Mémento pour Python en CPGE

I - Utilisation pratique

Python, dans son usage courant, est un langage *interprété*, c'est-à-dire que les instructions sont exécutées au fur et à mesure de leur validation par l'utilisateur. Cela peut se faire en *conversationnel* dans une fenêtre appelée *shell* ("coquille" où les étapes de la session en cours sont mémorisées). Mais lorsqu'il s'agit d'exécuter un grand nombre d'instructions, notamment des définitions de fonctions destinées à être testées et modifiées, il est préférable de les regrouper dans un *script*, simple texte que l'on peut enregistrer dans un fichier. Tout *environnement de développement intégré* (EDI, ou IDE en anglais) digne de ce nom permet d'exécuter dans un *shell* toutes les instructions contenues dans un tel script, séquentiellement de la première à la dernière ligne.

C'est le cas avec **IDLE**, EDI rudimentaire fourni avec Python.
C'est aussi le cas avec **Pyzo**, EDI plus sophistiqué, fourni aux candidats passant l'oral de Centrale.

Dans Pyzo, Ctrl+0 place le curseur dans la zone *shell* et Ctrl+9 dans la zone *script*.
F5 lance dans le *shell* l'exécution du script en cours. Ctrl+F5 réinitialise le *shell* avant d'exécuter le script, ce qui permet accessoirement à Python de trouver les fichiers à lire dans le dossier où est enregistré le script, sans avoir besoin de préciser le chemin complet.

La zone *shell* de Pyzo offre plusieurs possibilités intéressantes, notamment :

- parcourir l'historique des commandes à l'aide des flèches "haut" et "bas"
- obtenir l'aide d'une fonction (à condition de connaître son nom !) en tapant dans le *shell* `help(nom)`, où `nom` est le nom de la fonction souhaitée. Si ladite fonction est dans un module qui n'a pas été chargé, `nom` doit être le "chemin" complet entre apostrophes, par exemple `help('numpy.linalg.eig')` ; noter que l'on obtient ainsi une aide en mode texte, pas très agréable à lire... Penser à consulter l'aide en ligne, au format `html` ou `pdf`, mieux présentée !

Pour tester les fonctions définies dans un script, deux solutions (qui ne s'excluent pas l'une l'autre) :

- inclure à la fin du script des instructions exécutant les tests (`print` permettant d'afficher des résultats dans le *shell*). Solution à privilégier si les tests impliquent plusieurs instructions... Dans Pyzo, Ctrl+R transforme les lignes sélectionnées en *commentaires* (les instructions ne seront pas exécutées), opération annulée par Ctrl+T. Cela est utile lorsque l'on veut tester séparément plusieurs morceaux de code
- exécuter le script définissant les fonctions, puis les tester en conversationnel dans le *shell*. Solution adaptée aux tests simples.

Attention ! Le *noyau de base* de Python connaît très peu de fonctions mathématiques, graphiques ou numériques. C'est pourquoi il faut souvent charger des outils provenant de *modules* (ou *bibliothèques*) complémentaires. Les plus usuels sont :

- `math`, où l'on trouve notamment `sqrt`, `exp`, `log` (le logarithme népérien !), `pi` et les fonctions trigonométriques, `floor` (la partie entière), `factorial`, etc.
- `matplotlib`, qui offre de nombreux outils graphiques, notamment dans le sous-module `pyplot`
- `numpy`, qui facilite le calcul numérique et la manipulation de tableaux comme les matrices, avec un sous-module `linalg` pour l'algèbre linéaire
- `scipy`, contenant des fonctions avancées de calcul scientifique, avec notamment un sous-module `integrate` pour les intégrales et les équations différentielles
- `sympy`, qui permet du calcul symbolique et du calcul numérique en *multiprécision* via le sous-module `mpmath`.

Deux syntaxes sont recommandées pour les importations :

1) Importation d'un module avec un *alias* : par exemple, `import matplotlib.pyplot as plt` permet d'utiliser les commandes de `pyplot` en les faisant juste précéder de l'alias et d'un point (`plt.plot()`, `plt.show()`...). L'importation sans alias est possible, mais il faut alors recopier le nom complet du module à chaque invocation.

2) Importation de certaines fonctions d'un module : par exemple, `from math import pi,sin,cos` permet d'utiliser `pi`, `sin` et `cos` tels quels.

II - Vocabulaire de la "programmation orientée objet"

La POO n'est pas au programme, mais *tout est objet* dans Python, il est donc difficile d'éliminer complètement le vocabulaire et les notations associés aux objets.
Voici le minimum à savoir pour pouvoir lire les documents un peu hors programme mais pas trop :

- une *classe* est une catégorie abstraite d'objets (un peu comme un *type* pour les données)
- un *attribut* est une valeur contenue dans un objet et accessible par la syntaxe `nom_objet.nom_attribut`, par exemple `z.real` pour la partie réelle d'un complexe
- une *méthode* est une fonction s'appliquant à un objet, par la syntaxe `nom_objet.nom_méthode(params)`, par exemple `ligne.split(sep=';')` pour scinder une chaîne de caractères
- une *instance* d'une classe est un exemplaire concret d'un objet de cette classe, avec un état (cf. les attributs) et un comportement (cf. les méthodes) définis par la classe.

III - Types prédéfinis en Python

1) Types simples

a) Booléens

Type `bool` : deux valeurs, `True` et `False`, souvent obtenues comme résultat de l'évaluation d'une *expression booléenne*, comme `n==0` ou `abs(x)<=e`.

Attention ! Ne pas confondre le *test* `n==0` avec l'*affectation* `n=0`.
Le connecteur "*différent de*" s'écrit `!=` (et non pas <> comme dans certains langages...)

Remarque importante : Python met en œuvre l'*évaluation paresseuse* des expressions booléennes, c'est-à-dire que l'évaluation (de gauche à droite) est interrompue dès que le résultat est acquis. Par exemple pour `P and Q`, si l'évaluation de `P` donne `False`, `Q` ne sera pas évaluée. Cela peut être pratique, typiquement lors du parcours d'un tableau indexé de `0` à `n-1` : `k<n and t[k]!=x` ne déclenchera pas d'erreur même si `k` atteint la valeur `n` (contrairement à `t[k]!=x and k<n`).

b) Entiers

Type `int` : Python sait calculer sur les entiers relatifs arbitrairement grands (essayer `factorial(200)`), mais si l'on en abuse les opérations algébriques élémentaires ne peuvent plus être considérées comme s'exécutant en temps constant...

Si `a` et `b` sont entiers (`b` non nul), `a//b` et `a%b` donnent respectivement le quotient et le reste de la division euclidienne de `a` par `b` (tandis que `a/b` donne un flottant, même si le quotient est exact).

c) Flottants

Type `float` : Python utilise aujourd'hui sur toutes les plates-formes les flottants *double précision* définis par la norme IEEE 754, stockés sur 64 bits (8 octets). 1 bit pour le signe, 11 bits pour l'exposant de 2 et 52 bits pour la partie "après la virgule" de la mantisse (normalisée en binaire sous la forme 1,... ; le 1 avant la virgule est donc un 53^{e} bit, dit *implicite*). Comme $\log_{10} 2^{53} \approx 16$, la précision dans le système décimal est d'environ 16 chiffres significatifs.

Le module `mpmath` (livré avec `sympy`) permet d'augmenter cette précision si besoin (`mp` comme *multi-précision*).
Par exemple, à la suite des instructions
```
from mpmath import mp
mp.dps=50
```
, les calculs seront faits avec 50 chiffres significatifs.

NB : Python inclut dans le type `float` les valeurs $\pm\infty$, qui s'affichent `inf` ou `-inf` : après l'instruction `x=float('Infinity')`, `x` contient la valeur `inf` (`x=inf` renvoie une erreur, mais `x=numpy.inf` est accepté, après chargement du module `numpy` !). Ces valeurs se comportent "naturellement", avec une curiosité pour les "formes indéterminées" telles que `0*x` et `x-x`, qui renvoient `nan` (objet particulier de type `float` pourtant baptisé "***n**ot **a** **n**umber*"...).

Attention ! Le fait que Python calcule en base 2 peut occasionner quelques surprises : par exemple `0.1 + 0.2 - 0.3` renvoie `5.551115123125783e-17`...

d) Complexes

Type `complex` : le complexe $x + \mathbf{i}y$ (où x et y sont réels) est représenté en Python par `complex(x,y)` ou `x+1j*y` (en effet `1j` représente $\mathbf{i}$, `2j` représente $2\mathbf{i}$, etc., mais `j` n'est pas compris, ni `y*j`...).
Si `z` est un *objet* du type `complex`, ses *attributs* `real` et `imag`, accessibles par `z.real` et `z.imag`, sont respectivement sa partie réelle et sa partie imaginaire. Python sait faire les opérations élémentaires sur les complexes, mais pour des opérations avancées, cf. le module `cmath`.

2) Types structurés

a) Les "listes" Python

Type `list` : Python gère dynamiquement les familles **ordonnées** d'objets placés entre crochets et séparés par des virgules. Les éléments d'une liste ne sont pas nécessairement distincts (contrairement aux éléments d'un *ensemble*) ; ils ne sont pas nécessairement du même type (contrairement aux éléments des *tableaux* du module `numpy` et de la plupart des langages). Dans toute la documentation Python, ces familles sont appelées *listes*, malgré le risque de confusion avec le *type abstrait de données* appelé *liste (chaînée)* dans toute la littérature informatique. Or les listes Python permettent un accès direct à chaque élément (*accès aléatoire*), contrairement aux listes chaînées (à *accès séquentiel*). En ce sens, les listes Python s'apparentent aux *tableaux* des autres langages et nous les appellerons parfois ainsi.

Indexation et "slicing" : si L est une liste, $n =$`len`(L) renvoie *la longueur* (*i.e.* le nombre d'éléments) de L ; la *liste vide* `[]` a pour longueur 0. Les éléments sont indexés de 0 à $n-1$: si $0 \leq k < n$, $L[k]$ renvoie le $(k+1)$-ième élément (une erreur survient si $k \geq n$, il n'y a pas de périodicité). Et pour $0 < k \leq n$, $L[-k]$ renvoie le k-ième élément **à partir de la fin**, soit $L[n-k]$ (une erreur survient si $k > n$). Par exemple $L[-1]$ renvoie le dernier élément, cela sans avoir à calculer la longueur.

Le "*slicing*" permet de définir une *tranche* d'une liste. Il existe deux syntaxes.

- $L[debut : fin]$: liste des $L[k]$ pour $debut \leq k < fin$ (intervalle **ouvert à droite**). Si *debut* est omis, la tranche commence au début de la liste ; si fin est omis, la tranche se termine à la fin de la liste. Par exemple $L[:]$ renvoie la liste entière, $L[1:]$ renvoie L privée de son premier élément (ou bien une erreur si L est vide). Noter que *debut* et fin peuvent être négatifs : $L[:-1]$ renvoie L privée de son dernier élément (ou bien une erreur si L est vide).
- $L[debut : fin : pas]$: liste des $L[debut + i * pas]$ pour $i = 0, 1 \ldots$ tant que fin n'est pas atteint ou dépassé (*pas* peut être négatif).

Modification d'éléments d'une liste : les listes Python sont *mutables*, c'est-à-dire que l'on peut modifier un élément par une simple affectation `L[k]=v`. On peut aussi modifier une tranche (en lui affectant une liste, pas nécessairement de la même longueur).

Attention ! Si une tranche est située du côté **gauche** d'une affectation, les valeurs correspondantes de la liste seront **modifiées**. Si elle est située du côté **droit**, les valeurs correspondantes de la liste seront **copiées** dans la liste du côté gauche.

Ajout, suppression d'un élément : `L.append(v)` ajoute la **valeur** `v` à la fin de la liste `L` (`append` est une *méthode* des objets de type `list`). Noter que cette instruction renvoie le résultat `None` (donc en fait ne renvoie rien, son exécution modifie la liste `L`). En revanche, `L.pop()` renvoie le dernier élément de `L` **et le supprime de** `L`. Ces deux instructions sont essentielles pour une utilisation efficace de la *gestion dynamique de la mémoire*, car elles s'exécutent *en temps constant* par rapport à la longueur de `L` (plus précisément en *temps constant amorti* mais cela dépasse le cadre du programme des CPGE). C'est pourquoi on s'en sert pour implémenter le type *pile*.

NB : l'insertion de `v` dans `L` à l'indice `i` (`L.insert(i,v)`) est possible mais sa complexité est moins favorable, de l'ordre de `len(L)-i` car il faut décaler tous les éléments suivants. De même pour `L.pop(i)` qui renvoie et supprime l'élément d'indice `i`.

Concaténation de listes : `L1+L2` renvoie une nouvelle liste constituée des éléments de `L1` suivis de ceux de `L2`. Opération très commode, mais trompeuse car sa complexité est de l'ordre de `len(L1)+len(L2)` (car tous les éléments sont recopiés) ; il faut en tenir compte dans les éventuels calculs de complexité...
`L1.extend(L2)` "ajoute" les éléments de la **liste** `L2` à la fin de `L1`, avec une complexité de l'ordre de `len(L2)`, **mais** `L1` est modifiée.
L'expression `k*L`, où `k` est un entier naturel et `L` une liste, renvoie une liste formée de `k` exemplaires de `L` mis bout à bout. Ainsi `2*[1,2,3]` renvoie `[1,2,3,1,2,3]` (avec `k=0` on obtient la liste vide).

Attention ! On voit bien le risque de confusion avec les calculs vectoriels, qui explique sans doute l'absence de ces opérateurs dans le programme officiel...

Partage des données : dans Python, une variable de type `list` contient en fait l'adresse de la zone mémoire où sont stockées les valeurs (*pointeur*) ; cela peut induire des surprises, voire des erreurs. Par exemple, après les instructions `L=[1,2];M=L;L[1]=3`, l'affichage de `M` donne `[1,3]` (car `L` et `M` partagent la même zone mémoire après l'affectation `M=L`).

Copie d'une liste : compte tenu du comportement précédent, il est parfois utile d'effectuer une copie de `L` (les mêmes éléments dans le même ordre, stockés dans une autre zone mémoire !).
Deux possibilités : `M=L.copy()` ou `M=L[:]`.

Remarque technique pour une seconde lecture : `L.copy()` effectue une copie "superficielle" (*shallow copy*), au sens où, si par exemple l'un des éléments de `L` est lui-même une liste, seul le pointeur correspondant sera dupliqué et non les éléments de ladite liste... Pour que toutes les données soient dupliquées, utiliser la fonction `deepcopy` du module `copy`...

Entre nous... quelques autres commandes (hors programme, à ne pas utiliser si l'énoncé demande de les programmer !) : `max(L)`, `min(L)`, `sum(L)` renvoient ce que leur nom laisse présager, `L.sort()` effectue le *tri en place* de `L` (c'est-à-dire que `L` est modifiée de sorte que ses éléments soient triés, dans l'ordre croissant par défaut, voir les options dans l'aide). Tout cela bien sûr à condition que les éléments de `L` soient comparables ou sommables...
`x in L` renvoie `True` ou `False` selon que `x` est présent ou pas dans `L`.
`L.count(x)` compte le nombre d'occurrences de la valeur `x` dans `L`.
`L.index(x)` renvoie l'indice de la première occurrence de `x` dans `L` s'il est présent, une erreur sinon.
`L.remove(x)` supprime de `L` la première occurrence de `x` s'il est présent, renvoie une erreur sinon.
`L.reverse()` remplace `L` par son *image miroir* (mêmes éléments en ordre inverse).

b) Les "*tuples*"

Type `tuple` : ce type présente des analogies avec le type `list`, car il permet aussi de stocker une famille d'objets, mais placés entre **parenthèses** et avec une autre différence fondamentale, car les tuples sont ***non mutables***, leurs éléments ne peuvent pas être modifiés.
En revanche l'accès aux éléments se fait comme pour les listes (indexation et *slicing*).

Ce type ne figure pas explicitement au programme, mais il apparaît notamment dans `numpy`, où `shape(a)` renvoie les dimensions du tableau `a` dans un tuple. Python l'utilise souvent en interne car les opérations sur les tuples sont plus rapides que celles sur les listes.
Noter que les parenthèses sont parfois facultatives : `t=1,2` place dans `t` le tuple `(1,2)` ; les quatre affectations `a,b=1,2`, `a,b=(1,2)`, `(a,b)=1,2`, `(a,b)=(1,2)` placent `1` dans `a` et `2` dans `b`.

c) Les chaînes de caractères

Type `str` : les chaînes de caractères peuvent être considérées comme des "listes de caractères", au sens où l'on accède auxdits caractères par indexation et aux "sous-chaînes" par *slicing*. **Mais** elles sont ***non mutables*** comme les tuples.
De plus, pour une présentation plus naturelle, les caractères sont écrits l'un après l'autre sans virgules séparatrices et les extrémités sont marquées par des apostrophes (*quotes*) simples ou doubles : `s1='scrab'` et `s2="ble"` définissent deux chaînes de caractères. Alors `s1[3]` renvoie `'a'` mais `s1[3]='u'` déclenche une erreur (chaîne non mutable !).

De nombreuses opérations sur les des listes fonctionnent aussi sur les chaînes : concaténation avec `+`, duplication avec `*`, `len`, `max`, `min`, `count`, `index`, etc. **Mais pas** celles qui modifient les listes : `append`, `pop`, `remove`, `reverse`, `sort`, etc. !

La méthode `split` permet de scinder une chaîne "mot par mot", le séparateur par défaut étant l'espace : `'un deux trois'.split()` renvoie la liste `['un','deux,'trois']`.

On peut préciser le séparateur, option fort utile par exemple lors de la lecture ligne par ligne d'un fichier texte contenant des nombres séparés par des virgules ou des points-virgules :
`'un ; deux ; trois'.split(sep=' ; ')` renvoie aussi la liste `['un','deux,'trois']`
(penser à mettre les espaces dans la définition du séparateur... s'il y en a dans la chaîne source !).

Remarque technique : le *backslash* permet d'insérer certains *caractères de contrôle* dans une chaîne (on parle d'*échappement*) ; par exemple `\n` pour un saut de ligne, `\t` pour une tabulation, `\'` pour une simple quote dans une chaîne délimitée par de simples quotes (!), `\\` pour un backslash (!!). Noter que lesdits caractères de contrôle ne sont interprétés que lors de l'affichage de la chaîne par `print`. On peut demander à Python de ne pas interpréter ces caractères spéciaux, en faisant précéder la chaîne d'un `r` (pour *raw string*, chaîne brute) ; cela est utile notamment pour transmettre un chemin d'accès à un fichier sous Windows... Par exemple `r'U:\temp\new'` sera compris sans remplacement du `\t` et du `\n`.

d) Les "*ranges*"

Type `range` : ce type correspond à des *itérables* prenant très peu de place en mémoire et permettant d'engendrer l'une après l'autre des valeurs entières. Trois syntaxes disponibles :

- `range(`n`)` engendre les entiers consécutifs de `0` à `n-1`
- `range(`$debut, fin$`)` engendre les entiers consécutifs de $debut$ à $fin - 1$
- `range(`$debut, fin, pas$`)` engendre les entiers $debut + i * pas$ pour $i = 0, 1, \ldots$ tant que fin n'est pas atteint ou dépassé (pas peut être négatif)

Cela est notamment utilisé pour définir les valeurs du compteur d'une boucle `for` sans gâcher l'espace mémoire : `for k in range(n)` et `for k in list[range(n)]` donneront à `k` les mêmes valeurs, **mais** la seconde version créera vraiment en mémoire la liste de `n` valeurs alors que la première ne mémorisera que le code permettant d'engendrer les valeurs ! À rapprocher du `arange` de `numpy`.

3) Transtypage

On a parfois besoin de *convertir* un objet d'un type vers un autre type. Par exemple obtenir la chaîne de caractère représentant un nombre ou inversement, obtenir la liste des valeurs générées par un `range`, etc. Voici quelques exemples :

- `str(123)` renvoie la chaîne `'123'`, `int('45')` renvoie l'entier `45`
- `str(pi)` renvoie la chaîne `'3.141592653589793'` (si `pi` du module `math` a été chargé !)
- `float(123)` renvoie `123.0`, `float('1e-3')` renvoie `0.001`
- `int(-2.718)` renvoie `-2` (simple troncature, cf. `floor` et `round` du module `math`)
- `list(range(5))` renvoie `[0,1,2,3,4]`
- `list('abcd')` renvoie `['a','b','c','d']`

IV - Entrées – sorties

1) Clavier – écran

En mode "conversationnel" dans un *shell*, on peut définir la valeur d'une variable *globale* par une simple affectation et afficher le résultat de l'évaluation d'une expression en la tapant puis en la validant par la touche "Entrée".

Durant l'exécution d'un script, pour afficher une valeur dans le *shell* il faut utiliser `print(expression)`. Une autre sortie courante à l'écran est l'affichage d'une image dans une fenêtre créée par `pyplot`. Elle peut être copiée ou enregistrée pour une utilisation ultérieure.

Instruction `input` : il est possible dans un script de demander à l'utilisateur de saisir une valeur, par l'intermédiaire de `s=input(message)`, qui interrompt l'exécution du script, affiche la chaîne `message` et place dans `s` **la chaîne de caractères** tapée par l'utilisateur (utiliser le transtypage pour récupérer un nombre !). Cette pratique doit être réservée à des cas très particuliers, il vaut mieux en général utiliser les paramètres des fonctions pour transmettre des valeurs.

2) Utilisation de fichiers

Python permet de lire et d'écrire des fichiers sur un support de stockage.

a) Utilisation d'un fichier texte

L'instruction `f=open(chemin,mode)` crée un "objet fichier" `f` permettant d'accéder au fichier "physique" dont le chemin d'accès est donné dans la chaîne `chemin`. Le mode d'accès est précisé par la chaîne `mode`, les options les plus courantes étant `'r'` (lecture, valeur par défaut), `'w'` (écriture, si le fichier existe déjà il sera écrasé), `'a'` (ajout, si le fichier existe les nouvelles lignes seront écrites à la fin).

Attention ! À la fin de l'utilisation du fichier, **il faut** le "refermer" par `f.close()`. Opération nécessaire car c'est elle qui vide la mémoire tampon et crée le fichier physiquement sur le support de stockage.

À l'ouverture de `f`, un pointeur est créé vers le début de la première ligne du fichier. `f.read()` lit l'intégralité du fichier et la renvoie sous forme d'une unique chaîne de caractères (les différentes lignes étant séparées par des `'\n'`). On utilise plus souvent `f.readline()`, qui lit une ligne dans le fichier et déplace le pointeur vers la ligne suivante. Chaque ligne est terminée par un `'\n'`, sauf lorsqu'on est arrivé à la fin du fichier (alors `f.readline()` renvoie une chaîne vide).

En pratique, on utilise principalement la boucle `for ligne in f`, où `ligne` prend pour valeurs successives les lignes du fichier. Penser dans ce cas à la méthode `split` pour traiter chaque ligne (cf. **III-2)c)**).

NB : le transtypage `list(f)` renvoie la liste des lignes du fichier, comme l'instruction `f.readlines()`, **à ne pas confondre avec** `f.readline()` !

`f.write(chaine)` écrit dans le fichier les caractères composant `chaine` et renvoie le nombre de caractères écrits (le saut de ligne `'\n'` étant compté comme un seul caractère...).

b) Cas des tableaux `numpy`

Lorsque toutes les lignes d'un fichier texte contiennent un même nombre de valeurs numériques séparées par des points-virgules (fichier `.csv`, éventuellement créé lors d'une acquisition de résultats de mesures), la commande `tableau=numpy.loadtxt(chemin,delimiter=';')` crée un tableau `numpy` bidimensionnel contenant lesdites valeurs.

Attention ! Bien vérifier le format des données... Python ne comprend que le point comme séparateur décimal dans les flottants, alors que les logiciels "français" utilisent la virgule (et le point-virgule pour séparer les valeurs).

S'il n'y a pas eu d'erreur, `numpy.shape(tableau)` renvoie le tuple des dimensions du tableau créé.

NB : si besoin, `numpy.savetxt(chemin,tableau,delimiter=';')` permet d'enregistrer dans un fichier les valeurs contenues dans `tableau`.

c) Cas des images

Voir les commandes `imread`, `imsave` du module `matplotlib.image` pour l'utilisation de fichiers `.png` et `imshow` du module `matplotlib.pyplot` pour l'affichage à l'écran.

Le module `Pillow` offre des fonctionnalités avancées pour le traitement des images.

V - Programmation

Les instructions décrites dans cette section peuvent être utilisées dans un *shell*, mais servent le plus souvent pour exécuter un programme via un script.

1) Notion de bloc – indentation

De nombreuses instructions "encapsulent" d'autres instructions, notamment les tests, les boucles, les définitions de fonctions, etc. Dans ce cas, l'instruction "mère" est écrite sur une ligne terminée par deux points (`':'`) et les instructions encapsulées sont regroupées dans un *bloc*, formé des lignes suivantes, écrites avec un niveau d'indentation supplémentaire. La fin du bloc est marquée par le retour au niveau d'indentation de l'instruction "mère". On peut ainsi *imbriquer* plusieurs niveaux de blocs.

Noter que les éditeurs de texte "labellisés" Python (comme IDLE ou Pyzo) appliquent automatiquement l'indentation marquant le début du bloc, mais il faut signifier manuellement la fin du bloc (la touche "*backspace*" ramène le curseur au niveau d'indentation inférieur).

Noter également que la création d'un bloc n'est pas nécessaire, notamment lorsqu'on n'a qu'une instruction à encapsuler : elle peut alors être écrite après les deux points, **sur la même ligne**. Ainsi les deux codes suivants sont équivalents :

```
if n%2==0:return 'pair'
else:return 'impair'
```

```
if n%2==0:
    return 'pair'
else:
    return 'impair'
```

Les blocs prennent un peu de place mais améliorent la lisibilité. Leur usage est vivement recommandé.

2) Définition d'une fonction – visibilité des variables

On définit la fonction `f` par l'instruction `def f(x,y...):` (*l'en-tête* de la fonction), suivie du bloc formé des instructions à exécuter lors de l'appel de ladite fonction (*le corps* de la fonction). Les symboles `x`, `y`... sont les *paramètres formels* éventuels. Un appel à la fonction ainsi créée seprésente sour la forme `f(ex,ey,...)` où `ex`, `ey` sont les *paramètres effectifs*, des expressions qui sont évaluées au moment de l'appel ; le résultat de l'évaluation de `ex` est placé dans la variable `x`, variable *locale à la fonction* `f` (c'est-à-dire que `x` n'existera que durant cet appel). De même pour `ey`...

Attention ! Si `ex` est réduite au *nom* d'une liste, la "liste" *locale* `x` créée au moment de l'appel partagera ses données avec la liste *globale* `ex`, d'où un risque de modification intempestive de `ex` (on parle d'*effet de bord*). Si l'on veut que `x` soit une copie de `ex`, il faut la créer soi-même, par exemple en transmettant comme paramètre effectif `ex.copy()`.

Par défaut, une variable `v` apparaissant dans le corps de la fonction `f` dans une affectation `v=expr` est créée en tant que variable *locale à* `f` (initialisée avec pour valeur le résultat de l'évaluation de `expr` et détruite à la fin de l'appel en cours). Dans ce cas, si une variable de même nom `v` existe au niveau *global* de la session, elle ne sera pas utilisable, ni affectée par les modifications appliquées à son homonyme.

Si l'on souhaite au contraire modifier le contenu des variables *globales* `v`, `w`,... au sein de la fonction `f`, il faut le déclarer au début du bloc définissant `f`, par la clause `global v,w,...`

`nonlocal` vs `global` : dans le cas d'une fonction `g` définie à l'intérieur du bloc définissant la fonction `f`, pour modifier dans `g` des variables locales à `f` (donc au niveau d'indentation immédiatement supérieur), il faut les déclarer `nonlocal` et non pas `global`.

3) Fonctions *anonymes* (`lambda` vs `def`)

L'instruction `f=lambda x:expression` est équivalente à `def f(x):return expression`.

L'intérêt d'une "fonction lambda" est que l'on peut l'utiliser sans lui donner de nom. Cela est particulièrement utile pour transmettre une fonction comme **paramètre** à une autre fonction.

Par exemple, supposons que `I(f,a,b)` renvoie une valeur approchée de l'intégrale de la **fonction** f sur $[a,b]$ et que l'on a défini une suite de fonctions (f_n) par $F:(n,x)\mapsto f_n(x)$.
Alors F ne peut être transmise comme paramètre à I... mais on peut appeler `I(lambda x:F(1,x),a,b)` et même construire la liste `[I(lambda x:F(n,x),a,b) for n in range(10)]` (cf. **V-8**)).

Noter que l'on peut aussi définir la fonction $n\mapsto f_n$ par `def f(n):return lambda x:F(n,x)`. Alors `f(1)` renvoie la **fonction** $x\mapsto f_1(x)$, cela peut être utile aussi...

NB : la construction `lambda` fonctionne également avec plusieurs variables, par exemple `lambda x,y:sqrt(x**2+y**2)`.

Noter enfin que c'est l'existence de cette syntaxe qui empêche d'utiliser `lambda` comme nom de variable !

4) Résultat renvoyé par une fonction

Le plus souvent, une fonction fournit un résultat, qui peut être n'importe quel objet Python.

L'instruction `return r`, rencontrée n'importe où dans le corps d'une fonction, interrompt l'exécution de l'appel de ladite fonction et *renvoie* le résultat de l'évaluation de `r` (qui peut lui-même être une expression). C'est-à-dire qu'un appel de fonction peut être utilisé comme une expression, dont l'évaluation donnera le résultat renvoyé par la fonction.

Attention ! L'instruction `return X,Y` — commode pour renvoyer deux choses à la fois — renvoie malgré les apparences le *tuple* `(X,Y)`, ce qui peut surprendre, notamment lorsque `X` et `Y` doivent servir à leur tour comme paramètres dans un appel de fonction ; par exemple, `plt.plot(X,Y)` est très différent de `plt.plot((X,Y))`.

Cela dit, certaines fonctions exécutent une tâche sans avoir à *renvoyer* de résultat (dans certains langages, on parle de *procédures*). L'effet produit peut être la modification d'une variable, par exemple `L.append(v)` (étant entendu qu'une méthode d'un objet s'apparente à une fonction...), la création d'un fichier, l'affichage de valeurs via `print` ou l'affichage d'une image...
Si aucun `return` n'est rencontré durant un appel de la fonction, cette dernière renvoie tout de même le "résultat" `None`, valeur spéciale de Python signifiant "aucun résultat", comme son nom l'indique !

5) Instructions conditionnelles

On dispose de quatre syntaxes permettant de soumettre l'exécution d'un bloc à une condition :

```
if expr_bool:
    bloc_si_vrai
```
`expr_bool` est une expression booléenne,
`bloc_si_vrai` est exécuté si son évaluation renvoie `True`

```
if expr_bool:
    bloc_si_vrai
else:
    bloc_si_faux
```
`expr_bool` est une expression booléenne,
`bloc_si_vrai` est exécuté si son évaluation renvoie `True`
`bloc_si_faux` est exécuté si son évaluation renvoie `False`

```
if expr_bool_1:
    bloc_1
elif expr_bool_2:
    bloc_2
elif ...
```
`expr_bool_1` est une expression booléenne,
`bloc_1` est exécuté si son évaluation renvoie `True` ;
`expr_bool_2` est une autre expression booléenne,
`bloc_2` est exécuté si son évaluation renvoie `True` ;
on peut multiplier les clauses `elif`...

```
if expr_bool_1:
    bloc_1
elif expr_bool_2:
    bloc_2
elif ...
else:
    bloc_sinon
```
`expr_bool_1` est une expression booléenne,
`bloc_1` est exécuté si son évaluation renvoie `True` ;
`expr_bool_2` est une autre expression booléenne,
`bloc_2` est exécuté si son évaluation renvoie `True` ;
on peut multiplier les clauses `elif`...
`bloc_sinon` est exécuté si tous les tests ont donné `False` ;

Autrement dit, cette dernière syntaxe est la version générale, les clauses `elif` et `else` étant facultatives.

Si l'un des blocs sous condition est exécuté, on se branche aussitôt **après le dernier bloc de l'instruction conditionnelle.** Il faut donc faire attention à l'ordre des tests, qui sont évalués de haut en bas !

Comme son nom l'indique, `elif` peut être remplacé par `else:if` ; on peut en effet imbriquer plusieurs instructions conditionnelles, mais cela oblige à augmenter le niveau d'indentation.

6) Boucle `for`

```
for i in iterable:
    bloc
```
est la syntaxe générale en Python de la *boucle inconditionnelle* (c'est-à-dire que toutes les valeurs de `i` proposées par `iterable` seront utilisées, à moins d'une sortie de la boucle par un `return` ou un `break`).

Python manipule toutes sortes d'*itérables*. Ceux que l'on utilise en CPGE sont les types structurés présentés au **II–2)**.

Le cas le plus fréquent est le `range`, où `i` prend successivement les valeurs entières engendrées.
Attention aux intervalles **ouverts à droite** !
`range(n)` engendre $[\![0, n-1]\!]$, `range(1,n+1)` engendre $[\![1, n]\!]$...

Mais on utilise couramment pour `iterable` une liste, une chaîne de caractères, voire un tuple... Dans ces cas-là, `i` prend successivement les valeurs contenues dans `iterable` (dans l'ordre de leurs indices). Ainsi, si l'on n'a pas besoin de l'**indice** correspondant à une **valeur**, `for v in L:` suivi de manipulations de `v` sera plus élégant que `for k in range(len(L)):` suivi de manipulations de `L[k]`. Voir par exemple la recherche d'une valeur dans `L`.

Attention ! Ne pas confondre l'expression booléenne `v in L` et l'en-tête de boucle `for v in L:` !!

7) Boucle `while`

```
while expr_bool:
    bloc
```
est la syntaxe générale en Python de la *boucle conditionnelle* (c'est-à-dire que le bloc est exécuté *tant que* l'évaluation de `expr_bool` renvoie `True` ; dès qu'elle renvoie `False`, l'exécution se poursuit à la suite de `bloc`).

La boucle `while` permet un contrôle fin, mais le prix à payer est l'élaboration précise du test, la justification rigoureuse de la terminaison de la boucle et la gestion par le programmeur de l'évolution des indices lorsqu'il y en a au moins un. D'où l'intérêt de la boucle `for` (qui pourtant peut toujours être remplacée par une boucle `while` !).

NB : deux écoles (de programmeurs) s'affrontent au sujet du bien fondé de la sortie prématurée d'une boucle, par un `return` (dans le corps d'une fonction, lorsqu'on a trouvé le résultat à renvoyer) ou par un `break` (lorsqu'on constate qu'il n'y a plus lieu de poursuivre l'exécution de la boucle). Pour être pragmatique, dans le cadre de la programmation en CPGE, on peut se permettre ces instructions lorsqu'elles s'intègrent dans un code **concis et clair**... En revanche une boucle `while True:` suivie d'un bloc truffé de tests et de `return` et de `break` sera mal perçue.

Par exemple, pour la recherche (linéaire) d'une valeur `x` dans une liste `L` (non triée) on pourra écrire :

```
def est_present(x,L):
    for v in L:
        if v==x: return True
    return False
```

plutôt que la version "classique" :

```
def est_present(x,L):
    ok=False;k=0
    while not ok and k<len(L):
        if L[k]==x: ok=True
        k+=1
    return ok
```

8) Construction d'une liste "par compréhension"

Cette possibilité offerte par Python n'est pas explicitement au programme, mais elle permet la construction élégante d'une liste à l'aide d'une "pseudo" boucle `for`, éventuellement suivie d'un test (*filtrage*). Par exemple (cf. le sujet de Centrale 2015 !), une fonction effectuant la multiplication terme à terme des éléments d'une liste `L` par un scalaire `k` s'obtient par :

```
def lmul(t,L):
    return [t*x for x in L]
```

Et `[n for n in range(101) if test(n)]` construit la liste des n de $[\![0, 100]\!]$ pour lesquels la fonction booléenne `test` renvoie `True`.

VI - Commandes usuelles des modules usuels

1) `numpy` (supposé chargé par `import numpy as np`)

On crée un tableau T (à une dimension) plein de n zéros (de type `float` par défaut) par `T=np.zeros(n)`. Comme les listes Python, le tableau est indexé de 0 à $n-1$.
On peut ensuite modifier les valeurs stockées par des affectations telles que `T[k]=expr`.

Les tableaux `numpy` supportent le *slicing* (cf. **II-2)a)**) dans toutes les dimensions. Mais **attention** ! Une tranche ainsi obtenue n'est qu'une *vue* sur une partie du tableau. Une vue est "recalculée" à chaque fois que l'expression est rencontrée. Cela peut se traduire par des comportements surprenants, notamment si l'on essaie d'échanger deux tranches. Par exemple, si `M` est une matrice `numpy`, l'instruction `M[0],M[1]=M[1],M[0]` ne va pas échanger les deux premières lignes, comme on pourrait le penser ! La première affectation `M[0]=M[1]` va bien placer la 2e ligne dans la 1re, **mais** lors de la seconde affectation `M[1]=M[0]`, la vue `M[0]` va être recalculée et la 2e ligne sera finalement inchangée ! Cela peut-être évité en forçant la création d'une *copie* de la 1re ligne : `M[0],M[1]=M[1],M[0].copy()`.
En revanche, si `M` est une liste de listes Python, la même instruction va bien échanger les deux premières listes, car ce sont les adresses en mémoire desdites listes qui seront échangées *(pointeurs)*.

Utiles pour de nombreux calculs numériques, il existe deux commandes créant un tableau rempli avec des valeurs *équiréparties* :

- `np.arange(start,stop,step)` fonctionne de façon similaire à `range` mais renvoie un tableau `numpy`
- `np.linspace(start,stop,nb)` est similaire, mais **calcule le pas** en fonction du nombre `nb` de points souhaités. On peut récupérer le pas `dt` calculé par Python grâce à l'option `retstep` : en effet `np.linspace(start,stop,nb,retstep=True)` renvoie le couple `(T,dt)`.

Pour créer un tableau à plusieurs dimensions, c'est le *tuple* des dimensions (*shape*) qu'il faut fournir en paramètre (attention aux parenthèses !) :

- `np.zeros((3,3))` renvoie la matrice nulle, carrée d'ordre 3
- `np.ones((4,5))` renvoie la matrice de forme $(4,5)$ dont tous les coefficients valent 1
- `np.eye((6,6))` renvoie la matrice identité d'ordre 6.

On peut remplir "à la main" un tableau en convertissant une liste `L` par le *transtypage* `np.array(L)` ; on obtient un vecteur de $\mathbb{K}^n$ si `L` est une liste de scalaires et une matrice si `L` est une liste de listes de mêmes longueurs.
Enfin `np.diag(L)` renvoie la matrice diagonale avec les valeurs contenues dans la liste `L` sur la diagonale.

Type des éléments d'un tableau

Contrairement aux listes Python, les tableaux `numpy` doivent avoir des éléments tous du même type, donné à la création du tableau. Le type par défaut est en général `float`, mais on peut créer un tableau d'entiers en précisant `dtype=int`, un tableau de chaînes de caractères avec `dtype=str`, etc.

Ainsi `np.zeros((3,3))` renvoie une matrice remplie de flottants `0.`, `np.zeros((3,3),dtype=int)` une matrice remplie d'entiers `0` et `np.zeros((3,3),dtype=str)` une matrice remplie de chaînes vides `''`.

Forme d'un tableau : si `A` est un tableau `numpy`, l'appel de fonction `np.shape(A)` renvoie le *tuple* de ses dimensions (c'est aussi l'**attribut** `A.shape`).

`len(A)` renvoie **le nombre d'éléments** pour un tableau à une dimension, **le nombre de lignes** pour un tableau à deux dimensions.

Application d'une fonction à tous les éléments d'un tableau

Objectif essentiel pour tracer des graphes avec `plot`, puisqu'il faut fournir le tableau des abscisses et le tableau des ordonnées...
Deux solutions classiques (la première est conseillée car elle est plus efficace) :

- `numpy` fournit des versions *vectorisées* des fonctions usuelles : `np.exp(X)` renvoie le tableau rempli par les $\exp(x)$ pour x dans X, où le tableau X contient les abscisses x_k (cela ne fonctionne pas avec les listes Python...). On peut aussi utiliser la *vectorisation* d'une fonction : `np.vectorize(f)(t)` applique `f` à tous les éléments du tableau `t`.
- on peut construire *en compréhension* la liste `[f(x) for x in X]` et la convertir en tableau `numpy`.

Calculs vectoriels et matriciels

Contrairement aux liste, les tableaux `numpy` se prêtent aux *combinaisons linéaires*, avec les opérateurs `*` pour la multiplication d'un scalaire par un tableau et `+` pour l'addition de deux tableaux.

Mais **attention !** La multiplication par `*` de deux tableaux de même taille donne bien un résultat, mais effectue le produit **élément par élément**, il vaut donc mieux connaître les opérations suivantes !

- *Produit matriciel* : la *méthode* `dot` permet d'effectuer un produit matriciel ; `A.dot(B)` renvoie le produit AB si les matrices A et B sont de tailles convenables et stockées dans `A` et `B`. Noter que `np.dot(A,B)` renvoie le même résultat mais est moins commode lorsqu'il faut enchaîner plusieurs multiplications.
- *Produit scalaire* : `np.vdot(u,v)` renvoie le produit scalaire des "vecteurs" u, v (qui peuvent être fournis sous forme de simple liste ou de tuple). Noter que `np.dot(u,v)` renvoie $\sum u_k v_k$, donc convient aussi **dans le cas réel**, mais dans le cas complexe (hors programme en maths), seul `vdot` renvoie le résultat correct $\sum \overline{u_k} v_k$.
- *Produit vectoriel* : `np.cross(u,v)` renvoie le produit vectoriel des "vecteurs" u, v en dimension 3 (et le *produit mixte* en dimension 2).
- *Transposition* : `np.transpose(A)` et `A.T` renvoient la transposée de `A`.
- *Trace* : `np.trace(A)` renvoie la trace de `A`.

Les fonctions et méthodes précédentes sont disponibles dans le module `numpy`, mais il faut aller chercher dans le sous-module `numpy.linalg` les fonctions d'algèbre linéaire qui nécessitent des calculs plus élaborés (souvent basés sur l'algorithme du pivot de Gauss et donc soumis aux imprécisions de calcul sur les flottants...).

2) `linalg` (supposé chargé par `import numpy.linalg as alg`)

Si le tableau `numpy` `A` contient une matrice, `alg.matrix_rank(A)` renvoie le *rang* de `A`.

Pour toute la suite, `A` est une matrice carrée, `alg.det(A)` renvoie le *déterminant* de `A` et `alg.inv(A)` son inverse (sous réserve d'existence !) ; `alg.matrix_power(A,k)` calcule (efficacement !) A^k.

Résolution d'un système de Cramer

Si `A` est inversible d'ordre n et `B` un "vecteur", `alg.solve(A,B)` renvoie la solution (numérique approchée...) du système linéaire $AX = B$.
`B` peut être un tableau `numpy` de forme $(n,)$ ou $(n,1)$, ou bien une liste ou un tuple de taille n.
Le résultat est donné sous forme de liste.

Polynôme caratéristique

Si `A` est une matrice carrée, `np.poly(A)` renvoie un vecteur contenant les coefficients du polynôme caractéristique de `A`, par ordre de degré décroissant (et c'est bien la version **unitaire** du programme officiel, $\det(x.I - A)$).

NB : curieusement, si L est une liste de scalaires, `np.poly(L)` renvoie les coefficients du polynôme $\prod_{\lambda \in L} (X - \lambda)$. Attention donc à la syntaxe !

Éléments propres

Si `A` est carrée d'ordre n, `alg.eigvals(A)` renvoie une liste de n scalaires contenant les valeurs propres de `A` (ou plutôt des approximations numériques des valeurs propres de `A`), éventuellement complexes et répétées selon leur multiplicité.

`alg.eig(A)` renvoie un couple (L, P), où L est la liste des valeurs propres comme ci-dessus et P une matrice de même taille que A telle que $P^{-1}AP$ soit diagonale, à condition bien sûr que A soit diagonalisable ! Le tout sous forme numérique approchée.

Attention ! Si A n'est pas diagonalisable, Python ne cherche pas à trigonaliser, il renvoie une matrice P de déterminant (presque) nul, dont les vecteurs colonnes sont bien (approximativement) des vecteurs propres de A, mais ne forment pas une famille libre.

3) `pyplot` (supposé chargé par `import matplotlib.pyplot as plt`)

La commande de base est `plt.plot(X,Y)` qui trace la "ligne brisée" reliant les points de coordonnées $(X[k], Y[k])$, X et Y étant deux tableaux de même taille (à une dimension). Plusieurs commandes de cette forme permettent de mémoriser plusieurs "courbes" (qui sont en fait des lignes brisées !).

La commande `plt.show()` affiche le graphique à l'écran. Par défaut, la *fenêtre d'affichage* (les intervalles pour les valeurs des abscisses et des ordonnées) est déterminée automatiquement, de même que les unités et graduations sur les axes.
Un zoom interactif est possible dans la fenêtre graphique, ainsi que des translations et changements d'échelles.

Attention ! `plt.show()` bloque l'exécution du script, les instructions qui la suivent ne seront exécutées qu'après fermeture de la fenêtre graphique.

`plt.grid()` ajoute des lignes en pointillés pour chaque graduation des axes.
`plt.title('Titre')` définit le titre de la figure.
`plt.xlabel('Valeurs de $t$')` définit l'étiquette de l'axe des abscisses. De même avec `plt.ylabel` pour l'axe des ordonnées. Les `'$'` permettent de formater des formules comme en LaTeX.
`plt.xlim(a,b)` impose l'intervalle $[a, b]$ pour les valeurs des abscisses. De même `plt.ylim`.
`plt.axis([x_min,x_max,y_min,y_max])` définit la fenêtre d'un seul coup.
`plt.axis('equal')` impose la même unité sur les deux axes.

Graphiques multiples : il peut être intéressant d'afficher plusieurs images dans une même fenêtre. Pour cela, utiliser la commande `plt.subplot(nl,nc,n)`, qui prévoit dans la fenêtre à afficher `nl` lignes de `nc` figures, lesdites figures étant numérotées de gauche à droite et de haut en bas par l'entier `n`, qui peut donc prendre les valeurs de 1 à `nl*nc`. En pratique, pour préparer une fenêtre comportant plusieurs figures, on utilise `subplot` avec les mêmes valeurs de `nl` et `nc`, en faisant varier `n` : les instructions de tracé suivant l'instruction `plt.subplot(nl,nc,n)` affecteront la figure numéro `n`, jusqu'à l'instruction `subplot` suivante (ou jusqu'au tracé par `plt.show()`).

Sauvegarde :`plt.savefig(nom_de_fichier)` permet d'enregistrer la figure dans un fichier, dont on donne le nom en paramètre, dans une chaîne de caractères.
Divers formats sont disponibles et automatiquement utilisés si le nom du fichier se termine par l'extension correspondante : `.pdf`, `.eps`, `.png`, `.svg`, etc.

Options usuelles pour `plot`

La syntaxe générale `plt.plot(X,Y,options)` permet de personnaliser le tracé.
Voici les options les plus courantes :

- *couleur de la courbe* : `color='black'` donne un tracé en noir ; les autres couleurs disponibles sont : `'blue'`, `'green'`, `'red'`, `'cyan'`, `'magenta'`, `'yellow'`, `'white'`
- *style de trait* : `linestyle='-'` donne un trait continu, `'--'` des tirets, `':'` des pointillés, `'-.'` des tirets et points alternés. Pour ces deux premières options, on peut utiliser des abréviations : `'k:'` donnera des pointillés noirs, etc.
- *épaisseur du tracé* : `linewidth=2` donne un trait plus épais (valeur 1 par défaut) ; comme on s'y attend, plus la valeur est élevé, plus le trait est épais !
- *marques pour les points* : `marker='x'` matérialise la position de chaque point par un `'x'` ; il y a une grande variété de marqueurs disponibles, `o`, `+`, `*`, `.`, `v`, `^`, `<`, `>`, etc. (voir l'aide de `matplotlib` si besoin) ; la taille des marqueurs peut être réglée grâce à l'option `markersize`
- *légende* : `label='Euler'` définit le nom associé à un tracé ; moyennant quoi la commande `plt.legend()` affiche un cartouche avec les différents styles de tracés suivis du nom associé par l'option `label`. On peut choisir la position du cartouche en précisant par exemple `plt.legend(loc=2)` (0 pour laisser Python choisir, 1 pour en haut à droite, 2 pour en haut à gauche, etc.)

4) `scipy` pour le calcul numérique

Le module `scipy` contient plusieurs sous-modules permettant divers calculs approchés.

a) Résolution approchée d'équations

Après le chargement `from scipy.optimize import fsolve`, l'appel `fsolve(f,x0)` renvoie s'il en trouve une solution approchée de l'équation $f(x) = 0$, en prenant comme point de départ `x0`, élément de l'ensemble de définition de la fonction f, qui peut admettre comme argument un scalaire ou un "vecteur".

b) Résolution approchée d'équations différentielles ordinaires (*ode*)

Après le chargement `from scipy.integrate import odeint`, l'appel `odeint(F,y0,T)` renvoie une solution approchée de l'équation différentielle $y' = F(y,t)$ accompagnée de la condition initiale $y(T[0]) = y0$, T étant la liste des points utilisés pour l'approximation, typiquement `T=np.arange(0,1,0.01)`.

Les valeurs dans T doivent être des flottants, mais y et F peuvent être à valeurs scalaires ou vectorielles. Le cas vectoriel sert à résoudre les systèmes différentiels et les équations scalaires d'ordre au moins égal à 2 (cf. le cours de maths).

c) Calcul approché d'intégrales

Après le chargement `from scipy.integrate import quad`, l'appel `quad(f,a,b)` renvoie un couple (I, ε) où I est une valeur approchée de l'intégrale de la fonction f sur $]a,b[$ et ε un majorant de l'erreur commise. a et b peuvent être infinis (utiliser `np.inf`).

Par exemple `quad(lambda x:exp(-x**2),-np.inf,np.inf)` renvoie

$$(1.7724538509055159, 1.4202636781830878e-08)$$

tandis que $\sqrt{\pi} \approx 1.7724538509055159$ (intégrale de Gauss).

`scipy.integrate` contient aussi des fonctions de calcul approché d'intégrales pour une "fonction" donnée par une liste de points. Par exemple `trapz(Y,X)` applique la méthode des trapèzes à la liste de points dont les abscisses sont dans X et les ordonnées dans Y (attention à l'ordre des paramètres !). De même `simps(Y,X)` applique la méthode de Simpson.

La méthode de Romberg est également disponible, mais avec pour condition que Y contienne un nombre de valeurs de la forme 2^p+1 (c'est-à-dire que le nombre d'intervalles de la subdivision est une puissance de 2) et régulièrement espacés. D'ailleurs la syntaxe est un peu différente : c'est `romb(Y,dx)` qui renvoie la valeur approchée de l'intégrale, où `dx` est le *pas* de la subdivision. Rappelons à ce propos que `np.linspace(a,b,n,retstep=True)` renvoie le couple `(X,dx)` où `X` est le résultat "normal" de `linspace` et `dx` le pas de la subdivision calculé par Python.

5) `image` (supposé chargé par `import matplotlib.image as img`)

Pour des manipulations avancées d'images, il existe un module historique, développé à l'origine par le MIT, nommé "`PIL`" pour "Python Imaging Library". Le projet a été repris par une autre équipe sous le nom de "`Pillow`".

Mais pour des traitements simples, on pourra souvent se contenter des fonctions basiques du module `image` de `matplotlib`.

Les commandes suivantes sont utiles :

- `im=img.imread('Lena.png')` place dans `im` un tableau de forme $(n,p,3)$, où n (resp. p) est le nombre de lignes (resp. colonnes) de pixels dans l'image. La troisième dimension fournit pour chaque pixel les trois composantes RVB. Noter que, pour une image en niveaux de gris, les 3 composantes existent bien mais sont égales. On peut donc se ramener à un simple tableau de type (n,p) en "sliçant" : `im0=im[:,:,0]`. Pour remplir un tableau de niveaux de gris à partir d'une image en couleurs, il suffit d'affecter à chaque pixel la moyenne de ses trois composantes couleurs.

  N.B. : nativement, `matplotlib` ne sait lire et convertir que le format PNG ; toutefois, si le module `Pillow` est installé, `imread` pourra ouvrir d'autres types de fichiers. Lena est une mascotte des informaticiens dont le portrait est souvent utilisé pour les tests.

- `img.imsave('Lena.jpg',im)` enregistre l'image au format JPEG, qui est automatiquement sélectionné au vu de l'extension du nom de fichier. Cela suppose que le tableau `im` contient bien les trois composantes de couleurs. Pour enregistrer une image directement à partir d'un simple tableau de niveaux de gris (tel `im0` ci-dessus), deux solutions :
  - on peut copier trois fois les valeurs de niveau de gris en tant que composantes couleurs !
  - on peut aussi indiquer à `matplotlib` que l'on travaille avec la palette ("colormap") des niveaux de gris : `img.imsave('Lena.jpg',im0,cmap=plt.cm.gray)` ; cette option sera aussi utile pour l'affichage... Noter que la palette inversée (pour afficher le "négatif") existe aussi : `plt.cm.gray_r`.

- `plt.imshow(im)` prépare l'affichage par `pyplot` de l'image matricielle dont les composantes couleurs sont stockées dans le tableau `im`. Par défaut `imshow` attend un tableau tridimensionnel avec trois composantes couleurs par pixel. En cas de simple tableau de niveaux de gris, ajouter l'option "colormap" `plt.cm.gray`.

  L'affichage effectif (qui bloque l'exécution du script...) est obtenu par `plt.show()`.

Remarques importantes

Parfois, la lecture d'un fichier image remplit les composantes couleurs avec des entiers de $[\![0,255]\!]$; mais certaines commandes de `matplotlib` (enregistrement et affichage) requièrent des flottants de $[0,1]$.
On n'hésitera pas, si besoin, à utiliser les fonctions de conversion $n \mapsto n/255$ ou $x \mapsto int\,(255 * x)$.

Dans `matplotlib`, il semble que la palette des niveaux de gris accepte n'importe quelles valeurs et s'étale du min au max. Par contre, l'affichage en couleur requiert des flottants de $[0,1]$.
Dans certains cas (par exemple la détection des contours) on est amené à remplir un tableau `t` avec des nombres dont on ne connaît pas à l'avance les valeurs extrêmes ; on pensera alors à utiliser `M=np.max(t)` et `m=np.min(t)` pour tout ramener dans $[0,1]$ à l'aide la fonction $f : x \mapsto (x-m)\,/\,(M-m)$.