Informatique commune - Résumé de la première année

Lycée du Parc

1 Syntaxe Python

En Python, on fera toujours attention à l'indentation (c'est-à-dire les espaces en début de ligne). C'est l'indentation qui permet de définir les blocs (avec `def`, `for`, `while`, `if`, `elif`, `else`).

1.1 Affectation

```
# Ceci est un commentaire, qui ne sera pas évalué / exécuté par Python
variable = une_grosse_expression                          # Affectation classique
a, b, c = nouvelle_valeur_pour_a, idem_pour_b, idem_pour_c  # Affectations parallèles
```

Pour l'affectation, on aura toujours les noms de variables à gauche, et les expressions (calculs) à droite du symbole `=`

1.2 Types et Opérateurs

En Python, toutes les valeurs manipulées ont un type : `bool` (booléens, inventés par *George Boole*,"oo" se prononce "ou"), `int` (entiers, *integer*), `float` (nombres à virgule flottante), `str` (chaîne de caractère avec des guillemets, *string*), `list` (listes, en fait des tableaux redimensionnables), `tuple` (n-uplets), fonctions, ...

On peut demander à Python de convertir une valeur d'un type dans un autre type avec les fonctions suivantes :

```
>>> int("123")     # Convertit une chaîne de caractères ou un flottant en entier
123
>>> str(123)       # Convertit un entier ou un flottant en chaîne de caractères. Pratique pour l'affichage.
"123"
>>> float("123.4")
123.4
>>> 123 + 456, "123" + "456", int("123") + int("456"), str(123) + str(456)
(579, "123456", 579, "123456")
```

1.2.1 Nombres : entiers (int) et flottants (float)

```
>>> 1 + 1 (addition), 2 * 2 (produit), 3 ** 3 (puissance), 7 / 2 (division classique)
(2, 4, 27, 3.5)
>>> 7 // 2 (quotient de la division euclidienne), 7 % 2 (reste de la division euclidienne)
(3, 1)
>>> abs(-3.14) # valeur absolue
3.14
```

Remarque sur la division : en Python 2, la division dépendait du type des opérandes (`int` ou pas). On retiendra que si le résultat attendu est entier on utilise la division euclidienne, si le résultat attendu n'est pas forcément entier, on utilise la division classique.

1.2.2 Booléens

En Python : `True` (vrai) et `False` (faux), avec majuscule et sans guillemets


```
# Opérateurs de comparaison
>>> 1 + 1 == 2, 981 >= 1000, 981 <= 1000, 1 != 1 (différent de)
(True, False, True, False)
>>> n % d == 0 # Pour tester si n est divisible par d, on teste si le reste de la division euclidienne est nul
```

On peut faire des calculs avec les booléens (`a and b`, `a or b`, `not a`)

```
>>> (1 == 1) and (1 == 2), not True # Dans le doute, toujours parenthèser.
(False, False)
```

Attention, l'opérateur `and` ne s'utilise qu'avec les booléens ! Ça ne veut pas dire "je veux faire tel truc" `and` "tel autre truc"...

1.2.3 Listes / Tableaux

Le type `list` en python permet de représenter des séquences de valeurs. Pour un tableau de n cases, les cases sont numérotées de 0 à $n-1$ (de gauche à droite), et aussi de -1 à $-n$ (de droite à gauche). On évitera de nommer une liste `l` pour ne pas confondre avec `1`.

```
>>> t = [7, 12, 'ok', 9]
>>> len(t) # taille = length
4
>>> t[0], t[1], t[-1] # t[-1] pour le dernier élément
(7, 12, 9)
>>> t[2] = 42 # On peut modifier un élément : le type 'list' est mutable
>>> t[4]
# ERREUR : index out of range
>>> t
[7, 12, 42, 9]
>>> t.append(-3)
>>> t
[7, 12, 42, 9, -3]
>>> reversed(t)   # Ne modifie pas t : renvoie une copie à l'envers
[-3, 9, 42, 12, 7]
>>> sorted(t)     # Ne modifie pas t : renvoie une copie triée
[-3, 7, 9, 12, 42]
>>> a.reverse()   # Modifie t : "renverser"
>>> a.sort()      # Modifie t : "trier"
```

On remarque que `truc.append(bla)` ne renvoie rien, et donc on n'écrit jamais `t = t.append(e)` sous peine d'effacer `t` ni, dans une fonction, `return t.append(truc)` mais `return t + [truc]`

Pour obtenir plusieurs éléments on utilise le slicing : `t[a:b]` crée une copie des éléments de `t` depuis l'indice `a` inclus jusqu'à l'indice `b` exclu.

On peut concaténer des tableaux avec l'opérateur `+` :

```
>>> t = [1, 2, 3]
>>> t + [4]
[1, 2, 3, 4]
>>> 2 * t  # à connaître, et ne pas confondre avec les tableaux numpy (array)
[1, 2, 3, 1, 2, 3]
```

Listes en compréhension : pour créer un tableau à double entrée de taille $n \times m$ (i.e. n listes de taille m, ou "n lignes et m colonnes") on utilisera `t = [[0 for i in range(m)] for j in range(n)]`.

```
>>> [i * i for i in range(10) if i % 2 == 0]
[0, 4, 16, 36, 64]
>>> [[(i, j) for i in range(10, 12)] for j in range(4)]
[[(10, 0), (11, 0)], [(10, 1), (11, 1)], [(10, 2), (11, 2)], [(10, 3), (11, 3)]]
```

1.2.4 Tuples (ou n-uplets en bon français)

Comme les listes : contient une séquence de valeurs, mais pas de `append`, et on ne peut pas modifier ses éléments (les tuples sont non mutables).

```
>>> mon_tuple = (84, 07, 24)
>>> mon_tuple[0]
84
```

intervient aussi dans l'affectation multiple : `a, b = 1, 2` est pareil que `(a, b) = (1, 2)`

1.2.5 Ensembles

Des ensembles non ordonnés, sans doublons. La plupart des opérations sont très rapides (O(1))

```
>>> elements = set()
>>> elements.add(1)
>>> elements
{1}
>>> 1 in elements, 2 in elements
(True, False)
```

On peut convertir des `list`, `tuple` et `set` vers une autre stucture de donnée grâce aux fonctions éponymes.

1.2.6 Chaînes de caractères

```
# Plusieurs syntaxes pour la même chose
a = "Bonjour, je m'appelle ... \n"
b = 'Bonjour, je m\'appelle ... \n'
c = """ On peut utiliser des guillemets triples
pour écrire sur plusieurs lignes (et aussi parfois des commentaires...)
"""
```

On peut lire les différents caractères (lettres) d'une chaîne de caractère comme dans un tableau

```
>>> machaine = "plof"
>>> machaine[0]
'p'
```

mais pas modifier ses éléments ! (le type 'str' (string) est non mutable en Python, ce n'est pas le cas dans tous les langages)

```
>>> machaine[0] = 'b'
# ERREUR : le type 'str' est non mutable
```

Pour "modifier" une chaîne de caractère, on la remplacera par une nouvelle chaîne construite pour l'occasion :

```
>>> s = "monjour"
>>> s = "b" + s[1:]
>>> s
"bonjour"

for lettre in s:
    print(lettre) # Affiche séparément chaque lettre de "bonjour"

# Afficher un joli triangle
for i in range(10):
    ligne = "" # initialisation à la chaîne vide
    for j in range(i):
        ligne = ligne + str(j) + ""
    print(ligne)
```


1.3 Fonctions

C'est vraiment important !

Pour des raisons de lisibilité, de modularité et de c'est-comme-ça-au-concours, on prend rapidement l'habitude de définir des fonctions. On évitera d'appeler toutes les fonctions avec le même nom (`fonction1`, `fonction2`... est à proscrire).

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

def maxi(x, y):
    if x >= y:
        return x # l'instruction return renvoie un résultat et permet de sortir de la fonction
    else:
        return y # ça n'a de sens que dans une fonction bien entendu...

def moyenne(t):
    return sum(t) / len(t) # Quand on peut, on réutilise d'autres fonctions définies avant !
```

1.4 Boucles 'for' / Pour

Les boucles permettent de répéter une action : on a les boucles conditionnelles (`while`), et les autres (`for`).

On retiendra (même s'il y a des exceptions) : on utilise une boucle `for` quand on sait a priori combien d'itérations (= tours de boucle) on doit faire, et `while` quand on ne sait pas a priori quand on va s'arrêter (par exemple quand on veut sortir prématurément d'une boucle).

```
t = [9, 12, 4]
for x in t:   # la variable x va prendre les valeurs 9, 12, puis 4
    print(x)  # Indentation !!!
```

Quand on veut des valeurs entières régulièrement espacées, on utilisera `range`

```
>>> range(3, 23, 5) # Les valeurs de 3 (inclus) à 23 (exclu) par pas de 5
[3, 8, 13, 18]
# En python 3, pour obtenir le contenu d'un range on peut utiliser 'list(range(...))'
```

on utilisera donc `range` quand on veut accéder aux éléments d'un tableau en utilisant les indices des cases :

```
for i in range(len(t)): # i prend les valeurs de 0 à len(t) - 1
                        # ce qui correspond exactement aux indices des cases de t
    ... # On peut utiliser i pour les indices et t[i] pour les valeurs

for (i, x) in enumerate(t):
    ... # On peut utiliser i pour les indices et x pour les valeurs
```

1.5 Boucles 'while' / Tant que (boucles conditionnelles)

À retenir : on utilise des boucles `while` quand on ne sait pas a priori combien d'itérations on va faire avant de sortir de la boucle, ou pour sortir prématurément grâce à un booléen.

```
while condition: # condition est un booléen (True ou False) ou un calcul qui renvoie un booléen
    faire_des_trucs
```

Important : la condition est un booléen qui est réévalué à chaque itération : si le booléen vaut `True` on recommence une itération, si il vaut `False` on sort de la boucle.

Attention : contrairement à la boucle for, une boucle conditionnelle ne gère pas du tout de variables. On peut toujours remplacer une boucle `for` par une boucle `while` mais le contraire n'est pas toujours possible.

```python
i = 0
while i <= 12:
   print(i)
   i = i + 1
```

est équivalent à

```python
for i in range(0, 13):
    print(i)
```

- on sort de la boucle `while` la première fois que la condition (un booléen, donc) est fausse.
- l'ordinateur n'est pas intelligent et ne réfléchit pas : il recalcule le booléen à chaque nouvelle itération.
- Pour prouver qu'une boucle fait bien ce qu'on veut, on utlisera un invariant de boucle : une propriété qui est vraie quand on entre dans la boucle et qui reste vraie à la fin d'une itération si elle était vraie au début de cette itération.

1.6 Conditions

Les instructions python `if` (si), `elif` (contraction de `else if`), `else` (sinon) permettent de faire des distinctions de cas :

```python
if condition_0:   # condition_0 est un booléen : True ou False. Je l'ai déjà dit ?
   bloc_0
elif condition_1:
   bloc_1
elif condition_2:
   bloc_2
elif condition_n:
   bloc_n
else:             # Il n'y a pas de condition à tester avec else...
   bloc_sinon
```

Il est important de comprendre le fonctionnement de `if`, `elif`, `else` pour faire des distinctions de cas. Dans des cas très particuliers (dans des fonctions avec `return`), on peut se passer de `else` : dans le doute on évitera d'écrire n'importe quoi.

1.7 Modules

On passe notre temps à réinventer la roue, mais de nombreuses fonctions existent déjà en python. Plusieur syntaxes un peu différentes

```python
from math import sqrt, log, log10
x = sqrt(2)
# log correspond au logarithme néperien (base e), et log10 au logarithme en base 10

from math import *
x = sqrt(2) # On a accès à toutes les fonctions du module : un peu bourrin, à éviter

import math
x = math.sqrt(2) # Il faudra préciser math.le_nom_de_la_fonction à chaque fois

import math as mathematiques
x = mathematiques.sqrt(2) # Idem mais renommant le module, généralement pour un nom plus court
```

On pourra être amenés à utiliser :

```python
import math, random, numpy as np                                  # Les grands classiques à connaître
from scipy import constants, integrate, linalg, optimize, special # Les sous-modules de scipy

from matplotlib import animation, cm, colors, pyplot as plt       # Graphiques 2D
from mpl_toolkits.mplot3d import Axes3D                           # Graphiques 3D, cf oral de Centrale

from timeit import default_timer as time                          # Fonction 'time()' correcte
import turtle as T                                                # ;-)
```


2 Représentation des nombres

Dans l'ordinateur, tous les nombres sont représentés en base 2. Il faut comprendre comment ces nombres sont représentés car cela impose des limitations sur les calculs qui sont faisables de manière exacte ou pas, que ce soit sur les entiers (certaines de ces limites n'existent pas en python) ou les nombres à virgule (de nombreuses limitations).

2.1 Entiers

Les entiers positifs sont représentés en base 2 de manière classique : $\underline{1101}_2 = 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 = 8+4+0+1 = \underline{13}_{10}$

Remarque toujours vraie: dans la plupart des langages de programmation, le nombre de bits pour représenter un entier est fixé (64 bits par exemple), et on peut représenter uniquement 2^{64} valeurs différentes. Sur n bits on représente 2^n valeurs distinctes.

Convention : Quelles sont les valeurs représentées ? c'est uniquement une convention : par exemple de $-2^{63}+1$ à 2^{63} si on veut représenter des entiers positifs et négatifs. Ou de 0 à $2^{64}-1$ si tous les entiers sont positifs. Mais on pourrait imaginer n'importe quel ensemble de valeurs.

En python, on peut faire des calculs avec des entiers de taille arbitraire (pas vrai pour les flottants).

2.2 Entiers négatifs : complément à 2

Pour les nombres négatifs, dans l'ordinateur les entiers sont généralement représentés en "complément à 2" (Attention, le nombre de bits est fixé, ici 8 bits pour l'exemple) :

Exemple avec $n = 5$ sur 8 bits :

1. On part de la représentation de l'entier postif correspondant : $\underline{5}_{10} = \underline{00000101}_2$ en base 2 sur 8 bits
2. On inverse tous les bits (1 devient 0 et 0 devient 1) : en partant de 00000101 on obtient 11111010 par exemple
3. On ajoute 1 (avec des retenues si besoin, et on tronque à gauche à 8 bits si ça dépasse) : on obtient $\underline{111111011}_2$ pour la représentation de -5

- si on part de la représentation de -5 et qu'on applique les même opérations, on obtient celle de $+5$. Cette représentation est beaucoup utilisée en pratique car elle est compatible avec les additions et les soustractions sans distinguer les cas positifs et négatifs (et 0 a une seule représentation)
- En pratique on distingue facilement les entiers positifs (qui commencent par '0') et négatifs qui commencent par '1'.

2.3 Réels : en base 2

Il faut être à l'aise avec la représentation en base 2 classique. Dans cette section, on va parler de virgule pour représenter certains nombres en base 2, mais on verra juste après comment cela est représenté dans l'ordinateur en pratique.

$\underline{12,75}_{10} = 8+4+0+0,5+0,25 = 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^{-1} + 1\times 2^{-2} = \underline{110,11}_2$

On a décomposé le nombre précédent selon des sommes de puissances de 2 (positives pour la partie entière, négatives pour la partie décimale)

2.4 Flottants : dans l'ordinateur

Dans l'ordinateur, les nombres à virgule flottante sont représentés avec une convention très particulière (norme IEEE754 sur 64 bits par exemple). Pas besoin de connaître les détails, mais seulement la conclusion.

Il faut penser à la notation scientifique de votre calculatrice, mais en base 2 et avec quelques subtilités. Nous ne verrons pas les cas trop particuliers. Dans votre calculatrice, en base 10, vous avez l'habitudes des notations de type $-2345 = -2,345\times 10^3$:

$$\overbrace{-}^{signe}\ \overbrace{2,345}^{mantisse}\ \times 10\ \overbrace{3}^{exposant} \quad \text{(en base 10)}$$

- On peut représenter le signe par $s = 0$ ou $s = 1$ avec la convention 0 correspond à positif et 1 à négatif (pensez à $(-1)^s$)

- L'exposant : on veut pouvoir représenter des exposants positifs et négatifs. La convention est de réprésenter l'exposant par un nombre positif et de le "décaler" d'une certaine constante. Ce n'est pas à connaître par coeur.
- Pour la mantisse, on "coupe" (arrondir ? tronquer ? ce n'est pas important ici) après un certain nombre de "chiffres" (bits). On remarquera que le premier bit de la mantisse en base 2 es toujours 1 (si on avait $0,11... \times 2^3$ on se remènerait à $1,1... \times 2^2$ par exemple), donc en pratique on ne représentera pas le "1" avant la virgule.

En pratique : on reprend l'exemple de $\underline{12,75}_{10} = \underline{110,11}_2$

- Sur 64 bits, on a 1 bit pour le signe, 52 bits pour la mantisse, 11 bits pour l'exposant.
- On commence par écrire le nombre en base 2 comme en notation scientifique : $\underline{110,11}_2 = +\underline{1,1011}_2 \times 2^2$
- Signe : positif, représenté par `0`
- Exposant (pas besoin de tout comprendre, hein) : on doit représenter '2' mais en fait il y a un décalage de `1023` en 64 bits (c'est arbitraire), donc on écrit $(2 + 1023 = 1025)$ en binaire sur 11 bits : 10000000001
- Mantisse : `1,1011`, mais on jette le `1` avant la virgule et on ne garde que les bits après la virgule : `1011` et sur 64 bits : `101100000...` (52 bits en tout, arrondi si besoin)

Au final $\underline{12,75}_{10} = \underline{110,11}_2$ est représenté sur 64 bits par :

$$\overbrace{0}^{signe}\ \overbrace{10000000001}^{exposant}\ \overbrace{10110000000000000000000000000000000000000000000000000}^{\text{mantisse avec un bit implicite}}$$

2.5 À retenir sur les flottants

- La chose la plus importante à comprendre : la taille de la mantisse est fixée (52 bits explicites pour 53 bits), et donc la précision maximale est de 2^{-53} ce qui correspond en base 10 à 16 chiffres après la virgule ($53 \times log(2) = 15.955$). $2^{-53} \simeq 10^{-16}$
- Comme les calculs sont effectués en base 2 dans l'ordinateur, il est parfois difficile de comprendre certains résultats:
- En particulier, faire des tests d'égalité sur les flottants est souvent une très mauvaise idée !
- Au lieu de faire un test d'égalité, on pourra tester si une valeur est 'proche' de 0, avec les avantages et inconvénients qui vont avec…

```
>>> 0.1 * 0.1
0.010000000000000002
>>> 0.1 * 0.1 == 0.01
False
# En fait en base 2, le nombre 0.1 n'est pas représenté de manière exacte : on a déjà une valeur approchée
```

Erreurs liées au calcul :

- absorption : lors de l'addition de deux nombres ayant des ordres de grandeur très différents, perte d'information sur le plus petit des deux.
- cancellation : lors de la soustraction de deux nombres relativement proches, lorsque le résultat est petit par rapport aux opérandes.

```
# On peut travailler avec des nombres très petits ou très grands
>>> 10**-100 + 10**-100
2e-100
>>> 10**100 + 10**100
20000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000L
# Mais pas avec des ordres de grandeur différents (à cause de la précision de 53 bits)
>>> 1 + 10**(-3) -1   # presque ok
0.0009999999999998899
>>> 1 + 10**(-15) -1  # grosse erreur relative
1.1102230246251565e-15
>>> 1 + 10**(-16) -1  # pouf, disparu
0.0
```


2.6 Et en base 10 ?

On peut avoir la super idée méga intelligente suivante : et si on faisait directement les calculs en base 10 ?

Ce n'est pas très naturel pour l'ordinateur (tout est représenté avec des 0 et des 1), mais c'est très utile si on veut travailler avec une précision donnée en base 10. Il est existe un module python pour ça :

```
>>> from decimal import getcontext, Decimal # le constructeur Decimal (majuscule) du module decimal (sans)
>>> Decimal(0.1)                 # Convertit le flottant 0.1 en représentation décimale : on se rend compte que
                                 # 0.1 n'est pas représenté de manière exacte en flottant
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal("0.1")               # On peut utiliser une chaîne de caractères pour ne pas avoir ce problème
Decimal('0.1')                   # ici, on a vraiment 0.1
>>> Decimal("2") ** Decimal("0.5") # Calcul de racine de 2, avec 28 chiffres significatifs par défaut
Decimal('1.414213562373095048801688724')
>>> getcontext().prec = 10         # Pour changer la précision : ici 10 chiffres significatifs
>>> 100 * Decimal("2") ** Decimal("0.5")
Decimal('141.4213562')
```

2.7 Numpy

Dans `numpy`, les types sont un peu particuliers : il y a par exemple `np.ndarray` qui correspond à des tableaux à n dimensions. Ou le type `int32` : exactement comme les entiers de python, *mais* représentés sur 32 bits, donc on peut représenter uniquement des entiers entre $-2^{31} + 1$ et 2^{31} (soit 2^{32} valeurs différentes).

2.8 Stockage dans la vraie vie

On a déjà vu que l'élément de base de l'ordinateur est le bit (0 ou 1), mais la plupart du temps on utilise des octets (paquets de 8 bits) par exemple '00000101' pour représenter la valeur 5. Tout dans l'ordinateur (entiers, flottants, chaînes, listes, images, vidéo) est composé de valeurs représentées sur 1, 2, 3, 4 ou 8 octets (8, 16, 24, 32, 64 bits)

Par exemple pour représenter une couleur, on utilisera souvent un triplet de valeurs comprises entre 0 et 255 (donc au total 24 bits ou 3 octets).

Une question qui revient fréquement : quelle place faut-il en mémoire ou sur le disque dur pour représenter une image par exemple. Si on considère qu'on ne compresse par l'image et que chaque pixel (picture element) est représenté sur 3 octets, alors une image de taille 640 colonnes par 480 lignes utilisera $640 \times 480 \times 3 = 921600$ octets

On utilisera les préfixes du système international : 921600 octets = 921,6 kilooctets (Ko) = 0,9216 megaoctets (Mo). Comme en physique, en maths, en chimie, en SI…

Remarques :

- Ne pas confondre un bit (0 ou 1) avec un "byte" (la traduction anglaise de "octet")
- Avant 1998, on considérait qu'un kilooctet représentait $2^{10} = 1024$ octets, mais maintenant un kilooctet correspond à 1000 octets, et 1024 octets s'appelle un kibioctet.
- kilo = 10^3, méga = 10^6, giga = 10^9, téra = 10^{12}, péta = 10^{15}, exa = 10^{18}, zéta = 10^{21}, yotta = 10^{24}
- abbréviations associées : 1 kilooctet : 1Ko, 1 mégaoctet : 1 Mo, puis Go, To, Po, Eo, Zo, Yo
- kibi = 2^{10}, mébi = 2^{20}, gibi = 2^{30}… avec les abbréviations : 1 kibioctet : 1 Kio, 1 mébioctet : 1 Mio…

3 Algorithmes classiques sur les tableaux et chaînes de caractères

Un algorithme est :

- Une séquence finie d'instructions
- définie de manière formelle
- qui permet de résoudre un problème.

En général en informatique on distingue les tableaux (les cases d'un tableaux sont contigües et on peut accéder directement à n'importe quel élément, mais un tableau est de taille fixe) et les listes (pour parcourir une liste il faut regarder le premier élément puis le suivant puis le suivant puis…, mais il est facile de rajouter des éléments). En python, le type `list` est en fait plus proche d'un tableau, mais auquel on pourrait rajouter des éléments : donc un tableau redimensionnable.

3.1 Somme des éléments d'un tableau

```
def somme(t):
    """ Somme des éléments d'un tableau """
    som = 0            # 0 est le neutre pour l'addition
    for x in t:
        som = som + x  # On ajoute chaque élément successivement
    return som
```

3.2 Recherche d'un élément dans un tableau non trié

```
def appartient(t, elem):
    """ Renvoie True si l'élément elem appartient au tableau t, et False sinon """
    for x in t:
        if x == elem:
            return True # Dès qu'on trouve l'élément, on peut arrêter et répondre True
    return False        # Si on a testé tous les éléments du tableau sans succès, on répond False
```

Remarque : étant donné qu'on ne va pas forcément parcourir tout le tableau mais qu'on peut sortir prématurément, il peut sembler plus élégant d'utiliser une boucle `while`. Évidemment, il faut gérer les indices soi-même.

```
def appartient(t, elem):
    """ Renvoie un booléen disant si elem appartient à t ou pas """
    i, trouve = 0, False
    while i < len(t) and not trouve: # On sort à la fin du tableau ou quand on a trouvé !
        if t[i] == elem:
            trouve = True
        i = i + 1
    return trouve
```

3.3 Recherche du maximum

Astuce : éviter d'utiliser un mot-clé ou une fonction qui existe déjà dans python : ne pas utiliser `max` mais `maxi`

```
def maximum(t):
    """ Valeur maximum dans un tableau """
    maxi = t[0] # On initialise avec une valeur du tableau (t[0]) mais pas une valeur arbitraire (genre 0)
    for x in t:
        if x > maxi:
            maxi = t
    return maxi
```

La version précédente utilise directement les valeurs du tableaux ; on peut de manière équivalente utiliser les indices :

```
def maximum(t):
    maxi = t[0]
    for i in range(len(t)):
       if t[i] > maxi:
            maxi = t[i]
    return maxi
```

et pour garder l'indice du maximum : il faut utiliser la version avec les indices (`range(len(t))`)

```
def indice_maximum(t):
    indice = 0             # Au début le maximum correspond à l'élément numéro 0
    for i in range(len(t)):
       if t[i] > t[indice]:
            indice = i
    return i, t[i]         # Si on veut renvoyer l'indice et la valeur correspondante
```


3.4 Tester si un tableau est rangé dans l'ordre croissant

On compare chaque élément au suivant

```
def est_croissant(t):
    for i in range(len(t) - 1): # Pour ne pas sortir du tableau avec t[i+1]
        if t[i + 1] < t[i]:
            return False
    return True
```

3.5 Recherche de mot dans un chaîne

```
def sousmot(mot, chaine):
    """ Renvoie True ssi mot apparait dans chaine """
    lmot, lchaine = len(mot), len(chaine)
    for i in range(lchaine - lmot)
        correspond = True
        for j in range(lmot):
            if mot[j] != chaine[i + j]:
                correspond = False
        if correspond:
            return True
    return False
```

La complexité de la fonction précédente n'est pas bonne du tout ($O(n \times m)$ pour des tailles $n = |mot|$ et $m = |chaine|$, cf les boucles imbriquées) et on peut faire beaucoup mieux.

3.6 Dichotomie : recherche d'élément dans un tableau trié

Le mot "dichotomie" signifie "couper en deux". On peut utiliser cet algorithme pour chercher si un élément apparait dans un tableau trié. Si le tableau n'est pas trié, ça n'a pas de sens.

```
def dichotomie(tab, elem):          # Variantes, cf. cours.
    debut, fin = 0, len(tab) - 1
    while debut < fin:
        milieu = (debut + fin) // 2 # Division entière importante
        if elem <= tab[milieu]:
            fin = milieu
        else:
            debut = milieu + 1
    return elem == tab[debut]
```

Complexité : $c_n = 1 + c_{n/2}$, on en déduit : $c_n = O(log_2(n))$

4 Calculs scientifiques : valeurs approchées

Il faut avoir en tête la représentation des flottants et les erreurs de calcul liées aux arrondis.

Quand les résultats ne correspondent pas à ce qu'on attend, ça peut venir :

- On s'est trompé en écrivant le code python
- La modélisation du problème est mal faite
- On est dans un cas ou les erreurs de calcul (arrondis) sont trop importantes
- Il faut revoir le cours de physique / maths

4.1 Équation : Zéros d'une fonction

L'objectif est de trouver des zéros d'une fonction pour résoudre des équations du type $f(x) = 0$. Évidemment on peut souvent se ramener à ce type d'équations facilement : $g(x) = h(x) \Leftrightarrow g(x) - h(x) = 0$, wahou.

Il existe de nombreuses méthodes de résolution d'équation dont la dichotomie et la méthode de Newton.

- Dichotomie : pas très rapide, mais permet de trouver une solution dans un intervalle choisi à l'avance. On utilisera le cours de maths pour montrer qu'une solution existe avant de la chercher.
- Méthode de Newton : beaucoup plus rapide, mais ne fonctionne qu'à certaines conditions sur les dérivées de f. Difficile de "choisir" une solution.

Tout ça existe évidemment déjà dans python

```
>>> from scipy import optimize # optimize est un sous-module du module scipy
>>> optimize.bisect(f, a, b)   # Dichotomie
...
>>> optimize.newton(f, x0)     # Méthode de Newton
...
# il existe aussi 'brentq(f, a, b)' et 'root(f, x0)', cf. l'aide.
```

4.1.1 Dichotomie

Le mot "dichotomie" signifie encore "couper en deux"...

```
def dichotomie(f, epsilon, a, b):
    """ On cherche un zéro de la fonction f par dichotomie sur l'intervalle [a; b] à epsilon près """
    assert(f(a) * f(b) <= 0), "Il n'existe peut-être pas de solution dans [a; b]"
    while (b-a)/2. >= epsilon:  # On ne veut pas faire de test d'égalité avec 0.
        m = (a+b)/2.
        if f(a) * f(m) <= 0: # si f(a) et f(m) sont de signes contraires...
            b = m
        else:
            a = m
    return (a+b)/2.

>>> dichotomie(lambda x : x**2 - 2, 1e-7, 0, 3) # à 10^-7 près, sur [0, 3], on résout $x^2 - 2 = 0$
1.41421356797
```

Remarque : On ne parlera pas de complexité ici, mais de visitesse de convergence. En notant $a_0 = a, b_0 = b$ et plus généralement a_i, b_i au bout de i itérations, on a $b_i - a_i = \frac{b-a}{2^i}$ (l'intervalle est divisé en deux à chaque étape). Pour atteindre une précision $\epsilon > 0$, on doit avoir $\frac{b-a}{2^i} \leq \epsilon \implies i \geq log_2(\frac{b-a}{\epsilon})$. Si on regarde le résultat en base 2 : à chaque itération on gagne un chiffre significatif. Ce qui est bien mais pas top.

4.1.2 Méthode de Newton

Méthode classique pour trouver un zéro d'une fonction f : même principe que la méthode des sécantes, mais on utilise des tangentes. On part d'une valeur x_0 donnée par l'utilisateur, et on construit la suite définie par

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Écrivez l'équation de la tangente à $\mathcal{C}_f$ passant par $(x_n, f(x_n))$ et calculez les coordonnées du point d'intersection avec l'axe des abscisses. Et puis faites un dessin ;-)

- Il faut connaître la fonction dérivée de f (ou calculer une approximation avec un taux d'accroissement) et donc que f soit dérivable.
- Il n'est pas facile de savoir quand s'arrêter : souvent on s'arrête quand $|x_n - x_{n+1}|$ est "assez petit", mais cela ne correspond pas forcément à "x_n proche de la solution"

- Cette méthode fonctionne mal si la fonction f n'est pas régulière (cf. vitesse de convergence)
- On ne choisit pas l'intervalle d'étude comme on le peut avec la dichotomie

Vitesse de convergence : Si on cherche une racine α, on a $log|x_n - \alpha| \leq 2^n log(K|x_0 - \alpha|) - log(K)$ (cf cours de maths) avec $K = \frac{\max|f''|}{2\min|f'|}$, le min et le max étant pris sur l'intervalle où l'on travaille (que l'on ne connait pas forcément en informatique).

Donc pour que ça marche en pratique, on a besoin de f de classe $\mathcal{C}^2$ et pas trop irrégulière (f'' pas trop grande et f' pas trop petite).

4.2 Système d'équations : Pivot de Gauss

Exactement comme dans votre cours de maths : on a un système matriciel de la forme $A \cdot X = Y$

- On travaille ligne par ligne sur la matrice A, et attention à toujours effectuer les mêmes opérations sur la matrice colonne Y
- Pour des raisons de stabilité numérique (on veut éviter de diviser par une valeur proche de 0), on choisit à chaque fois le plus grand pivot possible en valeur absolue.
- Parfois on préférera faire une copie des matrices A et Y pour ne pas les modifier.

```
def echange_ligne(A, i, j):
    for k in range(len(A[i])):
        A[i][k], A[j][k] = A[j][k], A[i][k]

def transvection(A, i, j, mu):
    for k in range(len(A[i])):
        A[i][k] += mu*A[j][k]

def pivot_partiel(A, j0):
    i = j0 # le vainqueur provisoire
    for k in range(i+1, len(A)):
        if abs(A[k][j0]) > abs(A[i][j0]):
            i = k
    return i

def resolution_systeme(A, y):
    n = len(A)
    for i in range(n-1):
        j = pivot_partiel(A, i)  # On cherche le plus grand pivot dans la colonne.
        echange_ligne(A, i, j)   # On échange les lignes de A concernées...
        y[i], y[j] = y[j], y[i]  # ... et idem sur Y
        for k in range(i+1, n):
            mu = A[k][i]/float(A[i][i])
            transvection(A, k, i, -mu) # Transvection sur A...
            y[k] -= y[i]*mu            # ... et idem sur Y
    # A est maintenant triangulaire
    x = [0]*n
    for i in range(n-1, -1, -1):
        x[i] = (y[i]-sum(A[i][k]*x[k] for k in range(i+1,n)))/float(A[i][i])
    return x
```

Complexité : la plupart des opérations sont négligeables devant les tranvections. Il y a $(n-1) + (n-2) + ... + 2 = O(n^2)$ transvections et chaque transvection a une complexité en $\Theta(n)$. Au final, l'algorithme utilisé pour le pivot de Gauss est en $O(n^3)$.

4.3 Équations différentielles : Méthode d'Euler / ...

4.3.1 Principe : méthode d'Euler et EquaDiff d'ordre 1

Important : il faut faire le lien avec votre cours de mathématiques (problème de Cauchy), mais en ayant en tête les spécificités propres à la résolution numérique approchée :

- Les solutions trouvées sont des suites de points, et plus vraiment des fonctions...
- Les différents points trouvés n'appartiennent pas nécessairement à la même solution analytique
- Cette méthode est beaucoup moins précise que les solutions exactes de votre cours de maths/physique, mais applicable à des équadiffs plus compliquées !
- Dans la suite, on utilisera la notation $F(Y,t)$ parce que c'est cohérent avec `scipy.intergrate.odeint` mais dans votre cours de maths, c'est peut-être noté $F(t,Y)$. Adaptez-vous à l'énoncé.

Principe général : Soit fonction f de classe $\mathcal{C}^2$ définie sur un intervalle $[a,b]$. On subdivise $[a,b]$ en n sous-intervalles $[t_i, t_{i+1}[$ avec $a = t_0 < t_1 < \cdots < t_{n-1} = b$ et, sur chaque intervale en notant $h = t_{i+1} - t_i$ on *fait une approximation* (le dire à l'écrit !) :

$$y'(x) \simeq \frac{y(t_{t_{i+1}}) - y(t_i)}{h}.$$

Une fois cette approximation décidée, on oublie qu'on travaille avec des fonctions et on travaille avec une suite de points : $(t_i, y_i)_{i\in[|0;n-1|]}$, et pas $(t_i, y(t_i))$, c'est important.

On suppose de plus qu'on connait l'équation différentielle sous la forme $y' = F(y,t)$ ainsi que y_0 au temps t_0 et donc (t_0, y_0) (problème de Cauchy). On construit successivement les (t_i, y_i) (on connait déjà les t_i hein) grâce au *Schéma d'Euler explicite (progressif)* :

$$y_{i+1} = y_i + h \times F(y_i, t_i)$$

où F se lit directement sur l'EquaDiff, par exemple : $y' + y = sin(t)$ donne $F(y,t) = -y + sin(t)$, rien de difficile.

4.3.2 Autres méthodes

Il existe d'autres méthodes que la méthode d'Euler, souvent plus précises. Le principe reste le même : on fait une approximation sur la "pente" entre t_i et t_{i+1} et on utilise $y_{i+1} = y_i + h \times$ pente : la calcul de la pente peut nécessiter plusieurs approximations de la dérivée sur l'intervalle.

Voir Runge-Kutta d'ordre 4 dans la partie Code qui suit.

4.3.3 Ordre supérieur : exemples

On se ramène à l'ordre 1 en écrivant Y comme un vecteur colonne, ce qui correspond donc à un système d'équations.

Exemple 1 : Soit l'équation différentielle : $y'' + 3y - 4t = 0$. On note $Y = \begin{pmatrix} y \\ y' \end{pmatrix}$ où y et y' sont des fonctions de t, et on a alors $F(Y,t) = Y' = \begin{pmatrix} y' \\ y'' \end{pmatrix} = \begin{pmatrix} y' \\ -3y + 4t \end{pmatrix}$. La condition initiale est alors un couple $\begin{pmatrix} y_0 \\ y_0' \end{pmatrix}$, qu'on prendra pour un même t_0.

- On exprime y' en fonction de y et y' ce qui est trivial (on connait y'). On utilise l'équation différentielle pour exprimer y''.
- Cette méthode fonctionne pour n'importe quel ordre et aussi pour des EquaDiff plus compliquées :

Exemple 2 : A l'instant t_0, un mobile est en position $(x_0, y_0) = (0,0)$ et est lancé avec une vitesse initiale $v = 30km.h^{-1}$ avec un angle $\theta = 45°$. Il est soumis à une seule force, la pesanteur (on utilisera $g = 9,81m.s^{-2}$). Les fonctions étudiées sont $x(t)$ et $y(t)$, ainsi que leur dérivée et dérivée seconde respectives. On pose $Y = \begin{pmatrix} x \\ x' \\ y \\ y' \end{pmatrix}$ et on a $F(\begin{pmatrix} x \\ x' \\ y \\ y' \end{pmatrix}, t) = Y' = \begin{pmatrix} x' \\ 0 \\ y' \\ -9.81 \end{pmatrix}$ même si ça ne dépend pas explicitement de t. Pour Y_0, il faut penser à convertir la vitesse en mètres par secondes et l'angle en radians...

4.3.4 Code

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

def euler(F, y0, temps):
    """ Méthode d'Euler """
    y, t = np.array(y0, float), temps[0]
    val = [y]
    for k in range(len(temps) - 1):
        y = y + (temps[k+1] - temps[k]) * np.array(F(y, t))
        val.append(y)
    return val
```



```
def rk4(F, y0, temps):
    """ Runge-Kutta d'ordre 4 """
    t, y = temps[0], np.array(y0)
    LY = [y]
    for k in range(len(temps) - 1):
        t, h = temps[k], temps[k+1] - temps[k]
        p1 = np.array(F(y, t))
        p2 = np.array(F(y + p1 * h / 2, t + h / 2))
        p3 = np.array(F(y + p2 * h / 2, t + h / 2))
        p4 = np.array(F(y + p3 * h, t + h))
        pente = (p1 + 2 * p2 + 2 * p3 + p4) / 6
        y = y + h * pente
        LY.append(y)
    return np.array(LY)

def F(Y, t):
    y, yprime = Y
    return yprime, -3*y + 4*t

temps = np.arange(0, 10, 0.1)
Y0 = (1, 3) # Ordre 2 : penser à Y0 = (y0, y'0)

resultat = euler(F, Y0, temps) # ou rk4(F, Y0, temps)
```

Ça existe déjà dans Python :

```
>>> from scipy.integrate import odeint # Ordinary Differential Equation ; Integration
>>> odeint(F, y0, temps) # cette fonction de numpy utilise d'autres algorithmes que ceux vus en cours
```

4.4 Intégration

- La méthode des rectangles à gauche s'écrit simplement. On découpe l'intervalle $[a,b]$ en n intervalles égaux de taille $\frac{b-a}{n}$. Sur chaque intervalle, on approxime la fonction par une fonction constante :

$$R_n = \frac{b-a}{n} \sum_{i=0}^{n-1} f(x_i)$$

- Pour la méthode des trapèzes, on utilise une fonction affine,

$$T_n = \frac{b-a}{n} \sum_{i=0}^{n-1} \frac{f(x_i) + f(x_{i+1})}{2}$$

- La méthode de Simpson, on utilise un polynôme de degré 2, et on obtient

$$S_n = \frac{b-a}{n} \sum_{i=0}^{n-1} \frac{f(x_i) + 4 \times \frac{f(x_i)+f(x_{i+1})}{2} + f(x_{i+1})}{6}$$

```
def rectangles(f, a, b, n):
    """ Méthode des rectangles à gauche pour calculer l'intégrale de f sur [a, b] avec n rectangles"""
    somme = 0
    h = float(b - a) / n
    for k in range(0, n):     # Ici, on approxime avec le point "à gauche"
        somme += f(a + k * h) # On pourrait mettre f(a + (k + 0.5) * h) pour le milieu de l'intervalle :
    return h * somme          # attention, cela changerait un peu la vitesse de convergence (c'est mieux)

def trapezes(f, a, b, n):
    somme = 0
    h = float(b - a) / n
    for i in range(0, n):
        somme += h * (f(a + i * h) + f(a + (i + 1) * h)) / 2 # Revoir comment calculer l'aire d'un trapèze
    return somme
```

Ça n'a pas de sens de parler de complexité ici puisque on décide de faire n itérations dès le départ. On s'intéresse à la qualité de l'approximation et en particulier à quelle vitesse on s'approche de la solution :

Formule de Taylor-Lagrange (cf. cours de maths) : Si f est n fois dérivable sur un intervalle $[a,b]$, il existe $c \in]a,b[$ tel que :

$$f(b) = f(a) + \frac{f'(a)}{1!}(b-a) + \cdots + \frac{f^{(n-1)}(a)}{(n-1)!}(b-a)^{n-1} + \overbrace{\frac{f^{(n)}(c)}{n!}(c)^n}^{\text{pour } c \text{ bien choisi}}$$

Ici, on va appliquer Taylor-Lagrange à F une primitive de notre fonction f sur chaque intervalle $[x_k, x_{k+1}]$

Pour la méthode des rectangles à gauche, avec f de classe $\mathcal{C}^1$, sur chaque intervalle $[x_n, x_{n+1}]$, on a

$$\int_{x_k}^{x_{k+1}} f = F(x_{k+1}) - F(x_k) \overset{Taylor}{=} \frac{F'(x_k)}{1!}(x_{k+1}-x_k) + \frac{F''(c_k)}{2!}(x_{x+1}-x_k)^2 = f(x_k)(x_{k+1}-x_k) + \frac{f'(c_k)}{2!}(x_{x+1}-x_k)^2$$

avec $c_k \in]x_k, x_{k+1}[$ et on a de plus $x_{k+1} - x_k = \frac{b-a}{n}$, on a donc :

$$\int_{x_k}^{x_{k+1}} f - \frac{b-a}{n} \times f(x_k) = f'(c_k) \times \frac{(b-a)^2}{2! \times n^2}$$

On note M un majorant de $|f'|$ sur $]a,b[$ (donc des $|f'(c_k)|$), et on somme sur tout l'intervalle $[a,b]$ ($h := \frac{b-a}{n}$) :

$$|\int_a^b f - R_n| \leq M \sum_0^{n-1} \frac{(b-a)^2}{2! \times n^2} = M\frac{(b-a)^2}{2n} = O(h)$$

De même pour f de classe $\mathcal{C}^2$, on obtient

$$|\int_a^b f - T_n| \leq K\frac{(b-a)^3}{n^2} = O(h^2)$$

et pour la méthode de Simpson, on obtient

$$|\int_a^b f - S_n| \leq K\frac{(b-a)^5}{n^4} = O(h^4)$$

5 Utilisation de numpy

Le module `numpy` permet de faire des calculs sur des tableaux de nombres (matrices).

```
import numpy as np
```

Remarques :

- Ici, on décide d'utiliser les tableaux numpy (type `array`) et pas directement les matrices (type `matrix`) pour ne pas se faciliter la tâche… parce qu'on ne sait pas ce qu'on vous demandera au concours, hein.
- Il se trouve que la plupart des calculs avec numpy sont beaucoup plus rapides que les calculs avec des listes python classiques.
- Numpy impose que tous les éléments d'une matrice aient le même type (float32 par exemple) ce qui est une bonne chose !
- La plupart des méthodes vues dans cette section existent en fait déjà en python (dans `numpy`) : on réinvente la roue, c'est normal.

Important : l'utilisation des tableaux `numpy` (`array`) est parfois un peu différente des tableaux python (`list`) :

- Pour les tableaux à deux dimensions : `t[a, b]` au lieu de `t[a][b]`
- Certains opérations font des calculs "terme à terme" : `2 * [1, 2, 3]` donne `[2, 4, 6]` au lieu de `[1, 2, 3, 1, 2, 3]` en python. Sauf mention contraire on utilise par défaut les listes python à l'écrit.


```
# Matrices (en fait, array) avec numpy, utilisez l'aide
from numpy import array, dot, linspace
from numpy.linalg import solve, det, inv

# Quelques outils utiles dans scipy
from scipy.integrate import quad, odeint          # Intégration
from scipy.optimize import fsolve, bissect, newton # Résolution d'équations
from scipy import constants, integrate, linalg, optimize, special # sous-modules
```

6 Bases de données

Important : contrairement au raisonnement habituel en informatique (algorithmique), on ne cherche pas à savoir / comprendre comment les résultats sont calculés : le SGBD (Système de Gestion de Base de Données) est une boîte-noire. L'objectif est d'écrire des requêtes (questions) pour obtenir un résultat, mais pas de décomposer les étapes qui permettent d'y arriver

6.1 SQL : Structured Query Language

Schéma relationnel : `communes(id:texte, dep:texte, nom:texte, pop:entier)`

- attribut : nom de colonne ; domaine : type de donnée (integer, float, text)
- clé primaires (pas de doublons) ; clés étrangères (fait référence à une autre table)

```
SELECT attributs, calculs, agregats (AS ...)     -- penser à mettre des attributs de GROUP BY s'il y en a
FROM table
     JOIN table2 ON table.a = table2.b
     JOIN table3 ON table2.c = table3.d
WHERE condition1 AND condition2
GROUP BY attributs                               -- crée des agrégats
HAVING condition1 AND condition2                 -- Différent de WHERE : condition sur les agrégats
ORDER BY attribut ASC                            -- ou DESC pour l'ordre décroissant
```

- Fonctions d'agrégation : `MIN(attribut)`, `MAX(attribut)`, `AVG(attribut)`, `SUM(attribut)`, `COUNT(*)`, `COUNT(attribut)`, `COUNT(DISTINCT attribut)`
- On peut faire des calculs : `SELECT x * x + y * y FROM ...`
- On peut aussi faire des requêtes imbriquées
- Les jointures permettent de ne pas faire de produit cartésien ;-)

Opérations ensemblistes :

```
SELECT ...
FROM ...

UNION      -- ou INTERSECT, ou EXCEPT

SELECT ...
FROM ....
```

6.2 Algèbre relationnelle

Projection : $\pi_{attributs}(R)$, Sélection : $\sigma_{conditions}(R)$, Agrégation : ${}_{attributs}\gamma_{calculs}(R)$. R représente une relation : soit une table, soit une sous-requête. Ne pas confondre la projection (SELECT) et la sélection (WHERE, ou HAVING). Oui c'est bizarre.

Requêtes sans agrégation :

$$\pi_{attributs}(\sigma_{conditions}(R))$$

```
SELECT attributs
FROM matable
WHERE conditions
```

Requêtes avec agrégation :

$$\sigma_{postconditions}({}_{attributs}\gamma_{calculs}(\sigma_{precondtions}(R)))$$

```
SELECT dep, SUM(pop) AS s
FROM villes
WHERE pop > 10000
GROUP BY dep
HAVING s < 100000
```

Nom en SQL	Équivalent en Algèbre relationnelle
table	relation
colonne	attribut
ligne	enregistrement / tuple
SELECT colonne1, colonne2 FROM table	projection : $\pi_{colonne1,colonne2}(table)$
... WHERE condition1 AND condition2	selection : $\sigma_{condition1 \wedge condition2}(R)$
SELECT col1, col2, f(col3), g(col4) GROUP BY col1, col2	agrégation : ${}_{col1,col2}\gamma_{f(col3),g(col4)}(R)$
ORDER BY col ASC, ORDER BY col DESC	pas en Algèbre Relationnelle
t1 JOIN t2 ON t1.a = t2.b	jointure : $t1 \underset{t1.a=t2.b}{\bowtie} t2$

6.3 En python

Dans la vraie vie, il faut avant tout faire attention à la sécurité : voir bobby-tables.com.

```
import sqlite3
connexion = sqlite3.connect('cinema.db')       # On se connecte à la base de données
curseur = connexion.cursor()
requete = "SELECT * FROM acteurs ORDER BY nom" # Une requête SQL !
resultats = curseur.execute(requete)
for res in resultats:
        print(res)                             # ici, chaque résultat est un tuple de valeurs
connexion.close()
```

Outre le fait qu'on peut utiliser des bases de données directement dans python - ce qui est déjà bien, on peut en plus utiliser des variables python dans les requêtes SQL : c'est là que la notion de *sécurité* est ultra importante dans la vraie vie !

```
cmd = "SELECT * FROM acteurs WHERE nom = ? AND prenom = ?"
c.execute(cmd, ("Norris", "Chuck"))
```

7 Deuxième année, très très brièvement

7.1 Piles

Une pile d'assiettes, de crèpes, de feuilles : on ne peut accéder qu'à l'élément du dessus et pas aux autres, comme chacun le sait.

La notion de pile est une structure de donnée abstraite et on part du principe qu'on ne peut que : créer une pile vide (`creer_pile()`), empiler un élément (`push(p, e)`), dépiler un élément (`pop(p)` qui supprime et renvoie un élément), regarder le sommet de la pile (`sommet(p)` qui permet de regarder l'élément au sommet de la pile sans la modifier), tester si la pile est vide (`est_vide(p)` qui renvoie un booléen). L'objectif est d'abord de comprendre le concept, et dans un deuxième temps de voir différentes implémentations (avec une liste ou un tableau).


```
# Attention, ces fonctions n'existent pas dans python : on fait comme si.
>>> p = creer_pile()
>>> est_vide(p)
True
>>> push(p, 1)
>>> est_vide(p)
False
>>> push(p, 2)   # on met 2 au dessus de la pile
>>> sommet(p)    # on lit le sommet de la pile, mais sans la modifier
2
>>> pop(p)       # on supprime le sommet de la pile (2) et on récupère sa valeur
2
>>> pop(p)       # le nouveau sommet est 1
1
>>> pop(p)
# ERREUR : pile vide
```

Remarque : quand on s'impose d'utiliser les fonctions précédentes, on n'utilise pas `for x in pile:`, ça n'a pas de sens... on utilisera la plupart du temps `while not est_vide(pile):...`

```
def appartient(x, pile):
    while not est_vide(pile):
        if pop(pile) == x:    # 'pop' détruit les éléments !
            return True
    return False
```

7.2 Fonctions récursives

Les fonctions récursives sont des fonctions qui s'appellent elle-mêmes :

- Toujours penser au(x) cas de base !
- Généralement les appels récursifs se feront avec des valeurs de paramètres strictement plus petits (pour un "bon ordre" donné : dans $\mathbb{N}$, ou idem avec la taille d'un tableau, ou l'ordre lexicographique...)
- Il y a des avantages mais aussi des inconvénients : explosion de la complexité dans certains cas, taille de la pile d'appel
- Exemples classiques : Fibonacci, exponentiation rapide, tri fusion et tri rapide...
- Autres notions liées aux fonctions récursives : preuves de correction par récurrence, mémoïzation, backtracking.

7.3 Tris

On s'intéressera aux tris dits "par comparaison". En particulier pour la complexité, on calculera un grand-O du nombre de comparaisons.

- Tout (et même plus) sur le site : sorting-algorithms.com
- Tri par insertion, Tri sélection, Tri fusion (Merge), Tri rapide (Quick)

À retenir : la plupart des tris par comparaison usuels ont une complexité en $O(n^2)$ ou $O(n \times log(n))$. Il faut comprendre pourquoi tel tri a telle complexité, mais aussi comment et pourquoi ça fonctionne...

8 Complexité

Remarques :

- On étudiera généralement la complexité temporelle dans le pire cas : combien d'étapes de calcul sont nécéssaire pour résoudre un problème de taille n.

- On utilise la notation “grand-O” du cours de maths, mais souvent dans des cas plus simples car les suites étudiées sont positives, croissantes et simples : $O(n), O(n \times ln(n)), O(n^2)$.... Cela exprimera la complexité dans le pire cas.
- On utilisera parfois la notation $f = \Theta(g)$ qui signifie : $f = O(g)$ et $g = O(f)$. Cela signifie que le pire cas est aussi le cas général ; on n'est jamais plus efficace que le pire cas. C'est plus précis.
- La complexité n'a de sens que si on sait définir la taille du problème à résoudre : on travaille sur un tableau de taille n, on fait des opérations sur des nombres à n chiffres... (dichotomie sur un tableau, la plupart des algos rencontrés)
- Pour les algorithmes pour lesquels le nombre d'étapes de calcul est prédéfini, on ne parlera généralement pas de complexité mais de précision ou de vitesse convergence vers la solution (dichotomie pour chercher un zéro d'une fonction, intégration, Euler...)

8.1 Complexité facile : les boucles

```
for i in range(n):
    # calcul de f(i)
```

on fera simplement la somme : $\sum_{i=0}^{n-1} c_i$ où c_i correspond à la compelxité du calcul de $f(i)$. Pour les boucles `while` c'est le même principe mais parfois moins évident : on fera souvent une grosse approximation (quand c'est possible).

```
s = 0
for x in t: # tableau t de taille n
    s = s + x
```

on aura : $\sum_{i=0}^{n-1} 1 = n$. On notera $O(n)$ où même $\Theta(n)$ ici car on parcourt toujours tout le tableau sans exceptions.

```
s = 0
for i in range(n):
    for j in range(i):
        s = s + t[j] # pas forcément intéressant hein
```

on a : $\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i = \frac{(n-1)\times n}{2} = O(n^2)$

Parfois c'est plus compliqué : pour le crible d'Ératosthènes, on obtient :

$$\sum_{0<i<n,\ \text{i premier}} \frac{n}{i} \leq n \times \sum_{0<i<n} \frac{1}{i} = O(n \times ln(n))$$

On fait une grosse majoration en retirant “i premier”, puis on utilise un équivalent de la série harmonique ($H_n = \sum_{k=1}^{n} \frac{1}{k} = ln(n) + \gamma + o(1)$ où γ est une constante).

8.2 Complexité et récurrences (Diviser pour régner)

Divisier pour régner : on découpe le problème à résoudre en m sous-problèmes, puis on les résout (récursivement) et enfin on utilise les solutions des sous-problèmes pour résoudre le problème initial. On aura alors une complexité qui vérifie une relation de récurrence de la forme : $c_n = m \times c_{n/m} + f(n)$ où $f(n)$ représente le coût pour découper le problème initial en sous-problème *et* pour résoudre le problème en combinant / fusionnant les solutions partielles.

En pratique (tri rapide, tri fusion) on aura $m = 2$ et pour simplifier les calculs on supposera que le problème est de taille $n = 2^k$. De plus on aura $f(n) = O(n)$ dans ces deux cas : $c_n = 2 \times c_{n/2} + O(n)$ qui donnera $c_n = O(n \times ln(n))$. Astuce : quand on a une relation de récurrence du type $u_n = 2u_{n-1} + ...$, on peut se ramener à une somme téléscopique en étudiant $v_n := \frac{u_n}{2^n}$

Remarque : il peut sembler bizarre de ne pas connaître la complexité des cas de base, mais elle sera de toute façon bornée et ne modifie donc pas le résultat (notation grand-O).

Autre exemple de récurrence (mais pas Diviser pour régner) : pour la suite de Fibonacci, on a $c_n = 1 + c_{n-1} + 1 + c_{n-2}$. On peut poser $u_n = c_n + 2$ et on a la récurrence $u_n = u_{n-1} + u_{n-2}$ (suite récurrente linéaire). On obtient deux racines distinctes $\phi = \frac{1+\sqrt{5}}{2}$ et une autre racine dans $]-1, 1[$, et on en déduit que $u_n = O(\phi^n)$.

9 Miscélanées

9.1 Interagir avec l'utilisateur

Rappel : dans la plupart des cas on ne s'intéressera pas à interroger l'utilisateur / afficher un résultat, mais on utilisera simplement des fonctions qui prennent en entrée des paramètres et renvoient un résultat.

```
nom = input("Entrez votre nom svp : ")              # de type string en Python 3
age = int(input("Entrez votre âge svp : "))         # on essaie de convertir en entier avec int(...)
taille = float(input("Entrez votre taille svp : ")) # idem mais on voudrait un flottant
```

Utilisation de print : plusieurs syntaxes pour la même chose. La première méthode est pratique à l'écrit, la deuxième est plus pythonesque.

```
print("Bonjour " + nom + ", vous avez " + str(age) + " ans !") # On utilise str et la concaténation
print("Bonjour {}, vous avez {} ans !".format(nom, age))
print("Bonjour %s, vous avez %i ans !"%(nom, age))
```

9.2 Fichiers texte

Toujours dans le même ordre : ouvrir le fichier, lire/écrire, fermer le fichier

```
fich = open("mon_fichier.txt", "r") # Mode d'accès : r = read, w = write, a = append
for ligne in fich.readlines():      # Ou plus simplement "for ligne in fich:"
    print(ligne)
fich.close()
```

On peut aussi lire un fichier ligne par ligne avec `fich.readline()` (sans ‘s’ à la fin).

Très souvent quand on lit un fichier, on veut découper la ligne en “morceaux” et récupérer les valeurs des différents champs, par exemple :

```
>>> ligne = "0 ; 2.718281828459045 ; 3.141592653589793\n" # retour à la ligne '\n' à la fin de la ligne
>>> x, y, z = ligne.strip().split(';')                    # strip() retire '\n' et split() découpe la ligne
>>> x, y, z
('0 ', ' 2.718281828459045 ', ' 3.141592653589793')
>>> int(x), float(y), float(z)                            # Penser à utiliser int ou float si besoin
(0, 2.718281828459045, 3.141592653589793)
```

Et pour écrire, on utilise `write`

```
fich = open("resultats.txt", "a")   # 'append' : pour écrire à la fin du fichier sans l'effacer
LX = range(100)
LY = [f(x) for x in LX]
for i in range(len(LX)):
    fich.write(str(LX[i]) + ";" + str(LY[i]) + "\n") # ou mieux avec "...".format(...)
fich.close()
```

Attention à la syntaxe de `fichier = open(nom, mode)` (fonction qui renvoie un descripteur de fichier) qui est différente de `fichier.readline(), fichier.readlines(), fichier.write(chaine), fichier.close()` (méthodes du descripteur de fichier).

9.3 Graphiques

Le plus important est de comprendre le fonctionnement de `plot([x0, x1, x2, x3], [y0, y1, y2, y3])`.

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

LX = np.linspace(0, 1, 101) # 101 valeurs équiréparties entre 0 et 1 inclus
LY = [f(x) for x in LX]     # ou np.arange(0, 1, 0.01) (comme range mais avec des flottants)
plt.plot(LX, LY)            # l'avantage de linspace : on connait la taille du résultat
```

9.4 Probabilités

```
>>> from random import random, uniform, randint, choice, randrange
>>> help(...)
random()       -> x in the interval [0, 1).
uniform(a, b)  -> Get a random number in the range [a, b) or [a, b] depending on rounding.
randint(a, b)  -> Return random integer in range [a, b], including both end points.
choice(t)      -> Choose a random element from a non-empty sequence.
randange(a, b) -> Choose a random item from range(start, stop[, step]).
```

9.5 Images

Pour manipuler des images, on peut utiliser des modules comme the Python Image Library (PIL) : il est possibles de manipuler les images comme des tableaux `numpy` à 2 dimensions.

- la syntaxe du slicing n'est pas habituelle avec `numpy` : `t[a:b, c:d]` ou lieu de `t[a:b][c:d]`
- Faire attention aux conventions : `t[x, y]` ou `t[y, x]` ? être cohérent avec l'énoncé.
- le type des élément est homogène, par exemple `int8` : on fera attention dans ce cas aux dépassements quand on fait des calculs (additions, moyenne...)

9.6 Complexes

```
>>> complex(1, 3)
(1+3j)
>>> (1j) ** 2
(-1+0j)
>>> 1*j
# ERREUR : la variable 'j' n'existe pas, '1j' est une notation
```

9.7 Programmation Orientée Objet (POO)

Hors programme mais ça peut servir dans la vraie vie. Un exemple ci-dessous ; la seule grosse subtilité : le premier argument (appelé `self` par convention) correspond à l'objet lui-même et n'est pas utilisé explicitement. Regardez le nombre de paramètres de chaque fonctions et combien d'arguments sont utilisés en pratique (le premier est omis).

On définit une classe d'objets et les attributs et méthodes (fonctions) associés. On peut ensuite créer des instances de cette classe.

```
class Point():
    def __init__(self, x, y):      # Appelé quand on crée un objet (nouvelle instance de la classe)
        self.x = x                 # self.x et self.y sont des attributs propres à chaque objet
        self.y = y
    def distance_origine(self):    # Méthode qui renvoie un résultat
        return math.sqrt(self.x ** 2 + self.y ** 2)
    def deplacer_droite(self, dx): # Méthode qui modifie l'objet
        self.x += dx
    def __repr__(self):            # Méthode 'magique' qui sera appelée par 'print'
        return "(" + str(self.x) + ", " +  str(self.y) + ")"

>>> p1 = Point(1, 2)     # Appelle __init__ avec x = 1 et y = 2
>>> p1.distanceOrigine() # Remarquez qu'il n'y a pas d'arguments : on ne met pas 'self'
2.23606797749979
>>> print(p1)            # Python sait comment afficher p1 grâce à '__repr__'
(1, 2)
>>> p2 = Point(3, 4)     # Crée une autre instance de la classe Point : p1 et p2 sont différents
>>> p2.distanceOrigine()
5.0
>>> p2.deplacerDroite(3) # Appel avec dx = 3 : modifie les valeurs x et y dans p2
>>> p2.distanceOrigine()
```



```
7.211102550927978
>>> p1.distanceOrigine() # p1 n'a pas bougé
2.23606797749979
```

10 Exemples ultra classiques

On revoit les mêmes exemples en deuxième année mais généralement en version récursive. Pas toujours avec la même complexité.

10.1 Factorielle et Exponentiation naïve

```
def factorielle(n):
    p = 1 # neutre pour le produit
    for i in range(2, n): # On ne veut pas multiplier par 0...
        p = p * i
    return p

def exponentiation(x, n): # On peut faire beaucoup mieux hein...
    p = 1
    for i in range(n): # Faire n fois
        p = p * x      # Ou 'p *= x'
    return p
```

10.2 Suites définies par recurrence

Toujours faire attention ! Est-ce qu'on calcule bien les premiers termes ? Est-ce qu'on fait bien le bon nombre d'itérations ? Il faut réfléchir à chaque fois... Erreur classique : recopier l'énoncé sans réfléchir...

Exemple : $u_0 = 1$ et $u_{n+1} = n \times u_n + 3, n \in \mathbb{N}^*$

```
def exemple1(n):
    u = 1
    for i in range(0, n):
        u = i * u + 3          # Attention à ce que représentent 'n' et 'i'
    return u

def exemple2(n):
    u = 1
    for i in range(1, n + 1):
        u = (i - 1) * u + 3    # Il faut parfois adapter l'expression...
    return u
```

10.3 Suite de Fibonacci

```
def fibo(n):
    """ Calcule le n-ième terme de la suite de Fibonacci : F_0 = 0, F_1 = 1, F_{n+2} = F_{n} + F_{n+1} """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

11 Physique / Chimie / SI

De nombreux sujets de concours vont étudier des problèmes concrets de physique : en pratique il faudra pouvoir résoudre des équations ou des équations différentielles, intégrer et dériver des fonctions. À chaque fois avec des *méthodes proches du cours mais différentes.* D'où l'intérêt de comprendre : dichotomie, méthode de Newton, calcul d'intégrale, méthode d'Euler... !

Faire aussi attention à la rigueur : pour les questions de physique, utiliser le cours de physique (ex : lire un diagramme de phase) et pour les questions de maths, utiliser le cours de maths (ex : dire/prouver qu'une fonction est dérivable, ou de classe $\mathcal{C}^n$, avant de la dériver). Et surtout dire explicitement quand on fait des approximations !

Les problèmes concrets font souvent intervenir des unités, comme en cours de physique. La plupart du temps les calculs en python se font sans préciser l'unité du système international. Mais il faut quand même faire attention : parfois il faut convertir les unités. Ne pas hésiter à créer une fonction python :

```python
def mps(vitesse):
    """ Convertit une vitesse en km.h^-1 en m.s^-1 """
    return vitesse * 1000 / 3600
```