TP2

Partie 1 — Les expressions booléennes

Ex. 1

Cet exercice est à effectuer dans la console Python.

Question 1 : Exécuter dans la console:

```
>>> True
>>> 1+1==2
>>> False
>>> 1+1==3
>>> type(True)
>>> type(False)
>>> type(1+1==2)
```

Noter que `True` *et* `False` *sont des valeurs en python, au même titre que* `1, 2, 3...`. *Par contre* `True` *et* `False` *sont des valeurs de type* `bool` *et non pas* `int`.

De même que l'on peut faire des expressions arithmétiques qui calculent sur les entiers, ont peut faire des expressions booléennes qui calculent sur les booléens.

Les opérateurs booléens sont : * `and` : "et" * `or` : "ou" * `not` : "non"

Question 2 : Exécuter dans la console:

```
>>> True and False
>>> True and True
>>> True or False
>>> True or True
>>> True and (True or False)
>>> not False
>>> not True
```

Question 3 : Exécuter dans la console:

```
>>> a=False
>>> b=False
>>> not (a or b) ==  (not a) and (not b)
```

Réévaluer cette dernière expression booléenne avec les trois autres combinaisons de valeurs pour `a` et `b`.

En logique, cette égalité est connue sous le nom de Loi de De Morgan.

On peut bien sûr générer des valeurs booléennes en comparant des valeurs non-booléennes. Les opérateurs de comparaison sont:

- `<` : "strictement inferieur à"

- `<=` : “inférieur ou égal à”
- `>` : “strictement supérieur à”
- `>=` : “supérieur ou égal à”
- `==` : “égal à”
- `!=` : “différent de”

Attention: `==` correspond à une question : c'est deux valeurs sont-elles égales? Tandis que `=` correspond à une affectation : on décide qu'une certaine variable doit désormais valoir une certaine valeur.

Question 4 : Exécuter dans la console:

```
>>> 3 == 4
>>> 3 != 4
>>> 3 <= 4
>>> 3+4 >= 7
>>> 3+4==7 or 3<=7-5
>>> 24 % 2 == 0 and 24 % 3
```

Question 5 : Initialiser les variables a, b et c respectivement aux valeurs 2, 5 et 7. Déterminer si les tests suivants sont vrais ou faux en utilisant les opérateurs booléens et les opérateurs de comparaison correspondants : * a **égale à** b * (a + b) **différent de** c * **non** (a **égale à** c) * ((a + b) **supérieur ou égal à** c) **et** (a **inférieur ou égal à** (c − b)) * ((a + c) > b) **ou** ((a + c) < b)

Partie 2 — If

Ex. 2

L'instruction `if` signifie “si”. Elle permet de n'exécuter un bloc d'instruction que si une certaine condition (une expression booléenne) s'évalue à vrai.

```
if expr :
    # bloc d'instructions
    # qui n'est exécuté que
    # que si expr est vraie
# suite des instructions
```

IMPORTANT : Il y a une indentation de 4 espaces (ou une tabulation) entre la ligne du `if`, et les lignes du bloc. Ce type d'espace laissé dans la marge de gauche s'appelle l'indentation. L'indentation est primordiale en python, car elle permet de voir quand commence et s'arrête le bloc d'instruction qui est conditionné par le `if`.

Les deux points après la condition sont indispensables aussi. Ils signifient “alors”. Par exemple, la phrase française “si x est pair alors diviser x par deux” se traduit en python par:

```
if x % 2 == 0 :
    x = x // 2
```

Variation avec alternative:

```
if expr :
    # bloc d'instructions
    # qui n'est exécuté que
    # que si expr est vraie
else :
    # bloc d'instructions
    # qui n'est exécuté que
    # que si expr est fausse
# suite des instructions
```

Variation avec cas multiples:

3. Cas multiples

```
if expr1 :
    # bloc d'instructions
    # qui n'est exécuté que
    # que si expr1 est vraie
elif expr2:
    # bloc d'instructions
    # qui n'est exécuté que
    # que si expr2 est vraie
else :
    # bloc d'instructions
    # qui n'est exécuté que
    # que si expr1 et expr2 sont fausses
# suite des instructions
```

Question 1 : Affecter la valeur `17` à la variable `note`. Ecrire deux lignes de code qui correspondent à la phrase française "si la valeur de a est supérieure ou égal à 17 alors afficher bravo!". Tester vos deux lignes en changeant la valeur de `a`.

L'instruction `input(.)` sert à poser une question à *l'utilisateur* du programme (à ne pas confondre avec le programmeur du programme!). * A l'intérieur de la parenthèse on met la question que l'on veut poser. Elle s'affichera comme si c'était un `print(.)`. * L'expression entière prendre comme valeur la chaîne de caractère qu'aura saisi l'utilisateur en réponse à votre question.

Par exemple

```
rep=input("Comment allez-vous?")
```

Affiche "Comment allez-vous?"; attend de l'ulisateur qu'il tape une réponse; et met cette réponse dans `rep`.

Question 2 : Ecrivez un programme python qui pose la question “Bonjour, comment ça va?”. Si l'utilisateur répond “Bien” alors affichez le message “Super! Je suis content pour vous.” Si la réponse est différente de “Bien” affichez “Mince... courage!”.

Question 3 : Recopier et modifier la question 1 pour que le programme demande à l'utilisateur de saisir sa note, avant de lui dire si celle-ci est au-dessus de la moyenne ou pas.

Noter qu'il vous faut convertir le résultat de `input(.)` *en entier avec la fonction* `int(.)` *afin de pouvoir le comparer à 10.*

Question 4 : Ecrire un programme qui demande son âge à l'utilisateur : “Quel est votre âge ?” et s'il oui ou non il souhaite des pop corns : “Souhaitez-vous des pop corns ?”. Le programme affiche alors le `prix` de sa séance de cinéma. Si la personne est mineure, le prix d'une place est de 7EUR, si la personne est majeure le prix est de 12EUR. Le pop corn est à 5EUR.

Correction

```
#Q1
note = 17
if note >= 10:
    print("Vous avez la moyenne")

#Q2
rep = input("Bonjour, comment ça va?")
if rep == "Bien":
    print("Super! Je suis content pour vous.")
else:
    print("Mince... courage!")

#Q3
note = int(input("Quelle est votre note? "))
if note >= 10:
    print("Vous avez la moyenne")

#Q4
age = int(input("Quel est votre age ?"))
if age < 18:
    prix= 7
else:
    prix= 12
if input("Souhaitez-vous un pop corn ?")=="oui" :
```

```
    prix+= 5
print("Le prix total est de :",prix)
```

Partie 3 — For

Ex. 3

L'instruction `for` signifie "pour". Elle permet de répéter l'exécution d'un bloc d'instruction un nombre fixe de fois; tout en faisant varier la valeur d'une certaine variable.

```
for i in range(initial, final) :
    # bloc d'instructions
    # qui s'exécute pour `i` allant de
    # la valeur initiale incluse
    # à la valeur finale excluse
# suite des instructions
```

Par exemple, la phrase française "pour i allant de 1 à 5 affiche Hello!" se traduit en python par

```
for i in range(1,6) :
    print("Hello!")
```

Pour faire un "countdown" de fusée, on pourrait utiliser

```
for i in range(10,-1,-1) :
    print(i,"...")
print("Go !!!")
```

On peut aussi faire varier `i` dans une liste arbitraire.

```
print("Les premiers nombres premiers sont ")
for i in [2,3,5,7] :
    print(i)
```

D'ailleurs, nul besoin d'utiliser spécifiquement la variable `i`.

Question 1 : Ecrire un programme qui calcule et affiche les cubes de 1 à 100. Remarque: utiliser l'opérateur `**` pour la puissance.

Question 2 : Ecrire un programme qui calcule la moyenne des cubes de 1 à 100, et affiche le résultat final.

Question 3 : Ecrire un programme qui calcule la somme des nombres de la forme $1/2^i$, pour i allant de 1 à 100, et affiche le résultat final.

Correction

```
start=1
end=100

#Q1
for i in range(start,end+1):
    print(i**3)

#Q2
somme=0
for i in range(start, end+1):
    somme += i
print(somme /(end-start))

#Q3
somme=0
for i in range(start, end+1):
    somme += 1/2**i
print(somme)
```

Ex. 4 — Fibonacci

Écrire un programme qui affiche successivement les 20 premiers nombres de la suite de Fibonacci de 20. En termes mathématiques cette suite est définie par:

- $F(0) = 0$, $F(1) = 1$
- $F(n) = F(n-2) + F(n-1)$ pour $n > 1$

La difficulté, ici, est que nous ne savons par encore comment définir des fonctions du style $F(n)$, ni même des listes du style F_n. Nous apprendrons ça, mais pour l'heure, nous allons nous en passer.

Au lieu de ça nous utiliserons seulement trois variables: `Fn_moins_2` initialisée à `0`,`Fn_moins_1` initilisée à `1`, et `Fn`.

Nous afficherons les valeurs se la suite en procédant ainsi:

Afficher `0`

Afficher `1`

Pour les 18 prochains termes:

- calculer `Fn=Fn_moins_1+Fn_moins_2` et l'afficher.
- Mettre `Fn_moins_1` dans `Fn_moins_2`.
- Mettre `Fn` dans `Fn_moins_1`.

Question 1 : Ecrire le programme en question.

Ex. 5 — Pyramides

Si je répète 5 fois, le fait de répéter 10 fois “Hello!”, combien de fois est-ce que je répète “Hello!” au total?

Une boucle imbriquée est une boucle inclue dans une autre. Cela correspond à l'idée de répéter (boucle extérieure) une répétition (boucle intérieure).

Voici un example.

```
num_lins = 10
num_cols = 5
for i in range(num_lins):
    for j in range(num_cols):
        print('*', end="")
    print("")
```

Affichage après exécution :

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Remarquer le `print("")` *qui affiche la chaîne vide… mais affiche aussi un saut de ligne, car tout* `print(.)` *termine son affichage par un saut de ligne. Sauf si on lui demande de ne pas le faire, en spécifiant* `print(.,end="")`.

Question : Écrire un programme qui affiche à l'écran une pyramide remplie d'étoiles, sur le modèle ci-dessous. La dernière ligne contient 10 étoiles

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
* * * * * * * * * *
```

Correction

```
 #Q1
for i in range(1, 11):
    for j in range(1, i+1):
        print("* ", end ="")
    print()
```

Partie 4 — Boucle While

Ex. 6

L'instruction `while` signifie "tant que". Elle permet d'exécuter un bloc d'instructions, de façon répétée, tant qu'une condition est vraie. Elle sert donc, comme le `for`, à répéter des instructions. Mais, à la différence du `for`, elle permet de répéter ces instructions un nombre indéterminé de fois.

```
while expr :
    # bloc d'instructions
    # exécuté tant que
    # expr est vraie
# suite des instructions
```

Autrement dit: * L'interpréteur évalue la condition. * Si la condition est vraie, il exécute le bloc et retourne évaluer la condition. * Si la condition est fausse il saute le bloc et poursuit la suite du programme.

Bien sûr, un `while` permet de faire ce qu'aurait déjà su faire un `for`, mais comme c'est un peu plus compliqué que le `for` on ne l'utilise généralement pas pour ça:

```
i=10
while  i >= 0:
    print(i,"...")
    i = i - 1
print("Go !!!")
```

Ce qui affiche

```
10 ...
9 ...
8 ...
7 ...
6 ...
5 ...
4 ...
3 ...
2 ...
```

```
1 ...
0 ...
Go !!!
```

Le `while` est aussi plus risqué que le `for`, car il faut s'assurer que le programme ne restera pas prisonnier d'une boucle infinie; répétant à jamais le même bloc d'instructions. Il faut donc se convaincre que la condition finira par être fausse.

Voici un exemple simple de boucle infinie:

```
while True:
    print("Hello!")
```

Question 1 : Ecrire un programme qui demande à l'utilisateur "Entrez un nombre entre 1 et 10 :". Si l'utilisateur ne respecte pas la consigne, il affiche "Essayez encore!" et redemande ce nombre. Ainsi de suite jusqu'à ce que l'utilisateur respecte la consigne.

Question 2 : Écrire un programme qui fait la somme des nombres entiers positifs entrés par l'utilisateur, tant que ce dernier souhaite continuer. Il s'arrête dès que l'utilisateur entre un nombre négatif.

Question 2 : Écrire un programme qui demande à l'utilisateur de calculer la suite des i^3 tant que le résultat de ce calcul est inférieur à $1000i^2 + 500i + 7$. En déduire la partie entière de la solution de l'équation $-i^3 + 1000i^2 + 500i + 7 = 0$.

Correction

```
#Q1
n = int(input("Entrez un nombre entre 1 et 10 :"))
while n<0 or n>10:
    print("Essayez encore!")
    n = int(input("Entrez un nombre entre 1 et 10 :"))

#Q2
n = 0
somme = 0
while n >= 0:
    somme += n
    n = int(input("Entrez un nombre positif:"))
print("La somme des nombres positifs est", somme)

#Q3
i=0
while i**3<1000*i**2+500*i+7:
```

```
    i += 1
print("La partie entière de la solution est ",i-1)
```