Région académique
PAYS DE LA LOIRE

Gestion des entrées et sorties en Python :

Objectif : Le but de ce document est de recenser les différentes façons de programmer les entrées et sorties des programmes créés en Python. En effet suivant les situations pédagogiques chaque forme peut avoir ses avantages et ses inconvénients.

1 Le typage d'une variable

La première chose à se demander lorsque l'on va utiliser une variable, c'est son type. En scratch, il n'existe que deux éléments différents : les variables et les listes. Python offre plusieurs possibilités, les listes sont toujours présentent mais les variables sont de trois types distincts : entier (`int`), décimal (`float`) ou textuel (`str`).

En Scratch

En Python

```
>>> n=2
>>> m=3
>>> n+m
5
```

```
>>> x=2.0
>>> y=3.0
>>> x+y
5.0
```

```
>>> a='2'
>>> b='3'
>>> a+b
'23'
```

On peut voir que la simple addition ne donne pas le même effet suivant les types. On pourra aussi tester que entiers et décimaux sont compatibles entre eux mais pas avec du texte. En cas de problème de type, voici le message obtenu :

```
>>> 2+'3'
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    2+'3'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Le problème du typage est donc essentiel tant en entrée pour pouvoir ensuite effectuer les calculs voulus, qu'en sortie si l'on veut réutiliser le résultat obtenu.

2 Exemples de solution d'entrée-sortie

Voici quatre programmes qui permettent le calcul de la fonction $f(x) = x^2 - 2x + 3$.

Programme 1 :

```
# la fonction f:
x=float(input("valeur de x?"))
y=x**2-2*x+3
print(y)
```

Programme 2 :

```
def f():
    x=float(input("valeur de x?"))
    y=x**2-2*x+3
    return y
```

Programme 3 :

```
def f():
    x=float(input("valeur de x?"))
    y=x**2-2*x+3
    print(y)
```

Programme 4 :

```
def f(x):
    y=x**2-2*x+3
    return y
```

Il est intéressant d'essayer avec ces quatre programmes de calculer $f(1)$, $f(5)$, $f(10)$ et $4/f(4)$, pour les voir les avantages et inconvénients de chacun.

3 Les entrées : def et input

L'utilisation de `def` permet de créer un programme que l'on pourra appeler plusieurs fois sans recompiler le fichier d'origine.

- On peut donc s'en passer lorsque le programme est prévu pour effectuer une tache bien précise : pour quelle valeur de n la suite $u_n = 2^n$ passe au dessus de 10^9,

```
n=0
while 2**n<10**9:
    n=n+1
```

- Par contre, si on va effectuer plusieurs fois la même action, l'utilisation de `def` devient nécessaire : lancer un dé à 6 faces pour avoir un résultat aléatoire,

```
from random import*

def dé6():
    d=randint(1,6)
    print(d)
```

Il suffit alors de taper `dé6()` dans la console pour relancer un dé.

D'autre part, on peut compléter la commande `def` d'un argument, ou utiliser la commande `input` pour permettre d'adapter un paramètre. Par exemple, si l'on veut pouvoir changer le nombre de faces du dé :

Idée 1

```
from random import*

def dé(n):
    d=randint(1,n)
    print(d)
```

Idée 2

```
from random import*

def dé():
    n=int(input('Nombre de faces?'))
    d=randint(1,n)
    print(d)
```

La première idée a le défaut de typer de manière implicite n et mais pourra être appelée par un autre programme si nécessaire. La deuxième nécessite l'intervention d'une personne pour répondre à une question, cela rend ainsi le programme plus compréhensible pour un utilisateur et donc plus simple à utiliser.

On pourra ainsi se servir de la commande input dans les premiers programmes que l'on fournira aux élèves. Il s'agira alors de comprendre ce que fait le fichier fourni. Par contre, lors du passage à la production d'un programme par l'élève, il me semble nécessaire d'introduire rapidement `def` pour profiter pleinement de l'aspect fonction du langage Python et simplifier l'écriture des programmes.

4 Les sorties : print ou return ?

La commande `print` permet l'affichage d'un texte ou d'un nombre à l'écran dans la console, sans stopper l'exécution du programme. Elle prend donc tout son sens pour afficher plusieurs résultats intermédiaires. Par exemple, on peut rechercher l'ensemble des diviseurs d'un nombre :

```
def diviseurs(n):
    d=1
    while d*d<n:
        if n%d==0:
            print(d,' et ',n//d)
        d=d+1
```

La commande `return` est ne peut être utilisée qu'après la commande `def`. Elle stoppe l'exécution de la fonction en renvoyant une réponse sous la forme d'une (ou plusieurs) variable(s) ou d'une liste. Cette réponse aura l'avantage de pouvoir être réutilisée dans une autre fonction. On peut ainsi programmer la suite de Syracuse puis avoir de le temps de retour à 1 pour une valeur de départ choisie :

```
def syracuse(n):
    if n%2==0:
        return n//2
    else:
        return 3*n+1
```

```
def retour1(n):
    k=0
    while n!=1:
        k=k+1
        n=syracuse(n)
    return k
```

Au final, la commande `return` est à privilégier dès que la sortie désirée est unique et elle devient obligatoire dans le cas d'une ré-utilisation. Il reste primordial cependant de se servir de la commande `print` par exemple lors de la phase de programmation. Elle permettra de faire apparaitre des étapes intermédiaires et de débugger un programme en cas de difficulté.

5 Bilan

Comme on peut le voir dans les différents exemples présentés, le choix des entrées-sorties dépend à la fois du programme recherché et du niveau de programmation de l'utilisateur.

Il reste néanmoins certain que, pour bénéficier vraiment de l'avantage du langage Python et de l'introduction des fonctions, la maitrise de commandes `def` et `return` doit être un objectif en soi.