J.Courtin *PC* LVH Caen*

INFORMATIQUE

Les classes en Python

5 - Les classes en Python

L'idée de classe en python est de créer sa propre structure de données :

C'est à dire un objet !!!

La modélisation algorithmique d'un problème spécifique nécessite, pour des raisons de performances, à la fois en terme de mémoire et surtout en terme de temps d'exécution, **de créer une structure de données elle aussi spécifique**.

<u>ex 1 :</u> Les classes Built-in ou types primitifs

Les listes, ensembles, dictionnaires de Python sont en fait des structures de données bien particulières codées en C pour les rendre très efficaces dans l'interpréteur Python.

<u>ex 2 :</u> Des algorithmes plus élaborés

arbre de tri binaire

- **Algorithme de tri** [arbres, et noeuds]
- **Implémenter une pile de données, une queue**

class point

- **Trouver le contour d'un ensemble de points** (enveloppe convexe),
- **Trouver des alignements de points** en 2D en 3D etc...

[=> **Application médicale de la reconnaissance de formes en traitement des images**]

- Etude d'interconnection dans un réseau :

Soient N noeuds dont certains sont connectés entre eux avec N >>> 1
est-ce que deux ordinateurs distants quelconques sont bien sur le même réseau ?
[cf PB de percolation]

collection / connectivité

- Réalisation d'un planificateur de tâches

ex : emploi du temps
ex : optimisation de l'utilisation des machines dans une usine

pile

graph

- Modéliser efficacement un problème à N-corps.

- Algorithme Naïf en N^2 beaucoup trop lent en astrophysique => algorithme en N.Log(N)
- décrire les nuées d'oiseaux, bancs de poissons
  => application à la circulation / couloir aériens / drônes.

queue prioritaire

- Trouver le chemin le plus court dans une ville avec des sens unique.

- urbanisme : conception des voies de circulation
- Application : «**TomTom**»
  [=> il faut implémenter des graphes]

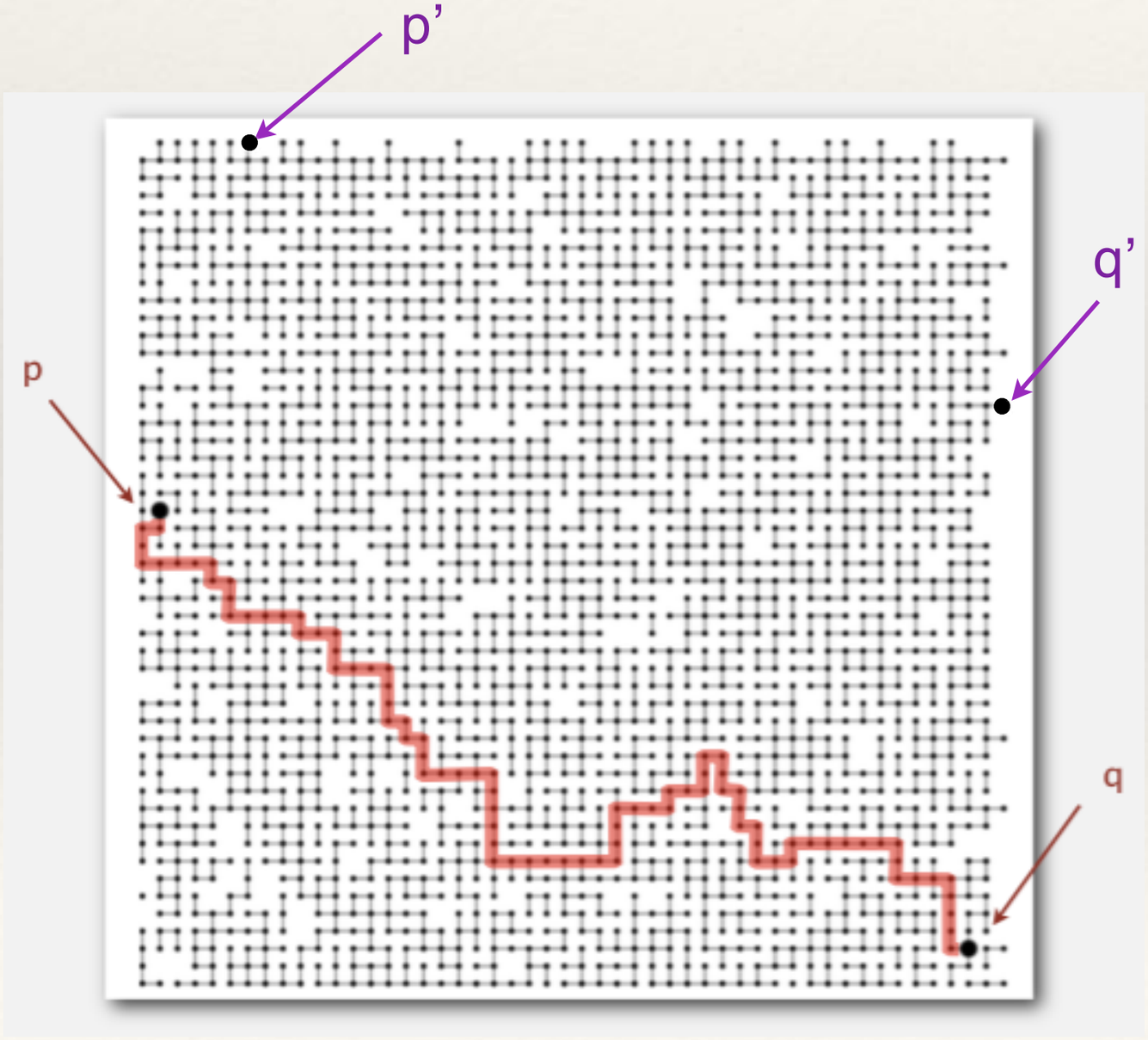
graph + queue prioritaire

Ces algorithmes ne pourront être efficaces, voire faisables, qu'avec une structure de données adaptée au problème.

Ex : Etude d'interconnection dans un réseau

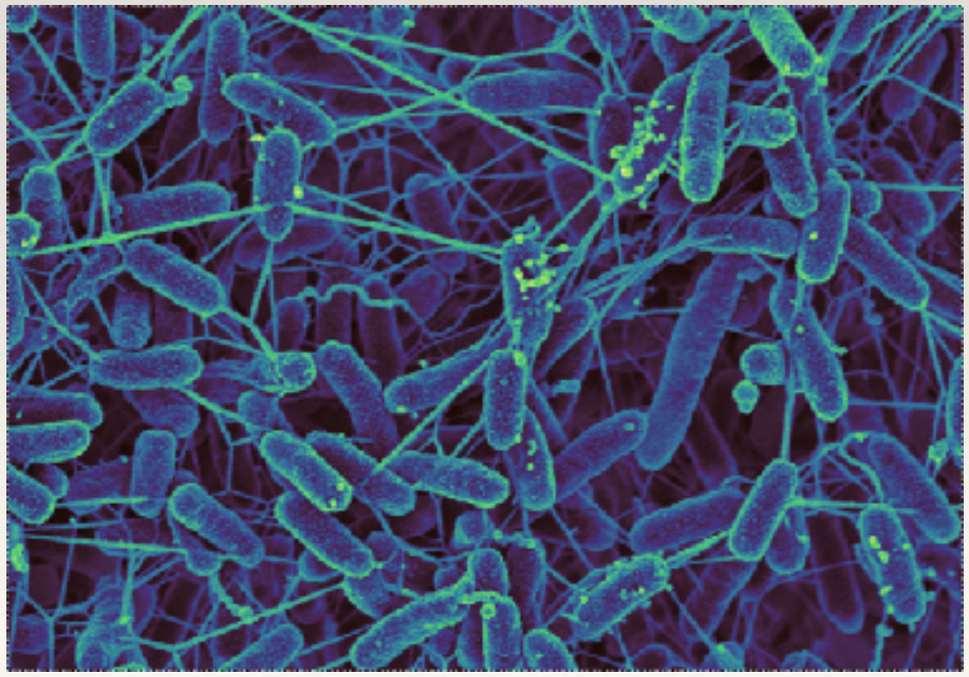
On considère un réseau dynamique où des connexions se font et se défont au cours du temps (ex : terminaux informatiques, routeurs, followers Twitter
population de bactéries, contamination grippe, MST (SIDA),
passage du courant par un réseau d'impuretées)

Comment savoir à un instant t, si il existe un chemin reliant deux points du réseau ?

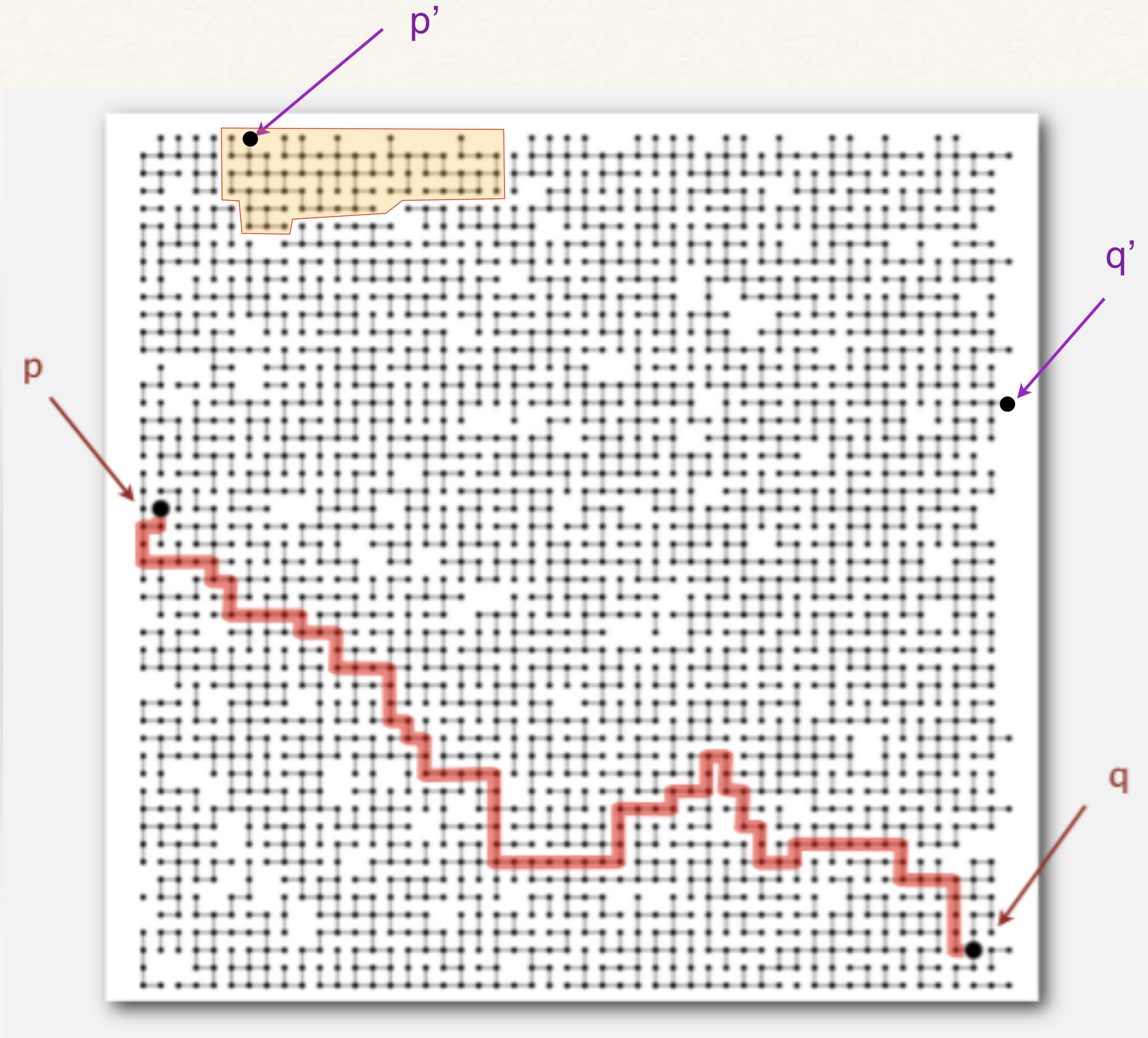


collection / connectivité

Bactéries reliées par des micro-canneaux



Il y a des îlots dans le réseau : p’ et q’ ne sont pas connectés !



L'idée générale d'une classe d'objets est de **structurer l'information en mémoire :**

- demande plus de mémoire (a priori).
- demande plus de temps pour organiser l'information, la mettre à jour.

Mais on va réduire considérablement le nombre de tâches a traiter
=> Une bonne structure de données doit amener un gain de temps considérable.

<u>Ex : Pb. à N-corps</u>

Comment ne garder pour chaque poisson que la donnée des poissons de son voisinage ?
=> comment définir un voisinage (de façon dynamique)

queue prioritaire

Nuées d'oiseaux

Bancs de poissons

--- Un exemple de principe ---

```
>>> class maClass:
...         prop_vide = None          #On peut créer de base un ensemble de variables propres à l'objet
...         prop_long = 0             #ces variables sont parfois initialisées ou inconnnues : None -> pas d'objet
...         data = []

...        def  __init__(self, L):   #CONSTRUCTEUR :   C'est lui qui «fabrique»  les instances mémoire de l'objet
...               self.data = L
...               self.prop_long = len(L)
...               self.prop_vide = (len(L)==0)
...


>>>  C = maClass([1, 2, 3, "trois ptits chats"])    #On crée deux instances de maClass indépendantes
>>>  D = maClass([])
>>>  C, D
(<__main__ . maClass instance at 0x2c9558>,  <__main__ . maClass instance at 0x2c9468>)


>>> C.data ; D.data
[1, 2, 3, 'trois ptits chats']
[]


>>> C.maNouvelleVariable = ['Toto']      #On peut ajouter une variable à
                                         #l'espace de nommage de l'instance C
>>> dir(C); dir(D)                       #Sans rien changer à l'instance D
['__doc__', '__init__', '__module__', 'data', 'maNouvelleVariable', 'prop_long', 'prop_vide']
['__doc__', '__init__', '__module__', 'data', 'prop_long', 'prop_vide']

>>> C.prop_vide,  C.prop_long,  C.maNouvelleVariable  ;   D.prop_vide,  D.prop_long
(False, 4, ['Toto'])
(True, 0)
```

Mais ce n'est pas un usage très propre d'une classe

Attention : >>> D.maNouvelleVariable [C'est une altération de maClass]
Traceback (most recent call last): File "<stdin>", line 1, in <module>
AttributeError: maClass instance has no attribute 'maNouvelleVariable'

maNouvelleVariable : N'apparaît pas dans les attributs de la classe bien qu'elle soit dans l'espace de nommage de C.
=> usage impropre de la classe : maClasse

```
>>> help(C)

Help on instance of maClass in module __main__:

class maClass
 |  Methods defined here:
 |  __init__(self, L)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  data = []
 |  prop_long = 0
 |  prop_vide = None
```

Ensemble des méthodes ou fonctions :
- agissant sur l'instance d'objet lui même (self)
- prenant d'autres variables

Rq : L'instance de l'objet est tjrs. désigné par «self» par convention.

Ensemble des variables ou «Attributs» de l'objet :
- son contenu sous forme de liste
- le nombre d'élément(s)
- est-ce que le contenu est vide

Conclusion sur la construction d'un objet :

- On comprend que la création d'un objet consiste en Python à la construction d'un espace de nommage qui liste à la fois les données et les méthodes agissant sur l'objet.

- A nous de bien définir l'ensemble des propriétés que l'on veut garder en mémoire, et de laisser de coté celles, plus superflues, qui occuperaient de la mémoire inutilement. **[l'objet pourra être dupliqué à volonté -> instances de classe]**

Les méthodes «built-in» pour les classes :

Certains nom de fonction sont réservés, car ils correspondent à de la syntaxe Python de base : cela permet par exemple de contrôler l'usage qui sera fait de nos objets pour les commandes natives en Python :

Ex : **objet1 + objet2** le symbole « **+** » n'a pas la même signification pour une chaîne, un entier, un booléen => on veut pouvoir écrire cette opération avec ce symbole et éviter d'écrire addition(objet1, objet2)

__init__(self, ...) Cette fonction crée l'instance de classe lorsque l'on appelle la classe par son nom C = maClass(....)

__str__(self) Cette fonction sera appelée losque l'on veut afficher la valeur de l'instance à l'aide de la commande **print**

__add__(self, other) **Réalise les 4 opérations : + - * /** entre l'instance (self) et une autre instance.
__sub__(self, other)
__mul__(self, other)
__truediv__(self, other) Attention __div__ pour les versions > 2.1 ....

__pow__(self, other) Fonction puissance **

Operation	Syntax	Function
Addition	`a + b`	`add(a, b)`
Concatenation	`seq1 + seq2`	`concat(seq1, seq2)`
Containment Test	`obj in seq`	`contains(seq, obj)`
Division	`a / b`	`div(a, b)` (without `__future__.division`)
Division	`a / b`	`truediv(a, b)` (with `__future__.division`)
Division	`a // b`	`floordiv(a, b)`
Bitwise And	`a & b`	`and_(a, b)`
Bitwise Exclusive Or	`a ^ b`	`xor(a, b)`
Bitwise Inversion	`~ a`	`invert(a)`
Bitwise Or	`a | b`	`or_(a, b)`
Exponentiation	`a ** b`	`pow(a, b)`
Identity	`a is b`	`is_(a, b)`
Identity	`a is not b`	`is_not(a, b)`
Indexed Assignment	`obj[k] = v`	`setitem(obj, k, v)`
Indexed Deletion	`del obj[k]`	`delitem(obj, k)`
Indexing	`obj[k]`	`getitem(obj, k)`
Left Shift	`a << b`	`lshift(a, b)`
Modulo	`a % b`	`mod(a, b)`
Multiplication	`a * b`	`mul(a, b)`

Negation (Arithmetic)	`- a`	`neg(a)`
Negation (Logical)	`not a`	`not_(a)`
Positive	`+ a`	`pos(a)`
Right Shift	`a >> b`	`rshift(a, b)`
Sequence Repetition	`seq * i`	`repeat(seq, i)`
Slice Assignment	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
Slice Deletion	`del seq[i:j]`	`delitem(seq, slice(i, j))`
Slicing	`seq[i:j]`	`getitem(seq, slice(i, j))`
String Formatting	`s % obj`	`mod(s, obj)`
Subtraction	`a - b`	`sub(a, b)`
Truth Test	`obj`	`truth(obj)`
Ordering	`a < b`	`lt(a, b)`
Ordering	`a <= b`	`le(a, b)`
Equality	`a == b`	`eq(a, b)`
Difference	`a != b`	`ne(a, b)`
Ordering	`a >= b`	`ge(a, b)`
Ordering	`a > b`	`gt(a, b)`

https://docs.python.org/2/library/operator.html#module-operator

La class simpleBoole :

Une implémentation très simple des règles de Boole, avec un objet qui prend la valeur de type string : «Vrai» ou «Faux» ou None en l'absence d'argument.

```
class simpleBoole(builtins.object)
 |  Methods defined here:
 |
 |  __add__(self, other)
 |      #Opération "OU"  => addition : +
 |
 |  __init__(self, Val=None)
 |      #Constructeur
 |
 |  __mul__(self, other)
 |      #Opération "AND"  --> multiplication : x
 |
 |  __neg__(self)
 |      #Opération "NOT"
 |
 |  __str__(self)
 |      #afficheur sous forme textuelle
 |
 |  __sub__(self, other)
 |      #Opération "OU not"  => soustraction : -
```

Python 3

```
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |
----------------------------------------------------------------------

 |  Data and other attributes defined here:
 |
 |  Value = None
```

```
A = simpleBoole("Vrai")
B = simpleBoole("Faux")
C = simpleBoole()

NotA = -A
NotB = -B
NotC = -C


print("A : ", A) ; print("B : ",B) ; print("C : ",C)
print("C = simpleBoole() => None par défaut d'argument")
print("")


print("A + B :", A+B)
print("A + NotA :", A+NotA)
print("A - B : ", A - B)
print("A + C : ", A + C )
print("")
print("A * B :", A * B)
print("A * NotA :", A * NotA)
print("A * C : ", A * C )
```

```
A :  Vrai
B :  Faux
C :  None
C = simpleBoole() => None par défaut d'argument


A + B : Vrai
A + NotA : Vrai
A - B :  Vrai
A + C :  None

A * B : Faux
A * NotA : Faux
A * C :  None
```

Création d'une classe simple : simpleBooleen.py

```
1  class simpleBoole:
2
3      Value = None                                  #valeur par défaut d'argument
4
5      ##Constructeur
6      def __init__(self, Val=None):
7          if (Val==None):
8              print("'Vrai' ou 'Faux'")           # On ne peut pas empêcher la
9              return                              # création d'une instance
10         else:                                   # de valeur None ...
11             self.Value=Val
12
13
14     ##afficheur sous forme textuelle
15     def __str__(self):
16         return str(self.Value)                  # str() nécessaire si None
17
18     ## opérations de base
19
20     #Opération "NOT"
21     def __neg__(self):                          # Attention à bien retourner
22                                                 # un objet simpleBoole
23         if (self.Value=="Vrai"):                # et pas simplement une
24             return simpleBoole("Faux")          # chaine "Vrai" ou "Faux"
25         elif (self.Value=="Faux"):
26             return simpleBoole("Vrai")
27
```

```
28
29      #Opération "OU"  => addition : +
30      def __add__(self, other):
31
32          if (self.Value=="Faux" and other.Value=="Faux"):
33              return simpleBoole("Faux")
34
35          elif (self.Value != None and other.Value != None): # il faut gérer le
36              return simpleBoole("Vrai")                     # cas None ...
37
38
39      #Opération "OU not"  => soustraction : -
40      def __sub__(self, other):
41                                                             # On préfèrera ré-utiliser
42          return self.__add__(other.__neg__())               # les fct. déjà réalisées.
43
44
45      #Opération "AND"  --> multiplication : x
46      def __mul__(self, other):
47
48          if (self.Value=="Vrai" and other.Value=="Vrai"):
49              return simpleBoole("Vrai")
50
51          elif (self.Value != None and other.Value != None): # il faut gérer le
52              return simpleBoole("Faux")                     # cas None ...
53
```

La class expComplexe :

Une implémentation des complexes en passant uniquement par module et argument.

```
class expComplexe(builtins.object)
 |  Methods defined here:
 |
 |      Imag(self)
 |      #partie imaginaire
 |
 |      Real(self)
 |      #partie réelle
 |
 |  __add__(self, other)
 |      #Addition de deux complexes
 |
 |  __init__(self, module, argument)
 |      #Constructeur
 |
 |  __mul__(self, other)
 |      #multiplication
 |
 |  __pow__(self, alpha)
 |      #puissance
 |
 |  __str__(self)
 |      #afficheur textuel => obtenu lors de l'appel par print
```

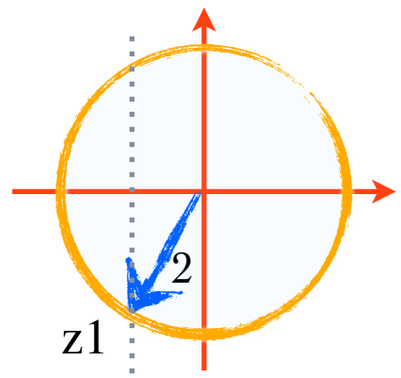
```
 |  __sub__(self, other)
 |      #soustraction
 |
 |  __truediv__(self, other)
 |      #division
 |
 |      conjugate(self)
 |      #Conjugué
```

Python 3

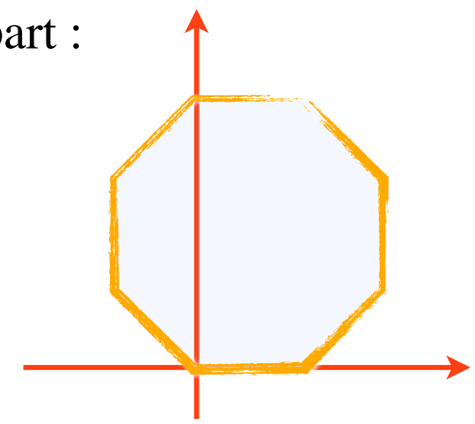
```
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |
----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  arg = None
```

```
zero = expComplexe(0,0)
un =expCompexe(1,0)
z1=expComplexe(2,-2*pi/3)
z2=expComplexe(2,0)

print("zero :", zero)
print("z1 : ",z1,"\nz2 : ", z2,"\n")
print("z1 + z2 : ", z1+z2)

print("z1_bar   : ", z1.conjugate())
print("  1/z1     : ", un/z1)
# "1" n'est pas de la classe mais
# "un" le représente !


print(" z1**2   : ",z1**2)
print("Real(z1) : ",z1.Real())

#parcours d'un octogone
z=zero
for i in range(8):
    z+=expComplexe(1,i*pi/4)
print(z)
```

```
zero : 0 exp(J 0 )
z1 :  2 exp(J -2.094395102393195 3 )
z2 :  2 exp(J 0 )
z1 + z2 :  2.0000000000000013 exp(J 5.235987755982989 )


z1_bar    :     2 exp(J 2.0943951023931953 )
1/z1      :  0.5 exp(J 2.0943951023931953 )


 z1**2    :     4 exp(J -4.1887902047863905 )
Real(z1) :     -0.9999999999999996
```




```
#on revient au point de départ :
0 exp(J 0 )
```



Création d'une classe simple : expComplex.py

```
6  class expComplexe:
7
8      from math import cos, sin, atan, pi, acos
9      mod = None
10     arg = None
11
12     #Constructeur
13     def __init__(self, module, argument):
14
15         self.mod = module
16         self.arg = argument
17
18     #afficheur textuel => obtenu lors de l'appel par print
19     def __str__(self):
20         monAffichage = str(self.mod)+" exp(J "+str(self.arg)+" )\n"
21         return monAffichage
22
```

```
    ##Opération entre les instances

    #self et other sont de la classe expComplexe

    #Conjugué
    def conjugate(self):
        return expComplexe(self.mod, -self.arg)

    #Addition de deux complexes
    def __add__(self, other):

        u = self.mod            ;  v = other.mod ;
        phi1 = self.arg%(2*pi) ;  phi2 = other.arg%(2*pi)

        if (u==0): return expComplexe(v, phi2)
        if (v==0): return expComplexe(u, phi1)
        if (u==v and (phi1-phi2)%(2*pi)-pi<1e-14): return expComplexe(0,0)

        theta = (pi - phi1 + phi2)%(2*pi)
        w = (u**2 + v**2 - 2*u*v*cos(theta))**0.5   #Th. Al'Kashi

        alpha=acos((u**2 + w**2 - v**2)/(2*u*w))

        if (0 < theta%(2*pi) < pi):
            Phi = phi1 - alpha
        else:
            Phi = phi1 + alpha

        return expComplexe(w, Phi)
```

```
54      #multiplication
55      def __mul__(self, other):
56          return expComplexe(self.mod * other.mod, self.arg + other.arg)
57
58      #soustraction
59      def __sub__(self, other):
60          return self.__add__(other.__mul__(expComplexe(1,pi)))
61
62      #division
63      def __truediv__(self, other):
64          return  expComplexe(self.mod / other.mod, self.arg - other.arg)
65
66      #puissance
67      def __pow__(self, alpha):
68          return expComplexe(self.mod**alpha, self.arg*alpha)
69
70
71      ##Fonctions de base
72
73      #partie réelle
74      def Real(self):
75          return self.mod*cos(self.arg)
76
77      #partie imaginaire
78      def Imag(self):
79          return self.mod*sin(self.arg)
80
```

Création d'une classe Liste : maListe.py

Les listes en Python sont optimisées [fonction de Hash & tableaux en C compilé] pour gagner en rapidité mais cela ne correspondent pas à la définition exacte d'une liste :

- Une liste est la donnée d'**un élément de départ** et du **nombre d'élément** au total.
- Chaque élément est une structure de données contenant :
  la **valeur de l'élément**, une méthode **.next()** qui désigne l'élément suivant

classe maListe :

Cette classe permet de créer des instances de liste au sein du module.
chaque instance de maListe contient :
- la **longueur totale** de la liste
- l'**élément de départ**
- une méthode **push** : ajoute un élément en fin de liste
- une méthode **pop** : supprime le dernier élément et renvoie sa valeur.

classe Element:

Cette classe permet de créer des instances d'élements au sein de la classe liste. Chaque instance Element contient :
- la **valeur** de l'élément que l'on souhaite garder en mémoire.
- une méthode **next :** qui permet de pointer l'élément suivant dans la liste.