvoie un entier.

```
def suite(n) :
    if n == 0:
        return 0
    else :
        return 3 + 2 * suite(n - 2)
```

a. Quelle valeur renvoie l'appel de `suite(6)` ?

> **"Réponse"**
>
> On commence par évaluer `suite(6)` .
>
> `suite(6) = 3 + 2 * suite(6 - 2) = 3 + 2 * suite(4)`
>
> Or `suite(4) = 3 + 2 * suite(4 - 2) = 3 + 2 * suite(2)`
>
> On continue la descente : `suite(2) = 3 + 2 * suite(2 - 2)= 3 + 2 * suite(0)`
>
> 0 est un cas de base, la descente s'arrête : `suite(0) = 0` .
>
> On peut remonter pour calculer toutes les valeurs en attente :
>
> `suite(2) = 3 + 2 * suite(0) = 3` puis `suite(4) = 3 + 2 * suite(2) = 9` et enfin
>
> `suite(6) = 3 + 2 * suite(4) = 21`

b. Que se passe-t-il si on exécute `suite(7)` ?

> **"Réponse"**
>
> On essaie d'évaluer `suite(7)` de la même façon.
>
> `suite(7) = 3 + 2 * suite(7 - 2) = 3 +2 * suite(5)` . Or
>
> `suite(5) = 3 + 2 * suite(5 - 2) = 3 + 2 * suite(3)` . On continue la descente :
>
> `suite(3) = 3 + 2 * suite(3 - 2)= 3 + 2 * suite(1)` . On continue la descente :
>
> `suite(1) = 3 + 2 * suite(1 - 2)= 3 + 2 * suite(-1)` . Zut, on a passé le cas de base et la descente ne s'arrêtera jamais, elle sera infinie ! Sauf que ...
>
> Le calcul de `suite(7)` sera stoppé par l'interpréteur Python au bout de 1000 appels récursifs et une erreur sera affichée. Il est possible de modifier cette valeur ; un garde fou.

5. On considère le code Python ci-dessous :

```
x = 4
L = []
def modif(x, L):
    x = x + 1
    L.append(2 * x)
    return x, L

print(modif(x, L))
print(x, L)
```

a. Qu'affiche le premier `print` ?

b. Qu'affiche le second `print` ?

"Réponse"

Observons une sortie en console.

```
>>> x = 4
>>> L = []
>>> print(modif(x, L))
(5, [10])
>>> print(x, L)
4, [10]
```

Le premier `#!py print` affiche un tuple, les valeurs de `x` et `L` renvoyées par `modif(x, L)` . Ce sont des valeurs de `x` et `L` dans la portée locale de la fonction `modif` : c'est-à-dire `#!py 5` pour `x` et `#!py [10]` pour `L` à la fin de l'évaluation de `modif(x, L)`

Le second `#!py print` affiche les valeurs des variables `x` et `L` après exécution de `modif(x, L)` mais dans la portée globale du script. Les valeurs de `x` et `L` ont été transmises en paramètres à `modif` et recopiées dans des variables locales de même nom. `modif(x, L)` a donc modifié des copies des variables globales `x` et `L` de valeurs initiales respectives `#!py 4` et `[]` . La modification de la valeur reçue par la variable locale `x` n'a aucune incidence sur la variable globale `x` car cette valeur est un entier de type simple. En revanche la valeur de la variable globale `L` de type `list` est une référence vers la séquence de valeurs. La variable locale `L` et la variable globale `L` partagent la même référence donc par effet de bord, les modifications appliquées à la variable locale sont répercutées à la variable globale.

Ainsi, après évaluation de `modif(x, L)`, `L` est modifiée mais pas `x`. Le second `print` affiche la valeur `#!py 4` pour `x` et la valeur `#!py [10]` pour `L`.