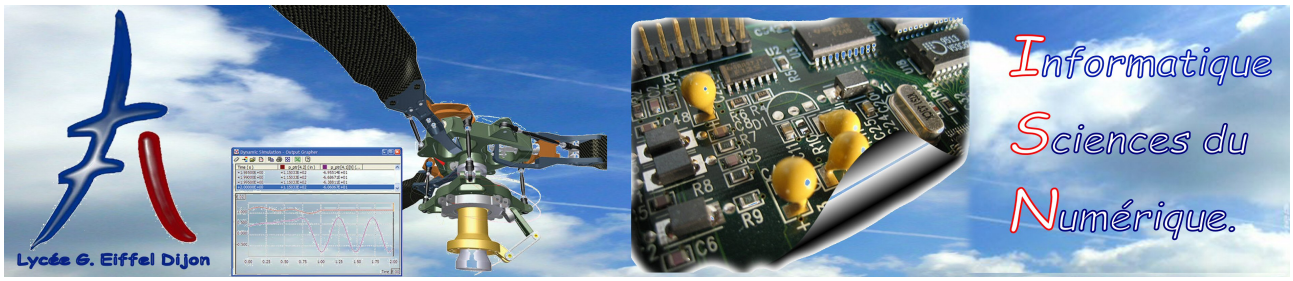

Les fonctions en Python

Objectifs :

- ➢ **Savoir définir et utiliser une fonction**
- ➢ **Savoir utiliser les exceptions**

1 Fonctions : définition, intérêt

1.1 Exemple

On a un programme qui présente des répétitions :

```
print("Vous êtes monsieur Martin.")
print("Vous habitez à  Dijon.")
print("Votre compte en banque s'élève à  1 552 000 euros.")
print("\----------------------------------------------")

print("Vous êtes monsieur Durand.")
print("Vous habitez à  Marsannay.")
print("Votre compte en banque s'élève à  3204.25 euros.")
print("\----------------------------------------------")

print("Vous êtes monsieur Brunet.")
print("Vous habitez à  Chenôve.")
print("Votre compte en banque s'élève à  24788,32 euros.")
print("\----------------------------------------------")
```

Inconvénients de ces répétitions :

- C'est long à écrire.
- C'est difficile à lire.
- Si on veut modifier une phrase, il faut la modifier à 3 endroits dans le code source.

On utilise donc une fonction à la place.

1.2 Définition, passage d'arguments

Une fonction est une instruction isolée du reste du programme, qui possède un nom, et qui peut être appelée par ce nom à n'importe quel endroit du programme et autant de fois que l'on veut.
Les trois blocs de texte précédents ne sont pas totalement identiques : le nom, le lieu et le montant du compte diffèrent : il faut paramétrer l'instruction qu'on isole pour pouvoir lui passer le nom, le lieu et le montant : on passe donc des arguments à la fonction.
Exemple : proposer une fonction et un programme principal qui permettent d'afficher les données de Martin, Durand et Brunet.
Avantages des fonctions :

- Pour utiliser une fonction dans un programme principal, il est inutile de savoir comment la fonction est programmée, connaî tre son utilisation est suffisant.
- Les fonctions permettent d'organiser le travail de développement : un programmeur peut mettre au point une fonction, pendant qu'un autre crée le programme principal.

— En cas de changement, s'il y a x appels de la fonction dans le programme principal, il suffit de modifier une fois la fonction au lieu de modifier x fois le programme principal.

1.3 Retour d'une valeur

Les arguments permettent de passer des informations du programme principal vers une fonction. Inversement, une fonction peut passer une valeur qu'elle a calculée au programme principal. On utilise alors l'instruction *return*.
Par exemple, la fonction suivante prend en argument un entier n et renvoie au programme principal la somme des cubes des n premiers entiers non nuls :

```
def somme_cubes(n) :
    """ Renvoie la somme des cubes des entiers de 1  n
    o n est un entier """
    s = 0
    for k in range(1,n+1) :
        s = s + pow(k,3)
    return s

assert somme_cubes(3)==36
assert somme_cubes(5)==225
```

Remarques :
— l'instruction *return* interrompt l'exécution de la fonction.
— La première ligne de la fonction ci-dessus s'appelle l'en-tête de la fonction.
— L'instruction écrite après l'en-tête s'appelle le corps de la fonction. Le corps de la fonction est indenté.

1.4 Conclusion

Pour écrire une fonction :

1. On détermine l'en-tête :
   — Mot clé *def*.
   — Nom (explicite) de la fonction : par convention, il s'écrit en minuscules et chaque mot est séparé par _ (underscore).
   — Liste des paramètres entre parenthèses (avec un nom explicite) séparés par des virgules, sans préciser le type.
   — : à la fin
2. On écrit la docstring de la fonction en l'indentant
3. On écrit le corps de la fonction en l'indentant, avec l'instruction *return* si nécessaire.
4. C'est très conseillé : on s'assure par des tests que la fonction fonctionne **quelles que soient les valeurs passées en arguments**. Pour cela, on utilise le mot clé *assert* avec un test qui suit : si le test est faux, une erreur est signalée lors de l'exécution.

1.5 Remarque sur les paramètres

On considère l'exemple suivant :

```
    def affiche(a, b, c, d) :
        print "a = {}, b = {}, c = {}, d ={}".format(a,b,c,d)
```

— L'ordre des arguments est important lors de l'appel de la fonction : *affiche(1,2,3,4)* donnera un résultat différent de *affiche(4,3,2,1)*
— On peut donner des valeurs par défaut aux paramètres :

```
def affiche(a, b, c=6, d=8) :
    print "a = {}, b = {}, c = {}, d ={}".format(a,b,c,d)
```

L'appel *affiche(1,5)* affiche $a = 1$, $b = 5$, $c = 6$, $d = 8$. L'appel *affiche(1,5,3)* affiche $a = 1$, $b = 5$, $c = 3$, $d = 8$.
Les paramètres admettant une valeur par défaut sont forcément les derniers :

```
def affiche(a, b=5, c=6, d) :
    print "a = {}, b = {}, c = {}, d ={}".format(a,b,c,d)
```

ne fonctionne pas.

— On peut nommer les arguments lors de l'appel de la fonction : L'appel *affiche(a=4,b=6,c=9,d=11)* affiche $a = 4$, $b = 6$, $c = 9$, $d = 11$.

2 Les exceptions

2.1 Comment intercepter une exception

Lors de l'exécution d'une fonction, il peut y avoir une situation exceptionnelle (division par 0, saisie par l'utilisateur d'un nombre négatif alors qu'un nombre positif est requis, fichier non trouvé...). La fonction peut alors lever une exception qui peut être interceptée par l'utilisateur qui appelle la fonction. Un message d'erreur est en même temps affiché dans la console.
Le programmeur peut choisir ou pas d'intercepter une exception (par exemple une division par 0 : si elle n'est pas interceptée, le programme s'arrête).
La syntaxe la plus simple est :

```
try :
    Instructions qui peuvent provoquer une erreur
except :
    Instructions excuter si une erreur survient
```

Une autre syntaxe possible est :

```
try :
    Instructions qui peuvent provoquer une erreur
except :
    Instructions  excuter si une erreur survient
else :
    Instructions si tout se passe bien
```

De nombreux programmeurs n'utilisent pas le *else* et préfèrent tout écrire dans le *try* mais le *else* peut être utile dans certains cas.
Exemple : saisie d'un nombre dans la console :

```
saisie_valide = False
while not saisie_valide :
    #On boucle tant qu'on n'a pas saisi un nombre entier.
    try :
        #Une exception est levée par la fonction input si la chaîne
            saisie ne représente pas un nombre entier
        saisie = int(input("Entrer un nombre : "))
    except :
        #En cas d'erreur, on affiche un message
        print("Il faut entrer un nombre entier.")
    else :
        #S'il n'y a pas d'erreur, la saisie est validée
        saisie_valide = True
print("Le nombre entier saisi est {}.".format(saisie))
```

2.2 Créer une fonction qui lève une exception

On utilise le mot clé *raise* et le type de l'exception à lever :
raise TypeDeLException("message à afficher")
Pour le type de l'exception, on pourra utiliser *TypeError* ou un type plus précis, par exemple *ValueError* (Python possède de nombreux types d'exceptions).
Exemple :

```
def calcul(n) :
    """ On veut que le nombre saisi soit strictement positif """
    if n<=0:
        raise ValueError("Vous devez saisir un nombre strictement
            positif :")
    else :
        return 2+sqrt(n)
```

Le programmeur qui utilisera cette fonction pourra intercepter l'exception levée, même s'il n'y est pas obligé.
S'il écrit *a = calcul(0)* (sans intercepter), le programme s'arrête avec un message d'erreur.
S'il décide d'intercepter, il pourra choisir ce que fait le programme en cas d'erreur :

```
try :
    a = calcul(0)
except :
    a = 1
```

En cas d'erreur, la variable *a* prend la valeur 1 et le programme continue.