Chapitre 2 : Structures itératives

Instructions introduites dans ce cours : `for, in, range, //, \%`.

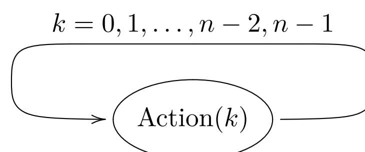
L'une des tâches dont s'acquittent le mieux les machines est l'exécution à grande fréquence et sans erreur de tâches répétitives. Il existe bien entendu différentes méthodes pour programmer l'exécution de ces tâches, et nous allons commencer par la plus courante d'entre elles : la structure itérative ou *boucle for*.

1 Principe

La *structure itérative* est un procédé informatique analogue au raisonnement par récurrence en mathématiques. Elle ordonne à la machine d'opérer à la suite et dans l'ordre n actions, que nous appellerons Action(0), Action(1), ..., Action($n-1$). Elle s'exprime intuitivement de la manière suivante :

Pour k allant de 0 à $n-1$ faire Action(k).

Cette structure est communément appelée une **boucle**, car sa représentation la plus simple est celle d'une boucle (symbolisant le sens d'évaluation du programme) autour d'une bulle figurant les actions considérées.



Lorsque la machine évalue la ligne de code *Pour k allant de* 0 *à* $n-1$ *faire Action*(k), alors la variable `k` prend successivement les valeurs $0, 1, ..., n-2, n-1$, et pour chacune d'elles la machine exécute Action(k).

Remarque 1.1.
Les n actions effectuées sont notées de 0 à $n-1$ et non pas de 1 à n (convention courante en informatique).

2 Syntaxe

La syntaxe Python correspondant à l'expression *Pour k allant de* 0 *à* $n-1$ *faire Action*(k) est la suivante :

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

Il faut immédiatement remarquer la présence d'une indentation à la deuxième ligne du code ci-dessus : on a affaire à un bloc d'instructions, donc une indentation devra obligatoirement précéder l'action `Action(k)` que l'on souhaite répéter (sinon l'exécution du programme renverra un message d'erreur). Si on souhaite inclure cette structure dans un programme plus grand, il faudra après `Action(k)` revenir à la ligne sans indentation pour indiquer à la machine que l'on souhaite sortir de la boucle et continuer l'écriture du programme.

La signification de l'instruction `range` sera détaillée ultérieurement.

Remarque 2.1.
Dans la syntaxe donnée ci-dessus, la variable `k` n'a aucun sens précis : on peut remplacer `k` par n'importe quelle autre variable, par exemple `i`, mais bien entendu il faut alors changer `Action(k)` en `Action(i)`. D'autre part, si la variable `k` contient une valeur avant l'évaluation de la boucle, celle-ci est écrasée lorsque la boucle débute.

3 Premiers exemples : affichages à l'écran

3 A) Entiers, flottants et chaînes de caractères

Pour le moment, en Python, nous avons uniquement utilisé des opérations sur les nombres décimaux (addition, différence, produit, puissance, quotient par exemple).

De manière précise, Python peut manipuler des entiers et des *flottants* (nous verrons la définition précise de la notion de flottants plus tard dans l'année). Les entiers et les flottants ne sont pas du même *type* ou *classe* en Python. On peut connaître le type d'un objet en Python avec la fonction `type` :

```
>>> type(2)
<class 'int'>
>>> type(2.0)
<class 'float'>
```

La langage Python est un langage à *typage dynamique*, c'est à dire que le type d'un objet peut changer lors de sa manipulation.

Exemple 3.1. Que fait de manière précise Python lorsqu'on entre dans la console :

```
>>> 2+2.0
```

Il est possible de convertir des objets d'un type à l'autre avec : la fonction `float` convertit en flottant et `int` convertit en entier.

Exemple 3.2. On donne les lignes suivantes :

```
>>> int(2.5)
2

>>> int(-2.5)
-2

>>> int(0)
0
```

Commentaires :

Un autre type d'objet en Python est le type *chaîne de caractères* (string en anglais) qu'on entre de la façon suivante :

```
>>> 'jules ferry'
```

On indique les caractères entre des apostrophes (aussi appelés (à tort) guillemets ou simple quotes).

Exemple 3.3. Expliquer les lignes suivantes :

```
>>> 'julesferry'+'PTSI'
'julesferryPTSI'

>>> 'julesferry'*'PTSI'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'

>>> 'julesferry'*2
'julesferryjulesferry'

>>> 2+'julesferry'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
>>> str(2)+'julesferry'
'2julesferry'

>>> int('julesferry')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'julesferry'

>>> float('julesferry')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: could not convert string to float: 'julesferry'
```

Commentaires :

Remarque 3.4. On ne confondra surtout pas une variable `julesferry` et la chaîne de caractères `'julesferry'`. Cette partie est une introduction brève aux chaînes de caractères qui seront étudiées plus précisément dans le cours suivant.

3 B) Affichages répétés

Nous pouvons maintenant implémenter un exemple de fonction qui affiche n fois « bonjour » à l'écran : il suffit pour cela de répéter n fois l'instruction `print('bonjour')`, ce qui se fait avec une boucle for :

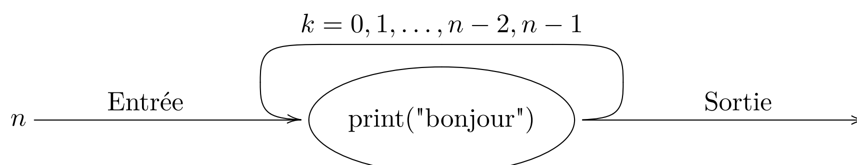
Pour k *allant de* 0 *à* $n-1$ *faire* `print('bonjour')`.

En utilisant la syntaxe Python, cela donne la déclaration suivante :

```
def ecrire(n):
    for k in range(n):
        print('bonjour')
```

```
>>> ecrire(3)
bonjour
bonjour
bonjour
```

Le graphe suivant représente l'évaluation du programme ci-dessus par la machine.



Remarque 3.5.
Si l'on souhaite afficher *bonjour* n fois à l'écran, il suffit d'entrer n fois la commande `print('bonjour')`. Néanmoins, l'entier n n'est pas fixé dans le problème qui nous occupe, puisque c'est l'argument d'entrée de la fonction : nous sommes donc obligés d'employer une boucle for, car on ne peut écrire n fois `print('bonjour')`. Bien entendu, une alternative serait d'écrire deux fois `print('bonjour')`, et de séparer les deux par des points de suspension... si un lecteur attentif comprendrait peut-être ce que cela signifie, cela n'a aucun sens pour une machine. On pourra garder à l'esprit cette idée un peu naïve : on emploie des boucles for pour éviter d'avoir recours à des points de suspension.

3 C) La table de 2

Dans l'exemple précédent, l'action `print('bonjour')` sur laquelle portait la boucle for ne dépendait pas de l'entier k : en d'autres termes, on répétait k fois la même action. Considérons maintenant un exemple un peu plus évolué, dans lequel l'action Action(k) sur laquelle porte la boucle dépend de l'entier k.

Cherchons à implémenter une fonction `table` qui prend en entrée un entier n, puis qui affiche à l'écran les n premiers multiples de 2 : l'appel de `table(n)` fera donc apparaître à l'écran la suite $0, 2, 4, 6, \dots, 2(n-1)$. Il s'agit ici d'utiliser simplement la boucle for suivante :

Pour k *allant de* 0 *à* $n-1$ *faire* `print(2*k)`.

En syntaxe Python, cela donne le programme suivant :

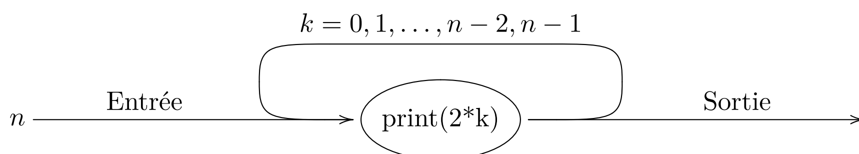
```
def table(n):
    for k in range(n):
        print(2*k)
```

```
>>> table(3)
0
2
4
```

Les différentes étapes de l'évaluation de ce programme par la machine sont les suivantes :

— à l'étape 1, on a $k = 0$ et donc le processeur exécute l'action `print(2*0)` ;
— à l'étape 2, on a $k = 1$ et donc le processeur exécute l'action `print(2*1)` ;
...
— à l'étape n, on a $k = n - 1$ et donc le processeur exécute l'action `print(2*(n-1))` ;

On peut également représenter l'exécution par le graphe suivant :



Exemple 3.6.
On souhaite construire une fonction `multiple(n,p)` qui prend en entrée deux entiers p et n, et qui affiche les n premiers multiples de p. On propose le code suivant :

```
def multiple(n,p):
    for k in range(n):
        return(p*k)
```

Expliquer pourquoi ce code ne fonctionne pas et écrire une fonction qui donne le bon résultat.

Remarque 3.7.
De façon générale, rien n'impose au code d'une fonction de se terminer par `return` : il est parfois plus commode de faire apparaître `return` dans une boucle ou dans les deux cas d'un `if`. Néanmoins, cela n'est pas conseillé car l'appel d'une fonction est systématiquement interrompu au moment où une instruction `return` est exécutée (même s'il y a d'autres instructions ensuite).

Exercice 3.8.
Que fait la fonction ci-dessous ?

```
def table(n):
    for k in range(n):
        print('2*k')
```

4 Second exemple : les suites récurrentes

4 A) Une suite arithmétique

On considère dans cette section la suite arithmétique $(u_k)_{k\in\mathbb{N}}$ de raison r et de premier terme a, définie par $u_0 = a$ et $\forall k \geqslant 0,\ u_{k+1} = u_k + r$.

Pour calculer le n-ème terme u_n de la suite, on commence donc par calculer $u_0 = a$. On calcule alors u_1 en écrivant $u_1 = u_0 + r$, puis on calcule u_2 en écrivant $u_2 = u_1 + r$, puis u_3 en écrivant $u_3 = u_2 + r$,... Au bout de n étapes on arrive donc à la valeur de u_n, qui est égale à $u_n = a + nr$.

Dans la suite nous allons écrire un programme qui prend en entrée un entier n puis renvoie u_n. Bien entendu c'est totalement trivial si on utilise la formule $u_n = a + nr$, le programme suivant convenant alors :

```
def suite_arithmetique(n):
    return(a+n*r)
```

Remarque 4.1.
Dans ce programme, `a` et `r` sont des variables globales, qui doivent donc être affectées avant de lancer le programme : sinon son exécution renverra un message d'erreur.

Dans la suite nous allons faire comme si nous ne connaissions pas cette formule : on cherche donc à effectuer successivement les n opérations décrites ci-dessus, et pour cela nous allons utiliser une boucle for. Cela peut vous sembler artificiel, mais vous verrez rapidement que la méthode mise en œuvre sur cet exemple nous sera par la suite fréquemment utile.

Le principe de l'algorithme est le suivant : partant d'une variable u valant a nous allons, pour k allant de 0 à $n-1$, lui ajouter r. En somme, on ne fait que mettre en œuvre la définition de la suite. De manière informelle cela donne l'algorithme ci-contre.

> $u_0 = a$
> Pour k allant de 0 à $n-1$ faire
> $u_{k+1} = u_k + r$
> Renvoyer u_n

Lorsque la machine évaluera ce programme, on calculera successivement les $n+1$ valeurs suivantes, la dernière étant le résultat : $u_0, u_1, \ldots, u_n$. Mais on remarque alors que la machine va conserver en mémoire toutes les valeurs calculées, ce qui d'une part est inutile, et d'autre part peut poser des problèmes (dans certains cas on peut se retrouver à court de mémoire. . .).

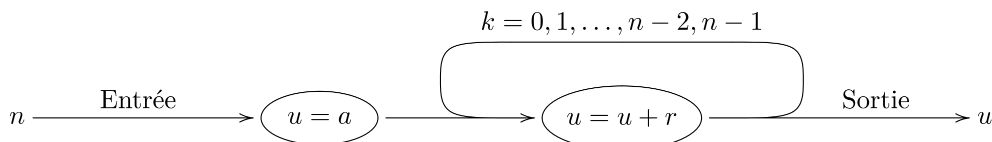
Nous pouvons donc modifier sensiblement le programme, en effaçant toutes les références à l'indice k : on n'utilisera ainsi qu'une seule case mémoire u, qui ne contiendra donc à chaque étape que le dernier terme calculé. On obtient ainsi l'algorithme ci-contre.

> $u = a$
> Pour k allant de 0 à $n-1$ faire
> $u = u + r$
> Renvoyer u

Dans l'exemple qui nous occupe, on obtient donc le code ci-dessous en traduisant le programme précédent dans le langage Python :

```
def suite_aritm2(n):
    u=a
    for k in range(n):
        u=u+r
    return(u)
```

Lorsque la machine évaluera ce programme, la variable u prendra successivement les $n+1$ valeurs suivantes, la dernière étant le résultat : $u_0, u_1, \ldots, u_n$. On dit que la ligne $u = a$ constitue *l'initialisation* de la variable u. Le graphe suivant représente l'évaluation du programme ci-dessus par la machine.



Exercice 4.2.

1. Écrire une fonction qui prend en entrée un entier n, puis calcule la somme des n premiers entiers non nuls.
2. Montrer que pour tout $n \in \mathbb{N}$:
$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2}.$$
La fonction précédente peut-elle être simplifiée ?

4 B) Généralisation

Les boucles for sont le plus souvent utilisées pour construire des objets par étapes ou couches, par exemple le n-ième terme d'une suite arithmétique. Plus généralement, si on considère une suite récurrente, c'est-à-dire une suite définie par une relation de la forme :

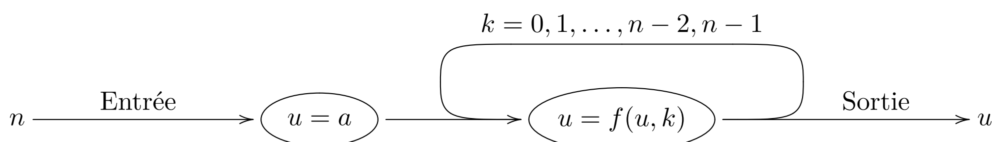
$$\begin{cases} u_0 = x \\ \forall k \in \mathbb{N},\ u_{k+1} = f(u_k, k) \end{cases}$$

alors le programme suivant permet de calculer le n-ième terme de celle-ci, après avoir pris en entrée n, x et f :

```
def suite_terme(x,f,n):
    u=x                       ← Initialisation : on pose u_0 = a
    for k in range(n):
        u=f(u,k)              ← Hérédité : on pose u_{k+1} = f(u_k, k), pour k allant de 0 à n − 1.
    return(u)
```

Il est important de noter ici que l'écriture du programme ressemble à la définition de la suite ; on pourrait presque dire que c'en est une traduction, du langage mathématique vers le langage Python.

Lorsque la machine évaluera ce programme, la variable u prendra successivement les $n+1$ valeurs suivantes, la dernière étant le résultat : $u_0, u_1, \ldots, u_n$. Le graphe suivant représente l'évaluation du programme ci-dessus par la machine.



Exercice 4.3.

Dans cet exercice, on cherche à calculer $u_p = \sum_{n=0}^{p} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{p!}$ où, pour tout $0! = 1$ et, pour tout $n \in \mathbb{N}^*$, $n! = 1 \times 2 \times \cdots \times n$.

1. Écrire la fonction factorielle, c'est-à-dire la fonction qui prend en entrée un entier n puis renvoie le produit des entiers inférieurs à n, soit $n \times (n-1) \times \cdots 2 \times 1$. On prendra encore la convention suivante : factorielle$(0) = 1$.
2. Donner un programme `Somme` convenant, en réutilisant la fonction implémentée dans l'exercice précédent.
3. Lors de l'appel de `Somme(p)`, pouvez-vous compter combien d'opérations arithmétiques effectue la machine ? Pouvez-vous trouver un programme qui effectue le calcul de u_p en effectuant moins d'opérations ?