Chapitre 5 : Boucle while et documentation

Instruction introduite dans ce cours : `while`.

1 Principe et syntaxe

Tout comme la boucle for, la boucle while est une structure itérative. Elle permet donc de répéter un certain nombre de fois une (ou plusieurs) action. Dans une boucle for, le nombre de répétitions est fixé à l'avance alors que dans une boucle while, l'action est répétée tant qu'une certaine condition est vérifiée. La boucle while s'exprime donc intuitivement de la manière suivante :

Tant que *Condition* Faire *Action* .

La syntaxe Python pour les boucles while est donc la suivante :

```
while Condition:
    Action
```

La syntaxe d'une boucle while est analogue à celle d'une boucle for : il faut utiliser les deux-points et l'indentation pour le bloc d'instructions à répéter.

Exercice 1.1.
On considère les deux suites de lignes de code suivants :

```
n=0
while 0<1:
    n=n+1
```

```
n=50
while n**2-2*n+2>0:
    n=n-1
```

Expliquons le fonctionnement de ceux-ci et les problèmes qu'ils posent.

Remarque 1.2.

2 Exemples d'utilisation

Dans cette section, nous allons voir trois utilisations possibles pour les boucles while.

2 A) Premier exemple

On peut toujours utiliser une boucle while à la place d'une boucle for. En ce sens, les boucles while sont plus générales. Considérons le code générique suivant :

```
for k in range(n):
        Action(k)
```

On peut obtenir un code réalisant la même spécification en utilisant une boucle while : l'action est alors effectuée tant que k est inférieur strictement à n. Il ne faut alors pas oublier d'initialiser k avant la boucle et de l'incrémenter à la fin de la boucle (l'incrémentation de la variable de contrôle k est automatique dans une boucle for, mais pas dans une boucle while). On obtient alors le code suivant :

```
k=0
while k<n:
        Action(k)
        k=k+1
```

Exercice 2.1.
Écrire avec une boucle for puis une boucle while un programme `multiples_2(n)` affichant les multiples de 2 de 0 à $2n$ où n est un entier pris comme argument.

2 B) Second exemple

Lorsqu'on a une boucle for, mais qu'on souhaite la terminer avant la fin de son exécution, on peut utiliser la commande `break`. Mais on peut également remplacer cette boucle for par une boucle while, ce qui permet d'avoir un code finalement plus lisible.

Considérons l'exemple rencontré lors du cours n° 3. On souhaite écrire une fonction qui prend en entrée une liste et qui renvoie l'indice du premier 0 de cette liste ou `False` si 0 n'est pas un élément de cette liste. La première des fonctions ci-dessous renvoie le résultat désiré à l'aide d'une boucle for et d'une instruction break.

```
def premier_zero1(L):
        n=len(L)
        y=False
        for k in range(n):
                if L[k]==0:
                        y=k
                        break
        return(y)
```

Écrivons une fonction réalisant la même chose avec une boucle while :

2 C) Troisième exemple

Ce troisième exemple d'utilisation est sûrement celui que vous rencontrerez le plus : on fait une boucle while quand on ne connaît pas à l'avance combien de fois on devra répéter les actions de la boucle.

Considérons par exemple la fonction qui prend en entrée un entier naturel non nul N et qui renvoie le nombre k de chiffres en base 10 de cet entier. Pour déterminer le nombre de chiffres de N, il faut déterminer l'entier k tel que $10^{k-1} \leqslant N < 10^k$. Autrement dit, k est le plus petit entier tel que $N < 10^k$. Une façon de procéder est de prendre successivement $k = 0, 1, 2, \ldots$ et de renvoyer la première valeur de k qui vérifie l'inégalité $N < 10^k$, c'est-à-dire qui ne vérifie pas l'inégalité $N \geqslant 10^k$. On obtient ainsi le code suivant :

```
def NombreChiffres(N):
        k=0
        while N>=10**k:
                k=k+1
        return(k)
```

```
>>> NombreChiffres(48752)
5
```

3 Recherche d'une valeur approchée de la limite d'une suite

Les boucles while seront couramment utilisées en ingénierie numérique pour déterminer une valeur approchée de la limite ℓ d'une suite $(u_n)_{n\in\mathbb{N}}$.

La recherche d'une telle valeur approchée suppose la donnée de la précision $\varepsilon > 0$ souhaitée. Pour déterminer une valeur approchée à ε près de la limite de la suite $(u_n)_{n\in\mathbb{N}}$, il suffit de calculer u_k, où k est un entier tel que $|u_k - \ell| \leqslant \varepsilon$. Mais comme a priori la valeur de ℓ nous est inconnue, cette condition est difficile à vérifier numériquement. Pour contourner ce problème, on doit connaître une estimation de $|u_n - \ell|$: l'erreur entre un terme de la suite et la limite de cette suite. En général, cette erreur sera donnée par une inégalité de la forme :

$$\forall n \in \mathbb{N},\ |u_n - \ell| \leqslant M_n,$$

où $(M_n)_{n\in\mathbb{N}}$ est une suite de réels positifs décroissante et qui tend vers 0. En connaissant la suite $(M_n)_{n\in\mathbb{N}}$, il suffit de déterminer un entier k tel que $M_k \leqslant \varepsilon$. On aura alors :

$$|u_k - \ell| \leqslant M_k \leqslant \varepsilon.$$

La valeur approchée souhaitée sera alors u_k. La fonction suivante prend en entrée la suite u (donnée sous python par une fonction), la suite M et la précision ε souhaitée.

```
def ValeurApprocheeLimite(u,M,epsilon):
        k=0
        while M(k)>epsilon:
                k=k+1
        return(u(k))
```

Bien évidemment, il faudra adapter cet algorithme général à la situation considérée.

Exemple 3.1.

Considérons la suite définie par $u_0 = 2$ et $\forall n \in \mathbb{N},\ u_{n+1} = \frac{1}{2}\left(u_n + \frac{5}{u_n}\right)$. On peut montrer que la suite $(u_n)_{n\in\mathbb{N}}$ tend vers $\sqrt{5}$ et que $\forall n \in \mathbb{N},\ 0 < u_n - \sqrt{5} < \frac{19}{4}\left(\frac{1}{17}\right)^{2^n}$. La fonction Pyhton suivante calcule une valeur approchée de $\sqrt{5}$ à ε près :

```
def u(n):
        x=2
        for k in range(n):
                x=(x+5/x)/2
        return(x)
```

```
def ValeurApprocheeLimite(epsilon):
        k=0
        while 19/4*(1/17)**(2**k)>epsilon:
                k=k+1
        return(u(k))
```

```
>>> ValeurApprocheeLimite(10**(-8))
2.2360679779158037
```

Expliquons pourquoi cet algorithme termine.

Remarque 3.2. Oon peut améliorer cet algorithme en calculant directement u_k à chaque étape de la boucle : on réunit ainsi les deux boucles en une seule.

```
def ValeurApprocheeLimite2(epsilon):
        u=2
        k=0
        while 19/4*(1/17)**(2**k)>epsilon:
                u=(u+5/u)/2
                k=k+1
        return(u)
```

4 Deux exemples

Dans les deux exemples suivants, on utilisera une boucle `while`.

Exercice 4.1. Écrire une fonction `croissant(L)` qui teste si les éléments d'une liste L sont rangés en ordre croissant.

Exercice 4.2. Écrire une fonction `premier(p)` qui teste si un nombre entier strictement positif p est un nombre premier.

5 Documentation des fonctions

Quand on déclare une fonction destinée à être réutilisée par la suite, il est indispensable de la *documenter* correctement pour que l'utilisateur futur puisse en comprendre l'usage (sans avoir besoin de se plonger dans le code source...). Ainsi, toutes les primitives existant sous Python sont documentées dans l'aide du logiciel, et on a accès à la documentation d'une fonction `f` en entrant `help(f)` dans la console.
La documentation d'une fonction doit nécessairement préciser les points suivants :

- quel est le type de chacun des arguments d'entrée de la fonction ;
- quel est le type de l'argument de sortie de la fonction ;
- quelle est la *spécification* de la fonction (c'est-à-dire comment la valeur renvoyée en sortie est-elle calculée en fonction des valeurs d'entrée).

D'autre part, la documentation d'une fonction contient fréquemment un ou plusieurs exemples d'utilisation de celle-ci, qui sont parfois plus parlant que la partie décrite ci-dessus.

Exemple 5.1.
Voici la documentation de la fonction `len`.

```
>>> help(len)
Help on built-in function len in module builtins:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or mapping.
```

Lorsque nous déclarerons une fonction sous Python, il nous sera possible d'insérer dans cette déclaration la documentation de la fonction, et celle-ci sera ensuite accessible à partir de la commande `help`. Il suffit pour cela d'employer la suite de caractères `"""` pour ouvrir et clore la documentation.

Exemple 5.2.

```
def puissance(x,n):
    """
    premier argument : x est de type float
    deuxieme argument : n est de type int
    valeur de sortie : de type float
    specification : puissance(x,n) calcule
    x a la puissance n
    """
    return(x**n)
```

```
>>> help(puissance)
Help on function puissance in module
__main__:

puissance(x, n)
    premier argument : x est de type float
    deuxieme argument : n est de type int
    valeur de sortie : de type float
    specification : puissance(x,n) calcule
    x a la puissance n
```

Bien entendu il n'y a aucun intérêt à documenter une fonction aussi simple, mais dès que l'on commence à faire des choses compliquées ou à travailler en équipe, il devient nécessaire de documenter ses fonctions, et particulièrement les fonctions « auxiliaires » qui sont invisibles à l'utilisateur et dont la spécification n'est pas toujours claire.

Plus généralement, on prendra désormais l'habitude de commenter ces programmes, afin d'expliquer simplement ce qu'ils font. Cela sera bien entendu utile pour le correcteur, mais également pour vous si vous souhaitez réutiliser ceux-ci ultérieurement.

Exercice 5.3. Écrire une fonction `nb_0_1(L)` qui prend en entrée une liste `L`, et qui renvoie un tuple (n, p) tel que n (resp. p) soit le nombre de `0` (resp. `1`) apparaissant dans `L`. On documentera cette fonction.