Fonctions avec ou sans valeur de retour

Complément - niveau basique

Le style procédural

Une procédure est une fonction qui se contente de dérouler des instructions. Prenons un exemple d'une telle fonction

```
def affiche_carre(n):
    print "le carre de", n, "vaut", n*n
```

qui s'utiliserait comme ceci

```
affiche_carre(12)
```

Le style fonctionnel

Mais en fait, il serait dans notre cas beaucoup plus commode de définir une fonction qui retourne le carré d'un nombre, afin de pouvoir écrire quelque chose comme

```
carre15 = carre(15)
```

quitte à imprimer cette valeur ensuite si nécessaire - même si jusqu'ici nous avons fait beaucoup appel à `print`, dans la pratique, imprimer n'est pas un but en soi, au contraire bien souvent.

L'instruction `return`

Voici comment on pourrait écrire une fonction `carre` qui renvoie le carré de son argument:

```
def carre(n):
    return n*n

if carre(8) <= 100:
    print 'petit appartement'
```

La sémantique (le mot savant pour "comportement") de l'instruction `return` est assez simple. La fonction qui est en cours d'exécution s'achève tout de suite, et l'objet passé en argument à `return` est retourné à l'appelant, qui peut utiliser cette valeur comme n'importe quelle expression.

Le singleton `None`

Le terme même de fonction, si vous vous rappelez vos souvenirs de mathématiques, suggère qu'on calcule un résultat à partir de valeurs d'entrée. Dans la pratique il est assez rare qu'on définisse une fonction qui ne retourne rien.

En fait **toutes** les fonctions retournent quelque chose. Lorsque le programmeur n'a pas prévu d'instruction `return`, python retourne un objet spécial, baptisé `None`. Voici par exemple ce qu'on obtient si on essaie d'afficher la valeur de retour de notre première fonction, qui, on le rappelle, ne retourne rien:

```
carre15 = affiche_carre(15)
print 'carre15=', carre15
```

Ceci est notre première rencontre avec l'objet `None`, qui est un singleton prédéfini par python, un peu comme `True` et `False`. Ce n'est pas par contre une valeur booléenne, nous aurons l'occasion d'en reparler.

Un exemple un peu plus réaliste

Pour illustrer l'utilisation de `return` sur un exemple plus utile, voyons le code suivant

```
def premier(n):
    """Retourne un booléen selon que n est premier ou non
Retourne None pour les entrées négatives ou nulles"""
    # retourner None pour les entrées non valides
    if n <= 0:
        return
    # traiter le cas singulier
    elif n == 1:
        return False
    # chercher un diviseur dans [2..n-1]
    else:
        for i in range(2,n):
            if n%i == 0:
                # on a trouve un diviseur
                return False
    # a ce stade c'est que le nombre est bien premier
    return True
```

Cette fonction teste si un entier est premier ou non; il s'agit naturellement d'une version d'école, il existe bien entendu d'autres méthodes beaucoup plus adaptées à cette tâche. On peut toutefois vérifier que cette version est fonctionnelle pour de petits entiers comme suit. On rappelle que 1 n'est pas considéré comme un nombre premier:

```
for test in [-2, 1, 2, 4, 19, 35]:
    print "n", test, "premier", premier(test)
```

`return` sans valeur

Pour les besoins de cette discussion, nous avons choisi de retourner `None` pour les entiers négatifs ou nuls, une manière comme une autre de signaler que la valeur en entrée n'est pas valide.

Ceci n'est pas forcément une bonne pratique, mais elle nous permet ici d'illustrer que dans le cas ou on ne mentionne pas de valeur de retour, python retourne `None`.

`return` interrompt la fonction

Comme on peut s'en convaincre en instrumentant le code - que vous pouvez faire à titre d'exercice, dans le cas d'un nombre qui n'est pas premier la boucle for ne va pas jusqu'à son terme

On aurait pu d'ailleurs tirer profit de cette propriété pour écrire la fonction de manière légèrement différente comme ceci

```
def premier_bis(n):
    """Retourne un booléen selon que n est premier ou non
Retourne None pour les entrées négatives ou nulles"""
    # retourner None pour les entrées non valides
    if n <= 0:
        return
    # traiter le cas singulier
    if n == 1:
        return False
    # chercher un diviseur dans [2..n-1]
    for i in range(2, n):
        if n%i == 0:
            # on a trouve un diviseur
            return False
    # a ce stade c'est que le nombre est bien premier
    return True
```

Ici encore c'est une question de style et de goût. En tous cas, les deux versions sont tout à fait équivalentes, comme on le voit ici

```
for test in [-2, 2, 4, 19, 35]:
    print "n", test, premier(test) == premier_bis(test)
```