2. Structures de contrôle avec Sage

2.1. Boucles

Boucle `for` (pour)

Pour faire varier un élément dans un ensemble, on utilise l'instruction "`for x in ensemble:`". Le bloc d'instructions qui suit sera alors successivement exécuté pour toutes les valeurs de x.

Code 1 (*structures.sage (1)*).
```
for x in ensemble:
    première ligne de la boucle
    deuxième ligne de la boucle
    ...
    dernière ligne de la boucle
instructions suivantes
```

Notez encore une fois que le bloc d'instructions exécuté est délimité par les espaces en début de ligne. Un exemple fréquent d'utilisation est "`for k in range(n):`" qui fait varier k de 0 à $n-1$.

Liste `range` (intervalle)

En fait `range(n)` renvoie la liste des n premiers entiers : `[0,1,2,...,n-1]`
Plus généralement `range(a,b)` renvoie la liste `[a,a+1,...,b-1]` d'entiers consécutifs alors que `range(a,b,c)` effectue une saut de c termes, par exemple `range(0,101,5)` renvoie la liste `[0,5,10,15,...,100]`.
Une autre façon légèrement différente pour définir des listes d'entiers est l'instruction `[a..b]` qui renvoie la liste des entiers k tels que $a \leqslant k \leqslant b$. Par exemple après l'instruction `for k in [-7..12]:` l'entier k va prendre successivement les valeurs -7, -6, -5, ..., -1, 0, $+1$,... jusqu'à $+12$.
La boucle `for` permet plus généralement de parcourir n'importe quelle liste, par exemple `[0.13,0.31,0.53,0.98]`. L'instruction `for x in [0.13,0.31,0.53,0.98]:` fera prendre à x les 4 valeurs de $\{0.13, 0.31, 0.53, 0.98\}$.

Boucle `while` (tant que)

Code 2 (*structures.sage (2)*).
```
while condition:
    première ligne de la boucle
    deuxième ligne de la boucle
    ...
    dernière ligne de la boucle
instructions suivantes
```

La boucle `while` exécute un bloc d'instructions tant que l'expression `condition` est vérifiée. Lorsqu'elle ne l'est plus, on passe aux instructions suivantes.
Voici, pour illustrer, le calcul de la racine carrée entière d'un entier n :

Code 3 (*structures.sage (3)*).
```
n = 123456789      # l'entier dont on cherche la racine carrée entière
k = 1              # le premier candidat
while k*k <= n:    # tant que le carré de k ne dépasse pas n
    k = k+1        # on passe au candidat suivant
print(k-1)         # la racine entière cherchée
```

Lorsque la recherche est terminée, k est le plus petit entier dont le carré dépasse n, par conséquent la racine carrée entière de n est $k-1$.
Notez qu'utiliser une boucle « tant que » comporte des risques, en effet il faut toujours s'assurer que la boucle se termine.

Test `if... else` (si... sinon)

Code 4 (*structures.sage (4)*).
```
if condition:
    première ligne du premier bloc
    ...
    dernière ligne du premier bloc
else:
    première ligne du second bloc
    ...
    dernière ligne du second bloc
instructions suivantes
```

Si la condition est vérifiée, c'est le premier bloc d'instructions qui est exécuté, si la condition n'est pas vérifiée, c'est le second. On passe ensuite aux instructions suivantes.

2.2. Booléens et conditions

Booléens

Une ***expression booléenne*** est une expression qui peut seulement prendre les valeurs « Vrai » ou « Faux » et qui sont codées par `True` ou `False`. Les expressions booléennes s'obtiennent principalement par des conditions.

Quelques conditions

Voici une liste de conditions :
- `a < b` : teste l'inégalité stricte $a < b$,
- `a > b` : teste l'inégalité stricte $a > b$,
- `a <= b` : teste l'inégalité large $a \leqslant b$,
- `a >= b` : teste l'inégalité large $a \geqslant b$,
- `a == b` : teste l'égalité $a = b$,
- `a <> b` (ou `a != b`) : teste la non égalité $a \neq b$.
- `a in B` : teste si l'élément a appartient à B.

Remarque. 1. Une condition prend la valeur `True` si elle est vraie et `False` sinon. Par exemple `x == 2` renvoie `True` si x vaut 2 et `False` sinon. La valeur de x n'est pas modifiée pas le test.

2. Il ne faut surtout pas confondre le test d'égalité `x == 2` avec l'affectation `x = 2` (après cette dernière instruction x vaut 2).

3. On peut combiner des conditions avec les opérateurs `and`, `or`, `not`. Par exemple : `(n>0) and (not (is_prime(n)))` est vraie si et seulement si n est strictement positif et non premier.

Travaux pratiques 1.

1. Pour deux assertions logiques P et Q, écrire l'assertion $P \implies Q$.
2. Une ***tautologie*** est une assertion vraie quelles que soient les valeurs des paramètres, par exemple (P ou (non P)) est vraie que l'assertion P soit vraie ou fausse. Montrer que l'assertion suivante est une tautologie :

$$\text{non}\Big(\big[\text{non } (P \text{ et } Q) \text{ et } (Q \text{ ou } R)\big] \text{ et } \big[P \text{ et (non } R)\big]\Big)$$

Il faut se souvenir du cours de logique qui définit l'assertion « $P \implies Q$ » comme étant « non (P) ou Q ». Il suffit donc de renvoyer : `not(P) or Q`.
Pour l'examen de la tautologie, on teste toutes les possibilités et on vérifie que le résultat est vrai dans chaque cas !

Code 5 (*structures.sage (10)*).
```
for P in {True, False}:
    for Q in {True, False}:
        for R in {True, False}:
            print(not((not(P and Q) and (Q or R)) and (P and (not R))))
```

2.3. Fonctions informatiques

Une fonction informatique prend en entrée un ou plusieurs paramètres et renvoie un ou plusieurs résultats.

Code 6 (*structures.sage (5)*).
```
def mafonction (mesvariables):
    première ligne
    ...
    dernière ligne
    return monresultat
```

Par exemple, voici comment définir une fonction valeur absolue.

Code 7 (*structures.sage (6)*).
```
def valeur_absolue(x):
    if x >= 0:
        return x
    else:
        return -x
```

Dans ce cas, `valeur_absolue(-3)` renvoie +3.
Il est possible d'utiliser cette fonction dans le reste du programme ou dans une autre fonction. Notez aussi qu'une fonction peut prendre plusieurs paramètres.
Par exemple, que fait la fonction suivante ? Quel nom aurait été plus approprié ?

Code 8 (*structures.sage (7)*).
```
def ma_fonction(x,y):
    resultat = (x+y+valeur_absolue(x-y))/2
    return resultat
```

Travaux pratiques 2.

On considère trois réels distincts a, b et c, ainsi que le polynôme défini par :

$$P(X) = \frac{(X-a)(X-b)}{(c-a)(c-b)} + \frac{(X-b)(X-c)}{(a-b)(a-c)} + \frac{(X-a)(X-c)}{(b-a)(b-c)} - 1$$

1. Définir trois variables par `var('a,b,c')`. Définir une fonction `polynome(x)` qui renvoie $P(x)$.
2. Calculer $P(a)$, $P(b)$, $P(c)$.
3. Comment se fait-il qu'un polynôme de degré 2 ait 3 racines distinctes ? Expliquez ! (Vous pourrez appliquer la méthode `full_simplify()` à votre fonction.)

2.4. Variable locale/variable globale

Il faut bien faire la distinction entre les variables locales et les variables globales.
Par exemple, qu'affiche l'instruction `print(x)` dans ce petit programme ?

Code 9 (*structures.sage (9)*).

```
x = 2
def incremente(x):
    x = x+1
    return x
incremente(x)
print(x)
```

Il faut bien comprendre que tous les `x` de la fonction `incremente` représentent une variable locale qui n'existe que dans cette fonction et disparaît en dehors. On aurait tout aussi bien pu définir `def incremente(y): y = y+1 return y` ou bien utiliser la variable `truc` ou `zut`. Par contre, les `x` des lignes `x=2`, `incremente(x)`, `print(x)` correspondent à une variable globale. Une variable globale existe partout (sauf dans les fonctions où une variable locale porte le même nom !).
En mathématique, on parle plutôt de variable muette au lieu de variable locale, par exemple dans

$$\sum_{k=0}^{n} k^2$$

k est une variable muette, on aurait pu tout aussi bien la nommer i ou x. Par contre, il faut au préalable avoir défini n (comme une variable globale), par exemple « $n = 7$ » ou « fixons un entier n »...

2.5. Conjecture de Syracuse

Mettez immédiatement en pratique les structures de contrôle avec le travail suivant.

Travaux pratiques 3.
La ***suite de Syracuse*** de terme initial $u_0 \in \mathbb{N}^*$ est définie par récurrence pour $n \geqslant 0$ par :

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

1. Écrire une fonction qui, à partir de u_0 et de n, calcule le terme u_n de la suite de Syracuse.
2. Vérifier pour différentes valeurs du terme initial u_0, que la suite de Syracuse atteint la valeur 1 au bout d'un certain rang (puis devient périodique : ..., 1, 4, 2, 1, 4, 2, ...). Écrire une fonction qui pour un certain choix de u_0 renvoie le plus petit entier n tel que $u_n = 1$.

Personne ne sait à ce jour prouver que pour toutes les valeurs du terme initial u_0, la suite de Syracuse atteint toujours la valeur 1.
Voici le code pour calculer le n-ème terme de la suite.

Code 10 (*suite-syracuse.sage (1)*).

```
def syracuse(u0,n):
    u = u0
```

```
    for k in range(n):
        if u%2 == 0:
            u = u//2
        else:
            u = 3*u+1
    return u
```

Quelques remarques : on a utilisé une boucle `for` pour calculer les n premiers termes de la suite ainsi qu'un test `if...else`.
L'instruction `u%2` renvoie le reste de la division de u par 2. Le booléen `u%2 == 0` teste donc la parité de u. L'instruction `u//2` renvoie le quotient de la division euclidienne de u par 2.

Remarque.
Si $a \in \mathbb{Z}$ et $n \in \mathbb{N}^*$ alors

- `a%n` est le ***reste*** de la division euclidienne de a par n. C'est donc l'entier r tel que $r \equiv a \pmod{n}$ et $0 \leqslant r < n$.
- `a//n` est le ***quotient*** de la division euclidienne de a par n. C'est donc l'entier q tel que $a = nq + r$ et $0 \leqslant r < n$.

Il ne faut pas confondre la division de nombres réels : `a/b` et le quotient de la division euclidienne de deux entiers : `a//b`. Par exemple `7/2` est la fraction $\frac{7}{2}$ dont la valeur numérique est $3,5$ alors que `7//2` renvoie 3. On a `7%2` qui vaut 1. On a bien $7 = 2 \times 3 + 1$.
Dans le cadre de l'exercice, `u%2` renvoie le reste de la division euclidienne de u par 2. Celui-ci vaut 0 si u est pair et 1 si u est impair. Ainsi `u%2 == 0` teste si u est pair ou impair. Ensuite `u = u//2` divise u par 2 (en fait cette opération n'est effectuée que pour u pair).

Pour calculer à partir de quel rang on obtient la valeur 1, on utilise une boucle `while` dont les instructions sont exécutées tant que $u \neq 1$. Par construction, lorsque la boucle se termine c'est que la valeuur de u est 1.

Code 11 (*suite-syracuse.sage (2)*)**.**

```
def vol_syracuse(u0):
    u = u0
    n = 0
    while u <> 1:
        n = n+1
        if u%2 == 0:
            u = u//2
        else:
            u = 3*u+1
    return n
```