

## 2. Le Langage de Programmation Python

### Introduction

*« Everything that is good in Python was  
stolen from other languages »*

Guido van Rossum, 2000

# Introduction

Python est, à la fois, un langage procédural (procedural-oriented) et orienté objet (object-oriented)

## ✦ Langage Procédural

Utilisation de **fonctions** directement ou dans d'autres modules du programme

## ✦ Langage Orienté Objet

Utilisation de **classes** : une classe est un prototype de création d'**objets**

## 2.1. Introduction à la Programmation Fonctionnelle

# Introduction

- Programmes construits **uniquement** à l'aide de fonctions (*au sens mathématique*)  
    ➡ programmation fonctionnelle pure
- Programmes composés **principalement** de fonctions (*au sens informatique*)
- Programmes qui utilisent des fonctions (*au sens informatique*) comme des **objets du premier ordre**


# Introduction

- Décomposition du problème en un ensemble de fonctions
- Une fonction produit une sortie à partir d'entrées (idéal)  
    ➡ pas d'état interne qui modifie la sortie pour une entrée donnée
- **Pas d'effet de bord** (purement fonctionnelle)
  - Pas de modification de l'état interne
  - Pas de modifications non visibles dans la valeur de sortie

# Introduction

## Conséquences de l'absence de l'effet de bord

- Aucune structure de données mise à jour
- Pas d'affichage de message sur l'écran
- Pas d'écriture de fichier sur le disque
- Chaque sortie ne dépend que de son entrée



Pas toujours possible !

## Langages spécialisés

- LISP, Scheme (hybride)
- ML, CAML, ...(hybrides)
- Haskell (pur)

# Introduction

## Pourquoi vouloir éviter les objets et les effets de bord ?

- **Modularité** : décomposition du programme en petites fonctions (une fonction = une tâche)
  - Pas de fonctions complexes
  - Spécification et écriture simplifiées
  - Lecture et vérification aisées
- **Composabilité** : construction de nouveaux programmes à partir de fonctions existantes
  - Constitution d'une bibliothèque d'utilitaires

# Introduction

- **Preuve formelle** : construction aisée d'une preuve mathématique qu'un programme fonctionnel est correct
  - Preuve que le programme produit le bon résultat pour toutes les entrées possibles
- **Débogage et test** :
  - **Les petites fonctions sont plus faciles à déboguer** : isolement de la fonction à l'origine du problème
  - **Les petites fonctions sont plus faciles à tester** : une fonction est un sujet potentiel pour un test unitaire

## 2.2 La Programmation Fonctionnelle en Python

# Introduction

## Programmes Python

- exhibent une interface fonctionnelle **en apparence**
- utilisent des **fonctions impures** en interne

## Exemples :

- » Assigner dans des variables locales, mais **pas de modification de variables globales**
- » **Pas d'autre effet de bord**

# Fonctions : Objets de Première Classe

**En Python les fonctions sont des objets de première classe.**

Un **objet de première classe** est une entité du programme qui peut être :

- Créée à l'exécution (runtime)
- Affectée à une variable ou à un élément d'une structure de données
- Passée comme un argument à une fonction
- Renvoyée comme un résultat d'une fonction

# Fonctions : Objets de Première Classe

Exemple :

```
>>> def factorelle( n ) :  
    ''' renvoie n! '''  
    return 1 if n < 2 else n * factorelle( n - 1 )  
  
>>> factorelle( 5 )  
120  
  
>>> factorelle.__doc__  
'renvoie n! '  
  
>>> type( factorelle )  
< class 'function' >
```

# Fonctions : Objets de Première Classe

```
>>> fact = factorelle  
>>> fact  
< function factorelle at 0x... >  
  
>>> map( factorelle, range( 11 ) )  
< map object at 0x... >  
  
>>> list( map( factorelle, range( 11 ) ) )  
[ 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800 ]
```

*Exemple montrant la nature « première classe » d'un objet fonction*

# Les expressions Lambda

Une **expression lambda**, ou *fonction anonyme* (dans certains langages) est **une fonction courte (une ligne !)** qui prend :

- un ou plusieurs arguments,
- une expression qui combine ces arguments

**et** crée une fonction anonyme qui renvoie la valeur de l'expression.

# Les expressions Lambda

## Exemple :

<pre>&gt;&gt;&gt; somme = <b>lambda</b> x, y : x + y &gt;&gt;&gt; somme &lt;function &lt;lambda&gt; at 0x7fd ... &gt;  &gt;&gt;&gt; <b>print</b> ( somme ( 2, 3 ) ) 5</pre>	<b>=</b>	<pre>&gt;&gt;&gt; <b>def</b> somme (x, y):     <b>return</b> x +y  &gt;&gt;&gt; somme ( 2, 3 ) 5</pre>
---	----------	--

La méthode à préférer est une question de style.



# Les expressions Lambda

En général, on évite d'utiliser les expressions lambda car ceux sont des **fonctions à une seule ligne**.

➡ Impossible d'utiliser des structures de contrôle imbriquées

➡ Impossible d'y inclure des structures de type **try ... except**

*La lisibilité d'une expression lambda peut vite être remise en cause*

## Fonctions d'ordre supérieur

Exemple :

Instruction  
difficile à  
comprendre

```
>>> from functools import reduce
>>> items = [ (2, 3), (4, 5) ]
>>> total = reduce ( lambda a, b: (0, a[1] + b[1] ), items ) [ 1 ]
```

# Les expressions Lambda

## Cas pratiques d'utilisation

- Trier une liste

**Exemple :**

```
>>> maListe = [ (1, 2), (4, 1), (9, 10), (13, -3) ]
>>> maListe.sort ( key = lambda x: x[ 1 ] )
>>> print ( maListe )
[ (13, -3), (4, 1), (1, 2), (9, 10) ]
```

Fonction de tri  
selon l'élément  
à l'indice 1

# Les expressions Lambda

- Trier des listes en parallèle

**Exemple(1) :**

```
>>> maListe1 = [ (1, 2), (0, 1) ]
>>> maListe2 = [ (9, -2), (3, -3) ]
>>> mesData = zip( maListe1, maListe2 )
>>> mesData
< zip object at 0x7f9 ... >

>>> list( mesData )
[ ( (1, 2), (9, -2) ), ( (0, 1), (3, -3) ) ]
```

# Les expressions Lambda

## Exemple(2) :

A noter !

```
>>> mesData = sorted( mesData )
>>> mesData
[ ( (0, 1), (3, -3) ), ( (1, 2), (9, -2) ) ]
>>> maListe1, maListe2 = map( lambda t: list (t), zip(*mesData) )
>>> maListe1, maListe2
( [ (0, 1), (1, 2) ], [ (3, -3), (9, -2) ] )
>>> maListe1
[ (0, 1), (1, 2) ]
>>> maListe2
[ (3, -3), (9, -2) ]
```

## Fonctions d'ordre supérieur

Une **fonction d'ordre supérieur** est une fonction qui a l'une des caractéristiques suivantes :

- Elle prend une fonction comme argument
- Elle renvoie une fonction comme résultat

C'est le cas de fonctions telles que `map()` et `sorted()`.

# Fonctions d'ordre supérieur

**Exemple :** la fonction `sorted()`

```
>>> fruits = [ 'fraise', 'kiwi', 'pomme', 'pomelos' ]
```

```
>>> sorted( fruits, key = len )
```

```
['kiwi', 'pomme', 'fraise', 'pomelos' ]
```

Fonction passée en argument à appliquer à chaque élément

# Fonctions d'ordre supérieur

Toute fonction à un argument peut être passée avec **key**

```
>>> def inverse( mot ) :  
    return mot[ :: -1 ]
```

```
>>> inverse( 'test' )
```

```
'tset'
```

```
>>> sorted( fruits, key = inverse )
```

```
[ 'pomme', 'fraise', 'kiwi', 'pomelos' ]
```

# Fonctions d'ordre supérieur

Les langages fonctionnels fournissent des fonctions d'ordre supérieur. Les plus connues sont `map()`, `filter()`, `reduce()` et `apply()`.

- La fonction `apply()` a été supprimée de **Python 3** car elle n'est plus nécessaire

```
>>> apply( fn, args, kwargs )
```



```
>>> fn( *args, **keywords)
```

# Fonctions d'ordre supérieur

- `filter(prédicat, iter)` est une fonction qui renvoie un itérateur sur les éléments de la séquence qui vérifient une certaine condition.
  - Le **prédicat** est une fonction qui renvoie **VRAI** ou **FAUX**.  
Il ne peut prendre qu'un argument.

**Exemple :**

```
>>> def est_pair( x ) :  
    return ( x % 2 ) == 0  
>>> filter( est_pair, range( 10 ) )  
<filter object at 0x7...>  
  
>>> list( filter( est_pair, range( 10 ) ) )  
[ 0, 2, 4, 6, 8 ]
```

# Fonctions d'ordre supérieur

- `map( f, iterA, iterB, ... )` est une fonction qui renvoie un itérateur sur une séquence

`f( iterA[0], iterB[0] ), f( iterA[1], iterB[1] ), f( iterA[2], iterB[2] ), ...`

## Exemple :

```
>>> def majuscule ( s ) :  
|     return s.upper( )  
|  
>>> map( majuscule, [ 'phrase', 'fragment' ] )  
| <map object at 0x7f...>  
|  
>>> list( map( upper, [ 'phrase', 'fragment' ] ) )  
| ['PHRASE', 'FRAGMENT']
```

# Fonctions d'ordre supérieur

- `reduce( fn, iter, [initial_value] )` est une fonction qui applique une opération cumulative sur tous les éléments d'un itérable.
  - **fn** : fonction qui prend 2 arguments et renvoie une valeur
  - **iter** : séquence d'éléments sur laquelle appliquer **fn**, en commençant par les 2 premiers éléments.
  - **initial\_value** : paramètre optionnel, si spécifié sa valeur sera utilisée comme premier élément à cumuler.

# Fonctions d'ordre supérieur

## Exemple (1) :

```
>>> from functools import reduce
>>> from operator import concat
>>> reduce( concat, [ 'A', 'BB', 'C' ] )
'ABBC'

>>> reduce( concat, [] )
Traceback (most recent call last):
...
TypeError: reduce() of empty sequence with no initial value
```

# Fonctions d'ordre supérieur

## Exemple (2) :

```
>>> from functools import reduce
>>> from operator import mul
>>> reduce( mul, [ 4, 2, 3 ], 2 )
48

>>> reduce( mul, [ ], 2 )
2
```

# Fonctions d'ordre supérieur

## Exemple (3) :

```
>>> from functools import reduce
>>> from operator import add
>>> reduce( add, [ 1, 2, 3, 4 ], 2 )
```

12

```
>>> sum ( [ 1, 2, 3, 4 ], 2 )
```

12

```
>>> sum ( [ 1, 2, 3, 4 ] )
```

10

Beaucoup plus  
simple !

# Fonctions d'ordre supérieur

## Exemple (4) : expression lambda et reduce( )

```
>>> from functools import reduce
>>> items = [ (2, 3), (4, 5) ]
>>> total = reduce ( lambda a, b: (0, a[1] + b[1]), items ) [ 1 ]
```

```
>>> def combiner ( a , b ) :
```

```
    return 0, a[1] + b[1]
```

```
>>> total = reduce ( combiner, items ) [ 1 ]
```





# Fonctions d'ordre supérieur

../..

Mais l'idéal est d'utiliser une boucle **for**

```
>>> items = [ (2, 3), (4, 5) ]  
>>> total = 0  
>>> for a, b in items :  
        total += b
```

ou la fonction native `sum()` et une **expression génératrice**

```
>>> items = [ (2, 3), (4, 5) ]  
>>> total = sum ( b for a, b in items )  
>>> total  
8
```

# Fonctions d'ordre supérieur

**Remarque :**

- De meilleures alternatives aux fonctions `map()`, `filter()` et `reduce()` ont été proposées :
  - Les listes en compréhension (listcomp)
  - Les expressions génératrices (genexp)

# Le style fonctionnel en Python

**Les principales fonctionnalités** pour écrire un programme fonctionnel en Python

- Les itérateurs
- Les générateurs
- Les SDD en compréhension (listes, dictionnaires, ...)
- Les expressions génératrices (genexp)

## Les itérateurs

### Objet itérateur

Un **itérateur** est un objet qui représente un flux de données. Il renvoie les données élément par élément.

**L'interface standard** d'un itérateur possède deux méthodes :

- une méthode **next( )** qui renvoie l'élément suivant. Si le flux est vide, **next( )** doit lever une exception **StopIteration**
- une méthode **iter( )** qui renvoie *self*. Elle permet l'utilisation de l'itérateur là où un objet **itérable** est attendu (ex: boucle **for**)

# Les itérateurs

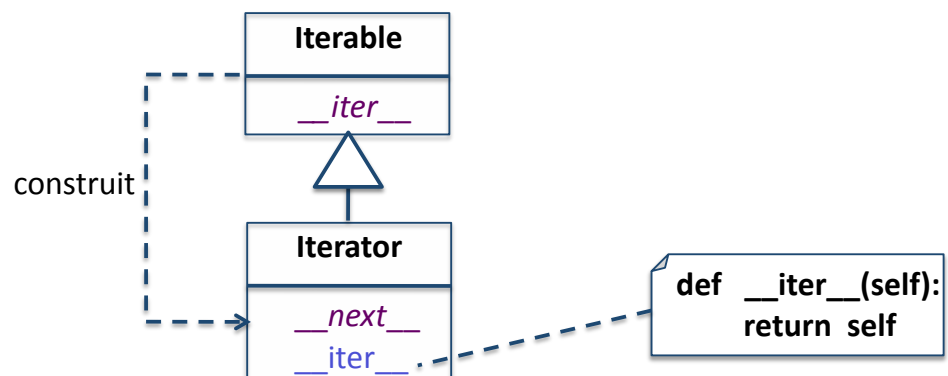
## Objet itérable

Un objet **itérable** est un objet à partir duquel la fonction **iter()** peut obtenir un **itérateur**.

```
>>> chaine = 'ABC'
>>> for car in chaine :
        print( car )
A
B
C
```

# Les itérateurs

## Itérateur et objet itérable



# Les itérateurs

**Attention :** ne pas confondre les deux !

- Les itérables ont une méthode `__iter__` qui instancie un nouvel itérateur à chaque fois
- Les itérateurs implémentent une méthode `__next__` qui renvoie les éléments individuellement **et** une méthode `__iter__` qui renvoie *self*

➡ Les itérateurs sont itérables, mais les itérables ne sont pas des itérateurs

# Les itérateurs

## Construction d'un itérateur

- Utilisation de la méthode `iter( obj )` qui construit un itérateur sur l'objet `obj`.
- Si l'objet n'est pas itérable, une exception `TypeError` est levée

**Exemple :**

```
>>> maListe = [1, 5, 9]
>>> monIterateur = iter( maListe ) # création de l'objet itérateur
>>> monIterateur                    # monIterateur sur l'objet maListe
<list_iterator object at 0x108c23100>
```

# Les itérateurs

../..

```
>>> monIterateur.__next__( )
1
>>> next( monIterateur )    # équivalent à monIterateur.__next__( )
5
>>> next( monletrateur )
9
>>> next( monIterateur )
Message d'erreur se terminant par :
StopIteration
```

**NB :** Il est impossible de récupérer l'élément précédent, de réinitialiser l'itérateur ou d'en créer une copie.

# Les itérateurs

Un itérateur peut être aussi créé à partir d'une *fonction régulière* ou tout *objet callable*.

La fonction **iter( )** prend alors 2 arguments :

- une fonction qui ne prend pas d'argument,
- une valeur d'arrêt (sentinelle) qui, si rencontrée, entraîne la levée de l'exception **StopIteration**.

# Les itérateurs

Exemple :

```
>>> def monDice( ):
    return randint( 1, 6 )

>>> iterObjet = iter( monDice, 1 )

>>> for elem in iterObjet :
    print( elem )
```

# Elle peut itérer longtemps, mais  
# elle n'affichera jamais 1 car c'est  
# la valeur sentinelle

2  
3  
2  
2

# Les itérateurs

Dans l'expression `for elem in L`, L doit être un itérateur ou un objet pour lequel `iter()` peut générer un itérateur.

```
>>> for elem in iter( maListe ):
    print( elem )
```

Ce code est équivalent à :

```
>>> for elem in maListe :
    print( elem )
```

# Les itérateurs

## Les séquences sont toujours itérables

- Appeler **iter( )** sur un dictionnaire renvoie un itérateur sur l'ensemble des clés.

```
>>> calendrier = {'Jan' : 1, 'Feb' : 2, 'Mar' : 3, 'Apr' : 4,
                  'May' : 5, 'Jun' : 6, 'Jul' : 7, 'Aug' : 8,
                  'Sep' : 9, 'Oct' : 10, 'Nov' : 11, 'Dec' : 12}

>>> for clé in calendrier :
    print(clé, calendrier[clé])
Mar 3
Feb 2
Aug 8
...
```

# Les itérateurs

- Le constructeur **dict()** accepte de prendre un itérateur en argument et renvoie un flux fini de **paires (clé, valeur)**.

```
>>> maListe = [ ( 'France', 'Paris' ), ( 'Italie', 'Rome' ),
                ( 'Espagne', 'Madrid' ) ]

>>> dict(iter( maListe ))
{'France' : 'Paris' , 'Italie' : 'Rome' , 'Espagne' : 'Madrid'}
```

# Les itérateurs

- La méthode **readline()** permet de gérer l'itération sur les fichiers.

```
>>> with open( 'monFichier.txt' ) as fp :  
    for ligne in iter( fp.readline, '\n' ) :  
        traiter( ligne )
```

**NB** : ici l'itérateur a 2 arguments, la fonction **readline()** appliquée à fp et la valeur sentinelle **'\n'** indiquant l'arrêt de la boucle si une ligne vide se terminant par **'\n'** est rencontrée.

# Les itérateurs

## Opérations sur les itérateurs

Le module **itertools** contient des itérateurs et des fonctions pour combiner différents itérateurs.

### Quelques exemples :

- Création de nouveaux itérateurs : **count(start, step)**
  - count ( )      ➡ 0, 1, 2, 3, ...
  - count (10)     ➡ 10, 11, 12, ...
  - count (10, 5) ➡ 10, 15, 20, ...



# Les itérateurs

- Sauvegarder une copie de l'itérable : `cycle(iter)`
  - `cycle( [ 1, 2, 3 ] )` → 1, 2, 3, 1, 2, 3, ...
- Répétition d'un élément : `repeat( elem, [n] )`
  - `repeat ( 'ABC' )` → ABC, ABC, ABC, ...
  - `repeat ( 'ABC, [3] )` → ABC, ABC, ABC
- Sélectionner des éléments : `filterfalse( predicat, iter )`
  - `filterfalse ( is_even, count() )` → 1, 3, 5, 7, 9, ...
- Sélectionner des éléments : `takewhile( predicat, iter )`
  - `takewhile ( is_even, count() )` → 0, 2, 4, 6, 8, ...

# Les itérateurs

## Opérations avec les itérateurs

- Les itérateurs peuvent être transformés en listes ou en tuples

```
>>> maListe = [1, 5, 9]
>>> iterObjet = iter( maListe )
>>> monTuple = tuple( iterObj )      # Appel au constructeur
>>> monTuple
(1, 5, 9)
>>> L = list( iterObj )              # Appel au constructeur
>>> L
[1, 5, 9]
```

# Les itérateurs

- Les itérateurs permettent le dépaquetage de séquences

```
>>> maListe = [1, 5, 9]
>>> iterObjet = iter( maListe )
>>> a, b, c = iterObjet          # Dépaquetage dans un tuple
>>> a, b, c
(1, 5, 9)
```

# Les itérateurs

## Remarques :

- ✓ Certaines fonctions natives telles que **max()** et **min()** prennent un itérateur en argument et renvoient le plus grand ou le plus petit élément, respectivement.
- ✓ Les opérateurs booléens **in** et **not in** gèrent également les itérateurs.

## Les générateurs

## Les générateurs constituent une classe spéciale de fonctions

- Toute fonction contenant le mot clé **yield** est un générateur
- Un générateur retourne un **objet générateur**, c-à-d un itérateur qui **produit** les valeurs des expressions passées à **yield**.
- Un générateur est une fonction qui peut être interrompue et relancée (rappelée) sans perdre sa progression
  - L'état du générateur est suspendu
  - Les variables locales sont conservées

## Les générateurs

**Exemple 1 :**

[illegible]

# Les générateurs

../..

```
>> for i in genere123 ( ) :    # un générateur est un itérateur
    print( i )
1
2
3
>> g = genere123 ( )        # genere123() renvoie un objet générateur
>> next( g )                # utilisation de next() car g est un itérateur
1
>> next( g )
2
>> next( g )
3
>> next( g )
Message d'erreur se terminant par :
StopIteration
```

# Les générateurs

Lien entre la boucle **for** et le corps de la fonction

```
>> def genere_AB ( ) :
    print( 'start' )
    yield 'A'
    print( 'continue' )
    yield 'B'
    print( 'end.' )

>> for c in genere_AB ( ) :
    print( '--> ', c )

start
--> A
continue
--> B
end
```

=

```
>> g = iter( genere_AB ( ) )
>> next( g )
```

# Les générateurs

## Exemple 2 : modification

```
>>> def compteur ( max ) :
>>>     i = 0
>>>     while i < max :
>>>         val = ( yield i )
>>>         if val is not None : # si val est fournie,
>>>             i = val         # alors changer le compteur
>>>         else :
>>>             i += 1
```

# Les générateurs

../..

```
>>> it = compteur(10)
>>> next( it )
0
| >>> next( it )
1
| >>> it.send( 8 )
8
| >>> next( it )
9
| >>> next( it )
```

Message se terminant par

StopIteration

# Les générateurs

## Exemple 3 : utilisation de **yield** / **yield from**

```
>>> def chain ( *iterables ) :  
    for it in iterables :  
        for i in it :  
            yield i  
    } = for i in iterables :  
        yield from i  
  
>>> s = 'ABC'  
>>> t = tuple ( range ( 3 ) )  
>>> list( chain( s, t ) )  
[ 'A', 'B', 'C', 0, 1, 2 ]
```

# Les générateurs

Deux traitements courants réalisables sur la sortie d'un itérateur :

- **réaliser une opération** pour chaque élément,
- **extraire un sous-ensemble des éléments** qui vérifient une certaine condition.

Les **compréhensions de listes** et les **expressions génératrices** sont des **façons concises** d'exprimer ces opérations.

# Compréhensions de listes

Une **compréhension de liste (listcomp)** construit une liste à partir d'une séquence ou tout autre type itérable en **filtrant** et **transformant** les éléments de la séquence grâce à **un itérateur**.

# Compréhensions de listes

## Schémas de manipulation des listes

- **Construction d'une liste** : construire une liste de façon algorithmique à partir d'une séquence autre qu'une liste,
- **Transformation d'une liste** : transformer une liste en appliquant à chaque élément une fonction unaire donnée,
- **Filtrage des éléments d'une liste** : filtrer les éléments d'une liste pour un prédicat donné,
- **Réduction d'une liste** : réduire une liste en une information synthétisée à partir de ses éléments.

# Compréhensions de listes

## Construction d'une liste

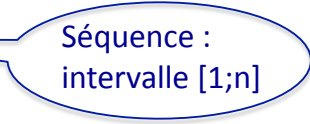
Construire une liste de façon algorithmique (élément par élément) à partir d'une séquence autre qu'une liste.

Exemple :

```
>>> def naturels_1( n ) :  
    'Retourne la liste des n premiers entiers naturels non-nuls.'  
    k = 1          # élément courant  
    maListe = []   # liste résultat  
    while k <= n :  
        maListe.append(k)  
        k=k+1  
    return maListe
```

# Compréhensions de listes

```
>>> def naturels_1( n ) :  
    ''' Retourne la liste des n premiers entiers naturels non-nuls.'''  
    k = 1          # élément courant  
    maListe = []   # liste résultat  
    while k <= n :  
        maListe.append(k)  
        k=k+1  
    return maListe  
  
>>> naturels_1(5)  
[ 1, 2, 3, 4, 5 ]
```



Séquence :  
intervalle [1;n]



# Compréhensions de listes

```
>>> def naturels_2( n ) :  
    """ Retourne la liste des n premiers entiers  
        naturels non-nuls. """  
  
    maListe = []          # liste résultat  
  
    for k in range( 1, n + 1 ) :  
        maListe.append(k)  
  
    return maListe  
  
>>> naturels_2(5)  
[ 1, 2, 3, 4, 5 ]
```

Séquence :  
intervalle [1;n+1[

Construction concise grâce à **une expression de compréhension simple**.

# Compréhensions de listes

## Expression de compréhension simple

```
| >>> [ k for k in range( 1, 6 ) ]  
[ 1, 2, 3, 4, 5 ]
```

➡ construit la liste des  $k$  pour  $k$  dans l'intervalle  $[1; 6[$

```
| >>> [ k for k in range( 3, 10 ) ]  
[ 3, 4, 5, 6, 7, 8, 9 ]
```

# Compréhensions de listes

## Syntaxe d'une expression de compréhension simple

[ <elem> for <var> in <seq> ]

➡ Construire la liste des <elem> pour <var> dans la séquence <seq>

- <var> est une variable de compréhension
- <elem> est une expression à appliquer aux valeurs successives de la variable <var>
- <seq> est une expression retournant une séquence, notamment : range, string ou list.

# Compréhensions de listes

### Remarque :

Afin que les expressions de compréhensions restent concises, le type de la variable de compréhension <var> n'est pas déclaré.

Il peut être déduit du type de la séquence <seq>.

### Exemple :

|>>> [ k for k in range( 3, 10 ) ] ➡ k est un int

# Compréhensions de listes

Version très concise de la fonction `naturels_1()`

```
>>> def naturels_3( n ) :  
    """ Retourne la liste des n premiers entiers  
        naturels non-nuls. """  
    return [ k for k in range( 1, n + 1 ) ]  
  
>>> naturels_3(5)  
[ 1, 2, 3, 4, 5 ]
```

**Solution Python très proche de la spécification du problème**

➡ **Les compréhensions ont un caractère déclaratif**

# Compréhensions de listes

Dans la syntaxe des compréhensions, **<elem>** peut être une expression complexe.

**Exemple :**

```
>>> def multiples ( j, n ) :  
    """ Retourne la liste des n premiers entiers  
        naturels non-nuls multipliés par j. """  
    return [ j * k for k in range( 1, n + 1 ) ]  
  
>>> multiples ( 2, 5 )  
[ 2, 4, 6, 8, 10 ]
```

# Compréhensions de listes

## Transformation d'une liste

Le schéma de transformation d'une liste ou schéma **map** consiste à transformer une liste en appliquant à chaque élément **une fonction unaire**.

## Construction à partir d'une liste

Exemple 1 (a) :

```
>>> def carre ( n ) :  
    """ Retourne le carré d'un entier """  
    return n * n
```

# Compréhensions de listes

Exemple 1 (b) :

```
>>> def liste_carres_1( L ) :  
    'Retourne la liste des n carrés des éléments de L.'  
    maListe = []          # liste résultat  
    for k in L :  
        maListe.append( carre (k) )  
    return maListe  
  
>>> liste_carres_1( [ 1, 2, 3, 4, 5 ] )  
[ 1, 4, 9, 16, 25 ]
```

# Compréhensions de listes

## Exemple 1 (c) : utilisation d'une expression de compréhension

```
>>> def liste_carres_2 ( L ) :  
    """ Retourne la liste des n carrés des éléments de L. """  
    return [ carre ( k ) for k in L ]  
  
>>> liste_carres_2 ( [ 1, 2, 3, 4, 5 ] )  
[ 1, 4, 9, 16, 25 ]
```

# Compréhensions de listes

## Filtrage d'une liste

Le schéma de filtrage d'une liste L consiste à construire une sous-liste de L dont les éléments vérifient un certain prédicat.

### Exemple 1 (a) :

```
>>> def est_positif( n ) :  
    """ Retourne True si n est (strictement) positif,  
    False sinon """  
    return n > 0
```

# Compréhensions de listes

## Exemple 1 (b) :

```
>>> def liste_positifs( L ) :  
    """ Retourne la sous-liste des entiers positifs de L """  
    LR = []  
    for k in L :  
        if est_positif ( k ) :  
            LR.append( k )  
    return LR  
  
>>> liste_positifs ( [ 1, -1, 2, -2, 3, -3, -4 ] )  
[ 1, 2, 3 ]
```

# Compréhensions de listes

## Exemple 1 (c) : Utilisation des compréhensions de listes

```
>>> def liste_positifs_2( L ) :  
    """ Retourne la sous-liste des entiers positifs de L """  
    [ k for k in L if est_positif ( k ) ]  
  
>>> liste_positifs_2( [ 1, -1, 2, -2, 3, -3, -4 ] )  
[ 1, 2, 3 ]
```

# Compréhensions de listes

## Exemple 2 (a) :

```
>>> def est_pair( n ) :  
    """ Retourne True si n est pair, sinon False """  
    return n % 2 == 0  
  
>>> est_pair( 1 )  
False  
  
>>> est_pair( 2 )  
True
```

# Compréhensions de listes

## Exemple 2 (b) :

```
>>> def liste_pairs( L ) :  
    """ Retourne la sous-liste des entiers pairs de L """  
    LR = []  
    for k in L :  
        if est_pair( k ) :  
            LR.append( k )  
    return LR  
  
>>> liste_pairs ( [ 1, 2, 3, 4, 5, 6, 7 ] )  
[ 2, 4, 6 ]
```