http://deptinfo.unice.fr/~roy

Modules
Polynômes

from  import  , 

Les modules de Python

- Le **nom d'un fichier Python** se termine par .py et ne contient que des lettres minuscules, des chiffres et des soulignés. Aucun espace !

  essai2.py essai_tortue.py ~~Essai tortue.py~~

- Un **module** est un fichier xxxxxx.py écrit en Python et contenant :
  - des définitions
  - des instructions (par exemple d'affichage)

- Un **module** peut être destiné :

- à être directement **exécuté**. Lorsqu'il est court et effectue une action ré-utilisable, on parle souvent d'un ***script***.	ou et	- à être **utilisé** par un autre module. Il exporte alors un certain nombre de fonctionnalités.

Un exemple de script

- Un programme Python peut être **exécuté** sur la ligne de commande du système d'exploitation : Unix (Mac, Linux) ou Windows.
- Exemple au Terminal Unix sur Mac :

fichier monscript.py

```
#!/usr/local/bin/python3.4
import sys
print(sys.version)
print('Salut tout le monde !')
```

chemin vers Python
importation d'un module
instructions

Terminal Unix

```
~$ chmod u+x monscript.py
~$ ./monscript.py
3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 00:54:21)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)]
Salut tout le monde !
~$
```


Un exemple de module utilitaire

fichier "tva.py" ← *Un module utilitaire*

```
TVA = 19.6
COEF = 1 + TVA / 100
def prix_ttc(p) :
    return p * COEF
print('Module tva.py chargé !')
```

Mon programme

fichier "use_tva.py"

```
import tva
def bilan(x) :
    prix = tva.prix_ttc(x)
    print('Avec un taux de ',tva.TVA,end = '%, ')
    print('le prix ttc est ',prix)
bilan(26)
```

Exécution → `Avec un taux de 19.6%, le prix ttc est 31.096`

Un module s'exécute dans un espace de noms

• Un module Python s'exécute dans un **espace de noms** (*namespace*).

• Dans l'espace de noms du module `tva.py`, les variables `TVA`, `COEF` et `prix_ttc` existent. Elles ne sont pas visibles d'un autre module. Un autre module peut introduire une variable `COEF` dans son propre espace de noms, qui n'aura rien à voir avec la variable `COEF` de `tva.py`.

fichier "tva.py"

```
TVA = 19.6
COEF = 1 + TVA / 100
def prix_ttc(p) :
    return p * COEF
#print('Module tva.py chargé !')
```

TVA COEF
prix_ttc

espace de noms de tva.py

• L'existence d'espaces de noms permet de protéger le programmeur de conflits entre des variables de même nom situées dans des modules distincts. La sécurité avant tout !

Que fait l'instruction 'import...' ?

• Utilisée dans le module `use-tva`, l'instruction `import tva` introduit le mot `tva` dans l'espace de noms de `use-tva`. Le mot `tva` devient un nom de variable dont la valeur est un module :

```
>>> import tva
>>> tva
<module 'tva' from '/Users/roy/Documents/Python/tva.py'>
```

• Je peux alors accéder aux variables du module `tva` en préfixant leur nom par le nom de leur module :

```
>>> TVA
NameError: name 'TVA' is not defined
>>> tva.TVA
19.6
```

```
>>> TVA = tva.TVA
>>> TVA
19.6
```


Que fait l'instruction 'from... import...' ?

• Utilisée dans un module M, l'instruction `from tva import TVA` introduit le mot `TVA` dans l'espace de noms du module M. Le mot `TVA` devient un nom de variable du module M :

```
>>> from tva import TVA
>>> TVA       # mais tva.TVA n'existe pas !!
19.6
```

*car le mot **tva** n'est pas importé !*

```
>>> from tva import TVA, prix_ttc
>>> (TVA, prix_ttc)
(19.6, <function prix_ttc at 0x10863c7a0>)
```

• Pour importer tout l'espace de noms du module `tva` :

```
from tva import *
```

N.B. En général, `import...` est *moins risqué* que `from... import...` car ce dernier peut introduire des conflits de noms !

Puis-je limiter ce qui est importé par * ?

• Dans le module `tva.py`, j'ai défini trois noms : `TVA`, `COEF` et `prix_ttc`.

• Or la variable `TVA` ne sert qu'au calcul de `COEF`. Je n'ai pas envie que l'utilisateur qui importe le module avec `from tva import *` la voit ! Je la considère comme privée au module.

• Comment l'empêcher de sortir ?

• SOLUTION 1 : je la nomme `_TVA` avec un souligné en tête.

• SOLUTION 2 : dans `tva.py`, je définis une variable `__all__` contenant le tuple des noms exportables : `__all__ = ('COEF', 'prix_ttc')`.

```
>>> import os                    # pour changer de répertoire courant
>>> os.chdir('/Users/roy/Documents/Python/')
>>> from tva import *
>>> COEF        >>> prix_ttc                              >>> TVA
1.196           <function prix_ttc at 0x103d28290>        Error
```

Que fait l'instruction 'from... import... as...' ?

• Utilisée dans un module M, l'instruction `from tva import TAUX as T` introduit le mot T dans l'espace de noms du module courant. Le mot T devient un nom de variable du module M, remplaçant le nom TAUX.

• Imaginons que la variable TVA soit définie dans les modules foo et bar.

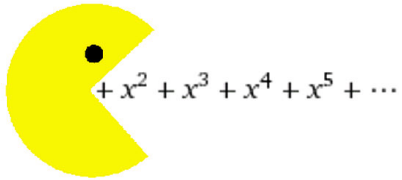
```
>>> from foo import TVA
>>> TVA
1
>>> from bar import TVA
>>> TVA
2
```

• Mieux vaut les nommer foo.TVA et bar.TVA, ou bien :

```
>>> from foo import TVA as FT
>>> from bar import TVA as BT
```

```
>>> (FT, BT)
(1, 2)
```


Un module polynôme en Python



Représentation des polynômes

• Si les **polynômes** ont beaucoup de coefficients ≠ 0, on opte pour une **représentation dense** avec des listes de **réels** :

$2x^4 - 7x^3 - 3x + 5$ ➟ `[5, -3, 0, -7, 2]`

*N.B. Suivant les **puissances croissantes** !*
Ainsi le coefficient de x^i sera p[i]...

• Si au contraire les polynômes ont beaucoup de coefficients = 0, on opte pour une **représentation creuse** avec des listes de **monômes** :

$2x^{100} - 3$ ➟ `[[2, 100], [-3, 0]]`

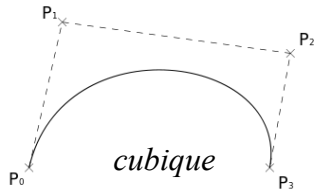
un "monôme"
`[coefficient, exposant]`

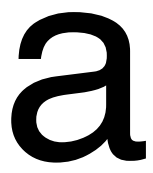
*N.B. Suivant les **puissances décroissantes** !*

Un module polynome.py

• Les **polynômes** sont des objets mathématiques très importants, pour les approximations de fonctions, le graphisme, etc.

• Les caractères par exemple sont dessinés grâce à des polynômes de degré 2 ou 3 (courbes de Bézier).





• Nous allons développer en détail un module polycreux.py contenant les définitions des opérations élémentaires de l'algèbre des polynômes (en représentation creuse).

• Un **polynôme** non nul sera représenté par une liste de monômes, suivant les puissances décroissantes.

• En maths :

$$p = 2x^5 - x^4 - 2x$$

$$q = x^4 + 7x^2 - 1$$

• En Python :

```
p = [[2,5],[-1,4],[-2,1]]

q = [[1,4],[7,2],[-1,0]]
```

• Le **polynôme nul** sera fourni par une fonction constante.

```
def poly0() :
    return []

def is_poly0(p) :
    return p == []
```


• Le **degré** d'un monôme [c,e] est son second élément. Le degré d'un polynôme est celui de son monôme de plus haut degré (en tête).

```
def degre(mp) :   # degré d'un monome ou d'un polynome
    if is_poly0(mp) : return float("-Inf")
    if type(mp[0]) == list :
       return mp[0][1]
    return mp[1]
```

```
>>> p[0]
[2,5]
```
⇒
```
>>> degre(p[0])
5
```
⇒
```
>>> degre(p)
5
```

• Le **coefficient** d'un monôme [c,e] est son premier élément. Le coefficient (dominant) d'un polynôme est celui de son monôme de tête.

```
def coef(mp) :           # coefficient dominant
    if is_poly0(mp) : return 0
    if type(mp[0]) == list :
       return mp[0][0]
    return mp[0]
```

```
>>> coef(p)
2
>>> coef(p[0])
2
```


Addition : monôme + polynôme

• Commençons par le cas simple : monôme + polynôme. Soit à insérer le monôme cx^e dans le polynôme p, à sa place !

• Comprenons-nous bien : il ne s'agit PAS de *modifier* le polynôme p en lui ajoutant cx^e, mais de *construire* le nouveau polynôme cx^e + p.

• Il s'agit d'un algorithme d'insertion :

insérer $5x^3$ dans $p = 2x^5 - x^4 - 2x$

insérer [5,3] dans [[2,5],[-1,4],[-2,1]]

• On cherche donc le premier monôme de degré d ≤ e

• S'il existe déjà un monôme de degré e, attention !


```
def mono_plus_poly(c,e,p) :                    # cx^e + p
    if is_poly0(p) :
        return [[c,e]]
    # on cherche le premier monôme de degré d ≤ e
    for i in range(len(p)) :
        m = p[i]
        d = degre(m)  # le degré du terme d'indice i de p
        if e == d :   # trouvé !
            essai = c + coef(m)          # attention !
            if essai == 0 : return p[:i] + p[i+1:]
            return p[:i] + [[essai,d]] + p[i+1:]
        elif e > d :    # insertion d'un nouvel élément !
            return p[:i] + [[c,e]] + p[i:]
    return p + [[c,e]]
```

```
>>> p
[[2,5],[-1,4],[-2,1]]
>>> mono_plus_poly(5,3,p)
[[2,5],[-1,4],[5,3],[-2,1]]
```

```
>>> mono_plus_poly(1,4,p)
[[2,5],[-2,1]]
>>> mono_plus_poly(6,0,p)
[[2,5],[-1,4],[-2,1],[6,0]]
```

Addition : polynôme + polynôme

• Pour additionner p1 et p2, il suffit d'additionner chaque monôme de p1 au polynôme p2, avec mono_plus_poly :

```
def add(p1,p2) :
    if is_poly0(p1) :
        return p2
    for m in p1 :
        p2 = mono_plus_poly(coef(m),degre(m),p2)
    return p2
```

ATTENTION : L'affectation `p2 = mono_plus_poly(...,p2)` *modifie (**sans mutation !**) le paramètre* `p2` *de la fonction* `add`*, mais un paramètre est une variable locale, donc restaurée à sa valeur en entrée de la fonction !*

```
>>> p
[[2,5],[-1,4],[-2,1]]
>>> q
[[1,4],[7,2],[-1,0]]
```

```
>>> add(p,q)
[[2,5],[7,2],[-2,1],[-1,0]]
>>> q        # non modifié !
[[1,4],[7,2],[-1,0]]
```


Soustraction : polynôme - polynôme

• Il est bien connu qu'une soustraction n'est qu'une addition !

$$p - q = p + (-1) * q$$

• Programmons la multiplication kp d'un polynôme p par un scalaire k. Il s'agit de la loi externe d'un espace vectoriel :

```
def mul_ext(k,p) :      # real x polynome --> polynome
    if is_poly0(p) : return poly0()
    res = poly0()
    for m in p :
        res.append([coef(m) * k, degre(m)])
    return res
```

```
>>> mul_ext(2,p)
[[4,5],[-2,4],[-4,1]]
>>> sub(p,q)
[[2, 5], [-2, 4], [-7, 2], [-2, 1], [1, 0]]
```

$2x^5-2x^4-7x^2-2x+1$

cf TP !

Multiplication : polynôme * polynôme

• Pour calculer le produit du polynôme p1 par le polynôme p2, un algorithme consiste à multiplier chaque monôme de p1 par p2 en additionnant les résultats obtenus :

$$\begin{aligned} p_1 \times p_2 &= (a_n x^n + a_{n-1}x^{n-1} + \cdots)p_2 \\ &= (a_n x^n)p_2 + (a_{n-1}x^{n-1})p_2 + \cdots \end{aligned}$$

• Tout revient donc à savoir multiplier un monôme par un polynôme :

```
def mono_mul_poly(c,e,p) :                    # cx^e x p
    if isPoly0(p) : return poly0()
    res = poly0()
    for m in p :
        res = mono_plus_poly(c * coef(m),e + degre(m),res)
    return res
```

```
>>> p
[[2, 5], [-1, 4], [-2, 1]]
```

```
>>> mono_mul_poly(2,3,p)
[[4, 8], [-2, 7], [-4, 4]]
```


Valeur d'un polynôme en un point

• Soit p un **polynôme** et x un nombre réel. Comment calculer la valeur de p en x, bien que p soit une **liste** et non une fonction ?

$p = 2x^5 - x^4 - 2x$ ⟹ `p = [[2,5],[-1,4],[-2,1]]`

• Valeur du monôme [2,5] en x ? Il s'agit bien de $2x^5$. Valeur du polynôme p en x : la somme des valeurs en x de ses monômes.

```
def valeur(p,x) :
    .....
```

obligé car :

```
>>> p(1)
'list' object is not callable
```

• Il reste possible de construire la **fonction polynôme** associée à p :

```
def fpoly(p) :
    def g(x) :
        return valeur(p,x)
    return g
```

⇔

```
def fpoly(p) :
    return lambda x : valeur(p,x)
```

```
>>> f = fpoly(p)
>>> f(1)
-1
```