Chapitre1
Programmation en Python

Dans ce premier chapitre nous révisons la syntaxe de programmation en langage Python, enseignée en première année. On pourra dans la suite s'y référer pour vérifier la syntaxe des principales commandes et fonctions. Nous concluons le chapitre par divers exercices et annales de concours.

Le cours en questions

1 Données, types, variables, expressions

1.1 Types de données

Toutes les données en mémoire sont constituées d'une suite de 0 et de 1 sur une ou plusieurs unités de stockage (usuellement 64 bits). Pour savoir comment les manipuler et leur appliquer des opérations et fonctions, l'interpréteur doit connaître leur type :

Définition. Une donnée est constituée d'un **type** et d'une valeur encodée par une suite de 0 et de 1 sur une ou plusieurs unités de stockage.

Pour manipuler des données, l'interpréteur mémorise l'emplacement de la donnée en mémoire et son type. C'est le type qui permettra la manipulation convenable de la donnée, et son affichage correct.

Le type d'une donnée peut s'obtenir grâce à la fonction **type** :

```
In [1]: type(1)
Out[1]: int

In [2]: type(1.)
Out[2]: float

In [3]: type(1j)
Out[3]: complex
```

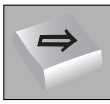
1.2 Données numériques

1.2.1 Types de données numériques

Les données numériques peuvent être de trois types :

- entiers (relatifs), (type **`int`**),
- nombres à virgule (flottante), (type **`float`**), et
- nombres complexes (type **`complex`**).

Type	Nombres	Exemples
`int`	Entiers relatifs	`0, 1, 2, ..., −1, −2, ...`
`float`	Nombres à virgule	`0.1, 1.25, −2.3, 1e10 ...`
`complex`	Nombres complexes	`1j, 1+2j, 2+1.5j, ...`



On peut écrire un flottant sous format scientifique, par exemple `1.5e10` ou `1.5E10` pour $1.5 \cdot 10^{10}$.

– Le type entier **`int`** est encodé en mémoire au format binaire signé. Il n'y a pas de limitation en Python pour le stockage des entiers (l'interpréteur augmente la taille de stockage pour éviter les erreurs de dépassement).
– Le type flottant **`float`** est encodé en mémoire au format nombre à virgule flottante en double précision (stockage sur 64 bits) ; ce format limite la représentation des flottants, et peut occasionner des erreurs de dépassement de capacité.
– Le type complexe **`complex`** est encodé en mémoire sous forme de deux flottants, sa partie réelle et sa partie imaginaire.
Par exemple :

• La donnée 1, de type entier, sera encodée sur 64 bits :

$$\underbrace{00000000\ 00000000\ \cdots\ 00000001}_{64\text{ bits}}$$

• La donnée `1.0`, de type flottant, sera encodée sur 64 bits :

$$\underbrace{0\ \overbrace{01111111}^{\text{exposant}}\ \overbrace{0000000\ \cdots\ 00000000}^{\text{mantisse}}}_{64\text{ bits}}$$

Le type flottant est défini par la présence d'un point décimal ou d'une notation scientifique. Par exemple 1 est entier, mais `1.0` , `1.` ou `1e0` est flottant. Les zéros après le séparateur décimal ou en préfixe pourront être omis (`.1` désigne le nombre 0,1).
On renvoie au chapitre 2 de l'ouvrage « Informatique 1ère année » du même auteur chez le même éditeur pour un rappel sur la représentation des nombres en mémoire.

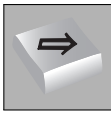
Une erreur courante consiste à se tromper de séparateur décimal.

Le séparateur décimal n'est pas la virgule mais le point.

1.2.2 Opérateurs arithmétiques

Les opérations arithmétiques, addition, soustraction, multiplication, puissance, division, s'obtiennent à l'aide des opérateurs arithmétiques :

Opérateurs arithmétiques	
+	Addition
−	Soustraction et opposé
*	Multiplication
/	Division
**	Puissance



L'élévation à la puissance est obtenue par l'opérateur `**`, et non pas `^`. L'exposant peut être un nombre à virgule. Par exemple une approximation décimale de $\sqrt{2}$ s'obtient par l'expression `2**0.5`.

Pour deux données de type entiers `a` et `b`, les quotient et reste dans la division euclidienne s'obtiennent par : :

Quotient et reste dans la division euclidienne	
`a // b`	Quotient dans la division euclidienne de `a` par `b`
`a % b`	Reste dans la division euclidienne de `a` par `b`

Où `a` et `b` peuvent être négatif. Dans ce cas `a // b` respecte les régles de signe d'un quotient, et `a % b` est toujours situé entre 0 inclus et `b` exclu.

Une valeur nulle pour `b` renverra une erreur de division par zéro :

```
In [1]: 13 // 5
Out[1]: 2

In [2]: 13 % 5
Out[2]: 3

In [3]: 13 % 0
...
ZeroDivisionError: integer division or modulo by zero
```

L'opérateur de reste euclidien `%` peut aussi être employé avec en opérandes des flottants, dans quel cas il correspond au modulo.

1.2.3 Valeur absolue et module

La fonction **`abs`** retourne :

- avec en paramètre un entier ou flottant : sa valeur absolue,
- avec en paramètre un complexe : son module.

```
In [1]: abs(-1.5)
Out[1]: 1.5

In [2]: abs(1-1j)
Out[2]: 1.4142135623730951
```

1.2.4 Fonctions mathématiques

Pour utiliser des fonctions et constantes mathématiques évolués il faudra les importer d'un module, du module `math` par exemple, par l'instruction :

```
from math import nom_de_la_fonction_ou_constante
```

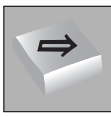
Voici quelques fonctions et constantes du modue `math` :

Fonctions et constantes du module `math`	
constantes (approximations)	
`pi`	le nombre π
`e`	le nombre $e = \exp(1)$
`inf`	l'infini $+\infty$. C'est la valeur +INF des flottants
fonctions mathématiques	
`fsum(L)`	retourne la somme d'une séquence numérique `L` de flottants. Meilleure précision que la fonction `sum()`
`gcd(a,b)`	le pgcd de `a` et `b`, deux entiers non tous nuls.
`factorial(n)`	fonction factorielle : $n \mapsto n!$. l'argument doit être un entier positif.
`floor(x)`	fonction partie entière : $x \mapsto \lfloor x \rfloor$.
`ceil(x)`	partie entière par valeur supérieure : $x \mapsto \lceil x \rceil$ c'est à dire le plus petit entier supérieur ou égal à x.
`trunc(x)`	troncature entière ; retourne l'entier obtenu en supprimant les décimales après la virgule.
`fabs(x)`	fonction valeur absolue $x \mapsto \lvert x \rvert$
`sqrt(x)`	fonction racine carrée $x \mapsto \sqrt{x}$; l'argument doit être un nombre positif
`pow(x,y)`	fonction puissance : $x \mapsto x^y$. Contrairement à l'opérateur `**` retourne toujours un flottant. Lorsqu'`y` n'est pas entier, `x` doit être > 0.
`exp(x)`	fonction exponentielle : $x \mapsto \exp(x)$.
`log(x)`	fonction logarithme néperien : $x \mapsto \ln(x)$. L'argument `x` doit être > 0.
`log(x,a)`	fonction logarithme en base `a` : $x \mapsto \log_a(x) = \dfrac{\ln(x)}{\ln(a)}$; `x` et `a` doivent être > 0 et `a` différent de 1.
`log2(x)`	fonction logarithme en base 2 ; `x` doit être > 0. Meilleure précision que `log(x,2)`.

`log10(x)`	fonction logarithme en base 10 ; `x` doit être > 0. Meilleure précision que `log(x,10)`.
`frep(x)`	retourne un couple constitué de la mantisse et de l'exposant dans la représentation en flottant de `x`, c'est à dire (m,e) tel que $x = m.2^e$ ($m \in [1,2[$ et lorsque $x = 0$ retourne $(0.0,0)$).
`cos(x)`	fonction cos ; pour `x` exprimé en radians.
`sin(x)`	fonction sin ; pour `x` exprimé en radians.
`tan(x)`	fonction tan ; pour `x` exprimé en radians.
`acos(x)`	fonction arccos ; retourne un angle en radians ; `x` doit être compris entre -1 et 1.
`asin(x)`	fonction arcsin ; retourne un angle en radians ; `x` doit être compris entre -1 et 1.
`atan(x)`	fonction arctan ; retourne un angle en radians.

Exemple.

```
In [3]: from math import cos, pi
In [4]: cos(pi/4)
Out[4]: 0.7071067811865476
```



Les fonctions et constantes mathématiques évoluées doivent être importés d'un module. On peut importer en une seule instruction plusieurs fonctions et constantes en séparant leur nom d'une virgule.

1.2.5 Nombres complexes

Partie réelle, partie imaginaire, conjuguée et modules d'un nombre complexe s'obtiennent grâce aux attributs `real`, `imag`, et à la méthode `conjugate()` de la classe complexe, et à la fonction **`abs`**. Les parties réelles et complexes obtenus sont des flottants.

Nombres complexes	
`x`	Une donnée de type complexe
`x.real`	Sa partie réelle
`x.imag`	Sa partie imaginaire
`x.conjugate()`	Son conjugué
`abs(x)`	Son module

```
In [1]: (1+2j).real
Out[1]: 1.0

In [2]: (1+2j).imag
Out[2]: 2.0

In [3]: (1+2j).conjugate()
Out[3]: (1-2j)
```

```
In [4]: abs(1+2j)
Out[4]: 2.23606797749979
```

1.3 Variables, expressions, affectation

1.3.1 Notion de variable

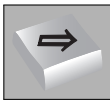
La notion de **variable** est essentielle en programmation. Elle permet de stocker en mémoire des valeurs, et de les utiliser et modifier à volonté au sein d'un programme. La valeur d'une variable évolue au cours de l'exécution d'un programme

Une **variable** a :

- un **identifiant** : c'est son nom, qui permet de manipuler la variable au sein d'un programme ou d'une instruction (en mode console). C'est une chaine de caractère alphanumérique, qui peut être composée de lettres de chiffres et du symbôle '_', qui ne doit pas débuter par un chiffre, ni être un mot clef réservé.
- Une **valeur**, c'est son contenu. Elle est dotée d'un type et est stockée dans la mémoire sous forme d'une écriture binaire.

Les mots-clefs réservés du langage sont :

`False`	`def`	`if`	`raise`
`None`	`del`	`import`	`return`
`True`	`elif`	`in`	`try`
`and`	`else`	`is`	`while`
`as`	`except`	`lambda`	`with`
`assert`	`finally`	`nonlocal`	`yield`
`break`	`for`	`not`	
`class`	`from`	`or`	
`continue`	`global`	`pass`	



Une variable a un identifiant (un nom) et une valeur (donnée). Elle permet de stocker en mémoire des valeurs et d'y accéder par son identifiant.

1.3.2 Expression

Définition.

Une **expression** est une formule bien formée construite à partir d'opérateurs, de parenthèses, de variables et de données. Elle est définit formellement par induction :

- Une variable ou une donnée est une expression, dont le type est celui de la valeur de la variable, ou de la donnée. Une donnée peut être obtenue à partir d'une fonction prédéfinie, une fonction mathématique par exemple.
- si `e` est une expression alors `(E)` est une expression de même type,
- si $\top$ est un opérateur prenant deux opérandes, et si `e` et `e'` sont deux expressions

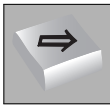
dont les types sont compatibles avec $\top$, alors `e`$\top$`e'` est une expression du type retourné par $\top$.

– si $\bot$ est un opérateur prenant un opérande, et si `e` est une expression dont le type est compatible avec $\bot$, alors $\bot$`e` est une expression du type retourné par $\bot$.

Une expression s'évalue en une valeur, du type de l'expression.

Exemples :

• `2*(-1+3**2)` est une expression dont la valeur est `16` et de type entier.
• Si `a` est un variable de valeur -1, alors `(3-1)**a` est une expression de type flottant dont la valeur est `0.5`.
• `sqrt(2)` est une expression de type flottant : c'est une donnée retournée par la fonction `sqrt`.



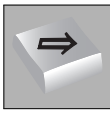
Une expression est une formule utilisant des données, des variables, des parenthèses et des opérations. Une expression s'évalue en une valeur, qui dépend des valeurs des variables au moment de l'évaluation.

1.3.3 Affectation et déclaration de variable

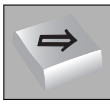
L'opération fondamentale pour une variable est l'**affectation** représentée par le symbôle `=`. Une instruction d'affectation est de la forme :

variable = expression

Lors d'une affectation l'expression à droite du symbôle `=` est évaluée, avant d'être affecté à la variable dont le nom figure à gauche du symbôle `=`, c'est à dire devenir sa nouvelle valeur.



Une affectation permet de définir ou de modifier la valeur d'une variable. En langage algorithmique on la symbolise par $variable \leftarrow expression$, en Python on la note `variable = expression`.

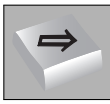


Le membre de gauche d'une affectation `=` ne peut être que le nom d'une variable.

Si la variable n'existe pas encore elle sera créée lors de l'affectation. On parle de déclaration de la variable :

En python la **déclaration d'une variable** (sa définition) se fait à l'aide d'une instruction d'affectation :

```
variable = expression
```



La déclaration de variable permet de définir une variable. Elle se fait à l'aide d'une affectation.

Un même nom de variable peut figurer des 2 côtés de `=`, lorsque l'ancienne valeur de la variable est utilisée pour calculer sa nouvelle valeur. Par exemple l'instruction `x = x + 1` incrémente la valeur de la variable `x`, c'est à dire ajoute 1 à sa valeur précédente pour obtenir sa nouvelle valeur.

L'affectation `x = x + 1` n'a rien à voir avec l'équation mathématiques (impossible) $x = x + 1$. Dans le dernier cas, l'équation signifie : déterminer tous les éléments x vérifiant l'égalité ; dans le premier, elle signifie : évaluer `x+1` puis stocker sa valeur dans la variable `x`.

Exemple. Déclaration et modification d'une variable.

```
In [1]: nbr = 7 # Déclaration d'une variable nbr de valeur 7
In [2]: nbr = 2 * nbr + 1
In [3]: nbr
Out[3]: 15
In [4]: nbr = nbr ** 2
In [5]: nbr
Out[5]: 225
```

1.3.4 Variantes utiles de l'affectation

• Une instruction de la forme :

```
variable †= expression
```

où † désigne n'importe quelle opérateur arithmétique : `+, -, *, /, //, %, **, etc....` produit le même résultat que l'instruction :

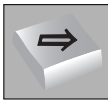
```
variable = variable † expression
```

L'instruction `x += expr` a même effet que `x = x + expr`.
L'instruction `x −= expr` a même effet que `x = x − expr`.
L'instruction `x *= expr` a même effet que `x = x * expr`.
etc...

L'instruction `variable †= expression` a un coût d'exécution moindre que son analogue `variable = variable † expression`.

Exemple. Échange du contenu de deux variables numériques.

```
In [1]: a = 1 ; b = 2
In [2]: a = a+b ; b = a−b ; a = a−b
In [3]: a
Out[3]: 2
In [4]: b
Out[4]: 1
```



Plusieurs instructions sur une même ligne sont séparés d'un point virgule `;`. Elles sont exécutées de gauche à droite.

• On peut affecter en une seule instruction, la même valeur à plusieurs variables :

```
variable_1 = variable_2 = ... = variable_k = expression
```

est équivalente à la suite d'affectation :

```
variable_1 = expression ; ... ; variable_k = expression
```

.

• **Affectation multiple.**

Python permet en une seule instruction d'affectation ('=') d'affecter plusieurs variables, selon le schéma :

```
var_1, var_2, ..., var_k = exp_1, exp_2, ..., exp_k
```

Exemple.

```
In [1]: a, b = 1, 2
In [2]: a
Out[2]: 1
In [3]: b
Out[3]: 2
```

Les variables, à gauche de l'instruction '=' d'affectation, sont séparées par des virgules, de même que les valeurs, à droite de '='. Elles doivent être en même nombre.

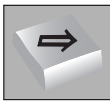
Une affectation multiple n'est **pas** équivalente à la suite d'affectations :

```
var_1=exp_1 ; var_2=exp_2 ; ... ; var_k = exp_k
```

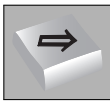
En effet, les expressions à droite sont évaluées **avant** d'être affectées aux variables à gauche.

Exemple. L'instruction `a,b = b,a` échange les valeurs de 2 variables `a` et `b`.

L'effet n'est pas du tout le même que l'instruction : `a = b = a`. C'est l'usage le plus pratique de l'affectation multiple.



L'affectation multiple `a, b = b, a` permet d'échanger les valeurs des variables `a` et `b`.



Bien noter que lors d'une affectation multiple toutes les valeurs des expressions (à droite du =) sont celles **avant** l'appel de l'instruction.

1.4 Le type booléen

Les booléens (type **`bool`**) : ils ne peuvent prendre que deux valeurs `True` ou `False` :

```
In [1]: type(True)
Out[1]: bool

 In [2]: type(False)
 Out[2]: bool
```

Ils correspondent intuitivement aux propositions "Vrai" et "Faux" et sont incontournables.

1.4.1 Opérateurs de comparaison

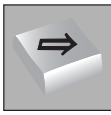
On peut obtenir une expression booléenne grâce aux opérateurs de comparaison.

• Les opérateurs suivants ne s'appliquent qu'avec opérandes des entiers (**`int`**) ou des flottants (**`float`**) :

Opérateurs de comparaison	
`a < b`	`a` a une valeur strictement inférieure à celle de `b`
`a > b`	`a` a une valeur strictement supérieure à celle de `b`
`a <= b`	`a` a une valeur inférieure ou égale à celle de `b`
`a >= b`	`a` a une valeur inférieure ou égale à celle de `b`

• Les opérateurs suivants s'appliquent avec des opérandes de tout type :

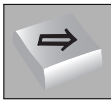
Opérateurs de comparaison	
`a == b`	`a` et `b` ont même valeur.
`a != b`	`a` et `b` ont des valeurs différentes.



La comparaison `a == b` retourne `True` si et seulement si `a` et `b` ont même valeur ; `a != b` retourne `True` si et seulement si `a` et `b` ont des valeurs différentes. Les expressions `a` et `b` peuvent être de type différents.

Par exemple lorsque l'on saisit les conditions suivantes les valeurs retournées peuvent prendre 2 valeurs de type booléen (**`bool`**) : `True` ou `False`.

```
In [1]: 1. < 2
Out[1]: True

In [2]: 1 < 2 < 3
Out[2]: True

In [3]: 2 == 2.0
Out[3]: True
```



L'interpréteur Python comprend les conditions de la forme `a < b < c`, etc...

il faut prendre bien garde à ne pas confondre l'affectation = avec la condition d'égalité ==.

1.4.2 Connecteurs logiques

On peut effectuer des opérations logiques sur les booléens (par ordre de priorité) :

`or`, **`and`**, **`not`**`()`

ils sont définis à l'aide des tables de vérité :

• Table de vérité du "ou logique" (**or**) :

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

• Table de vérité du "et logique" (**and**) :

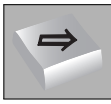
a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

• Table de vérité du "non logique" (**not**) :

a	not(a)
True	False
False	true

Exemple.

```
In [1]: a = (1.0 <= 2) ; b = (a == False)

In [2]: a or b
True

In [3]: not(a) and not(b)
False
```



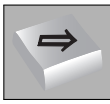
Les connecteurs logiques sont des opérateurs booléens ; ils prennent en opérandes des booléens et valent un booléen.

• Il est important de noter que Python effectue une **évaluation paresseuse** des connecteurs logiques, c'est-à-dire que l'évaluation des opérandes se fait de gauche à droite, et s'interrompt, dès que le résultat est assuré. Par exemple, l'évaluation de `(2<1) and (2<3)` s'interrompt après l'évaluation (=`False`) de `(2<1)`, et vaudra `False`.

• Le ou exclusif (xor) est important en informatique. Il est défini à partir de la table de vérité :

a	b	a xor b
True	True	False
True	False	True
False	True	True
False	False	False

il est équivalent à : `(a or b) and not(a and b)`
ou encore à : `(not(a) and b) or (a and not(b))`.



En Python, on peut obtenir le ou exclusif à l'aide de l'opérateur `^`.

1.5 Autres principaux types de données

1.5.1 Le type `Nonetype`

Le type `Nonetype` : il ne peut prendre qu'une seule valeur : `None`.

```
In [1]: type(None)
Out[1]: Nonetype
```

Il nous sera d'usage moins courant. Il correspond intuitivement à "Rien", et permet de formaliser lorsqu'une fonction ne renvoie rien à son appel.

1.5.2 Le type Liste (`list`)

En `python` une liste (**`list`**) est une donnée qui permet de collecter des données de type quelconque. On peut déclarer une liste **par extension**, c'est à dire en fournissant ses éléments, entre crochets et séparés par des virgules.

Exemple. Déclaration d'une liste `L` constituée de 5 éléments.

```
In [1]: L = [1, 2, 3, 4.0, True]
```

Les 3 premiers sont des entiers, le quatrième un flottant, et le dernier un booléen.

Le nombre d'éléments d'une liste s'obtient grâce à la fonction **`len`** :

```
In [2]: len(L)
Out[2]: 5
```

Les différentes données dans une liste sont repérées par leur **indice**, c'est un entier qui donne leur position dans la liste. Le premier élément a pour indice 0. Le dernier élément de la liste `L` a donc pour indice **`len`**`(L)`-1.

On accède aux données d'une liste par son nom, suivi de l'indice de la donnée entre crochets :

```
In [3]: L[0]
Out[3]: 1

In[4]: L[4]
Out[4]: True
```

On peut fournir comme indice un nombre entier négatif ; dans cas l'interpréteur lui ajoute **`len`**`(liste)`. Ainsi pour accéder au dernier élément d'une liste on peut fournir comme indice -1.

```
In [5]: L[-1]
Out[5]: True
```

Pour une liste constituée de `n` éléments les indices passés entre crochets peuvent valoir tout entier compris entre $-n$ et $n-1$. Au-delà on obtiendra l'erreur :
`IndexError : list index out of range`.

On peut modifier un élément d'une liste en y accédant pas son indice et en lui affectant une nouvelle valeur.

```
In [6]: L[4] = False ; L
Out[6]: [1, 2, 3, 4.0, False]
```

Le mot-clef **`in`** est un opérateur qui renvoie un booléen ; il permet de déterminer si une valeur apparaît ou non dans une liste.

```
In [1]: liste = [1, -3, 5, 17.0, 2]
In [2]: 1 in liste
Out[2]: True

In [3]: -1 in liste
Out[3]: False

In [4]: -3.0 in liste
Out[4]: True
```

C'est l'égalité == qui est recherchée ; c'est pourquoi `-3.0 in liste` renvoie `True` alors que le flottant `-3.0` ne figure pas dans la liste tandis que l'entier `-3` y figure.

• Extraction de liste.

On peut extraire une copie d'une sous-liste d'une liste à l'aide de la technique du slicing. La syntaxe est :

LISTE[Indice départ (inclus) : Indice arrivée (exclus) : Pas]

Une nouvelle liste est créée, contenant les copies des éléments de `LISTE` d'indices allant de `Indice de départ` (inclu) jusqu'à `Indice d'arrivée` (exclu) par pas de `Pas`. Les 3 arguments doivent être de type entier (**`int`**) et sont optionnels : par défaut le pas vaut 1, et les indices de départ et d'arrivée valent par défaut `0` et **`len`**`(LISTE)` si `pas > 0` et l'inverse si `pas < 0`.

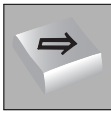
• Exemple.

```
In [1]: L = [0., 1., 2., 3., 4., 5., 6., 7., 8.]
In [2]: L1 = L[2:5]      # tranche des indices 2 <= . < 5
In [3]: L1
Out[3]: [2., 3., 4.]

In [4]: L[::2]
Out[4]: [0., 2., 4., 6., 8.]
```

```
In [5]: L[6:2:-1]        # tranche des indices 6 >= . > 2
Out[5]: [6., 5., 4., 3.]
```

Les indices entre crochets peuvent sortir de la plage d'indice de la liste :

```
In [4]: L2 = L[1:100]    # tranche des indices  1 <= . < 100
In [5]: L2
Out[5]: [1., 2., 3., 4., 5., 6., 7., 8.]
```



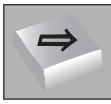
`L[i:j:k]` extrait une liste contenant des copies des éléments de `L` allant de l'indice `i` inclu à l'indice `j` exclu, par pas de `k`. Les indices sont optionnels et munis de valeurs par défaut.

En particulier l'instruction suivante :

```
L2 = L[:]
```

crée en mémoire une liste `L2` dont tous les éléments sont des **copies** des éléments de `liste`. C'est très utile, `L2 = L` ne crée pas une nouvelle liste, mais seulement un nouveau nom associé à une seule liste en mémoire.

```
In [1]: L = [1, 2, 3]
In [2]: L1 = L ; L2 = L[:]
In [3]: L[0] = 0  # Modification de L

In [4]: L
Out[4]: [0, 2, 3]

In [5]: L1        # L1 a aussi été modifiée
Out[5]: [0, 2, 3]

In [6]: L2        # L2 n'a pas été modifiée
Out[6]: [1, 2, 3]
```



`L1 = L[:]` crée une copie de la liste `L`, tandis que `L1 = L` ne fait que créer un nouveau nom associé à la liste `L`.

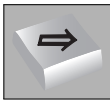
• Méthodes des listes.

Voici un tableau de méthodes disponibles pour les listes. L'exemple qui suit ci-après montre leur usage.

Ajout/retrait en fin de liste	
`L.append(x)`	Ajoute `x` à la fin de la liste `L`
`L.pop()`	Retire et renvoie l'élément en fin de `liste`
`L.extend(L2)`	Ajouter les éléments de `L2` à la suite de `L`

Exemple.

```
In [1]: L = []  # L est la liste vide
In [2]: L.append(1)      # Ajout de 1
In [3]: L.append(2)      # Ajout de 2
In [4]: L
Out[4]: [1, 2]
In [5]: L.pop()
Out[5]: 2
In [6]: L
Out[6]: [1]
```



Les méthodes `append()` et `pop()` des listes sont incontournables.

Voici d'autres méthodes des listes. Leur usage est généralement à proscrire ; elles cachent des opérations coûteuses pour le microprocesseur et qu'il faut savoir programmer.

Ajout/retrait d'un élément via son indice :	
`L.insert(i,x)`	Insère l'élément `x` en position `i` dans `L`
`L.pop(i)`	Retire et renvoie l'élément en position `i` dans `L`
Accès à un élément via sa valeur :	
`L.remove(x)`	Retire la première occurence de `x` dans `L`
`L.index(x)`	Renvoie la 1^{ere} position de `x` dans `L`. Message d'erreur si aucune.
`L.count(x)`	Renvoie le nombre d'occurrences de `x` dans `L`.
Tri et renversement	
`L.sort()`	Trie la liste par ordre croissant.
`L.reverse()`	Renverse l'ordre des éléments de la liste.

Noter que :

- `L.append(x)` est équivalent à `L += [x]`
- `L.extend(L2)` est équivalent à `L += L2`
- La suppression de l'élément d'indice `i` s'obtient aussi par : **`del`** `L[i]`.

• Définition de liste par compréhension.

On peut déclarer une liste à l'aide des mots-clés **`for`**, **`in`** et **`if`** comme on définit un ensemble en mathématiques **par compréhension** (on dit aussi **par intension**) :

`[f(x) for x in liste]` correspond à $\{f(x) \mid x \in \texttt{liste}\}$

A ceci près que les élément d'une liste sont ordonnés, et qu'il peut y avoir des répétitions.

L'expression retourne la liste des données `f(x)`, pour une fonction `f`, lorsque `x` parcourt une `liste`, séquence, ou un itérateur (**`range`**).

Exemple.

```
In [1]: L = [x**2 for x in range(10)]
In [2]: L
Out[2]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

L'instruction **for x in range**(n) fait parcourir à la variable x tous les entiers allant de 0 à n−1.
L'instruction **for x in range**(m,n) fait parcourir à la variable x tous les entiers allant de m à n−1. Voir paragraphe §3.2.2.

De même :

```
[f(x) for x in liste if Condition(x)]
```

correspond à :

$$\{f(x) \quad | \quad x \in \texttt{liste} \text{ et Condition}(x)\}$$

Exemple. Liste des multiplies de 12 entre 0 et 100 :

```
In [1]: L = [x for x in range(101) if x % 12 == 0]
In [2]: L
Out[2]: [0, 12, 24, 36, 48, 60, 72, 84, 96]
```

Liste des diviseurs d'un entier naturel N :

```
In [1]: N = 60
In [2]: [x for x in range(1,N+1) if (N % x == 0)]
Out[2]: [1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60]
```

Dans une compréhension de liste on peut utiliser autant de variables et de **for** que souhaité (toutes les variables doivent être déclarées).
Par exemple avec deux variables :

```
In [1]: [(i,j) for i in range(3) for j in range(3)]
Out[1]:
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0),
 (2, 1), (2, 2)]
```

Et ainsi de suite.

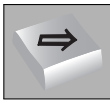
1.5.3 Le type chaîne de caractère (str)

Les données de type chaînes de caractères (type **str**), consistent en une suite de caractères alphanumériques. On les déclare en les délimitant par deux apostrophes '...', ou deux double-quotes "...", voire parfois par deux triples doubles-quotes """ ... """. Les deux premières possibilités de délimitations permettent de faire figurer dans une chaîne de caractère les caractères apostrophe ou double quotes :

```
In [1]: "Bonjour l'univers"
Out[1]: "Bonjour l'univers"
```

```
In [2]: 'Un double quote : "'
Out[2]: 'Un double quote : "'

In [3]: type('chaine de caractère')
Out[3]: str

In [4]: type("")        # La chaîne vide
Out[4]: str
```

La dernière possibilité """ ... """ autorise les sauts à la lignes et tabulations.



Le caractère # permet d'insérer un commentaire : tout ce qui le suit sur la ligne sera ignoré par l'interpréteur. On l'utilise pour commenter le code.

• **Méthodes des chaînes de caractères.** Voici quelques méthodes qui s'appliquent aux chaînes de caractères. Les exemples ci-après montrent comment les utiliser.

Méthodes des chaînes de caractères	
`upper()`	Convertit (les lettres de) la chaîne en majuscules
`lower()`	Convertit (les lettres de) la chaîne en minuscules
`split()`	Prend en argument une chaîne de caractère, et retourne la liste des sous-chaînes délimitées par le paramètre. Lorsqu'il est absent, retourne la liste des sous-chaînes délimitées par des espaces.
`rstrip()`	Prend en argument une chaine de caractère et retourne la chaine obtenue en supprimant à sa fin tous les caractères apparaissant dans l'argument. Sans paramètre, supprime les espaces en fin de chaine.
`eval()`	Prend en paramètre une chaine de caractère et l'évalue comme une expression ou l'exécute comme une commande.

Exemples.

```
In [1]: "-Bonjour le monde-".upper()
Out[1]: '-BONJOUR LE MONDE-'

In [2]: "-Bonjour le monde-".lower()
Out[2]: '-bonjour le monde-'

In [3]: "Bonjour le    monde".split()
Out[3]: ['Bonjour', 'le', 'monde']

In [4]: "Bonjour le   monde".split('o')
Out[4]: ['B', 'nj', 'ur le   m', 'nde']

In [5]: "Bonjour le monde   ".rstrip()
```

```
Out[5]: 'Bonjour le monde'

In [6]: "Bonjour le monde".rstrip('ed')
Out[6]: 'Bonjour le mon'

In [7]: eval("[1,2,3]")
Out[7]: [1, 2, 3]

In [8]: eval('1+1')
Out[8]: 2
```

1.5.4 Le type tuple (`tuple`)

En français `t-uplet`. Ce sont des objets séquentiels.

On les déclare par la liste de leurs éléments donnés entre parenthèses `(.)` et séparés par des virgules. À la définition les parenthèses sont optionnelles, ainsi lorsque l'on saisit plusieurs données séparées par des virgules, c'est un **`tuple`** qui est constitué.

```
In [1]: a = 2
In [2]: 2*a, 3*a, 4*a    # retourne un t-uplet
Out[2]: (4, 6, 8)
```

Tout comme pour les listes, l'accès à un élément se fait via son indice, et l'on peut extraire un sous-**`tuple`** par slicing.

```
In [3]: seq = (0,1,2,3)
In [4]: seq[0], seq[-1]
Out[4]: (0,3)

In [5]: seq[::-1]
Out[5]: (3,2,1,0)

In [6]: seq[0] = 1
...
TypeError: 'tuple' object does not support item assignment
```

Les t-uplets diffèrent des listes surtout en ce qu'ils sont **non-modifiables** : `seq[0] = 1` produit une erreur `TypeError`.

Les **`tuple`** sont non modifiables. Une fois déclaré on ne peut plus changer l'élément d'un **`tuple`**.

L'usage des tuple est préférable aux listes lorsque on souhaite stocker des t-uplets de valeurs dont on n'aura pas à modifier les valeurs.

1.5.5 Les types séquentiels (`list`, `str`, `tuple`)

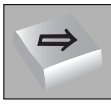
Les types **`int`**, **`float`**, **`bool`** sont des *types scalaires*.

Les types **`list`** (listes), **`str`** (chaînes de caractère) et **`tuple`** (*t*-uplets) sont des *types séquentiels*. Plus généralement, on appelle type séquentiel, un type de donnée conteneur dont les éléments sont repérés par un indice. Elles partagent des opérations communes.

• Opérations communes aux types séquentiels

Tous les objets de type séquentiel ont en commun les opérations suivantes. On considère `s` et `t` des données de même type séquentiel, et `i`, `j`, `k`, `n` des entiers.

Opération	Résultat
`s[i]`	élément d'indice `i` de `s`
`s[i:j]`	Tranche de `i` (inclus) à `j` (exclus)
`s[i:j:k]`	Tranche de `i` à `j` par pas de `k`
`len(s)`	Longueur de `s`
`sum(s)`	Somme des éléments de `s` pour `s` une séquence de nombres
`max(s), min(s)`	Plus grand et plus petit élément de `s` (pour des éléments ordonnables)
`x in s`	`True` si `x` est dans `s`, `False` sinon
`x not in s`	`True` si `x` n'est pas dans `s`, `False` sinon
`s + t`	Concaténation de `s` et `t`
`s * n, n * s`	Concaténation de `n` copies de `s` (duplication)
`s.index(x)`	Indice de la 1ère occurrence de `x` dans `s`
`s.count(x)`	Nombre d'occurrences de `x` dans `s`



Les opérateurs `s += t` et `s *= n` sont équivalents respectivement à : `s = s + t`, et à : `s = s * n`.

Exemples. Concaténation et duplication.

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

In [2]: 'aaa' + 'bbb'
Out[2]: 'aaabbb'

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

Exemple avec une chaîne de caractère :

```
In [1]: chaine = 'Andrea'
In [2]: len(chaine)
Out[2]: 6

In [3]: chaine[0], chaine[-2]
```

```
Out[3]: ('A', 'e')

In [4] chaine[::-1]
Out[4]: 'aerdnA'

In [5]: chaine.count('a')
Out[5]: 1

In [6]: chaine * 3
Out[6]: 'AndreaAndreaAndrea'
In [7]: max(chaine)
Out[7]: 'r'
```

On remarque au passage que **les chaines de caractères sont ordonnables**, selon un ordre lexicographique (alphabétique pour les lettres) déterminé par la position dans la table de caractère. On remarque que pour cet ordre les lettres minuscules sont supérieures aux lettres majuscules.

Tout comme les **tuple**, les chaînes de caractère ne sont pas modifiables : changer un élément produit une erreur `TypeError` :

```
In [8]: chaine[0] = 'Z'
...
TypeError: 'str' object does not support item assignment
```

1.5.6 Autres types de conteneurs non séquentiels (hors programme)

Les conteneurs de cette section sont hors-programme en CPGE. Ils pourront cependant s'avérer utiles. Il s'agit des ensembles (**set**) et des dictionnaires (**dict**).

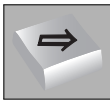
1.5.6.a) Les ensembles (set)

Un ensemble (**set**) est un conteneur non-ordonné et modifiable d'éléments de types quelconque.

• Déclaration d'un ensemble

On peut créer un ensemble par extension, ou par intension (compréhension) :

```
In [1]: X = {1, 2, 3}    # par extension
In [2]: X
Out[2]: {1, 2, 3}

In [3]: Y = { x for x in range(20) if x%3==0 } # par intension
In [4]: Y
Out[4]: {0, 3, 6, 9, 12, 15, 18}

In [5]: type(Y)
Out[5]: set
```



Le type **set** correspond à des conteneurs semblables à des ensembles. On peut déclarer un **set** par extension ou par intension.

• Opérations sur les ensembles

La fonction **len** retourne le nombre d'éléments d'un ensemble ; la fonction **sum** retourne la somme des éléments d'un ensemble de nombres.

```
In [6]: len(Y)
Out[6]: 7

In [7]: sum(Y)
Out[7]: 63
```

L'opérateur d'appartenance **in** fonctionne avec les ensembles :

```
In [8]: 8 in Y
Out[8]: False

In [9]: 9 in Y
Out[9]: True
```

• Les opérateurs ensemblistes sont la réunion, intersection, et différence.

opérateurs enemblistes	
|	Réunion
&	Intersection
−	Différence

```
In [10]: X|Y     # réunion
Out[10]: {1, 2, 3, 9, 9, 12, 15, 18}

In [11]: X&Y     # intersection
Out[11]: {3}

In [12]: X-Y     # différence
Out[12]: {1, 2}
```

• Les principales méthodes des ensembles sont :

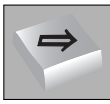
Méthodes des ensembles (set)	
`add(x)`	Ajoute l'élément `x` s'il n'est pas deja présent
`discard(x)`	Retire l'élément `x` s'il est présent
`clear()`	Vide l'ensemble de tous ses éléments

```
In [13]: X
Out[13]: {1,2,3}

In [14]: X.add(1)
In [15]: X
Out[15]: {1,2,3}

In [16]: X.discard(2)
In [17]: X
Out[17]: {1,3}

In [18]: X.clear()
In [19]: X
Out[19]: set()
```

1.5.6.b) Les dictionnaires

Ce sont des structures de données (**dict**) modifiables et non séquentielles ; un élément n'est pas repéré à l'aide d'un indice mais à l'aide d'un nom (sa **clef**). Un élément est constitué d'un **champ** : la donnée d'une clef et d'une valeur.

- la clef est un nombre une chaine ou un t-uplet.
- La valeur peut être de type quelconque,



Le type **dict** permet la définition de conteneurs dont les valeurs sont repérées non plus par des indices mais par des clefs.

1.5.7 Déclaration d'un dictionnaire

Un dictionnaire se définit par extension, par la suite des champs (la clef suivie de la valeur, séparés de :) entre accolades {.}.

• Déclaration

```
In [1]: Dic = { 'A' : 0, 'B' : 1, 'C' : 2, 'D' : 3 }
```

Ce qui est identique à :

```
In [1]: Dic = {}
In [2]: Dic['A'] = 0
In [3]: Dic['B'] = 1
In [4]: Dic['C'] = 2
In [5]: Dic['D'] = 3
```

• Opérations sur les dictionnaires

L'accès à un élément se fait à l'aide de sa clef :

```
In [6]: Dic['B']
Out[6]: 1
```

Pour ajouter un champ affecter une valeur à une nouvelle clef. Pour supprimer un champ utiliser la fonction **del()**.

```
In [7]: Dic['E'] = 4     # ajout d'un champ
In [8]: del(Dic['C'])    # suppression d'un champ
In [9]: Dic
{'A' : 0, 'B' : 1, 'D' : 3, 'E' : 4}
```

• Méthodes des dictionnaires

Les dictionnaires admettent les méthodes suivantes :

Méthodes des dictionnaires	
`D.keys()`	retourne la liste des clefs du dictionnaire `D`
`D.values()`	retourne la liste des valeurs du dictionnaire `D`
`D.items()`	retourne la liste des champs `(key,value)` de `D`
`D.copy()`	retourne une copie du dictionnaire `D`.

2 Fonctions

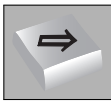
2.1 Déclaration de fonction

L'utilisateur peut définir ses propres *fonctions*. Le concept de fonction est fondamental en programmation structurée.

Exemple

```
def carre(x):
    return x**2
```

A l'interprétation, rien ne semblera se passer, elle a cependant consisté à définir la fonction `carre` par l'interpréteur. On peut alors l'utiliser, au sein du programme ou à partir de la console, de la façon suivante :



L'interprétation d'une fonction définit son fonctionnement à l'interpréteur. Elle correspond à la définition d'une variable de type `function`.

```
In [1]: carre(2)
Out[1]: 4

In[2]: carre(-2.)
Out[2]: 4.0
```

Ici l'appel de `carre(x)` avec pour paramètre `x` un nombre, retourne la valeur de l'expression x^2 grâce à l'instruction `return`.

L'instruction **`return`** `exp` renvoie l'expression `exp` à celui qui a appelé la fonction. Ce peut

être l'utilisateur, dans quel cas le résultat est affiché dans la console après `Out[n]:` qui suit l'appel à la ligne `In [n]:`. Mais ce peut être une autre fonction :

```
def puissance4(x):
    sq = carre(x)
    return carre(sq)
```

Après interprétation :

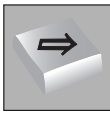
```
In [3]: puissance4(3)
Out[3]: 16
```

2.1.1 Syntaxe pour la déclaration d'une fonction

La définition d'une fonction se fait à l'aide du mot-clef **`def`** et le retour du résultat, s'il y a lieu, à l'aide du mot clef **`return`**. La syntaxe de définition d'une fonction est la suivante :

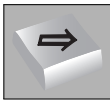
```
def NomdelaFonction(Paramètres):
    ......  Instruction1
    ......  Instruction2
            ⋮
    ......  Dernière instruction
```

meme espace — bloc d'instructions

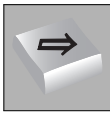


Une fonction prend aucun, un ou plusieurs paramètres, séparés par des virgules. Lorsqu'on l'invoque avec des valeurs en place des paramètres, elle s'exécute pour effectuer une action ou retourner un résultat (**`return`**).

- L'instruction '`def`' permet la définition d'une fonction.
- Il est suivi du nom de la fonction. Le nom d'une fonction suit les mêmes règles que les noms de variables : chaine alphanumérique uniquement constituée de lettres, de chiffres et de _, ne commençant pas par un chiffre, et différent d'un mot-clef du langae.
- Le nom de la fonction est obligatoirement suivi de parenthèses pouvant contenir des noms de variables, séparées si nécessaire de virgules : ce sont ses **paramètres**).
- La parenthèse fermante est suivie de deux points '`:`'.
- Suit un bloc d'instructions. **Elles doivent toutes être décalées du même nombre d'espaces** (en général une tabulation).
- Le résultat de la fonction, si il y a, est retourné grâce à l'instruction '`return`'.



L'instruction **`return`** provoque la sortie de la fonction. Elle peut être utilisée sans argument, dans quel cas elle renvoie `None`.



En l'absence de **`return`**, on peut parler de **procédure**. En Python, cependant, les procédures n'existent pas, une fonction renvoie toujours quelque-chose, même en l'absence de **`return`** ; par défaut une fonction renvoie la valeur `None`.

il faut respecter rigoureusement la syntaxe de définition d'une fonction. Le bloc d'instruction est décalé d'un espace de tabulation. Le non-respect de cette règle entrainera une erreur `Indentation Error`.

2.1.2 Valeurs d'arguments par défaut

On peut assigner des valeurs par défaut aux arguments d'une fonction. On peut assigner des valeurs par défaut à plusieurs arguments. Il faut respecter les règles suivantes :

- On ne peut assigner des valeurs par défaut qu'aux derniers argumentss de la fonction.
- Lorsqu'on l'on passe $k < n$ données à une fonction qui admet n arguments, dont certains ont une valeur par défaut, les données passées sont affectées aux k premiers arguments.

Exemple.

```
def f(m, n=2, p=3):
    return (m, n, p)
```

On peut appeler la fonction `f` avec 2 arguments, ou un seul, dans quel cas l'argument `n` prendra sa valeur par défaut, 1 :

```
In [1]: f(-1,-2,-3)
Out[1]: (-1, -2, -3)

In [2]: f(-1,-2)
Out[2]: (-1, -2, 3)

In [3]: f(-1)
Out[3]: (-1, 2, 3)
```

2.1.3 Commenter une fonction

Après la ligne d'entête d'une fonction on peut insérer un commentaire entre deux délimiteurs """. Dans ce cas, le passage en paramètre du nom de la fonction provoque l'affichage de l'entête et du commentaire.Cela permet de définir en même temps qu'un fonction, son aide en ligne.

Exemple :

```
def maFonction(x):
    """Retourne la valeur du polynôme X^2-2X+1
    evalué au paramètre x."""
    return x**2-2*x+1
```

Ainsi dans la console, après que la fonction ait été définie :

```
In [1]: help(maFonction)
Help on function maFonction in module __main__:
```

```
maFonction(x)
    Retourne la valeur du polynôme X^2-2X+1
    evalué au paramètre x.
```

2.1.4 Exemples : calcul de moyenne et de variance

Il faut savoir écrire des fonctions qui prennent en paramètre une séquence numérique et calculent sa moyenne, et sa variance.

```
def moyenne(X):
    """Calcule la moyenne d'une séquence
       numérique non vide"""
    return sum(X) / len(X)

def variance(X):
    """Calcule la variance d'une séquence
       numérique non vide"""
    m = moyenne(X)
    X2 = [ (x-m)**2 for x in X ]
    return moyenne(X2)
```

Ces deux fonctions sont de complexités linéaires dans le pire et le meilleur des cas, en fonction de la longueur de la séquence passée en paramètre.

La **complexité** d'une fonction ou programme est une notion fondamentale en Informatique. On désigne par *opération élémentaire* toute instruction dont le temps d'exécution reste borné quelque soit la taille de ses paramètres.

On compte le nombre d'opérations élémentaires effectuées durant l'exécution, et exprimées en fonction de la taille des paramètres. La complexité dans le pire des cas est l'ordre du nombre maximal d'opérations élémentaires. La complexité dans le meilleur des cas est l'ordre du nombre minimal d'opérations élémentaires ; tous deux s'expriment en fonction de la taille des données.

La complexité permet de mesurer comment évolue le temps d'exécution lorsque la taille des données en arguments augmentent. Par exemple, pour une complexité linéaire, comme ci-dessus, le temps de calcul double approximativement lorsque la taille de la séquence double.

On renvoie à l'ouvrage « Informatique 1ère année » du même auteur, chez le même éditeur, chapitre 6, pour une révision de cette notion, qu'il est essentiel de maîtriser.

2.1.5 Définir une fonction numérique avec `lambda`

Le mot-clef **`lambda`** permet une définition plus rapide, en une ligne, pour des fonctions simples qui :

- prennent en argument un ou plusieurs paramètres numériques,

- retourne la valeur d'une expression numérique

La syntaxe est :

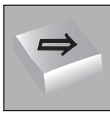
```
NomDeLaFonction = lambda paramètres : expression
```

Les paramètres, s'il y en a plusieurs, sont séparés par des virgules.

Exemples :

```
In [1]: maFonction = lambda x : x**2-2*x+1

In [2]: maFonction(0)
Out[2]: 1

In [3]: norme = lambda x,y : (x**2+y**2)**0.5

In [4]: norme(1,1)
Out[4]: 1.4142135623730951
```

C'est très utile pour définir des fonctions mathématiques de façon concise.



L'instruction :

```
f = lambda x1, ..., xn : expression(x1, ..., xn)
```

permet une définition concise d'une fonction numérique `f`.

2.1.6 Passage d'une valeur à une fonction

Une variable utilisée dans la définition d'une fonction est créée à l'appel de la fonction, et détruite à la sortie.

Lorsque l'on passe la valeur d'une variable en paramètre à une fonction, tout se passe comme si c'était **une copie de cette valeur** qui lui était passée. On parlerait de passage par valeur, par opposition à un passage par référence où ce serait la valeur elle-même et non pas une copie qui serait utilisée.

Exemple.

```
def incremente(x):
    x = x+1
    return x
```

```
In [1]: x = 1
In [2]: incremente(x)
Out[2]: 2

In [3]: x
1
```

Cependant dans le cas particulier d'une liste, la valeur de la liste est une référence, c'est à dire l'adresse où les données son stockées en mémoire. Modifier les éléments d'une liste passée en paramètre à une fonction, ou lui ajouter ou supprimer des éléments, modifiera effectivement la liste.

Exemple.

```
def f1(L):
    L = []
def f2(L):
    L.append(0)

def f3(L,a):
    L[-1] = a
```

```
In [1]: L = [1, 2, 3]
In [2]: f1(L)
In [3]: L        # La liste n'a pas été modifiée
Out[3]: [1, 2, 3]

In [4]: f2(L)
In [5]: L        # La liste a été modifiée
Out[5]: [1, 2, 3, 0]

In [6]: f3(L,4)
In [7]: L        # La liste a été modifiée
Out[7]: [1, 2, 3, 4]
```

Il faut prendre garde qu'une liste a pour valeur l'adresse où sont situés ses éléments au sein de la mémoire, c'est à dire une référence.

2.1.7 Précisions sur le passage d'une valeur à une fonction

En réalité, en Python, cette terminologie de passage par valeur/référence est un peu désuète (c'est pourquoi on précise « tout se passe comme si... »), et le passage d'un argument se fait plutôt par référence, non pas sur l'argument mais sur l'objet qu'il représente. Dans ce langage les variables sont des noms, figurant dans un espace de nom et associé à une valeur, (on parle d'objet). Le programme principal a un espace de noms, et chaque fonction a son propre espace de nom qui n'existe que durant son exécution en se substituant à l'espace de nom du programme.
Par exemple l'instruction `x = y = 1` dans le programme crée un objet de valeur l'entier `1`, auquel est associé la liste des noms `x` et `y`. L'appel de `incremente(x)` crée un espace de noms valable durant l'appel de la fonction, qui associe à l'objet entier `1`, valeur de `x`, un nouveau nom `x`, l'instruction `x = x + 1` crée à partir de l'objet entier `1` un nouvel objet entier `2`, auquel est associé le nom `x` de l'espace de nom de la fonction. Dans l'espace de noms du programme, le nom `x` demeure, lui, associé à l'objet entier `1`, qui demeure inchangé. À la sortie de la fonction son espace de noms est supprimé.

L'identifiant d'un objet peut s'obtenir à l'aide de la fonction **`id()`** qui prend en argument une expression. Par exemple :

```
In [1]: x = y = 1
```

```
In [2]: id(x)
Out[2]: 4297326624

In [3]: id(y)   # x et y font référence au même objet
Out[3]: 4297326624

In [4]: z = y
In [5]: id(z)   # x, y et z font référence au même objet
Out[5]: 4297326624

In [6]: id(1)
Out[6]: 4297326624

In [7]: id(2)
Out[7]: 4297326656
```

C'est l'adresse en mémoire où est située l'objet, auquel l'expression en argument fait référence, que **id()** retourne. On constate qu'un entier de petite taille est stocké sur 32 bits (4297326656 – 4297326624 = 32).

L'objet auquel fait référence un argument est passé par référence à la fonction, mais un nouvel espace de nom lui est associé durant l'appel.

```
In [1]: def f(a):
   ...:     return id(a)
   ...:

In [2]: x = 1

In [3]: id(x)
Out[3]: 4297326624

In [4]: f(x)    # Retourne la même référence
Out[4]: 4297326624
```

Au final pour l'utilisateur tout se passe comme si le passage de l'argument se faisait par valeur, car les entiers, comme la plupart des objets, sont **non-mutables** : la modification d'un valeur ne modifiera pas l'objet mais en créera un nouveau dans la mémoire, qui héritera de son propre espace de nom. Les listes quant à elles sont des objets **mutables** : modifier un élément, ajouter ou supprimer, modifie l'objet liste en mémoire.

L'opérateur booléen **is** permet de déterminer si deux variables sont associées au même objet. L'expression `a is b` a même valeur que `id(a) == id(b)`.

```
In [1]: L = [1,2]

In [2]: L1 = L ; L is L1
Out[2]: True
```

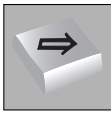
```
In [3]: L2 = L[:] ; L is L2
Out[3]: False
```

Ainsi `a is b` est une condition plus forte que `a == b`. L'usage, par exemple, de `a is None` ou de `L is []` est préférable, respectivement, à celui de `a == None` ou `L == []`.

2.1.8 Variables locales, variables globales

Une variable peut être **globale** ou **locale** selon qu'elle est définie :

- Au sein d'une définition de fonction : **variable locale**.
- Dans le programme hors d'une définition de fonction, ou dans la console : **variable globale**.

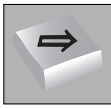


Une variable est locale lorsqu'elle est définie au sein d'une fonction. Sinon elle est globale. Une variable locale est définie dans l'espace de noms de la fonction, une variable globale l'est dans l'espace de noms du programme.

Le caractère global ou local d'une variable détermine sa durée de vie en mémoire et sa modifiabilité.

Durées de vie en mémoire :

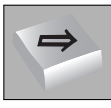
- Une variable locale est créée en mémoire à l'appel d'une fonction et détruite à la sortie de la fonction :
- Une variable globale définie dans le programme est créée en mémoire à l'interprétation du programme et y demeure jusqu'à l'interprétation suivante ou jusqu'à la fermeture de la console.
- Une variable globale définie dans la console est créée en mémoire à sa définition, et y demeure jusqu'à la fermeture de la console.



La durée de vie en mémoire d'une variable globale est celle du programme. La durée de vie en mémoire d'une variable locale est celle de l'exécution de la fonction où elle est définie.

En particulier, à partir de la console on ne peut manipuler que des variables globales.
Une variable globale est accessible et modifiable dans l'environnement du programme où elle est définie.

Une variable globale est accessible en lecture par une fonction définie dans le même environnement, mais elle est **non modifiable** par cette fonction.



Au sein d'une fonction, une variable globale peut être lue mais non modifiée.

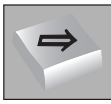
Si dans une fonction on crée une variable (locale) ayant même nom qu'une variable globale déjà définie, il s'agit d'une nouvelle variable qui est créée.

Pour créer ou pour modifier une variable globale à l'intérieur d'une fonction, utiliser le mot-clef **`global`** pour déclarer comme étant globale la variable dont le nom suit :

```
def incremente(x):
    global x     # Déclaration de la variable globale x
    x = x + 1    # Elle devient modifiable
```

En utilisant le mot-clé **global** la variable est recherchée dans l'espace de noms du programme, et en cas d'absence y est rajoutée.

```
In [1]: x = 1
In [2]: incremente(x)
In [3]: x
Out[3]: 2
```



global permet la déclaration d'une variable globale au sein d'une fonction, pour la créer ou pour pouvoir modifier une variable globale préexistante.

Il faut éviter tant que possible l'usage de variables globales : l'accès à l'espace de noms du programme ralentit très sensiblement l'exécution de la fonction.

2.2 Fonctions prédéfinies

On dispose de fonctions prédéfinies (on a déjà vues des fonctions numériques au paragraphe §1.2.), dont nous voyons certaines dont l'utilisation est incontournable.

2.2.1 Fonctions de conversion

A chaque type de donnée, est associée une fonction de conversion de même nom, qui renvoie le paramètre passé convertit dans le type correspondant (lorsque c'est possible).

Exemple.

```
In [1]: int(-3.5)
Out[1]: -3

In [2]: float(2)
Out[2]: 2.0

In [3]: str(4.5)
Out[3]: '4.5'

In [4]: int('3')
Out[4]: 3

In [5]: list('abc')
Out[5]: ['a','b','c']

In [6]: float('abc')
...
ValueError: could not convert string to float: 'abc'
```

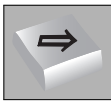
Voici les principales fonctions de conversion :

Fonctions de conversion	
`int()`	Convertit son paramètre en entier
`float()`	Convertit son paramètre en flottant
`complex()`	Convertit son paramètre en complexe
`bool()`	Convertit son paramètre en booléen
`str()`	Convertit son paramètre en chaîne de caractère
`list()`	Convertit son paramètre en liste
`tuple()`	Convertit son paramètre en t-uplet

2.2.2 Fonction de sortie `print()`

La fonction **`print()`** :

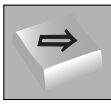
- prend en paramètre une ou plusieurs expressions, séparées par des virgules
- produit l'affichage de leur valeur dans la console, en les séparant d'un espace.



La fonction **`print`** affiche dans la console, les valeurs de ses arguments en les mettant en forme.

```
In [1]: from math import pi
In [2]: R = 3
In [3]: print("La circonférence d'un cercle de rayon", R,\
          "est", 2*pi*R)
La circonférence d'un cercle de rayon 3 est 18.84955592153876
```

Pour l'affichage la fonction **`print`** met en forme ce qui est affiché. Par exemple une chaîne de caractère est affichée sans ses délimiteurs `""` ou `''`, dans l'écriture d'un flottant les chiffres après la virgule sont limités à 15 décimales.



Pour écrire une instruction sur plusieurs lignes, on peut la sectionner par le caractère *backslash* : `\` (voir un exemple ci-dessus).

• Caractères spéciaux

Une chaîne de caractère peut contenir les caractères spéciaux suivants :

Caractères spéciaux	
`\n`	Saut de ligne
`\t`	Tabulation
`\b`	Retour arrière

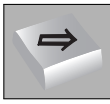
Dans une chaîne de caractère, ils seront mis en forme par la fonction **`print`** en provoquant respectivement :

- Un passage à la ligne

– Un espace de tabulation
– Un effacement du dernier caractère (Del)

Exemple :

```
In [4]: print("Bonjour l'univers")
Bonjour l'univers

In [5]: print("Bonjour\n l'univers")
Bonjour
 l'univers

In [6]: print("Bonjour\t l'univers")
Bonjour   l'univers

In [7]: print("Bonjour\bl'univers")
Bonjou l'univers
```



Le caractère spécial `\n` provoque un saut de ligne.

• Options de `print`

La fonction **`print`** admet trois options, que l'on peut modifier en paramètre.

– L'option `end`.

Par défaut, entre chaque appel de **`print`**, s'exécutera un passage à la ligne. C'est dû à la valeur par défaut `'\n'` de l'option `end`. La valeur de l'option définit ce qui doit être écrit à la fin de l'usage de la fonction **`print`**. On peut la modifier en passant le paramètre `end =` et pour nouvelle valeur une chaine de caractère (ou `None` pour garder la valeur par défaut).

Exemple :

```
print('Bonjour',end='----')
print("l'univers")
```

produit comme affichage dans la console :

```
Bonjour----l'univers
```

– L'option `sep`.

Par défaut, l'affichage de chaque paramètre de **`print`** est séparé d'un espace `' '`. C'est dû à la valeur par défaut `' '` de l'option `sep` :

```
In [1]: print('Bonjour', "l'univers")
Bonjour l'univers
In [2]: print('Bonjour', "l'univers", sep = '')
Bonjourl'univers
```

On change la valeur par défaut en changeant la valeur de l'option en passant en paramètre `sep = val` avec pour nouvelle valeur val une chaîne de caractère (ou `None` pour garder la valeur par défaut).

– L'option `file`.

Désigne le périphérique de sortie, où sera affiché le résultat de **`print()`**. Par défaut, il s'agit de la console, désignée par `sys.stdout`.

– L'option `flush`.

A pour valeur un booléen, par défaut `False`. Avec `True` l'affichage des arguments s'effectue impérativement à l'appel de l'instruction, alors qu'avec l'option par défaut, elle peut survenir après un court temps de latence.

• Formatage tel quel

On peut passer en paramètre à la fonction **`print()`** une chaîne de caractères limitée par trois doubles quotes. Dans ce cas la chaîne de caractère pourra être écrite sur plusieurs lignes, et elle s'affichera alors tel quel :

```
In [1]: print("""Bonjour
   ...: Voici une chaine
   ...: sur plusieurs lignes""")
Bonjour
Voici une chaine
sur plusieurs lignes
```

2.2.3 Fonction d'entrée `input()`

La fonction **`input()`** :

- prend en paramètre aucune ou une chaîne de caractère,
- Elle produit l'affichage de la chaine passée en paramètre, attend la saisie d'une chaine par l'utilisateur, et retourne la chaîne saisie.

La chaîne de caractère doit être saisie sans apostrophes '.' ou guillemets ".". Il faut ensuite appuyer sur la touche Entrée pour valider la saisie.

Pour récupérer la valeur de la chaine de caractère saisie on peut utiliser une affectation.

Exemple :

```
chaine = input('Quel est votre nom ? ')
print("Bonjour", chaine,"!")
```

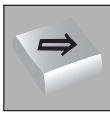
À l'exécution du programme la chaine `Quel est votre nom ?` s'inscrit dans la console :

```
Quel est votre nom ? _
```

son déroulement est suspendu à la saisie de l'utilisateur, qui se termine avec l'appui sur la touche Entrée :

```
Quel est votre nom ? Toto
Bonjour Toto !
```

Le déroulement du programme reprend alors normalement.



La fonction **input()** affiche dans la console son paramètre et attend que l'utilisateur saisisse une chaine suivie de Entrée. Elle permet à l'utilisateur d'interagir avec le programme.

Si l'on n'a besoin d'une valeur numérique il faut convertir le résultat retourné à l'aide d'une fonction de conversion de type : **int()** ou **float**.

```
In [1]: n = int(input('Saisissez un entier ')) ;\
        print('Son carré est', n**2)

Saisissez un entier 3
Son carré est 9
```

Autrement un calcul produira une erreur ou un résultat inattendu.

3 Structures de contrôle de flux

Python reconnaît les trois structures de contrôles :

- Le branchement conditionnel : **if ... [elif] ... [else]**,
- deux structures de boucles : **while ...**
  for ...

3.1 Branchement conditionnel

3.1.1 Syntaxe du branchement conditionnel

La syntaxe du branchement conditionnel est :

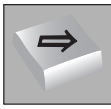
if *condition* **:**
. . . . . *Bloc d'Instructions 1*

else:
. . . . . *Bloc d'instructions 2*

même espace — bloc d'instructions

- Le branchement conditionnel débute avec **if** suivi d'une condition booléenne *condition*, (ou d'une expression de type quelconque qui sera alors convertie en booléen avec **bool**), suivie d'un double point **:**
- Si *condition* a valeur `True` le premier bloc d'instruction est exécuté, le deuxième ne l'est pas, puis l'exécution du programme se poursuit.
- Si *condition* a valeur `False` le deuxième bloc d'instruction est exécuté, le premier ne l'est pas, puis l'exécution du programme se poursuit.

Le deuxième bloc qui suit l'instruction **else** est optionnelle. Les deux blocs doivent être décalés d'un même espace de tabulation.



Le premier bloc n'est exécuté que lorsque la condition est `True`. Le **else** est optionnel ; le deuxième bloc, s'il est présent n'est exécuté que lorsque la condition est `False`.

Exemple Fonction qui détermine si un nombre entier est pair ou impair.

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

ou encore :

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

(Puisque **return** provoque la sortie de la fonction, l'instruction **return** `False` ne sera exécutée que si la condition `(n%2 == 0)` est fausse.)

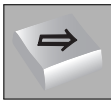
3.1.2 Branchements multiples `if ... elif ... else`

L'écriture de tests imbriqués :

if (*condition 1*) :
    Bloc d'instructions 1
else :
    if (*condition 2*) :
        Bloc d'instructions 2
    else :
        Bloc d'instructions 3

s'écrit à l'aide d'une seule structure de test :

if (*condition 1*) :
    Bloc d'instructions 1
elif (*condition 2*) :
    Bloc d'instructions 2
else :
    Bloc d'instructions 3



La *condition 2* de **elif** n'est évaluée que si *condition 1* a échouée. En cas d'échec c'est le bloc du **else** qui est exécuté.

elif et **else** sont optionnels. On peut utiliser plusieurs fois **elif** dans une structure de test : **if ... elif ... elif ... else**.

Exemple.

```
x = complex(input('Saisissez un nombre : '))
if x.imag != 0:
    print("C'est un nombre complexe")
elif x != int(x.real):
    print("C'est un nombre décimal")
elif x.real < 0:
    print("C'est un entier relatif")
else:
    print("C'est un entier naturel")
```

3.2 Structure de boucle for

Une boucle **for** permet de répéter une suite d'instructions un nombre prédéfini de fois. Au cours de l'exécution une variable est créée et ses valeurs varient au sein d'une séquence ; la séquence peut-être une liste, un tuple, une chaîne de caractère, ou un itérateur.

3.2.1 Syntaxe d'une boucle for

La syntaxe générale d'une boucle **for** est :

for *var* **in** *séquence*:
..... *Bloc d'Instruction*

même espace — bloc d'instructions

Exemple.

```
liste = [1, 2, 3, 4, 5]

for var in liste:
    print(var, end = ' ; ')
```

produit l'affichage :

```
1 ; 2 ; 3 ; 4 ; 5 ;
```

Ou encore, avec une chaîne de caractère :

```
chaine = 'TOTO'

for c in chaine:
    print(c, end = '-')
print('\b ')
```

produit :

```
T-O-T-O
```

On s'interdit dans une boucle **for** de modifier dans le bloc d'instruction la variable ou la séquence sur lesquelles s'applique la boucle.

3.2.2 Utilisation de l'itérateur range

La fonction **range**(n) retourne un *itérateur* sur les n premiers entiers naturels (de 0 à n−1 inclus). Les valeurs intermédiaires ne sont pas stockées en mémoire et une instruction utilisant **in** permet de déterminer l'appartenance d'un élément.

```
In [1]: 2 in range(10)
True
In [2]: 10 in range(10)
False
In [3]: 1.0 in range(10)
True
```

La fonction **range**() peut prendre 1, 2 ou 3 paramètres, tous entiers

Syntaxe.

- Avec N un entier (**int**), l'instruction :
  `x in range(N)`
  est une expression booléenne valant True si et seulement si x a une valeur entière comprise entre 0 et N−1 (inclus).
- Avec M et N deux entiers (**int**), l'instruction :
  `x in range(M,N)`
  est une expression booléenne valant True si et seulement si x a une valeur entière comprise entre M et N−1 (inclus).
  En particulier **range**(N) est équivalent à **range**(0,N).
- Avec M et N et K trois entiers (**int**), l'instruction :
  `x in range(M,N,K)`
  est une expression booléenne valant True si et seulement si x a une valeur entière de la forme M+k∗K pour k un entier positif, qui soit comprise entre M et N−K (inclus).
  En particulier **range**(M,N) est équivalent à **range**(M,N,1) et **range**(N) est équivalent à **range**(0,N,1).

```
In [1]: 2 in range(3,10)
False
In [2]: 5 in range(3,10)
True
In [3]: 3 in range(1,10,2)
True
In [4]: 4 in range(1,10,2)
False
```

```
In [5]: 1 in range(10,0,-1)
True
```

L'itérateur **range** est une fonction qui prend en paramètre un, deux, ou trois entiers, et permet de décrire une plage de valeurs entières. Le premier paramètre est le point de départ (inclu), le deuxième le point d'arrivée (exclu), et le troisième est le pas.

L'usage principal de la fonction **range()** se fait au sein d'une boucle **for**.

• Déclare une variable `var` variant de `0` à `N-1` :

```
for var in range(N):
.....   Bloc d'Instruction
```

même espace — bloc d'instructions

• Déclare une variable `var` variant de `M` à `N-1` :

```
for var in range(M,N):
.....   Bloc d'Instruction
```

même espace — bloc d'instructions

• Déclare une variable `var` variant de `M` à `N` exclu par pas de `K` :

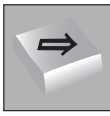
```
for var in range(M,N,K):
.....   Bloc d'Instruction
```

même espace — bloc d'instructions

Tout le bloc d'instruction doit être décalé d'un même espace de tabulation.

Le déroulement de la boucle s'effectue de la façon suivante :

- Au début de la boucle la variable `variable` de type entier (**int**) est créée et sa valeur initialisée à `0` (premier cas) ou `M` (deuxième et troisième cas).
- Le bloc d'instruction s'exécute ; ensuite
- la variable `var` est incrémentée de 1 (2 premiers cas) ou de `K` (dernier cas), et le bloc d'instruction se re-exécute.
- Dès que la variable {`var` a atteint la valeur `N`−1 (deux premiers cas) ou `N`−`K` (troisième cas) le bloc d'instruction s'exécute puis le suite du programme se poursuit.



Une boucle **for** `v` **in range**`(...)` répète une suite d'instruction un nombre prédéfini de fois, en faisant varier une variable entière entre deux valeurs prédéfinies, selon un pas prédéfini.

Une boucle **for** pratiquée sur une séquence – liste, tuple ou chaine de caractère– :

```
# Parcours d'une séquence par élément
for var in séquence:
    # Bloc d'instructions
```

est équivalente à la boucle **for** utilisant un itérateur :

```
# Parcours d'une séquence par indice
for k in range(len(séquence)):
    var = séquence[k]
    # Bloc d'instruction
```

Dans le premier cas on parlera d'un **parcours par élément**, et dans le deuxième d'un **parcours par indice**.

3.2.3 Variantes de parcours d'un liste avec une boucle for

Les deux fonctions suivantes s'avèrent utile pour parcourir une séquence avec une boucle **for**

• **La fonction reversed().**
Elle permet de parcourir une séquence par une boucle **for** du dernier au premier élément.

```
L = [1,2,3]
for elt in reversed(L):
    print(elt, end=' ; ')
```

produira :

```
3 ; 2 ; 1
```

• **La fonction enumerate().**

Elle est pratique pour parcourir une séquence à l'aide d'une boucle **for** simultanément par indice et par élément :

```
for k, var in enumerate(liste):
```

permet de parcourir la séquence `liste` en faisant simultanément varier `k` du premier au dernier indice et `var` du premier au dernier élément.

Elle est équivalente à :

```
for k in range(len(liste)):
    var = liste[k]
    # Bloc d'instruction
```

```
L = ['a','b','c']
for k, elt in enumerate(L):
    print("L'élément d'indice",k,"est",elt)
```

produira :

```
L'élément d'indice 0 est a
L'élément d'indice 1 est b
L'élément d'indice 2 est c
```

3.2.4 Exemples à connaître

On donne en exemple quelques algorithmes qu'il faut maîtriser.

• Recherche linéaire d'un élément dans une séquence

La fonction suivante prend en paramètre une séquence `L` et une donnée `e`. Elle renvoie le booléen `True` si `e` est un élément de `L`, `False` sinon.

```
def recherche(L,e):
    for x in L:
        if x == e:
            return True
    return False
```

Son action est identique à l'expression `e in L`, mais il faut savoir la programmer. Sa complexité est linéaire dans le pire des cas, en fonction de la longueur de la séquence, et $O(1)$ dans le meilleur des cas.

• Recherche du maximum dans une séquence numérique

La fonction suivante prend en paramètre une liste ou séquence de nombres (entiers, flottants) ou plus généralement d'éléments comparables (chaines de caractères), et renvoie son plus grand élément.

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

Sa complexité est linéaire dans le pire et le meilleur des cas, en fonction de la longueur de la séquence.

• Conversion de binaire vers décimal

La fonction prend en paramètre une chaîne de caractères représentant l'écriture binaire d'un entier naturel, et renvoie le nombre représenté.

```
def bin2dec(chaine):
    N = 0
    for bit in chaine:
        N = 2*N + int(bit)
    return N
```

Sa complexité est linéaire dans le pire et le meilleur des cas, en fonction de la longueur de la chaîne de caractères en paramètre.

3.3 La boucle `while`

La boucle **`while`** permet de répéter en boucle une séquence d'instruction, tant qu'une *condition* est vérifiée.

3.3.1 Syntaxe d'une boucle `while`

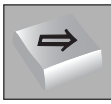
La syntaxe d'une boucle **`while`** est :

`while` *condition*`:`
..... *Bloc d'instruction*
même espace bloc d'instructions

Son fonctionnement est :

- Le mot clef **`while`** est suivi d'une expression booléenne, (ou de type quelconque qui dans ce cas est convertie en booléen par la fonction de conversion **`bool`**) ; elle est suivie de deux points `:`.
- Les instructions répétées dans la boucle **`while`** suivent dans un bloc, décalée d'un même espace de tabulation.
- Tant que *condition* est vérifiée, la suite d'instruction du bloc est exécutée.
- Dès que la condition n'est plus vérifiée, le programme se poursuit à la suite.

Tout ce qui peut être fait avec une boucle **`for`** peut l'être avec une boucle **`while`** ; la réciproque est fausse. Une boucle **`while`** est plus générale.



Privilégier une boucle **`for`** à une boucle **`while`** dès que le nombre d'itérations de la boucle est connu à l'avance.

3.3.2 Interruption du déroulement d'une boucle

Deux fonctions permettent d'altérer le déroulement d'une boucle **`for`** ou **`while`** :

• **`break`** provoque une sortie anticipé,

• **`continue`** provoque le passage à l'itération suivante.

On peut toujours se passer de leur usage, mais elles s'avèrent cependant très pratiques. Elles peuvent être utilisées aussi bien dans une boucle **`while`** que dans une boucle **`for`**.

Sortie anticipée et saut à l'itération suivante	
break	Sortie anticipée de la boucle. Le programme reprend à la suite de la boucle.
continue	Passage à l'itération suivante. Toute la partie entre l'instruction et la fin de boucle est ignorée.

3.3.3 Exemples à connaître

Les exemples que nous donnons ici sont à maîtriser.

• Calcul de $pgcd$ par l'algorithme d'Euclide.

Cet algorithme prend en paramètre deux entiers positifs et renvoie son pgcd. Très efficace, cet algorithme a été décrit par le mathématicien grec Euclide il y a plus de 23 siècles, dans son ouvrage fondateur des sciences Mathématiques, *les éléments d'Euclide.*

```
def pgcd(a,b):
    """Renvoie le pgcd de deux entiers naturels a et b"""
    while b != 0:
        a, b = b, a%b
    return a
```

Pour démontrer la correction du Théorème, on remarque d'abord que la boucle se termine car b est un variant de boucle, et que le résultat retourné est le $pgcd$ de a et b car $pgcd$(a,b) est un invariant de boucle et $pgcd(a,0) = a$).

Il faut savoir de démontrer qu'un algorithme renvoie bien le résultat qui est attendu. On appelle cela la *correction* de programme. Lorsque le programme est constitué d'une simple boucle **while**, il faut d'abord montrer que la boucle finit par s'arrêter. Pour celà il suffit d'exhiber un **variant de boucle**, c'est à dire une expression dont la valeur ne prend que des valeurs entières, est minorée, et décroît strictement à chaque passage dans la boucle.
Pour montrer qu'à la sortie de boucle on a bien calculé le résultat attendu, on utilise souvent un **invariant de boucle**, c'est à dire une expression, ou une proposition, dont la valeur demeure inchangée à chaque passage dans la boucle.

On renvoie au chapitre 6 de l'ouvrage « Informatique 1ère année » du même auteur, chez le même éditeur, pour une révision de cette notion, qu'il faut maîtriser pour des exemples simples, comme pour le cas d'un programme constitué d'une simple boucle **while**. L'algorithme d'Euclide en est un exemple fondateur. Voir aussi l'exercice 2.

Le *Théorème de Lamé* permet de montrer que la complexité de l'algorithme d'Euclide est logarithmique en fonction de $min(a,b)$; plus précisément, le nombre de passage dans la boucle **while** est au plus 5 fois le nombre de chiffres dans l'écriture décimale du plus petit des 2 nombres. Ce qui est particulièrement efficace ! On renvoie à l'exercice 7 du chapitre 6 de l'ouvrage « Informatique 1ère année » pour une preuve de ce résultat.

• Conversion de décimal en binaire.

L'algorithme prend en paramètre un entier positif, et renvoie une chaîne de caractères `'0'` et `'1'` représentant son écriture binaire.

```
def dec2bin(n):
    if n == 0:
        return '0'
    chaine = ""
    while n > 0:
        chaine = str(n%2) + chaine
        n = n//2
    return chaine
```

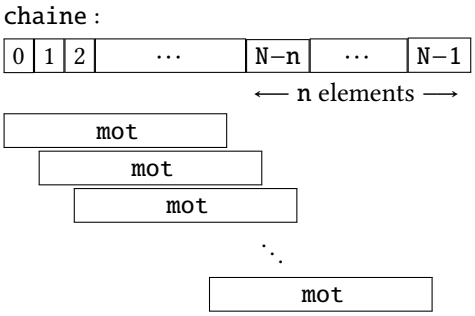
Sa complexité est en $O(\log(n)^2)$. Plus précisément le nombre de passage dans la boucle est $\lfloor \log_2(n) \rfloor + 1$, et le nombre d'opérations élémentaires a même ordre que $1+2+\cdots+\lfloor \log_2(n) \rfloor$ (à cause de la modification de `chaine`, les chaînes de caractère étant non-mutables, une nouvelle chaîne est créée à chaque étape). Un algorithme qui renverrait plutôt un tableau de bits, pourrait, lui, être implanté avec une complexité logarithmique en $O(\log(n))$.

Sur le même modèle, l'algorithme s'adapte pour écrire une entier naturel dans n'importe quelle base $a \in \mathbb{N} \setminus \{0, 1\}$. Voir à ce sujet l'exercice 3.

• Recherche d'une sous-chaîne dans une chaîne de caractère.

L'algorithme prend en paramètre deux chaînes de caractères, `chaine` et `mot`, et renvoie un booléen `True` ou `False` selon si la seconde chaîne apparaît comme sous-chaîne de la première.

Schématiquement, en notant `N = len(chaine)` et `n = len(mot)` :



On compare tous les caractères de `mot` avec ceux des sous-chaînes de `chaine` ayant longueur `n`, à toutes les positions, en partant de la première, à l'indice `0`, jusqu'à la dernière, à l'indice `N-n`. On commence par comparer le premier caractère de `mot`, puis le deuxième, etc., jusqu'à avoir trouvé la sous-chaîne recherchée ; à la première discordance, s'il en est, on reprend à l'indice suivant dans `chaine`.

```
def contient(chaine,mot):
    """ Renvoie True lorsque mot apparaît comme sous-chaîne
    de chaine, et False sinon"""
    N = len(chaine)
    n = len(mot)
    for i in range(N-n+1):
```

```
        recherche = True
        k = 0
        while recherche and k < n:
            if mot[k] != chaine[i+k]:
                recherche = False
            k += 1
        if recherche == True:
            return True
    return False
```

L'exécution est relativement lente : en notant N et n les longueurs respectives des première et deuxième chaîne, la fonction doit tester jusqu'à $N-n$ positions dans `chaine`, et pour chacune effectuer de 1 à n comparaisons de caractères, soit jusqu'à $(N-n)\times n$ opérations. La complexité de cet algorithme est dans le pire des cas $O((N-n)\times n)$.

La recherche de sous-chaîne a des applications importantes en Informatique, par exemple dans les moteurs de recherche comme Google. Cet algorithme –qui est à connaître– est dit naïf ; Nous l'appliquons aux exercices 4 et 5. Il existe des algorithmes bien plus efficaces, dont l'idée suit le même principe mais en effectuant la comparaison en sens inverse, en partant du dernier caractère de `mot`, ce qui permet de ne pas tester toutes les positions (voir l'exercice 6).

• Recherche dichotomique d'un élément dans une liste triée.

L'algorithme de recherche d'un élément dans une liste (voir paragraphe §2.1.4) a une complexité linéaire en fonction de la longueur de la liste. Lorsque la liste est composée d'éléments comparables, et qu'elle est triée, on peut écrire un algorithme de recherche bien plus efficace, de complexité logarithmique, appelé *recherche dichotomique*.

On l'applique ici sur une liste triée dans le sens croissant ; il est facile de l'adapter pour le sens décroissant.

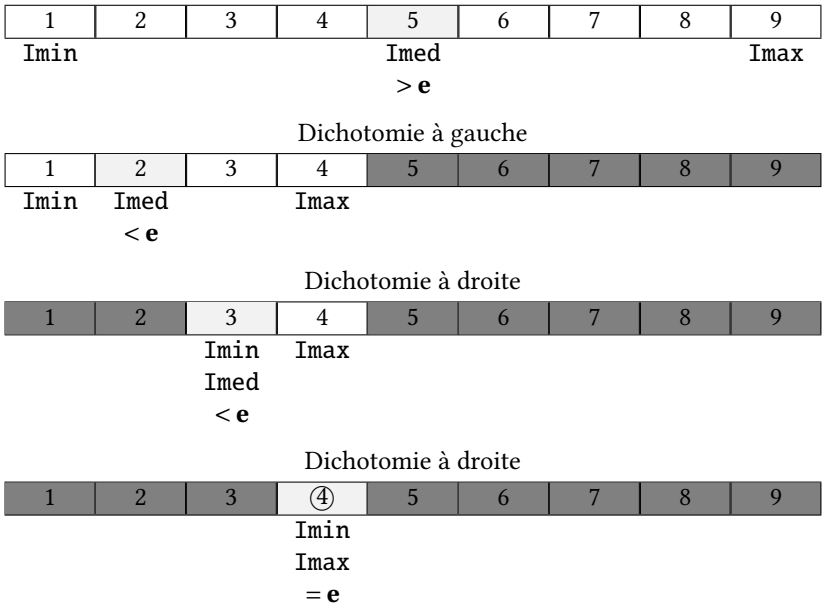
```
def rechercheDichotomique(L,e):
    """Recherche par dichotomie dans une liste triée dans le
    sens croissant. Renvoie True ou False selon que l'élément
    e est ou non présent dans L."""
    Imin, Imax = 0, len(L)-1    # Délimitation de la recherche
    while Imax - Imin >= 0:     # Boucle principale
        Imed = (Imin + Imax)//2 # indice médian
        if L[Imed] == e:        # Cas de succès
            return True
        elif L[Imed] < e:
            Imin = Imed + 1     # dichotomie à droite
        else:
            Imax = Imed - 1     # dichotomie à gauche
    return False                # Cas d'échec
```

L'algorithme utilise deux variables `Imin` et `Imax` de type entiers qui délimitent la partie du tableau `L` à inspecter ; initialement `Imin` et `Imax` sont les indices des premier et dernier éléments de `L`.

Tant que `Imin` et `Imax` délimitent une partie non-vide et que l'élément `e` recherché n'a pas été trouvé, on compare `e` avec l'élément au milieu de la zone délimitée, c'est-à-dire celui d'indice médian `Imed = (Imin+Imax)//2` dans `L`.
Si la comparaison réussit, on renvoie le booléen `True` ; `e` a été trouvé dans `L`. Sinon, si `e` est plus grand, on fait une dichotomie à droite, c'est-à-dire on change `Imin` en `Imed+1`, et si `e` est plus petit, une dichotomie à gauche, on change `Imax` en `Imed−1`.

Pour que l'algorithme fonctionne la liste `L` doit être triée dans le sens croissant. Pour une liste triée dans le sens décroissant on échange simplement les conditions de dichotomie à droite ou à gauche.
La complexité est $O(\log(n))$ dans le pire des cas (avec $n =$ `len(L)`). L'algorithme applique le principe de dichotomie, et plus généralement de « diviser pour régner » dont nous reparlerons dans la suite de l'ouvrage.
Cet algorithme est très efficace et largement utilisé : lorsqu'on est amené à répéter des recherche d'éléments dans une séquence d'éléments ordonnables, (un numéro dans l'annuaire téléphonique par exemple, ou un mot dans le dictionnaire), une bonne approche consiste à trier une fois pour toute la séquence pour ensuite pouvoir appliquer à chaque recherche une dichotomie.

Exemple. Recherche de **e** = **4** dans le tableau suivant :



L'élément `e` a été trouvé. Il a fallu deux dichotomies dans un tableau de 9 éléments.

4 Manipulation de fichier texte

Il y a deux types de fichiers :

– Les fichiers binaires, constitués de données brutes en binaire ; ils sont notamment utilisés pour les programmes.

– Les fichiers textes ; ils sont constitués de chaînes de caractères.

L'usage de fichiers textes est incontournable en programmation. Nous voyons comment manipuler des fichiers textes sous Python.

4.1 Ouverture et fermeture de fichier

Création et manipulation d'un fichier se font par l'intermédiaire d'un *objet-fichier*, généré par la fonction **open**() :

```
objet_fichier = open(nom du fichier, mode d'accès).
```

Les deux paramètres sont des chaînes de caractère :

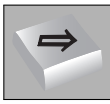
– `nom du fichier` est le nom du fichier, avec son extension, et éventuellement avec son chemin d'accès.

– `mode d'accès` est le mode d'accès pour l'ouverture du fichier.

Le premier paramètre est obligatoire.

Le mode d'accès peut être :

- `'w'` : (write) ouverture pour écriture seule. Lorsque le fichier n'existe pas il est créé dans le répertoire de travail ; lorsque le fichier existe il est écrasé.
- `'a'` : (append) ouverture pour écriture seule. Lorsque le fichier n'existe pas il est créé dans le répertoire de travail ; lorsque le fichier existe les données écrites le seront à la suite.
- `'r'` : (read) ouverture pour lecture seule. Le ficher doit exister dans le répertoire de travail. C'est le paramètre par défaut.
- `'+'` : ouverture pour lecture et écriture. Le fichier doit exister.



Lorsqu'aucun mode d'accès n'est passé en paramètre, par exemple `f.open("fichier")`, l'objet-fichier est ouvert en lecture seule.

Attention à ne pas ouvrir un fichier pré-existant en mode d'accès `'w'`. Son contenu serait supprimé.

On utilise le plus souvent :

- Le mode d'accès `'w'` pour créer le fichier, et y écrire ses premières lignes.
- Le mode d'accès `'a'` pour ouvrir le fichier et y rajouter des lignes à sa suite.
- Le mode d'accès `'r'` pour ouvrir le fichier pour une lecture seule.

Pour ouvrir le fichier, soit il doit être placé dans le répertoire courant (que l'on pourra redéfinir, voir le paragraphe suivant §4.2), soit donner avec le nom du fichier, son chemin d'accès.

Une fois la lecture et l'écriture dans le fichier terminés il faut refermer le fichier avec l'instruction :

```
objet_fichier.close()
```

Il est important de bien penser à refermer un fichier à la fin de sa manipulation, pour deux raisons :

- Les modification faites ne seront répercutées sur le véritable fichier qu'après fermeture de l'objet-fichier.
- Tant qu'un fichier n'a pas été refermé, il ne peut plus être ouvert.

Il faut penser à refermer un objet-fichier à la fin de sa manipulation.

4.2 Connaître et changer le répertoire de travail

Tous les fichiers manipulés doivent se trouver dans le répertoire de travail. Autrement l'ouverture d'un fichier absent du répertoire de travail générera le message d'erreur :

```
In [1]: f = open('fichier')
------------------------------------------------------------

...
FileNotFoundError: No such file or directory: 'fichier'
```

Aussi il est très utile de disposer de fonctions permettant de connaître et de redéfinir le répertoire de travail. De telles fonctions sont disponibles dans le module `os` qui contient des commandes du système d'exploitation.

• La fonction `getcwd()` du module `os` (pour *GET Current Working Directory*) permet de connaître le répertoire de travail :

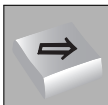
```
In [3]: os.getcwd()
Out[3]: '/Users/NomUtilisateur/documents'
```

• La fonction `chdir()` (pour *CHange DIRectory*) du module `os` permet de redéfinir le répertoire de travail. Elle prend en paramètre une chaine de caractère qui spécifie le chemin d'accès du nouveau répertoire de travail. Par exemple pour que le répertoire de travail devienne celui dont le chemin d'accès est `/Users/NomUtilisateur/documents` :

```
In [1]: import os
In [2]: os.chdir('/Users/NomUtilisateur/documents')
```

• Le répertoire de travail est souvent le répertoire utilisateur, c'est à dire celui dans le répertoire `Utilisateurs` de même nom que le nom de login de la session ouverte. Le chemin d'accès du répertoire utilisateur peut s'obtenir dans le dictionnaire `environ` par l'entrée de clé `'HOME'` :

```
In [4]: os.environ['HOME']
Out[4]: '/Users/NomUtilisateur'
```



Sous les systèmes d'exploitation Mac-OS et Linux, l'arborescence des dossiers est décrite à l'aide du caractère *slash* : `/`, qui désigne aussi seul le répertoire racine (partition principale du disque dur contenant tous les fichiers).

Sous le système d'exploitation Windows, le répertoire racine est désigné par le nom de la partition, le plus souvent `C:`, et l'arborescence est décrite par le caractère *antislash* : `\`, par exemple `C:\Users\NomUtilisateur`. Pour définir le chemin d'accès, dans les arguments de la fonction **`open()`** ou des fonctiions du module `os`, il faudra sous Windows remplacer l'antislash `\` par un *double-slash* : `//`.
Par exemple : `os.chdir('C://Users//NomUtilisateur//MonDossier')`.

Attention, le chemin d'accès d'un répertoire n'est pas toujours celui affiché dans l'explorateur de fichier (ou dans le finder sous Mac-OS). Par exemple, le répertoire `Users` et souvent francisé en `Utilisateurs`.
Les informations d'un fichier ou dossier obtenues après un clic-droit permettront par copie du chemin d'accès d'obtenir son véritable nom.

4.3 Méthode pour la lecture/écriture des objets fichiers

Les fichiers textes sont composés de lignes, séparées par le caractère de fin de ligne `'\n'`. Le fichier se termine par un caractère spécifique de fin de fichier EOF (End Of File). Par défaut, les fichiers texte de Python ne doivent contenir que des caractères de la table unicode (utf-8), ce qui autorise notamment les caractères accentués ; on peut la modifier via l'option `encoding = ...` de la fonction **`open()`**.

• Méthodes des objet-fichiers pour l'écriture.

Lorsque l'objet-fichier est ouvert en écriture :

- `write()` prend en paramètre une chaine de caractère ; l'écrit en en fin de fichier. La fonction renvoie le nombre de caractères écrits.
- `writelines()` prend en paramètre une liste de chaînes de caractères ; les écrit en fin de fichier en les concaténant.

La lecture se fait à partir de la position d'un curseur. Le curseur est initialement en début de fichier. Il se déplace au fur et à mesure de la lecture.

• Méthodes des objet-fichiers pour la lecture :
lorsque l'objet-fichier est ouvert en lecture :

- `read()` lit l'intégralité du fichier à partir de la position du curseur. Si on lui passe en paramètre un nombre entier `n` il limite la lecture à `n` caractères.
- `readline()` lit la ligne suivante. Si on lui passe en paramètre un nombre entier `n` il limite la lecture à `n` caractères.
- `readlines()` retourne une liste contenant toutes les lignes du fichier.

4.4 Autres fonctionnalités utiles des objets fichiers

Les objets-fichiers sont des objets séquentiels. Au fil de la lecture/écriture un curseur de position avance dans le fichier. Pour s'y déplacer, notamment revenir en début de fichier :

- `tell()` est une méthode qui renvoie la position dans le fichier, exprimée en caractère. Le début de fichier correspond à la position `0`.

– La méthode `seek()` prend en paramètre un entier positif `n` et déplace le curseur au `n`-ième caractère du fichier. Par exemple : **`file.seek(0)`** ramène le curseur de lecture en début de l'objet-fichier **`file`**.

Un objet-fichier est itérable, on peut le parcourir ligne après ligne avec une boucle **`for`** :

– **`for`** `ligne` **`in file:`**

La variable `ligne` prendra pour valeur successives toutes les lignes du fichier **`file`**.

Un fichier texte est constitué de caractères constitués en lignes, terminées par le caractère de fin de ligne `'n'`. La lecture se fait ligne après ligne (ou caractère après caractère). Au fur et à mesure de la lecture et de l'écriture, un curseur avance dans le fichier. On parle de fichiers séquentiels : les données sont à la suite les unes des autres.

4.5 Tableau des fonctionnalités des objets-fichiers

Voici un tableau qui résume les fonctionnalités des objets-fichiers.

Ouverture/fermeture	
`file = open(fichier, mode)`	Ouvre le fichier `fichier`, en mode `mode` et déclare l'objet-fichier `file`
`file.close()`	Referme l'objet-fichier `file`
Lecture	
`file.read()`	Lit l'intégralité de l'objet-fichier à partir de la position courante. Renvoie une chaîne de caractères
`file.read(n)`	Avec `n` un paramètre entier, lit n caractères à partir de la position dans l'objet-fichier. Renvoie une chaîne de caractères
`file.readline()`	Lit la ligne courante à partir de la position dans l'objet-fichier. Renvoie une chaîne de caractères
`file.readlines()`	Renvoie la liste des lignes de l'objet-fichier à partir de la position courante
Écriture	
`file.write()`	Prend en paramètre une chaîne de caractères et l'écrit en fin d'objet-fichier.
`file.writelines()`	Prend en paramètre une liste de chaînes de caractères et les écrit en fin d'objet-fichier après concaténation.
Déplacement du curseur	
`file.tell()`	Renvoie la position du curseur dans l'objet-fichier sous forme d'un entier positif
`file.seek(n)`	Avec en paramètre un entier positif `n`, déplace le curseur en position `n`
Parcours dans une boucle `for`	
`for` `l` **`in`** `file:`	Déclare une variable `l` de type chaîne de caractères qui parcourt au sein de la boucle toutes les lignes de l'objet-fichier `file`

5 Utilisation de modules

Un module est un fichier ayant pour extension `.py` contenant des définitions de constantes, classes et fonctions et placé dans un répertoire dédié de l'installation de Python. Importer un module permet d'utiliser ses constantes et fonctions. Nous avons déjà utilisé au paragraphe §1.2.4 le module `math` et au paragraphe §4.2 le module `os`.

Le langage Python a de très nombreux modules, qui en font une langage de programmation modulaire, évolutif, avec un très large champ d'application. Il a notamment d'excellents modules de calcul scientifique, tout à fait au niveau d'outils spécialisés comme Matlab, Scilab, etc.

5.1 Importation de module

Pour les utiliser il faut importer le module, ou n'importer que les fonctions et constantes dont on a besoin. N'importer que les fonctions et constantes nécessaires permet de préserver les ressources utilisées par l'interpréteur.

• Pour importer des fonctions/constantes à partir d'un module, utiliser la commande :

`from` *module* **`import`** *fonctions et constantes séparés par des virgules*

• Pour importer l'intégralité d'un module ; première méthode, à l'aide de la commande :

`from` *module* **`import`** `*`

qui se lit : "from *module* import ALL". Cela importe toutes les définitions incluses dans le module.

• Deuxième méthode ; avec la seule instruction **`import`** :

`import` *module*

Fonctions et constantes doivent alors être précédées d'un préfixe : le nom du module suivi d'un point.

Exemple.

```
In [1]: import math
In [2]: math.sqrt(2)
Out[2]: 1.4142135623730951
In [3]: math.cos(math.pi/4)
Out[3]: 0.7071067811865476
```

• Pour simplifier l'écriture du préfixe on peut définir un alias à l'aide de l'instruction `as` :

`import` *module* `as` *alias*

Exemple.

```
In [1]: import math as m          # Création d'un alias m
In [2]: m.sqrt(2)                 # Utiliser le préfixe m.
Out[2]: 1.4142135623730951
In [3]: m.cos(m.pi/4)
Out[3]: 0.7071067811865476
In [4]: m.tan(m.pi/4)
Out[4]: 0.9999999999999999
```

5.2 Le module `random`

Il permet de générer des nombres pseudo-aléatoires. Voici quelques fonctions fournies par le module `random` :

Fonctions du module `random`	
Générateurs aléatoires	
`random()`	Retourne un flottant aléatoire dans $[0,1[$.
`uniform(a,b)`	Avec `a` et `b` deux entiers ou flottants, renvoie un flottant aléatoire dans $[a,b[$. A le même effet que `a+(b-a)*random()`.
`randint(a,b)`	Prend en paramètre deux entiers `a` et `b` et renvoie un entier aléatoire dans $[[a,b]]$. A le même effet que `a+int((b-a+1)*random())`.
`randrange(a,b,k)`	Avec en paramètre 3 entiers `a`, `b` et `k`, renvoie un entier aléatoire dans `range(a,b,k)`. Le troisième paramètre `k` est optionnel et vaut par défaut 1.
aléas sur des séquences	
`choice(Seq)`	Renvoie un élément aléatoire dans une séquence non vide `Seq`. A le même effet que `Seq[randrange(0,len(Seq))]`.
`shuffle(List)`	Change aléatoirement l'ordre des éléments de la liste `List`.
`sample(Seq,k)`	Avec en paramètre une séquence `Seq` et un entier positif `k`, renvoie une liste, combinaison de `k` éléments aléatoirement choisie parmi la séquence `Seq`.

Ici aléatoirement signifie selon une loi (quasi-)uniforme.

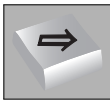
5.3 Le module `numpy`

Le module `numpy` permet de créer et de manipuler, des tableaux de nombres et de leur appliquer des opérations mathématiques courantes ; leur type est `numpy.ndarray`.

- Ce sont des tableaux homogènes : toutes les données doivent être de même type : que des flottants, ou des entiers, des booléens, complexes, etc.
- Ils sont non-redimensionnables : leur taille est fixée lors de leur déclaration et ne pourra plus être modifiée.

Les tableaux de `numpy` sont :

- séquentiels : on accède à leurs éléments par leur indice placé entre crochets. On peut leur appliquer du slicing (voir le paragraphe §1.5.5). Par contre les opérations + et * n'effectuent pas une concaténation et une duplication.
- mutables : après déclaration, on peut modifier leurs éléments.
- itérables : on peut les parcourir à l'aide d'une boucle **`for`** ; par exemple **`for x in`** `Tab:` déclare une variable `x` qui parcourt les éléments du `numpy.ndarray` par indice croissant.
- on peut appliquer à des tableaux de nombres les opérateurs arithmétiques, qui sont appliqués termes à termes.



On a coutume d'importer le module `numpy` avec pour alias `np`, par la commande **`import`** `numpy as np`.

5.3.1 Déclaration de tableaux `numpy`

• On peut créer des tableaux `numpy` unidimensionnels à l'aide des fonctions suivantes du module.

Création de tableaux unidimensionnels `ndarray`	
`zeros(p)`	crée un tableau de taille `p` rempli de zéros
`ones(p)`	crée un tableau de taille `p` rempli de uns
`empty(p)`	crée un tableau de taille `p` vide
`array(séquence)`	convertit en tableau une séquence de nombres
`arange(a,b,k)`	crée le tableau de tous les `a+k.N` entre `a` (inclu) et `b` (exclu). L'écart entre deux points est `k`
`linspace(a,b,n)`	crée le tableau des `n` valeurs régulièrement espacées entre `a` et `b` (inclus). L'écart entre 2 points est `(b-a)/(n-1)`
`logspace(a,b,n)`	crée un tableau de `n` valeurs allant de 10^a à 10^b étalés selon une échelle logarithmique ; la base, 10 par défaut, peut être changée avec l'option `base`.
`random.random(n)`	crée un tableau de `n` valeurs aléatoires choisies dans $[0,1[$.

• Les fonctions suivantes permettent de créer des tableaux bidimensionnels (matrices).

Création de tableaux bidimensionnels `ndarray`	
`array(L)`	crée une matrice à partir d'une liste `L` de listes de mêmes tailles
`zeros((p,q))`	crée un matrice de taille `(p,q)` remplie de zéros
`ones((p,q))`	crée un matrice de taille `(p,q)` remplie de uns
`empty((p,q))`	crée une matrice vide de taille `(p,q)`
`eye(n)`	crée une matrice Identité de taille `(n,n)`
`random.random((p,q))`	crée une matrice de taille `(p,q)` de valeurs aléatoires choisies dans $[0,1[$.

L'élément ligne `i` colonne `j` d'un tableau bidimensionnel `T` s'obtient par `T[i,j]` ou `T[i][j]`. On peut appliquer du slicing sur chacun des 2 indices. Avec un tableau bidimensionnel, le parcours au sein d'une boucle **`for`** s'effectue ligne par ligne.

On dispose aussi de la fonction `matrix()` qui prend en paramètre une liste de listes de mêmes tailles, et crée une matrice (instance de la classe `matrix`) ; les opérations de multiplication, puissance et inverse seront interprétées dans leur sens matriciel, ce qui n'est pas le cas des `ndarray` ou ces opérations sont effectuées terme à terme.

Les fonctions `array`, `empty`, `zeros` et `ones` permettent aussi, sur le même modèle, la création de tableaux de toutes tailles (tridimensionnelles, etc.).

5.3.2 Typage des données avec `dtype`

Lors de la création d'un tableau usuellement les données sont de type flottants. Sauf avec `array`, qui lorsque on lui passe en argument une séquence d'entiers, définit des données de type entier. On peut utiliser l'option `dtype` (data type) pour forcer la conversion dans le type numérique de son choix, par exemple :

```
In [1]: import numpy as np

In [2]: np.empty(3, dtype = int)
Out[2]: array([ 0, 0, 672901172428802])

In [3]: np.array((0,0,0))
Out[3]: array([0, 0, 0])

In [4]: np.array((0,0,0), dtype = float)
Out[4]: array([0., 0., 0.])

In [5]: np.array((0, 0, 0), dtype = complex)
Out[5]: array([ 0.+0.j,  0.+0.j,  0.+0.j])
```

L'option `dtype` ne s'applique qu'aux fonctions `array`, `zeros`, `ones` et `empty`. Elle ne s'applique pas aux fonctions `arange` et `linspace` dont les éléments sont toujours des flottants.

On peut utiliser l'option `dtype` en lui donnant pour valeur les types **`int`**, **`float`**, **`complex`**, **`bool`**, **`str`**, mais aussi avec les types définis dans `numpy` :

Types prédéfinis de `numpy`	
`int8`, `int16`, `int32`, `int64`	Entiers signés sur 8, 16, 32 ou 64 bits.
`uint8`, `uint16`, `uint32`, `uint64`	Entiers non-signés sur 8, 16, 32 ou 64 bits.
`float16`, `float32`, `float64`, `float128`	Flottants sur 16, 32, 64 ou 128 bits.
`complex64`, `complex128`, `complex256`	Complexes sur 64, 128 ou 256 bits.

La fonction de même nom, `dtype()` permet aussi de créer ses propres formats, à l'aide d'une chaîne de caractère définissant le type (`b` : booléen, `i` : entier signé, `u` : entier non-signé, `f` : flottant, `c` : complexe, `a` : caractère), par exemple `d = np.dtype('a10')` définit le type `d` consistant en une chaîne de 10 caractères.

5.3.3 Fonctions s'appliquant à un tableau `ndarray`

Les fonction suivantes de `numpy` s'appliquent à un `ndarray` de dimension quelconque.

On pourra trouver beaucoup d'autres fonctions de `numpy` dans sa documentation en ligne, à l'adresse :

`http://www.numpy.org`

Fonctions s'appliquant à un tableau	
Taille et nombre d'éléments	
`shape(T)`	Renvoie la taille du tableau `T`
`size(T)`	Renvoie le nombre d'éléments du tableau `T`
Copie d'un tableau	
`copy(T)`	retourne une copie du tableau `T`
`sort(T)`	retourne une copie ordonnée (dans le sens croissant) du tableau `T`. Pour un tableau bidimensionnel le tri s'effectue indépendamment sur chacune des lignes
`reshape()`	Prend en argument un tableau `T` et des dimensions et renvoie une copie de `T` redimensionée ; les dimensions doivent être compatibles avec le nombre d'éléments de `T`
Fonctions statistiques s'appliquant à un tableau numérique	
`mean(T)`	retourne la moyenne des valeurs du tableau `T`
`var(T)`	retourne la variance des valeurs du tableau `T`
`max(T)`	retourne le maximum des valeurs du tableau `T`
`min(T)`	retourne le minimum des valeurs du tableau `T`
`size(T)`	retourne le nombre d'éléments du tableau `T`

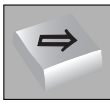
La fonction **`len`**`()` de `python` permet aussi de retourner le nombre d'éléments d'un tableau unidimensionnel, et le nombre de lignes d'un tableau bidimensionnel.

La fonction **`sum`**`()` s'applique aussi aux `ndarray` ; pour un tableau unidimensionnel de nombres, elle renvoie la somme de ses éléments ; pour un tableau bidimensionnel elle renvoie un `ndarray` unidimensionnel obtenu en faisant la somme sur chaque colonne.

5.3.4 Fonctions universelles

Le module `numpy` contient aussi toutes les fonctions mathématiques présentes dans `math` mais en versions *universelles*, c'est à dire pouvant s'appliquer à un tableau numérique terme à terme :

```
In [6]: v = np.linspace(0,np.pi,5)
In [7]: np.cos(v)
Out[7]:
array([  1.00000000e+00,   7.07106781e-01,   6.12323400e-17,
        -7.07106781e-01,  -1.00000000e+00])
```



Un avantage de tableaux numériques `numpy` est qu'on peut leur appliquer les opérations mathématiques qui sont exécutées membre à membre.

• Certaines fonctions peuvent ne pas s'appliquer à un tableau `numpy`, notamment si elles utilisent un branchement conditionnel ambigüe à résoudre pour un tableau. On peut les

transformer en une fonction universelle s'appliquant terme à terme à l'aide de la fonction `vectorize` :

```
In [8]: def minimum(x,y):
   ...:     if x < y:
                return x
   ...:     else:
                return y
   ....
In [9]: X = np.array([1, 2, 3, 4])
In [10]: minimum(X,2)
ValueError : The truth value of an array with more than one
             element is ambiguous
```

• Pour cela on peut créer une fonction universelle à l'aide de `np.vectorize()`.

```
In [11]: min_v = np.vectorize(minimum)  # version universelle
In [12]: min_v(X,2)
out[12]: array([1, 2, 2, 2])
```

• Voici un tableau des principales fonctions mathématiques (universelles) disponibles dans le module.

Constantes et fonctions Mathématiques du module `numpy`	
constantes (approximations)	
`pi`	le nombre π
`e`	le nombre e
`inf`	l'infini $+\infty$. C'est la valeur +INF des flottants
fonctions mathématiques	
`trunc(x,n)`	troncature de `x` à la `n`-ième décimale
`around(x,n)`	arrondi de `x` à la `n`-ième décimale
`rint(x)`	arrondi de `x` à l'entier le plus proche.
`floor(x)`	fonction partie entière : $x \mapsto \lfloor x \rfloor$.
`ceil(x)`	partie entière par valeur supérieure : $x \mapsto \lceil x \rceil$ c'est à dire le plus petit entier supérieur ou égal à x.
`trunc(x)`	troncature entière ; retourne l'entier obtenu en supprimant les décimales après la virgule.
`sqrt(x)`	fonction racine carrée $x \mapsto \sqrt{x}$; l'argument doit être un nombre positif
`exp(x)`	fonction exponentielle : $x \mapsto \exp(x)$.
`log(x)`	fonction logarithme néperien : $x \mapsto \ln(x)$. L'argument `x` doit être > 0.
`log2(x)`	fonction logarithme en base 2 ; `x` doit être > 0.
`log10(x)`	fonction logarithme en base 10 ; `x` doit être > 0.

`cos(x)`	fonction cos ; pour `x` exprimé en radians.
`sin(x)`	fonction sin ; pour `x` exprimé en radians.
`tan(x)`	fonction tan ; pour `x` exprimé en radians.
`sinc(x)`	fonction sinus cardinal sinc_π : $\mathrm{sinc}_\pi(x) = \sin(\pi x)/\pi x$, prolongée par continuité en 0.
`arccos(x)`	fonction arccos ; renvoie un nombre exprimé en radian.
`arcsin(x)`	fonction arcsin ; renvoie un nombre exprimé en radian.
`arctan(x)`	fonction arctan ; renvoie un nombre exprimé en radian.

`degrees(x)`	conversion de radians en degrés
`radians(x)`	conversion de degrés en radians

`cosh(x)`	fonction cosinus hyperbolique
`sinh(x)`	fonction sinus hyperbolique
`tanh(x)`	fonction tangente hyperbolique
`arccosh(x)`	fonction arccosinus hyperbolique
`arcsinh(x)`	fonction arcsinus hyperbolique
`arctanh(x)`	fonction arctangente hyperbolique

`real(x)`	partie réelle d'un nombre complexe
`imag(x)`	partie imaginaire d'un nombre complexe
`conj(x)`	conjugué d'un nombre complexe
`abs(x)`	module d'un nombre complexe
`angle(x)`	argument d'un nombre complexe

`sum(L)`	retourne la somme des éléments du tableau numérique `L`
`prod(L)`	retourne le produit des éléments du tableau numérique `L`

Lorsque `x` est un tableau numérique, la fonction est appliquée terme à terme.

Ces fonction s'appliquent aussi lorsque `x` est de type scalaire.

5.3.5 Calcul matriciel sur des `ndarray`

Les tableaux de `numpy` sont très pratiques pour le calcul matriciel. C'est le module à utiliser pour manipuler des tableaux bidimensionnels, et notamment pour tout ce qui relève du calcul matriciel.

On déclarera :

- Un `ndarray` unidimensionnel pour définir un vecteur (matrice colonne),
- Un `ndarray` bidimensionnel pour définir une matrice.

On peut alors effectuer sur les `ndarray` unidimensionnel et bidimensionnels, vus comme des vecteurs et matrices, toutes les opérations usuelles du calcul matriciel.

Les fonctions dans le tableau suivant permettent de leur appliquer les opérations matricielles basiques.

Opérations matricielles	
`dot(A,B)`	renvoie le produit matriciel `A`×`B` des 2 matrices `A` et `B` en argument
`vdot(u,v)`	renvoie le produit scalaire des 2 "vecteurs" `u` et `v`
`transpose()`	renvoie la matrice transposée de la matrice en argument
`A.T`	retourne la matrice transposée du tableau 2d `numpy A`
`concatenate((A,B),axis=i)`	retourne la matrice obtenue en concaténant `A` et `B` le long de l'axe `i`= 0 (ligne après ligne) ou 1 (colonnes après colonnes). Elles doivent avoir des tailles compatibles.
Fonctions s'appliquant à un tableau	
`copy()`	effectue une copie profonde d'un tableau
`mean()`	valeur moyenne d'un tableau
`var()`	variance des valeurs d'un tableau
`max()`	maximum des valeurs d'un tableau
`min()`	minimum des valeurs d'un tableau

• Le sous-module `numpy.linalg` contient d'autres fonctions du calcul matriciel plus spécifiques.

Fonctions de `numpy.linalg` pour le calcul matriciel	
`dot(A,B)`	retourne le produit matriciel des matrices `A` et `B`
`vdot(v,w)`	retourne le produit scalaire des vecteurs `v` et `w`
`matrix_power(A,n)`	retourne l'élévation à la puissance `n` (entier) de la matrice `A`
`inv(A)`	retourne l'inverse de la matrice `A`; la matrice doit être inversible
`det(A)`	retourne le déterminant de la matrice `A`
`matrix_rank(A)`	retourne le rang de la matrice `A`
`trace(A)`	retourne la trace de la matrice `A` (somme des éléments diagonaux)
`solve(A,b)`	résolution du système linéaire $A.X = b$

Les fonctions de `numpy.linalg` s'appliquent aussi bien avec en argument des `ndarray` que des séquences numériques. Toutes ces fonctions renvoient des `ndarray`.

5.3.6 Polynomes avec la classe `poly1d()` de `numpy`

La classe `poly1d` de `numpy` permet de manipuler des polynômes sous forme d'un tableau de ses coefficients.

• Par exemple, pour déclarer un polynôme arbitraire :

```
P = np.poly1d([a, b, c, d])
```

avec en argument une liste de coefficients `[a,b,c,d]` définit le polynôme :

$$P(X) = a.X^3 + b.X^2 + c.X + d \in \mathbb{C}[X]$$

La taille de la liste L passée en argument détermine le degré du polynôme $deg(P) =$`len(L)`-1.

Classe `poly1d()` **de** `numpy`	
`poly1d()`	avec en paramètre une liste de nombres définit un polynôme par ses coefficients.
`P.c`	retourne le tableau `numpy` des coefficients de `P`.
`P.order`	retourne le degré du polynôme `P`.
`P(x)`	évalue le polynôme `P` au nombre (complexe) `x`.
`P[i]`	où `i` est un paramètre entier, retourne le `i`ème coefficient de `P`.
`roots(P)`	retourne le tableau `numpy` des racines de `P`.
`poly1d.deriv(P)`	retourne le polynôme dérivée de `P`.
`poly1d.int(P)`	retourne un polynôme primitive du polynôme `P`.

Toutes les opérations `+,-,*,**` sont correctement implémentées sur les polynômes.

L'opérateur `/` effectue la division euclidienne de deux polynômes. Il retourne le couple constitué du quotient et du reste dans la division euclidienne. (Les opérations `//` et `\%` ne sont pas définies pour les polynômes.)

```
In [1]: import numpy as np
In [2]: P1 = np.poly1d([1,1])      # P1 = X+1
In [3]: P2 = np.poly1d([1,-1])     # P2 = X-1
In [4]: P = P1*P2
In [5]: P
Out[5]: np.poly1d([1., 0., -1.])  # P = P1xP2 = X^2-1
In [6]: P / P1
Out[6]: (poly1d([ 1.,  -1.]), poly1d([ 0.])) # P = P1x(X-1)+0
In [7]: Q = np.poly1d([1,0])
In [8]: P / Q   # Division euclidienne de P=X^2-1 par X
Out[8]: (poly1d([ 1.,  0.]), poly1d([-1.]))   # P = XxX+(-1)
```

5.3.7 La fonction `meshgrid()`

La fonction `meshgrid()` prend en argument deux tableaux unidimensionnel `L` et `C` et retourne deux matrices `ndarray`, de même type :

```
X, Y = np.meshgrid(L,C)
```

telles que :

- le nombre de colonne de `X` et `Y` est égal au nombre d'éléments de `L`,
- le nombre de lignes de `X` et `Y` est égal au nombre d'éléments de `C`,
- toutes les lignes de `X` sont identiques à `L`,
- toutes les colonnes de `Y` sont identiques à C.

Ainsi tous les couples `(L[i],C[j])` $\in L \times C$ sont égaux aux couples `(X[i,j],Y[i,j])`.

Par exemple :

```
In [1]: L = [1,2]
In [2]: C = [3,4,5]
In [3]: X, Y = meshgrid(L,C)
In [4]: X
array([[1, 2],
       [1, 2],
       [1, 2]]
In [5]: Y
array([[3, 3],
       [4, 4],
       [5, 5]]
```

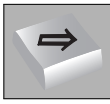
La fonction `meshgrid` est très utile pour la représentation graphique des fonctions réelles à deux variables.

5.4 Le module `matplotlib.pyplot`

Le sous-module `pyplot` du module `matplotlib` permet d'ouvrir une fenêtre graphique pour y tracer des courbes, nuages de points, histogrammes et autres figures dans le plan.

On l'importe par la commande :

```
import matplotlib.pyplot as plt
```



On a coutume d'importer le module `matplotlib.pyplot` avec l'alias `plt`. Ce module est à savoir utiliser pour le tracé de figures.

5.4.1 Tracé avec la fonction `plot`

Deux fonctions sont essentielles pour le tracé de courbes avec ou nuages de points avec `pyplot` ; ce sont :

- `plot()` pour le tracé de points ou de courbes, et
- `show()` pour provoquer l'affichage du graphique.

• Utiliser `plot` avec :

- en 1er argument la liste, séquence ou `ndarray` des abscisses des points,
- en 2ème argument la liste, séquence ou `ndarray` des ordonnées des points,
- On peut tracer plusieurs courbes en ajoutant d'autres arguments, listes/séquences des abscisses et ordonnées d'autres courbes.
- La fonction `plot` admet de multiples options.

Les options de la fonction `plot()` permettent de personnaliser le tracé, en précisant, couleur, épaisseur, style, etc. qui se substituent aux valeurs par défaut. On les passe en argument à la fonction, sous la forme `option = valeur`.

• Les options de `plot()` les plus utiles sont décrites dans le tableau qui suit :

Options de `plot()` : `plot(X, Y, option = ...)`	
`color =`	Couleur du tracé ; prend pour valeur une chaîne de caractère, par exemple `color = 'black'`. Par défaut vaut `'blue'` pour le premier tracé, `'green'` pour le deuxième, `'red'` pour le troisième, etc.
`linewidth =`	Épaisseur du tracé ; prend en paramètre un entier ou flottant, par exemple `linewidth = 2`. Vaut par défaut 1.
`visible =`	Affichage du tracé, valeur `True` ou `False` ; par défaut `True`.
`alpha =`	Transparence, valeur un flottant entre `0.0` et `1.0` (par défaut)
`linestyle =`	Style du tracé des traits entre les points : `'-'` traits continus (par défaut) `'--'` traits en pointillés longs `':'` traits en pointillés courts `'-.'` traits en pointillés longs/courts `'None'` aucun trait
`marker =`	Style des points : `''` aucun motif (par défaut) `'.'` petit points `'o'` gros points `'+'` croix droite `'x'` croix oblique `'*'` étoile
`label =`	Prend pour valeur la chaîne de caractère qui sera affichée dans la légende affichée par la fonction `legend()`

Toutes ces options sont généralement aussi accessibles pour les autres commandes de tracé.

5.4.2 Fonctions pour le tracé dans le plan

Les principales fonctions de `pyplot` pour le tracé dans le plan sont données dans le tableau suivant. Elles permettent le tracé de courbes, nuages de points, diagrammes et histogrammes. Elles permettent aussi d'ajouter titres, légendes, ou de définir les fenêtres et sous-fenêtres du tracé.

Fonctions de `matplotlib.pyplot` pour le tracé dans le plan	
Fonctions de tracé	
`plot(X,Y)`	Trace une courbe ou un nuage de point : prend en paramètre les tableaux des `X` abscisses et `Y` des ordonnées, éventuellement plusieurs, et plusieurs options.
`axhline()`	Trace l'axe des abscisses ; prend les mêmes options que `plot()`.
`axvline()`	Trace l'axe des ordonnées ; prend les mêmes options que `plot()`.

`fill(X,Y,color='black')`	Remplit la figure délimitée par la courbe fermée d'abscisses `X` et d'ordonnée `Y` par la couleur spécifiée par l'option `color`
`bar(X,Y)`	Trace un diagramme en baton de hauteurs `Y` au dessus des abscisses `X` de largeur définie par l'option `width` qui par défaut vaut 0,8.
`hist(X, range = (x0,x1), bins = n)`	Trace un histogramme du tableau `X` : chacune des n barres a pour hauteur le nombre d'occurrences d'éléments du tableau `X` dans l'un des `n` sous-intervalles régulierèment espacés de la plage de valeurs [`x0`,`x1`]. Avec l'option `normed = True`, c'est un histogramme de fréquence, normalisé pour que l'aire totale soit égale à 1.
`contour(X,Y,Z,N)`	Tracé dans le plan de courbes de niveaux d'une fonction de $\mathbb{R}^2$ dans $\mathbb{R}$. Les tableaux `X`, `Y` et `Z` sont 3 matrices de mêmes tailles, `N` est la liste des niveaux à tracer.

Gestion de l'affichage	
`clf()`	Effacement de la fenêtre active.
`show()`	Provoque l'affichage du graphique.
`draw()`	Provoque le rafraichissement de l'affichage.

Fenêtres et axes de coordonnées	
`grid()`	prend en paramètre un booléen, qui par défaut vaut `False`, et qui pour `True` affiche un quadrillage de la fenêtre délimité par les graduations des axes.
`axis([x0,x1,y0,y1])`	définit le domaine du plan à afficher.
`axis('equal')`	impose un repère orthonormé.
`xscale('log',basex=a)`	avec `a` un nombre, définit une échelle logarithmique de base `a` pour l'axe des abscisses.
`xticks()`	avec en paramètre une séquence de nombre définit les graduations et labels de l'axe des abscisses à afficher. Liste vide pour aucun affichage.
`yticks()`	idem pour l'axe des ordonnées.

Gestion des figures	
`figure(n)`	Rend active, ou crée si elle n'existe pas, la figure `n`. Le paramètre `n` peut être un entier ou une chaîne de caractère.
`subplot()`	rend active une sous-fenêtre de la fenêtre active. Prend en paramètres : Nombre de lignes, Nombre de colonnes, Numéro de Ligne, Numéro de Colonne.
`savefig(file)`	Sauvegarde la figure en un fichier image **`file`** (sans extension le format sera au format png).

Titres et légendes	
`title()`	ajoute un titre à la figure active. Le paramètre est une chaîne de caractère.
`legend()`	ajoute une légende à la figure active. Prend une option `loc =` pour la position de la légende, qui peut être : `'upper'`, `'upper left'`, `'upper right'`, `'center right'`, `'right'`, `'bottom right'`, etc.
`xlabel()`	rajoute une description sur l'axe des abscisses ; prend en paramètre une chaine de caractère.
`ylabel()`	rajoute une description sur l'axe des ordonnées ; prend en paramètre une chaine de caractère.

5.4.3 Utilisation de commandes TEX dans les chaînes de `pyplot`

Dans les chaines de caractères passés en argument à des fonctions de `pyplot`, comme `title`, `legend`, `xlabel`, etc..., on peut insérer certaines commandes TEX qui permettent une typographie parfaite pour les symbôles et équations mathématiques. Pour cela les équations mathématiques doivent être délimitées entre deux symbôles dollar : $...$.

Voici une table de quelques commandes TEX : Les commandes TEX débutent toutes par un caractère antislash \.

Quelques commandes TEX	
`\pi`, `\infty`	Symbôles π, ∞.
`a^{b}`, `a_{b}`, `\sqrt{a}`	Exposants, indices et radicaux : a^b, a_b, $\sqrt{a}$.
`\tilde{a}`, `\bar{a}`, `\hat{a}`, `\vec{a}`	Accents $\tilde{a}$, $\bar{a}$, $\hat{a}$, $\vec{a}$.
`a'`, `a''`, `\dot{a}`, `\ddot{a}`, `\partial`	Symbôles de dérivation a', a'', $\dot{a}$, $\ddot{a}$, ∂.
`\cos`, `\sin`, `\exp`, `\ln`, etc.	Les fonctions cos, sin, exp, ln, etc.
`\mathrm{arccos}`, `\mathrm{arcsin}`, ...	Pour d'autres fonctions non prédéfinies
`\alpha`, `\beta`, ..., `\omega`	Lettres grecques minuscules α, β, ..., ω
`\Phi`, `\Psi`, `\Xi`, etc.	Lettres grecques majuscules Φ, Ψ, Ξ, etc.
`=`, `\approx`, `\equiv`, `\simeq`	Symbôles égalitaires : $=$, $\approx$, $\equiv$, $\simeq$.
`<`, `>`, `\leq`, `\geq`	Symbôles inégalitaires : $<$, $>$, $\leqslant$, $\geqslant$.
`\times`, `\cdot`	Opérateurs de multiplication $\times$ et $\cdot$.
`\ldots`, `\cdots`	Points de suspension : $\ldots$ et $\cdots$.
`\mapsto`, `\rightarrow`, `\longrightarrow`, `\Rightarrow`, etc.	Flèches : $\mapsto$, $\rightarrow$, $\longrightarrow$, $\Rightarrow$.
`\lim_{n \rightarrow \infty}`	Limite $\lim_{n\to\infty}$
`\sum_{a}^{b}`, `\prod_{a}^{b}`	Symbôles de sommation $\sum_a^b$ et produit $\prod_a^b$.
`\int_{a}^{b}`	Symbôle intégral $\int_a^b$.

On trouvera de très nombreuses autres commandes TEX en ligne, par exemple à l'adresse :

`https://fr.wikipedia.org/wiki/Aide:Formules_TeX`

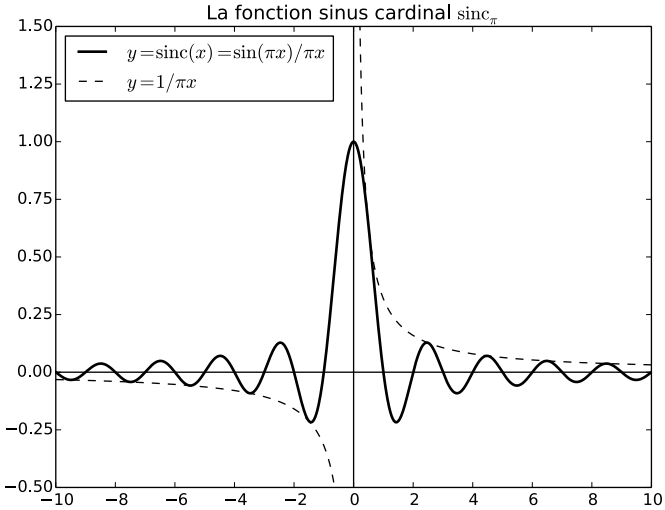
Attention, `pyplot` ne les comprend pas toutes.

5.4.4 Exemple

On a tracé dans le plan, les courbes représentatives des deux fonctions

$$\mathrm{sinc}_\pi(x) = \frac{\sin(\pi x)}{\pi x} \quad \text{et} \quad x \mapsto \frac{1}{\pi x}\,.$$

On a produit le tracé suivant :



Ce graphique a été obtenu à l'aide du code qui suit. Son exécution, provoque l'ouverture d'une fenêtre graphique avec le tracé ci-dessus.

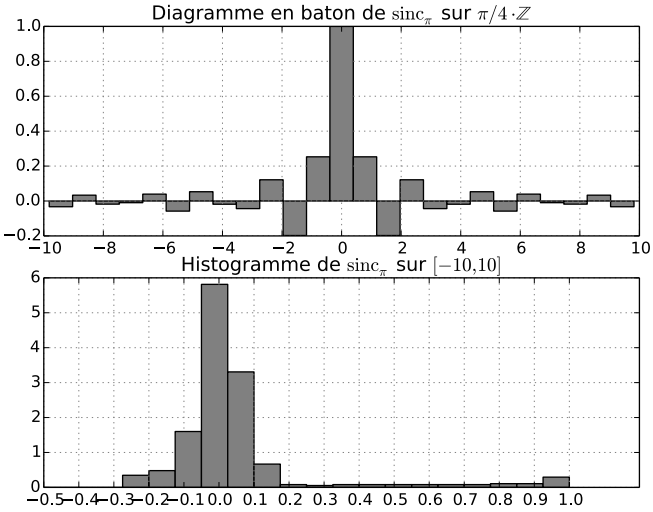
```
# importation des modules
import numpy as np       # Numpy pour les ndarray et fonctions
import matplotlib.pyplot as plt # Pyplot pour le tracé

# Définition des tableaux
# Courbe y = sinc(x)
xmin, xmax = -10, 10
X = np.linspace(xmin,xmax,1000)
Y = np.sinc(X)

# Courbe y = 1/pi.x
Xn = np.arange(xmin,0,0.01)
Yn = 1/Xn/np.pi
```

```
# Tracé des courbes
plt.figure(1)
plt.axhline(color='black')
plt.axvline(color='black')

plt.plot(X,Y,linewidth = 2, color = 'black',\
         label = '$y=\mathrm{sinc}(x)=\sin(\pi x)/\pi x$')
plt.plot(Xn,Yn, linestyle = '--', color='black',\
                                  label = '$y=1/\pi x$')
plt.plot(-Xn,-Yn, linestyle = '--', color='black')

plt.axis([-10,10,-0.5,1.5])
plt.xticks([k for k in range(xmin,xmax+1,2)])
plt.yticks([k for k in np.arange(-0.5,1.51,0.25)])
plt.legend(loc = 'upper left')
plt.title('La fonction sinus cardinal $\mathrm{sinc}_\pi$')
plt.show()
```

On a ensuite tracé le diagramme à baton de la fonction sinc_π au dessus des points de $\frac{\pi}{4}\cdot\mathbb{Z}\cap[[-10,10]]$, ainsi que l'histogramme des fréquences en vingt sous-intervalles de l'image $[-0,5,1]$ de la fonction ; ces deux graphiques ont été tracés dans la même figure, grâce à la fonction `subplot`, qui l'a divisé verticalement en deux sous-fenêtres graphiques.

```
# Diagramme en baton et Histogramme des fréquences de sinc(x)
# Tableaux pour le diagramme en baton
amin = np.ceil(xmin/(np.pi/4))
amax = np.floor(xmax/(np.pi/4))
Xb = np.arange(amin*np.pi/4, amax*np.pi/4+0.1,np.pi/4)
Yb = np.sinc(Xb)

plt.figure(2)
plt.subplot(2,1,1)       # Diagramme en baton
plt.title('Diagramme en baton de $\mathrm{sinc}_\pi$ sur\
                                  $\pi/4\cdot\mathbb{Z}$')
plt.bar(Xb-np.pi/8,Yb,width=np.pi/4,color='gray')
plt.grid(True,color='gray')
plt.xticks(range(-10,11,2))

plt.subplot(2,1,2)       # Histogramme
plt.title('Histogramme de $\mathrm{sinc}_\pi$ sur $[-10,10]$')
plt.grid(True, color='gray')
plt.hist(Y, range=(-0.5,1), bins = 20, normed = True,\
                                       color='gray')
plt.xticks(np.arange(-0.5,1.1,0.1))
plt.show()
```

On obtient dans une nouvelle fenêtre graphique, le tracé suivant :



5.4.5 Tracé de courbes de niveaux avec `contour`

La fonction `contour()` permet de tracer les lignes de niveau d'une application $f : \mathbb{R}^2 \longrightarrow \mathbb{R}$.

Les deux premiers paramètres de `contour(X,Y,Z,N)` sont deux matrices `X` et `Y` de même tailles, dont les lignes de `X` et les colonnes de `Y` contiennent respectivement les différentes abscisses (x_i) et ordonnées (y_j) du domaine où se situe le tracé. Le troisième paramètre `Z` est une matrice de même taille que `X` et `Y` dont l'élément `Z[i,j]` contient f(`X[i,j]`,`Y[i,j]`) c'est à dire la valeur $f(x_i, y_j)$. La séquence `N` contient les niveaux à tracer, c'est à dire que l'on trace les courbes $f^{-1}(n_k)$ pour toute valeur n_k dans `N` (dans le domaine rectangulaire du plan $[\min_i x_i, \max_i x_i] \times [\min_j y_j, \max_j y_j]$).

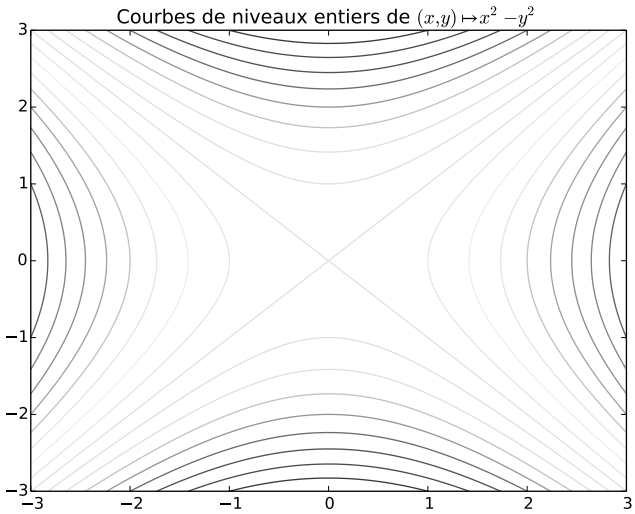
Pour appliquer la fonction `contour()` on utilisera la fonction `meshgrid()` de `numpy` qui permettra des constituer les deux premiers paramètres `X` et `Y` à partir de tableaux unidimensionnels des valeurs (x_i) et (y_j).

Par exemple, traçons dans le domaine $[-3,3] \times [-3,3]$ du plan les courbes de niveaux -9, -8, $\ldots, 8, 9$ de la fonction $f(x,y) = x^2 - y^2$.

```
import numpy as np
import matplotlib.pyplot as plt

a = 3
x = np.linspace(-a,a,1000)
y = np.linspace(-a,a,1000)
X, Y = np.meshgrid(x,y)
f = lambda x, y : x**2-y**2       # f(x,y)=x^2-y^2
Z = f(X,Y)
N = range(-9,10)
```

```
plt.figure()
plt.title('Courbes de niveaux entiers de\
                          $(x,y)\mapsto x^2-y^2$')
plt.contour(X,Y,Z,N)
plt.show()
```

On obtient le graphique :



5.4.6 Tracé d'une surface dans l'espace

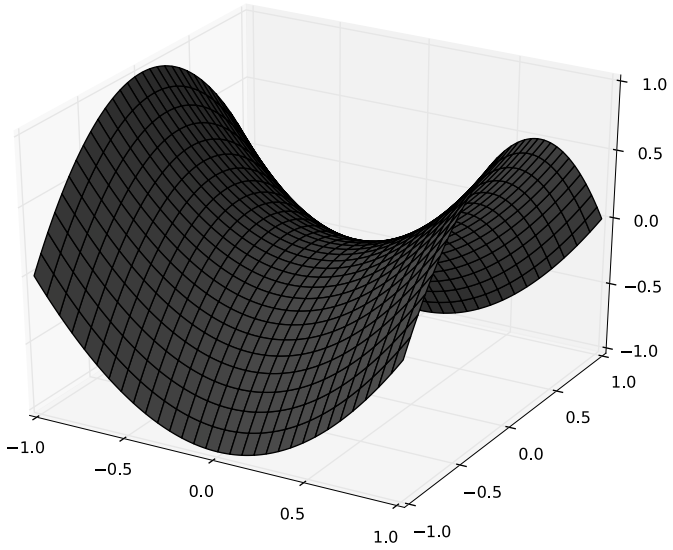
On peut représenter graphiquement dans l'espace muni d'un repère orthonormé, une fonction $f : \mathbb{R}^2 \longrightarrow \mathbb{R}$ par la surface :

$$\mathscr{S}_f = \left\{(x, y, z) \in \mathbb{R}^3 \mid (x, y) \in \mathscr{D}_f,\ z = f(x, y)\right\}$$

Pour cela on utilisera avec `pyplot` la fonction `Axes3D()` que l'on aura importée du module `mpl_toolkits.mplot3d`, et sa méthode `plot_surface()` ; `plot_surface()` prend en arguments, trois matrices de mêmes tailles, `X`, `Y` et `Z`, contenant les coordonnées x, y et z des points de la surface ; les matrices `X` et `Y` seront crées grâce à la fonction `meshgrid()` de `numpy`. Par exemple, traçons la surface représentative de $f(x, y) = x^2 - y^2$ au-dessus du domaine $[-1, 1] \times [-1, 1]$ du plan.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig3d = Axes3D(plt.figure())     # Création d'une figure 3D
f = lambda x,y : x**2-y**2       # f(x,y)=x^2-y^2
```

```
x = np.linspace(-1,1,300)
y = np.linspace(-1,1,300)
X, Y = np.meshgrid(x,y)          # Abscisses et ordonnées
Z = f(X,Y)
fig3d.plot_surface(X,Y,Z)        # Tracé
plt.show()
```



5.4.7 Tracé d'une courbe dans l'espace

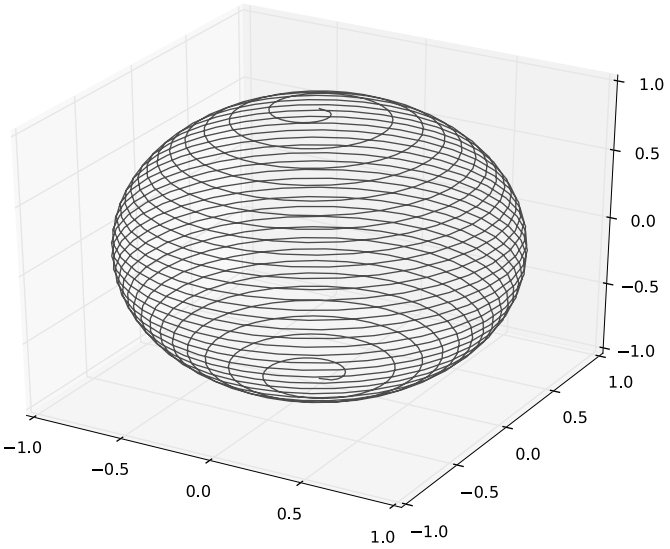
Pour tracer une courbe dans l'espace on utilise encore la fonction `Axes3D()` importée de `mpl_toolkits.mplot3d` et sa méthode `plot()` avec en paramètres les trois séquences des coordonnées x, y, z des points du tracé.

Exemple. Pour représenter la courbe d'une spirale sur une sphère :

$$\begin{array}{ccc} [0,60\pi] & \longrightarrow & \mathbb{R}^3 \\ t & \longmapsto & (x(t),y(t),z(t)) \end{array} \quad \text{avec} \quad \begin{cases} x(t) = \sqrt{1-\left(1-\dfrac{t}{30\pi}\right)^2} \times \cos(t) \\ y(t) = \sqrt{1-\left(1-\dfrac{t}{30\pi}\right)^2} \times \sin(t) \\ z(t) = \dfrac{t}{30\pi} - 1 \end{cases}$$

```
from numpy import pi, cos, sin, sqrt, linspace
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig3d = Axes3D(plt.figure())
tmax = 60*pi
```

```
T = linspace(0,tmax,1000)
X = sqrt(1-(2*T/tmax-1)**2)*cos(T)
Y = sqrt(1-(2*T/tmax-1)**2)*sin(T)
Z = -1+2*T/tmax
fig3d.plot(X,Y,Z)
plt.show()
```



Faire varier la valeur de `tmax` modifiera le tracé de la façon suivante : `tmax`$/2\pi$ est le nombre de tours qu'effectue la courbe autour de l'axe vertical de la sphère.

1) ■ Opérateurs de division euclidienne

1. On souhaite écrire un script permettant de convertir une durée exprimée en secondes en années, jours, heures, minutes et secondes.

Il demandera à l'utilisateur de saisir le nombre (entier) de secondes, et écrira dans la console la durée exprimée en années, jours, minutes, secondes. (On considérera qu'une année dure 365 jours.)

Par exemple :

```
Nombre de secondes à convertir : 100000000
3 années
62 jours
9 heures
46 minutes
40 secondes
```

a) Pour une durée de N secondes comment obtenir le nombre A d'années ? comment obtenir le nombre de secondes restantes au delà de A années ?

b) Mêmes questions pour les jours, les heures, et les minutes.

c) Écrire le code du script demandé.

d) Modifier le script pour que l'emploi du singulier/pluriel (les 's') soit correct. Par exemple :

```
Nombre de secondes à convertir : 3700
1 heure
1 minute
40 secondes
```

(sans 's' à heure et minute).
On prendra soin de ne pas allonger exagérément le code.

2. On souhaite écrire un script pour déterminer les entiers naturels qui sont égaux à la somme des cubes des chiffres de leur écriture décimale.

a) Soit `a` une variable de type entier ; quelle expression renvoie son chiffre des unités ? Comment obtenir son chiffre des dizaines ? des centaines ?

b) Écrire une fonction `sommeCube(n)` qui prend en paramètre un entier positif `n` et qui renvoie la somme des cubes de ses chiffres.

c) Écrire un script qui permettra d'obtenir tous les entiers entre 1 et 10 000 qui sont égaux à la somme des cubes de leurs chiffres. Par exemple :

$$153 \ : \ 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$$

1.a-b. Utiliser les opérateurs `//` et `%`.
1.c. Convertir en entier la donnée retournée par **`input()`**. Utiliser plusieurs branchements conditionnels.

1.d. Utiliser une boucle **for** après avoir déclaré deux listes des entiers et des chaînes de caractères appropriées.

1.a. Le nombre d'année s'obtient par l'expression : `N // (365*24*3600)`.

Le nombre de secondes restantes est alors : `N % (365*24*3600)`.

1.b. Pour les jours : `N // (24*3600)` et `N % (24*3600)`.

Pour les heures : `N // 3600` et `N % 3600`.

Pour les minutes : `N // 60` et `N % 60`.

1.c. Code du script :

```
N = int(input('Nombre de secondes à convertir :'))
annee, jour, heure, minute = 60*60*24*365, 60*60*24, 60*60,60
if N >= annee:
    print(N//annee, "annees")
    N = N % annee
if N >= jour:
    print(N//jour, "jours")
    N = N % jour
if N >= heure:
    print(N//heure, "heures")
    N = N % heure
if N >= minute:
    print(N//minute, "minutes")
    N = N % minute
if N > 0:
    print(N, "secondes")
```

1.d. Code du script amélioré, avec une orthographe correcte. Pour ne pas allonger le code, on utilise une boucle **for** et deux listes contenant, pour la première, les nombres de secondes par année, jour, heure et minute, et pour la seconde, les chaînes de caractères correspondantes.

```
N = int(input('Nombre de secondes à convertir :'))
a, j, h, m = 60*60*24*365, 60*60*24, 60*60,60
Lv = [a, j, h, m]
Lc = ['annee', 'jour', 'heure', 'minute']
for k in range(4):
    var = Lv[k]
    if N >= var:
    n = N//var
    print(n, Lc[k], end='')
    if n== 1:
        print('')
    else:
```

```
        print('s')
    N = N % var
if N > 0:
    print(N,"seconde", end='')
    if N != 1:
        print('s')
```

2.a. Le chifre des unités de `a` s'obtient par l'expression : `a%10`.

Le chiffres des dizaines de `a` s'obtient par l'expression : `(a//10)%10`.

Le chiffre des centaines de `a` s'obtient par l'expression : `(a//100)%10`.

2.b. Code de la fonction `sommeCube()`.

```
def sommeCube(n):
    S = 0
    while n > 0:
        S += (n%10)**3
        n = n//10
    return S
```

2.c. Code du script. Le mieux est de constituer une liste.

```
L = []
for n in range(1001):
    if n == somCube(n):
        L.append(n)
print(L)
```

ou encore par compréhension de liste :

```
L = [n for n in range(1001) if n == somCube(n)]
```

On obtient : `[0, 1, 153, 370, 371, 407]`.

2) ■ Correction de programme

On rappelle que pour une boucle **`while`**, on appelle **variant de boucle** une expression qui est minorée, ne prend que des valeurs entières, et décroît strictement à chaque passage dans la boucle. Exhiber un variant de boucle démontre la terminaison de la boucle **`while`**, c'est à dire qu'elle finira par s'arrêter.

On rappelle qu'on appelle **invariant de boucle**, une expression, ou proposition, dont la valeur demeure inchangée à chaque passage dans la boucle. Exhiber un invariant de boucle approprié est utilisé pour montrer la correction d'une boucle, c'est à dire qu'à sa sortie on a bien le résultat qui était attendu.

1. On considère la fonction `mystere1()` suivante; elle prend en paramètre deux entiers strictement positifs `a` et `b` :

```
def mystere1(a,b):
    c = 0
    while a >= b:
        a = a - b
        c += 1
    return c, a
```

a. Justifier de la terminaison de la fonction en donnant un variant de boucle.

b. Déterminer parmi les expressions suivantes laquelle est un invariant de boucle.

- `a + b`
- `a + b * c`
- `a * c`
- `a`
- `a - b`
- `a - b * c`
- `b * c`
- `c`

Justifier.

c. Que représentent pour `a` et `b` les valeurs renvoyées par l'appel de `mystere1(a,b)` ? Justifier.

2. On considère la fonction `mystere2()` suivante; elle prend en paramètre deux entiers strictement positifs `a` et `b` :

```
def mystere2(a,b):
    while a != b:
        if a > b:
            a -= b
        else:
            b -= a
    return a
```

a) Pour justifier de la terminaison de la fonction, préciser parmi les expressions suivantes, laquelle est un variant de boucle :

- `a`
- `a - b`
- `max(a,b)`
- `b`
- `b - a`
- `min(a,b)`

Justifier.

b) Démontrer que "L'ensemble des diviseurs communs à `a` et `b`" est un invariant de boucle.

c) Que renvoie `mystere2(a,b)` ? Justifier.

3. On considère la fonction `mystere3()` suivante ; elle prend en paramètre deux entiers positifs `a` et `b` :

```
def mystere3(a,b):
    c = 1
    while b != 0:
        if b % 2 == 1:
            c *= a
        a = a*a
        b = b//2
    return c
```

a) Justifier la terminaison de la fonction en donnant un variant de boucle.

b) Parmi les expressions suivantes laquelle est un invariant de boucle ?

- `c + a ** b`
- `a + b * c`
- `a * c`
- `a + b`
- `c * a ** b`
- `a - b * c`
- `b * c`
- `b + c`

Justifier.

c) Que représente pour `a` et `b` la valeur renvoyée par l'appel de `mystere3(a,b)` ?

d) Quelle est la complexité de cette fonction en fonction de `b` ?

1.a., **2.a.**, **2.b.**, **3.a.** Noter a_k, b_k, c_k les valeurs prises par les variables `a`, `b` et `c` après le k-ième passage dans la boucle, et donner des relations de récurrence.
1.c., **2.c.**, **3.c.** Noter a, b, c les valeurs initiales des variables `a`, `b` et `c` et a', b', c' leurs valeurs finales. Appliquer les invariants de boucle établis.

Dans tout ce qui suit, on notera a_k, b_k, c_k les valeurs prises par les variables `a`, `b` et `c` après le k-ième passage dans la boucle, a, b, c leurs valeurs initiales, et a', b', c' leurs valeurs finales.

1.a. L'expression `a` est un variant de boucle, puisqu'à chaque passage dans la boucle on lui retranche l'entier strictement positif `b`. D'où la terminaison de la boucle et donc de la fonction.

1.b. L'invariant de boucle est `a+b*c`. En effet les instructions dans la boucle montrent qu'on a les relations de récurrences :

$$a_{k+1} = a_k - b_k \quad ; \quad b_{k+1} = b_k \quad ; \quad c_{k+1} = c_k + 1$$

Ainsi :

$$a_{k+1} + b_{k+1} \times c_{k+1} = a_k - b_k + b_k \times (c_k + 1) = a_k + b_k \times c_k$$

1.c. En particulier : $a' + b' \times c' = a + b \times c$. Puisque $c = 0$: $a' + b' \times c' = a$ et puisque à l'arrêt de la boucle $a' < b'$, les valeur c' et a' du tuple retourné sont respectivement le quotient et le reste dans la division euclidienne de `a` par `b`.

2.a. Le variant de boucle est **`max(a,b)`**. En effet on a la relation de récurrence :

$$\max(a_{k+1}, b_{k_1}) = \max(a_k, b_k) - \min(a_k, b_k)$$

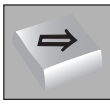
car à chaque passage dans la boucle on soustrait au plus grand de `a` et `b`, l'autre : `a=a-b` ou `b=b-a`. De plus puisqu'à chaque passage $a_k \neq b_k$, et $a_0, b_0 > 0$, une récurrence immédiate montre que $\min(a_k, b_k) > 0$.
Ainsi la quantité $\max(a_k, b_k)$ décroît strictement à chaque passage dans la boucle et reste positive (et entière), **`max`**`(a,b)` est donc un variant de boucle. D'où la terminaison de la boucle et donc de la fonction.

2.b. On a soit $a_{k+1} = a_k$, $b_{k+1} = b_k - a_k$ soit $a_{k+1} = a_k - b_k$, $b_{k+1} = b_k$. Or tout diviseur commun à α et β est aussi diviseur commun à α et $\alpha - \beta$; réciproquement tout diviseur commun à α et $\alpha - \beta$ est diviseur commun à α et β. Ainsi a_k, b_k et a_{k+1}, b_{k+1} ont mêmes diviseurs communs. Ainsi l'ensemble des diviseurs communs à `a` et `b` est un invariant de boucle.

2.c. C'est en particulier vrai pour le plus grand des diviseurs communs :

$$pgcd(a_k, b_k) = pgcd(a_{k+1}, b_{k+1}) \ .$$

À l'issue de la boucle $a' = b' \neq 0$, ainsi $pgcd(a', b') = a'$, et c'est ce que renvoie la fonction. D'après l'invariant de boucle, la valeur renvoyée est égale à $pgcd(a, b)$, c'est à dire au plus grand commun diviseur des paramètres `a` et `b` passés en argument. La fonction calcule le $pgcd$ de ses arguments.



Cet algorithme de calcul du $pgcd$ s'appelle l'*algorithme des différences*.

3.a. L'expression `b` est un variant de boucle, puisque $b_k > 0$ et b_{k+1} est le quotient de b_k par 2.

3.b. L'invariant de boucles est `c * a ** b`. En effet on a les relations de récurrence :

$$a_{k+1} = a_k^2 \quad ; \quad b_{k+1} = \lfloor b_k/2 \rfloor \quad ; \quad c_{k+1} = \begin{cases} c_k & \text{si } b_k \text{ est pair} \\ c_k \times a_k & \text{si } b_k \text{ est impair} \end{cases}$$

Si b_k est pair, alors $b_{k+1} = b_k/2$ et :

$$c_{k+1} \times a_{k+1}^{b_{k+1}} = c_k \times a_k^{2b_k/2} = c_k \times a_k^{b_k}$$

Si b_k est impair, alors $b_{k+1} = (b_k - 1)/2$ et :

$$c_{k+1} \times a_{k+1}^{b_{k+1}} = c_k \times a_k \times a_k^{2(b_k-1)/2} = c_k \times a_k^{b_k}$$

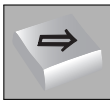
3.c. D'après l'invariant de boucle, on a $c \times a^b = c' \times (a')^{b'}$, or initialement $c = 1$ et en sortie de boucle $b' = 0$. Ainsi on a la relation

$$a^b = c'$$

La fonction renvoie donc `a` élevé à la puissance `b`.

3.d. En dehors et hors de la boucle ne s'exécutent que des opérations élémentaires. La complexité de la fonction a donc même ordre que le nombre de passage dans la boucle **`while`**. Puisqu'on a la relation de récurrence $b_{k+1} = \lfloor b_k/2 \rfloor$ et que la boucle s'arrête lorsque `b` est

nul, ce nombre de passage est $\lfloor \log_2(b) \rfloor + 1$. La complexité de cet algorithme d'élévation à la puissance est donc logarithmique, en $O(\log(b))$.



Cet algorithme rapide d'élévation à la puissance, s'appelle l'*algorithme d'exponentiation rapide*. Nous en donnerons une autre implantation au chapitre 5.

3) ■ Lecture d'un fichier hexadécimal

1. Ecriture d'un entier naturel dans une base.

Dans cet exercice on aura déclaré une variable globale de type chaîne de caractères :

```
chiffres = '0123456789ABCDEF'
```

a) Ecrire une fonction `base(n,a)` qui prend en paramètre un nombre entier positif `n` et un entier `a` entre 2 et 16, et qui retourne une chaine de caractères contenant l'écriture de l'entier `n` en base `a`.
Par exemple `base(27,16)` renverra l'écriture du nombre 27 en base 16 : `'1B'`.

b) Vérifier son fonctionnement en obtenant l'écriture du nombre 255 en bases 2, 10 et 16.

c) Ecrire la fonction `index` prenant en paramètre un caractère apparaissant dans la chaine `chiffres` et qui renvoie l'indice où il apparait. A quoi correspond cet indice ?

d) Ecrire la fonction inverse `valeur(ch,a)` prenant en paramètre un entier `a` entre 2 et 16 et une chaine de caractères `ch` représentant l'écriture en base `a` d'un nombre entier `n`, et qui renvoie l'entier `n`.

2. Lecture d'un fichier hexadécimal.

Un fichier texte `texteHexa.txt`, placé dans le répertoire de travail, contient un texte encodé de la façon suivante :

- chaque caractère du texte original a été remplacé par son code dans la table ASCII, qui est un entier entre 0 et 255.
- ce code a été écrit en hexadécimal sur deux caractères. Par exemple le caractère `z` a pour code ASCII 122 qui s'écrit en base 16 : 7A. Plutôt que le caractère `z`, a été écrit dans le fichier la chaîne de 2 caractères `'7A'`.

a) Ecrire le code permettant de récupérer le contenu du fichier dans une variable `contenu` de type chaîne de caractère et globale.

b) Ecrire le code permettant de décoder, d'afficher dans la console, puis de sauvegarder dans un autre fichier texte, tout le texte contenu dans le fichier `texteHexa.txt`.

On utilisera la fonction **`chr()`** qui avec en paramètre un entier `n` entre 0 et 255, renvoie le caractère de la table ASCII de code `n`. On devra utilisera une fonction appropriée écrite en 1).

1.a. Adapter l'algorithme de conversion décimal binaire donné au paragraphe §3.3.3. en remplaçant 2 par a. Les chiffres sont à récupérer dans la chaîne `chiffres`.
1.d. Adapter l'algorithme de conversion de binaire vers décimal écrite au paragraphe §3.2.4 en remplaçant 2 par a. Il faudra utiliser la fonction `index`.
2.b. Parcourir le contenu du fichier par pas de 2, et en extraire des chaînes de 2 caractères.

1.a. Fonction `base(n,a)`.

```
def base(n,a):
    if n == 0:
        return '0'
    ch = ''
    while n>0:
        r = n%a
        ch = chiffres[r] + ch
        n = n//a
    return ch
```

1.b. Test de la fonction `base`.

```
In [2]: base(255,2), base(255,10), base(255,16)
Out[2]: ('11111111', '255', 'FF')
```

1.c. Fonction `index`. L'indice renvoyé correspond à la valeur numérique du chiffre.

```
def index(c):
    for k in range(len(chiffres)):
        if c == chiffres[k]:
            return k
```

1.d. Fonction `valeur` qui convertit une écriture en base a vers l'entier qu'elle représente.

```
def valeur(ch,a):
    n = 0
    for x in ch:
        n *= a
        n += index(x)
    return n
```

2.a. Récupération du contenu du fichier texte.

```
f = open('texteHexa.txt','r')
contenu = f.read()
f.close()
```

2.b. Décodage et sauvegarde.

```
texte = ""
for k in range(0,len(contenu),2):
    hexa = contenu[k:k+2]
    texte += chr(valeur(hexa,16))
print(texte)

f = open('texteClair.txt','w')
f.write(texte)
f.close()
```

4) ■ Annuaire téléphonique

Un fichier texte nommé `annuaire.txt` contient les données d'un annuaire téléphonique ; chaque ligne contient : numéro de la ligne, NOM, PRENOM et numéro de téléphone portable de la personne concernée, toutes séparées d'une seul espace, et sans accent. Les noms ne sont pas classées dans l'ordre alphabétique (afin de permettre de rajouter des entrées en fin de fichier).

L'affichage dans un éditeur de texte produirait pour ses premières lignes :

1 ABAS JULIE 0675762133
2 ABBE ANDRE 0627185431
⋮
1027 BERTRAND MARIE 0731244566
...

On suppose le fichier placé dans le répertoire de travail. Le but de l'exercice est d'écrire un programme qui permette à l'utilisateur de récupérer dans le fichier le numéro de telephone d'une personne à partir de ses noms et prénoms.

1. Ecrire une fonction `indiceMotif(ligne,motif)` prenant en paramètre deux chaines de caractère : `ligne` et `motif`. La fonction cherchera si la chaine `motif` apparaît comme sous-chaine dans `ligne` et renverra :
   - L'indice dans `ligne` où apparaît (le premier caractère de) `motif`.
   - -1 si `motif` n'apparaît pas dans `ligne`.

   Par exemple dans la chaine `"1 ABAS JULIE 0675762133"`, le motif `"ABAS JULIE"` apparaît à l'indice 2.
2. En déduire le code de la fonction `numero()` prenant en paramètre une chaine de caractère qu'on supposera de la forme `"NOM PRENOM"`. La fonction :
   - ouvrira le fichier `annuaire.txt`,
   - affichera toutes les lignes du fichier où apparaît "NOM PRENOM",
   - Refermera le fichier.
3. Améliorer le code de la fonction `numero()` pour qu'elle n'affiche plus toutes les lignes où apparaît `"NOM PRENOM"`, mais seulement les numéros de telephone y figurant.

Pour que les deux fonctions précédentes fonctionnent il faudra passer en paramètre une chaine de la forme `NOM PRENOM`; par exemple `numero("Abas Julie")` n'affichera rien. On souhaite améliorer la fonction pour qu'elle ne soit plus sensible à la casse (majuscule/-minuscule). Pour cela on a défini dans le programme les deux variables globales que l'on pourra utiliser dans la suite :

```
minus = "abcdefghijklmnopqrstuvwxyz-' "
majus = "ABCDEFGHIJKLMNOPQRSTUVWXYZ-' "
```

4. Ecrire une fonction `majuscule()` qui prend en paramètre une chaine de caractères constituée de lettres minuscules ou de caractères tiret `"-"`, apostrophe `"'"` ou espace `" "` et qui renvoie la chaine avec lettres minuscules converties en majuscules. On pourra utiliser la méthode `index` des chaine de caractère.
5. Ecrire une fonction `recherche()` qui :
   - ne prend aucun paramètre,
   - demande à l'utilisateur de saisir le nom,
   - demande à l'utilisateur de saisir le prénom,
   - affiche tous les numéros de téléphone figurant dans le fichier associé à ce nom et prénom.
   - la saisie des noms et prénoms pourra se faire en minuscule/majuscule : la recherche ne sera plus sensible à la casse.

Lorsque la recherche n'a pas trouvé le nom recherché, cela peut être à cause d'une faute de frappe. On souhaiterait que le programme affiche dans ce cas nom, prénom et numéro de téléphone des personnes dont le nom prénom ne diffère de celui recherché que d'un seul caractère inexact ; par exemple `ABAC JULIE` au lieu d'`ABAS JULIE` (par contre `ABAS JULI` ne renverra rien).

6. Modifier la fonction `indiceMotif(ligne,motif)` pour écrire une fonction `indiceMotif_m(ligne,motif)` de façon ce que la fonction renvoie l'indice où `motif` apparaît dans `ligne` à au plus un caractère près.
7. Modifier la fonction `numero()` pour que lorsque la recherche est infructueuse, elle affiche les noms, prénoms et numéros de téléphone des personnes dont les noms et prénoms de diffèrent de ceux recherchés qu'à un caractère près.

1. Adapter l'algorithme de recherche d'un mot dans une chaîne.
3. Effectuer un slicing sur la ligne pour en extraire le numéro de téléphone.
7. Utiliser la méthode `seek` des objets-fichiers avec en paramètre `0` pour revenir en début de fichier. Ne pas afficher le numéro de ligne.

1. Code de la fonction `indiceMotif`.

```
def indiceMotif(ligne,motif):
    N = len(ligne)
    n = len(motif)
    for i in range(N-n+1):
```

```
        k = 0
        while k < n and ligne[i+k] == motif[k]:
            k += 1
        if k == n:
            return i
    return -1
```

2. Code de la fonction numero.

```
def numero(nomPrenom):
    fichier = open('annuaire.txt', 'r')
    for ligne in fichier:
        if indiceMotif(ligne,nomPrenom) != -1:
            print(ligne)
    fichier.close()
```

3. Amélioration de la fonction numero.

```
def numero(nomPrenom):
    fichier = open('annuaire.txt', 'r')
    lgr = len(nomPrenom)
    for ligne in fichier:
        ind = indiceMotif(ligne,nomPrenom)
        if ind != -1:
            telephone = ligne[ind+lgr+1:ind+lgr+11]
            print(telephone)
    fichier.close()
```

4. Fonction majuscule.

```
def majuscule(chaine):
    ch = ""
    for x in ch:
        i = minus.index(x)
        ch += majus[i]
    return ch
```

5. Fonction recherche.

```
def recherche():
    nom = input('Saisir le nom : ')
    prenom = input('Saisir le prenom')
    chaine = nom + " " + prenom
    nomPrenom = ""
    for x in chaine:
        if 'a' <= x <= 'z':
            nomPrenom += majuscule(x)
```

```
        else:
            nomPrenom += x
    numero(nomPrenom)
```

6. Modification `indiceMotif_m` de la fonction `indiceMotif`.

```
def indiceMotif_m(ligne,motif):
    N = len(ligne)
    n = len(motif)
    for i in range(N-n+1):
        tolerance = True # Aucun faux caractère rencontré
        k = 0
        while k < n :
            if ligne[i+k] != motif[k]:
                if not(tolerance):      # >1 caractère faux
                    break
                else:                   # 1 caractère faux
                    tolerance = False
            k += 1
        if k == n:
            return i
    return -1
```

7. Modification de la fonction `numero`.

```
def numero(nomPrenom):
    fichier = open('annuaire.txt', 'r')
    lgr = len(nomPrenom)
    Trouve = False
    for ligne in fichier:
        ind = indiceMotif(ligne,nomPrenom)
        if ind != -1:
            telephone = ligne[ind+lgr+1:ind+lgr+11]
            print(telephone)
            Trouve = True
    if not(Trouve):     # Cas d'échec de la recherche
        fichier.seek(0) # Retour en début de fichier
        for ligne in fichier:
            ind = indiceMotif_m(ligne,nomPrenom)
            if ind != -1:
                i = 0   # Avancer jusqu'au nom
                while ligne[i] != " ":
                    i += 1
                print(ligne[i+1:])
    fichier.close()
```

5) ■ Lecture d'informations dans un fichier PDF

Le format de fichier PDF (Portable Document Format) permet de stocker des documents constitués de textes et d'images dans un format compressé, léger et portable (un document PDF peut être ouvert, et avoir une apparence identique sur la plupart des systèmes d'exploitation).
Un fichier `PDF` est un fichier texte commençant par la chaine de caractère `%PDF` et finissant par `%%EOF` (il peut y avoir d'autres caractères avant et après, qui seront ignorés par le lecteur PDF). Il utilise un jeu de caractères plus large que les tables de caractères ASCII et Unicode.
Pour l'ouvrir sous Python, on pourra utiliser l'option `errors` de la fonction **`open`**`()` avec pour valeur `'surogateescape'` ; tous les caractères du fichier ne figurant pas dans la table Unicode seront alors ignorés, mais l'ouverture s'exécutera sans aucun message d'erreur d'encodage.

```
fichier = open('nom_du_fichier', 'r', errors="surrogateescape")
texte = fichier.read()
fichier.close()
```

1. Écrire une fonction `indice(chaine,mot)` prenant en paramètres deux chaînes de caractère `chaine` et `mot` et qui retourne -1 si `chaine` ne contient pas `mot` comme sous-chaîne, et sinon l'indice dans `chaine` où la première occurence de `mot` a été trouvée.
2. Écrire une fonction `estPDF()` qui permet de vérifier qu'un fichier `fichier` placé dans le répertoire de travail soit bien au format PDF. Elle est écrira dans la console le résultat trouvé.

Un fichier PDF contient sous forme de chaînes de caractères, des commandes PDF donnant, lorsqu'elles sont renseignées, des informations sur le fichier.
Le nom du créateur du fichier (ou du logiciel ayant encodé le fichier) apparaît souvent, entre parenthèses, après la chaine de caractère `'/Creator'`, éventuellement séparés d'un ou plusieurs espaces.
Par exemple si le fichier contient la chaine de caractère `'/Creator (Jean Martin)'`, son créateur est Jean Martin.

Les date, heure et fuseau horaire de création du fichier sont souvent renseignés après la chaîne de caractère `/CreationDate` sous la forme :

`/CreationDate(D:aaaammjjhhmmss+XX'00')`

où `aaaa` : année, `mm` : mois, `jj` : jour, `hh` : heure, `mm` : minutes, `ss` : secondes, `+XX` décalage GMT. La commande `/CreationDate` et ses paramètres entres parenthèses `(...)` peuvent être séparés d'aucun, un ou plusieurs espaces.
Par exemple :

`/CreationDate(D:20150611214310+01'00')`

décrit une fichier créé le 11 juin 2015, à 21h 43m 10s, à la longitude du méridien de Paris (GMT+01).

3. Ecrire une fonction `createurPDF(fichier)` où `fichier` est un fichier PDF placé dans le répertoire courant ; la fonction écrira dans la console le nom de son créateur, si il y figure, et la chaine de caractère `'Auteur non trouvé'` sinon.

4. Écrire une fonction `datePDF()` qui prend en paramètre le nom d'un fichier PDF placé dans le répertoire courant qui, si possible, écrit dans la console, dates et heures de création (avec décalage GMT).

1. Adapter l'algorithme de recherche dans une chaîne de caractères d'une sous-chaîne.
2. Comparer les indices où apparaissent les sous-chaînes `'%PDF'` et `'%%EOF'` dans le texte du fichier.
3. Copier la chaîne après `/Creator` et délimitée par deux parenthèses.
4. On supposera que dès que la chaîne `/CreationDate` apparaît et est suivie de `(D:` la date est présente. Extraire alors les champs significatifs par slicing.

1. Fonction `indice`.

```
def indice(chaine,mot):
    N = len(chaine)
    n = len(mot)
    for i in range(N-n+1):
        k = 0
        while k<n and chaine[i+k]==mot[k]:
            k+=1
        if k==n:
            return i
    return -1
```

2. Script qui vérifie si un fichier est bien au format PDF.

```
def estPDF(fichier):
    f = open(fichier,'r', errors="surrogateescape")
    texte = f.read()
    f.close()
    if -1 < indice(texte,'%PDF') < indice(texte,'%%EOF') :
       print("Le fichier est au format PDF")
    else:
       print("Le fichier n'est pas au format PDF")
```

3. Fonction `createurPDF`.

```
def createurPDF(fichier):
    f = open(fichier,'r', errors="surrogateescape")
    texte = f.read()
    f.close()
    ind = indice(texte,'/Creator')
    if ind == -1:
        print('Auteur non trouvé')
    i = ind + len('/Creator')
```

```
    while texte[i] != '(':
        i += 1
    i += 1
    ch = ''
    while texte[i] != ')':
        ch += texte[i]
        i += 1
    print('Auteur :', ch)
```

4. Fonction `datePDF`.

```
def datePDF(fichier):
    f = open(fichier,'r', errors="surrogateescape")
    texte = f.read()
    f.close()
    i = indice(texte,'/CreationDate')
    if i == -1:
        print("Date de création introuvable")
        return
    k = i+len('/CreationDate')
    while texte[k] == ' ':
        k += 1
    date = texte[k+1:k+1+23]
    if date[:2] != 'D:':
        print("Format de date illisible")
        print(date)
        return
    print("Fichier PDF créé le :")
    print(date[8:10],'/',date[6:8],'/',date[2:6])
    print(date[10:12],'h',date[12:14],'m',date[14:16],\
                                        's. GMT',date[16:19])
```

6) ■ Algorithme efficace de recherche d'une sous-chaîne

Rechercher une sous-chaîne `mot` dans une chaîne de caractère `chaine` est un algorithme de grande utilité. L'algorithme vu en cours est naïf, on peut écrire des algorithmes bien plus efficaces.
Une bonne idée consiste à procéder en comparant `chaine` et `mot` comme dans l'algorithme naïf, mais en sens inverse, en commençant par le dernier caractère de `mot` :

– Pour i variant de 0 à `len(chaine)-len(mot)-1` :

– chercher si le motif recherché (de longueur `n`) apparaît à l'indice `i` en comparant à partir de la fin du mot, c'est à dire commencer par tester si le caractère `mot[n-1]` est identique à `chaine[i+n-1]`,

– si c'est le cas on compare l'avant dernière lettre : `mot[n-2]` avec `chaine[i+n-2]`, etc. jusqu'à avoir trouvé le mot.

– Si on ne trouve pas le mot à cette position i, on vérifie si le caractère discordant `chaine[i+k]` ($\neq$ `mot[k]`) est un caractère de `mot`. Si c'est le cas on recommence à la position `i+1`, mais sinon on poursuit la recherche non pas à en `i+1` mais à la position `i+1+k`.

Exemple : avec `chaine = 'ADCBABCD'` et `mot = 'ABC'` :

A	D	C	B	A	B	C	D	
A	B	C						$i=0,\ k=2$
A	D	C	B	A	B	C	D	
A	B	C						$k=1$
A	D	C	B	A	B	C	D	
		A	B	C				$i=2,\ k=2$
A	D	C	B	A	B	C	D	
			A	B	C			$i=3,\ k=2$
A	D	C	B	A	B	C	D	
				A	B	C		$i=4,\ k=2$
A	D	C	B	A	B	C	D	
				A	B	C		$k=1$
A	D	C	B	A	B	C	D	
				A	B	C		$k=0$

Le mot a été trouvé à la position $i=4$ dans la chaîne.

L'avantage de cette approche, est que si un caractère `c` rencontré dans `chaine` n'apparaît pas dans `mot`, on décale `mot` dans `chaine` vers la droite de façon à ce que `mot` débute sur le caractère suivant. On ne teste donc plus toutes les positions dans `chaine`, contrairement à l'algorithme naïf.

1. Écrire une fonction `caracteres` prenant en paramètre une chaine de caractère `mot` et qui renvoie une chaine contenant les mêmes caractères que `mot`, mais en seul exemplaire pour chacun. C'est la fonction de pré-traitement de la chaîne `mot`.
2. Écrire une fonction qui exécute la recherche d'une sous-chaîne par cette méthode.
3. En proposer une amélioration : on constitue les listes des indices dans `mot` pour chacun de ses caractères. À chaque caractère `c` discordant durant la recherche mais présent dans `mot`, on ne décale par `mot` d'une position sur la droite dans `chaine`, mais à la première position à droite où le caractère `c` coïncide dans `mot` et `chaine`. Ainsi on teste encore moins de positions que dans l'approche précédente.

Exemple : avec `chaine = 'ADCBABCD'` et `mot = 'ABC'` :

A	D	C	B	A	B	C	D	
A	B	C						$i=0,\ k=2$
A	D	C	B	A	B	C	D	
A	B	C						$k=1$
A	D	C	B	A	B	C	D	
		A	B	C				$i=2,\ k=2$
A	D	C	B	A	B	C	D	
				A	B	C		$i=4,\ k=2$

A	D	C	B	A	B	C	D	
				A	B	C		$k=1$
A	D	C	B	A	B	C	D	
				A	B	C		$k=0$

1. Parcourir les caractères de `mot`. Pour chaque nouveau caractère le concaténer dans la chaîne `ch` initialement vide et que la fonction renverra. On pourra utiliser l'expression `c not in ch`.
2. Adapter l'algorithme naïf vu en cours pour qu'à chaque position, la comparaison débute en fin de `mot` et s'exécute en sens inverse. Si le caractère discordant apparaît à l'indice `k` dans `mot`, décaler `mot` sur la droite d'une position dans `chaine`, sinon de `k+1`.
3. Question ouverte et difficile. Écrire d'abord une fonction qui effectue le pré-traitement de la chaîne recherchée et renvoie les listes d'occurence de chaque caractère. Modifier ensuite dans la fonction écrite en 2, la partie où le caractère lu dans `chaine` est un caractère de `mot`, pour faire avancer `mot` dans `chaine` à la position adéquate.

1. Fonction `caracteres`.

```
def caracteres(mot):
    ch = ""
    for c in mot:
        if c not in ch:
            ch += c
    return ch
```

2. Fonction de recherche obtenue par cette approche.

```
def recherche(chaine,mot):
    N = len(chaine)
    n = len(mot)
    carac = caracteres(mot)
    i = 0
    while i < N-n+1:
        k = n-1
        while k>=0 and chaine[i+k]==mot[k]:
            k-=1
        if k==-1:
            return True
        elif chaine[i+k] not in carac:
            i += k+1      # décalage au caractère suivant
        else:
            i += 1
    return False
```

3. Pour améliorer l'algorithme :

```
carac = caracteres(mot)
```

permet d'obtenir (prétraitement) la chaine des différents caractères de `mot`. On constitue aussi la liste des occurrences où ces caractères apparaissent dans `mot` :

```
carac, occur = occurence(mot)
```

en écrivant une fonction `occurence` pour qu'elle renvoie la chaine des caractères `carac`, renvoyée par `caracteres`, ainsi que la liste des occurrences des caractères dans `mot`. Par exemple : `carac, occur = occurence('abracadabra')` renverra :

pour `carac` la chaîne `"abrcd"`,

pour `occur` la liste `[[0,3,5,7,10], [1, 8], [2, 9], [4], [6]]`.

La dernière partie la plus délicate, consiste à modifier la fonction `recherche` de la façon suivante : dans la partie **`else:`**, à l'indice `k`, plutôt que d'incrémenter `i` de 1 : on cherche grâce à la liste `occur` le plus grand indice `m<k` où se situe le caractère `chaine[i+k]` dans `mot`. Si on ne le trouve pas, on incrémente `i` de `k+1`. Sinon on incrémente `i` de `k−m` (et non plus de 1).

On obtient un algorithme efficace. Le meilleur algorithme connu (Algorithme de Boyer-Moore), est basé sur ce principe.

Voici le code de cet algorithme de recherche amélioré que nous venons de décrire.

```
# Algorithme de recherche d'une sous-chaine

def caracteres(mot):
    ch = ""
    for c in mot:
        if c not in ch:
            ch += c
    return ch

def occurences(mot):
    """Renvoie la liste des occurences de chaque
    caractère dans mot (int list list)"""
    carac = caracteres(mot)
    occur = []
    for c in carac:
        L = []
        for i, d in enumerate(mot):
            if c == d:
                L.append(i)
        occur.append(L)
    return carac, occur

def recherche(chaine,mot):
    N = len(chaine)
    n = len(mot)
    carac, occur = occurences(mot)
```

```
    i = 0
    while i < N-n+1:
        k = n-1
        while k>=0 and chaine[i+k]==mot[k]:
            k-=1
        if k==-1:
            return True
        elif chaine[i+k] not in carac:
            i += k+1
        else:   # translation de mot en avant dans chaine
            j = carac.index(chaine[i+k]) # Indice dans occur
            m = len(occur[j])-1          # Initialisation
            while m >-1 and occur[j][m] >= k: # Recherche
                m -= 1
            if m == -1:    # si non trouvé translation +k+1
                i += k+1
            else:          # si trouvé à l'indice m
                i += k-m  # translation +k-m
    return False
```

Exemple.

```
In [2]: recherche('abracadabra','da')
Out[2]: True

In [3]: recherche('abracadabra','abra')
Out[3]: True

In [4]: recherche('abracadabra','obra')
Out[4]: False

In [5]: recherche('abracadabra','bara')
Out[5]: False

In [6]: recherche('ADCBABCD','ABC')
Out[6]: True

In [7]: recherche('ADCBABCD','ACB')
Out[7]: False
```

1. Intersection d'ensemble de points. . . . . D'après X-ENS 2017, MP-PC-PSI.

Soit n un entier naturel. on note D_n l'ensemble des entiers naturels compris entre 0 et $2^n - 1$. On appelle "point de $D_n \times D_n$" tout couple d'entiers $(x, y) \in D_n \times D_n$. Soient P et Q deux parties de $D_n \times D_n$. On cherche à calculer l'intersection des ensembles de points P et Q.

Un point de coordonnées $(x, y) \in D_n \times D_n$ est représenté en Python par une liste de deux entiers naturels `[x,y]`. Un ensemble de points est représenté par une liste de points sans répétition, donc comme une liste de listes d'entiers naturels de longueur 2.

1. Écrire une fonction `membre(p,q)` qui renvoie `True` si le point `p` est dans l'ensemble représenté par la liste `q` et qui renvoie `False` dans le cas contraire.
2. Écrire une fonction `intersection(p,q)` qui renvoie une liste repéresentant l'intersection des ensembles représentés par `p` et `q`. On implémentera l'algorithme qui consiste à itérer sur tous les points de `p` et à insérer dans le résultat ceux qui sont aussi dans `q`.
3. Si la comparaison entre deux entiers naturels est prise comme opération élémentaire, quelle est la complexité de l'algorithme de la question précédente exprimée en fonction de la longueur de `p` et `q` ?

1, 2. Pour faciliter le calcul de complexité en 3, utiliser des boucles et tests, en évitant les fonctionnalités de plus haut niveau.

1. Fonction `membre(p,q)` qui teste si `p` est membre de `q`.

```
def membre(p,q):
    for x in q:
        if x == p:
            return True
    return False
```

2. Fonction `intersection(p,q)` qui renvoie l'intersection de `p` et `q`.

```
def intersection(p,q):
    L = []
    for x in p:
        if membre(x,q):
            L.append(x)
    return L
```

3. Si la comparaison de deux entiers est une opération élémentaire, alors la comparaison de deux listes de deux entiers nécessite 2 opérations élémentaires. Ainsi la fonction `membre(p,q)` a pour complexité $O(\texttt{len(q)})$ dans le pire des cas.

L'ajout d'un élément en fin de liste est en $O(1)$ en temps amorti. Aussi, dans le pire des cas, celui où l'intersection est vide, la complexité est en $O(\texttt{len(p)} \times \texttt{len(q)})$.

2. Tracé des données d'un fichier. . . . . D'après CCP 2016, Modélisation, PC.

Des données expérimentales sont disponibles sous forme d'un fichier texte nommé « `data.txt` » dans lequel les informations sont présentées sous forme de colonnes. La première colonne du fichier correspond au temps (unité : s), la deuxième au débit molaire d'un gaz A (unité : mol·s – 1) enregistré au cours d'une première expérience, et la troisième colonne au débit molaire de A (unité : mol·s – 1) enregistré au cours d'une seconde expérience.

```
00.0 1.9231E-04 1.9230E-04
10.0 1.8293E-04 1.9108E-04
20.0 1.7401E-04 1.8991E-04
30.0 1.6552E-04 1.8878E-04
40.0 1.5745E-04 1.8771E-04
50.0 1.4977E-04 1.8667E-04
```

1. Indiquer la syntaxe à utiliser pour charger le fichier « `data.txt` » qui contient les données expérimentales. Donner le code permettant de créer trois vecteurs `T`, `A1` et `A2` correspondant respectivement aux données des première, deuxième et troisième colonnes. Donner également le code qui permet de déterminer le nombre de points expérimentaux n.
2. Donner la syntaxe permettant de tracer sur un graphe les évolutions du débit molaire de A en fonction du temps au cours des deux expériences.

1. Il est très utile d'utiliser la méthode `split` des chaînes de caractères. Autrement on pourra utiliser la méthode `index` des séquences et du slicing.
2. Utiliser la fonction `plot` de `pyplot`.

1. Le mieux est d'utiliser la méthode `split` qui sans argument renvoie la liste des sous-chaînes délimitées par le caractère espace.

```
F = open('data.txt', 'r')
T, A1, A2 = [], [], []
for ligne in F:
    t, a1, a2 = ligne.split()
    T.append(float(t))
    A1.append(float(a1))
    A2.append(float(a2))
F.close()
n = len(T)
```

Ou encore, sans utiliser la méthode `split` :

```
F = open('data.txt', 'r')
T, A1, A2 = [], [], []
for ligne in F:
    i = ligne.index(' ')
```

```
    j = ligne[i+1:].index(' ')
    t = ligne[:i]
    a1 = ligne[i+1:j]
    a2 = ligne[j+1:]
    T.append(float(t))
    A1.append(float(a1))
    A2.append(float(a2))
F.close()
n = len(T)
```

2. Tracé des deux courbes de l'évolution des concentrations.

```
import matplotlib.pyplot as plt
plt.figure()
plt.plot(T,A1, label = '1ere expérience')
plt.plot(T,A2, label = '2eme expérience')
plt.legend()
plt.show()
```

3. Nombres premiers jumeaux. . . . . D'après e3a 2016, Mathématiques 1, PSI.

1. On donne les programmes python P0 et P1 suivants. Que renvoient les appels `P0(5)`, `P1(5)` et `P0(9)`, `P1(9)` ?

Dire en une phrase ce que fait chacun des programmes P0 et P1 ?

```
def P0(N): # N entier naturel
    if N == 1:
        return False
    if N == 2:
        return True
    for d in range(2,N):
        if N % d == 0:
            return False
        return True
```

```
def P1(N): # N entier naturel
    if N == 1:
        return False
    if N == 2:
        return True
    for d in range(2,N):
        if N % d == 0:
            return False
    return True
```

2. En une phrase dire ce que fait le programme python, P2, qui utilise le programme P1 précédent :

```
def P2(N): # N entier naturel
    L = []
    k = 0
    n = k*k+1
    while n <= N:
        if P1(n):
            L.append(n)
        k = k+1
        n = k*k+1
    return L
```

Que renvoie l'appel `P2(127)` ?

3. Écrire une fonction `nextPrime` en langage python qui prend un argument entier `N` et qui retourne comme valeur le premier nombre premier qui est strictement supérieur à `N`.

4. **Nombres jumeaux**
   On appelle couple de nombres premiers jumeaux toute liste `[p,q]` telle que `p`, `q` sont deux nombres premiers vérifiant `p < q` et `q = p + 2`. Par exemple `[3,5]`, ou `[11,13]` sont des couples de nombres premiers jumeaux alors que `[2,3]` ne l'est pas.

   a) Écrire à l'aide de la fonction `nextPrime` précédente, une fonction python nommée `jumeau`, prenant comme argument un entier `N` et renvoyant le couple `[p,q]` de nombres premiers jumeaux tel que `p` strictement supérieur à N est le plus petit possible.
   Par exemple, `jumeau(5)`, renvoie comme valeur : `[11, 13]`.

   b) Écrire avec les mêmes consignes une fonction, `lesJumeaux`, prenant en argument un entier `N` et renvoyant la liste de tous les couples de nombres premiers jumeaux `[p,q]` tels que `q` soit inférieur ou égal à `N`.
   Par exemple, `lesJumeaux(18)` retourne `[[3, 5], [5, 7], [11, 13]]` (le couple `[17, 19]` n'en fait donc pas pas partie.)

2. Pour déterminer le retour de `P2(127)` calculer tous les nombres de la forme k^2+1 pour k variant de 0 à 10. Garder à l'esprit qu'un nombre n est premier si et seulement si il n'est pas divisible par un nombre premier $\leqslant \sqrt{n}$.
3. Appeler la fonction `P1` au sein d'une boucle perpétuelle **`while True:`**.
4.a. Utiliser encore une boucle perpétuelle.
4.b. Prendre garde à `p < q < N`.

1. Les appels `P0(5)`, `P1(5)`, `P0(9)` et `P1(9)` renvoient respectivement `True`, `True`, `True`, `False`.

La fonction `P1` renvoie un booléen, `True` si et seulement si l'argument est un nombre premier.

La fonction `P0` renvoie un booléen, `True` si et seulement si l'argument est `2` ou impair. (C'est une version érronée de la fonction `P1`, la dernière instruction étant incorrectement indentée.)

2. La fonction `P2` renvoie la liste de tous les nombres premiers inférieurs ou égaux à l'argument `N` et qui sont de la forme k^2+1.

L'appel de `P2(127)` renvoie `[1, 2, 5, 17, 37, 101]`.

3. Fonction `nextPrime`.

```
def nextPrime(N):
    N += 1
    while True:
        if P1(N):
```

```
            return N
        N += 1
```

4.a. Fonction `jumeau` qui renvoie le plus petit couple de nombres premiers jumeaux strictement supérieurs à `N`.

```
def jumeau(N):
    p = N
    while True:
        p = nextPrime(p)
        if P1(p+2):
            return [p, p+2]
```

4.b. Fonction `lesJumeaux`.

```
def lesJumeaux(N):
    L = []
    p = 2
    while p < N-2:
        q = nextPrime(p)
        if q == p + 2:
            L.append([p, q])
        p = q
    return L
```

4. Matrices magiques. . . . . Épreuve type. Oral Banque PT.

Une matrice carrée d'ordre n est dite magique si elle contient tous les nombres de 1 à n^2 et si les sommes des nombres de chaque ligne, de chaque colonne et de chaque diagonale sont toutes égales à une même constante s.

1. Au brouillon, exprimer la constante s en fonction de n.
2. Créer une fonction `EstMagique`, d'argument une matrice `T` (carrée de taille n) et qui renvoie un booléen indiquant si `T` est magique ou pas.

   Tester cette fonction sur les matrices :

$$A = \begin{pmatrix} 4 & 9 & 2 \\ 3 & 5 & 7 \\ 8 & 1 & 6 \end{pmatrix} \quad \text{et} \quad B = \begin{pmatrix} 1 & 8 & 2 \\ 4 & 5 & 7 \\ 6 & 9 & 3 \end{pmatrix}$$

Une méthode permettant de construire une matrice magique de taille impaire $n = 2p+1$ est la suivante :

– On construit une matrice de taille n remplie de zéros. On considère cette matrice comme la représentation sur une période d'une matrice infinie n-périodique en lignes et en colonnes.

– On remplace ensuite les zéros de la matrice avec les nombres de 1 à n^2 comme suit :

– on met le 1 dans la case située sous la case centrale de la matrice ;
– on place ensuite chaque nombre de 2 à n^2 dans la case située ligne suivante et colonne suivante de celles où on a mis le nombre précédent. Si cette case est déjà remplie, on avance encore d'une ligne et on recule d'une colonne. On admet que la matrice ainsi construite est magique.

On admet que la matrice ainsi construite est magique.

3. Construire *à la main*, selon cette méthode, une matrice magique d'ordre 3.

4. Écrire une fonction `Magique` d'argument un entier `p` qui renvoie la matrice magique de taille $2p+1$ créée à l'aide de la méthode précédente.

1. « Au brouillon » doit alerter qu'exprimer s en fonction de n sera utilisé par la suite.

2. Utiliser l'expression de `s` obtenue en 1) pour la comparer à la somme sur chaque ligne et chaque colonne. Vérifier aussi que tous les entiers entre 1 et n^2 figurent dans la matrice ; pour cela, une bonne idée consiste à déclarer une liste `L` de `n^2 False`, que l'on change par `True` à l'indice `k` dès que `k` apparaît dans la matrice. Renvoyer `False` dès qu'on trouve dans `M` un nombre hors de $[\![1, n^2]\!]$ ou déjà rencontré.
Attention, pour pouvoir l'appliquer à des tableaux `ndarray` il faudra le caractère entier d'un élément `x` ne pourra être testé par **`type(x) == int`** ; utiliser plutôt l'expression **`x == int(x)`**.

4. Il est pratique d'utiliser la fonction `zeros` de `numpy` (avec l'option `dtype=int`) ; sinon initialiser la matrice par l'instruction : `M = [[0]*n for k in range(n)]`. Tous les indices doivent être pris modulo n.

1. En sommant tous les éléments d'une matrice magique de deux façons différentes, on obtient :

$$n \times s = \sum_{k=1}^{n^2} k = \frac{n^2(n^2+1)}{2} \implies s = \frac{n(n^2+1)}{2}$$

2. Fonction `EstMagique`.

```
def EstMagique(T):
    n = len(T)
    # 1 : tester que tous les entiers de 1 à n^2 apparaissent
    L = [False] * n**2
    for i in range(n):
        for j in range(n):
            x = T[i][j]
            if not(x == int(x) and 0 <x<= n**2) or L[x-1]:
                return False
            L[x-1] = True
    # 2 : tester que la somme de chaque ligne/colonne est s
    s = n*(n**2+1)/2
    for i in range(n):
        S1 = S2 = 0
```

```
        for j in range(n):
            S1 += T[i][j]
            S2 += T[j][i]
        if S1 != s or S2 != s:
            return False
    return True
```

Test : on utilise la fonction `array` de `numpy`.

```
In [2]: from numpy import array

In [2]: A = array([[4,9,2],[3,5,7],[8,1,6]])

In [3]: B = array([[1,8,2],[4,5,7],[6,9,3]])

In [4]: EstMagique(A)
Out[4]: True

In [5]: EstMagique(B)
Out[5]: False
```

3. La matrice d'ordre 3 obtenue par cette méthode est :

$$\begin{pmatrix} 3 & 5 & 7 \\ 8 & 1 & 6 \\ 4 & 9 & 2 \end{pmatrix}$$

4. Fonction `Magique`.

```
from numpy import zeros

def Magique(n):
    if n%2==0:
        return
    M = zeros((n,n),dtype=int)
    # ou bien : M = [[0]*n for k in range(n)]
    i = j = n//2
    M[i][j] = 1
    for k in range(2,n**2+1):
        i = (i+1)%n
        j = (j+1)%n
        if M[i][j] == 0:
            M[i][j] = k
        else:
            i = (i+1)%n
            j = (j-1)%n
            M[i][j] = k
    return M
```

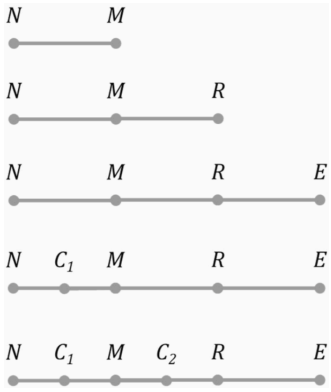
5. Algorithme de Nelder-Mead. . . . . D'après CCP 2016, Modélisation, PC-TPC.

L'algorithme de Nelder-Mead est une méthode utilisée pour trouver le minimum d'une fonction. Il s'agit d'un algorithme d'optimisation non linéaire basé sur le concept de simplexe à $n+1$ sommets dans un espace à n dimensions. Dans le cas d'une fonction f d'une variable ($n = 1$), un simplexe est un segment.

L'algorithme de Nelder-Mead est une procédure itérative. Partant d'un segment initial $[M, N]$, l'algorithme va générer une succession de segments par des transformations simples au cours des itérations : le segment se déplace et se réduit jusqu'à ce que ses extrémités se rapprochent d'un point où la fonction présente un minimum (ce minimum peut être un minimum local).

Soient x_M et x_N les abscisses des points M et N. Les transformations subies par le segment (illustrées à la figure suivante) sont basées sur la comparaison des valeurs de la fonction f aux extrémités du segment.

- La première étape consiste à réindexer si nécessaire les deux extrémités du segment de manière à ce que $f(x_M) \leqslant f(x_N)$.
- L'extrémité N pour laquelle la fonction f est maximale est remplacée par une nouvelle extrémité. On introduit le point R, réflexion de N par rapport à M, tel que $x_R = x_M + (x_N - x_M)$.
- Si $f(x_R) < f(x_M)$, le segment est étiré dans cette direction (car la réflexion se rapproche du minimum). On introduit un point E (étirement du segment) tel que $x_E = x_M + 2(x_M - x_N)$ qui permet éventuellement de se rapprocher encore un peu plus du minimum. Le point N est substitué par le point R si $f(x_R) < f(x_E)$, sinon par E.
- Si $f(x_R) > f(x_M)$, le segment est réduit dans la direction opposée de la réflexion (car la réflexion s'éloigne du minimum). On introduit le point C_1 (contraction du segment) qui est défini par $x_{C_1} = x_N + 1/2(x_M - x_N)$. Si $f(x_{C_1}) < f(x_M)$, N est remplacé par C_1. Sinon N est remplacé par C_2, homothétie de rapport -1 et de centre M du point C_1 ($x_{C_2} = x_M + 1/2(x_M - x_N)$).



Pour l'implantation de l'algorithme demandé, on supposera que les abscisses x_M, x_N des deux extrémités du segment initial $[M, N]$, ainsi que la fonction f, sont donnés par les 3 variables globales `xM`, `xN` et `f`.

1. Soit $f(x)$ une fonction dont on cherche le minimum. Soient x_M et x_N les abscisses des extrémités d'un segment $[M, N]$. Écrire le code permettant de réindexer les deux extrémités du segment de manière à ce que $f(x_M)$ soit inférieure ou égale à $f(x_N)$.
2. Partant du segment $[M, N]$ réindexé, écrire le code permettant de transformer le segment $[M, N]$ au cours d'une itération de la méthode de Nelder-Mead. Chaque étape du code devra être commentée.
3. Construire une boucle permettant de réaliser des itérations successives de la méthode

de Nelder-Mead. Cette boucle sera interrompue lorsque l'écart relatif $\left|\frac{x_M - x_n}{x_M}\right|$ sera inférieur à 10^{-4} ou lorsque le nombre maximum d'itérations, fixé par l'utilisateur, sera atteint soit $Itmax = 1000$.

3. Indiquer simplement à l'aide de commentaires où placer dans la boucle le code écrit aux questions 1 et 2.

1. Première étape d'une itération de l'algorithme de Nelder-Mead.

```
if f(xm) > f(xn):
   xm, xn = xn, xm
```

2. Deuxième étape d'une itération de l'algorithme de Nelder-Mead.

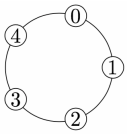
```
xr = xm + (xm-xn)          # point R
if f(xr) < f(xm):          # étirement du segment
   xe = xm + 2*(xm-xn)
   # modification de N
   if f(xr) <= f(xe):
      xn = xr
   else:
      xn = xe
else:                      # contraction du segment
   xc1 = xm + 0.5*(xm-xn)
   if f(xc1) < f(xm):      # N devient C1
      xn = xc1
   else:                   # N devient C2
      xn = xm + 0.5*(xm-xn)
```

3. Boucle principale.

```
compteur = 0
itmax = 1000
while abs((xm-xn)/xm) > 1e-4 and compteur < itmax:
   # 1ère étape :
   # code de la question 1 à insérer ici
   # ...
   # 2ème étape :
   # code de la question 2 à insérer ici
   # ...
   # ...
   # ...
   # ...
   compteur += 1
```

6. Élimination circulaire. . . . . Épreuve type. Oral Banque PT.

On considère n personnes numérotées de 0 à $(n-1)$ disposées en cercle, comme le montre la figure suivante pour $n = 5$:



En commençant par la personne numéro 1 et en tournant dans le sens des numéros croissants (sens horaire sur la figure), on retire une personne sur deux, en ne prenant en compte que les personnes restant dans le cercle. Par exemple, pour $n = 5$, on retirera successivement les personnes 1, 3, 0 puis 4 ; il restera la personne 2.

Pour simuler cette procédure d'élimination progressive, on décrit le cercle par une liste de n booléens : la valeur numéro i est `True` si la personne i est éliminée, et `False` si elle est encore dans le cercle. Au départ, la liste ne contient que des `False` puis ses valeurs passent progressivement à `True`, jusqu'à ce qu'il ne reste plus qu'un seul `False`. Ainsi, pour $n = 5$, la liste passe par les états successifs :

Liste de booléens `E`	rang `p` du dernier élément éliminé
`[False, True, False, False, False]`	1
`[False, True, False, True, False]`	3
`[True, True, False, True, False]`	0
`[True, True, False, True, True]`	4

Il reste 2

1. Écrire une fonction `suivant` de deux arguments, une liste `E` de n booléens et un entier `p` entre 0 et $(n-1)$ qui renvoie la position `q` du premier `False` rencontré en partant de la position juste après la position `p`, en parcourant la liste de façon circulaire. Par exemple :

   `suivant([True,True,False,True,False],0)` donne 2 ;
   `suivant([True,True,False,True,False],2)` donne 4 ;
   `suivant([True,True,False,True,False],4)` donne 2.

2. Se servir de la fonction suivant pour simuler le cas $n = 5$ décrit dans le tableau ci-dessus.

3. Écrire une fonction `reste` d'argument un entier naturel non nul n qui renvoie le numéro de la dernière personne restante parmi n personnes.

4. Pour $2 \leqslant n \leqslant 140$, afficher n suivi du numéro du dernier restant. En observant le résultat, que peut-on conjecturer sur le calcul du dernier restant ?

5. En supposant que la conjecture est exacte, deviner sans utiliser le programme Python quel sera le numéro du restant pour les valeurs de n : 256, 513, 1023, 1041.

6. Écrire une fonction `reste2` qui renvoie pour n le résultat conjecturé, et comparer les résultats obtenus pour $n = 5104$.

1. Pour parcourir cycliquement la liste, tous les indices calculés doivent l'être module `n` ; utiliser l'opérateur `%`.
3. Généraliser le code écrit en 2) à n personnes.

4. Remarquer que le reste vaut 0 pour toutes les puissances de 2, puis augmente par pas de 2.
5. Importer la fonction `log_2` du module `math`. Pour la partie entière on peut utiliser la fonction de conversion **`int`**.

1. Fonction `suivant`

```
def suivant(E,p):
    n = len(E)
    personne = True
    while personne:
        p = (p+1) % n
        personne = E[p]
    return p
```

2. Simulation du cas $n = 5$.

```
E = [False] * 5
p = 4
for boucle in range(4):
    p = suivant(E,p)
    p = suivant(E,p)
    E[p] = True
    print(E)
```

L'exécution affiche, comme attendu :

```
[False, True, False, False, False]
[False, True, False, True, False]
[True, True, False, True, False]
[True, True, False, True, True]
```

3. Fonction principale, `reste`. Au sein d'une boucle, on appelle deux fois la fonction `suivant` avant de changer la valeur de l'élément trouvé dans la liste.

```
def reste(n):
    E = [False] * n
    p = n-1
    for boucle in range(n-1):
        p = suivant(E,p)
        p = suivant(E,p)
        E[p] = True
    return suivant(E,p)
```

4. Code pour l'affichage.

```
for n in range(2,141):
    print("n =",n,"Dernier restant :",reste(n))
```

On remarque que le reste vaut 0 pour toutes les puissance de 2 : 2, 4, 16, 32, 64, 128, puis s'incrémente de 2 à chaque nouvelle itération.

En notant N la plus grande puissance de 2 inférieure ou égale à n, c'est-à-dire : $N = 2^{\lfloor \log_2(n) \rfloor}$, on conjecture que le dernier restant parmi n personnes vaut $2 \times (n - N)$, soit :

$$2 \times \left(n - 2^{\lfloor \log_2(n) \rfloor} \right)$$

5. En admettant la conjecture faite :

n	$N = 2^{\lfloor \log_2(n) \rfloor}$	$(n - N)$	Dernier restant
256	256	0	0
513	512	1	2
1023	512	511	1022
1041	1024	17	34

6. Fonction `reste2`.

```
from math import log2

def reste2(n):
    N = 2**int(log2(n))
    return (n-N)*2
```

Vérification :

```
In [2]: reste(5104), reste2(5104))
(2016, 2016)
```

La conjecture est vérifiée pour $n = 5104$.

7. Calendrier grégorien. ..... Épreuve type. Oral Banque PT.

On travaille avec des triplets (jour, mois, année) où jour et année sont des nombres entiers (avec la condition année > 1582 : date de mise en place du calendrier grégorien) et mois est une chaîne de caractères.

1. Créer une liste notée `MC` contenant les mois de l'année qui ont 30 jours et une liste notée `ML` contenant les mois de l'année qui ont 31 jours.
2. On rappelle qu'une année est bissextile lorsqu'elle est divisible par 4 mais pas par 100, ou bien lorsqu'elle est divisible par 400. Créer une fonction `estBissextile` d'un argument `an` qui renvoie `True` si l'année `an` est bissextile et `False` sinon. Créer une fonction `longueurmois` de deux arguments `ms` et `an` qui renvoie le nombre de jours du mois `ms` de l'année `an`.
3. Créer une fonction `valide` de trois arguments `jr`, `ms`, et `an` qui renvoie `True` si le triplet `(jr,ms,an)` est valide et `False` sinon.
   Par exemple `valide(25,'janvier',1896)` devra renvoyer `True` alors que `valide(30,'février',1972)` devra renvoyer `False`.

4. Écrire une fonction `nab` de deux arguments `date1` et `date2` qui renvoie le nombre de « 29 février » entre ces deux dates. `date1` et `date2` sont deux triplets sous la forme `(jr,ms,an)`.
5. écrire une fonction `jours` de deux arguments `date1` et `date2` qui renvoie le nombre de jours séparant les deux dates.

2. Utiliser les listes `MC` et `ML`.
3. Tester d'abord le type des 3 arguments (par exemple **`type`**`(a) == int` permet de teste si `a` est de type entier. Puis tester que `an > 1582`, que `ms` est bien une chaîne valide, et finalement que `jr` est valide.
4. et **5.** Il faudra soit supposer que `date1` est une date antérieure à `date2`, soit écrire une fonction qui les échange pour les mettre dans l'ordre chronologique. Traiter séparément les années pleines (du 1er janvier au 31 décembre), puis les jours restants la première et dernière année.

1. Déclaration des variables globales de type liste, `MC` et `ML`.

```
MC = ['avril','juin','septembre','novembre']
ML = ['janvier','mars','mai','juillet','aout','octobre',\
                                                'decembre']
```

2. Fonction `estBissextile`.

```
def estBissextile(an):
    if an%400 == 0 or (an%4 == 0 and an%100 != 0):
        return True
    else:
        return False
```

Fonction `longueurmois`.

```
def longueurmois(ms,an):
    if ms == 'fevrier':
        if estBissextile(an):
            return 29
        else:
            return 28
    else:
        if ms in MC:
            return 30
        elif ms in ML:
            return 31
```

3. Fonction `valide`.

```
def valide(jr,ms,an):
```

```
    if type(jr) != int or type(ms) != str or type(an) != int:
        return False
    if an <= 1582:
        return False
    if ms == 'fevrier' or ms in MC or ms in ML:
        if 0 < jr <= longueurmois(ms,an):
            return True
    return False
```

4. On commence par écrire une fonction qui prend en arguments deux dates, et les renvoie ordonnés par ordre chronologique.

```
def ordre(date1,date2):
    mois = ['janvier', 'fevrier', 'mars', 'avril', 'mai',\
     'juin', 'juillet', 'aout', 'septembre', 'octobre', \
     'novembre', 'decembre']
    mois1, mois2 = mois.index(date1[1]), mois.index(date2[1])
    if date2[2] < date1[2]:
        date1, date2 = date2, date1
    elif date2[2] == date1[2]:
        if mois2 < mois1:
            date1, date2 = date2, date1
        elif mois2 == mois1:
            if date2[0] < date1[0]:
                date1, date2 = date2, date1
    return date1, date2
```

Fonction `nab` qui renvoie le nombre « 29 février » entre deux dates. Elle utilise la fonction `ordre` pour que `date1` soit antérieure à `date2`.

```
def nab(date1,date2):
    date1, date2 = ordre(date1,date2)
    an1, an2 = date1[2], date2[2]
    B = 0
    for an in range(an1+1,an2):
        if estBissextile(an):
            B += 1
    if estBissextile(an1):
        if date1[1] in ('janvier','fevrier'):
            if an2 > an1\
                 or (date2[1] not in ('janvier','fevrier')):
                B += 1
    if estBissextile(an2):
        if date2[1] not in ('janvier','fevrier'):
            if an1 < an2\
                 or (date1[1] in ('janvier','fevrier')):
```

```
                B += 1
    return B
```

4. Fonction `jours`.
Comme c'est la fonction principale (par exemple pour réaliser un 'calendrier perpétuel') elle commence par vérifier que les deux dates sont valides. Elle utilise ensuite la fonction `ordre` pour que `date1` soit antérieure à `date2`.
On compte d'abord le nombre de jours des années pleines (du 1er janvier au 31 décembre), puis on lui ajoute le nombre de jour résiduel les première et dernière années.

```
def jours(date1,date2):
    # Vérification de la validité des dates
    if not(valide(date1[0], date1[1], date1[2])\
         and valide(date2[0], date2[1], date2[2])):
        print('Dates invalides')
        return
    # Ordre chronologique
    date1, date2 = ordre(date1,date2)
    # Nombre de jours les années pleines
    J = (date2[2]-date1[2]-1)*365 + nab(date1,date2)
    # Liste des mois
    mois = ['janvier', 'fevrier', 'mars', 'avril', 'mai',\
        'juin', 'juillet', 'aout', 'septembre', 'octobre',\
                'novembre', 'decembre']
    # Calcul des jours résiduels
    mois1 = mois.index(date1[1])
    mois2 = mois.index(date2[1])
    for i in range(mois2):
        nomMois = mois[i]
        if nomMois == 'fevrier':
            J += 28
        elif nomMois in MC:
            J += 30
        else:
            J += 31
    J += date2[0]
    for i in range(mois1, 12):
        nomMois = mois[i]
        if nomMois == 'fevrier':
            J += 28
        elif nomMois in MC:
            J += 30
        else:
            J += 31
    J -= date1[0]
    return J
```

8. Jeu de Nim. . . . . Épreuve type d'Oral. Polytechnique - PT.

Le jeu de Nim se joue à deux joueurs. On considère m vases, le i-ème vase contenant x_i fruits. Les joueurs, Alice et Bob, jouent alternativement et Alice commence. Un tour de jeu consiste à retirer autant de fruits que l'on désire (mais au moins un) dans un même vase. Le joueur qui gagne est celui qui enlève le dernier fruit.

1. On appelle configuration du jeu un m-uplet $(y_1, \dots, y_m)$ décrivant le nombre de fruits dans chaque vase à un instant de la partie. Exprimer en fonction des nombre de fruits initialement dans chaque vase le nombre de configurations possibles du jeu. Commentez cet ordre de grandeur.

On considère la fonction suivante G appelée *fonction de Grundy* qui prend en argument les nombres de fruits dans chaque vase et vaut un entier $G(x_1, x_2, \dots, x_m) = a$ défini de la façon suivante : si l'on note $a[i]$ le i-ème bit de la décom !position en base 2 de a, c'est-à-dire que $a = \sum_{i=0}^{n} 2^i a[i]$, alors $a[i]$ est défini par $a[i] = \left(\sum_{l=1}^{m} x_l[i] \right) \mod 2$, où $x_l[i]$ désigne le i-ème bit de la décomposition en base 2 de x_l.
Ainsi, pour calculer $G(6, 9, 1, 10)$, on écrit tout d'abord 6,9,1,10 en base 2 et on fait la somme par colonne modulo 2, ce qui donne la représentation en base 2 de a. On n'a alors plus qu'a convertir a en base 10 :

6	=	0	1	1	0
9	=	1	0	0	1
1	=	0	0	0	1
10	=	1	0	1	0
a	=	0	1	0	0

Ainsi on a $G(6, 9, 1, 10) = a = 4$.

2. Calculer $G(10, 5, 2, 4)$.
3. Écrire une fonction `taille` qui prend en argument un entier x et qui renvoie le plus petit n tel que x puisse s'écrire en base 2 sur n bits. Votre programme ne devra pas utiliser la fonction `log` de Python. Vous préciserez la complexité.
4. Écrire une fonction `binaire` qui prend en argument un entier x et un entier n (tel que x puisse s'écrire en base 2 sur n bits) et qui renvoie une liste a de n bits tels que est $a[i]$ le i-ème bit de la décomposition en base 2 de x sur n bits. Vous préciserez la complexité.
5. Écrire une fonction `decimal` qui prend en argument un suite a de bits et renvoie l'entier x dont a est la décomposition binaire. Vous préciserez la complexité.
6. Écrire une fonction `Grundy` qui prend en argument une liste x de m entiers $x_1, \dots, x_m$ et renvoie la valeur de $G(x_1, \dots, x_m)$. Vous préciserez la complexité.

On définit une stratégie pour Alice comme une fonction qui étant donnée une configuration $(x_1, \dots, x_m)$ du jeu donne un indice $1 \leqslant i \leqslant m$ et un entier $1 \leqslant z \leqslant x_i$ de fruits à retirer d'un vase i. On définit les stratégies de Bob de la même façon. Une stratégie est respectée par un joueur au cours d'une partie si à chaque fois que ce joueur doit jouer il joue le coup indiqué par sa stratégie. On dit qu'un stratégie est gagnante pour Alice si elle assure la victoire à cette dernière dans toute partie où elle la respecte. Enfin, une configuration est

gagnante pour Alice si cette dernière possède une stratégie gagnante pour toute partie débutant dans cette configuration.

7. Écrire une fonction `jouer_coup` qui prend en argument une liste x de m entiers $x_1, \dots, x_m$, un indice $1 \leqslant i \leqslant m$ et un entier $1 \leqslant z \leqslant x_i$ et renvoie une liste y décrivant la configuration obtenue en retirant z éléments au vase d'indice i.

On va maintenant prouver le résultat suivant :

> *Soit la configuration de jeu où les vases contiennent respectivement $x_1, x_2, \dots, x_m$ fruits. Alors le joueur dont c'est le tour a une stratégie gagnante si et seulement si $G(x_1, \dots, x_m) \neq 0$.*

Comme ce résultat est une équivalence, on aura aussi que lorsque $G(x_1, \dots, x_m) = 0$, le joueur qui doit jouer va perdre si son adversaire joue correctement. On partitionne en Y_0 et Y_1 l'ensemble des configurations de jeu, celles de Y_0 correspondant aux cas où la fonction de Grundy est nulle.

8. Prouver que, pour toute configuration dans Y_0, et quel que soit le coup joué, la configuration suivante est dans Y_1.

9. Réciproquement, prouver que, pour toute configuration dans Y_1, il existe un coup tel que la configuration suivante soit dans Y_0.

10. Conclure et déduire de la preuve une stratégie gagnante associée.

11. Écrire une fonction `strategie` qui prend en argument une liste x de m entiers $x_1, \dots, x_m$ et renvoie `None` si la configuration n'est pas gagnante pour le joueur dont c'est le tour, et sinon renvoie (i, z) où (i, z) est un coup qui permet d'arriver dans une configuration de Y_0. Vous préciserez la complexité.

4. Exprimer la complexité en fonction de x et n.
6. Commencer par calculer le nombre n de bits nécessaires à l'écriture binaire des x_i, avant d'initialiser une liste de n zéros.
8. Remarquer qu'un coup inverse nécessairement au moins un bit dans l'écriture binaire de $G(x_1, x_2, \dots, x_m)$.
9. Considérer tous les poids $i_1, \dots, i_p$ où l'écriture binaire de $G(x_1, x_2, \dots, x_m)$ a pour bit 1, et prouver l'existence de x_i et de $0 \leqslant x'_i < x_i$, tel que retirer $x_i - x'_i$ fruits dans le vase i a pour effet d'inverser dans l'écriture binaire de $G(x_1, x_2, \dots, x_m)$ tous les bits de poids $i_1, \dots, i_p$.
10. Remarquer qu'un joueur perdant finira sur une configuration dans Y_0. Remarquer aussi qu'il y aura forcément un gagnant et un perdant.
11. Appliquer l'argument utilisé à la question 9 pour trouver la stratégie permettant de passer de Y_1 à Y_0. Pour un entier a, l'expression $taille(a) - 1$ donne le bits de poids le plus fort (et valant 1) dans l'écriture binaire de `a`.

1. Puisque chaque joueur choisit à chaque tour un vase et peut y retirer s'il le souhaite un seul fruit, tous les m-uplets de la forme $(y_1, y_2, \dots, y_m)$ avec $0 \leqslant y_i \leqslant x_i$ ($i \in [\![1, m]\!]$) sont des configurations possibles du jeu. Aussi le nombre de configurations possibles est :

$$\prod_{i=1}^{m} (1 + x_i) \ .$$

(on a compté aussi la configuration finale, où tous les vases sont vides.)

Le nombre de configurations du jeu varie linéairement en fonction du nombre de fruits x_i dans le seul vase i, et exponentiellement en fonction du nombre m de vases.

2. Calcul de $G(10,5,2,4)$.

10	=	1	0	1	0
5	=	0	1	0	1
2	=	0	0	1	0
4	=	0	1	0	0
a	=	1	0	0	1

Ainsi on a $G(10,5,2,4) = a = 9$.

3. Fonction `taille`.

```
def taille(x):
    a = 2
    n = 1
    while a <= x:
        a *= 2
        n += 1
    return n
```

La complexité a même ordre que le nombre de passages dans la boucle **`while`**, puisqu'il n'y a que des opérations élémentaires effectuées dans et hors de la boucle. Après p passages dans la boucle, $a = 2^{p+1}$, et celle-ci s'arrête dès que $a > x$. Soit exactement $p = \lceil log_2(x) \rceil - 1$ passages dans la boucle. Ainsi la complexité de la fonction `taille` est logarithmique en fonction de x.

4. Fonction `binaire`.

```
def binaire(x,n):
    bits = [0]*n
    i = n-1
    while x > 0:
        if x%2 == 1:
            bits[i] = 1
        x //= 2
        i -= 1
    return bits
```

La complexité de la fonction `binaire` est en $O\big(\log_2(x) + n\big)$, logarithmique en fonction de x et linéaire en fonction de n.

Après p passages dans la boucle **`while`**, x est devenu $\lfloor x/2^p \rfloor$, dont on déduit que s'exécutent au plus $\lceil log_2(x) \rceil$ passages. D'autre part initialiser la liste `bits` nécessitent n opérations élémentaires. Toute les autres instructions sont des opérations élémentaires, d'où la complexité en $O\big(\log_2(x) + n\big)$.

5. Fonction `decimal`.

```
def decimal(a):
    x = 0
    for b in a:
        x = 2*x + b
    return x
```

La complexité est linéaire en fonction de la longueur de la longueur de `a`.

6. Fonction `Grundy`.

```
def Grundy(x):
    # Détermination du nombre de bits nécessaires
    n = 0
    for xi in x:
        if taille(xi) > n:
            n = xi
    # Somme par colonne modulo 2
    G = [0] * n
    for xi in x:
        B = binaire(xi,n)
        for i in range(n):
            G[i] = (G[i] + B[i]) % 2
    return decimal(G)
```

On a implicitement utilisé le fait que $\left(\sum_l x_l\right) \mod 2 = \sum_l (x_l \mod 2)$.

La recherche du nombre de bits nécessaires utilise la fonction `taille` et nécessite $O\left(\sum_i \log_2(x_i)\right)$ opérations élémentaires. Il faut aussi $O(n) = O\left(\max_i \log_2(x_i)\right)$ opérations pour initialiser la liste `G` et $O\left(m \times \left(\sum_i \log_2(x_i) + n\right)\right) = O\left(m \times \max_i \log_2(x)\right)$ opérations pour effectuer la sommation modulo 2 par colonne. La fonction a donc une complexité en $O\left(m \times \max_i \log(x_i)\right) = O\left(m \times log(x_{\max})\right)$ en notant $x_{\max} = \max_i x_i$.

7. Fonction `jouer_coup`.

```
def jouer_coup(x,i,z):
    y = x[:]
    y[i] -= z
    return y
```

8. Soit $(x_1, \ldots, x_i, \ldots, x_m)$ une configuration quelconque dans Y_0 ; c'est-à-dire que l'on a $G(x_1, \ldots, x_i, \ldots x_m) = 0$. La configuration suivante s'obtient en choisissant i entre 1 et m et en changeant la valeur de x_i en $0 \leqslant x'_i < x_i$. En particulier au moins un bit dans l'écriture binaire de x_i est inversé dans celle de x'_i, et sur la colonne correspondante le bit de $G(x_1, \ldots, x'_i, \ldots x_m)$ sera lui aussi inversé.

En particulier puisque $G(x_1, \ldots, x_i, \ldots, x_m) = 0$, c'est-à-dire qu'il s'écrit en binaire $\overline{00\cdots00}$, nécessairement l'écriture binaire de $G(x_1, \ldots, x'_i, \ldots, x_m)$ est différente de $\overline{00\cdots00}$ et donc

$G(x_1,\dots,x'_i,\dots,x_m) \neq 0$. Ainsi la configuration suivante $(x_1,\dots,x'_i,\dots x_m)$ est, elle, dans Y_1.

9. Soit $(x_1,\dots,x_i,\dots,x_m)$ une configuration quelconque dans Y_1 ; soit $i_1,\dots,i_p$ les poids de tous les bits égaux à 1 dans l'écriture binaire de $G(x_1,\dots,x_i,\dots,x_m)$, classés par poids décroissants. Nécessairement il existe $i \in [[1,m]]$ tel que x_i ait son bit de poids i_1 égal à 1. En inversant dans l'écriture binaire de x_i tous les bits de poids $i_1,\dots,i_p$, on obtient une entier x'_i qui nécessairement, puisque $x_i[i_1] = 1$ et i_1 est le poids le plus fort parmi $i_1,\dots,i_p$, vérifie $0 \leqslant x'_i < x_i$. Par ailleurs, ce faisant tous les bits de poids $i_1,\dots,i_p$, et seulement eux, dans l'écriture binaire de $G(x_1,\dots,x_i,\dots,x_m)$ sont inversés, et donc $G(x_1,\dots,x'_i,\dots,x_m) = 0$. Ainsi la stratégie consistant à retirer $x_i - x'_i \geqslant 1$ fruits dans le vase i pour la configuration $(x_1,\dots,x_i,\dots,x_m) \in Y_1$ aboutit à une configuration $(x_1,\dots,x'_i,\dots,x_m) \in Y_0$.

Exemple. Reprenons la configuration $(10,5,2,4) \in Y_1$ dont le calcul de la valeur par la fonction de Grundy a été effectuée à la question 2.

10	=	1	0	1	0
5	=	0	1	0	1
2	=	0	0	1	0
4	=	0	1	0	0
a	=	1	0	0	1

On a $G(10,5,2,4) = a = 9$.

Les bits égaux à 1 dans a sont ceux de poids 3 et 0. L'élément $x_1 = 10$ vérifie $x_1[3] = 1$, et on inverse ses bits de poids 3 et 0 pour obtenir $x'_i = \overline{0011} = 3$. La configuration $(3,5,2,4)$ a pour valeur par la fonction de Grundy :

3	=	**0**	0	1	**1**
5	=	0	1	0	1
2	=	0	0	1	0
4	=	0	1	0	0
a'	=	**0**	0	0	**0**

Ainsi le coup consistant à retirer $10-3 = 7$ fruits dans le vase 1 a permis d'obtenir une configuration dans Y_0. Ici, de plus, c'était le seul coup permettant de passer dans Y_0.

10. Passons à la preuve du résultat.

Le nombre total de fruits diminue strictement après chaque coup, et donc le jeu aura toujours un gagnant et un perdant. Un joueur a perdu lorsqu'il ne reste plus aucun fruit. En particulier sa configuration est dans Y_0.

Si un joueur dont vient le tour de jouer se trouve dans une configuration Y_1, il a (d'après la question 9) un coup à jouer de façon à ce que la configuration suivante soit dans Y_0. Par contre pour un joueur qui joue avec une configuration dans Y_0, quelque soit son coup, la configuration suivante qu'il laissera à l'adversaire sera dans Y_1 (d'après la question 8).

Ainsi si un joueur se trouve dans une configuration Y_1, une stratégie lui permet d'y rester jusqu'à la din du jeu en laissant l'adversaire dans une configuration Y_0. Il a donc une stratégie gagnante. Réciproquement, si un joueur a une stratégie gagnante, il ne peut pas se trouver dans une configuration dans Y_0, car autrement son adversaire aura une stratégie lui permettant de rester toujours dans une configuration Y_1, et donc de ne pas perdre.

Une stratégie gagnante, consiste donc si on se trouve à une étape dans la configuration Y_1, de jouer une coup de façon à passer à une configuration dans Y_0, et ce alors jusqu'à la fin du jeu.

11. On met en pratique l'argument appliqué à la question 9. Sur une configuration $(x_1, \dots, x_m)$ dans Y_1, on cherche les poids $i_1, \dots, i_p$ des bits valant 1 dans l'écriture binaire de $G(x_1, \dots, x_m)$, dont i_1 celui de poids le plus fort, et on modifie un x_i tel que $x_i[i_1] = 1$ en inversant dans son écriture binaire tous ses bits aux poids $i_1, \dots, i_p$.

```
def strategie(x):
    a = Grundy(x)
    if a == 0: # pas de stratégie gagnante
        return
    # Liste des poids valant 1 dans a
    I = []
    while a > 0:
        i = taille(a)-1
        I.append(i)
        a -= 2**i
    i1 = I[0]
    for j in range(len(x)):
        n = taille(x[j])
        bits = binaire(x[j],n)
        if n > i1 and bits[n-1-i1] == 1: # xi trouvé !
            inverse = bits[:]
            for k in I: # inversion dans xi des bits dans I
                inverse[n-1-k] = 1-inverse[n-1-k]
            return j+1, decimal(bits)-decimal(inverse)
```

L'appel de `Grundy(x)` a pour complexité $O\big(m \times \max_i \log(x_i)\big)$. La boucle **`while`** a pour complexité dans le pire des cas $O(\log_2(a)) = O\big(\max_i \log(x_i)\big)$. Enfin la boucle **`for`** a pour complexité dans le pire des cas $O\big(\sum_i \log(x_i)\big)$. Ainsi la complexité est dans le pire des cas $O\big(m \times \max_i \log(x_i)\big)$.

9) Enveloppe convexe de points du plan. . . . . D'après X-ENS 2015, MP-PC.

Le problème a pour objectif de calculer des enveloppes convexes de nuages de points dans le plan affine. On rappelle qu'un ensemble $C \subseteq \mathbb{R}^2$ est convexe si et seulement si pour toute paire de points $p, q \in C$, le segment de droite $[p, q]$ est inclus dans C. L'enveloppe convexe d'un ensemble $P \subseteq \mathbb{R}^2$, notée $Conv(P)$, est le plus petit convexe contenant P. Dans le cas où P est un ensemble fini (appelé nuage de points), le bord de $Conv(P)$ est un polygone convexe dont les sommets appartiennent à P, comme illustré dans la figure 1 ci-après.

Le calcul de l'enveloppe convexe d'un nuage de points est un problème fondamental en informatique, qui trouve des applications dans de nombreux domaines comme la robotique pour l'évitement de collisions, le traitement d'image pour la détection d'objets en 2D, le graphisme 3D pour l'accélération de rendu, la vérification formelle pour la détermination du risque qu'une variable dépasse sa capacité de stockage, etc...

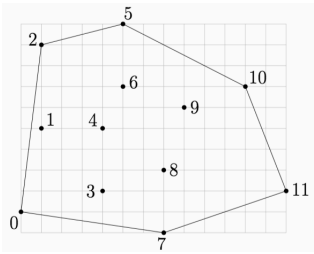


Figure 1

Nous allons écrire un algorithmes de calcul du bord de l'enveloppe convexe d'un nuage de points P dans le plan affine appelé *algorithme du paquet cadeau*; il consiste à envelopper le nuage de points P progressivement en faisant pivoter une droite tout autour. La complexité de cet algorithme est $O(nm)$, où n désigne le nombre total de points de P et m le nombre de ceux appartenant au bord de $Conv(P)$.

Sauf mention contraire, on n'aura pas à justifier des complexité des programmes écrits. Toutefois, il faudra veiller à ce que l'algorithme obtenu soit bien de complexité $O(nm)$.

Dans toute la suite on supposera que le nuage de points P est de taille $n \geqslant 3$ et en position générale, c'est-à-dire qu'il ne contient pas 3 points distincts alignés.

Ces hypothèses vont permettre de simplifier les calculs en ignorant les cas pathologiques, comme par exemple la présence de 3 points alignés sur le bord de l'enveloppe convexe. Nos programmes prendront en entrée un nuage de points P dont les coordonnées sont stockées dans un tableau `tab` à 2 dimensions, comme dans l'exemple ci-dessous qui contient les coordonnées du nuage de points de la figure ci-dessus :

i\j	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	0	1	1	4	4	5	5	7	7	8	11	13
1	0	4	8	1	4	9	6	-1	2	5	6	1

Précisons que les coordonnées, supposées entières, sont données dans une base orthonormée du plan, orientée dans le sens direct. La première ligne du tableau contient les abscisses, tandis que la deuxième contient les ordonnées. Ainsi, la colonne d'indice j contient les deux coordonnées du point d'indice j. Ce dernier sera nommé p_j dans la suite.

I. Préliminaires

1. Écrire une fonction `plusBas(tab,n)` qui prend en paramètre le tableau `tab` de taille `2×n` et qui renvoie l'indice `j` du point le plus bas (c'est-à-dire de plus petite ordonnée) parmi les points du nuage P. En cas d'égalité, votre fonction devra renvoyer l'indice du point de plus petite abscisse parmi les points les plus bas.

Sur le tableau exemple précédent, le résultat de la fonction doit être l'indice 7.

Dans la suite nous aurons besoin d'effectuer un seul type de test géométrique : celui de l'orientation.

Définition 1. *Étant donnés trois points p_i, p_j, p_k du nuage P, distincts ou non, le test d'orientation renvoie $+1$ si la séquence (p_i, p_j, p_k) est orientée positivement, -1 si elle est orientée négativement, et 0 si les trois points sont alignés (c'est-à-dire si deux au moins sont égaux, d'après l'hypothèse de position générale).*

Pour déterminer l'orientation de (p_i, p_j, p_k), il suffit de calculer l'aire signée du triangle, comme illustré sur la figure suivante. Cette aire est la moitié du déterminant de la matrice 2×2 formée par les coordonnées des vecteurs $\overrightarrow{p_i p_j}$ et $\overrightarrow{p_i p_k}$.

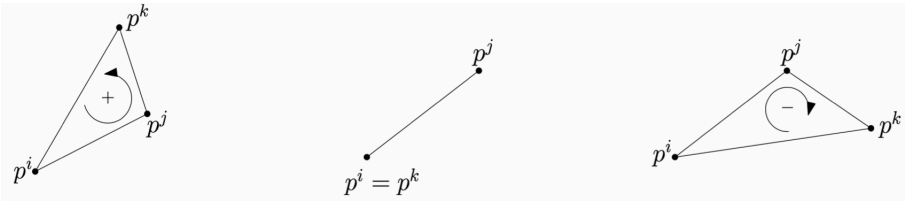


Figure 2

2. Sur le tableau `tab` précédent, donner le résultat du test d'orientation pour les deux choix d'indices suivants :

$$i = 0,\ j = 3,\ k = 4, \qquad \text{et} \qquad i = 8,\ j = 9,\ k = 10.$$

3. Écrire une fonction `orient(tab,i,j,k)` qui prend en paramètres le tableau `tab` et trois indices de colonnes, potentiellement égaux, et qui renvoie le résultat (-1, 0 ou $+1$) du test d'orientation sur la séquence (p_i, p_j, p_k) de points de P.

II. Algorithme du paquet cadeau

Cet algorithme a été proposé par R. Jarvis en 1973. Il consiste à envelopper peu à peu le nuage de points P dans une sorte de paquet cadeau, qui à la fin du processus est exactement le bord de $Conv(P)$. On commence par insérer le point de plus petite ordonnée (celui d'indice 7 dans l'exemple de la figure 1) dans le paquet cadeau, puis à chaque étape de la procédure on sélectionne le prochain point du nuage P à insérer.

La procédure de sélection fonctionne comme suit. Soit p_i le dernier point inséré dans le paquet cadeau à cet instant. Par exemple, $i = 10$ dans l'exemple de la figure 3.

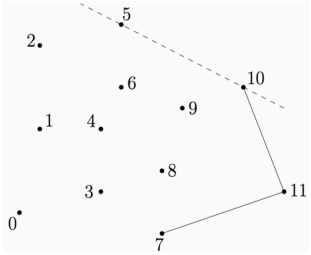


Figure 3

Considérons la relation binaire $\preceq$ définie sur l'ensemble $P \setminus \{p_i\}$ par :

$$p_j \preceq p_k \quad \Longleftrightarrow \quad \texttt{orient(tab,i,j,k)} \leqslant 0.$$

4. Justifier brièvement le fait que $\preceq$ est une relation d'ordre total sur l'ensemble $P \setminus \{p_i\}$, c'est-à-dire :

- (réfléxivité) pour tout $j \neq i$, $p_j \preceq p_j$,
- (antisymétrie) pour tous $j, k \neq i$, $p_j \preceq p_k$ et $p_k \preceq p_j$ implique $j = k$,
- (transitivité) pour tous $j, k, l \neq i$, $p_j \preceq p_k$ et $p_k \preceq p_l$ implique $p_j \preceq p_l$,
- (totalité) pour tous $j, k \neq i$, $p_j \preceq p_k$ ou $p_k \preceq p_j$.

Ainsi, le prochain point à insérer (le point d'indice 5 dans la figure 3) est l'élément maximum pour la relation d'ordre $\preceq$. Il peut se calculer en temps linéaire (c'est-à-dire majoré par une constante fois n) par une simple itération sur les points de $P \setminus \{pi\}$.

5. Décrire une réalisation en Python de la procédure. Elle prendra la forme d'une fonction `prochainPoint(tab, n, i)`, qui prend en paramètre le tableau `tab` de taille 2×`n` ainsi que l'indice `i` du point inséré en dernier dans le paquet cadeau, et qui renvoie l'indice du prochain point à insérer. Le temps d'exécution de votre fonction doit être majoré par une constante fois `n`, pour tous `n` et `i`. La constante doit être indépendante de `n` et `i`, et on ne demande pas de la préciser.

6. Décrire à la main le déroulement de la procédure `prochainPoint` sur l'exemple de la figure 3. Plus précisément, indiquer la séquence des points de $P \setminus \{p_{10}\}$ considérés et la valeur de l'indice du maximum à chaque itération.

On peut maintenant combiner la fonction `prochainPoint` avec la fonction `plusBas` de la question 1 pour calculer le bord de l'enveloppe convexe de P. On commence par insérer le point p_i d'ordonnée la plus basse, puis on itère le processus de mise à jour du paquet cadeau jusqu'à ce que le prochain point à insérer soit de nouveau p_i. À ce moment-là on renvoie le paquet cadeau comme résultat sans insérer p_i une seconde fois.

Un détail technique : comme la taille du paquet cadeau augmente peu à peu lors du processus, et qu'à la fin elle peut être petite par rapport au nombre n de points de P, nous stockerons les indices des points du paquet cadeau dans une liste. Par exemple, sur le nuage de la figure 1, le résultat sera la liste `[7, 11, 10, 5, 2, 0]`.

7. Écrire une fonction `convJarvis(tab,n)` qui prend en paramètre le tableau `tab` de taille $2 \times n$ représentant le nuage P , et qui renvoie une liste contenant les indices des sommets du bord de l'enveloppe convexe de P, sans doublon. Le temps d'exécution de votre fonction doit être majoré par une constante fois nm, où m est le nombre de points de P situés sur le bord de $Conv(P)$.

8. Justifier brièvement le temps d'exécution de l'algorithme du paquet cadeau.

1. Ne pas se tromper dans les indices de lignes et colonnes. Recherche de minimum avec une condition supplémentaire à tester en cas d'égalité ; utiliser un branchement **`if ... elif`**.
4. Attention, la transitivité est fausse (erreur grossière dans l'énoncé). Ne pas en tenir compte dans la suite.
5. Recherche de maximum pour $\preceq$. Utiliser la fonction `orient`.
7. Appeler la fonction `prochainPoint` au sein d'une boucle **`while`**. Attention à ne pas insérer le point p_i deux fois.

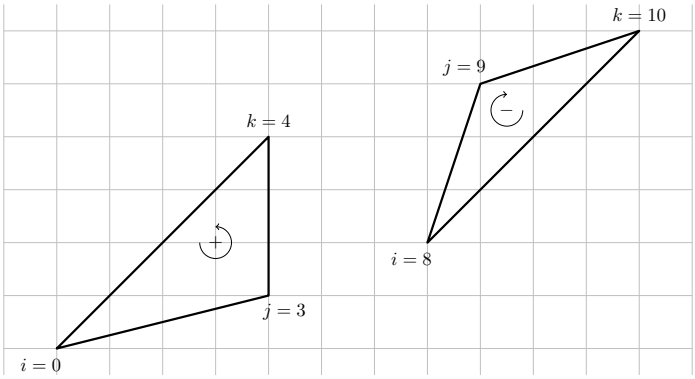
1. Fonction `plusBas`. Dans le tableau `tab` passé en paramètre, le second indice est celui des points, le premier permet d'accéder aux abscisses et ordonnées d'un point.

```
def plusBas(tab,n):
    iMin = 0
    for i in range(1,len(tab[0])):
        if tab[1][i] < tab[1][iMin]:
            iMin = i
        elif tab[1][i] == tab[1][iMin]:
            if tab[0][i] < tab[0][iMin]:
                iMin = i
    return iMin
```

2. On a tracé les 2 triangles, pour les points d'indices dans `tab` :

$$i = 0,\ j = 3,\ k = 4$$
$$i = 8,\ j = 9,\ k = 10.$$

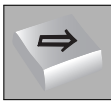
Ainsi le premier est orienté positivement, le test d'orientation renvoie $+1$.
Le deuxième est orienté négativement, le test d'orientation renvoie -1.



3. Fonction `orient(tab,i,j,k)` : on calcule le déterminant d'une matrice 2×2 ; leur signe détermine la valeur du test d'orientation.

```
def orient(tab,i,j,k):
    a = tab[0][j]-tab[0][i]
    c = tab[1][j]-tab[1][i]
    b = tab[0][k]-tab[0][i]
    d = tab[1][k]-tab[1][i]
    det = a*d-b*c
    if det > 0:
        return +1
    elif det < 0:
```

```
        return -1
    else:
        return 0
```



$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc.$$

4. Justification que $\preceq$ est une relation d'ordre total sur $P \setminus \{p_i\}$.

– Réfléxivité. Soit $j \neq i$: `orient(tab,i,j,j)`$= 0$ donc $p_i \preceq p_j$.

– Antisymétrie. Soient $j, k \neq i$; $p_j \preceq p_k$ et $p_k \preceq p_j$ implique `orient(tab,i,j,k)`$\leqslant 0$ et `orient(tab,i,k,j)`$\leqslant 0$; or `orient(tab,i,k,j)`$=$ $-$`orient(tab,i,j,k)`. Ainsi `orient(tab,i,j,k)`$= 0$, donc p_i, p_j et p_k sont alignés, et puisqu'on a supposé que les points de P étaient en position générale, nécessairement $j = k$.

– Transitivité. La propriété est fausse, comme le montre l'exemple :

$$p_i = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad ; \quad p_j = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad ; \quad p_k = \begin{pmatrix} -1 \\ -1 \end{pmatrix} \quad ; \quad p_l = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

En effet :

$$\det(\overrightarrow{p_i p_j}, \overrightarrow{p_i p_k}) = \begin{vmatrix} 1 & -1 \\ 0 & -1 \end{vmatrix} = -1 \leqslant 0 \quad ; \quad \det(\overrightarrow{p_i p_k}, \overrightarrow{p_i p_l}) = \begin{vmatrix} -1 & -1 \\ -1 & 1 \end{vmatrix} = -2 \leqslant 0$$

Ainsi $p_j \preceq p_k$, $p_k \preceq p_l$. Pourtant $p_j \not\preceq p_l$:

$$\det(\overrightarrow{p_i p_j}, \overrightarrow{p_i p_l}) = \begin{vmatrix} 1 & -1 \\ 0 & 1 \end{vmatrix} = 1 \geqslant 0$$

On a pourtant bien $j, k, l \neq i$ et p_i, p_j, p_k, p_l en position générale.

– Totalité. Triviale puisque $\det(\overrightarrow{p_i p_j}, \overrightarrow{p_i p_k})$ est un réel, soit positif, soit négatif.

5. Fonction `prochainPoint`.

```
def prochainPoint(tab,n,i):
    if i == 0:
        iMax = 1
    else:
        iMax = 0
    for k in range(n):
        if orient(tab,i,iMax,k) < 0:
            k = iMax
    return iMax
```

6. Après avoir inséré initialement le point d'indice 10, le déroulement de l'exécution de `prochainPoint(tab,12,10)` verra l'état des variables `k` et `iMax` évoluer de la façon suivante :

k	iMax
	0
1	1
2	2
3	2
4	2
5	5
6	5
7	5
8	5
9	5
10	5
11	5

7. Fonction `convJarvis`.

```
def conJarvis(tab,n):
    L = [plusBas(tab,n)]
    recherche = True
    i = L[0]
    while recherche:
        i = prochainPoint(tab,n,i)
        if i != L[0]:
            L.append(i)
        else:
            recherche = False
    return L
```

8. La complexité de la fonction est bien en $O(nm)$. À chaque passage dans la boucle **`while`**, on insérera un nouveau point du bord de $Conv(P)$ dans la liste, soit au plus m passages, au cours duquel on appellera la fonction `prochainPoint(tab,n,i)` dont la complexité est $O(n)$. Le remplissage de la liste `L` prendra en tout $O(m)$ opérations élémentaires. Tout le reste est constitué d'opérations élémentaires.