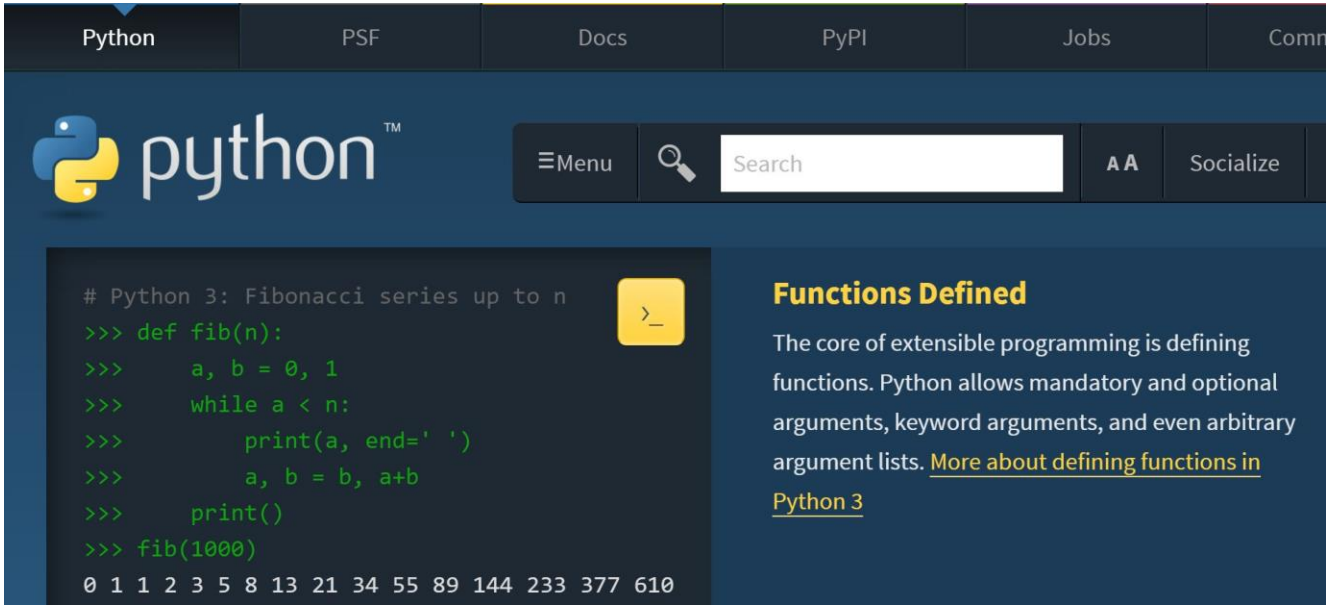
# Introduction à la programmation Python

Hervé Hocquard, hocquard@labri.fr
IREM d'Aquitaine – Groupe Algorithmique

# Le langage Python

Le langage Python est né dans les années 1990. C'est un langage libre et gratuit, facile d'accès (il possède une syntaxe très simple), puissant, et est utilisé comme langage d'apprentissage par de nombreuses universités. On trouve de nombreuses ressources libres (en français) en fouillant le web...

Site officiel du langage Python : http://www.python.org/



# Le langage Python

Le langage Python peut être utilisé :

- En mode interprété (chaque ligne du code source est analysée et traduite au fur et à mesure en instructions directement exécutées)

**Mode interprété**

- En mode mixte (le code source est compilé et traduit en *bytecode* qui est interprété par la *machine virtuelle* Python), avec l'environnement de développement IDLE

**Mode mixte**

# Le langage Python

Le script Python EntierParfait :

```
N = int(input ("donnez un entier : "))

somme = 0
for diviseur in range(1,(N//2)+1):
    if (N % diviseur == 0):
        somme = somme + diviseur

if (N == somme):
    print (N, "est parfait")
else:
    print (N, "n'est pas parfait - somme des diviseurs =", somme)
```

## Deuxième partie

# Variables, types, instructions de base

# Types et variables

Il n'y a pas de déclaration de variables en Python.

La déclaration s'effectue lors de la première affectation. Le type de la variable est alors déterminé par le type de la valeur (de l'expression) qui lui est affectée.

```
i = 4                  # i est de type entier (int)
r = 6.25               # r est de type flottant (float)
ch = 'bonjour'         # ch est de type chaîne (string)
ch2 = "hello"          # ch2 est de type chaîne (string)
encore = True          # encore est de type booléen (bool)
j = int(r)             # j est de type entier (int), valeur 6
```

# Types de base

**bool** - int - float - string

Les expressions de type `bool` peuvent prendre les valeurs `True` ou `False`.

Opérateurs logiques : `and`, `or`, `not`.

Opérateurs de comparaison : `==`, `!=`, `<`, `>`, `<=`, `>=`.

```
fini = False
encore = not fini
petits = ( i < 15 ) and ( j < 8 )
bool1 = (( i >= 8 ) or ( i <= 2 )) and ( j != 4 )
```

# Types de base

bool - **int** - float - string

Les valeurs d'une expression de type `int` ne sont limitées que par la taille mémoire...

Opérateurs arithmétiques : `+`, `-`, `*`, `//` (division entière), `%` (modulo), `**` (exponentiation), `abs` (valeur absolue)

```
i = 3 ** ( 5 // 2 )      # i vaut 9
j = abs ( 2*i - 32 )     # j vaut 14
k = j % 5                # k vaut 4
```

# Types de base

bool - int - **float** - string

Les valeurs d'une expression de type `float` ont une précision finie (modifiable).

Opérateurs arithmétiques : `+, -, *, /`

Fonctions mathématiques : module `math`

```
import math
print(math.cos(math.pi/3))      # 0.5000000000000001
print(math.factorial(6))        # 720
print(math.log(32,2))           # 5.0
```

# Types de base

bool - int - float - **string**

Un littéral de type `string` est délimité par des quotes, simples ou doubles.

Les caractères sont en position 0, 1, etc.

```
ch1 = "aujourd'hui"
print(ch1[0])                    # a

ch2 = 'ceci est une chaine'
ch2[17] = 's'
print(ch2)                       # ceci est une chaise
```

# Types de base

bool - int - float - **string**

Différentes primitives sont définies sur les variables de type `string` :

```
chaine1 = 'bonjour'
print(len(chaine1))              # 7 (len = longueur)

chaine2 = chaine1 + ' monde'     # + = concaténation
print(chaine2)                   # bonjour monde

chaine2 = chaine1.upper()        # passage en majuscules
print(chaine2)                   # BONJOUR
```

# Types de base

bool - int - float - **string**

```
chaine2 = chaine2.lower()       # passage en minuscules
print(chaine2)                  # bonjour

print(chaine2[0])               # b    (1er indice = 0)
print(chaine2[3])               # j

print(chaine2[1:4])             # onj  (indice 1 à 4
                                #      non compris)
print(chaine2[:3])              # bo   (du début à l'indice
                                #      3 non compris)

print(chaine2[4:])              # our  (de l'indice 4 à la
                                #      fin)
```

# Instructions de base

Affectation : symbole =

Saisie : <var> = <type> (input (<message>))

Affichage : print ( <liste-expressions> )

```
i = 14

nbTours = int(input('Nombre de tours : '))
nomEleve = string(input("Nom de l'élève : "))

print('resultat :',res)
print('le produit de',a,'par',b,'vaut',a*b)
```

# Instructions de base

## Options d'affichage

`sep = '-'` : séparateur entre éléments de la liste (espace par défaut)

`end = '***\n'` : en fin d'affichage (fin de ligne par défaut)

```
i = 3
j = 17
print(i,j)                  # 3 17
print(i,j,sep='')           # 317
print(i,j,sep=' --- ')      # 3 --- 17
print(i,j,i+j,sep='-',end='***')
                            # 3-17-20***
```

# Instructions de base

## Quelques caractères spéciaux

\' : apostrophe

\" : guillement

\n : fin de ligne

\t : tabulation

\a : sonnerie (bip)

```
i = 3
j = 17
print(i,j,sep='\t')                 # 3    17
print('aujourd\'hui')               # aujourd'hui
print(i,j, sep=' **\n')             # 3 **
                                    # 17
```

# Instructions de base

Le module random offre un certain nombre de fonctions outils permettant de générer des nombres de façon aléatoire.

```
import random     # on intègre le module random
...
random.seed() # initialise le générateur
i = random.randrange(12)
                # nombre entier aléatoire entre 0 et 11
i = random.randrange(4,12)
                # nombre entier aléatoire entre 4 et 11
i = random.randrange(4,12,3)
                # nombre entier aléatoire parmi 4, 7, 10
r = random.random()
                # nombre flottant aléatoire entre 0.0 et
                # 1.0 non compris (16 décimales)
```

## Troisième partie

# Structures de contrôle

# Blocs d'instructions

Un bloc d'instructions est une séquence d'instructions (une par ligne) ayant même *indentation* (décalage en début de ligne).

(pas de délimiteurs de type `début ... fin`)

L'indentation des blocs imbriqués progresse de 4 en 4 (touche `tabulation` sous IDLE).

```
i = 3                  # bloc A
...
    j = 17             # début du bloc B
    print(i,j)
...                    # fin du bloc B, suite du bloc A
print(i,j,i+j)
```

# La structure if-else

La structure `if-else` est la traduction du si-alors-sinon algorithmique (la partie `else` est facultative).

Format *général* :

```
if <condition> :
      <bloc>
else :
      <bloc>
```

```
if i>j :
    print('maximum :', i)
else :
    print('maximum :', j)
```

# La structure if-elif-else

La structure `if-elif-else` est la traduction du selonque algorithmique (la partie `else` est facultative).

```
note = float(input('Note du bac :'))

if note >= 16 :
    print ('mention TB')
elif note >= 14 :
    print ('mention B')
elif note >= 12 :
    print ('mention AB')
elif note >= 10 :
    print ('mention Passable')
else :
    print ('désolé...')
```

# La boucle while

La boucle `while` est la traduction du tant-que algorithmique.

Format *général* :

```
while <condition> :
    <bloc>
```

```
n = int(input('nombre :'))
nn = n
fact = 1
while nn > 1 :
    fact = fact * nn
    nn = nn - 1
print('La factorielle de',n,'vaut',fact)
```

# La boucle for

La boucle `for` est la traduction du pour algorithmique.

## Format général :

```
for <var> in range(<deb>,<fin+1>{,<pas>}) :
        <bloc>
```

```
n = int(input('nombre :'))
    # affichage de la table de multiplication de n
for i in range(0,11) :
    print(n,'*',i,'vaut',n*i)

    # idem... mais à l'envers
for i in range(10,-1,-1) :
    print(n,'*',i,'vaut',n*i)
```

## Quatrième partie

# Dessiner en Python

# Le module turtle

Le module `turtle` permet de dessiner dans une fenêtre graphique à l'aide d'un ensemble de primitives dédiées.

```
# script exemple turtle : dessin de différentes figures

    # import module de dessin
import turtle

    # reinitialise la fenêtre
turtle.reset()

    # titre de la fenêtre
turtle.title("Dessin de différentes figures")

    # paramètres de dessin
turtle.color('red')   # couleur de trait
turtle.width(10)      # épaisseur du trait
turtle.speed(3)       # vitesse d'exécution du tracé
```

# Le module turtle

```
    # dessin d'un carré
monCote = 200
turtle.pendown()      # on abaisse le crayon
for i in range(4):
    turtle.forward(monCote)    # la tortue avance
    turtle.left(90)            # la tortue tourne à
                               # gauche


    # partons ailleurs dessiner un cercle vert
turtle.penup()              # on lève le crayon avant
                            #     de se déplacer
turtle.goto(-60,0)          # on se déplace
turtle.pendown()            # on abaisse le crayon
turtle.color('green')       # on change de couleur
turtle.circle(100)          # on trace un cercle
```

# Le module turtle

```
    # puis un arc de cercle jaune
turtle.penup()
turtle.goto(-260,0)
turtle.pendown()
turtle.color('yellow')
turtle.circle(100,90)          # arc de cercle 90°


    # et enfin un gros point bleu
turtle.penup()
turtle.goto(-240,100)
turtle.pendown()
turtle.dot(40,'blue')          # diamètre du point 40
```

**Démonstration**

## Cinquième partie

# Pour aller (un tout petit peu) plus loin...

# Le type complex

Python offre le type numérique `complex` (notation cartésienne avec deux flottants, partie imaginaire suffixée par j), qui s'utilise ainsi :

```
print(1j)                          # 1j
print(3.4 + 1.2j)                  # (3.4+1.2j)
print((3.4 + 1.2j).real)           # 3.4
print((3.4 + 1.2j).imag)           # 1.2
print(abs(3 + 9j))       # 9.486832980505138 (module)
```

Le module `cmath` donne accès à d'autres primitives. Pour plus de détails, voir :

http://docs.python.org/py3k/library/cmath.html#module-cmath

# Utilisation de modules Python

Certaines fonctions (notamment mathématiques) ne sont utilisables qu'à la condition d'importer le module Python correspondant.

Pour nous (aujourd'hui), cela concernera essentiellement les modules math et random.

On devra donc écrire, avant toute utilisation de telles fonctions :

```
import math
```
ou
```
import random
```

**Recommandation :** On évitera d'écrire

```
from math import *
```
ou
```
from math import sqrt
```

*En effet, cela permet d'utiliser la fonction « racine carrée » (par exemple), en écrivant simplement sqrt et non math.sqrt, ce qui est contraire aux bonnes pratiques de programmation (risque de conflit d'identificateurs, perte de réutilisabilité, etc.)*

# Utilisation de fonctions (1)

**Rappel (lettre d'information – septembre 2017) :**

*Dans un objectif de simplicité et de cohérence, il est proposé dès à présent de faire évoluer l'écriture des algorithmes dans les sujets de baccalauréat, conformément aux principes suivants :*

- *suppression de la déclaration des variables, les hypothèses faites sur les variables étant précisées par ailleurs ;*
- *suppression des entrées-sorties ;*
- *simplification de la syntaxe, avec le symbole ← pour l'affectation.*

Nous sommes donc fortement incités à mettre sous forme de fonction les algorithmes que nous allons programmer...

Ce qui, d'un point de vue informatique, est une excellente chose (réutilisabilité du code produit, décomposition d'un problème...).

# Utilisation de fonctions (3)

**Format de définition d'une fonction Python :**

```
def maximum (a, b):
    # cette fonction renvoie le plus grand des
    # deux entiers a et b
    if a > b:
        return a
    else:
        return b
```

*nom de la fonction et liste des paramètres*

*Commentaire explicitant le rôle de la fonction et le type attendu des paramètres*

*Corps de la fonction (avec décalage)*

**Utilisation :**

```
>>> maximum(11,5)
11
>>> maximum(14,maximum(3,22))
22
```

*Remarque : la fonction max est déjà prédéfinie en Python ;-)...*

# Utilisation de fonctions (4)

**Format de définition d'une fonction Python :**

```
def maximum (a, b):
    # cette fonction renvoie le plus grand des
    # deux entiers a et b
    if a > b:
        return a
    else:
        return b
```

**Quelques remarques...**

- Ce n'est pas à la fonction de vérifier que les arguments fournis respectent les types attendus.
- L'instruction 'return' arrête l'exécution de la fonction.
- Si plusieurs 'return' sont présents, ils renvoient des résultats de même type.

# Utilisation de fonctions (5)

**Une fonction Python peut renvoyer plusieurs résultats :**

```
def coordonnéesMilieu (X1, Y1, X2, Y2):
    # cette fonction renvoie les coordonnées du milieu
    # du segment ayant pour extrémités les points de
    # coordonnées (X1,Y1) et (X2,Y2)

    return (X1 + X2) / 2, (Y1 + Y2) / 2
```

```
>>> x, y = coordonnéesMilieu(2,2,8,10)
>>> x
5.0
>>> y
6.0
```

***Remarque.*** *Il est également possible d'avoir des fonctions sans paramètres, et/ou qui ne renvoient pas de valeur (par exemple, une fonction qui réalise un certain graphique, toujours le même...).*

# Utilisation de fonctions (6)

**À retenir :**

- Une fonction se doit d'être universelle (utilisable dans n'importe quel contexte) ; en particulier, cela implique qu'elle ne lit pas de données au clavier, ni n'affiche de résultats à l'écran...
  C'est le programme utilisant cette fonction qui décide de la provenance des données et de la destination des résultats.

- Le commentaire associé à une fonction doit toujours préciser le rôle de la fonction, le type des arguments et, le cas échéant, les modules nécessaires à son utilisation (le programme utilisant cette fonction devra donc les importer).

- Une fonction n'a pas à vérifier le type des arguments, ni les conditions requises (c'est le programme utilisant cette fonction qui doit s'assurer de leur correction).

# Les listes

Une liste est une collection ordonnée d'éléments, éventuellement de nature distincte. Les éléments sont repérés par leur numéro d'ordre au sein de la liste (ces numéros démarrent à 0).

```
joursSemaine = ['lun', 'mar', 'mer', 'jeu', 'ven']
pairs = [0, 2, 4, 6, 8]
reponses = ['o', 'O', 'n', 'N']
listeBizarre = [jours, 2, 'hello', 54.8, 2+7j]
```

```
liste = [ 2, 3, 4 ]
print(liste)              # [2, 3, 4]
print(liste[1])           # 3
liste[2] = 28
print(liste)              # [2, 3, 28]
```

# Les listes

```
liste = list(range(4))   # convertit en liste
print(liste)             # [0, 1, 2, 3]

print(1 in liste)        # True

print(5 in liste)        # False

liste = [6, 8, 1, 4]
liste.sort()             # tri de la liste
print(liste)             # [1, 4, 6, 8]

liste.append(14)         # ajout en fin de liste
print(liste)             # [1, 4, 6, 8, 14]

liste.reverse()          # retourne la liste
print(liste)             # [14, 8, 6, 4, 1]

liste.remove(8)          # supprime la première
                         # occurrence de 8
print(liste)             # [14, 6, 4, 1]
```

# Les listes

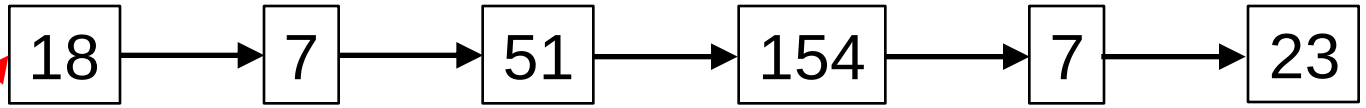
```
liste = [14, 6, 4, 1]
i = liste.pop()             # récupère et supprime le
                            # dernier élément
print(i)                    # 1
print(liste)                # [14, 6, 4]

liste.extend([6,5])
print(liste)                # [14, 6, 4, 6, 5]

print(liste.count(6))       # 2 (nombre d'éléments)

print(liste[1:3])           # [6,4] (pos. 1 à 3 non compris)

liste[0:3] = [5,4,3]        # remplace une portion de liste
print(liste)                # [5, 4, 3, 4, 6, 5]

liste[4:] = []              # positions 4 à fin de liste
print(liste)                # [5, 4, 3, 4]

liste[:2] = [1,1,1,1]       # positions 1 à 2 non compris
print(liste)                # [1, 1, 1, 1, 3, 4]
```

## Sixième partie

# Utilisation des listes

# Qu'est-ce qu'une liste ?...

Une liste est une *structure de données* permettant de stocker de l'information (des données), cette information étant organisée (structurée) de la façon suivante :




*La liste L*

*Le 4e élément... d'indice **3** !*

```
>>> L = [18, 7, 51, 154, 7, 23]
>>> type(L)
<class 'list'>
>>>
>>> L[3]
154
>>> type(L[3])
<class 'int'>
```

- *Le 1er élément a pour indice 0*
- *Le nombre d'éléments n'est pas limité*
- *Les éléments ne sont pas nécessairement de même nature (listes hétérogènes)*
- *On peut avoir des « listes de listes »...*

# Initialiser une liste (1)

On peut donner une valeur (i.e. une suite de valeurs !...) à une liste *en extension*, à l'aide d'une affectation :

```
>>> L1 = [2, 5, 7]
>>> L1
[2, 5, 7]
>>> L2 = [ ]
>>> L2
[]
```

*liste vide*

Mais on peut aussi donner une valeur à une liste *en compréhension* :

```
>>> L1 = [ x*x for x in range(1,6) ]
>>> L1
[1, 4, 9, 16, 25]
>>>
>>> L2 = [ 2*x + 3 for x in L1 if x % 2 == 1 ]
>>> L2
[5, 21, 53]
```

**Format général (la partie « if ... » est facultative) :**

maListe **= [** *expression avec x* **for** x **in** liste **if** *condition sur x* **]**

# Initialiser une liste (2)

Ce mécanisme de définition d'une liste en compréhension est similaire à la définition mathématique « usuelle » :

L1 = [ x*x for x in range (1,6) ] $\Leftrightarrow$ L1 = { $x^2$ | x $\in$ [1,6[ }

*Mais attention, une liste Python n'est pas un ensemble : répétitions, éléments ordonnés...*

Il est particulièrement « puissant » :

- Produit cartésien de deux listes

```
>>> L1, L2 = [1, 3, 5], [2, 4, 6]
>>> [[x, y] for x in L1 for y in L2]
[[1, 2], [1, 4], [1, 6], [3, 2], [3, 4], [3, 6], [5, 2], [5, 4],
[5, 6]]
```

- Triplets pythagoriciens

```
>>> [[x, y, z] for x in range(1, 20) for y in range(x, 20) for z
in range(y, 20) if x * x + y * y == z * z]
[[3, 4, 5], [5, 12, 13], [6, 8, 10], [8, 15, 17], [9, 12, 15]]
```

# Initialiser une liste (3)

Ce mécanisme de définition d'une liste en compréhension n'est en fait qu'un « raccourci » de la forme d'écriture plus classique à l'aide de la structure répétitive « for » :

- Liste de carrés

```
>>> L1 = [ x*x for x in range(1,6) ]
```

```
>>> L1 = [ ]
>>> for x in range(1,6):
        L1.append(x*x)
```

# Initialiser une liste (4)

Ce mécanisme de définition d'une liste en compréhension n'est en fait qu'un « raccourci » de la forme d'écriture plus classique à l'aide de la structure répétitive « for » :

- Produit cartésien de deux listes

```
>>> L1, L2 = [1, 3, 5], [2, 4, 6]
>>> L3 = [ [x,y] for x in L1 for y in L2 ]
```

```
>>> L1, L2 = [1, 3, 5], [2, 4, 6]
>>> L3 = [ ]
>>> for x in L1:
        for y in L2:
            L3.append([x,y])
```

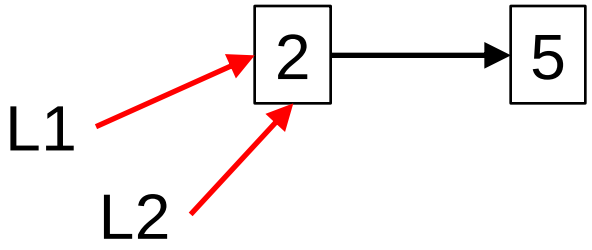
# Opérations sur les listes (voir mémento)

Python offre de nombreuses opérations (appelées *primitives*) utilisables sur les listes, permettant notamment :

- de « concaténer » des listes (opérateur '**+**')
- de récupérer le nombre d'éléments d'une liste (**len**) ; les indices des éléments de la liste L vont donc de **0** à **len(L) – 1**...
- de rajouter des éléments dans une liste (**insert**, **append**, **extend**)
- de supprimer des éléments (**remove**, **pop**, **del**)
- d'extraire des « sous-listes » (notation : **L[2:7]**)
- de tester l'appartenance d'un élément à une liste (opérateur **in**)
- de parcourir une liste (**for elem in L**)
- etc.

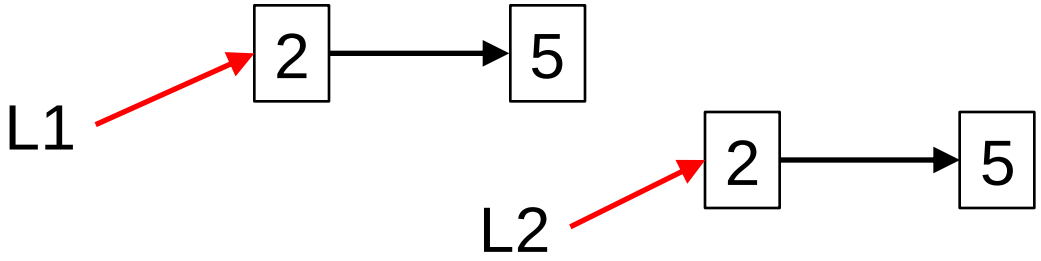
# ATTENTION : identification vs. recopie...

L'affectation de listes (opérateur '=') est une *identification* (L1 et L2 sont deux *références* vers le même objet liste) :




```
>>> L1 = [2, 5]
>>> L2 = L1
>>> L2.append(6)
>>> L2
[2, 5, 6]
>>> L1
[2, 5, 6]
```

Pour *recopier* la liste L1 dans la liste L2 (de façon à avoir deux objets listes distincts et *indépendants*), on écrira :




```
>>> L1 = [2, 5]
>>> L2 = list(L1)
>>> L2.append(6)
>>> L2
[2, 5, 6]
>>> L1
[2, 5]
```

# Un petit exemple de fonction pour finir ?...

Que fait selon vous la fonction suivante ?

```
def mystere (liste):
    # fonction mystérieuse...
    L = liste
    for i in range(0, len(L)):
        L[i] = L[i] + 1
    return L
```

**Attention à l'identification !...**

```
L = list(liste)
```

Si nous écrivons :

```
>>> T = [ 5, 10, 4, 8 ]
>>> T1 = mystere (T)
>>> T1
>>> T
```

que « répond » Python ?

```
>>> T = [5,10,4,8]
>>> T1 = mystere(T)
>>> T1
[6, 11, 5, 9]
>>> T
[6, 11, 5, 9]
```

# Merci de votre attention...

Virgil Solis - Apollon tuant Python - *Wikimedia Commons*