Fiche 4 - Fonctions

Une fonction est une variable de type `function`. Pour qu'une fonction soit reconnue et donc utilisable, il faut qu'elle soit *chargée* :

- il y a les fonctions connues (ou contenues dans un module déjà chargé) ;
- il y a les fonctions que l'on définit dans la console : elle sont chargées tant que la console est active ;
- il y a les fonctions que l'on peut écrire dans une feuille de script. Il convient d'exécuter la feuille pour que les fonctions soient chargées.

Remarque : L'analyse descendante consiste à décomposer un programme en fonctions élémentaires. Les avantages sont multiples :

- limiter la programmation à des fonctions de tâche *simple*
- les fonctions élémentaires sont faciles à corriger
- les fonctions élémentaires peuvent être réutilisées pour d'autres programmes. On retrouve ici, l'intérêt des modules.

Attention ! Le terme **procédure** est réservé aux fonctions qui opèrent des transformations (sur des variables mutables ou qui affiches des résultats ou des graphiques) mais qui ne retournent pas de variables.

I. Définition d'une fonction

I.1. Les fonctions lambda

Cette approche s'applique aux fonctions numériques *simples*, la définition se fait en une ligne d'instructions :

```
nom=lambda parametre(s) d entree:expression de sortie
```

$f : t \mapsto t^2$

```
>>> f=lambda t:t**2
>>> type(f)
<class 'function'>
>>> f(5)
25
```

$h : (x,y,z) \mapsto (x-y, y-z)$

```
>>> h=lambda x,y,z:(x-y,y-z)
>>> h(5,2,1)
(3, 1)
>>> (a,b)=h(5,2,1)
>>> a;b
3
1
```

I.2. Les fonctions définies avec def

Cadre générale de définition d'une fonction, d'une procédure :

```
def nom(parametre(s) d entree)
    instructions
   [return sortie]
```

$h : (x,y,z) \mapsto (x-y, y-z)$

```
>>> def h(x,y,z):
...     return x-y,y-z
...
>>> type(h)
<class 'function'>
```

➜ La ou les valeurs de sortie sont retournées par l'instruction `return`.

➜ L'éxécution de `f(x)` évalue d'abord x puis exécute f avec la valeur calculée : ceci est *l'appel de fonction par valeur*.

Attention ! Une fonction peut ne pas avoir de paramètre d'entrée, ni de valeur retournée. Elle peut aussi inclure une interactivité avec l'utilisateur :

```
def bonjour():
    nom=input('Votre prenom : ')
    print('Bonjour',nom,'!')
```

```
>>> bonjour()
Votre prenom : Matt
Bonjour Matt !
```

➜ Il est possible de définir une fonction dans les paramètres d'entrée ou de sortie ou encore à l'intérieure d'une autre fonction :

```
def translate(f,a):
    '''effectue une translation de f
    de vecteur a selon (Ox)
    avec f une fct d une variable '''
    return lambda t:f(t-a)

g=translate(lambda x:(x+1)**2,1)
```

Ce qui donne :

```
>>> g(5)
25
```

Nous aurions pu aussi écrire :

```
def translate(f,a):
    def F(t):
        return f(t-a)
    return F
```

➜ Les trois apostrophes permette d'écrire un commentaire contenant des retours à la ligne. De plus, en début de fonction, ce permet de définir la **signature** de la fonction : ce sont les informations qui décrivent ce que fait la fonction et qui spé-

cifient chaque variable d'entrée et de sortie. La signature est l'explication nécessaire et suffisante pour pouvoir appeler la fonction ; elle est affichée par l'instruction `help( )`.

Avec l'exemple précédent cela donne :

```
>>> help(translate)
Help on function translate in module __main__:

translate(f, a)
    effectue une translation de f
    de vecteur a selon (Ox)
    avec f une fct d une variable
```

Nous prendrons l'habitude de spécifier systématiquement les signatures.

➜ Il est possible de définir des valeurs par défaut dans les paramètres d'une fonction :

```
def puissance(x,a=2):
    return x**a
```

```
>>> puissance(5,3)
125
>>> puissance(5)
25
```

I.3. L'INSTRUCTION assert

L'instruction `assert` permet de conditionner l'exécution de la fonction à une condition et d'afficher un message d'erreur si cette dernière n'est pas vérifiée : c'est un moyen de vérifier la nature des paramètres d'entrée.

Utilisation de assert

```
def parite(n):
    assert type(n)==int, "n_doit_etre_entier"
    if n%2==0:
        print('pair')
    else:
        print('impair')
```

```
>>> parite(1.2)
AssertionError: n doit etre entier
```

II. VARIABLES GLOBALES ET LOCALES

Lorsqu'une expression fait référence à une variable à l'intérieur d'une fonction, PYTHON cherche la valeur définie à l'intérieur de la fonction et à défaut la valeur dans l'espace global de l'environnement et du module : ceci s'appelle la *portée lexicale* des variables

II.1. VARIABLES LOCALES

Les **variables locales** sont les variables introduites dans la fonction ; leur rôle est interne à la fonction.

Factorielle

```
def factorielle(n):
    assert type(n)==int and n>=0, "entier_naturel"
    p=1
    for i in range(1,n+1):
        p=p*i
    return p
```

```
>>> factorielle(5)
120
```

Les variables locales sont : `p` et `i`.

- Ces variables sont crées à l'appel de la fonction et détruites à la fin de l'exécution.

```
>>> p
... NameError: name 'p' is not defined
>>> factorielle(5)
120
>>> p
... NameError: name 'p' is not defined
```

- PYTHON crée une variable spécifique lors du déroulement de la fonction ; le nom utilisé lors de la définition n'a donc aucune importance, PYTHON fait la distinction entre une éventuelle variable existante, de même nom, et celle locale : on pourrait dire que les variables locales sont *muettes*.

```
>>> p=12
>>> factorielle(5)
120
>>> p
12
```

- On peut effectuer des appels successifs de fonctions utilisant des variables locales *apparemment* de même nom. PYTHON ne fait aucune confusion.
- Le nom des variables locales, lors de la définition n'a aucune importance. Il est pratique de donner un nom en lien avec l'objet représenté, mais ceci ne constitue qu'un agrément de relecture.

II.2. VARIABLES GLOBALES

Les **variables globales** sont les variables définies avant l'appel d'une fonction. Ces variables sont toujours accessibles lors de l'exécution de la fonction.

- Les variables globales sont directement accessibles mais non modifiables :

```
def op2(a):
    a=a+2
    return a+b
```

```
>>> n,b=5,1
>>> op2(n)
8
```

```
def op2(a):
    a,b=a+2,b+1
    return a+b
```

```
>>> n,b=5,1
>>> op2(n)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<tmp_1>", line 2, in op2
    a,b=a+2,b+1
UnboundLocalError: local variable 'b' referenced
before assignment
```

- Pour pouvoir modifier une variable globale, il convient de préciser son statut par la mention `global`

```
def op2(b):
    global a
    a=a+2
    return a+b
```

```
>>> a,p=5,1
>>> op2(p)
8
>>> a
7
```

Attention ! La modification des variables globales provoque des *effets de bords* générateurs d'erreurs difficiles à corriger. Éviter tant que possible leur modification.

II.3. Variables globales passées par paramètres

Les variables globales peuvent être utilisées comme paramètres d'entrée d'une fonction.

- Si la variable est immuable (`int`, `str`, `tuple`, etc.) alors elle se comporte comme une variable locale : les modifications de la variable ne sont pas conservées à la fin de la fonction.

```
def op(a):
    a=a+2
    return a
```

```
>>> n=5
>>> op(n)
8
>>> n
5
```

- Si la variable est mutable (`list`, `array`, etc.) alors la modification d'une valeur à une portée globale : on put changer une valeur, ajouter un élément ou en supprimer un.

```
def g(L):
    L[0][1]=12
    L[1]=[1,2]
    L.append(3)
```

Ce qui donne :

```
>>> L=[[1,2,3],[4,5]]
>>> g(L)
>>> L
[[1, 12, 3], [1, 2],3]
```