Instructions conditionnelles et boucles

7 juillet 2016

1 L'instruction conditionnelle

1.1 Algorithme

Quand on veut écrire un programme, on souhaite établir des connections logiques entre les instructions. Ainsi, l'instruction conditionnelle a pour objet d'intervenir dans le choix de l'instruction suivante en fonction d'une expression booléenne qu'on désignera par **condition** :

Si condition
    alors bloc d'instructions 1
    sinon bloc d'instructions 2
Fin-du-Si

signifie que

— **si** la condition est vérifiée (expression booléenne=`True`)
alors le programme exécute les instructions du bloc 1

— **si** la condition n'est pas vérifiée (expression booléenne=`False`)
alors le programme exécute les instructions du bloc 2

1.2 Syntaxe en `Python`

```
if condition :
        bloc d'instructions 1
else :
        bloc d'instructions 2
```

— **Si** et **Sinon** se traduisent par `if` et `else`.

— **alors** se traduit par « **:** » en bout de ligne et une indentation de toutes les lignes du bloc 1.

— **Fin-du-Si** se traduit par un retour à la ligne sans indentation.

1.3 Exemple

On veut tester si un nombre x est proche de 3 à 10^{-4} près.

```
>>> nombre = x
>>> distance  = abs(x-3) #la distance se mesure en valeur ↙
   absolue.
>>> if distance <= 10**(-4) :
        print("x est proche de 3 à 0.0001 près")
    else :
        print("x n'est pas proche de 3 à 0.0001 près")
```

Remarque 1.3.1. La partie **sinon** est optionnelle. Sans elle, si la condition n'est pas vérifiée, alors la machine n'exécute rien.

1.4 À propos des conditions

L'expression booléenne derrière le **si** joue le rôle de test. Pour exprimer cette condition, on a besoin des **opérateurs de comparaison** (inférieur strict, supérieur strict, inférieur ou égal, supérieur ou égal, égal à, différent de) et des **connecteurs logiques** (non, et, ou).

Exemples 1.4.1.

— Calcul du carré d'un nombre positif.

```
>>> x =  4
>>> if x>= 0 :
        car  = x**2
>>> car
16              # La réponse est 16.
```

— Comprenez-vous le message d'erreur ?

```
>>> x = -5
>>> if x>= 0 :
        carr  = x**2
>>> carr
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    carr
NameError: name 'carr' is not defined
```

— Condition avec un “ et ”.

```
>>> x = 0.5
>>> if x >= -1 and x<= 1 :
        print("Il existe un angle theta tel que x = ↙
           cos(theta).")

Il existe un angle theta tel que x = cos(theta).
```

1.5 Imbrication de plusieurs conditions

On peut se trouver face à un problème qui se scinde en plus de deux cas (par exemple, dans le cas des équations du second degré, on teste si le discriminant est strictement positif, nul ou strictement négatif). Dans ce cas, voici comment procéder :

Si condition 1
alors bloc d'instructions 1
sinon si condition 2
alors bloc d'instructions 2
sinon si condition 3
alors bloc d'instructions 3
...
sinon bloc final
Fin-du-Si

signifie que

— **si** la condition 1 est vérifiée (expression booléenne=*True*)
alors le programme exécute les instructions du bloc 1

— **si** la condition 1 n'est pas vérifiée (expression booléenne=*False*) mais que la condition 2 est vérifiée
alors le programme exécute les instructions du bloc 2

— **si** les conditions 1 et 2 ne sont pas vérifiées mais que la condition 3 est vérifiée
alors le programme exécute les instructions du bloc 3...

— **si** aucune condition n'est vérifiée
alors le programme exécute les instructions du bloc final

Remarque 1.5.1. — Si la `condition 1` est vérifiée, le programme ne se préoccupe pas des conditions suivantes.

— Chaque **sinon si** agit comme un **filtre**. L'ordre de ces filtres est important. Ainsi, les séquences d'instructions :

```
x <- 3
Si x > 6
      alors renvoyer x + 2
      sinon si x > 4
                  alors renvoyer x + 4
                  sinon si x > 2
                              alors renvoyer x + 6
                              sinon renvoyer x
Fin-du-Si
```

et

```
x <- 3
Si x > 6
      alors renvoyer x + 2
      sinon si x > 2
                  alors renvoyer x + 6
                  sinon si x > 4
                              alors renvoyer x + 4
                              sinon renvoyer x
Fin-du-Si
```

ne renvoient pas la même chose.

1.6 Syntaxe en `Python`

```
if condition 1 :
        bloc d'instructions 1
elif condition 2 :
        bloc d'instructions 2
elif condition 3 :
        bloc d'instructions 3
.
.
.
else :
        bloc final
```

— **Sinon si** se traduit par `elif`.

Exercice 1.6.1. Écrire les deux séquences d'instructions de la remarque **1.5.1** en `Python`, et constater que l'on obtient bien deux résultats différents.

Exemple 1.6.2. Voici une fonction qui calcule le maximum de trois entiers `a, b, c` :

```
def max3 (a, b, c) :
```

```
""" renvoie le maximum de a, b ,c.
précondition : a, ,b et c sont 3 entiers """
assert type(a) == type(b) == type(c) == int, "les ↙
   arguments ne sont pas des entiers"
if a <= c and b <= c :
    m = c
elif a <= b and c < b :
    m = b
else :
    m = a
return m
```

2 Boucles définies

2.1 Un programme très simple

Écrivons un programme pour saluer la classe (disons les 8 premiers élèves) :

```
print('Bonjour Baptiste')
print('Bonjour Lisa')
print('Bonjour Pierrick')
print('Bonjour Louise-Eugénie')
print('Bonjour Qâsim')
print('Bonjour Lorenzo')
print('Bonjour Arthur')
print('Bonjour Ylies')
```

Mais combien de travail faut-il faire si :

— On veut dire bonsoir et non bonjour ?

— On veut dire «Baptiste, comment vas-tu ? *etc.*» et non «Bonjour Baptiste, *etc.*» ?

Et s'il y a 500 élèves ?

2.2 Le principe DRY

«**D**on't **R**epeat **Y**ourself» est une philosophie en programmation informatique consistant à éviter la redondance de code au travers de l'ensemble d'une application afin de faciliter la maintenance, le test, le débogage et les évolutions de cette dernière.
La philosophie DRY est explicitée par la phrase suivante :

> «Dans un système, toute connaissance doit avoir une représentation unique, non-ambiguë, faisant autorité ».

Ici le programme devrait :

— définir la liste des prénoms auxquels dire bonjour ;

— dire qu'on veut effectuer un même traitement sur tous les prénoms ;

— dire que ce traitement consiste à afficher «Bonjour» suivi du prénom.

Nous avons déjà vu comment définir une liste :

```
prenoms = [ 'Baptiste', 'Lisa', 'Pierrick',\
'Louise-Eugénie', 'Qâsim', 'Lorenzo', 'Arthur', 'Ylies' ]
```

Il faut maintenant s'occuper des deux points suivants : définir le traitement à effectuer sur chacun de ces prénoms, et l'**itérer**. Pour cela nous allons utiliser une **boucle itérative définie**, autrement dit nous allons **répéter** l'application d'une même séquences d'instructions sur une liste **définie à l'avance** :

Pour variable **dans** liste **répeter** bloc d'instructions b **Fin-de-la-boucle**

signifie que

pour chaque élément de la liste `liste`,
le programme exécute les instructions du bloc b.

2.3 Syntaxe en `Python`

```
for variable in liste :
    instructions
```

Ici encore, la ligne contenant le mot-clé **for** doit se finir par un « **:** » et les instructions du bloc doivent être indentées. La fin de la boucle est marquée par un retour à la ligne non indenté.

Maintenant que la liste `prenoms` est définie, dire bonjour à nos huit élèves s'écrit ainsi :

```
for x in prenoms:
    print('Bonjour ' + x)
```

2.4 Autre problème ...

On souhaite calculer u_{20} où u est la suite définie par

$$u_0 = 5$$
$$\forall n \in \mathbb{N}\ u_{n+1} = 2u_n - n - 3$$

Il y a deux approches possibles :

1. Résoudre ça mathématiquement, c'est-à-dire déterminer une expression de u_n en fonction de n, avant de faire une application numérique.
2. Calculer de proche en proche u_1, u_2, u_3,u_4 $\cdots$ et enfin u_{20}.

La première méthode est la meilleure si on sait faire.
La seconde promet d'être longue et pénible, surtout si l'on vous demande de calculer u_{10000}. Mais là encore la boucle itérative va nous aider.

On peut proposer un algorithme du type :

> x <- 5
> **Pour** k **de** 0 **à** *? ? ?* **répéter**
> x <- 2x - k -3
> **Fin-de-la-boucle**

et on espère qu'après cette boucle, x vaut u_{20}.

Mais il reste deux questions :

1. Y a t'il un lien entre cette boucle et celle vue précédemment où on parcourt les éléments d'une liste ?
2. Jusqu'où aller dans cette boucle *? ? ?* 18, 19, 20, 21, 22 ? Comment être sûr de ne pas se tromper ?

2.5 Les intervalles d'entiers en Python

Pour répondre à la première question, il suffit de remarquer que les entiers de 0 à 19 par exemple, sont en fait les éléments d'une liste : `[0, 1, 2, ··· , 19]`. En `Python`, cette liste s'écrit de la manière suivante : `range(20)`.
Précisément, si a et b sont deux entiers, `range(a,b)` contient les éléments de l'intervalle semi-ouvert $[\![a, b[\![$, dans l'ordre croissant. Avec un seul argument, `range(b)` signifie `range(0,b)`.
Redisons-le, car c'est un fait important en `Python` : `range(a,b)` est intervalle **fermé** à gauche, **ouvert** à droite [1].

Avec `range`, nous pouvons maintenant itérer sur une liste d'entiers :

```
x = 5
for k in range(20):
  x = 2 * x - k - 3
```

Résultat obtenu : 1048600.

Mais est-ce bien u_{20} ?

1. Voir *Why numbering should start at zero*, E. W. Dijkstra, EWD831. Disponible en ligne.

2.6 Les invariants de boucle

Un **invariant de boucle** est une proposition vérifiée à chaque tour d'une boucle. Plus précisément, on distinguer les invariants d'**entrée de boucle** et ceux de **sortie de boucle**. C'est en fait une proposition de récurrence, qui doit être vrai au début de la première boucle, puis se propager de tour de boucle en tour de boucle, jusqu'au dernier. Les invariants de boucles ont essentiellement deux intérêts :

1. **Démontrer** l'algorithme, grâce au principe de récurrence ;
2. Répondre à notre question précédente : où commencer et où finir une boucle ?

Il est **fortement conseillé** de toujours indiquer les invariants de boucle en commentaire dans vos algorithmes.

Par exemple :

```
x = 5                       # x = u_0 (initialisation)
for k in range(20):         # x = u_k (invariant en entrée de ↙
    boucle)
    x = 2 * x - k - 3       # x = u_{k+1} (invariant en sortie ↙
        de boucle)

# après la boucle for, k vaut 19, et x vaut donc u_{20}.
```

2.7 Un exemple : test de primalité

On veut tester si un entier `n` est premier :

```
def est_premier(n):
    """ Renvoie True si n est premier, False sinon
        Préconditon : n est un entier."""
    b = True
    for d in range(2,n):
        # b => n non divisible par 2, 3, ..., d-1.
        if n % d == 0:
            b = False
    # b <=> n premier
    return b
```

Remarque : les derniers tours de boucle sont inutiles dès que la variable `b` a été mise à `False`. Les éxécuter tout de même est une perte de temps. Il existe plusieurs possibilités pour améliorer cela :

L'instruction `break` :

```
def est_premier(n):
    """ Renvoie True si n est premier, False sinon
        Préconditon : n est un entier."""
```

```
    b = True
    for d in range(2,n):
        # b => n pas divisible par 2, 3, ..., d-1.
        if n % d == 0:
            b = False
            break
    # b <=> n est premier
    return b
```

L'utilisation d'un `return` en milieu de boucle, à favoriser en `Python` pour une question de style et d'élégance :

```
def est_premier(n):
    """ Renvoie True si n est premier, False sinon
        Préconditon : n est un entier."""
    for d in range(2,n):
        # n n'est pas divisible par 2, 3, ..., d-1.
        if n % d == 0:
            return False
    return True
```

3 Boucles indéfinies ou conditionnelles

3.1 Algorithme

On peut aussi être amené à répéter un bloc d'instructions sans savoir combien de fois on devra le répéter.

Exemple 3.1.1. Disposant d'une suite croissante, non majorée, on cherche à trouver le plus petit entier p tel que la valeur au rang p dépasse 10000.

Dans ce cas, on utilise la boucle **Tant que** qui permet de répéter le bloc d'instructions tant qu'une certaine condition est vérifiée.

	Tant que condition **faire** bloc d'instructions **Fin-du-Tant-que**
	signifie que
Tant que	la condition est vérifiée (expression booléenne=*True*)
Faire	le bloc d'instructions.

3.2 Syntaxe en `Python`

```
while condition :
    instructions
```

Exemple 3.2.1. Rechercher le premier entier n tel que la somme des entiers de 1 à n dépasse 11.

```
n = 1
s = 1
while s < 11 :
    n += 1
    s += n
return n
```

REPONSE : $n = 5$ (dans ce cas $s = 15$)

Exemple 3.2.2. Conjecture de Syracuse :
On note $f : \mathbb{N}^* \to \mathbb{N}^*$ l'application vérifiant, pour tout n pair $f(n) = n/2$ et tout n impair et $f(n) = 3n + 1$.

On conjecture que pour tout entier n, il existe k tel que $f^k(n) = 1$.

Voici un algorithme calculant, pour tout n donné, le plus petit entier k vérifiant $f^k(n) = 1$:

```
def f(n):
  """ fonction de Syracuse
  précondition : n est un entier strictement positif"""
  if n % 2 == 0:
    return n / 2
  else:
    return 3 * n + 1

def syracuse(n):
  """ renvoie le premier entier k tel que
  f^k(n) = 0.
  précondition : n est un entier strictement positif"""
  x = n
  k = 0
  while x != 1:
    x = f(x)
    k += 1
  return k
```

Comme pour les boucles définies, nous sommes confrontés à un problème : comment démontrer un algorithme reposant sur une boucle indéfinie ? Pour cela, nous utilisons

encore les invariants de boucle, afin de prouver qu'après la boucle, le résultat est bien celui voulu.
Mais avant même cela, il y a un point épineux : la boucle **while** va-t-elle vraiment se finir ? Il faut démontrer ce que l'on appelle la **terminaison** de l'algorithme. C'est là que réside le danger des boucles **while** : si elles sont mal écrites, la condition de la boucle ne devient jamais fausse, et la boucle est infinie.
Pour montrer la terminaison d'un algorithme, on utilise cette fois un **variant** de boucle. Il consiste à mettre en avant une variable dont la valeur est modifiée au cours des tours de boucle, de telle sorte que cette modification finisse par rendre fausse la condition de la boucle.
Donnons ces invariant et variant pour l'algorithme de l'exemple **3.2.1** (nous ne pouvons malheureusement pas le faire pour l'exemple **3.2.2**, puisque comme son nom l'indique, la conjecture de Syracuse n'a jamais été demontrée) :

```
n = 1
s = 1              # s = somme des i de 1 à n
while s < 11 :     # invariant : s = somme des i de 1 à n
    n += 1
    s += n         # variant : s, qui est entier et ↙
       strictement croissant
return n
```

Exercice 3.2.3. Démontrer cet algorithme.

3.3 Exercices

Exercice 3.3.1. Calculer 2^9 à l'aide d'une boucle itérative.

Exercice 3.3.2. Écrire un algorithme affichant la table de multiplication de 9.

Exercice 3.3.3. Calculer $16!$ à l'aide d'une boucle itérative.

Exercice 3.3.4. Calculer

$$\sum_{k=0}^{15} \frac{1}{k!}$$

Exercice 3.3.5. Écrire une fonction calculant le nombre de chiffres d'un entier écrit en base 10.

Exercice 3.3.6. On considère la suite u définie par

$$\forall n \in \mathbb{N}^* \quad u_n = \sum_{k=1}^{n} \frac{1}{\sqrt{k}}$$

Quel est la plus petite valeur n pour laquelle $u_n \geq 1000$?

Exercice 3.3.7. Écrire une fonction trouvant le plus petit nombre premier supérieur ou égal à un entier donné.

Exercice 3.3.8. Écrire une fonction calculant le nombre de diviseurs d'un entier n donné.

Exercice 3.3.9. Calculer p_5/q_5 où p et q sont définies par :

$$\begin{aligned} p_0 &= 1 \\ q_0 &= 1 \\ \forall n \in \mathbb{N} \quad p_{n+1} &= p_n^2 + 2q_n^2 \\ \forall n \in \mathbb{N} \quad q_{n+1} &= 2p_n q_n \end{aligned}$$