Récursivité

Exercice

1. La fonction ordre est récursive car la fonction s'appelle elle-même.
   Le cas de base est pour $n = 0$, provoquant l'arrêt des appels récursifs et qui renvoie la valeur '0' pour l'appel `ordre(0)`.
2. L'appel `ordre(4)` renvoie la chaine de caractères '01234'.
3. Si on intervertit dans la concaténation l'appel récursif et str(n), cela affiche les valeurs de n au début de la chaine de caractères dans un ordre décroissant.

[43]:
```python
# Fonction récursive ordre
def ordre(n):
    if n == 0:
        return '0'
    else:
        return ordre(n-1)+str(n)

print(ordre(4))
```

```
01234
```

[44]:
```python
# Fonction récursive ordre
def ordre(n):
    if n == 0:
        return '0'
    else:
        return str(n)+ordre(n-1)

print(ordre(4))
```

```
43210
```

Exercice

1. La fonction **compte** est récursive car elle s'appelle elle-même.
   Le cas de base est pour $n = 1$ arrêtant les appels récursifs et renvoyant la valeur 1 pour l'appel `compte(1)`.
2. L'appel `compte(5)` renvoie la valeur 5 puisqu'on ajoute 1 à chaque appel récursif au nombre de 5.
3. Si on remplace la dernière instruction par `return 1+compte(n-2)`, on a deux cas différents:

- si n est impair, alors $n-2$ est toujours impair et finit par être égal à 1, arrêtant la récursivité et renvoyant la valeur 1 à l'appel `compte(1)`. Au final, on a compté les valeurs impaires soit $(n+1)/2$;
- si n est pair, alors $n-2$ est toujours pair et n'est donc jamais égal à 1. Les appels récursifs ne s'arrêtent pas provoquant une erreur de type **RecursionError: maximum recursion depth exceeded in comparison** soit trop d'appels récursifs. Le cas de base n'est jamais réalisé.

[1]:
```python
# Fonction récursive compte
def compte(n):
    if n == 1:
        return 1
    else:
        return 1 + compte(n-1)

print(compte(5))
```

```
5
```

[2]:
```python
# Fonction récursive compte
# Une erreur est provoquée si n est pair !
def compte(n):
    if n == 1:
        return 1
    else:
        return 1+compte(n-2)

print(compte(5))
```

```
3
```

Exercice

1. La commande **chr(65)** renvoie le caractère associé à la valeur décimale 65. Or $65_{10} = 41_{16}$ car $4 \times 16 + 1 = 65$.
   Dans la table ASCII, le caractère codé en hexadécimal 41 est la lettre majuscule **A**.
2. La boucle du script récupère les 26 lettres majuscules de l'alphabet et crée la chaine de caractères 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' qui est affichée avec la fonction **print**.
3. La version récursive est proposée ci-après.

[4]:
```python
# Version itérative
mot=''
for i in range(65,91): # ou range(int('41',16),int('5B',16))
    mot += chr(i)
print(mot)

# Version récursive
def affiche_recursif(i):
    if i == 90:
```

```
        return chr(90)
    else:
        return chr(i)+affiche_recursif(i+1)

print(affiche_recursif(65))
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Exercice

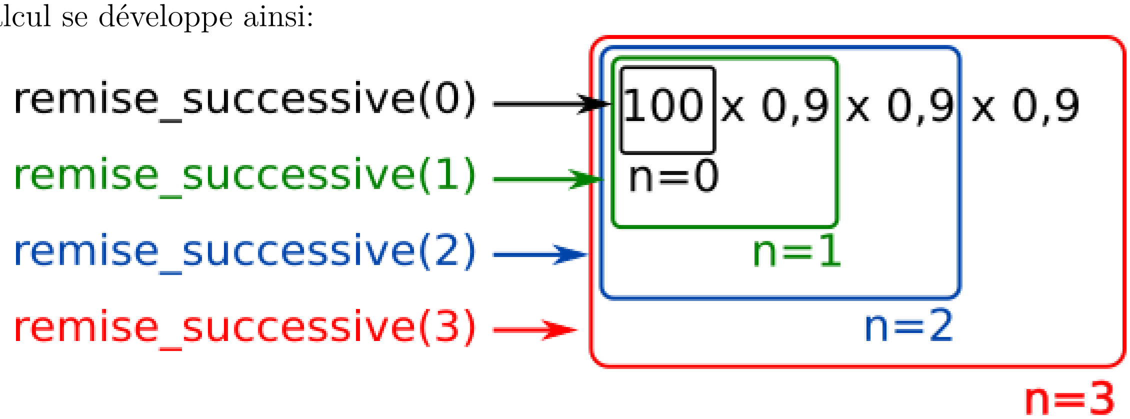
Lorsqu'on effectue une remise de 10% sur un prix, cela revient à multiplier ce prix par la valeur $1 - 10/100$.

On veut calculer des baisses successives de 10% sur une valeur, le nombre de remises étant défini à l'avance.

1. Calculer trois remises successives de 10% sur un prix de 100 €.
2. Montrer en détaillant le calcul qu'un algorithme récursif peut s'appliquer.
3. Écrire un script itératif qui calcule n remises successives de 10% sur un prix. On utilisera les variables **prix** et **n**. La variable **prix** contiendra la valeur finale après les remises.
4. Vérifier votre script avec un prix de 100 pour $n = 3$ remises.
5. Écrire la fonction récursive **remise_successive** qui calcule n remise de 10% sur un prix défini à l'avance.

Solution

1. $100 * (1 - 10/100)^3 = 100 * 0,9^3 = 100 * 0,729 = 72,9$
2. Le calcul se développe ainsi:



[ ]:
```
# 3. et 4.
# Version itérative d'une remise de 10 % successives
prix = 100
n=3
for i in range(n):
    prix = prix*(1-10/100) # prix *= (1-10/100)
print(prix)
```


[1]:
```python
# 5.
# Version récursive
def remise_successive(prix,n):
    if n == 0:
        return prix
    else:
        return (1-10/100)*remise_successive(prix,n-1)

print(remise_successive(100,3))
```

```
72.9
```

Exercice

Algorithme d'Euclide L'algorithme d'Euclide permet de trouver le **plus grand commun diviseur** de deux nombres entiers positifs a et b : on le note $PGCD(a;b)$.

Par exemple, le plus grand commun diviseur de 28 et 42 est 14 que l'on note $PGCD(28;42) = 14$.

[22]:
```python
def pgcd(a,b):
    if a<b:
        print(a,b)
        a,b=b,a
        print(a,b)
    if b==0:
        return a
    else:
        return pgcd(b,a%b)
```

[24]:
```python
print(pgcd(280,420))
```

```
280 420
420 280
140
```

[19]:
```python
a=28
b=42
if a<b:
    a,b=b,a
print(a,b)
```

```
42 28
```

[18]:
```python
28%14
```

[18]: 0

Exercice

Factorielle On donne le script de la fonction factorielle à compléter:

```
def factorielle(n):
    if ...:
        return ...
    else:
        return ...
```

[27]:
```
# version récursive

def factorielle(n):
    if n==1:
        return 1
    else:
        return n*factorielle(n-1)
```

[28]:
```
factorielle(5)
```

[28]: 120

[1]:
```
# version itérative avec while

def factorielle(n):
    p = 1
    while n>0:
        p *= n # p = p * n
        n -= 1 # n = n-1
    return p

factorielle(5)
```

[1]: 120

[2]:
```
# version itérative avec for

def factorielle(n):
    p = 1
    for i in range(1,n+1):
        p *= i # p = p * i
    return p

factorielle(5)
```

[2]: 120

Exercice

Dessin récursif avec Turtle à réaliser sur Thonny ou autre idle Python.

[1]:
```python
from turtle import *

couleurs=['blue','green','yellow','orange','red','purple']
bgcolor('black')

def dessin():
    for i in range(180):
        color(couleurs[i%6])
        forward(i)
        right(59)

dessin()
```

Exercice

Calcul récursif d'une puissance.

[18]:
```python
def puissance(x,n):
    if n==0:
        return 1
    else:
        #print(x,n)
        return x*puissance(x,n-1)
```

[21]:
```python
puissance(3,7)
```

```
3 7
3 6
3 5
3 4
3 3
3 2
3 1
```

[21]: 2187

[20]:
```python
"""
On peut améliorer l'algorithme des puissances en regardant la parité de␣
↪l'exposant.
- si n est pair, alors n=2p et x**n = (x**2)**p
- si n est impair, alors n=2p+1 et x**n = x(x**2)**p

Exemple:
- x**12 = (x**2)**6 = ((x**2)**2)**3 = ((x**2)**2)**2)x
```



```
soit 4 appels de la fonction au lieu de 12 avec la version classique.

- x**8 = (x**2)**4 = ((x**2)**2)**2
soit 3 appels de la fonction au lieu de 8 avec la version classique.

Remarque: pour un exposant égal à n=256, il y aura 256 appels récursifs avec la␣
 ↪première version
contre seulement 8 avec la seconde. En plus des appels récursifs, ce sont␣
 ↪nettement moins de calculs !
"""

def puissance2(x,n):
    if n==1:
        return x
    else:
        #print(x,n)
        if n%2==0:
            return puissance2(x**2,n//2)
        else:
            return x*puissance2(x**2,n//2)
```

[14]:
```
puissance2(3,256)
```

```
3 256
9 128
81 64
6561 32
43046721 16
1853020188851841 8
3433683820292512484657849089281 4
11790184577738583171520872861412518665678211592275841109096961 2
```

[14]:
```
13900845237714473276493978678966130311421885080852913799160482443003607262976643
5941001769154109609521811665540548899435521
```

Exercice

Algorithme mystère

[3]:
```
def mystere(n):
    if n<2:
        return str(n)
    else:
        return mystere(n//2)+str(n%2)
```

[4]:
```
mystere(10)
```

[4]:
```
'1010'
```


Complément à 2

En première, on a vu l'écriture binaire des nombres signés pour les nombres entiers relatifs.

On rappelle qu'un nombre signé positif a son bit de poids fort égal à 0 et un nombre négatif a son bit de poids fort égal à 1.

Méthode (1ère) La méthode consite à : - donner l'écriture binaire de la valeur absolue du nombre - donner le complément à 1 de cette écriture binaire, - puis le complément à 2 en ajoutant 1 au complément à 1.

La méthode du complément à 2 se généralise de la façon suivante: #### Propriété: Soit **n** le nombre de bits utilisés pour coder les entiers relatifs. On peut coder tous les nombres compris entre -2^{n-1} et $2^{n-1}-1$.

Exemple Par exemple, sur $n = 4$ bits, on peut coder les entiers compris entre $-2^3 = -8$ et $2^{n-1} - 1 = 8 - 1 = 7$

Méthode Pour coder un nombre r entier relatif : - il faut déterminer le nombre minimal de bits à utiliser, - si r est positif, on le code en binaire - si r est négatif, on code le nombre $r + 2^n$

Exemple Écriture binaire de -5

On a : $-8 < -5 < 7$ ce qui est équivalent à $-2^3 < -5 < 2^3 - 1$
On en déduit que $n = 4$ bits. L'écriture binaire de -5 est la même que son complément à 2^4 soit $-5 + 2^4 = 11$
Or $11_{10} = 1011_2$ donc $-5_{10} = 1011_2$

[5]:
```python
def binaire(r,n):
    if r>=0:
        return mystere(n)
    else:
        return mystere(r+2**n)
```

[8]:
```python
# 2^5=32 et 2^6=64
# on a -64 < -35 < 63 donc -2^6 < -35 < 2^6-1 donc n=7 bits
binaire(-35,7)
```

[8]: '1011101'

Exercice

Suite de Fibonacci La suite de fibonacci se construit par addition des deux nombres précédents: Les deux premiers nombres étant 0 et 1.

Donc : 0,1,1,2,3,5,8,13,21,34,etc.

[5]:
```python
def fibonacci(n):
    if n==0:
        return 0
```



```python
    elif n==1:
        return 1
    else:
        return fibonacci(n-2)+fibonacci(n-1)
```

[6]:

```python
for i in range(20):
    print(fibonacci(i))
```

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
```

