Editions ENI

Algorithmique

Techniques fondamentales de programmation

Exemples en Python (nombreux exercices corrigés) BTS, DUT informatique

(2e édition)

Collection
Ressources Informatiques

Extrait

Chapitre 3
Tests et logique booléenne

1. Les tests et conditions

1.1 Principe

Dans le précédent chapitre vous avez pu vous familiariser avec les expressions mettant en place des opérateurs, qu'ils soient de calcul, de comparaison (l'égalité par exemple) ou booléens. Ces opérateurs et expressions trouvent tout leur sens une fois utilisés dans des conditions (qu'on appelle aussi des branchements conditionnels). Une expression évaluée est ou vraie (le résultat est différent de zéro) ou fausse. Suivant ce résultat, l'algorithme va effectuer une action, ou une autre. C'est le principe de la condition.

Grâce aux opérateurs booléens, l'expression peut être composée : plusieurs expressions sont liées entre elles à l'aide d'un opérateur booléen, éventuellement regroupées avec des parenthèses pour en modifier la priorité.

```
(a=1 OU (b*3=6)) ET c>10
```

est une expression tout à fait valable. Celle-ci sera vraie si chacun de ses composants respecte les conditions imposées. Cette expression est vraie si a vaut 1 et c est supérieur à 10 ou si b vaut 2 (2*3=6) et c est supérieur à 10.

Reprenez l'algorithme du précédent chapitre qui calcule les deux résultats possibles d'une équation du second degré. L'énoncé simplifié disait que pour des raisons pratiques seul le cas où l'équation a deux solutions fonctionne. Autrement dit, l'algorithme n'est pas faux dans ce cas de figure, mais il est incomplet. Il manque des conditions pour tester la valeur du déterminant : celui-ci est-il positif, négatif ou nul ? Et dans ces cas, que faire et comment le faire ?

Imaginez un second algorithme permettant de se rendre d'un point A à un point B. Vous n'allez pas le faire ici réellement, car c'est quelque chose de très complexe sur un réseau routier important. De nombreux sites Internet vous proposent d'établir un trajet avec des indications. C'est le résultat qui est intéressant. Les indications sont simples : allez tout droit, tournez à droite au prochain carrefour, faites trois kilomètres et au rond-point prenez la troisième sortie direction B. Dans la plupart des cas si vous suivez ce trajet vous arrivez à bon port. Mais quid des impondérables ? Par où allez-vous passer si la route à droite au prochain carrefour est devenue un sens interdit (ça arrive parfois, y compris avec un GPS, prudence) ou que des travaux empêchent de prendre la troisième sortie du rond-point ?

Reprenez le trajet : allez tout droit. Si au prochain carrefour la route à droite est en sens interdit : continuez tout droit puis prenez à droite au carrefour suivant puis à gauche sur deux kilomètres jusqu'au rond-point. Sinon : tournez à droite et faites trois kilomètres jusqu'au rond-point. Au rond-point, si la sortie vers B est libre, prenez cette sortie. Sinon, prenez vers C puis trois cents mètres plus loin tournez à droite vers B.

Ce petit parcours ne met pas uniquement en lumière la complexité d'un trajet en cas de détour, mais aussi les nombreuses conditions qui permettent d'établir un trajet en cas de problème. Si vous en possédez, certains logiciels de navigation par GPS disposent de possibilités d'itinéraire Bis, de trajectoire d'évitement sur telle section, ou encore pour éviter les sections à péage. Pouvez-vous maintenant imaginer le nombre d'expressions à évaluer dans tous ces cas de figure, en plus de la vitesse de chaque route pour optimiser l'heure d'arrivée ?

1.2 Que tester ?

Les opérateurs s'appliquent sur quasiment tous les types de données, y compris les chaînes de caractères, tout au moins en pseudo-code algorithmique (il faudra souvent utiliser des instructions spéciales du langage de programmation pour comparer des chaînes de caractères). Vous pouvez donc quasiment tout tester. Par tester, comprenez ici évaluer une expression qui est une condition. Une condition est donc le fait d'effectuer des tests pour, en fonction du résultat de ceux-ci, effectuer certaines actions ou d'autres.

Une condition est donc une affirmation : l'algorithme et le programme ensuite détermineront si celle-ci est vraie, ou fausse.

Une condition retournant VRAI ou FAUX a comme résultat un **booléen**.

Une condition est souvent une comparaison. Pour rappel, une comparaison est une expression composée de trois éléments :

– une première valeur : variable ou scalaire.
– un opérateur de comparaison.
– une seconde valeur : variable ou scalaire.

Les opérateurs de comparaison sont :

– L'égalité : **=**
– La différence : **!=** ou **<>**
– Inférieur : **<**
– Inférieur ou égal : **<=**
– Supérieur : **>**
– Supérieur ou égal : **>=**

Il est aussi possible que la condition soit une expression unaire : une valeur avec ou non un opérateur. Une variable pouvant être vraie ou fausse, elle peut donc suffire à évaluer la condition donnée.

Le pseudo-code algorithmique n'interdit pas de comparer des chaînes de caractères. Vous prendrez soin de ne comparer que les variables de types compatibles. Dans une condition une expression, quel que soit le résultat de celle-ci, sera toujours évaluée comme étant soit vraie, soit fausse.

Remarque

L'opérateur d'affectation peut aussi être utilisé dans une condition. Dans ce cas si vous affectez 0 à une variable, l'expression sera fausse, et si vous affectez n'importe quelle autre valeur, elle sera vraie.

En langage courant, il vous arrive de dire "choisissez un nombre entre 1 et 10". En mathématique, vous écrivez cela comme ceci :

```
1 ≤ nombre ≤ 10
```

Si vous écrivez ceci dans votre algorithme, attendez-vous à des résultats surprenants le jour où vous allez le convertir en véritable programme. En effet les opérateurs de comparaison ont une priorité, ce que vous savez déjà, mais l'expression qu'ils composent est aussi souvent évaluée de gauche à droite. Si la variable nombre contient la valeur 15, voici ce qui ce passe :

– L'expression 1 ≤ 15 est évaluée : elle est vraie.
– Et ensuite ? Tout va dépendre du langage de programmation utilisé, mais pour la plupart l'expression vrai ≤ 10 est vraie : "vrai" est ici le résultat de l'expression 1 ≤ 15. Vrai vaut généralement 1. Donc 1 ≤ 10 est vraie, ce n'est pas le résultat attendu.
– Le résultat est épouvantable : le programme considère l'expression comme étant vraie et le code correspondant va être exécuté.

Ce n'est probablement pas ce que vous attendiez. Vous devez donc proscrire cette forme d'expression.

Voici celles qui conviennent dans ce cas :

```
nombre>=1 ET nombre<=10
```

Ou encore

```
1<=nombre ET nombre<=10
```

1.3 Tests SI

1.3.1 Forme simple

Il n'y a, en algorithmique, qu'une seule instruction de test : "**Si**", qui prend cependant deux formes : une simple et une complexe. Le test SI permet d'exécuter un bloc d'instructions si la condition (la ou les expressions qui la composent) est vraie. La forme simple est la suivante :

```
Si booléen Alors
  Bloc d'instructions
FinSi
```

Notez ici que le booléen est la condition (ou expression). Comme indiqué précédemment, la condition peut aussi être représentée par une seule variable. Si elle contient 0, elle représente le booléen FAUX, sinon le booléen VRAI.

Que se passe-t-il si la condition est vraie ? Le bloc d'instructions situé après le "**Alors**" est exécuté. Sa taille (le nombre d'instructions) n'a aucune importance : de une ligne à n lignes, sans limite. Dans le cas contraire, le programme continue à l'instruction suivant le "**FinSi**". L'exemple suivant montre comment obtenir la valeur absolue d'un nombre avec cette méthode.

```
PROGRAMME ABS
VAR
  Nombre :entier
DEBUT
  nombre←-15
  Si nombre<0 Alors
    nombre←-nombre
  FinSi
  Afficher nombre
FIN
```

En Python, c'est le "if" qui doit être utilisé avec l'expression booléenne entre parenthèses. La syntaxe est celle-ci :

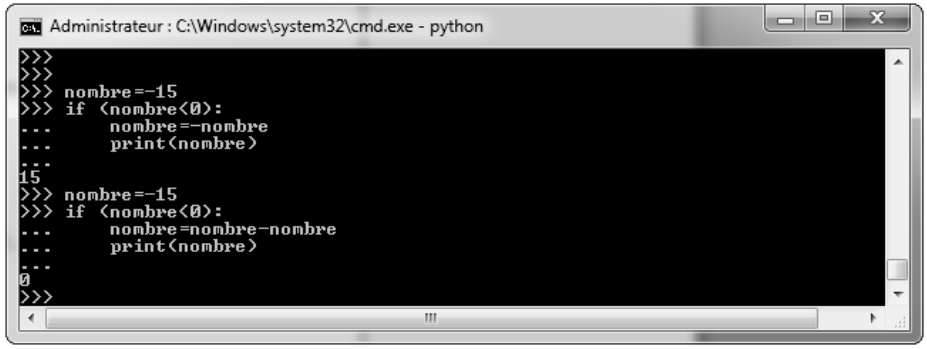
```
if(boolean):
/*code */
```

Une notion importante ici est que Python, contrairement a beaucoup de langages tels que le C ou le Java, n'utilise pas les accolades pour déterminer un bloc d'instruction mais l'indentation.

Si vous regardez le code ci-dessus, nous avons le test if avec la condition qui se termine par ":".

La ligne du dessous est indentée, c'est-à-dire que nous avons appuyé sur la touche [Tabulation] avant de commencer la ligne. Tout le code qui sera tabulé en dessous appartiendra au if.

```
nombre=-15
if(nombre<0):
    nombre=-nombre
    print(nombre)
```



Une condition avec une valeur booléenne se fait de la même manière. Le code suivant n'a qu'un rôle pédagogique : il met en place un drapeau ou flag permettant d'indiquer si la condition est vérifiée. Si le drapeau est vrai, alors le nombre est négatif. Le drapeau est ensuite testé pour afficher le résultat opposé.

```
PROGRAMME ABS
VAR
  nombre :entier

  drapeau :booleen
DEBUT

  drapeau←FAUX
  nombre←-15
  Si nombre<0 Alors
    drapeau←VRAI
  FinSi
```

Editions ENI

Python 3
Les fondamentaux du langage

(2e édition)

Collection
Ressources Informatiques

Extrait

Chapitre 4.4
Programmation système et réseau

1. Présentation

1.1 Définition

La programmation système se définit par opposition à la programmation d'applications. Il ne s'agit pas de concevoir un logiciel qui va utiliser le système et ses ressources pour effectuer une action, mais de concevoir une des briques qui va s'intégrer au système lui-même. Cela peut donc être le développement d'un pilote pour un matériel, d'une interface réseau ou encore la gestion des ressources.

Par extension, la création d'un programme qui utilise d'autres programmes systèmes est de la programmation système. Le terme programmation système s'étend alors à tout ce qui peut permettre à un administrateur système de résoudre les problématiques usuelles concernant son domaine de compétence, à savoir la gestion des utilisateurs, des périphériques, des processus, de la sauvegarde…

Ainsi, par extension, l'utilisation des commandes **`bash`** est de la programmation système. L'utilisation des commandes **`mysql`** et **`mysqldump`** pour opérer des actions de sauvegarde quotidiennes l'est également.

Un système d'exploitation moderne est écrit majoritairement en C, le reste étant de l'assembleur spécifique à la machine. Ce même système d'exploitation – moderne – a des fonctionnalités de haut niveau, telles que le gestionnaire de paquets qui a besoin par exemple d'effectuer des opérations diverses telles que des échanges réseau pour télécharger des paquets, vérifier leur intégrité, les ouvrir, les installer.

Celles-ci sont le plus souvent écrites en Python, parce qu'il est un langage facile à manipuler, efficace, complet et surtout fiable, disposant de l'outillage nécessaire.

1.2 Objectifs du chapitre

Dans ce chapitre sont présentés les moyens mis à disposition par Python pour permettre d'exécuter des commandes système de manière à effectuer des opérations de maintenance, l'utilisation du système de fichiers, les moyens permettant à Python d'être une alternative crédible à Bash, en particulier par le traitement du passage d'arguments.

Les problématiques liées à la gestion des protocoles réseau, souvent associées à de la programmation système, sont également présentées, et orientées vers la communication entre différents types de clients et de serveurs et différentes technologies. Cela couvre également les services web.

La gestion des tâches et processus à haut niveau est également une problématique faisant partie de la programmation système. Mais étant de haut niveau, elle est utilisée beaucoup plus pour des applications qu'en tant qu'élément de programmation système pure. C'est la raison pour laquelle cette partie est traitée dans un chapitre dédié, le chapitre Programmation parallèle.

2. Écrire des scripts système

2.1 Appréhender son système d'exploitation

2.1.1 Avertissement

L'exécution de commandes externes est intimement liée au système sur lequel se trouve installé Python. D'une part, chaque système d'exploitation possède ses propres commandes. Par exemple, pour lister un répertoire, on utilisera **`ls`** ou **`dir`**.

Cette section traite principalement les commandes Unix.

Au-delà des commandes système classiques, certaines commandes comme **`mysql`** ou **`mysqldump`** ne peuvent être utilisées que si les programmes adéquats ont été installés, quel que soit le système.

2.1.2 Système d'exploitation

Python propose un module de bas niveau permettant de gérer des informations sur le système d'exploitation :

```
>>> import os
```

Voici les deux principaux moyens de vérifier la nature du système :

```
>>> os.name
'posix'
>>> os.uname()
('Linux', 'nom_donne_au_host', '2.6.38-11-generic', '#50-Ubuntu
SMP Mon Sep 12 21:17:25 UTC 2011', 'x86_64')
```

La première commande donne une standardisation de l'environnement et la seconde des détails sur le nom du système d'exploitation, de la machine, le nom du noyau et sa version ainsi que l'architecture de la machine. Tester ces valeurs permet d'effectuer des choix et d'adapter une application à un environnement précis pour certaines opérations qui le nécessitent.

Voici comment trouver la liste des variables d'environnement :

```
>>> list(os.environ.keys())
```

Et voici comment aller chercher la valeur d'une de ces variables :

```
>>> os.getenv('LANGUAGE')
'fr_FR:fr'
```

L'exploitation de ces variables d'environnement permet également de diriger des choix permettant l'adaptation de l'application. Dans le cas qui vient d'être vu, le choix de la locale peut servir à produire une interface adaptée au langage de l'utilisateur. Il est possible de lire les variables d'environnement sous forme d'octets avec **`os.environb`** (utile lorsque Python est unicode, mais pas le système).
Python permet également de modifier ces variables d'environnement à l'aide de la méthode **`putenv`**. L'environnement est alors affecté dans tous les sous-processus.

2.1.3 Processus courant

Python permet d'obtenir des informations sur le processus courant. Ces fonctionnalités ne sont disponibles que pour Linux.
La principale d'entre elles est l'identifiant du processus courant et celui du parent :

```
>>> os.getpid()
5256
>>> os.getppid()
3293
```

Le parent peut correspondre à l'identifiant d'un processus Python père ou à celui de la console lorsque Python est lancé en mode console depuis la console.
Il peut récupérer l'utilisateur affecté au processus et l'utilisateur effectif :

```
>>> os.getuid()
1000
>>> os.geteuid()
1000
```

On peut également avoir des informations textuelles :

```
>>> os.getlogin()
'sch'
```

Et des informations sur les groupes rattachés au processus :

```
>>> os.getgroups()
[4, 20, 24, 46, 112, 120, 122, 1000]
```

Ainsi que sur le 3-uplet (utilisateur courant, effectif, sauvegardé) :

```
>>> os.getresuid()
(1000, 1000, 1000)
```

Voici le 3-uplet des groupes associés (courant, effectif, sauvegardé) :

```
>>> os.getresgid()
(1000, 1000, 1000)
```

L'identifiant 0 est celui de root, 1000 celui du premier utilisateur créé (lors de l'installation du système) pour Unix.
Si l'on ne veut pas que l'application puisse être lancée par root, on peut faire :

```
>>> if os.getuid() == 0:
...     print('Ne doit pas être lancé avec root...')
```

Enfin, il est possible de retrouver le terminal contrôlant le processus :

```
>>> os.ctermid()
'/dev/tty'
```

Pour plus d'informations :

```
$ man tty
```

Pour tester les différences, la console peut être lancée en root :

```
$ sudo python3
```

2.1.4 Utilisateurs et groupes

Un système d'exploitation moderne est multi-utilisateur et permet de retrouver des informations sur les utilisateurs déclarés.
Python permet de rechercher des renseignements sur les utilisateurs en se servant des fichiers qui stockent ces informations :

```
>>> with open("/etc/passwd") as f:
...     users = [l.split(':', 6) for l in f]
...
```

On peut ainsi afficher les données de l'utilisateur simplement :

```
>>> users[0]
['root', 'x', '0', '0', 'root', '/root', '/bin/bash\n']
```

Elles correspondent respectivement aux :
– nom d'utilisateur ;
– mot de passe encrypté (ou stocké chiffré dans un fichier séparé) ;
– identifiant de l'utilisateur ;
– identifiant de son groupe ;
– nom complet ;
– répertoire d'accueil ;
– shell au démarrage de sa session.
Avec ces données, celui qui connaît le fonctionnement de son système sait à quels niveaux il peut intervenir pour effectuer des modifications.
Voici comment obtenir l'ensemble des valeurs utilisées pour les différents utilisateurs :

```
>>> {u[6] for u in users}
{'/bin/sh\n', '/bin/false\n', '/bin/sync\n',
'/bin/bash\n', '/usr/sbin/nologin\n'}
>>> {u[5] for u in users}
{'/home/sch', '/var/www', '/root', '/var/lib/bacula', [...]}
```

On peut utiliser les mêmes procédés pour les groupes :

```
>>> with open("/etc/group") as f:
...     groups = [l.split(':', 3) for l in f]
...
>>> groups[0]
['root', 'x', '0', '\n']
```

Le troisième élément est le numéro du groupe servant de lien avec l'utilisateur.
Voici comment mettre cette relation en évidence :

```
>>> guser = {u[3]: u[0] for u in users}
>>> user_group = [(guser.get(g[2]), g[0]) for g in groups]
```

On voit donc en peu d'exemples comment utiliser la puissance des types de Python. Le bon type pour la bonne utilisation.
Voici un script testant l'existence des éléments caractéristiques d'un utilisateur, nom d'utilisateur, le nom de groupe associé et son dossier personnel à l'emplacement par défaut :

```
>>> for username in ['sch', 'existepas']:
...     if username in (u[0] for u in users):
...         print("L'utilisateur %s existe déjà" % username)
...     if username in (g[0] for g in groups):
...         print("Le groupe %s existe déjà" % username)
...     home = '/home/%s' % username
...     if os.path.exists(home):
...         print('Le dossier %s existe déjà' % username)
...
L'utilisateur sch existe déjà
Le groupe sch existe déjà
Le dossier sch existe déjà
```

Enfin, pour terminer, les utilisateurs courants du système (pas les utilisateurs Apache, Bacula ou autre) ont un identifiant compris entre certaines bornes :

```
>>> max_user_id = max([id for id in (int(u[2]) for u in users) if
1000 < id < 19999])
>>> max_group_id = max([id for id in (int(g[2]) for g in groups)
if 1000 < id < 19999])
```

Voici les résultats pour ma machine qui ne contient que deux comptes d'utilisateurs courants :

```
>>> max_user_id, max_group_id
(1001, 1001)
```

2.1.5 Constantes pour le système de fichiers

Le système d'exploitation définit certaines caractéristiques par rapport au système de fichiers. Il s'agit de la notation du répertoire courant, du répertoire parent, du séparateur de répertoire (il peut y en avoir un second), du séparateur d'extensions, de la séparation entre chemins lorsqu'ils sont écrits l'un après l'autre, et le séparateur qui définit le changement de ligne :

```
>>> os.curdir, os.pardir
('.', '..')
>>> os.sep, os.altsep, os.extsep, os.pathsep, os.linesep
('/', None, '.', ':', '\n')
```

Chaque système définit également un chemin par défaut (liste de répertoires séparés par le séparateur **`os.pathsep`**) et un chemin vers une interface nulle :

```
>>> os.defpath
':/bin:/usr/bin'
>>> os.devnull
'/dev/null'
```

Il existe encore un certain nombre d'opérations plus spécialisées, mais celles présentées permettent déjà de réaliser un code indépendant du système :

```
>>> os.sep.join([os.curdir, 'rep', 'fname' + os.extsep + 'ext'])
'./rep/fname.ext'
```

2.1.6 Gérer les chemins

Python 3.4 introduit une sémantique objet permettant de gérer les chemins beaucoup plus simplement que des chaînes de caractères :

```
>>> from pathlib import Path
>>> mypath = Path('/var/www')
>>> mypath.exists()
True
>>> mypath.is_dir()
True
```

Pour les versions antérieures à Python 3.4, il est possible d'installer le module **`pathlib2`** :

```
$ sudo pip install pathlib2
```

Ce module est parfaitement intégré avec les modules Python déjà existants, parmi lesquels le fameux module **`glob`** qui permet de faire des recherches de fichier très efficaces :

```
>>> www = Path('/var/www')
>>> list(www.glob('**/*.html'))
[PosixPath('index.html'), PosixPath('doc/index.html')]
```

Là où l'outil devient vraiment très efficace sémantiquement parlant, c'est qu'il profite à fond de la capacité de Python à surcharger les opérateurs et qu'il l'utilise intelligemment de manière à rendre son utilisation très basique et surtout particulièrement lisible :

```
>>> doc, index = PurePath('doc'), PurePath('index.html')
>>> myfile = mypath / doc / index
```