python

```
print("Hello, world!")
```

FORMATION PYTHON : Les fonctions

@IPEIN

Créer par : Anis SAIED

said_anis@hotmail.com

http://cahier-de-prepa.fr/info-ipein

Les fonctions en Python
Partie 1

1. Introduction
2. Déclaration et syntaxe
3. Passage de paramètres
4. Modules
5. Exercices

Introduction

- Python offre plusieurs fonctions prédéfinies prêtes à l'utilisation.
- On distingue les fonctions qui se chargent au moment du démarrage de l'interpréteur comme la fonction abs(), len()
  - abs(-1)
- D'autres qui se chargent au moment de l'importation de modules (cas de sqrt du module math)
  - From math import sqrt
  - Sqrt(2)

Définition

- Une fonction est un bloc d'instructions qui possède un nom, qui reçoit un certain nombre de paramètres et renvoie un résultat.
- L'usage des fonctions permet :
  - d'éviter la répétition
  - de rendre le code réutilisable à travers l'importation de fichiers de fonctions… (vu Par la suite)
  - De décomposer les tâches complexes en tâches simples plus abordables.

Syntaxe

- Une fonction Python a la syntaxe suivante :

```
>>> def NomFonction(paramètres):
        ''' Documentation
        ......
        '''
        bloc_instructions
```

- **def** sert à déclarer un objet de type fonction
- on ne précise jamais le type des paramètres (notion de typage dynamique)
- La documentation est optionnelle mais fortement recommandée (entre triple côtes/guillemets)
- L'instruction **pass** permet d'avoir un bloc d'instruction vide (si on n'a pas encore désigné de quoi est chargé notre fonction).

Première fonction en Python


```
def TableMul5():
    ''' affiche la table de multiplication de 5'''
    for i in range(1,11):
        print('{0}*{1}={2}'.format(5,i,5*i))
```

- La saisie peut être effectuée au niveau du shell ou dans un nouveau fichier .py
- Une fois déclarée, la fonction peut être appelée par son nom suivi par ses paramètres : TableMul5( )

Réutilisabilité

- Pour généraliser l'utilisation de cette fonction pour tous les autres entiers, On redéfinit TableMul en réservant un paramètre **n** qui représentera la valeur de l'entier dont on cherche à afficher sa table de multiplication :

```
def TableMul(n):
    for i in range(1,11):
            print('{0}*{1}={2}'.format(n,i,n*i))
```

Exemple d'appel : TableMul (4) ; TableMul(True)

Retour de résultat

- Le mot clé **return** permet d'arrêter l'exécution d'une fonction et de renvoyer un résultat.
- Tous les types vus en Python peuvent être utilisés comme un retour.
- Quand la fonction ne programme pas d'instructions de type return, Python retourne tout de même un objet vide ayant comme type **NoneType**.

Retour de résultat

- Exemple :

```
def som(a,b):
    s=a+b
    print(s)
```

```
>>> a=2
>>> b=4
>>> s=som(a,b)
6
```

- En vérifiant le type du résultat obtenu, on tombe sur le cas d'un objet vide, très logique **print** ne fait pas le travail de **return**

```
>>> s
>>> type(s)
<class 'NoneType'>
```

Retour de résultat

- Rectifions la fonction som de sorte qu'elle soit une vraie fonction avec un return :

```
>>> def som(a,b):
        return a+b

>>> s=som(5,4)
>>> s
9
>>> type(s)
<class 'int'>
```

Retour de résultat

- Le mot clé **return** peut figurer dans plusieurs endroits au sein d'une fonction.
- La fonction signe retourne une chaine exprimant le signe d'un argument x

```
def signe(x):
    if x>0:
        return 'strictement positif'
    elif x<0:
        return 'strictement négatif'
    else:
        return 'nul'
```

Retour de résultat

- Quand une fonction programme plus d'un paramètre en retour, le type **tuple** s'avère très utile.
- **Exemple** :
  Créer une fonction somdifprod qui calcule la somme, la différence et le produit de deux entiers a et b passés en paramètres, un objet de type tuple constitué des trois résultats sera alors retourné.

```
def somdifprod(a,b):
    return a+b,a-b,a*b
```

Retour de résultat

- Au moment de l'appel, il faut prendre en considération la nature du résultat. Soit qu'on utilise un objet qui stockera tout le tuple puis pour accéder aux résultats on utilise les indices respectifs suivant la taille de ce dernier, soit qu'on effectue un appel basé sur trois variables

```
>>> res=somdifprod(4,5)
>>> res[0];res[1];res[2]
9
-1
20
```

```
>>> s,d,p=somdifprod(4,3)
>>> s,d,p
(7, 1, 12)
```

Passage de paramètres

- L'appel est une substitution (replacement) des paramètres formels par des paramètres effectifs. Python différencie les données des résultats selon le type du paramètre :
- Les types non mutables (int, float, str, complex, bool, tuple…) sont passés par valeur.
- Les types mutables (listes, ensembles, dictionnaires …) imitent le mode de passage par variable **(VAR**, **S** et **ES** en algorithmique).

Passage par valeur

- La première version de la fonction permut est supposée échanger le contenu des deux variables a et b :

```
>>> def permut(a,b):
        a,b=b,a
        print(a,b)


>>> a=2;b=3
>>> permut(a,b)
3 2
>>> a;b
2
3
```

Le contenu des deux variables reste inchangé preuve que les types non mutables ne peuvent pas être modifiés.

Passage par valeur

- Pour réussir la permutation, l'appel doit écraser l'ancien contenu de a et de b en utilisant une affectation :

```
def permut(a,b):
    return b,a
```

```
>>> a=2;b=3
>>> a,b=permut(a,b)
>>> a;b
3
2
```

Passage par variable

- Les objets mutables sont modifiables par nature.
- La fonction ajout ajoute à la fin d'une liste l un objet x.

```
>>> def ajout(l,x):
        l.append(x)


>>> l=list(range(4))
>>> l
[0, 1, 2, 3]
>>> ajout(l,43)
>>> l
[0, 1, 2, 3, 43]
```

Passage par variable

- Exemple 1:

```
>>> def inc(x):
        '''
        Cette fonction incrémente x de 1
        '''
        print('En entrant dans inc():id(x)={}'.format(id(x)))
        x+=1
        print('Apres incrementation:id(x)={}'.format(id(x)))


>>> x=1
>>> inc(x)
En entrant dans inc():id(x)=1762244272
Apres incrementation:id(x)=1762244304

>>> print('Apres appel:id(x)={0} et x={1}'.format(id(x),x))
Apres appel:id(x)=1762244272 et x=1
```

Passage par variable

- Exemple 2:

```
>>> def ajout(l,x):
        print('En entrant dans la fonction id(l)={}'.format(id(l)))
        l.append(x)
        print('En quittant la fonction id(l)={}'.format(id(l)))


>>> l=[4,3,1]
>>> ajout(l,4444)
En entrant dans la fonction id(l)=64797000
En quittant la fonction id(l)=64797000
>>> l
[4, 3, 1, 4444]
>>> print('Apres execution de ajout id(l)={}'.format(id(l)))
Apres execution de ajout id(l)=64797000
```

Passage de paramètres : Valeur par défaut

- Une fonction peut être déclarée avec des paramètres ayant une valeur de départ.
- Si l'utilisateur omet l'un des paramètres au moment de l'appel, la valeur par défaut sera prise en considération

```
>>> def construire_point(x=0,y=0):
        print('(x,y)=({0},{1})'.format(x,y))


>>> construire_point()
(x,y)=(0,0)
>>> construire_point(4,3)
(x,y)=(4,3)
```

Passage de paramètres : Paramètres positionnels

- Python autorise d'appeler la fonction sans trop respecter l'ordre des arguments au moment de la déclaration.
- Il suffit d'utiliser le nom du paramètre au moment de l'appel et de préciser la valeur qu'il prend

```
>>> def aff(nom,prenom,age):
        print('nom=',nom)
        print('prenom=',prenom)
        print('age=',age)
```

```
>>> aff(age=20,prenom='XXX',nom='UUUU')
nom= UUUU
prenom= XXX
age= 20
```

Fin Partie 1

Les fonctions en Python (Partie 2)

1. Rappel
2. Nombre de paramètres variable
3. Variables locales et globales
4. Fonctions imbriquées

Rappel

- La syntaxe de base d'une fonction python :

```
def  nom_fonction([param1, param2,…]):
   corps de fonction
   [return valeur]
```

- Limiter le corps de la fonction par les indentations
- **Exercice :**

Créer une fonction fact(n) qui retourne le factoriel d'un entier n passé en paramètre

Rappel

- Solution:

```
>>> def fact(n):
        """Retourne le factoriel de nombre donné en paramètre."""
        r = 1
        while n > 0:
                r = r * n
                n = n - 1
        return r

>>>
```

- fact.__doc__
- help(fact)
- **r** est la valeur retournée au programme qui appelle fact(n)

Rappel

- Procédure ou fonction:
- En python on a seulement des fonctions
- Si **return** n'existe pas, Python retourne implicitement la valeur **None (return None)**

Passage de paramètres :
Nombre de paramètres variables

- Python permet de définir une fonction avec un nombre arbitraire de paramètres.
- Si l'appel de la fonction est basé sur un tuple de valeurs dont leurs nombre peut varier d'un appel à un autre, on précède la paramètre formel (généralement nommé : *args* ou *params*) du caractère *

Passage de paramètres :
Nombre de paramètres variables

- Exemple :

```
>>> def somme(*args):
        s=0
        for i in args:
            s+=i
        return s
```

- L'appel de la fonction est basé sur un tuple de valeurs

```
>>> somme()
0
>>> somme(4)
4
>>> somme(4,5,2.3)
11.3
```

```
>>> t=(1,2,3);somme(*t)
6
```

Passage de paramètres :
Nombre de paramètres variables

- Exemple :

```
>>> def f(x,y,z=1):
        print("x ={0}, y ={1}, z ={2}".format(x,y,z))
```

- L'appel de la fonction est basé sur un tuple de valeurs

Appel à f	x	y	z
f(1,2)	1	2	1
f(y=3,x=4)	4	3	1
t=(5,6,7);f(*t)	5	6	7
t=(5,6);f(*t)	5	6	1
t=(5,6);f(t)	(5,6)	vide	1
t=(5,);f(*t)	5	vide	1

Passage de paramètres :
Nombre de paramètres variables

Avantage :

- Utiliser ce modèle de fonctions permet une flexibilité de la programmation quand on ne sait pas vraiment quel sera le nombre de paramètres au moment de l'appel.

Passage de paramètres :
Nombre de paramètres variables

- Exemple 2 :

```
>>> def maximum(*numbers):
        if len(numbers) == 0:
                return None
        else:
                maxnum = numbers[0]
                for n in numbers[1:]:
                        if n > maxnum:
                                maxnum = n
                return maxnum


>>> maximum(3, 2, 8)
8
>>> maximum(1, 5, 9, -2, 2)

9
>>>
```

Passage de paramètres :
Nombre de paramètres variables

- Exemple 3 :

```
>>> def f(*args):
        for i in args:
            print(i)


>>> t=(1,2,8)
>>> f(*t)
1
2
8
>>> f(1,2,5)
1
2
5
>>> f(1,7)
1
7
>>> f(t)
(1, 2, 8)
```

Les variables locales

- Toutes les variables de la liste des paramètres d'une fonction, et toutes les variables créées dans une fonction sont des variables locales de la fonction

```
def f(var_local1, var_local2):
    var_local3=var_local1 + var_local2
```

Les variables locales

- Les changements des variables locales faites lorsque la fonction est en cours d'exécution n'ont aucun effet sur toutes les variables en dehors de la fonction.

```
def fact(n):
    r = 1
    while n > 0:
        r = r * n
        n = n - 1
    return r
```

```
r=0
Print(r)
Print(fact(6))
Print(r)
```

Les variables globales

- La fonction inc1 utilise la variable x sans la déclarer ni comme paramètre ni comme variable globale.

```
>>> def inc1():
        x+=1

>>> x=1
```

```
>>> inc1()
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    inc1()
  File "<pyshell#7>", line 2, in inc1
    x+=1
UnboundLocalError: local variable 'x' referenced before assignment
```

- (x=x+1) accès en lecture à une variable non initialisée. L'interpréteur ne reconnait pas x et déclenche une exception
- Pour travailler avec la valeur 1 de x dans inc1, il faut déclarer x comme variable globale à inc1

Les variables globales

- On peut explicitement déclarer une variable comme globale avant de l'utiliser, en utilisant le mot clé globlal suivi de nom de la variable

```
>>> def incl():
        global x
        x+=1
        print("x=",x)


>>> x=1
>>> x
1
>>> incl()
x= 2
>>> x
2
>>>
```

Les variables globales

- Une variable globale est accessible et modifiable par :
  - la fonction où elle a été déclarée
  - Autres fonctions qui la déclarent comme global
  - En dehors de toutes les fonctions

Les variables globales

- Voilà un exemple qui illustre la différence entre variables locales et globales

```
>>> def test():
        global a # a : variable globale
        a = 1
        b = 2 # b : variable locale


>>> a = "un"
>>> b = "deux"
>>> test()
>>> a
1
>>> b
'deux'
>>>
```

Les variables globales


```
>>> def inc2():
        global x
        x+=1


>>> x=1
>>> inc2()
>>> x
2
```

- Dans la deuxième version, inc2(), la variable x est déclarée globale d'une manière explicite.
- La variable est initialisée à l'extérieur de la fonction et son contenu est modifiable par les fonctions qui la déclarent comme globale avec le mot clé « global ».

Exercices : 1

- Qu'affiche le programme suivant ? Pourquoi ?

```
def g(x):
    global a
    a = 10
    return 2 * x

def f(x):
    v = 1
    return g(x) + v

a = 3
print(f(6)+a)
```

Exercices : 2

- Qu'affiche le programme suivant ? Pourquoi ?

```
def g(x):
    global v
    v = 1000
    return 2 * x

def f(x):
    v = 1
    return g(x) + v

a = 3
print(f(6)+a)
```

Les fonctions imbriquées

- Une ***fonction imbriquée*** ou ***fonction interne*** est une fonction encapsulée dans une autre.
- Elle ne peut être appelée que par la fonction englobante ou par des fonctions imbriquées dans la même fonction englobante.
- En d'autres termes, la portée de la fonction imbriquée est limitée par la fonction englobante.

Les fonctions imbriquées

- Exemple :

```
def max3(x, y, z):
    def max2(u, v):
        if u > v:
            return u
        else:
            return v
    return max2(x, max2(y, z))
```

```
>>> max3(6,-8,5)
6
```

Les fonctions imbriquées

- Donc, Il est possible de déclarer une fonction g à l'intérieur d'une fonction f.
- Seule la fonction f est capable d'appeler g.
- Cette dernière a un comportement local au sein de f.
- Python offre une commande qui permet de retourner une liste composée des noms des objets considérés comme locaux à une fonction : locals()

Les fonctions imbriquées


```
>>> def max3(x, y, z):
        def max2(u, v):
                print(locals())
                if u > v:
                        return u
                else:
                        return v
        print(locals())
        return max2(x, max2(y, z))

>>> max3(6,-8,5)
{'z': 5, 'max2': <function max3.<locals>.max2 at 0x020032B8>, 'x': 6, 'y': -8}
{'u': -8, 'v': 5}
{'u': 6, 'v': 5}
6
```

Remarque

- On peut assigner une fonction à une variable comme tout autre objet python
  - `X = fact`
  - `X(5)`
- On peut placer les fonctions dans des listes, tuples ou dictionnaires
  - `d = {'a': fact, 'b': fact1}`
  - `d['a'](5)`

FIN Partie 2

Les fonctions en Python Partie 3

1. La fonction lambda()
2. La fonction map()
3. La fonction filter()
4. Exercice

Les fonctions en Python

lambda ( )

Fonctions : Utilisations

- En Python, une fonction est une valeur comme une autre, c'est-à-dire qu'elle peut être passée en argument (1), renvoyée comme résultat (2) ou encore stockée dans une variable (3).

```
def g():
    return 4

def f(fonction):
    return fonction()+1

print(f(g))
```

1

```
def f():
    def g():
        return 4
    return g
print(f()())
```

2

```
def f():
    return 3
p=f
print(p())
```

3

Fonctions : Utilisations

- On peut assigner une fonction à une variable comme tout autre objet python
  - `X = fact`
  - `X(5)`
- On peut placer les fonctions dans des listes, tuples ou dictionnaires
  - `d = {'a': fact, 'b': fact1}`
  - `d['a'](5)`

Fonctions : Utilisations

- Exemple :

Calculer la somme $\sum_{i=0} f(i)$ pour une fonction f qui retourne le carré du paramètre n=10.

- Solution :

On commence par :

1. définir une fonction **somme_fonction(f, n)**

```
>>> def somme_fonction(f, n):
        s = 0
        for i in range(n+1):
            s = s + f(i)
        return s
```

Fonctions : Utilisations


```
>>> def somme_fonction(f, n):
        s = 0
        for i in range(n+1):
            s = s + f(i)
        return s
```

On défini une fonction **`carre(n)`**

```
>>> def carre(x):
        return x*x
```

- On appelle **`somme_fonction`** avec comme arguments : la fonction carré et la valeur 10

```
>>> somme_fonction(carre, 10)
385
```

lambda : définition

- Pour optimiser ce code, on peut même éviter de nommer la fonction **carre**, puisqu'elle est réduite à une simple expression, , cela nécessiterait d'y incorporer un concept supplémentaire (les *expressions lambda*)
- Une *fonction anonyme* s'écrit **lambda** x: *e*

où

e est une expression pouvant comporter la variable x.

lambda x: *e* désigne la fonction $x \rightarrow e(x)$.

lambda : définition


```
>>> def carre(x):
        return x*x
```

```
>>> def somme_fonction(f, n):
        s = 0
        for i in range(n+1):
                s = s + f(i)
        return s
```

```
>>> somme_fonction(carre, 10)
385
```

L'exemple précédent se réécrit-il plus simplement :

```
>>> somme_fonction(lambda x: x*x, 10)
385
```

lambda : définition

- En Python, une fonction lambda est une fonction :
  - anonyme (n'a pas de nom),
  - et qu'on peut appliquer "à la volée" dans une expression.
- La syntaxe est :

```
Lambda [Liste_de_paramètres] : expression_retournée
```

Liste_de paramètres : séparés par des virgules

expression_retournée : doit être un expression, ne contient pas : if, print(), etc.

$$f = \text{lambda } x : e \quad \Leftrightarrow \quad f : x \rightarrow e$$

lambda est mot réservé du langage.
e est l'expression pouvant comporter la variable x

Lambda : caractéristiques

- Les fonctions lambda sont réservées à des situations relativement simples.
- Leur définition ne peut pas contenir d'instructions composées (pas d'affectation, pas de boucle, etc.).
- Elles consistent donc essentiellement en la définition d'une expression calculée en fonction des paramètres qui lui sont passés.

Lambda : Exemple 1

- Exemple (simple et pas très utile):

Les deux définitions suivantes de la fonction f sont équivalentes :

```
>>> def f(x,y,z):
        return 100*x+10*y+z

>>> f=lambda x,y,z:100*x+10*y+z
```

lambda

- L'exemple suivant d'une fonction lambda retourne la somme de ses deux arguments:

```
Somme = lambda x, y: x + y

print(Somme(3,4))

7
```

- Nous aurions pu avoir le même effet en utilisant simplement la définition de la fonction classique suivante:

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

print(somme(3,4))

7
```

Lambda : Exemple 2


```
#fonction lambda sans arguments
f = lambda : a if True else b
a = 6
b = 7
Print(f())
```

```
f = lambda x,y : 2 * x + y

print(f)
<function <lambda> at 0x87d30>

print(f(3, 4))
10
```

Lambda : Exemple 3 (Composition)


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

def deux_fois(f):
    return lambda x: f(f(x))

print(deux_fois)
<function deux_fois at 0x01D75B70>

quad = deux_fois(carre)

print(quad)
<function deux_fois.<locals>.<lambda> at 0x01D759C0>

print(quad(5))
625
```

La fonction

map()

La fonction map()

- map () est une fonction qui prend deux arguments:

```
r = list(map(fonction, sequence))
```

- Le premier argument «**fonction**» est le nom d'une fonction et le deuxième est une «**sequence**» (par exemple une liste)
- Avec Python 3, map () renvoie un itérateur. (comme range())
- Un itérateur est un objet dont on peut parcourir ses éléments (conteneur)

la fonction map()

La syntaxe de map est la suivante :

```
list(map(f,L))
```

La fonction ***map()*** *:*

Applique une fonction ***f*** *sur tous les éléments d'une liste* ***L*** *pour former une nouvelle liste distincte de l'initiale.*

La fonction map()

- L'avantage de l'opérateur lambda peut être vu quand on l'utilise en combinaison avec la fonction map().

La fonction map() : exemple 1

Exemple :
Supposons qu'on veut ajouter 1 à tous les éléments d'une liste L :

```
f= lambda x : x+1

L=[1,2,3]

print(L)
[1, 2, 3]

L = list(map(f,L))

print(L)
[2, 3, 4]
```

La fonction map() : exemple 2

- Exemple :

```
def fahrenheit(T):
    return ((float(9)/5)*T + 32)

def celsius(T):
    return (float(5)/9)*(T-32)

temperatures = (36.5, 37, 37.5, 38, 39)
F = map(fahrenheit, temperatures)
C = map(celsius, F)

temperatures_in_Fahrenheit = list(map(fahrenheit, temperatures))
temperatures_in_Celsius = list(map(celsius, temperatures_in_Fahrenheit))

print(temperatures_in_Fahrenheit)
[97.7, 98.60000000000001, 99.5, 100.4, 102.2]

print(temperatures_in_Celsius)
[36.5, 37.00000000000001, 37.5, 38.00000000000001, 39.0]
```

La fonction map() : exemple 3

- Dans l'exemple précédent on n'a pas utilisé lambda.
- En utilisant lambda, nous aurions pas eu à définir et nommer les fonctions Fahrenheit fonctions () et Celsius () :

```
C = [39.2, 36.5, 37.3, 38, 37.8]
F = list(map(lambda x: (float(9)/5)*x + 32, C))

print(F)
[102.56, 97.7, 99.14, 100.4, 100.03999999999999]

C = list(map(lambda x: (float(5)/9)*(x-32), F))

print(C)
[39.2, 36.5, 37.300000000000004, 38.00000000000001, 37.8]
```

La fonction map() : arguments

- map () peut être appliquée à plus d'une liste.
- Les listes doivent avoir la même longueur.
- map () appliquera sa fonction lambda pour les éléments des listes d'arguments
- map() applique d'abord la fonction f aux éléments avec l'indice 0, puis aux éléments d'indice 1 et ainsi de suite jusqu'à ce que le dernier indice est atteint

La fonction map() : exemple 4

- Exemple :

```
a = [1,2,3,4]
b = [17,12,11,10]
c = [-1,-4,5,9]

list(map(lambda x,y:x+y, a,b))
[18, 14, 14, 14]

list(map(lambda x,y,z:x+y+z, a,b,c))
[17, 10, 19, 23]

list(map(lambda x,y,z : 2.5*x + 2*y - z, a,b,c))
[37.5, 33.0, 24.5, 21.0]
```

- Nous pouvons voir dans l'exemple ci-dessus que le paramètre x obtient ses valeurs à partir de la liste A, tout en y obtient ses valeurs de B et Z de la liste c.

La fonction

filter()

La fonction filter()

- Syntaxe : **`filter(fonction, sequence)`**
- La fonction filter offre un moyen élégant de filtrer tous les éléments d'une séquence **«`sequence`»**, pour lequel la fonction **«`fonction`»** retourne True.
- Un élément sera produit par le résultat de du filtre si l'élément est inclus dans la "séquence" et si la fonction renvoie True.

La fonction filter()

En d'autres termes:

- La fonction `filter(f, L)` a besoin d'une fonction f comme premier argument.
- f doit retourner une valeur booléenne, soit Vrai ou Faux.
- Cette fonction sera appliquée à tous les éléments de *la liste L*.
- Seulement les éléments où la fonction f retourne Vrai seront produites par l'itérateur, qui est la valeur de retour de **`filtre (fonction, séquence)`**.

La fonction filter() : Exemple 1

- **Exemple :** Supposons qu'on veut supprimer tous les multiples de 3 d'une liste d'entiers :

```
>>> g=lambda x:x%3
>>> l=list(range(20))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> l1=list(filter(g,l))
>>> l1
[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

- g retourne un entier (!=0 ⇔ True et ==0 ⇔ False)
- Filter produit en résultat seulement les entiers où g retourne True (x%3!=0 ⇔ non divisibles par 3)
- Donc filter supprime les entiers divisibles par 3

La fonction filter() : Exemple 2

Dans l'exemple suivant, nous filtrons d'abord les nombre impairs puis les nombres pairs des éléments de la séquence des 11 premiers nombres de Fibonacci:

```
fibonacci = [0,1,1,2,3,5,8,13,21,34,55]
odd_numbers = list(filter(lambda x: x % 2, fibonacci))
print(odd_numbers)
[1, 1, 3, 5, 13, 21, 55]
even_numbers = list(filter(lambda x: x % 2 == 0, fibonacci))
print(even_numbers)
[0, 2, 8, 34]

# or alternatively:
even_numbers = list(filter(lambda x: x % 2 -1, fibonacci))
print(even_numbers)
[0, 2, 8, 34]
```

Exercices

Exercice 1 : Enoncé

Imaginez une routine de comptabilité utilisée dans une librairie. Elle fonctionne sur une liste de sous-listes, qui ressemblent à ceci:

N° Commande	Titre, Auteur	Quantité	Prix par article DT
34587	Learning Python, Mark Lutz	4	40.95
98762	Programmation Python, Mark Lutz	5	56.80
77226	Head First Python, Paul Barry	3	32.95

- Ecrire un programme Python, en utilisant lambda et la fonction map(), qui retourne une liste de tuples. Chaque tuple se compose d'un numéro de commande ,le prix par article multiplié par la quantité. Le prix doit être augmenté de 10 DT si la valeur de la commande est inférieur à 100,00 DT.

Exercice 1 : Solution

N° Commande	Titre, Auteur	Quantité	Prix par article DT
34587	Learning Python, Mark Lutz	4	40.95
98762	Programmation Python, Mark Lutz	5	56.80
77226	Head First Python, Paul Barry	3	32.95

```
commandes= [ ["34587","Learning Python, Mark Lutz", 4, 40.95],
             ["98762","Programming Python, Mark Lutz", 5, 56.80],
             ["77226","Head First Python, Paul Barry",3,32.95]]

qte_min = 100
tot_factures = list(
    map(lambda x: x
              if x[1] >= qte_min
              else (x[0], x[1] + 10)
        , map(lambda x: (x[0],x[2] * x[3]), commandes)
        )
    )

print(tot_factures)
```

Fin Partie 3, Merci

Autres cours :

http://cahier-de-prepa.fr/info-ipein

Créer par : Anis SAIED
said_anis@hotmail.com