

TP01 - Cadre des TP et Révisions de Python

October 4, 2021

1 Présentation des feuilles Jupyter

Dans une telle feuille il y a deux types de cellules : * les cellules de texte comme celle-ci qui servent à vous présenter ce cadre. * les cellules de code dans lesquelles du code peut être saisi et qui peuvent être évaluées en appuyant sur le bouton **Run** (ou **Exécuter**) dans la barre d'outils ou avec la combinaison de touches **Ctrl+Entrée**. * Certaines de ces cellules de code sont en lecture seule, elle correspondent à des exemples ou des tests qui ont déjà été écrits par vos professeurs. Vous devez tout de même les évaluer pour suivre les explications ou pour valider vos résultats.

```
[6]: # Évaluez cette cellule  
print('Bravo')
```

Bravo

- Afin d'évaluer automatiquement votre code, on vous demandera d'écrire votre code à la place d'une ligne qui est prévue pour produire une erreur si vous ne répondez pas à la question.

```
[13]: # Écrivez x = 3 comme réponse  
## BEGIN SOLUTION  
x = 3  
## END SOLUTION
```

- Des tests sont alors proposés pour vous permettre d'évaluer votre code en vérifiant qu'il fonctionne. Il est possible que des tests soient cachés pour moins vous guider.

```
[14]: # si vous n'avez pas répondu à la question précédente, ce test produira une  
      ↪ erreur  
from nose.tools import assert_equal  
  
assert_equal(x, 3)
```

2 Révisions de Python

Dans cette partie du TP, on va revoir très rapidement la syntaxe de **Python** qui a déjà été vu au lycée en mathématiques ainsi qu'en physique-chimie.

2.1 Valeurs

En **Python**, on manipule des valeurs de différents types.

2.1.1 Entiers relatifs `int`

Le type `int` correspond aux entiers relatifs qu'on peut ajouter, multiplier, ... avec les opérateurs usuels et qui permettent de faire des calculs en précision arbitraire (nombres aussi grands qu'on le souhaite, du moment que l'ordinateur a la capacité pour).

```
[1]: 3 + 5
```

```
[1]: 8
```

```
[1]: """la précision étant arbitraire, on peut manipuler de grands nombres"""  
2 ** 120
```

```
[1]: 1329227995784915872903807060280344576
```

```
[6]: 12 // 5 # division entière : renvoie le quotient de la division euclidienne
```

```
[6]: 2
```

```
[8]: 12 % 5 # opérateur modulo : renvoie le reste de la division euclidienne
```

```
[8]: 2
```

2.1.2 Nombres à virgule flottante `float`

Le type `float` correspond aux nombres à virgule flottante comme 1.3 ou -2e-3, qui permettent de faire des calculs scientifiques mais avec une précision donnée (erreurs d'arrondis ou de calcul)

```
[4]: 2.2 * 1.2
```

```
[4]: 2.64
```

```
[4]: 2.1 - 1 # si on mélange int et float, on passe en float
```

```
[4]: 1.1
```

```
[7]: 12 / 5 # division flottante
```

```
[7]: 2.4
```

2.1.3 Chaînes de caractères `str`

Le type `str` correspond aux chaînes de caractères comme "Bonjour " ou 'tout le monde' qui permettent de représenter une valeur constante textuelle. On peut utiliser + pour calculer la chaîne obtenue en mettant à la suite deux chaînes (on parle de **concaténation**).

```
[8]: "Bonjour " + 'tout le monde'
```

```
[8]: 'Bonjour tout le monde'
```

On peut accéder au nième caractère d'une chaîne `s` avec `s[n]` mais uniquement en lecture.

```
[9]: 'test'[1] # on compte à partir de 0
```

```
[9]: 'e'
```

2.2 Variables

En Python, on appelle *variable* un lien entre un nom et une valeur. L'instruction `var = valeur` permet de définir ou de rédefinir ce lien. Par exemple,

```
a = 3
```

lie la valeur 3 au nom `a`. Après exécution de cette instruction, on peut donc utiliser `a` à la place de cette valeur.

```
[9]: var1 = 5  
var2 = 'Chaine'
```

```
[6]: var2
```

```
[6]: 'Chaine'
```

```
[7]: var2 = 3 # on définit un nouveau lien, les types des valeurs n'importe pas  
var2+var1
```

```
[7]: 8
```

2.2.1 Échanger deux variables

Pour échanger deux variables il faut prendre soin d'utiliser une variable temporaire. En effet, si on effectue juste les instructions

```
a = 3  
b = 4  
b = a  
a = b
```

Toutes les variables seront liées à la valeur 3.

```
[1]: # Échangez les valeurs liées à a et à b  
a = 5  
b = 2  
## BEGIN SOLUTION  
t = b  
b = a  
a = t  
## END SOLUTION
```

```
[13]: from nose.tools import assert_equal

assert_equal(b, 5)
assert_equal(a, 2)
```

2.3 Instructions

On a déjà vu l'instruction d'affectation `var = valeur` permettant de lier une valeur à un nom de variable.

Il existe d'autres instructions en Python. Par exemple, `print(expression)` affiche le résultat de l'expression sur ce qu'on appelle la sortie standard du programme. C'est une instruction qu'on utilisera peu ici car pour afficher une valeur avec Jupyter, il suffit de l'écrire dans une cellule et d'évaluer la cellule comme on a pu le voir précédemment.

2.3.1 Instruction conditionnelle

Certaines instructions permettent de manipuler le flot de contrôle du programme, c'est-à-dire la manière dont les instructions sont exécutées. L'instruction conditionnelle est une de ces instructions, sa syntaxe consiste à écrire :

```
if expression:
    bloc d'instructions
    si vraie
else:
    bloc d'instructions
    si fausse
```

Cette instruction permet donc de considérer deux blocs d'instructions, séparés par ce qu'on appelle une indentation (des espaces ou, à éviter, des tabulations) ainsi qu'une expression qui permet de déterminer le bloc à exécuter. Cela fonctionne donc comme un système d'aiguillage pour un train, on choisit la direction où le flot va continuer.

Il est également possible de ne pas préciser de bloc au cas où l'expression soit fausse :

```
if expression:
    bloc d'instructions
    si vraie
```

ce qui est équivalent au bloc factice `pass` :

```
if expression:
    bloc d'instructions
    si vraie
else:
    pass
```

On peut enchaîner les tests avec `elif`, ainsi, au lieu d'écrire

```
if e1
    b1
else:
```

```

if e2:
    b2
else:
    b3

```

on écrira plutôt :

```

if e1:
    b1
elif e2:
    b2
else:
    b3

```

3 Fonctions

Un point essentiel en informatique est celui de la réutilisation efficace et sûre des programmes. C'est pour cela qu'on a introduit la notion de fonction afin d'organiser des instructions. Une fonction c'est une suite d'instructions paramétrées par des entrées, les arguments, et qui produit une valeur, appelée *valeur de retour* quand on l'appelle.

On déclare une fonction ainsi :

```

def mafonction(arg1, ..., argn):
    bloc d'instructions

```

Pour la valeur de retour, on utilise l'instruction **return valeur**. Pour appeler la fonction, il suffit d'utiliser son nom et de donner des valeurs pour les arguments : **mafonction(val1, ..., valn)** le bloc d'instruction va alors s'exécuter en liant des variables **arg1, ...** aux valeurs **val1, ...** La valeur de cette expression sera alors la valeur de retour correspondante (ou **None** s'il n'y a pas d'instruction **return** dans le bloc).

Ainsi, la fonction suivante permet de passer deux nombres et renvoie leur somme :

```

def ajoute(a, b):
    return a+b

```

Si on écrit **ajoute(1,2)*4** cela va alors exécuter la fonction et évaluer l'expression comme s'il y avait **3*4**.

A partir de maintenant, les questions consisteront, étant donné le début de définition d'une fonction, à écrire les instructions qui la constitue. Les tests de validation sont alors des appels de vos fonctions.

```

[5]: def multiplie(a,b):
      '''Prend en entrée deux nombres a et b et renvoie leur produit.'''
      ## BEGIN SOLUTION
      return a*b
      ## END SOLUTION

```

```

[8]: from nose.tools import assert_equal

      assert_equal(multiplie(2,3), 6)

```

Une fonction peut très bien avoir pour argument une autre fonction :

```
[3]: def applique_deux_fois(f, a):  
    '''Prend en entrée une fonction f de deux arguments, l'applique sur a et a_1  
    → et  
    renvoie le résultat.'''  
    ## BEGIN SOLUTION  
    return f(a,a)  
    ## END SOLUTION
```

```
[4]: from nose.tools import assert_equal  
  
assert_equal(applique_deux_fois(multiplie, 3), 9)
```

4 Boucles

Les boucles sont un élément essentiel de l'informatique : ce sont les instructions qui permettent de répéter quelque chose. On parle d'itération pour une répétition d'un même bloc d'instructions.

On distingue deux types de boucle s:

- les boucles **conditionnelles** dont une condition permet de déterminer si on doit en sortir ou si on doit continuer avec une autre itération. La syntaxe en Python est :

```
while condition:  
    bloc d'instruction  
    à exécuter
```

- les boucles **non conditionnelles** qui effectuent un bloc d'instruction un nombre de fois donné. En Python ces boucles sont régies par des valeurs dont on peut énumérer les éléments, on parle d'**itérable**. Ce sont des valeurs pour lesquelles on peut identifier un premier élément et un ordre de parcours après celui-ci. L'itérable le plus simple est `range(a,b)` qui contient les entiers a, a+1, ..., b-1. Ainsi `range(0,5)` énumère 0, 1, 2, 3, 4. Comme ce cas est fréquent, le premier paramètre peut être ignoré, on peut ainsi juste écrire `range(5)`. La syntaxe d'une boucle non conditionnelle en Python est alors :

```
for element in iterable:  
    bloc d'instruction  
    à exécuter
```

Quand on va exécuter ce bloc, on va donc modifier la variable `element` qui va être tour à tour liée à chaque élément présent dans `iterable`.

```
[2]: # Exemple  
for i in range(5):  
    print('La valeur de i est :', i)
```

```
La valeur de i est : 0  
La valeur de i est : 1  
La valeur de i est : 2
```

La valeur de i est : 3
La valeur de i est : 4

Notons qu'on peut tout à fait faire l'exemple précédent avec une boucle `while` en définissant nous même la variable `i` :

```
[3]: i = 0
while i < 5:
    print('La valeur de i est :', i)
    i = i + 1 # à ne pas oublier sinon on reste indéfiniment dans la boucle !
```

La valeur de i est : 0
La valeur de i est : 1
La valeur de i est : 2
La valeur de i est : 3
La valeur de i est : 4

4.1 Calcul de la somme des carrés

On vous demande ici de calculer $\sum_{k=0}^n k^2$ à l'aide d'une boucle `for`.

```
[2]: def somme_carres(n):
    '''Renvoie la somme des k**2 pour k allant de 0 à n'''
    ## BEGIN SOLUTION
    somme = 0
    for k in range(n+1):
        somme = somme + k**2
    return somme
    ## END SOLUTION
```

```
[27]: from nose.tools import assert_equal

assert_equal(somme_carres(12), 650)
assert_equal(somme_carres(0), 0)
```

4.2 Puissance décroissante et factorielle

Soit n et k des entiers, on note $n^{(k)} = n \times (n-1) \times \dots \times (n-k+1)$ appelée n à la puissance décroissante k . On note $n! = n^{(n)}$ appelée la factorielle de n .

```
[4]: def puissance_decroissante(n, k):
    '''Renvoie n à la puissance décroissante k'''
    ## BEGIN SOLUTION
    produit = 1
    for i in range(k):
        produit = produit * (n-i)
    return produit
    ## END SOLUTION
```

```
[7]: from nose.tools import assert_equal

assert_equal(puissance_decroissante(4, 4), 24)
assert_equal(puissance_decroissante(12, 3), 1320)
assert_equal(puissance_decroissante(5, 1), 5)
assert_equal(puissance_decroissante(5, 0), 1)
```

```
[9]: def factorielle(n):
    '''Renvoie n! en utilisant la fonction précédente'''
    ## BEGIN SOLUTION
    return puissance_decroissante(n, n)
    ## END SOLUTION
```

```
[10]: from nose.tools import assert_equal

assert_equal(factorielle(0), 1)
assert_equal(factorielle(1), 1)
assert_equal(factorielle(4), 24)
```

4.3 Plus grande puissance 2 supérieure

Il est très important en informatique de savoir trouver le nombre minimal de bits sur lesquels on peut coder une information. Pour un entier naturel n , il s'agit de trouver le plus petit d tel que $2^d > n \geq 0$.

Étant donné n , on ne peut pas savoir (*bon en fait on pourrait avec un logarithme, mais on va supposer qu'on ne sait pas*) à l'avance

```
[8]: def plus_grande_puissance(n):
    '''Renvoie le plus petit entier d tel que n < 2**d'''
    ## BEGIN SOLUTION
    d = 0
    p = 1
    while p <= n:
        p = p * 2
        d = d + 1
    return d
    ## END SOLUTION
```

```
[9]: from nose.tools import assert_equal

assert_equal(plus_grande_puissance(0), 0)
assert_equal(plus_grande_puissance(1), 1)
assert_equal(plus_grande_puissance(15), 4)
```

5 Autour de la suite de Syracuse

Étant donné $a \in \mathbb{N}^*$, on appelle suite de Syracuse de premier terme a , la suite $(u_n)_{n \in \mathbb{N}}$ définie par $u_0 = a$ et pour $n \in \mathbb{N}$:

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

Rappel : pour déterminer si un nombre a est pair on peut utiliser le modulo et tester `a % 2 == 0`. La division entière par deux se note `a // 2`.

```
[10]: def suivant(u):  
    '''Etant donné un entier u représentant un terme u_n de la suite  
    renvoie le terme u_(n+1)'''  
    ## BEGIN SOLUTION  
    if u % 2 == 0:  
        return u // 2  
    else:  
        return 3 * u + 1  
    ## END SOLUTION
```

```
[11]: from nose.tools import assert_equal  
  
assert_equal(suivant(14), 7)  
assert_equal(suivant(7), 22)
```

```
[12]: def syracuse(a, n):  
    '''Renvoie u_n le nième terme de la suite de Syracuse de premier terme a'''  
    ## BEGIN SOLUTION  
    u = a  
    for i in range(n):  
        u = suivant(u)  
    return u  
    ## END SOLUTION
```

```
[13]: from nose.tools import assert_equal  
  
assert_equal(syracuse(3,5), 4)  
assert_equal(syracuse(1,0), 1)  
assert_equal(syracuse(1,1), 4)  
assert_equal(syracuse(1,2), 2)
```

En évaluant les termes de la suite, on constate qu'elle semble devenir périodique à partir d'un certain rang. Par exemple, en partant de $a = 14$:

14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...

Quelle que soit la valeur de u_0 , on retombe sur ce cycle. (*Demandez une preuve à votre professeur de mathématiques à 12h05*)

On appelle temps de vol pour a , le plus petit n tel que $u_n = 1$ et altitude maximale pour a la plus grande valeur prise par la suite. Ici, le temps de vol pour 14 est 17 et son altitude maximale est 52.

```
[1]: def temps_de_vol(a):  
    '''Renvoie le temps de vol pour a'''  
    # On pourra réutiliser syracuse mais c'est plus efficace de reprendre le  
    ↪ calcul  
    ## BEGIN SOLUTION  
    u = a  
    n = 0  
    while u != 1:  
        u = suivant(u)  
        n = n + 1  
    return n  
    ## END SOLUTION
```

```
[15]: from nose.tools import assert_equal  
  
assert_equal(temps_de_vol(1), 0)  
assert_equal(temps_de_vol(14), 17)
```

```
[17]: def altitude_maximale(a):  
    '''Renvoie l'altitude maximale pour a'''  
    ## BEGIN SOLUTION  
    if a == 1 or a == 2:  
        return 4  
    u = a  
    alt = a  
    while u != 1:  
        u = suivant(u)  
        if u > alt:  
            alt = u  
    return alt  
    ## END SOLUTION
```

```
[18]: from nose.tools import assert_equal  
  
assert_equal(altitude_maximale(1), 4)  
assert_equal(altitude_maximale(2), 4)  
assert_equal(altitude_maximale(14), 52)
```

```
[20]: def plus_long_temps_de_vol(N):  
    '''Renvoie le plus petit entier  $0 < a \leq N$  tel que le temps de vol pour  $a$   
    ↪ soit maximal  
    par rapport aux temps de vol des autres entiers  $\leq N$ '''  
    ## BEGIN SOLUTION  
    a_max = 1
```

```
t_max = 0
for a in range(2,N+1):
    t = temps_de_vol(a)
    if t > t_max:
        t_max = t
        a_max = a
return a_max
## END SOLUTION
```

```
[21]: from nose.tools import assert_equal

assert_equal(plus_long_temps_de_vol(1), 1)
assert_equal(plus_long_temps_de_vol(12), 9)
assert_equal(plus_long_temps_de_vol(50), 27)
```