XXXI. Algorithmique

Dans ce dernier TP on recense quelques bonnes pratiques de rédaction des algorithmes.

1 - Spécification d'un algorithme

- Ecrire la **signature** d'une fonction (ou d'un algorithme) consiste à :
  - la nommer ;
  - préciser ses arguments et leurs types ;
  - préciser le type des valeurs renvoyées.

Exemple 1 :

La signature de la fonction `pgcd` qui à deux entiers a et b associe leur pgcd est :

`pgcd : (a : int, b: int)` $\mapsto$ `int`

- **Spécifier** une fonction c'est préciser sa signature **et** ajouter :
  - les **pré-conditions** que les arguments doivent vérifier ;
  - les **post-conditions** portant sur les valeurs renvoyées.

Exemple 2 :

Les pré-conditions de la fonction `pgcd` sont : $a \geq 0$ et $b \geq 0$ et $(a,b) \neq (0,0)$.
La post-condition est que la valeur renvoyée est bien le pgcd de a et b.

On a rarement spécifié nos fonctions cette année tout simplement parce que dans les exercices ces éléments figurent déjà dans l'énoncé. Dans la pratique d'un informaticien, lorsqu'on écrit une fonction il est nécessaire de toujours décrire sa spécification. En règle générale on le fait en début de fonctions :

```
def pgcd(a,b):
    """
    a,b : int avec la pre-condition
       a>=0, b>=0 et (a,b) != (0,0)
    renvoie le pgcd de a et b de type int
    """
```

Ex. 1 :

Ajouter une spécification à la fonction suivante :

```
def mystere(L):
    n = len(L)
    L1 = []
    while n > 0:
        L1.append(L.pop())
    return L
```

2 - Deboggage

Dans un algorithme une erreur peut être due à :

- une erreur de syntaxe (par exemple `=` au lieu de `==` dans une condition) ;
- une **exception**

Une exception survient à cause d'une exécution invalide. Cela peut venir de l'utilisateur (la saisie d'entrée ne correspond pas à la pré-condition). Cela peut aussi venir d'une erreur de programmation. Une exception est levée lorsqu'elle apparaît : on ne peut donc pas la détecter au moment de l'élaboration du programme.

Exemple 3 :

On considère la fonction :

```
def divise(a,b):
    return a / b
```

L'exécution de `divise(13,0)` lève l'exception "division par zéro".

Les exceptions les plus fréquentes sont :

- `TypError` si le type de l'argument donné ne correspond pas au type attendu ;
- `IndexError` si dans le parcours d'une liste on appelle un indice absent ;
- `KeyError` si on appelle une clé absente dans un dictionnaire ;
- `ZeroDivisionError`

a) L'instruction `assert`

Pour vérifier les pré-conditions d'une fonction, on peut utiliser l'instruction `assert` dont la syntaxe est `assert condition`. Si la condition est évaluée à `True` alors le programme continue, sinon l'exception `AssertionError` est levée et le programme est interrompu.
La fonction `isinstance()` permet de vérifier les types des pré-conditions :

```
def divise(a,b):
    assert isinstance(a,float) and isintance(b,float) and b != 0
    return a / b
```

L'instruction `assert` ne doit figurer que dans un programme en construction ou pendant sa phase de test.

b) L'instruction `try ... except`

Le bloc `try` permet de tester un bloc de code et de ne l'exécuter que s'il ne contient aucune erreur. Dans le cas contraire, le bloc est ignoré. Le bloc `except` permet de gérer l'exception levée dans le bloc `try`.

```
def divise():
    num = int(input("numerateur ?"))
    den = int(input("denominateur ?"))
    try:
        resultat = num/den
    except ZeroDivisionError:
        print("il y a division par zero")
    else:
        return resultat
```

Ex. 2 :

On considère la fonction suivante qui associe à un dictionnaire `dico` et une clé `k` la valeur de `dico[k]`.

```
def eval_dico(dico,k):
    return dico[k]
```

1. Ajouter en préambule de la fonction sa spécification.
2. Proposer une instruction `assert` qui vérifie les pré-conditions : `dico` est de type `dic` et k est une clé de dico.
3. Proposer un bloc `try ... except` qui permet de signaler l'exception `KeyError` levée si k n'est pas une clé de `dico`

3 - Construction d'un QCM

On présente un QCM sous la forme d'une liste de questions. Chaque question est un dictionnaire dont les clés sont :

- "libel" : l'énoncé de la question ;
- "choix" : une liste de réponses possibles ;
- "reponse" : l'indice de la bonne réponse dans la liste précédente.

```
q1 = {"libel" : "De quelle couleur est le cheval blanc d'Henry IV ?",
"choix" : ["blanc","noir","marron"],"reponse" : 0}
q2 = {"libel" : "Qui est ami avec Cauchy ?",
"choix":["Rolle","Schwarz","Tchebychev"],"reponse" : 1}
QCM = [q1,q2]
```

1. Ecrire une fonction `pose_question(QCM,i)` qui affiche la i-ème question du QCM et la liste des réponses possibles puis qui demande à l'utilisateur sa réponse et renvoie `True` si la réponse est juste et `False` sinon.
2. Ajouter un bloc `try ... except` qui renvoie "la réponse n'est pas possible" si l'utilisateur fait un choix impossible.
3. Ecrire une fonction `qcm_entier(QCM)` qui pose toutes les questions du QCM et qui affiche le score final du joueur.