Conversion de bases et palindromes

Dans ce TP, on ne travaille qu'avec les bases 2 à 10.

I. Conversion

1. (a) Déterminer à la main l'écriture en base 10 du nombre 1101_2.

   (b) Exécuter à la main le programme suivant.

```
1 nBin=1101
2 nDec=0
3 puis2=1
4 while nBin!=0:
5     c=nBin%10
6     nDec=nDec+c*puis2
7     nBin=(nBin-c)//10
8     puis2=2*puis2
9 print(nDec)
```

nBin	nDec	puis2	c

   (c) Que renvoie ce programme ? Décrire par une phrase chacune de ses lignes.

   (d) À l'aide de ce programme, donner l'écriture en base 10 de 101010_2 et 1001101_2.
   Enregistrer le programme dans un fichier nommé `DuBinaireAuDecimal.py`.

2. (a) Déterminer à la main l'écriture de 23 en base 2.

   (b) En prenant pour modèle le programme précédent, écrire un programme qui renvoie l'écriture en base 2 d'un nombre écrit en base 10.
   Enregistrer ce nouveau programme dans un fichier nommé `DuDecimalAuBinaire.py`.

   (c) Vérifier les résultats obtenus à la question 1d.

3. Adapter les deux programmes précédents en remplaçant la base 2 par une base quelconque :
   - Les programmes sont nommés `DUneBaseAuDecimal.py` et `DuDecimalAUneBase.py` ;
   - Une variable `base` est utilisée pour enregistrer la base ;
   - Les variables `nBin` et `puis2` sont remplacées respectivement par `nBase` et `puisBase`.

II. Fonctions

Devoir changer la valeur à convertir à l'intérieur du programme n'est pas très pratique ; il y a plusieurs façon d'y remédier, comme par exemple en utilisant la commande `input`.
Dans ce TP, on se propose d'utiliser la notion de **fonction**.
La fonction suivante affiche, lors de son exécution, les carrés des entiers de 1 à 5.

```
>>> def carres():
        for i in range(1,6):
            print(i**2)
>>> carres()
1
4
9
16
25
```

Employée de cette façon, une fonction python s'apparente à une **procédure**, une portion de code exécutée à chaque appel de son nom.
Comme pour les fonctions mathématiques, on peut utiliser des **arguments**.

```
>>> def carres(debut,fin):
        for i in range(debut,fin+1):
            print(i**2)
>>> carres(2,4)
4
9
16
```

Certains arguments peuvent avoir des valeurs par défaut.

```
>>> def carres(fin,debut=5):
        for i in range(debut,fin+1):
            print(i**2)
>>> carres(8)
25
36
49
64
>>> carres(8,7)
49
64
>>> carres(debut=7,fin=8)
49
64
>>> carres(debut=7)
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    carres(debut=7)
TypeError: carres() takes at least 1 argument (1 given)
```

- 1 : les arguments possédant une valeur par défaut sont donnés après les autres.
- 4, 9 et 15 : les valeurs données en arguments concernent d'abord, et dans l'ordre, **toutes** les variables n'ayant pas de valeurs par défaut, puis **éventuellement** les autres, toujours dans l'ordre.
- 12 : pour modifier une variable particulière, on peut indiquer son nom (utile lorsqu'il y a plusieurs valeurs par défaut ou pour ne pas se soucier de l'ordre des variables).

Les fonctions ne modifient pas les valeurs des variables passées en arguments.

```
def test(n):
        n=n-1
        print(n)
>>> nombre=10
>>> test(nombre)
9
>>> nombre
10
>>> n=11
>>> test(11)
10
>>> n
11
```

- 1 : python crée un espace de noms propre à chaque fonction.

- 6 : la variable qui est modifiée se trouve dans cet espace.
- 8 : une fois sorti de cet espace, on retrouve la valeur initiale.
- 9 : c'est encore plus flagrant lorsqu'on utilise les mêmes noms de variables.

Pour modifier une variable extérieure à une fonction, on utilise le mot réservé `global`.

```
>>> n=11
>>> def test():
        n=n-1

>>> test()
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    test()
  File "<pyshell#41>", line 2, in test
    n=n-1
UnboundLocalError: local variable 'n' referenced before assignment
>>> def test():
        global n
        n=n-1

>>> test()
>>> n
10
```

- 6-11 : `n` n'a pas d'existence dans l'espace de noms de la fonction.

Pour qu'une fonction python se rapproche davantage d'une fonction plutôt que d'une procédure, on utilise le mot réservé `return`. La fonction retourne alors une valeur qui peut être réutilisée, dans une variable par exemple.

```
>>> def sommeCarres(debut,fin):
        S=0
        for i in range(debut,fin+1):
                S=S+i**2
        return S
>>> sommeCarres(1,5)
55
>>> sommeCarres(1,3)+sommeCarres(4,5)
55
```

1. Reprendre les codes de la partie I et les adapter pour créer deux fonctions (dans un même fichier) :
   (a) `baseDecimal(nBase,base)` : retourne l'écriture en base 10 du nombre `nBase` écrit en base `base` ;
   (b) `decimalBase(nDec,base)` : retourne l'écriture en base `base` du nombre `nDec` écrit en base 10.
2. À partir de ces fonctions, créer les deux fonctions suivantes (dans le même fichier) :
   (a) `baseBase(n,baseInitiale,baseFinale)` : retourne l'écriture en base `baseFinale` du nombre `n` écrit en base `baseInitiale` ;
   (b) `test()` : vérifie pour les bases 2 à 10 et entiers de 1 à 10 000 que les fonctions `baseDecimal` et `decimalBase` sont réciproques (en comptant le nombre d'erreurs `nbErreurs`).

III. Palindromes

1. À l'aide des fonctions créées dans la partie précédente, donner l'écriture en base 6 et en base 9 de 212, et l'écriture en base 5 et en base 7 de 2 601.
   Que remarque-t-on ?
   Dans ce TP, de tels nombres seront appelés **palindromes**.
2. Écrire une fonction `reflet(n)` qui, étant donné une suite de chiffres (donnée sous la forme d'un nombre entier), retourne la suite énumérée dans l'ordre contraire. On pensera à réutiliser les méthodes employées dans la partie 1.

```
>>> reflet(2569711)
1179652
```

3. Compléter la fonction suivante pour qu'elle renvoie `True` si le nombre `n` est un palindrome dans les bases `base1` et `base2`, et `False` sinon.

```
def palindrome(n,base1,base2):
    a=
    b=
    if a==reflet(b):
        return True
    else:
        return False
```

- 5 et 7 : `True` et `False` sont des **booléens** ; cette fonction pourra être utilisée directement dans un programme sous la forme :

```
if palindrome(153,7,9):
```

Tester cette fonction avec le nombre 10 dans les bases 2 et 3.
Modifier la fonction pour que de tels résultats soient exclus.

4. Créer une fonction `recherchePalindromes1(borne,base1,base2)` qui recherche et affiche les nombres (écrits en base 10) compris entre 1 et `borne` qui sont des palindromes dans les bases `base1` et `base2`. On affichera dans l'ordre : l'écriture en base 10, l'écriture en base `base1` et l'écriture en base `base2`.
5. Créer une fonction `recherchePalindromes2(borne)` qui recherche et affiche les nombres (écrits en base 10) compris entre 1 et `borne` qui sont des palindromes dans des bases différentes (comprises entre 2 et 10).
On affichera dans l'ordre : l'écriture en base 10, la base `base1`, l'écriture en base `base1`, la base `base2` et l'écriture en base `base2`.
Existe-t-il des nombres qui sont des palindromes dans plusieurs couples de bases ?

IV. Retour sur la commande `input`

1. Modifier comme suit la première ligne du fichier `DuBinaireAuDecimal.py` et exécuter le programme.

```
n=input()
nBin=int(n)
nDec=0
puis2=1
puis10=10
while nBin!=0:
    c=nBin%10
    nDec=nDec+c*puis2
    nBin=(nBin-c)//10
    puis2=2*puis2
print(nDec)
```

2. Utiliser la fonction `type` comme ci-dessous.

```
>>> type(nBin)
>>> type(10)
```

Cette fonction retourne le **type** des objets python. 10 est un nombre entier (`int` : **integer**) alors que `n` est une chaîne de caractères (`str` : **string**). Ce type de données permet de gérer les textes et sera traité dans un prochain TP.