Informatique 4 : Compléments sur la programmation

I Imbrication de plusieurs fonctions

On a déjà vu en cours et en TP qu'on pouvait charger des **modules de fonctions**. Lorsqu'on crée plusieurs fonctions dans un même fichier à l'aide de l'éditeur de texte, nous créons notre propre module de fonctions. Cette façon de procéder permet d'appeler des fonctions déjà créées à l'intérieur d'autres fonctions. Nos fichiers textes sont structurés de la manière suivante :

1. des commandes pour importer d'éventuelles fonctions prédéfinies dans Python,
2. des **fonctions secondaires** nécessaires à la suite du programme,
3. la partie principale du fichier texte, appelée aussi **fonction principale**.

Ainsi, quand on cherche à comprendre un script, on veillera à le déchiffrer à l'envers, en commençant par la **fonction principale**, puis en étudiant les fonctions secondaires lorsqu'elles sont appelées.

Exemple 1

1. Écrire une fonction `test(n)` qui renvoie `'argument non entier'` si n n'est pas entier et ne fait rien sinon.
2. En utilisant la fonction précédente, écrire une fonction `facto(n)` qui renvoie $n!$ lorsque n est un entier et qui renvoie `'argument non entier'` sinon.

Exemple 2

On appelle nombre parfait tout entier $n \in \mathbb{N}$ dont la somme des diviseurs est égale à $2n$. Construire une fonction `parfait(n)` qui pour un entier donné n teste s'il est parfait. On construira :

1. une fonction secondaire `diviseurs(n)` qui renvoie la liste des diviseurs de n,
2. la fonction `parfait` en faisant appel à la fonction `sum` qui somme tous les éléments d'une liste.

II Commandes de contrôle de boucle et gestion des erreurs

1 Commandes `pass`, `continue` et `break`

Certaines commandes Python nous permettent d'imposer certaines commandes qui faciliteront la gestion d'un programme contrôlant l'exécution des structures conditionnelles et itératives.

La commande `pass` n'apporte aucune instruction, mais elle permet de structurer visuellement un programme en ajoutant une ligne sans instruction : par exemple dans une structure conditionnelle, il peut arriver que lors d'une commande `else`, il n'y ait rien à faire. C'est notamment dans ce type de cas que cette commande s'avère utile.

Exemple 3

Á l'aide de la fonction `diviseurs`, écrire une fonction Python `premiers(n)` qui donne la liste de tous les nombres premiers plus petits que n. On utilisera la commande `pass` et une boucle `while`.

Remarque 4

Les commandes `continue` et `break` peuvent être utiles dans la mise en place des boucles : la commande `continue` oblige le programme à continuer la boucle à l'étape suivante, la commande `break` interrompt celle-ci et arrête l'exécution de la fonction.

Exemple 5

Déterminons ce qu'affichent les fonctions `testa` et `testb` données par :

```
def testa() :
    for i in range(5):
        if i==3:
            break
        print i
```

```
def testb() :
    for i in range(5):
        if i==3:
            continue
     print i
```

2 Gestion des erreurs

Quand on construit un script d'instructions, on essaie souvent d'éviter les **erreurs sémantiques** ou les **erreurs syntaxiques**. Par contre, à l'exécution, il n'est pas rare de voir l'interpréteur renvoyer des erreurs dont la nature sera souvent précisée :

```
>>> variable
Traceback (most recent call last) :
(...)
NameError : name 'variable' is not defined
```

Dans le langage Python, on parle plutôt d'**exceptions** dont les plus courantes sont :

type	interprétation
`NameError`	
`TypeError`	
`ZeroDivisionError`	
`IndexError`	
`ImportError`	
`OverflowError`	

Exemple 6

On réutilisera les programmes précédents. Chaque ligne suivante provoque une erreur. Déterminer de quel type d'erreur il s'agit :

1.
```
>>> diviseurs(1.5)
```
2.
```
>>> L = [1,2] ; L[2]
```
3.
```
>>> 5+[1,2]
```

1.

2.

3.

On peut forcer l'interpréteur à lever une exception grâce à la commande `raise`, et ceci en ajoutant un commentaire adéquat. Par exemple, pour calculer les images par la fonction

$$x \mapsto \frac{x^2}{x-1},$$

on pourra écrire :

```
def fonction(x):
    if x != 1:              # on teste s'il s'agit d'une valeur interdite
        return x**2/(x-1)
    else:
        raise ZeroDivisionError('attention au domaine de définition')
```

Exemple 7

Écrire la fonction `facto(n)` qui calcule $n!$ et qui renverra une erreur lorsque n n'est pas un entier avec le message « `l argument n est pas un entier` »

```

```

3 Exceptions et essais

Même lorsque l'interpréteur de Python rencontre une erreur, on peut contrôler son comportement selon la nature de l'exception : on peut tout à fait ordonner l'exécution de nouvelles séquences d'instructions grâce aux commandes `try`, `except` et `else`. On les présente sous la forme :

`try` :
indentation Séquence d'instructions
(...)
`except` *exception 1* :
indentation Instructions alternatives
`except` *exception 2* :
indentation Instructions alternatives
(...)
`else` :
indentation Séquence d'instructions si
aucune exception n'a été levée
(...)

On peut alors réécrire le programme précédent :

```
def fonction(x):
    try:
        image = x**2/(x-1)
    except ZeroDivisionError:
        print('en x = '+str(x)+' la fonction n'est pas définie')
    else:
        return image
```

Remarque 8

Exemple 9

Réécrire la fonction `facto(n)` précédente avec cette nouvelle méthode.

III Programmation itérative ou récursive

Précédemment, nous avons été amenés à utiliser des structures itératives, qu'il s'agisse des boucles `for` ou `while`. nous allons voir une manière de programmer qui permettra de s'affranchir de ces boucles dans la plupart des cas.

1 Un premier exemple : la fonction factorielle

On peut définir la factorielle de deux façons :

1. **de façon itérative** avec $n! = \begin{cases} 1 \text{ , si } n = 0 \\ 1 \times \cdots \times n \text{ , sinon} \end{cases}$
2. **de façon récursive** avec $n! = \begin{cases} 1 \text{ , si } n = 0 \\ n.(n-1)! \text{ , sinon} \end{cases}$

En Python, si on suit la première définition de la factorielle, on définit la fonction `facto_iter(n)` de la façon suivante :

```
def facto_iter(n)
    if n==0:
        return 1
    else:
        P = 1
        for i in range(1,n+1)
            P = P*i
        return P
```

En Python, il est possible d'appeler la fonction que l'on est en train de créer à l'intérieur de cette même fonction. Réécrivons la fonction donnant la factorielle avec la seconde définition sous le nom `facto_recur(n)`

2 Notion de récursivité

Définition 10

Un algorithme **récursif** est un algorithme qui résout un problème en calculant des solutions d'instances plus petites du même problème.

Remarque 11

Exemple 12

On considère la suite (S_n) définie pour tout $n \in \mathbb{N}$ par :

$$S_n = \sum_{k=0}^{n} \frac{(-1)^k}{2k+1}.$$

Construisons deux fonctions Python `S_iter(n)` et `S_recur(n)` donnant la valeur de la somme pour un entier n donné.

1. *Programmation itérative* :

2. *Programmation récursive* :
   On a la relation :

Remarque 13 (*Une limite de la programmation récursive*)

Exemple 14 (*Suite de Fibonacci*)

Écrivons deux fonctions Python `fibo(n)` donnant la valeur de u_n définie par :

$$\begin{cases} u_0 = u_1 = 1 \\ \forall\, n \in \mathbb{N},\ u_{n+2} = u_{n+1} + u_n \end{cases}.$$

On utilisera la programmation itérative puis la programmation récursive.

1. *Programmation itérative.*

2. *Programmation récursive.* On commence par réécrire la relation :