# Informatique et Programmation

## Cours 11

**Jean-Jacques Lévy**

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-py

# Plan

- programmation dynamique (plus longue chaîne commune)
- classes et objets (rappel)
- représentation des arbres

dès maintenant: télécharger Python 3 en `http://www.python.org`

# Programmation dynamique

- plus longue sous-séquence commune entre 2 chaînes de caractères (commande Unix `diff`)

  [ on mémorise les solutions partielles — $m \times n$ opérations ]

```
GAUCHE = 1
HAUT = 2
DIAG = 3
```

```
def new_matrix (m, n, v) :
    a = [ [v for _ in range (n) ] for _ in range (m) ]
    return a
```

```
def longueurSSC (u, v) :
  m = len(u); n = len(v)
  lg = new_matrix (m+1, n+1, 0)
  p = new_matrix (m+1, n+1, 0)
  for i in range(1, m+1):
    for j in range(1, n+1):
      if u[i-1] == v[j-1]:
        lg[i][j] = 1 + lg[i-1][j-1]
        p[i][j] = DIAG;
      elif lg[i][j-1] > lg[i-1][j]:
        lg[i][j] = lg[i][j-1]
        p[i][j] = GAUCHE
      else :
        lg[i][j] = lg[i-1][j]
        p[i][j] = HAUT
  return (lg[m][n], p)
```

u = ‘abcadefg’ v = ‘fbcexyg’

lg

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1
0 0 1 2 2 2 2 2
0 0 1 2 2 2 2 2
0 0 1 2 2 2 2 2
0 0 1 2 3 3 3 3
0 1 1 2 3 3 3 3
0 1 1 2 3 3 3 4
```

p

```
0 0 0 0 0 0 0 0
0 2 2 2 2 2 2 2
0 2 3 1 1 1 1 1
0 2 2 3 1 1 1 1
0 2 2 2 2 2 2 2
0 2 2 2 2 2 2 2
0 2 2 2 3 1 1 1
0 3 2 2 2 2 2 2
0 2 2 2 2 2 2 3
```

# Programmation dynamique

- plus longue sous-séquence commune entre 2 chaînes de caractères (commande Unix `diff`)

  [ on mémorise les solutions partielles — $m \times n$ opérations ]

```
def ssc (u, v) :
    m = len(u); n = len(v)
    lgp = longueurSSC (u, v)
    lg = lgp[0]; p = lgp[1]
    r = ''; i = m; j = n;
    while lg > 0 :
        if p[i][j] == DIAG :
           r = u[i-1] + r
           i = i - 1; j = j - 1;
           lg = lg - 1
        elif p[i][j] == GAUCHE :
           j = j - 1
        else :
           i = i - 1
    return r


print (ssc ('abcadefg', 'fbcexyg'))
```

u = ‘abcadefg’ v = ‘fbcexyg’

lg

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1
0 0 1 2 2 2 2 2
0 0 1 2 2 2 2 2
0 0 1 2 2 2 2 2
0 0 1 2 3 3 3 3
0 1 1 2 3 3 3 3
0 1 1 2 3 3 3 4
```

p

```
0 0 0 0 0 0 0 0
0 2 2 2 2 2 2 2
0 2 3 1 1 1 1 1
0 2 2 3 1 1 1 1
0 2 2 2 2 2 2 2
0 2 2 2 2 2 2 2
0 2 2 2 3 1 1 1
0 3 2 2 2 2 2 2
0 2 2 2 2 2 2 3
```

# Programmation dynamique

- calcul de fibonacci

  [ on mémorise les calculs intermédiaires ]

```
def fib (n) :
    if n == 0 or n == 1 :
        return n
    else :
        return fib (n-1) + fib (n-2)
```

fibonacci

```
>>> fib (10)
55
>>> fib (20)
6765
>>> fib (35)
9227465
```

calcul en temps exponentiel

[ en consommant l'espace de la récursion
ici aussi espace linéaire ]

```
def fib (n) :
    a = (n+1)*[0]
    a[1] = 1
    for i in range(2, n+1) :
        a[i] = a[i-1] + a[i-2]
    return a[n]
```

fibonacci

```
>>> fib (10)
55
>>> fib (20)
6765
>>> fib (35)
9227465
```

calcul en temps linéaire

[ en consommant plus de mémoire ]

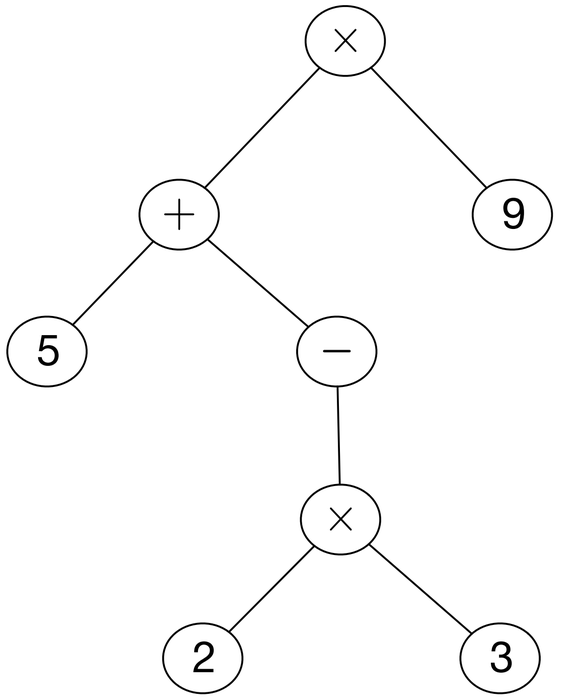
- plus court chemin dans un graphe [Dijkstra]

  [ cf. plus tard]

# Arbres (représentation 4)

• une arborescence est une représentation par lien arrière

```
arbre = [ ('x', None),
          ('+', 0),
          ('5', 1),
          ('-', 1),
          ('x', 3),
          ('2', 4),
          ('3', 4),
          ('9', 4)]
```




• ou encore

```
val = ['x', '+', '5', '-', 'x', '2', '3', '9']
pere = [None, 0, 1, 1, 3, 4, 4, 0]
```

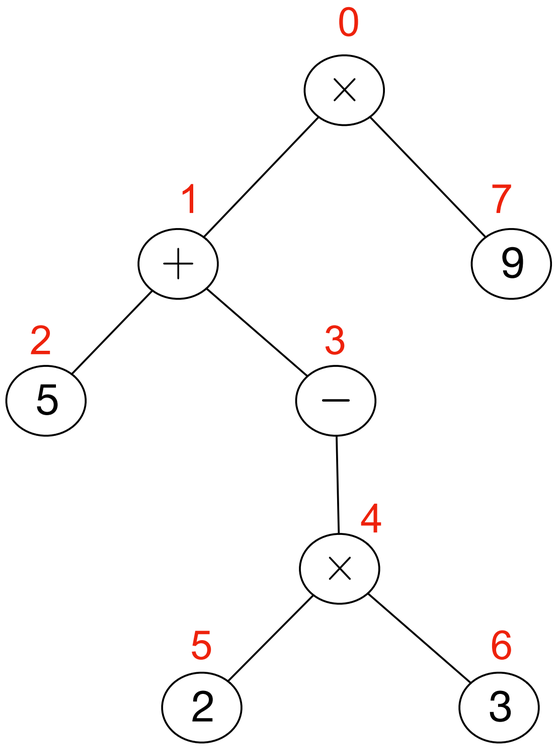
• la représentation plus souple car ne distinguant pas l'arité des noeuds

• mais elle ne permet pas un simple parcours d'arbre

**Exercice** Passer de la représentation 1 à la représentation 4 et réciproquement.

# Parcours d’arbre

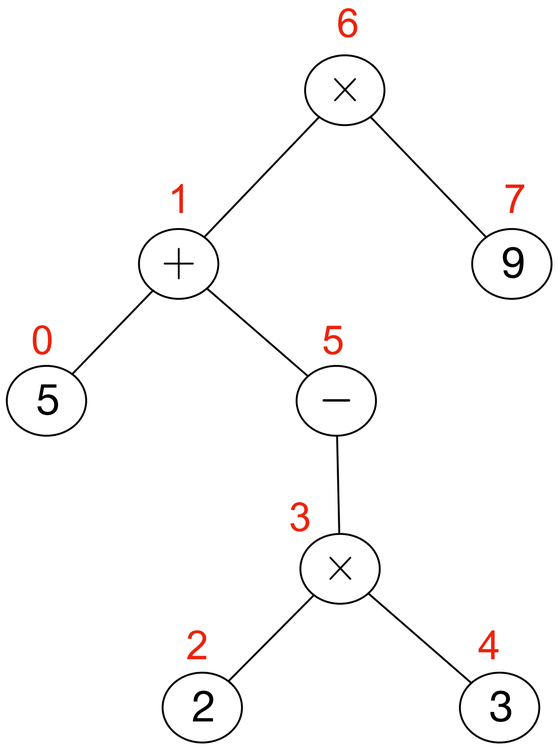
• 3 parcours d’arbre (préfixe, infixe, postfixe)




préfixe

• notation polonaise préfixe

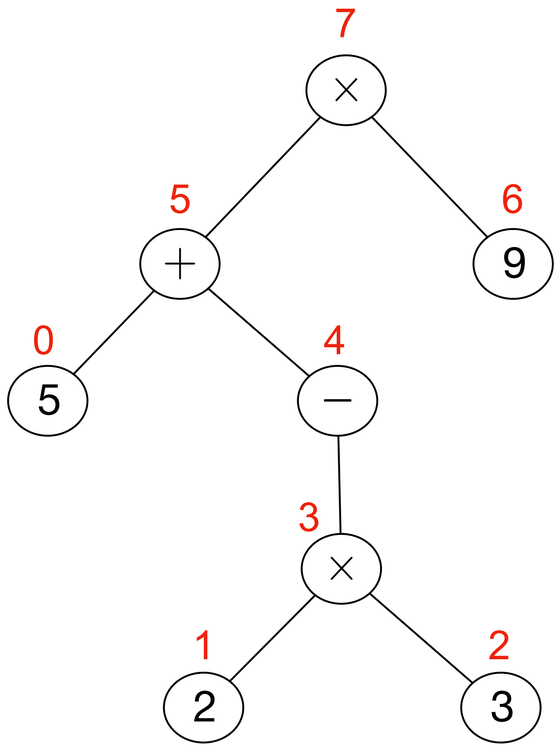
x + 5 - x 2 3 9




infixe

• notation arithmétique

(5 + - (2 x 3)) x 9




postfixe

• notation polonaise postfixe

5 2 3 x - + 9 x

# Parcours d'arbre

- générer les notations préfixe, postfixe et infixe

```
def polprefix (a) :
    if isinstance (a, Feuille) :
        return a.val
    elif isinstance (a, Noeud_Un) :
        return a.val + ' ' + polprefix (a.fils)
    else :
        return a.val \
              + ' ' + polprefix (a.gauche) \
              + ' ' + polprefix (a.droit)
```

```
def polpostfix (a) :
    if isinstance (a, Feuille) :
        return a.val
    elif isinstance (a, Noeud_Un) :
        return polpostfix (a.fils) + ' ' + a.val
    else :
        return polpostfix (a.gauche) \
             + ' ' + polpostfix (a.droit) \
             + ' ' + a.val
```

```
def notinfixe (a) :
    if isinstance (a, Feuille) :
        return a.val
    elif isinstance (a, Noeud_Un) :
        return '(' + a.val + ' ' + notinfixe (a.fils) + ')'
    else :
        return '(' + notinfixe (a.gauche) \
             + ' ' + a.val \
             + ' ' + notinfixe (a.droit) + ')'
```

trop de parenthèses

[ on verra plus tard pour les enlever ]

- notation polonaise préfixe

  x + 5 - x 2 3 9

- notation infixe

  (5 + - (2 x 3)) x 9

- notation polonaise postfixe

  5 2 3 x - + 9 x

# Arbres (représentation 5)

- Arbres n-aires avec nombre arbitraire de fils

```
class Noeud:
    def __init__ (self, x, l) :
        self.val = x
        self.fils = l
    #
    def __str__ (self) :
        r = ''
        for a in self.fils :
            r = r + ', ' + str(a)
        return "Noeud ({})".format (r[2:])

class Feuille:
    def __init__ (self, x) :
        self.val = x
    #
    def __str__ (self) :
        return "Feuille ({})".format (self.val)

a = Noeud (10,
        [Noeud (12, [Feuille (3)]),
                Feuille (4), Feuille (5)])
print (a)
```

```
def __str__ (self) :
    r = ', '.join(map(str, self.fils))
    return "Noeud ({})".format (r)
```

# à faire

- retour sur les objets et les arbres
- analyses lexicale et syntaxique
- modularité et programmation objet
- programmation graphique
- algorithmes géométriques
- calculs flottants et méthodes numériques
- programmation de plusieurs fils de calcul
- assertions et logique des programmes
- introduction à l'informatique théorique
- etc

vive l’informatique

et

la programmation !