CPES Scientifique

Informatique

TP 3

Fonctions

Résumé

Ce TP introduit la notion fondamentale de **fonction** Python.

1 Premiers pas dans l'éditeur

Les questions suivantes utilisent la fenêtre gauche de Spyder. Il peut être nécessaire d'attendre les explications au tableau.

Question 1.1 (Premiers pas dans l'éditeur)**.** *On considère des variables « `u` » et « `v` » non encore initialisées :*

```
>>> u
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'u' is not defined
>>> v
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'v' is not defined
```

Utiliser l'éditeur pour qu'après exécution du fichier, on ait dans la console

```
>>> u
False
>>> v
50
```

2 Fonctions

Comme tout langage de programmation, Python permet de définir et d'utiliser ce que l'on appelle des **fonctions**.

Remarque 2.1. *Pour définir des fonctions, il est beaucoup plus pratique de travailler dans l'éditeur. Les programmes Python écrits dans un fichier ouvert dans l'éditeur peuvent être exécutés dans la console gâce à la commande appropriée.*

Les fonctions permettent de donner un nom à une suite d'instructions. Cette suite d'instructions peut renvoyer un résultat et/ou modifier l'état de la mémoire ; elle peut dépendre ou non de **paramètres** (ou arguments).

Les fonctions sans paramètres sont définies comme suit :

```
def nom_de_la_fonction ():
    #Corps de la fonction
```

Le « `nom_de_la_fonction` » doit être un indentificateur Python (voir le TP02). Le « **Corps de la fonction** » contient les instructions qui doivent être exécutées lorsque le fonction est appellée.

Par exemple, considérons la fonction :

```
def return_false ():
    return False
```

Aprés avoir exécuté le fichier dans lequel on a écrit la fonction, on peut l'appeler dans la console :

```
>>> return_false ()
False
```

Les instructions du corps de la fonction peuvent initialiser des variables et les utiliser. Par exemple, avec

```
def return_false_bis ():
    x = False
    return x
```

on obtient :

```
>>> return_false_bis ()
False
```

Question 2.1. *Utiliser l'éditeur pour définir les fonctions «* `return_false` *» et «* `return_false_bis` *» ci-dessus. Les utiliser dans la console.*

Les fonctions peuvent aussi avoir des **paramètres** (ou **arguments**). Les fonctions avec paramètres sont définies de la manière suivante :

```
def nom_de_la_fonction (parametres_formels):
    #Corps de la fonction
```

Dans ce cas, le corps d'une fonction peut aussi utiliser des variables déclarées dans les « `parametres_formels` ». Dans le corps de la fonction, ces identificateurs sont utilisés comme des variables.

Par exemple, considérons :

```
def f (x):
    return x + 1
```

Pour exécuter une fonction, il suffit de lui donner ses paramètres entre parenthèses. Par exemple, dans

```
>>> f(1)
2
```

La fonction « `f` » est exécutée en prennant la valeur initialle `1` pour la variable « `x` ».

Une fonction peut bien entendu avoir plusieurs paramètres formels. Dans ce cas, ils doivent être séparés par des virgules :

```
def g (x,y):
    return x + y
```

Question 2.2 (Test de parité)**.** *Écrire et tester une fonction Python qui prend en argument un entier et qui renvoie « `True` » si cet entier est pair et « `False` » sinon.*

Question 2.3 (« **ou exclusif** »)**.** *Écrire et tester une fonction Python implémentant la fonction « `xor` » dont la table de vérité est la suivante.*

x `xor` y		y	
		`True`	`False`
x	`True`	`False`	`True`
	`False`	`True`	`False`

Lorsqu'on appelle une fonction, on doit lui donner exactement le nombre d'arguments qu'elle attend. Par exemple, la fonction « `f` » ci-dessus attend exactement un argument, et on peut observer :

```
>>> f(1)
2
>>> f ()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes exactly 1 argument (0 given)
>>> f (1,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes exactly 1 argument (2 given)
```

Remarque 2.2 (Sur l'exécution des fonctions avec paramètres)**.** *Lorsqu'une fonction est appelée, les instructions du corps sont exécutées. Dans ces instructions, les identificateurs correspondant à des paramètres formels sont vus comme des variables dont les valeurs initialles sont données par les arguments effectifs donnés à la fonction. La partie 3 ci-dessous donne des compléments sur l'utilisation des variables dans les fonctions.*

Remarque 2.3. *Les fonctions sont en fait stockées dans des emplacements mémoires qui peuvent, en Python, être manipulés par les variables, au même titre que toutes les valeurs littérales déjà vues.*

3 Utilisation des variables dans les fonctions

Ainsi, lorsqu'on utilise une variable dans une fonction, plusieurs cas peuvent se produire selon que cette variable soit ou non un argument de la fonction.

En Python, la règle est la suivante :

— Lors de l'exécution d'une fonction, Python recherche les définitions des identificateurs d'abord dans le corps de la fonction, ensuite parmi les variables déclaréees dans l'environnement qui appelle la fonction.

Considérons la fonction

```
def f(x):
    return x
```

La variable « `x` » est un paramètre de la fonction. Lorsque la fonction « `f` » est exécutée, la variable « `x` » est **locale** à la fonction « `f` », elle est indépendante des variables du même nom initialisées à l'extérieur de la fonction.

Question 3.1. *Considérons la fonction « `f` » définie ci-dessus.*

(a) Donner la valeur de la variable « `x` » après l'exécution des instructions ci-dessous :

```
>>> x = 0
>>> f(1)
```

(b) Même question avec la suite d'instructions :

```
>>> a = 4
>>> f(a)
```

En particulier, changer à l'intérieur d'une fonction la valeur d'une variable locale n'impacte pas la valeur d'une variable du même nom définie à l'extérieur de la fonction.

Question 3.2. *Considérons la fonction « `g` » ci-dessous :*

```
def g(y) :
    y = 1
    return y
```

(a) Donner la valeur de la variable « `y` » après l'exécution des instructions suivantes :

```
>>> y = 0
>>> g(0)
>>> y
```

(b) Même question avec la suite d'instructions :

```
>>> y = 0
>>> g(y)
>>> y
```

Les variables locales peuvent ne pas être des arguments de la fonction, comme par exemple la variable « `zz` » dans la fonction « `k` » de la Question 3.3 ci-dessous.

Question 3.3. *Considérons la fonction « `k` » ci-dessous :*

```
def k(x) :
    zz = 1
    return zz
```

(a) Que se passe-t-il lors de l'exécution de la suite d'instructions suivante :

```
>>> zz
>>> k(0)
>>> zz
```

(b) Même question avec la suite d'instructions :

```
>>> zz=0
>>> k(0)
>>> zz
```

Enfin, une fonction peut aussi utiliser des variables qu'elle n'initialise pas, comme par exemple la variable « `yy` » dans la fonction « `h` » de la Question 3.4 ci-dessous.

Question 3.4. *Considérons la fonction « `h` » définie* **dans la console** *comme ci-dessous :*

```
>>> def h(x) : return yy
...
>>>
```

(a) Que se passe-t-il lors de l'exécution de la suite d'instructions suivante :

```
>>> yy
>>> h(0)
```

(b) Même question avec la suite d'instructions :

```
>>> yy = 1
>>> h(0)
```

(c) Même question avec la suite d'instructions :

```
>>> yy = 2
>>> h(0)
```