Introduction à Python

V. Monbet
(document fortement inspiré de Jean-Pierre Becirspahic)

Université de Rennes 1/IRMAR & INRIA/Aspi

5 novembre 2017

Outline

Structuration et indexation

Un bloc d'instructions PYTHON est défini par son indentation. Le début d'un bloc est marqué par un double-point (:) ; le retour à l'indentation de l'en-tête marque la fin du bloc.

```
en-tete:
    bloc ..........................
    ..............................
    d'instructions ................
```

Il est possible d'imbriquer des blocs d'instructions les uns dans les autres :

```
en-tete 1:
     ..............................
     ..............................
     en-tete 2:
         bloc ......................
         ..........................
         d'instructions ............
     ..............................
     ..............................
```

Définition d'une fonction

On définit une fonction à l'aide du mot clé def :

```
def nomdelafcn (liste de parametres):
    bloc ..........................
    d instructions ................
    a realiser ....................
```

Traditionnellement on distingue deux types de routines : les procédures ne retournent pas de résultat et se contentent d'agir sur l'environnement, les fonctions retournent un résultat. En PYTHON la distinction n'existe pas réellement : les procédures sont des fonctions qui retournent la valeur None.
Un résultat retourné par une fonction peut être réutilisé dans un calcul, à 'inverse d'une procédure :

```
In [3]: 1 + len("45")
Out[3]: 3

In [4]: 1 + print(45)
TypeError: unsupported operand type(s)
    for +: 'int' and 'NoneType'
```

Arguments d'une fonction

Une fonction peut posséder un ou plusieurs arguments séparés par une virgule. Pour définir la fonction $(x, y) \mapsto \sqrt{x^2 + y^2}$:

```
from numpy import sqrt

def norme(x, y):
    return sqrt(x**2 + y**2)
```

Une fois définie, une fonction 'utilise à l'instar de toute autre fonction prédéfinie :

```
In [1]: norme(3, 4)
Out[1]: 5.0
```

En effet, $\sqrt{3^2 + 4^2} = 5$.

Arguments d'une fonction

Une fonction peut posséder un ou plusieurs arguments séparés par une virgule. Pour définir la fonction $(x, y) \mapsto (x^k + y^k)^{1/k}$:

```
from numpy import sqrt

def norme(x, y,k):
    return(x**k + y**k)**(1/k)
```

Avec cette nouvelle définition, on a :

```
In [2]: norme(3, 4, 2)
Out[2]: 5.0
In [3]: norme(3, 4, 3)
Out[3]: 4.497941445275415
```

En effet, $(3^3 + 4^3)^{1/3} = 4.4979$.

Arguments d'une fonction

Une fonction peut posséder un ou plusieurs arguments séparés par une virgule. Pour définir la fonction $(x, y) \mapsto (x^k + y^k)^{1/k}$:

```
from numpy import sqrt

def norme(x, y,k):
    return(x**k + y**k)**(1/k)
```

Avec cette nouvelle définition, on a :

```
In [2]: norme(3, 4, 2)
Out[2]: 5.0

In [3]: norme(3, 4, 3)
Out[3]: 4.497941445275415

In [4]: norme(3, 4)
TypeError: norme() takes
    exactly 3 arguments (2 given)
```

Une erreur est déclenchée dès lors que le nombre d'arguments donnés est incorrect.

Arguments d'une fonction
Arguments optionnels

Il est possible de préciser les valeurs par défaut que doivent prendre certains arguments.

```
from numpy import sqrt

def norme(x, y,k=2):
    return(x**k + y**k)**(1/k)
```

Si on omet de préciser le troisième paramètre, ce dernier sera égal à 2 :

```
In [5]: norme(3, 4, 3)
Out[5]: 4.497941445275415

In [6]: norme(3, 4)
Out[6]: 5.0
```

Il est préférable de nommer les arguments optionnels pour éviter toute ambiguïté :

```
In [7]: norme(3, 4, k=3)
Out[7]: 4.497941445275415
```

Arguments d'une fonction
Arguments optionnels

C'est le cas de la fonction **print** qui possède deux paramètres optionnels sep (valeur par défaut : `' '`) qui est inséré entre chacun des arguments de la fonction et end (valeur par défaut : `'\n'`) qui est ajouté à la fin du dernier des arguments :

```
In [8]: print(1, 2, 3, sep='+', end='=6\n')
1+2+3=6
```

Portée des variables

Les variables définies dans une fonction ne sont accessibles que dans la fonction elle-même. De telles variables sont qualifiées de locales, par opposition aux variables globales. Si on souhaite modifier le contenu d'une variable globale à l'intérieur du bloc d'instructions d'une fonction, il faut utiliser l'instruction **global** pour déclarer celles des variables qui doivent être traitées globalement.

```
def h():
    global a # declaration
        #d une variable globale
    a = 2
    return a

In [17]: h()
Out[17]: 2
In [18]: a # la variable definie ligne 9
    #a bien ete modifiee
Out[18]: 2
```

En règle générale, il est conseillé de peu faire usage de variables globales.

Portée des variables - Exercice

```
def f():
    global a
    a = a + 1
    return a
```

```
def g():
    a = 1
    a = a + 1
    return a
```

```
def h():
    a = a + 1
    return a
```

Que produit le script suivant ?

```
a = 1
print(f(), a)
print(a, f())
print(a, g())
print(a, h())
```

Portée des variables - Exercice

```
def f():
    global a
    a = a + 1
    return a
```

```
def g():
    a = 1
    a = a + 1
    return a
```

```
def h():
    a = a + 1
    return a
```

Que produit le script suivant ?

```
a = 1
print(f(), a)
print(a, f())
print(a, g())
print(a, h())
```

Résultat

```
2 2
2 3
3 2
UnboundLocalError: local variable 'a' referenced
      before assignment
```

Portée des variables - Exercice

```
def f():
    global a
    a = a + 1
    return a
```

```
def h():
    a = a + 1
    return a
```

On utilise la fonction `dis` du module du même nom qui désassemble le bytecode :

```
dis.dis(f)
  3   0 LOAD_GLOBAL    0 (a)
      3 LOAD_CONST     1 (1)
      6 BINARY_ADD
      7 STORE_GLOBAL   0 (a)

  4  10 LOAD_GLOBAL    0 (a)
     13 RETURN_VALUE
```

```
dis.dis(h)
 14   0 LOAD_FAST      0 (a)
      3 LOAD_CONST     1 (1)
      6 BINARY_ADD
      7 STORE_FAST     0 (a)

 15  10 LOAD_FAST      0 (a)
     13 RETURN_VALUE
```

Dans la fonction h, la variable locale a est référencée (LOAD_FAST) avant d'être assignée (STORE_FAST). Il n'y a pas d'erreur dans f puisque la variable globale a est déjà assignée lorsqu'elle est référencée par LOAD_GLOBAL.

Outline

Les instructions conditionnelles

Les instructions conditionnelles se définissent à l'aide du mot clé **if** :

```
if expression booleenne:
    bloc..............
    d instructions 1..
else:
    bloc..............
    d instructions 2..
```

Si l'expression booléenne est vraie, le premier bloc d'instructions est réalisé, dans le cas contraire c'est le second.
Opérateurs courants (à valeurs booléennes) :
$x < y$ (x est strictement plus petit que y) ;
$x > y$ (x est strictement plus grand que y) ;
$x <= y$ (x est inférieur ou égal à y) ;
$x >= y$ (x est supérieur ou égal à y) ;
$x == y$ (x est égal à y) ;
$x! = y$ (x est différent de y).

Les instructions conditionnelles

Les instructions conditionnelles se définissent à l'aide du mot clé **if** :

```
if expression booleenne:
    bloc..............
    d instructions 1..
else:
    bloc..............
    d instructions 2..
```

Les chaînes de caractères peuvent être comparées suivant l'ordre lexicographique :

```
In [1]: 'alpha' < 'omega'
Out[1]: True
In [2]: 'gamma' <= 'beta'
Out[2]: False
```

Les instructions conditionnelles

Les instructions conditionnelles se définissent à l'aide du mot clé **if** :

```
if expression booleenne:
    bloc..............
    d instructions 1..
else:
    bloc..............
    d instructions 2..
```

Expressions booléennes.
L'évaluation d'une expression logique n'est qu'un calcul dont le résultat ne peut prendre que deux valeurs : le vrai (True) et le faux (False).
Les deux fonctions suivantes sont équivalentes :

```
def est_pair(n):
    return n % 2 == 0
```

```
def est_pair(n):
    if n % 2 == 0:
        return True
    else:
        return False
```

Les instructions conditionnelles

Les instructions Il est possible d'imbriquer plusieurs tests à l'aide du mot-clé **elif** :

```
if expression booleenne 1:
    bloc..............
    d instructions 1..
elif expression booleenne 2:
    bloc..............
    d instructions 2..
else:
    bloc..............
    d instructions 3..
```

- Si l'expression 1 est vraie, le bloc d'instructions 1 est réalisé ;
- Si l'expression 1 est fausse et l'expression 2 vraie, le bloc d'instructions 2 est réalisé ;
- Si les deux expressions sont fausses, le bloc d'instructions 3 est réalisé.

Plusieurs **elif** à la suite peuvent être utilisés pour multiplier les cas possibles.

Outline

Les boucles énumérées

La fonction range

La fonction **range** peut prendre entre 1 et 3 arguments entiers :
- **range**(b) énumère les entiers 0,1,2, $\cdots$,b-1 ;
- **range**(a,b) énumère les entiers a,a+1,$\cdots$, b-1 ;
- **range**(a,b) énumère les entiers a, a+c, a+2c, $\cdots$, a+nc où n est le plus grand entier vérifiant a +nc $<$ b si *c* est positif (ie n est la partie entière de (b-a)/c) et a +nc $>$ b si c est négatif (ie que n est la partie entière de (b-1)/c+1).

```
In [1]: list(range(10))
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [2]: list(range(5, 15))
Out[2]: [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
In [3]: list(range(1, 20, 3))
Out[3]: [1, 4, 7, 10, 13, 16, 19]
```

- L'énumération **range**(b) comporte b éléments ; - l'énumération **range**(a,b) comporte b-a éléments.

Les boucles énumérées

Boucles indexées

On définit une boucle indexée par la construction suivante :

```
for ... in range(...):
    bloc ..........................
    ..............................
    d instructions ................
```

Immédiatement après le mot clé **for** figure le nom d'une variable, qui va prendre les différentes valeurs de l'énumération produite par range. Pour chacune de ces valeurs, le bloc d'instructions qui suit sera exécuté.

```
In [4]: for x in range(2, 10, 3):
...:         print(x, x**2)
2 4
5 25
8 64
```

Les boucles énumérées

Boucles indexées

On définit une boucle indexée par la construction suivante :

```
for ... in range(...):
    bloc ..........................
    ................................
    d instructions ................
```

Il est possible d'imbriquer des boucles à l'intérieur d'autres boucles :

```
In [5]: for x in range(1, 6):
...:             for y in range(1, 6):
...:                 print(x * y, end=' ')
...:             print('/')
1 2 3 4 5 /
2 4 6 8 10 /
3 6 9 12 15 /
4 8 12 16 20 /
5 10 15 20 25 /
```

Les boucles énumérées

Invariant de boucle

$\rightarrow$ toute assertion vérifiant les conditions suivantes :
Initialisation cette assertion est vraie avant la première itération ;
Conservation si cette assertion est vraie avant une itération, elle le reste avant l'itération suivante ;
Terminaison une fois la boucle terminée, l'invariant fournit une propriété utile pour établir/prouver/analyser l'algorithme.

Exemple : $u_0 = 0$ et $\forall n \in \mathbf{N}$, $u_{n+1} = 2u_n + 1$

```
def u(n):
    x = 0
    for k in range(n):
        x = 2 * x + 1
    return x
```

à l'entrée de la boucle indexée par k, x référence la valeur de u_k.

Les boucles énumérées

Invariant de boucle

$\rightarrow$ toute assertion vérifiant les conditions suivantes :
Initialisation cette assertion est vraie avant la première itération ;
Conservation si cette assertion est vraie avant une itération, elle le reste avant l'itération suivante ;
Terminaison une fois la boucle terminée, l'invariant fournit une propriété utile pour établir/prouver/analyser l'algorithme.

Exemple : $u_0 = 0$ et $\forall n \in \mathbf{N}$, $u_{n+1} = 2u_n + 1$

```
def u(n):
    x = 0
    for k in range(n):
        x = 2 * x + 1
    return x
```

à l'entrée de la boucle indexée par k, x référence la valeur de u_k.
Initialisation : à l'entrée de la boucle indexée par 0, $x = 0 = u_0$.

Les boucles énumérées

Invariant de boucle

$\rightarrow$ toute assertion vérifiant les conditions suivantes :
Initialisation cette assertion est vraie avant la première itération ;
Conservation si cette assertion est vraie avant une itération, elle le reste avant l'itération suivante ;
Terminaison une fois la boucle terminée, l'invariant fournit une propriété utile pour établir/prouver/analyser l'algorithme.

Exemple : $u_0 = 0$ et $\forall n \in \mathbf{N}$, $u_{n+1} = 2u_n + 1$

```
def u(n):
    x = 0
    for k in range(n):
        x = 2 * x + 1
    return x
```

à l'entrée de la boucle indexée par k, x référence la valeur de u_k.
Conservation : si à l'entrée de la boucle indexée par k, $x = u_k$, alors à l'entrée de la boucle indexée par $k+1$, $x = 2u_k + 1 = u_{k+1}$.

Les boucles énumérées

Invariant de boulce

$\rightarrow$ toute assertion vérifiant les conditions suivantes :
Initialisation cette assertion est vraie avant la première itération ;
Conservation si cette assertion est vraie avant une itération, elle le reste avant l'itération suivante ;
Terminaison une fois la boucle terminée, l'invariant fournit une propriété utile pour établir/prouver/analyser l'algorithme.

Exemple : $u_0 = 0$ et $\forall n \in \mathbf{N}$, $u_{n+1} = 2u_n + 1$

```
def u(n):
    x = 0
    for k in range(n):
        x = 2 * x + 1
    return x
```

à l'entrée de la boucle indexée par k, x référence la valeur de u_k.
Terminaison : Le résultat retourné est la valeur de x à l'entrée de la boucle indexée par n, soit u_n.

Les boucles énumérées

Invariant de boucle

$\rightarrow$ toute assertion vérifiant les conditions suivantes :
Initialisation cette assertion est vraie avant la première itération ;
Conservation si cette assertion est vraie avant une itération, elle le reste avant l'itération suivante ;
Terminaison une fois la boucle terminée, l'invariant fournit une propriété utile pour établir/prouver/analyser l'algorithme.

Exemple : $u_0 = 0$ et $\forall n \in \mathbf{N}$, $u_{n+1} = 2u_n + 1$

```
def u(n):
    x = 0
    for k in range(n):
        x = 2 * x + 1
    return x
```

à l'entrée de la boucle indexée par k, x référence la valeur de u_k.
Ici, l'invariant de boucle énoncé prouve la validité de l'algorithme.

Les boucles énumérées

Invariant de boucle

On souhaite calculer $u_n = n!$. Pour cela, on cherche à obtenir l'invariant :
à l'entrée de la boucle indexée par k, x référence la valeur $k!$.
Cet invariant conditionne l'initialisation et la conservation :

```
def fact(n):
    x = 1 # initialisation
    for k in range(n):
        x = x * (k + 1) # conservation
    return x
```

Ici, l'invariant de boucle permet de rédiger l'algorithme.

Les boucles énumérées

Invariant de boucle

On souhaite calculer u_n définie par $u_0 = 0$, $u_1 = 1$ et $u_{k+2} = uk + 1 + uk$.
On cherche à obtenir l'invariant :
à l'entrée de la boucle indexée par k, x référence la valeur de u_k .

Problème : comment calculer $u_k + 1$ à l'aide de la seule valeur de u_k ? Il manque la valeur de u_{k-1}.

Les boucles énumérées

Invariant de boucle

On souhaite calculer u_n définie par $u_0 = 0$, $u_1 = 1$ et $u_{k+2} = uk + 1 + uk$.
On cherche à obtenir l'invariant :
à l'entrée de la boucle indexée par k, x référence la valeur de u_k et y référence la valeur de u_{k-1} .

On en déduit la fonction :

```
def fib(n):
    x, y = 0, 1 # initialisation
    for k in range(n):
        x, y = x + y, x # conservation
    return x
```

Les boucles énumérées

Exercice

On considère $p(x) = \sum_{i=0}^{n-1} a_i x^i$ représenté par $p = [a_0, a_1, a_2, ...]$.

Déterminer un invariant pour établir le rôle de la fonction :

```
def mystere(p, x):
    n = len(p)
    s = 0
    for k in range(n):
        s = x * s + p[n-1-k]
    return s
```

À l'entrée de la boucle indexée par k, ...

Les boucles énumérées

Exercice

On considère $p(x) = \sum_{i=0}^{n-1} a_i x^i$ représenté par $p = [a_0, a_1, a_2, ...]$.

Déterminer un invariant pour établir le rôle de la fonction :

```
def mystere(p, x):
    n = len(p)
    s = 0
    for k in range(n):
        s = x * s + p[n-1-k]
    return s
```

À l'entrée de la boucle indexée par k, s référence la valeur de $\sum_{i=n-k}^{n-1} a_i x^{i+k-n}$

On en déduit que cet algorithme retourne la valeur de $\sum_{i=0}^{n-1} a_i x^i = p(x)$.
Ici, l'invariant de boucle permet d'analyser l'algorithme.

Les boucles énumérées

Exercice

Que font les deux fonctions définies ci-dessous ?

```
def entiers(i, j):
    for k in range(i, j):
        print(k)
    print(j)
entiers(7, 22)
```

```
def entiers(i, j):
    if j % 7 == 0:
    j -= 1
    for k in range(i, j):
        if k % 7 != 0:
            print(k)
    print(j)
entiers(7, 22)
```

Les boucles énumérées

Exercice

Écrire une fonction PYTHON qui correspondra à la fonction $f : \mathbb{N} \to \mathbb{N}$

$$\forall n \in \mathbb{N},\quad f(n) = \begin{cases} n/2 & \text{si} \quad n \quad \text{est pair} \\ 3n+1 & \text{si} \quad n \quad \text{est impair} \end{cases}$$

Comment calculer $fofof(17)$?
Quel résultat doit-on obtenir ?

Outline

Parcours d'une chaîne de caractères

Syntaxe générale de l'instruction**for** :

```
for ... in ...:
    bloc ..........................
    ..............................
    dinstructions ................
```

Ce qui suit le mot-clé **in** doit être une structure de données énumérable (intervalles, chaîne de caractères, listes, fichiers . . .).
Par exemple :

```
def epeler(mot):
    for c in mot:
        print(c)
In [1]: epeler('Pourquoi')

P
o
u
r
q
u
```

Parcours d'une chaîne de caractères

Syntaxe générale de l'instruction**for** :

```
for ... in ...:
    bloc ..........................
    ................................
    dinstructions ................
```

Ce qui suit le mot-clé **in** doit être une structure de données énumérable (intervalles, chaîne de caractères, listes, fichiers . . .).
Dans un langage de programmation n'autorisant que des itérations suivant une progression arithmétique, il faudrait écrire :

```
def epeler(mot):
    for i in range(len(mot)):
        print(mot[i])
```

Parcours d'une chaîne de caractères

Syntaxe générale de l'instruction **for** :

```
for ... in ...:
    bloc ..........................
    ................................
    dinstructions ................
```

Ce qui suit le mot-clé **in** doit être une structure de données énumérable (intervalles, chaîne de caractères, listes, fichiers . . .).
Il est possible d'énumérer à la fois l'indice et la valeur d'un élément :

```
for (i, c) in enumerate('Pourquoi'):
    print(i, c)
(0, 'P')
(1, 'o')
(2, 'u')
(3, 'r')
(4, 'q')
(5, 'u')
(6, 'o')
(7, 'i')
```

Parcours d'une chaîne de caractères

Syntaxe générale de l'instruction**for** :

```
for ... in ...:
    bloc ..........................
    ..............................
    dinstructions ................
```

Ce qui suit le mot-clé **in** doit être une structure de données énumérable (intervalles, chaîne de caractères, listes, fichiers . . .).
Il est possible d'énumérer 2 énumérables à la fois :

```
for (i, c) in zip(range(1, 10), 'Pourquoi'):
    print(i, c)

(1, 'P')
(2, 'o')
(3, 'u')
(4, 'r')
(5, 'q')
(6, 'u')
(7, 'o')
```

Outline

Boucles conditionnelles

Une boucle conditionnelle exécute une suite d'instructions tant qu'une certaine condition est réalisée.

```
while condition:
    bloc ..........................
    ..............................
    d instructions ................
```

La condition doit être une expression à valeurs booléennes.

```
In [1]: while 1 + 1 == 3:
...:       print('abc')
...:   print('def')
def
```

L'instruction print('abc') n'est jamais exécutée puisque la condition est fausse.

Boucles conditionnelles

Une boucle conditionnelle exécute une suite d'instructions tant qu'une certaine condition est réalisée.

```
while condition:
    bloc ..........................
    ..............................
    d instructions ................
```

La condition doit être une expression à valeurs booléennes.

```
In [1]: while 1 + 1 == 2:
...:       print('abc')
...:   print('def')
abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc
abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc ...
```

L'instruction est éternellement vérifiée.... ce qui conduit au blocage de l'interprète.

Boucles conditionnelles

Une boucle conditionnelle exécute une suite d'instructions tant qu'une certaine condition est réalisée.

```
while condition:
    bloc ..........................
    ..............................
    d instructions ................
```

En général, la condition dépend d'une variable au moins dont le contenu sera susceptible d'être modifié dans le corps de la boucle.
Par exemple :

```
In [3]: x = 10
In [4]: while x > 0:
...:           print(x)
...:           x -= 1
10 9 8 7 6 5 4 3 2 1
```

Terminaison d'une boucle conditionnelle

Exercice

Donner le rôle des fonctions suivantes :

```
def mystere(n):
    p = 0
    while (p+1)*(p+1) <= n:
        p += 1
    return p
```

Terminaison d'une boucle conditionnelle

Prouver la terminaison d'une boucle conditionnelle, c'est prouver qu'elle retourne un résultat en **un temps fini**. La recherche d'un invariant de boucle adéquat fournit le plus souvent la solution.
Exemple :

```
def mystere(a, b):
    q, r = 0, a
    while r >= b:
        q, r = q + 1, r - b
    return q, r
```

En général, la condition dépend d'une variable au moins dont le contenu sera susceptible d'être modifié dans le corps de la boucle.
Par exemple :

```
In [3]: x = 10
In [4]: while x > 0:
...:           print(x)
...:           x -= 1
10 9 8 7 6 5 4 3 2 1
```

Terminaison d'une boucle conditionnelle

Prouver la terminaison d'une boucle conditionnelle, c'est prouver qu'elle retourne un résultat en **un temps fini**. La recherche d'un invariant de boucle adéquat fournit le plus souvent la solution.
Exemple :

```
def mystere(a, b):
    q, r = 0, a
    while r >= b:
        q, r = q + 1, r - b
    return q, r
```

Exemple :

```
def mystere(a, b):
    q, r = 0, a
    while r >= b:
    q, r = q + 1, r - b
    return q, r
```

La boucle conditionnelle réalise l'itération de deux suites $(q_n)_{n\in\mathbb{N}}$ et $(r_n)_{n\in\mathbb{N}}$ définies par $q_0 = 0$, $r_0 = a$, et $q_{n+1} = q_n + 1$, $r_{n+1} = r_n - b$.

Terminaison d'une boucle conditionnelle

Prouver la terminaison d'une boucle conditionnelle, c'est prouver qu'elle retourne un résultat en **un temps fini**. La recherche d'un invariant de boucle adéquat fournit le plus souvent la solution.
Exemple :

```
def mystere(a, b):
    q, r = 0, a
    while r >= b:
        q, r = q + 1, r - b
    return q, r
```

Exemple :

```
def mystere(a, b):
    q, r = 0, a
    while r >= b:
    q, r = q + 1, r - b
    return q, r
```

Conclusion : cette fonction retourne la valeur d'un couple (q, r) vérifiant

Terminaison d'une boucle conditionnelle

Exercice

Donner le rôle de la fonction suivante :

```
def mystere(n):
    i, s = 0, 0
    while s < n:
        s += 2 * i + 1
        i += 1
    return i
```

Terminaison d'une boucle conditionnelle

Exercice

Donner le rôle de la fonction suivante :

```
def mystere(n):
    i, s = 0, 0
    while s < n:
        s += 2 * i + 1
        i += 1
    return i
```

À l'entrée de la boucle de rang k, $i = k$ et $s = \sum_{i=0}^{k-1}(2i+1) = k^2$.
Il faut observer que $(p+1)^2 = p^2 + 2p + 1$.
Il existe un unique entier k tel que $(k-1)^2 < n \leq k^2$ donc l'algorithme se termine et retourne cette valeur de $k = [\sqrt{n}]$.

Forcer la sortie d'une boucle

Pour sortir prématurément d'une boucle : **return** ou **break**.
Exemple : recherche d'un caractère dans une chaîne de caractères.

```
def cherche(c, chn):
    for x in chn:
        if x == c:
        return True
    return False
```

Dès que le caractère *c* est trouvé dans la chaîne *chn*, le parcours cesse.

Forcer la sortie d'une boucle

Pour sortir prématurément d'une boucle : **return** ou **break**.
break permet d'interrompre le déroulement des instructions du bloc interne à la boucle.

```
s = 0
while True:
    s += 1
    if randint(1,7) == 6 and randint(1,7) == 6:
        break
```

La boucle ne se termine que si on réalise un double 6 (la terminaison n'est que *probable*).

Forcer la sortie d'une boucle

Pour obtenir le nombre moyen de jets nécessaire à l'obtention d'un double 6, on réalise cette expérience un nombre suffisant de fois :

```
from numpy.random import randint
def test(n):
    e = 0
    for k in range(n):
        s = 0
        while True:
            s += 1
            if randint(1, 7) == 6 and randint(1, 7) == 6:
                break
        e += s
    return e / n
```

```
test(100000)
Out[61]: 36.19483

test(100000)
Out[62]: 35.78437
```

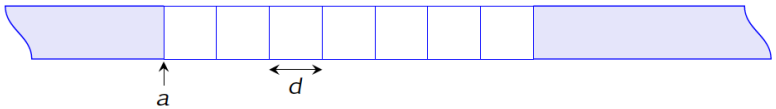
```
test(100000)
Out[63]: 36.08243

test(100000)
Out[64]: 36.05108
```

Outline

Structures de données

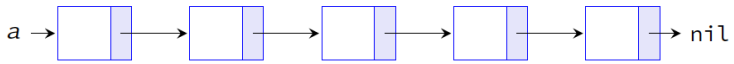
Les listes sont les principales structures de données en PYTHON. C'est une structure hybride, qui cherche à tirer avantage de deux structures de données fondamentales en informatique : les tableaux et les listes chaînées.

Les tableaux forment une suite de variables de même type associées à des emplacements consécutifs de la mémoire :



Les tableaux existent en Python : c'est la classe `array` fournie par la bibliothèque NUMPY.

Les listes associent à chaque donnée un pointeur indiquant la localisation dans la mémoire de la donnée suivante :



Les listes n'existent pas en PYTHON : la classe **`list`** n'est pas une liste chaînée mais une structure de données qui concilie les avantages des tableaux et des listes chaînées.
c'est une structure de donnée dynamique dans laquelle les éléments sont accessibles à coût constant.

Construction d'une liste

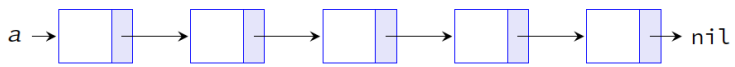
Une liste est une structure de données linéaire constituée d'objets de types hétérogènes :

```
a = [0, 1, 'abc', 4.5, 'de', 6]
b = [[], [1], [1,2], [1, 2, 3]]
```

La liste a contient 6 éléments et la liste b 4 éléments.
Une liste peut très bien contenir d'autres listes.

On accède à chaque élément de la liste par son index, qui débute à 0.

```
In [1]: a[2]
Out[1]: 'abc'

In [2]: b[3][1]
Out[2]: 2
```



Construction d'une liste

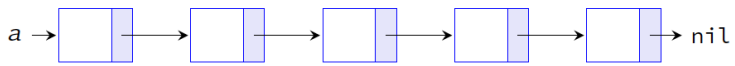
Une liste est une structure de données linéaire constituée d'objets de types hétérogènes :

```
a = [0, 1, 'abc', 4.5, 'de', 6]
b = [[], [1], [1,2], [1, 2, 3]]
```

La liste a contient 6 éléments et la liste b 4 éléments.
Une liste peut très bien contenir d'autres listes.

Lorsque l'index est négatif, le décompte est pris en partant de la fin. Ainsi, les éléments d'indices -k et ℓ-k sont les mêmes.

```
In [4]: a[-2]
Out[4]: 'de'
```



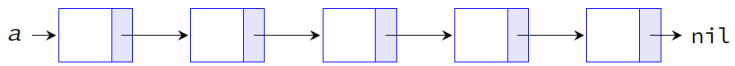
Slicing

PYTHON permet le « découpage en tranches » : si l est une liste, alors l[i :j] est une nouvelle liste constituée des éléments dont les index sont compris entre i et j - 1 :

```
In [5]: a[1:5]
Out[5]: [1, 'abc', 4.5, 'de']

In [6]: a[1:-1]
Out[6]: [1, 'abc', 4.5, 'de']
```

Lorsque i est absent, il est pris par défaut égal à 0 ; lorsque j est absent, il est pris par défaut égal à la longueur de la liste.

```
In [7]: a[:4]
Out[7]: [0, 1, 'abc', 4.5]

In [8]: a[-3:]
Out[8]: [4.5, 'de', 6]
```



À retenir l[:n] calcule la liste des n premiers éléments et l[-n :] la liste des n derniers.

Slicing

Sélection partielle

La syntaxe `lst[debut:fin]` possède un troisième paramètre (égal par défaut à 1) indiquant le pas de la sélection :

```
In [9]: l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
In [10]: l[2:9:3]
Out[10]: [2, 5, 8]
In [11]: l[3::2]
Out[11]: [3, 5, 7, 9]
In [12]: l[:-1:2]
Out[12]: [0, 2, 4, 6, 8]
```

Il est possible de choisir un pas négatif :

```
In [13]: l[-1:0:-1]
Out[13]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Création d'une liste

par énumération

Les listes sont de type **list** et la fonction **list** convertit lorsque c'est possible un objet d'un certain type vers le type **list**. C'est le cas en particulier des énumérations produites par la fonction **range** :

```
In [14]: list(range(11))
Out[14]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [15]: list(range(13, 2, -3))
Out[15]: [13, 10, 7, 4]
```

Création d'une liste

par compréhension

Il est possible de définir une liste par filtrage du contenu d'une énumération selon un principe analogue à une définition mathématique.

$\{x \in [0, 10] | x^2 \leq 50\}$

```
In [17]: [x for x in range(11) if x * x <= 50]
Out[17]: [0, 1, 2, 3, 4, 5, 6, 7]
```

Produit cartésien de deux listes :

```
In [18]: a, b = [1, 3, 5], [2, 4, 6]

In [19]: [(x, y) for x in a for y in b]
Out[19]: [(1, 2), (1, 4), (1, 6), (3, 2), (3, 4), (3, 6),
(5, 2), (5, 4),(5, 6)]
```

Opérations sur les listes

L'opération de concaténation se note + :

```
In [1]: [2, 4, 6, 8] + [1, 3, 5, 7]
Out[1]: [2, 4, 6, 8, 1, 3, 5, 7]
```

L'opérateur de duplication se note * ; si lst est une liste et n un entier alors lst * n est équivalent à lst + lst + ... + lst (n fois).

```
In [2]: [1, 2, 3] * 3
Out[2]: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Suppression d'un ou plusieurs éléments :

```
In [9]: l = list(range(11))
In [10]: del l[3:6]
In [11]: l
Out[11]: [0, 1, 2, 6, 7, 8, 9, 10]
```

Mutation d'une liste

Une liste est un objet mutable, c'est à dire modifiable.
Suppression d'un ou plusieurs éléments :

```
In [9]: l = list(range(11))
In [10]: del l[3:6]
In [11]: l
Out[11]: [0, 1, 2, 6, 7, 8, 9, 10]
```

Insertion d'un élément :

```
In [12]: l = ['a', 'b', 'c', 'd']
In [13]: l.append('e')
In [14]: l
Out[14]: ['a', 'b', 'c', 'd', 'e']
In [15]: l.insert(2, 'x')
In [16]: l
Out[16]: ['a', 'b', 'x', 'c', 'd', 'e']
```

Quelques méthodes associées aux listes

```
In [17]: l = [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
In [18]: l.remove(4) # retire les 4
In [19]: l
Out[19]: [1, 2, 3, 5, 1, 2, 3, 4, 5]
In [20]: l.pop(-1) # supprime l element d indice i et le retourne
Out[20]: 5
In [21]: l
Out[21]: [1, 2, 3, 5, 1, 2, 3, 4]
In [22]: l = [1, 2, 3, 4, 1, 2, 3, 4]
In [23]: l.reverse()
In [24]: l
Out[24]: [4, 3, 2, 1, 4, 3, 2, 1]
In [25]: l.sort()
In [26]: l
Out[26]: [1, 1, 2, 2, 3, 3, 4, 4]
```

Parcours d'une liste

Parcours complet par boucle énumérée

La syntaxe **for** x **in** seq : permet de parcourir tous les éléments x d'une séquence seq.
Exemple : calcul de la somme des éléments d'une liste (ou d'un tuple).

```
def somme(l):
    s = 0
    for x in l:
        s += x
    return s
```

Calcul de la moyenne $\bar{x} = \frac{1}{n}\sum_{k=1}^{n} x_k$

```
def moyenne(l):
    s = 0
    for x in l:
        s += x
    return s/len(l)
```

Parcours d'une liste

Exercices

1. Écrire une fonction pour calculer la variance des éléments d'une liste.
2. Écrire une fonction pour trouver le maximum des éléments d'une liste.
3. Écrire une fonction pour trouver l'indice du maximum des éléments d'une liste.

Parcours d'une liste

Calcul du maximum

Calcul de la valeur maximale d'une liste :

```
def maximum(l):
    m = l[0]
    for x in l:
        if x > m:
            m = x
    return m
```

Calcul de l'indice de l'élément maximal : on parcourt la liste des indices.

```
def indice_max(l):
    i, m = 0, l[0]
    for k in range (1, len(l)):
        if l[k] > m:
            i, m = k, l[k]
    return i
```

Parcours d'une liste

Calcul du maximum

Calcul de la valeur maximale d'une liste :

```
def maximum(l):
    m = l[0]
    for x in l:
        if x > m:
            m = x
    return m
```

La fonction **enumerate** renvoie une liste de tuples (indice, valeur) lorsqu'on l'applique à un objet énumérable.

```
def indice_du_max(l):
    i, m = 0, l[0]
    for (k, x) in enumerate(l):
        if x > m:
            i, m = k, x
    return i
```

Parcours d'une liste

Parcours incomplet (recherche d'un élément)

Un problème fréquent : déterminer la présence ou non d'un élément x dans une liste l.

Première méthode : parcourir la liste par une boucle conditionnelle.

```
def cherche(x, l):
    k, rep = 0, False
    while k < len(l) and not rep:
        rep = l[k] == x
        k += 1
    return rep
```

Deuxième méthode : interrompre une boucle énumérée dès l'élément trouvé.

```
def cherche(x, l):
    for y in l:
        if y == x:
            return True
    return False
```

Exercices

(1) Écrire une fonction qui renvoie le plus petit élément d'une liste.

(2) Écrire une fonction qui renvoie la liste triée par ordre croissant du dernier élément des tuples.
`Liste = [(2, 5), (1, 2), (4, 4), (2, 3), (2, 1)]`
Résultat : `[(2, 1), (1, 2), (2, 3), (4, 4), (2, 5)]`

```
?sorted
Signature: sorted(iterable, /, *, key=None, reverse=False)
Docstring: Return a new list containing all items
from the iterable  in ascending order.
A custom key function can be supplied to customize the sort order,
and the reverse flag can be set to request the result
in descending order.
```

(3) Écrire une fonction qui renvoie le nombre de chaines telles que la chaine est de longueur supérieure à 2 et ses premier et dernier caractères sont égaux.
`Liste = ['abc', 'xyz', 'aba', '1221']`
Résultat : 2

(4) Écrire un programme qui supprime les duplicats dans une liste.

Banque d'exercices : https ://www.w3resource.com/python-exercises/list/

Exercices

solutions

(1) Écrire une fonction qui renvoie le plus petit élément d'une liste.

```
def max_num_in_list( list ):
    max = list[ 0 ]
    for a in list:
        if a > max:
            max = a
    return max
print(max_num_in_list([1, 2, -8, 0]))
```

Autres solutions

```
min(liste)

sorted(liste)[1]
```

Exercices

solutions

(2) Écrire une fonction qui renvoie la liste triée par ordre croissant du dernier élément des tuples.

```
def last(n): return n[-1]

def sort_list_last(tuples):
  return sorted(tuples, key=last)

print(sort_list_last([(2, 5), (1, 2), (4, 4), (2, 3), (2, 1)]))
```

Exercices

solutions

(3) Écrire une fonction qui renvoie le nombre de chaines telles que la chaine est de longueur supérieure à 2 et ses premier et dernier caractères sont égaux.

```
def match_words(words):
  ctr = 0

  for word in words:
    if len(word) > 1 and word[0] == word[-1]:
      ctr += 1
  return ctr

print(match_words(['abc', 'xyz', 'aba', '1221']))
```

Exercices

solutions

(3) Écrire une fonction qui supprime les duplicats dans une liste.

```
a = [10,20,30,20,10,50,60,40,80,50,40]

dup_items = set()
uniq_items = []
for x in a:
    if x not in dup_items:
        uniq_items.append(x)
        dup_items.add(x)

print(dup_items)
```

Outline

Numpy, premiers pas

- NUMPY est une librairie pour les tableaux multidimensionnels, en particulier les matrices ;
- Son implémentation est proche du hardware, et donc beaucoup plus efficace pour les calculs ;

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6]])
A
Out[2]:
array([[1, 2, 3],
       [4, 5, 6]])
A.ndim
Out[3]: 2
A.shape
Out[4]: (2, 3)
A.dtype
Out[5]: dtype('int64')
A = np.array([[1, 2, .3], [4, 5, 6]])
A.dtype
Out[7]: dtype('float64')
```

Numpy, créer des tableaux

- En pratique, on créer rarement des tableaux à la main.
- Il existe en effet des outils pour créer des tableaux de suites incrémentées

Suites

```
import numpy as np
debut = 0.5
fin = 3.5
pas = 1
A = np.arange(debut,fin,pas)
A
Out[24]: array([ 0.5,  1.5,  2.5])
B = np.linspace(debut,fin,nb_points)
B
Out[28]: array([ 0.5 ,  1.25,  2.  ,  2.75,  3.5 ])
```

mpy, créer des tableaux

- En pratique, on créer rarement des tableaux à la main.
- Il existe en effet des outils pour créer des tableaux de constantes

Tableaux constants

```
import numpy as np

A = np.zeros(5)
A
Out[29]: array([ 0.,  0.,  0.,  0.,  0.])
B = np.ones((3,4))
B
Out[30]:
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
C = np.ones((2,3,2))
C
Out[37]:
array([[[ 1.,  1.],
        [ 1.,  1.],
        [ 1.,  1.]],

       [[ 1.,  1.],
        [ 1.,  1.],
```

Numpy, créer des tableaux

- En pratique, on créer rarement des tableaux à la main.
- Il existe en effet des outils pour créer des matrices diagonales

Matrices diagonales

```
import numpy as np
I = np.eye(3)
I
Out[39]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
D = np.diag([3,2,4,1])
D
Out[40]:
array([[3, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 4, 0],
       [0, 0, 0, 1]])
```

Numpy, slicing

- On peut accéder à sous tableaux
- Par élément

```
A = np.array([[1, 2, 3,4],
        [5, 6,7,8]])
A
Out[41]: array([[1, 2, 3, 4],
        [5, 6, 7, 8]])
A[0,1]
Out[42]: 2
```

- Par block

```
A[1,2:4]
Out[54]: array([7, 8])
A[0,:]
Out[55]: array([1, 2, 3, 4])
A[:,3]
array([4, 8])
```

- Par lignes ou colonnes

Numpy, slicing

- On peut accéder à sous tableaux
- Par lignes ou colonnes

```
A[0,:]
Out[55]: array([1, 2, 3, 4])
A[:,3]
array([4, 8])
A[:,0:2]
array([[1, 2],
       [5, 6]])
```

Numpy, slicing

- On peut modifier les valeurs d'un tableau :
- par élément ou sous tableaux

```
A
A[0,2:] = 0
A
Out[58]: array([[1, 2, 0, 0],
        [5, 6, 7, 8]])
B =B = np.arange(4,0,-1).reshape((2,2))
B
Out[68]: array([[4, 3],
        [2, 1]])
A[:2,:2] = B
A
Out[72]: array([[4, 3, 0, 0],
        [2, 1, 7, 8]])
```

- On note ci-dessus le `reshape` qui permet de modifier la taille d'un tableau.

copy or not copy

```
T1 = np.zeros((2, 3),
     dtype="uint8")
array([[0, 0, 0],
       [0, 0, 0]], dtype=uint8)
T1[0, 1] = 128
T2 = T1[:, 1:]
T2[1, 1] = 50
Out[16]:
array([[128,   0],
       [  0,
50]], dtype=uint8)
T1[0, 2] = 23
T2
Out[17]:
array([[128,  23],
       [  0,
50]], dtype=uint8)
```

```
T3 = T2.copy()
T3[1, 1] = 17
T3
Out[21]:
array([[128,  23],
       [  0,
17]], dtype=uint8)
T2
Out[22]:
array([[128,  23],
       [  0,
50]], dtype=uint8)
```

NUMPY, opérations sur les tableaux

- Les opérations sur la tableaux peuvent se faire terme à terme. En particulier, les opérations simples comme `+, -, *, /, **` sont effectuées sur chaque élément du tableau.

Opérations simples

```
B
Out[73]: array([[4, 3],
       [2, 1]])
B+1
Out[74]:
array([[5, 4],
       [3, 2]])
B**2
array([[16,  9],
       [ 4,  1]])
```

- NUMPY dispose aussi de fonctions scientifiques, par exemple : `np.exp()`, `np.cos()`, $\cdots$

NUMPY, opérations sur les tableaux

- NUMPY permet bien sûr de faire des opération matricielles comme la transposition, le calcul d'une trace

```
B
Out[73]: array([[4, 3],
        [2, 1]])
B.T
array([[4, 2],
        [3, 1]])
 B.trace()
Out[86]: 5
```

NUMPY, opérations sur les tableaux

- NUMPY permet bien sûr de faire des opérations matricielles comme le produit matriciel

```
B
Out[73]: array([[4, 3],
        [2, 1]])
np.dot(B,B)
Out[89]: array([[22, 15],
        [10,  7]])
```

Opérations sur les tableaux

- NUMPY permet bien sûr de faire des opérations sur l'ensemble du tableau comme trouver la valeur minimum ou maximum, calculer la somme ou le produit de tous les éléments du tableau, ...

```
B
Out[73]: array([[4, 3],
        [2, 1]])
np.amin(B)
Out[73]: 1
np.sum(B)
Out[74]: 10
```

Trouver des éléments du tableau

- Il est souvent utile de retrouver les éléments d'un tableau qui vérifient certaines conditions

```
a
Out[1]: array([[2, 3, 4],
[4, 5, 6]])
 i = np.nonzero(a > 3)
 i
Out[2]: (array([0, 1, 1, 1]), array([2, 0, 1, 2]))
i[0]
Out[3]: array([0, 1, 1, 1])
a[i]
Out[4]: array([4, 4, 5, 6])
a[i] = 0
a
Out[5]array([[2, 3, 0],
[0, 0, 0]])
```

Exercices

(1) Écrire un programme pour renverser un tableau.
Tableau : [12 13 14 15 16 17 18 19 20]
Tableau renversé : [20 19 18 17 16 15 14 13 12]

(2) Écrire un programme pour transformer un tableau de type **`int`** en un tableau de type **`float`**.

(3) Écrire un programme qui des 0 tout autour d'un tableau.

```
Original array          0 on the border
[[ 1. 1. 1.]            [[ 0. 0. 0. 0. 0.]
[ 1. 1. 1.]             [ 0. 1. 1. 1. 0.]
[ 1. 1. 1.]]            [ 0. 1. 3. 1. 0.]
                        [ 0. 1. 1. 1. 0.]
                        [ 0. 0. 0. 0. 0.]]
```

(4) Écrire un programme qui renvoie un tableau dont les lignes sont renversées.

```
Original array          with reversed lines
[[ 1. 2. 3. 4.]            [[ 4. 3. 2. 1.]
[ 5. 6. 7. 8.]             [ 8. 7. 6. 5.]
[ 9. 10. 11. 12.]]            [ 12. 11. 10. 9.] ]
```

(4) Écrire un programme qui les valeurs puis les indices des éléments supérieurs à 10.

```
Original array
[[ 0. 10. 20. 30.]
[ 1. 20. 30. 4.]]
```

Banque d'exercices : https ://www.w3resource.com/python-exercises/list/

Exercices

solutions

(1) Écrire un programme pour renverser un tableau.

```
import numpy as np
x = np.arange(12, 21)
print("Original array:")
print(x)
print("Reverse array:")
x = x[::-1]
print(x)
```

(2) Écrire un programme pour transformer un tableau de type **int** en un tableau de type **float**.

```
import numpy as np
a = [1, 2, 3, 4]
x = np.array(a,dtype=float)
print(x)
x = np.asarray(a,dtype=float)
```

Exercices

solutions

(3) Écrire un programme qui des 0 tout autour d'un tableau.

```
import numpy as np
x = np.ones((3,3))
print("Original array:")
print(x)
y = np.zeros((x.shape[0]+2,x.shape[1]+2))
y[1:-1,1:-1] = x
print(y)
```

Autre solution

```
x = np.pad(x, pad_width=1, mode='constant', constant_values=0)
```

Exercices

solutions

(3) Écrire un programme qui renvoie un tableau dont les lignes sont renversées.

```
import numpy as np
x = np.arange(12).reshape(3,4)
print("Original array:")
print(x)
y = x[:,range(np.shape[1],-1,-1)]
print(y)
```

(4) Écrire un programme qui les valeurs puis les indices des éléments supérieurs à 10.

```
import numpy as np
x = np.array([[0, 10, 20], [20, 30, 40]])
print("Original array: ")
print(x)
print("Values bigger than 10 =", x[x>10])
print("Their indices are ", np.nonzero(x > 10))
```

Outline

Outline

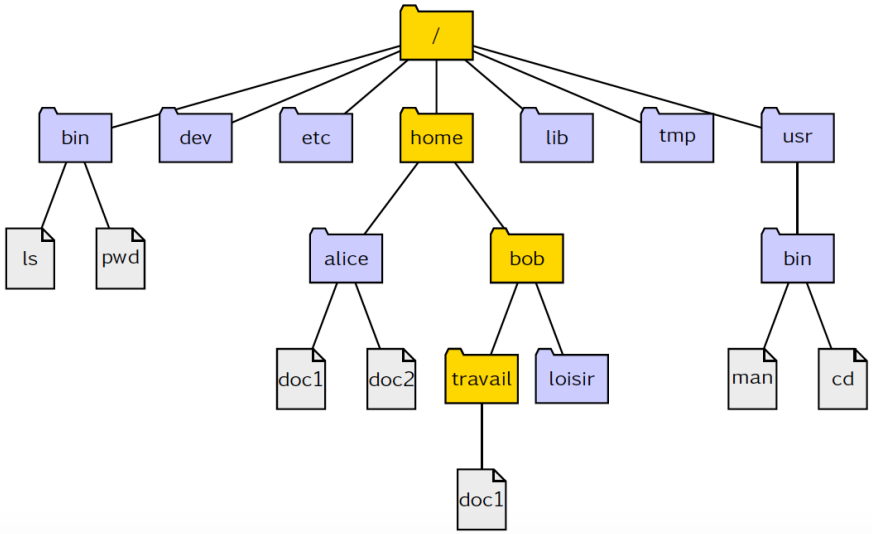
Le module os

- Les instructions permettant à l'interprète de dialoguer avec le système d'exploitation font partie du module os :

```
import os
```

La fonction listdir liste le contenu d'un répertoire :

```
>>> os.listdir('/home/bob/travail')
['doc1']
```



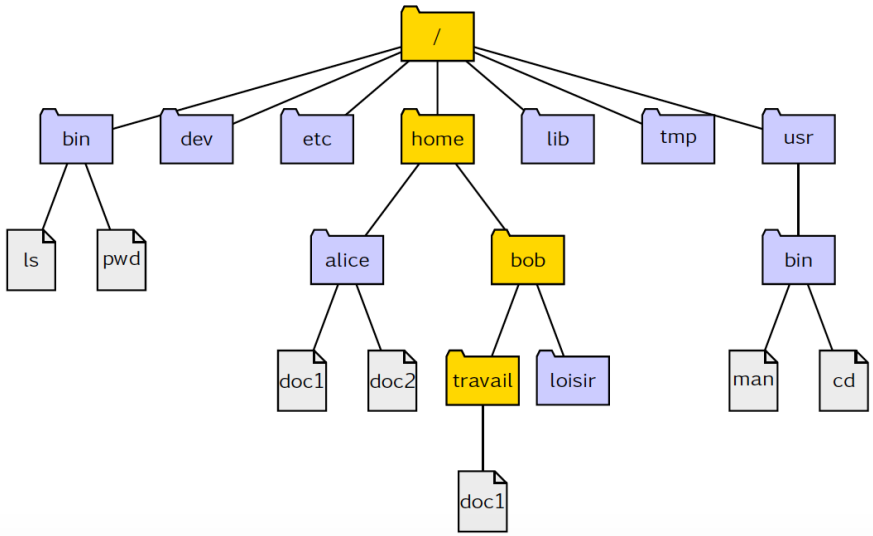
Le module os

La fonction `getcwd` permet d'afficher le répertoire courant :

```
>>> os.getcwd()
```

La fonction `chdir` permet de changer de répertoire :

```
>>> os.chdir('home/alice')
```



Outline

Lecture d'un fichier texte

La fonction **open** propose trois modes d'ouverture d'un fichier :
- en lecture ('r') ;
- en écriture ('w') ;
- en ajout ('a').

Pour ouvrir en lecture le fichier exemple.txt du répertoire courant :

```
>>> comptine = open('exemple.txt', 'r')
```

Nous venons de créer un objet comptine faisant référence au fichier exemple.txt :

```
>>> comptine
<_io.TextIOWrapper name='exemple.txt' mode='r' encoding='UTF-8'>
```

Cet objet est un flux : les caractères sont lisibles uniquement les uns après les autres, sans possibilité de retour en arrière ni de saut en avant.

Lecture d'un fichier texte

Pour lire le fichier dans son entier : la méthode read().

```
>>> comptine.open('r')
>>> comptine.read()
'Am, stram, gram,\nPic et pic et colegram,\nBour et bour et
ratatam,\nAm, stram, gram.'
>>> comptine.close()
```

Pour lire le fichier ligne par ligne : la méthode readlines(n).

```
>>> comptine.open('r')
>>> comptine.readlines()
['Am, stram, gram,\n', 'Pic et pic et colegram,\n',
'Bour et bour et ratatam,\n', 'Am, stram, gram.']
>>> comptine.close()
```

Lecture d'un fichier texte

Lecture par énumération des lignes :

```
>>> comptine.open('r')
>>> n = 0
>>> for l in comptine:
... n += 1
... print('{} :'.format(n), l, end='')
1 : Am, stram, gram
2 : Pic et pic et colegram,
3 : Bour et bour et ratatam,
4 : Am, stram, gram.
>>> comptine.close()
```

Outline

Fichiers CSV Comma-Separated Value

On considère le fichier planetes.txt contenant le texte suivant :

```
Mercure, 2439, 3.7, 88
Vénus, 6052, 8.9, 225
Terre, 6378, 9.8, 365
Mars, 3396, 3.7, 687
```

On ouvre le fichier et on découpe le texte en lignes :

```
>>> planetes = open('planetes.txt', 'r')
>>> lignes = planetes.readlines()
>>> planetes.close()
```

Á cette étape, lignes est une liste de chaînes de caractères égale à :

```
['Mercure, 2439, 3.7, 88\n', 'Vénus, 6052, 8.9, 225\n', 'Terre, 6378, 9
365\n', 'Mars, 3396, 3.7, 687\n']
```

LFichiers CSV Comma-Separated Value

On considère le fichier planetes.txt contenant le texte suivant :

```
Mercure, 2439, 3.7, 88
Vénus, 6052, 8.9, 225
Terre, 6378, 9.8, 365
Mars, 3396, 3.7, 687
```

Chaque ligne est découpée en colonnes par la méthode split :

```
>>> tab = []
>>> for chn in lignes:
... tab.append(chn.split(','))
```

Á cette étape, tab est une liste de listes égale à :

```
[['Mercure', ' 2439', ' 3.7', ' 88\n'], ['Vénus', ' 6052', ' 8.9', ' 22
['Terre', ' 6378', ' 9.8', ' 365\n'], ['Mars', ' 3396', ' 3.7', ' 687\n
```

Fichiers CSV Comma-Separated Value

On considère le fichier planetes.txt contenant le texte suivant :

```
Mercure, 2439, 3.7, 88
Vénus, 6052, 8.9, 225
Terre, 6378, 9.8, 365
Mars, 3396, 3.7, 687
```

On convertit les données numériques :

```
>>> for lst in tab:
...     lst[1] = int(lst[1])
...     lst[2] = float(lst[2])
...     lst[3] = int(lst[3])
```

La liste tab est maintenant prête à être utilisée :

```
[['Mercure', 2439, 3.7, 88], ['Vénus', 6052, 8.9, 225], ['Terre', 6378,
365], ['Mars', 3396, 3.7, 687]]
```

Outline

Fichiers image

Images en noir et blanc

Une image binaire peut être représentée par une matrice p×q dont les éléments, des 0 ou des 1 (plus exactement des booléens), indiquent la couleur du pixel : 0 pour le noir et 1 pour le blanc.

```
1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1
1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1
1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1
1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1
1 1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
1 1 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 1 1
1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 1 1 1 1
1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1
1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 0 1 1 1
1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1
1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 0 1 0 0 1
1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 1 1 1 0 0 0 1 0 1
1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1 1 1 0 1 1 1 0 0 0 1 0 0
1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1 1 1 0 0 1 1 0 0 0 1 0 0
1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1 1 1 0 0 1 1 0 0 0 1 0 0
1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 1 1 1 0 0 1 1 1 0 0 1 1 0
1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 1 1 1 1 1 1 0
1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0 0 0 1 0 0
1 1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 1 0
1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 0 0 1 1 0 0 0 0 0 0 1 0 0
1 1 0 0 0 0 1 1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
1 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1
1 1 1 1 1 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0 1
1 1 1 1 1 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 0 1 1
1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1
1 1 1 1 1 1 1 0 0 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1
1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 0 0 1 0 0 1 1 1 1
1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 1 0 0 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
```

Fichiers image

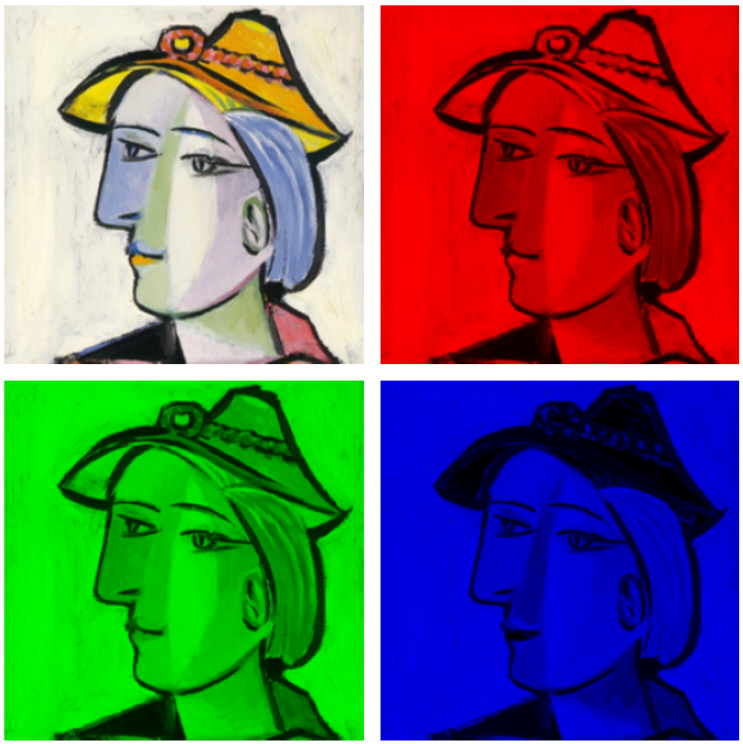
Images en gris

Une image en gris est aussi représentée par une matrice, mais chaque élément détermine la luminance du pixel correspondant (en général un entier non signé codé sur 8 bits). Voici par exemple huit niveaux de gris différents :

Fichiers image

Images en couleurs

Une image en couleur peut être représentée par trois matrices, chacune déterminant la quantité respective de rouge, de vert et de bleu qui constitue l'image (c'est le modèle RGB). Les éléments de ces matrices sont des nombres entiers compris entre 0 et 255 (des entiers non signés sur 8 bits) qui déterminent la luminance de la couleur de la matrice pour le pixel correspondant.



Fichiers image

Lecture

Les fichiers images au format `.png`, `.jpg`, `.jpeg` sont lus, par exemple, avec la fonction imread du module misc de scipy

```
import matplotlib.pyplot as plt
from scipy import misc
image_gray = misc.imread("imgray.png")
plt.imshow(image_gray,cmap="gray")
```

pour une image en couleurs

```
import matplotlib.pyplot as plt
from scipy import misc
image_couleurs = misc.imread("imcouleurs.png")
plt.imshow(image_couleurs)
```

Outline

Paradigme de programmation procédurale

- En programmation procédurale, on sépare code et données.
- Concrètement, on utilise
  - des variables (globales) qui contiennent les données et sont utilisées par les fonctions
  - des fonctions qui peuvent modifier les variables et les passer à d'autres fonctions.

Limites de la programmation procédurale

Exemple

- Dans une entreprise située à Biarritz, Robert (num de sécu 10573123456), est employé sur un poste de technicien

```
# Version 1 : variables globales
# Robert
num_secu_robert = 10573123456
nom_robert = "Robert"
qualifiction_robert = "Technicien"
lieudetravail_robert = "Biarritz"
```

- Pour ajouter un employé, on doit créer de nouvelles variables...

```
# Bernard
num_secu_bernard = 1881111001
nom_bernard = "Bernard"
qualifiction_bernard = "Ingenieur"
lieudetravail_bernard = "Biarritz"
```

Le code devient vite fastidieux !

Limites de la programmation procédurale

Exemple

- Une solution alternative consiste à utiliser des conténeurs (ou listes)

```
# Version 2 : utilisation de conteneurs
tab_robert = [10573123456 ,"Robert","Technicien","Biarritz"]
tab_bernard = [1881111001, "Bernard","Ingenieur","Biarritz"]
tab =[ ]
tab.append(tab_robert)
tab.append(tab_bernard)
print(tab[0][0])
```

Sans être le concepteur, comment interpréter facilement les données de chaque conteneur ? Cela reste délicat. Le paradigme objet propose des réponses.

Vers la programmation orientée objet

Classe

- En programmation orientée objet, nous pouvons définir une structure de donnée particulière par la notion de classe.

```
# Version 3 : utilisation de classes
class Employe:
    num_secu
    nom
    qualification
    lieudetravail
```

- On appelle classe un ensemble d'objets partageant certaines propriétés. Il s'agit d'un concept abstrait (comme le plan d'une maison).
- Exemple : la classe *Voiture* possède les propriétés suivantes (les attributs) :
  - couleur
  - puissance
- Les attributs sont les entités qui définissent les propriétés d'objet.

Vers la programmation orientée objet

Objet

- Un objet est une définition de caractéristiques propres à un élément particulier. Il s'agit d'un élément concret qui contient les propriétés de sa classe, comme une maison qui suit les plans définis préalablement.
- Exemple : instance de la classe Voiture
  1. Voiture "Clio 2007 version roland garros"
  - couleur : verte
  - puissance : 70 Ch
  2. Voiture "307 blue lagoon"
  - couleur : bleue
  - puissance : 90 Ch
- On appelle instance d'une classe un objet avec un comportement et un état, tous deux définis par la classe.

Vers la programmation orientée objet

Objet, exemple

- Mise en place d'objets concrets

```
# Robert
e1 = Employe() # objet e1 : instance de la classe Employe
e1.num_secu = 10573123456
e1.num = "Robert"
e1.qualification = "Technicien"
e1.lieudetravail = "Biarritz"

# Bernard
e2 = Employe() # objet e2 : instance de la classe Employe
e2.num_secu = 1881111001
e2.num = "Bernard"
e2.qualification = "Ingenieur"
e2.lieudetravail = "Biarritz"
```

Vers la programmation orientée objet

Constructeur

- En pratique, on doit initialiser les attributs par une valeur par défaut. Nous faisons alors appelle à une fonction particulière appelée constructeur.

```
class Employe:
    def __init__(self): # constructeur de la classe Employe
        self.num_secu = 000000000000
        self.nom = "no_name"
        self.qualificaition = "novice"
        self.lieudetravail = "Paris"
e3 = Employe()
print( e3.num_secu,e3.nom)
```

- En PYTHON, le constructeur s'appelle en utilisant la fonction **__init__**.
- Le premier argument est toujours **self**.
- En PYTHON, la fonction **__init__** est appelée implicitement à la création d'un objet.

Vers la programmation orientée objet

Constructeur

- La fonction **__init__** peut prendre plusieurs arguments.

```
class Employe:
    def __init__(self,
                         num_secu,
                         nom,
                         qualification,
                         lieudetravail):
        self.num_secu = num_secu
        self.nom = nom
        self.qualificaition = qualification
        self.lieudetravail = lieudetravail
```

Vers la programmation orientée objet

Constructeur

- La fonction **__init__** peut prendre plusieurs arguments.
- On peut alors utiliser les arguments du constructeur pour mettre des valeurs par défaut et instancier des objets "plus facilement" en ne précisant que certaines valeurs.

```
class Employe:
    def __init__(self,
                         num_secu = 00000000,
                         nom = "no_name",
                         qualification = "novice",
                         lieudetravail = "Paris"):
        self.num_secu = num_secu
        self.nom = nom
        self.qualificaition = qualification
        self.lieudetravail = lieudetravail
e1 = Employe(lieudetravail="Brest")
```

Vers la programmation orientée objet

Exercice

- Écrire la classe Voiture.
- Écrire son constructeur avec la possibilité de préciser les valeurs de certains attributs à l'instanciation.
- Instancier 3 objets de la classe Voiture.

Concepts fondateurs

Qu'est-ce qu'une classe ?

- Pour créer un objet, il faut d'abord créer une classe !
- Exemple : pour construire une maison, on a besoin d'un plan d'architecte : - classe = plan, - l'objet = maison.
- "Créer une classe", c'est dessiner les plans de l'objet.
- Une classe est la description d'une famille d'objets qui ont la même structure et les mêmes comportements. Une classe est une sorte de moule à partir duquel sont générés les objets.

Les classes

- Une classe est constituée
  1. d'un ensemble de variables (appelées attributs) qui décrivent la structure des objets ;
  2. d'un ensemble de fonctions (appelées méthodes) qui sont applicables aux objets, et qui décrivent leurs comportements.

Concepts fondateurs

Qu'est-ce qu'un objet ?

- Une classe est une abstraction, elle défini une infinité d'objets.
- Un objet est caractérisé par un état, des comportements et une identité.
  Concrètement, il s'agit :
  - Attributs
  - Méthodes
  - Identité
- On créé des classes pour définir le fonctionnement des objets. On utilise des objets.
- L'état d'un objet
  - regroupe les valeurs instantanées de tous ses attributs
  - évolue au cours du temps
  - est la conséquence de ses comportements passés

Objets

Attributs

- Les attributs sont les caractéristiques de l'objet. Ce sont des variables stockant des informations d'état de l'objet.

```
# definition classe
class Voiture:
    def __init__(self,v1="nomarque",v2="nocolor",v3=0)
        self.marque = v1
        self.couleur = v2
        self.reserveEssence = v3
# Objet
e1 = Voiture()
# Etat
e1.marque = "peugeot"
e1.couleur = "noire"
e1.reserveEssence = 30
```

Objets

Comportement

- Le comportement d'un objet regroupe toutes ses compétences, il décrit les actions et les réactions d'un objet. Il se représente sous la forme de méthodes.

```
class Voiture:
    def __init__(self,v1="nomarque",v2="nocolor",v3=0):
        self.marque = v1
        self.couleur = v2
        self.reserveEssence = v3

    def demarrer(self):
        print("je demarre")
    def arreter(self):
        print("je m'arrete")
# Objet
e1 = Voiture()
e1.demarrer()
e1.arreter()
```

Objets

Comportement

- On peut bien sûr complexifier les méthodes, par exemple

```
class Voiture:
    def __init__(self,v1="nomarque",v2="nocolor",v3=0):
        self.marque = v1
        self.couleur = v2
        self.reserveEssence = v3

    def demarrer(self):
        if self.reserveEssence>0:
            print("je demarre")
        else :
            print("je ne peux pas demarrer")
    def arreter(self):
        print("je m'arrete")
    def rouler(self,nbKm):
        self.reserveEssence = self.reserveEssence - 5*nbKm/100
e1 = Voiture()
e1.reserveEssence = 30
e1.demarrer()
e1.rouler(100)
print(e1.reserveEssence)
```

Objets

Identité

- L'identité est propre à l'objet et le caractérise. Elle permet de distinguer tout objet indépendamment de son état
- En pratique on peut distinguer les objets par leur nom, un numéro de référence, ...

Objets

Exercice : gestion de stock

Écrire une classe telle que

- Un Article du stock est défini par 4 champs :
  - sa référence (numéro)
  - sa désignation (texte)
  - son prix HT
  - sa quantité (nombre d'articles disponibles)
- Pour manipuler ces champs, les services suivants sont fournis :
  - prix TTC
  - prix Transport (taxe 5% prix HT)
  - retirer
  - ajouter

Objets

Destruction (?)

Encapsulation

Principe

- Exemple introductif : classe `Compte` avec un attribut solde

```
class Compte:
    def __init__(self):
        self.solde = 0
    def affiche(self):
        print("le solde de votre compte est de ", self.solde)

c = Compte()
c.affiche()
c.solde = 100000
```

- N'importe qui ayant accès à une instance de la classe `Compte` peut modifier le solde. Ceci n'est pas acceptable.

Encapsulation

Principe

- Lors de la définition d'une classe il est possible de restreindre voire d'interdire l'accès (aux objets des autres classes) à des attributs ou méthodes des instances de cette classe. Comment ?
- Lors de la déclaration d'attributs ou méthodes, en précisant leur utilisation :
  privée ie accessible uniquement depuis "l'intérieur" de la classe ;
  public ie accessible par tout le monde.

```
class Test:
    def __init__(self,valeurPrivee,valeurPublic):
        self.__valeurPrivee = valeurPrivee
        self.valeurPublic = valeurPublic
    def f1(self):
        ...
    def __f2(self):
        ...
t1 = Test()
print(t1.valeurPublic) # OK
print(t1.__valeurPrivee) # ERREUR
```

Encapsulation

Principe

- Dans l'exemple du compte en banque

```
class Compte:
    def __init__(self):
        self.__solde = 0
    def affiche(self):
        print("le solde de votre compte est de ", self.solde)
    def debut(self,laValeur):
        if self.__solde - laValeur>0:
            self.__solde = self.__solde-laValeur
        else :
            print("Solde insuffisant")
```

Encapsulation

Intérêt

- L'encapsulation est donc l"idée de cacher l'information contenue dans un objet et de ne proposer que des méthodes de manipulation de cet objet. Ainsi, les propriétés et axiomes associés aux informations contenues dans l'objet seront assurés/validés par les méthodes de l'objet et ne seront plus de la responsabilité de l'utilisateur extérieur.
- L'objet est ainsi vu de l'extérieur comme une boîte noire ayant certaines propriétés et ayant un comportement spécifié. La manière dont ces propriétés ont été implémentées est alors cachée aux utilisateurs de la classe.

Encapsulation

Règles à respecter

- Rendre privés les attributs caractérisant l'état de l'objet. Si nous voulons respecter le concept d'encapsulation, nous devons donc conserver la portée de cet attribut et définir des accesseurs pour y avoir accès.
- Fournir des méthodes publiques permettant de accéder/modifier l'attribut.

Exemple

```
class Temperature:
    '''
    La classe Temperature represente une grandeur physique
    liee a la notion de chaud et froid.
    '''
    # Constructeur et destructeur
    def __init__(self,laValeur=0):
        self.__valeur = laValeur
    def __del__(self):
        print("Destruction")

    # Methodes publiques : acces aux attributs
    def getValeur(self): # accesseur
        return self.__valeur
    def setValeur(self,value): # mutateur
        self.__valeur = value
            self.__valeur = value
    # Methodes publiques : operations
    def getValeurC(self):
        return self.__valeur - 273.16
    def getValeurF(self):
        return 1.8*delf.getValeurC()+32
```

Intérêt d'utiliser des accesseurs

Lorsque vous utilisez des accesseurs, vous n'êtes pas obligé de vous contenter de lire un attribut dans un getter ou de lui affecter une nouvelle valeur dans un setter : en effet, il est tout à fait possible d'ajouter du code supplémentaire, voire de ne pas manipuler d'attribut en particulier !

```
class Voiture:
    '''
    La classe Voiture est definie par sa largeur
    '''
    # Constructeur et destructeur
    def __init__(self,laValeur=0):
        self.__largeur = laValeur
    def __del__(self):
        print("Destruction")

    # Methodes publiques : acces aux attributs
    def getLargeur(self): # accesseur
        return self.__largeur
    def setLargeur(self,value): # mutateur
        self.__largeur = value
            self.__largeur = valie
    # Methodes publiques : operations
    def getValeurC(self):
        return self.__valeur - 273.16
```

Intérêt d'utiliser des accesseurs

Imaginons que dans notre classe, nous disposions d'une méthode `mettreAJour()` qui redessine l'affichage de notre objet en fonction de la valeur de l'attribut.
Maintenant, nous aimerions limiter les valeurs possibles de l'attribut largeur ; disons qu'il doit être supérieur à 100 et inférieur à 200.

```
    def setLargeur(self,value): # mutateur
        self.__largeur = value
            self.__largeur = value
            if self.getLargeur()<100 :
                self.__largeur = 100
            else if self.getLargeur()>200:
                self.__largeur = 200
            self.mettreAJour()
```