18 Gestion des erreurs

La gestion des erreurs permet d'éviter que votre programme plante en prévoyant vous même les sources d'erreurs éventuelles.

Voici un exemple dans lequel on demande à l'utilisateur d'entrer un nombre, puis on affiche ce nombre.

```
>>> nb = int(raw_input("Entrez un nombre: "))
Entrez un nombre: 23
>>> print nb
23
```

La fonction `raw_input()` permet à l'utilisateur de saisir une chaîne de caractères. Cette chaîne de caractères est ensuite transformée en nombre entier avec la fonction `int()`.

Si l'utilisateur ne rentre pas un nombre, voici ce qui se passe :

```
>>> nb = int(raw_input("Entrez un nombre: "))
Entrez un nombre: ATCG
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'ATCG'
```

L'erreur provient de la fonction `int()` qui n'a pas pu convertir la chaîne de caractères `"ATCG"` en nombre, ce qui est normal.

Le jeu d'instruction `try / except` permet de tester l'exécution d'une commande et d'intervenir en cas d'erreur.

```
>>> try:
...     nb = int(raw_input("Entrez un nombre: "))
... except:
...     print "Vous n'avez pas entré un nombre !"
...
Entrez un nombre: ATCG
Vous n'avez pas entré un nombre !
```

Dans cette exemple, l'erreur renvoyée par la fonction `int()` (qui ne peut pas convertir `"ATCG"` en nombre) est interceptée et déclenche l'affichage du message d'avertissement.

On peut ainsi redemander sans cesse un nombre à l'utilisateur, jusqu'à ce que celui-ci en rentre bien un.

```
>>> while 1:
...     try:
...             nb = int(raw_input("Entrez un nombre: "))
...             print "Le nombre est", nb
...             break
...     except:
...             print "Vous n'avez pas entré un nombre !"
...             print "Essayez encore"
...
Entrez un nombre: ATCG
Vous n'avez pas entré un nombre !
Essayez encore
Entrez un nombre: toto
```

```
Vous n'avez pas entré un nombre !
Essayez encore
Entrez un nombre: 55
Le nombre est 55
```

Notez que dans cet exemple, l'instruction `while 1` est une boucle infinie (car la condition 1 est toujours vérifiée) dont l'arrêt est forcé par la commande `break` lorsque l'utilisateur a effectivement bien rentré un nombre.

La gestion des erreurs est très utile dès lors que des données extérieures entrent dans le programme, que ce soit directement par l'utilisateur (avec la fonction `raw_input()`) ou par des fichiers.

Vous pouvez par exemple vérifier qu'un fichier a bien été ouvert.

```
>>> nom = "toto.pdb"
>>> try:
...     f = open(nom, "r")
... except:
...     print "Impossible d'ouvrir le fichier", nom
```

Si une erreur est déclenchée, c'est sans doute que le fichier n'existe pas à l'emplacement indiqué sur le disque ou que (sous Unix), vous n'ayez pas les droits d'accès pour le lire.

Il est également possible de spécifier le type d'erreur à gérer. Le premier exemple que nous avons étudié peut s'écrire :

```
>>> try:
...     nb = int(raw_input("Entrez un nombre: "))
... except ValueError:
...     print "Vous n'avez pas entré un nombre !"
...
Entrez un nombre: ATCG
Vous n'avez pas entré un nombre !
```

Ici, on intercepte une erreur de type `ValueError`, ce qui correspond bien à un problème de conversion avec `int()`. Il existe d'autres types d'erreurs comme `RuntimeError`, `TypeError`, `NameError`, `IOError`, etc.

Enfin, on peut aussi être très précis dans le message d'erreur. Observez la fonction `downloadPage()` qui, avec le module `urllib2`, télécharge un fichier sur internet.

```
import urllib2

def downloadPage(address):
    error = ""
    page = ""
    try:
        data = urllib2.urlopen(address)
        page = data.read()
    except IOError, e:
        if hasattr(e, 'reason'):
            error =  "Cannot reach web server: " + str(e.reason)
        if hasattr(e, 'code'):
            error = "Server failed %d" %(e.code)
    return page, error
```

La variable `e` est une instance (un représentant) de l'erreur de type `IOError`. Certains de ces attributs sont testés avec la fonction `hasattr()` pour ainsi affiner le message renvoyé (ici contenu dans la variable `error`).

19 Trucs et astuces

19.1 Shebang et /usr/bin/env python

Lorsque vous programmez sur un sytème Unix, le shebang correspond aux caractères `#!` qui se trouve au début de la première ligne d'un script. Le shebang est suivi du chemin complet du programme qui interprète le script.

En Python, on trouve souvent la notation

```
#! /usr/bin/python
```

Cependant, l'exécutable `python` ne se trouve pas toujours dans le répertoire `/usr/bin`. Pour maximiser la portabilité de votre script Python sur plusieurs systèmes Unix, utilisez plutôt cette notation :

```
#! /usr/bin/env python
```

Dans le cas présent, on appelle le programme d'environnement `env` (qui se situe toujours dans le répertoire `/usr/bin`) pour lui demander où se trouve l'exécutable `python`.

19.2 Python et utf-8

Si vous utilisez des caractères accentués dans des chaînes de caractères ou bien même dans des commentaires, cela occasionnera une erreur lors de l'exécution de votre script.

Pour éviter ce genre de désagrément, ajoutez la ligne suivante à la deuxième ligne (la position est importante) de votre script :

```
# -*- coding: utf-8 -*-
```

En résumé, tous vos scripts Python devraient ainsi débuter par les lignes :

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
```

19.3 Vitesse d'itération dans les boucles

La vitesse d'itération (de parcours) des élements d'une liste peut être très différente selon la structure de boucle utilisée. Pour vous en convaincre, copiez les lignes suivantes dans un script Python (par exemple `boucles.py`) puis exécutez-le.

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import time

# création d'une liste de 5 000 000 d'éléments
# (à adapter suivant la vitess de vos machines)
taille = 5000000
print "Création d'une liste avec %d élements" %( taille )
toto = range( taille )

# la variable 'a' accède à un élément de la liste

# méthode 1
start = time.time()
```

```
for i in range( len(toto) ):
    a = toto[i]
print "méthode 1 (for in range) : %.1f secondes" %( time.time() - start )

# méthode 2
start = time.time()
for ele in toto:
    a = ele
print "méthode 2 (for in) : %.1f secondes" %( time.time() - start )

# méthode 3
start = time.time()
for i in xrange( len(toto) ):
    a = toto[i]
print "méthode 3 (for in xrange) : %.1f secondes" %( time.time() - start )

# méthode 4
start = time.time()
for idx, ele in enumerate( toto ):
    a = ele
print "méthode 4 (for in enumerate): %.1f secondes" %( time.time() - start )
```

Vous devriez obtenir une sortie similaire à celle-ci :

```
poulain@cumin> ./boucles.py
Création d'une liste avec 5000000 élements
méthode 1 (for in range) : 1.8 secondes
méthode 2 (for in) : 1.0 secondes
méthode 3 (for in xrange) : 1.2 secondes
méthode 4 (for in enumerate): 1.4 secondes
```

La méthode la plus rapide pour parcourir une liste est donc d'itérer directement sur les éléments (`for element in liste`). Cette instruction est à privilégier le plus possible.

La méthode `for i in range(len(liste))` est particulièrement lente car la commande `range(len(liste))` génère une énorme liste avec tous les indices des éléments (la création de liste est assez lente en Python). Si vous voulez absolument parcourir une liste avec les indices des élements, utilisez plutôt la commande `for i in xrange(len(liste))` car l'instruction `xrange()` ne va pas créer une liste mais incrémenter un compteur qui correspond à l'indice des éléments successifs de la liste.

Enfin, la commande `for indice, element in enumerate(liste)` est particulièrement efficace pour récupérer en même temps l'élément et son indice.

19.4 Liste de compréhension

Une manière originale et très puissante de générer des listes est la compréhension de liste. Pour plus de détails, consultez à ce sujet le site de Python et celui de wikipédia.

Voici quelques exemples :

- Nombres pairs compris entre 0 et 99
  `print [i for i in range(99) if i%2 == 0]`
  Le même résultat est obtenu avec
  `print range(0,99,2)`
  ou

```
print range(0,99)[::2]
```

– Jeu sur la casse des mots d'une phrase.

```
message = "C'est sympa la BioInfo"
msg_lst = message.split()
print [[m.upper(), m.lower(), len(m)] for m in msg_lst]
```

– Formatage d'une séquence avec 60 caractères par ligne

```
# exemple d'une séquence de 150 alanines
seq = "A"*150
width= 60
print "\n".join( [seq[i:i+width] for i in range(0,len(seq),width)] )
```

Formatage fasta d'une séquence (avec la ligne de commentaire)

```
commentaire = "mon commentaire"
# exemple d'une séquence de 150 alanines.
seq = "A"*150
width= 60
print "> "+commentaire+"\n"+"\n".join( [seq[i:i+width] for i in range(0,len(seq),width)] )
```

– Sélection des lignes correspondantes aux carbones alpha dans un fichier pdb

```
# ouverture du fichier pdb (par exemple 1BTA.pdb)
pdb = open("1BTA.pdb", "r")
# sélection des lignes correspondantes aux C alpha
CA_line = [line for line in pdb if line.split()[0] == "ATOM" and line.split()[2] == "CA"]
# fermeture du fichier
pdb.close()
# éventuellement affichage des lignes
print CA_line
```

19.5 Sauvegardez votre historique de commandes

Vous pouvez sauvegarder l'historique des commandes utilisées dans l'interpréteur Python avec le module `readline`.

```
>>> print "hello"
hello
>>> a = 22
>>> a = a + 11
>>> print a
33
>>> import readline
>>> readline.write_history_file()
```

Quittez Python. L'historique de toutes vos commandes est dans votre répertoire personnel, dans le fichier `.history`. Relancez l'interpréteur Python.

```
>>> import readline
>>> readline.read_history_file()
```

Vous pouvez accéder aux commandes de la session précédente avec la flèche du haut de votre clavier.

```
>>> print "hello"
hello
>>> a = 22
>>> a = a + 11
>>> print a
33
```