1 Les bases de la programmation en Python

Ce chapitre n'est pas un chapitre d'apprentissage du Python (pour cela, se reporter au tutoriel en ligne sur le site `https://docs.python.org/3/tutorial/index.html`, référence absolue en la matière). Il n'est pas destiné à être dispensé en cours, mais est plutôt à voir comme un aide-mémoire auquel se reporter tout au long de l'année pour les fonctionnalités les plus fréquemment utilisées de Python. Bien sûr, on ne peut pas être exhaustif, et l'aide en ligne, ainsi que le tutoriel, restent les références privilégiées pour aller au-delà de ce qui est exposé dans ce chapitre. Certains aspects évoqués dans ce chapitre seront développés dans les chapitres suivants.

En revenant à la page principale du site sus-cité `https://www.python.org/`, vous trouverez également différentes distributions de Python à télécharger et installer sur vos ordinateurs personnels (plus que conseillé !), notamment pour Windows ou MacOS. Pour une distibution Linux, renseignez-vous sur les forums ; il y a moyen de l'installer avec les moteurs d'installation (`yum install` pour Fedora par exemple). Sur certains systèmes, Python est installé par défaut (vérifier dans ce cas la version). Il existe des différentes notables entre les versions 2 et 3, les rendant incompatibles. Nous utiliserons la version 3 en TP. Choisissez donc la version 3 la plus récente.

Il peut également être utilise de télécharger un environnement de programmation (essentiellement un programme incluant dans un même environnement graphique un éditeur de code et un interpréteur permettant de taper des instructions en ligne, et de visualiser les résultats de vos programmes). Certains environnements sont généralistes (prévus pour la programmation dans plusieurs langages différents), comme Eclipse ; d'autres sont plus spécifiques à Python, comme IDLE, conçu par le concepteur de Python, ou Pyzo, que nous utiliserons en TP. Ce dernier est donc à privilégier sur vos ordinateurs personnels , afin de vous habituer. IDLE est plus minimaliste, pour ceux qui préfère la sobriété. Certains environnements viennent avec une distribution précise de Python (qu'il n'est dans ce cas pas nécessaire d'installer avant).

On peut aussi utiliser un éditeur de texte standard, comme `emacs`, puis lancer Python sur le fichier en ligne de commande. La plupart des éditeurs reconnaissent le langage Python et proposent une coloration syntaxique, mais ils ne disposent pas en revanche des aides syntaxiques lorsqu'on tape le nom d'une fonction.

I Python dans le paysage informatique

Les concepts évoqués dans cette mise en contexte seront expliqués un peu plus dans le chapitre suivant.

- Python est un langage hybride, adapté à la programmation impérative, tout en utilisant certaines facilités de la programmation objet, au travers de méthodes, applicables à des objets d'un certain type (on parle de classe). Il existe la possibilité de créer de nouvelles classes d'objets, et d'y définir des méthodes. Nous n'explorerons pas ces possibilités cette année, même s'il s'agit d'un point de vue très important.
- Python est un langage de haut niveau. Il gère tout le côté matériel sans que le programmeur ait à s'en préoccuper. La seule tâche du programmeur est l'aspect purement algorithmique. Le haut niveau se traduit aussi par la possibilité d'utiliser des méthodes élaborées sur les objets (recherche d'éléments dans une liste, tri, etc), et d'importer des modules complémentaires spécialisés suivant les besoins, dans lesquels sont définies des fonctions complexes (calcul matriciel, résolution d'équations différentielles, résolution de systèmes linéaires, méthode de Newton...). Les méthodes et fonctions ainsi définies cachent une grande partie de la technique au programmeur, qui n'a donc qu'à se préoccuper de l'essentiel. L'avantage est une facilité de programmation, et une utilisation de méthodes le plus souvent optimisées (plus efficaces en général que ce que peut espérer faire un programmeur débutant). L'inconvénient est une connaissance beaucoup plus floue de la complexité des algorithmes écrits. Par ailleurs, ces fonctions et méthodes, ainsi que la gestion du matériel, sont optimisés pour la situation la plus générale possible, mais pas pour les situations particulières. C'est pour cela qu'un programme en langage de plus bas niveau (en C par exemple) est souvent plus efficace, s'il est bien conçu, qu'un programme en Python. Pour cette raison, il est fréquent en entreprise d'utiliser Python comme langage de prototypage, puis de traduire en C pour plus d'efficacité, une fois que l'algorithmique est en place. Cela permet de scinder l'écriture du code en 2 phases, la phase algorithmique, et la phase programmation. De la sorte, il est possible d'utiliser du personnel plus spécialisé (des concepteurs d'algorithmes d'une part, et des programmeurs d'autre part, plus au fait du fonctionnement-même de l'ordinateur)
- Python est un langage semi-compilé. Il ne nécessite pas une compilation spécifique avant lancement. Il est compilé sur le tas, à l'exécution, de sorte à repérer certaines erreurs de syntaxe avant le lancement-même du programme, mais il ne ressort pas de fichier compilé utilisable directement. L'avantage est une certaine souplesse liée à la possibilité d'utiliser Python en ligne de commande (n'exécuter qu'une seule instruction dans un shell) : on peut ainsi tester certaines fonctions avant de les utiliser dans un programme, ce qui est parfois assez pratique. L'inconvénient est une lenteur d'exécution du programme définitif, puisqu'il faut le compiler à chaque utilisation : un langage compilé peut directement être exécuté à partir du code compilé, en assembleur, proche du langage machine, et est donc plus rapide à l'exécution. C'est une autre raison pour que Python servent plus au prototypage qu'à la finalisation, notamment pour tous les dispositifs nécessitant une réactivité assez importante (téléphones...)
- Contrairement à beaucoup de langages, Python est un langage dans lequel la présentation est un élément de syntaxe : les délimitations des blocs d'instructions se font par indentation (en pratique de 4 caractères), contrairement à de nombreux autres langages qui utilisent des débuts et fins de blocs du type `begin` et `end`. Ces indentations sont donc ici à faire de façon très rigoureuses pour une correction du programme. Une mauvaise indentation peut amener une erreur de compilation, voire un mauvais résultat (ce qui est plus dur à détecter). L'avantage est une plus grande clarté (imposée) du code par une délimitation très visuelle des blocs (même si cette indentation est de fait très fortement conseillée dans les autres langages, mais de nombreux programmeurs débutants la négligent).

Une fonction essentielle, et inclassable, est la fonction d'aide, que vous pouvez appliquer à la plupart des noms d'objets, de modules, de fonctions :

```
help()  # ouverture de la page d'aide de l'objet spécifié
```

Par exemple `help(print)` renvoie la page d'aide sur la fonction `print()` réalisant l'affichage d'une chaine de caractères.

II Les variables

En Python , les variables bénéficient d'un typage dynamique : le type est détecté automatiquement lors de la création de la variable par affectation. Il n'est donc pas nécessaire de déclarer la variable avant de l'utiliser (comme dans la plupart des autres langages).

II.1 Affectation

L'affectation est l'action consistant à donner une valeur à une variable.

```
a = 2  # affectation
```

L'affectation d'une valeur à une variable se fait avec l'égalité. La ligne ci-dessus donne la valeur 2 à la variable `a`, ceci jusqu'à la prochaine modification.
Une affectation peut prendre en argument le résultat d'une opération :

```
a = 2 + 4
```

Ce calcul peut même utiliser la variable `a` elle-même :

```
a = a + 4*a*a
```

Dans ce cas, le calcul du membre de droite est effectué d'abord, et ensuite seulement l'affectation. Ainsi, le calcul est effectué avec l'ancienne valeur de `a`. Il faut en particulier qu'une valeur ait déjà été attribuée à `a`. Par exemple, si `a` a initialement la valeur 2, la ligne ci-dessus attribue la valeur 18 à a.
Il existe des raccourcis pour certaines opérations de ce type :

```
Affectations avec opération:
a += 2   # Équivaut à a = a + 2
a -= 1   # Équivaut à a = a - 1
a *= 3   # Équivaut à a = a * 3
a /= 2   # Équivaut à a = a / 2 (division réelle)
a //= 2  # Équivaut à a = a // 2 (division entière)
a %= 3   # Équivaut à a = a % 3 (reste modulo 3)
a **= 4  # Équivaut à a = a ** 4 (puissance)
etc.
```

II.2 Affichage

En ligne de commande, il suffit de taper le nom de la variable après l'invite de commande :

```
>>> a = 2
>>> a
2
```

Faire un affichage du contenu d'une variable lors de l'exécution d'un programme nécessite la fonction `print()`, qui affiche une chaine de caractère. On peut aussi l'utiliser en ligne de commande

```
>>> print(a)
2
```

Cette fonction réalise une conversion automatique préalable du contenu de la variable en chaîne de caractères, car la fonction `print()` prend toujours en argument une chaîne de caractères (voir section « chaînes de caractères »).

II.3 Type et identifiant

Chaque variable possède un identifiant (l'adresse mémoire associée), et un type (la nature de l'objet stocké dans la variable). L'identifiant change à chaque réaffectation, le type peut changer lui-aussi. Les méthodes associées à un objet peuvent le modifier sans changement d'identifiant.

```
type(a)  # affiche le type (la classe) de la variable ou de l'objet
id(a)    # affiche l'identifiant (l'adresse en mémoire)
```

Les principaux types (hors modules complémentaires) :

```
int      # Entiers, de longueur non bornée
float    # Flottants (réels, en norme IEEE 754 sur 64 bits, voir chapitre 1)
complex  # Nombres complexes
bool     # Booléens (True / False)
list     # Listes
set      # Ensembles
tuple    # $n$-uplets
str      # Chaînes de caractères (string)
function # Fonctions
```

Par exemple :

```
>>> type(87877654)
<class 'int'>
>>> type(1.22)
<class 'float'>
>>> type(True)
<class 'bool'>
>>> type([1,2])
<class 'list'>
>>> type({1,2})
<class 'set'>
>>> type('abc')
<class 'str'>
>>> type(lambda x:x*x)
<class 'function'>
```

D'autres types plus complexes peuvent être rattachés à un type donné, comme les itérateurs de liste (objet permettant d'énumérer les uns après les autres les éléments d'une liste, ce qui permet d'appliquer à la liste une boucle `for`.

Nous rencontrerons d'autres types dans des modules spécifiques, en particulier les type `array` et `matrix` du module `numpy`, pour les tableaux et les matrices.

III Objets et méthodes

Python utilise certains concepts de la programmation objet, notamment à travers la possibilité de modifier un objet en lui appliquant une méthode. Pour lister l'ensemble des méthodes associées à une classe d'objets nommée `cl` :

```
help(cl)    # aide associée à la classe 'cl', en particulier, description des
            # méthodes définies sur les objets de cette classe
```

III.1 Les nombres

Trois classes essentiellement (entiers, flottants, complexes).
Les complexes s'écrivent sous la forme `a + bj`. Le réel b doit être spécifié, même si $b = 1$, et accolé à la lettre `j`. Par exemple `1+1j` désigne le complexe $1 + i$.

```
Opérations sur les nombres:
x + y          # somme de deux entiers, réels ou complexes
x * y          # produit de deux entiers, réels ou complexes
x - y          # différence de deux entiers, réels ou complexes
x / y          # division de réels (retourne un objet de type float même si
               # le résultat théorique est un entier; peut être appliqué à
               # des entiers, qui sont alors convertis en flottants
x // y         # division entière (quotient de la division euclidienne)
               # s'applique à des entiers ou des réels. Appliqué à des
               # entier, retourne un 'int', sinon un 'float'
x % y          # modulo (reste de la division euclidienne)
               # s'applique à des entiers ou réels.
divmod(x)      # renvoie le couple (x // y, x % y)
               # permet de faire les deux calculs en utilisant une seule
               # fois l'algorithme de la division euclidienne.
x ** y         # x puissance y
abs(x)         # valeur absolue
int(x)         # partie entière de l'écriture décimale (ne correspond pas
               # à la partie entière mathématique si x < 0:
               # par exemple int(-2.2)=-2.
x.conjugate()  # retourne le conjugué du nombre complexe x
x.real         # partie réelle
x.imag         # partie imaginaire
```

III.2 Les booléens et les tests

Les deux éléments de la classe des booléens sont `True` et `False` (avec les majuscules).

```
Opérations sur les booléens:
x.__and__(y)  ou   x & y     ou   x and y    # et
x.__or__(y)   ou   x | y     ou   x or y     # ou
x.__xor__(y)  ou   x ^ y                     # ou exculsif
                                 not x       # négation de x
```

Toute opération valable sur des entiers l'est aussi sur les booléens : le calcul se fait en prenant la valeur 0 pour `False`, et la valeur 1 pour `True`
La plupart du temps, les booléens sont obtenus au moyen de tests :

```
Tests:
x == y       # égalité (la double égalité permet de distinguer
             # syntaxiquement de l'affectation)
x < y        # infériorité stricte
x > y        # supériorité stricte
x <= y       # infériorité large
x >= y       # supériorité large
x != y       # différent (non égalité)
x in y       # appartenance (pour les listes, ensembles, chaînes de caratères)
x is y       # identité (comparaison des identifiants de x et y)
```

Attention, suivant la classe de `y`, le test de l'appartenance est plus ou moins long. Ainsi, si y est une liste, il se fait en temps linéaire, alors qu'il est en temps quasi-constant pour un ensemble.

III.3 Les listes

Une liste est une suite ordonnée d'objets, pouvant être de types différents. Ces objets peuvent éventuellement être eux-même des listes (listes imbriquées). Ils peuvent même être égaux à la liste globale (définition récursive).

```
Notation d'une liste
[1,2,3,4]      # énumération des objets entre []
[[1,2],[1],1] # liste dont les deux premiers termes sont des listes
[]             # liste vide
```

L'accès à un élément d'une liste est direct (contrairement à l'usage algorithmique théorique, et à de nombreux langages, ou l'accès se fait par chaînage, en suivant les pointeurs à partir du premier élément). L'indexation des éléments commence à 0 :

```
Accès aux éléments d'une liste par indexation positive:
>>> li = [1,2,3]
>>> li[0]
1
>>> li[2]
3
>>> li[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

On peut aussi accéder aux éléments par indexation négative. Le dernier élément de la liste est alors numéroté -1. Avec la liste de l'exemple précédent, on obtient par exemple :

```
>>> li[-1]
3
>>> li[-3]
1
>>> li[-4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Cette possibilité est surtout intéressante pour accéder aux derniers éléments d'une liste.

Voici une liste des opérations et méthodes les plus utilisées sur les listes. Pour une liste plus exhaustive, utilisez `help(list)`. On indique par un (m) le fait que la méthode modifie directement la liste sur laquelle elle agit. Les autres retourne le résultat en sortie de fonction.

```
Opérations et méthodes applicables à une liste:
len(L)              # longueur (nombre d'éléments) de L
L1 + L2             # concaténation des listes
n * L               # pour n entier: concaténation répétée de L avec elle-même.
L.append(a)         # ajout de l'objet a en fin de liste (m)
L.insert(i,a)       # insertion de l'objet a en position i (m)
L.remove(a)         # retrait de la première occurrence de a (m)
L.pop(i)            # retrait et renvoi de l'élément d'indice i (m)
                    # par défaut, si i non précisé: dernier élément
```

```
del(L[i])            # suppression de l'attribut d'incide i
L.index(a)           # position de la première occurrence de a
                     # ValueError si a n'est par dans la liste
L.count(a)           # nombre d'occurrences de a dans la liste
a in L               # teste l'appartenance de a à L
L.copy()             # copie simple de L
                     # attention aux problèmes de dépendance des attributs
L.reverse()          # retourne la liste (inversion des indexations) (m)
L.sort()             # trie la liste dans l'ordre croissant (m)
                     # (si composée d'objets comparables)
                     # voir ci-dessous pour des paramètres
```

Le tri d'une liste possède deux paramètres : un qui permet de préciser si le tri est croissant ou décroissant, l'autre qui permet de définir une clé de tri (une fonction : on trie alors la liste suivant les valeurs que la fonction prend sur les éléments de la liste)

```
Paramètres de tri:
li.sort()              # tri simple
li.sort(reverse=True)  # tri inversé
li.sort(key = f)       # tri suivant la clé donnée par la fonction f
```

La fonction donnant la clé de tri peut être une fonction prédéfinie dans un module (par exemple la fonction `sin` du module `math`), une fonction définie précédemment par le programmeur (par `def`), ou une fonction définie sur place (par `lambda`). Voir plus loin pour plus d'explications à ce propos.
Lors de la copie d'une liste (par la méthode `copy`, ou par affectation directe), les attributs gardent même adresse. Donc modifier la copie modifie aussi l'original, et réciproquement.
Pour copier avec modification d'adresse des attributs, utiliser un slicing (voir ci-dessous) `M = L[:]`. Cela ne rend pas la copie complètement indépendante de l'original, si les objets de la liste sont eux-même des structures composées (listes, ensembles...). Il faut dans ce cas faire une copie profonde, à l'aide d'une fonction définie dans le module `copy` :

```
Copie profonde (complètement indépendante à tous les niveaux):
>>> import copy
>>> M = copy.deepcopy(L)
```

Aux méthodes précédentes, s'ajoute pour les listes une technique particulièrement utile, appelée saucissonnage, ou slicing. Il s'agit simplement de l'extraction d'une tranche de la liste, en précisant un indice initial et un indice final. On peut définir un pas p, ce qui permet de n'extraire qu'un terme sur p (par exemple les termes d'indice pair ou impair en prenant $p = 2$)

```
Technique de slicing:
L[i:j]       # Extraction de la tranche [L[i], ... , L[j-1]]
L[i:j:p]     # De même de p en p à partir de L[i], tant que i+k*p < j
```

À noter que :

- Si le premier indice est omis, il est pris égal à 0 par défaut.
- Si le deuxième indice est omis, il est pris égal à la longueur de la liste par défaut (on extrait la tranche finale)
- Si le troisième indice est omis, il est pris égal à 1 par défaut (cas de la première instruction ci-dessus)
- Un pas négatif permet d'inverser l'ordre des termes
- Le slicing est possible aussi avec des indexations négatives.

Par exemple :

```
>>> M = [0,1,2,3,4,5,6,7,8,9,10]
```

```
>>> M[3:6]
[3, 4, 5]
>>> M[2:8:2]
[2, 4, 6]
>>> M[:3]
[0, 1, 2]
>>> M[3::3]
[3, 6, 9]
>>> M[::5]
[0, 5, 10]
>>> M[:2:4]
[0]
>>> M[:5:4]
[0, 4]
>>> M[-3:-1]
[8, 9]
>>> M[2:6:-3]
[]
>>> M[6:2:-3]
[6, 3]
```

III.4 Les ensembles

La structure d'ensemble (`set`) ressemble à celle de liste, mais sans ordre défini sur les ensembles, et sans répétition possible des éléments. Elle n'est en théorie pas au programme, mais peut s'avérer utile à l'occasion. On peut par exemple s'en servir pour supprimer les doublons dans une liste (en convertissant en ensemble puis à nouveau en liste : attention, cela perturbe en général fortement l'ordre des termes dans la liste). Elle peut aussi être intéressante lorsqu'on est surtout intéressé par des tests d'appartenance, plus rapides sur des ensembles que sur des listes.

```
Notation d'un ensemble
{1,2,3,4}      # énumération des objets entre {}
{1,1,2,3,4,4} # même ensemble que le premier
set()          # ensemble vide
```

Attention, `{}` désigne non pas l'ensemble vide, mais le dictionnaire vide (voir ci-dessous).

```
Principales opérations et méthodes applicables à un ensemble
len(S)                    # cardinal
S.add(a)                  # ajoute a à l'ensemble S, si pas déjà  présent (m)
S.discard(a)              # enlève a de S si c'en est un élément (m)
S.remove(a)               # comme discard, mais retourne une erreur
                          # si a n'est pas élément de S (m)
S.pop()                   # retourne et enlève un élément au hasard de S (m)
S.intersection(T)         # retourne l'intersection de S et T
S.union(T)                # retourne l'union de S et T
S.difference(T)           # retourne les éléments de S qui ne sont pas dans T
S.symmetric_difference(T) # retourne la différence symétrique
S.isdisjoint(T)           # retourne True ssi S et T sont disjoints
S.issubset(T)             # retourne True ssi S est inclus dans T
a in S                    # teste l'appartenance de a à S
S.copy()                  # copie simple de S
                          # Attention aux problèmes de dépendance des attributs
```

Ici encore, on peut faire une copie complètement indépendante à l'aide de `deepcopy` du module `copy`

Du fait même de la nature d'un ensemble, on ne peut pas extraire un élément par indexation, ni appliquer la méthode de slicing à un ensemble.

III.5 Les tuples

Les `tuples` sont des n-uplets. Ils sont notés entre parenthèses :

```
Notation d'un tuple:
(1,2,3,4)      # énumération des objets entre ()
```

Dans certains cas, l'omission des parenthèses est supportée.

Contrairement à une liste, ils ont une taille fixée (aucune méthode ne peut ajouter une coordonnée en place). On peut accéder à une coordonnée par indexation, mais pas la modifier :

```
>>> u = (2,3)
>>> u[1]
3
>>> u[1]=2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

```
Principales opérations et méthodes applicables à un tuple
len(U)                    # nombre de coordonnées
U+V                       # retourne la concaténation
n * U   ou   U * n        # concaténation répétée n fois
U.count(a)                # retourne le nombre d'occurrences de a
U.index(a)                # retourne le premier indice de a, ou une
                          # erreur si a n'est pas un attribut
a in U                    # teste l'appartenance de a à U
```

III.6 Les dictionnaires (HP)

Un dictionnaire (classe `'dict'`) permet de définir une association entre un ensemble d'indices (appelés clés, et pouvant être de type quelconque) et des valeurs. L'association entre une clé et une valeur est appelée un objet ou une entrée du dictionnaire (item)

```
d = {}                      # dictionnaire vide
d = {key1:val1, key2:val2} # création d'un dictionnaire à 2 entrées accessibles
                            # par les indices key1 et key2
d[key1]                     # accès à val1, associé à l'indice key1
del(d[key2])                # suppression de l'entrée associée à l'indice key2
```

Voici par exemple quelques manipulations sur les dictionnaires :

```
# Création d'un dictionnaire par ajouts successifs
>>> d = {}
>>> d['a']= (2,3,4)
>>> d[42]= 'quarante-deux'
>>> d['un']=1
```

```
# Affichage du dictionnaire d
>>> d
{'un': 1, 42: 'quarante-deux', 'a': (2, 3, 4)}

# Affichage d'une valeur du dictionnaire, associée à une certaine clé
>>> d[42]
'quarante-deux'

# Accès à une clé non définie
>>> d[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2

# Modification de la valeur associée à une clé
>>> d['a'] = 23
>>> d['a']
23
>>> d['a']+=3
>>> d['a']
26

# Suppression d'un élément
>>> del d['un']
>>> d
{42: 'quarante-deux', 'a': 26}

# Définition globale d'un dictionnaire
>>> e = {'a':1,'b':2,'c':3}
>>> e['a']
1
```

En plus de ce qui précède, voici quelques possibilités sur des objets de type dictionnaire :

```
d.keys()          # donne la liste des clés sous forme d'un itérable
d.values()        # donne la liste des valeurs sous forme d'un itérable
d.items()         # donne la liste des objets (entrées) sous forme d'un itérable
                  # Les éléments de cet itérable sont des couples (key,val)
d.pop(key)        # ressort l'objet associé à la clé key et supprime
                  # l'entrée correspondante dans le dictionnaire
d.popitem()       # ressort un couple (key,val) du dictionnaire et le supprime
                  # on n'a pas la main sur le choix du couple retourné
                  # Peut être utilisé lorsque le dictionnaire représente une
                  # pile d'attente de tâches à effectuer.
d.clear()         # Efface tout le contenu du dictionnaire.
```

Par exemple :

```
>>> d.keys()
dict_keys([42, 'a'])
>>> for a in d.keys():
...     print(a, ' ', d[a])
...
```

```
42  quarante-deux
a  26
>>> d.values()
dict_values(['quarante-deux', 26])
>>> d.items()
dict_items([(42, 'quarante-deux'), ('a', 26)])
>>> d
{42: 'quarante-deux', 'a': 26}
>>> d.pop('a')
26
>>> d
{42: 'quarante-deux'}
>>> d[(2,3)]= 5
>>> d.popitem()
(42, 'quarante-deux')
>>> d
{(2, 3): 5}
```

III.7 Les chaînes de caractères

Il s'agit d'un assemblage non interprété de caractères.

```
Notation d'une chaîne de caractères:
'abcd'       # énumération des caractères successifs entre ''
"abcd"       # énumération des caractères successifs entre ""
"""abcd"""   # énumération des caractères successifs entre """ """ (raw string)
```

L'intérêt d'avoir 2 délimiteurs possibles est de pouvoir utiliser sans problème une apostrophe ou un guillemet dans une chaîne de caractères. Si les deux apparaissent dans la chaîne, on peut utiliser une raw string. Celle-ci permet aussi d'utiliser des retours-chariot explicites. Sans raw string, les retours à la ligne se notent avec le caractère spécial '\n'.
Les raw string sont aussi utilisés pour commenter les fonctions (voir plus loin)

L'accès à un caractère d'une chaîne se fait par indexation de la même manière que pour les listes. Le premier caractère est indexé par 0. Les indexations négatives sont possibles dans la même mesure que pour les listes.

```
Accès aux éléments d'une chaîne par indexation positive:
>>> ch = 'abcd'
>>> ch[2]
'c'
>>> ch[-1]
'd'
```

Les slicings (saucissonnages) peuvent se faire également sur les chaînes de caractères, de la même manière que pour les listes :

```
>>> ch = 'azertyuiopqsdfghjklmwxcvbn'
>>> ch[3:12]
'rtyuiopqs'
>>> ch[20:10:-1]
'wmlkjhgfds'
>>> ch[::2]
'aetuoqdgjlwcb'
```

Voici une liste (non exhaustive) des opérations et méthodes les plus utiles. Voir `help(str)` pour (beaucoup) plus.

```
Opérations et méthodes sur les chaînes de caractères:
len(c)              # longueur (nombre de caractères)
print(c)            # affichage à l'écran de la chaîne c
c + d               # concaténation des chaînes
n * c               # concaténation répétée n fois
c.join(li)          # concaténation des chaînes de la liste li
                    # valable avec d'autres itérables
c.split(sep)        # scinde c en une liste de chaînes, en
                    # recherchant les séparateurs sep
c.center(n)         # centre n dans une chaîne de longueur n
c.format(x,y,...)   # insertion des nombres x, y,... aux endroits
                    # délimités par des {} dans la chaîne
d in c              # teste si d est sous-chaîne de c
c.count(d)          # nombre d'occurrences ne se recouvrant pas de la chaîne d
c.find(d)           # indice de la première occurrence de d
                    # renvoie -1 si pas d'occurrence
c.index(d)          # indice de la première occurrence de d
                    # erreur si pas d'occurrence
c.rfind(d)          # de même que find pour la dernière occurrence
c.rindex(d)         # de même que index pour la dernière occurrence
c.replace(d,e)      # remplace toutes les sous-chaînes d par e
```

Découvrez les nombreuses autres méthodes avec `help`. En particulier, plusieurs méthodes pour gérer majuscules et minuscules, ou pour des tests sur la nature des caractères.

Nous précisons la méthode `format`, particulièrement utile pour paramétrer l'aspect des insertions de nombres dans le texte. Pour commencer, nous donnons des exemples simples :

```
>>> 'Il y a {} arbres'.format(10)
'Il y a 10 arbres'
>>> 'Il y a {} tulipes et {} roses'.format(3,4)
'Il y a 3 tulipes et 4 roses'
```

Il existe différents paramètres sur le format des valeurs numériques. Ces paramètres sont à indiquer entre les accolades :

```
:g      # choisit le format le plus adapté
:.4f    # Écriture en virgule flottante, fixe le nombre de décimales, ici 4.
:.5e    # Écriture en notation scientifique, fixe le nombre de décimales, ici 5.
:<15.2e # Fixe la longueur de la chaîne (elle est remplie par des espaces),
        # et justifie à gauche. Le 2e a la même signification que plus haut.
:>15.2e # Fixe la longueur de la chaîne, et justifie à droite.
:^15.2e # Fixe la longueur de la chaîne, et centre.
```

Par exemple, après import du module `math` :

```
>>> 'pi␣vaut␣environ␣:␣{:^12.6f}!'.format(math.pi)
'pi␣vaut␣environ␣:␣␣␣3.141593␣␣!'
```

III.8 Les itérateurs

Les itérateurs sont des objets égrénant des valeurs les unes après les autres en les retournant successivement. On peut utiliser ces valeurs successives grace à l'instruction `for` (voir section sur les boucles)

L'itérateur le plus utile pour nous sera `range`, énumérant des entiers dans un intervalle donné.

```
L'itérateur range:
range(a,b,p)        # retourne les entiers de a à b-1, avec un pas p
range(a,b)          # de même, avec la valeur par défaut p=1
range(b)            # de même, avec la valeur par défaut a=0
```

Ainsi, `range(b)` permet d'énumérer les entiers de 0 à $b-1$.
Certaines classes composées possèdent une méthode `objet.__iter__()`, permettant de le transformer en itérateur. Cette conversion est rarement effectuée explicitement, mais intervient de façon implicite. Elle permet de faire des boucles dont l'indice porte sur les éléments de ces structures (par exemple les éléments d'une liste, d'un ensemble ou d'une chaîne de caractères).

```
Boucles portant sur des objets itérables:
for i in range(n): # exécute l'instruction instr pour toute valeur de
    instr          #     i entre 0 et n-1
for i in liste:    # exécute l'instruction instr pour toute valeur (non
    instr          #     nécessairement numérique) de i dans la liste liste
for i in chaine:   # exécute l'instruction instr pour tout caractère
    instr          #     i de la chaîne de caractère chaine.
```

On dit que les types `list`, `set` et `str` sont itérables.
Cela permet également des définitions de structures composées par construction puis par compréhension, en rajoutant une condition :

```
Définitions de listes et ensembles par compréhension:
[ f(i) for i in range(n) if g(i)] # liste des f(i) si la condition
                                  # g(i) est satisfaite
De même avec les ensembles. La clause if est facultative.
range peut être remplacé par un itérateur ou un objet itérable.
```

Par exemple :

```
>>> [i*i for i in [2,5,7]]
[4, 25, 49]
>>> {i for i in range(100) if i%7 == 2}
{65, 2, 37, 72, 9, 44, 79, 16, 51, 86, 23, 58, 93, 30}
```

Remarquez le désordre !

III.9 Conversions de types

Certains objets d'un type donné peuvent être convertis en un autre type compatible.

```
Conversions de type.
float(a)         # Convertit l'entier a en réel flottant
complex(a)       # Convertit l'entier ou le flottant a en complexe
str(x)           # Convertit l'objet x de type quelconque en
                 # une chaîne de caractères.
list(x)          # Convertit un itérable x (set, tuple, str, range)
                 # en objet de type liste.
set(x)           # Convertit un itérable x en set
tuple(x)         # Convertit un itérable x en tuple
eval(x)          # Convertit la chaîne x en un objet adéquat, si possible
float(x)         # convertit la chaîne x en flottant si possible
int(x)           # convertit la chaîne x en entier si possible
```

Par exemple :

```
>>> float(2)
2.0
>>> complex(2)
(2+0j)
>>> str({2,3,1})
'{1, 2, 3}'
>>> str(77+2)
'79'
>>> list({1,2,3})
[1, 2, 3]
>>> list('ens')
['e', 'n', 's']
>>> set([1,2,3])
{1, 2, 3}
>>> set(range(10))
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> tuple('abc')
('a', 'b', 'c')
>>> eval('2')
2
>>> eval('[2,3]')
[2, 3]
>>> float('24')
24.0
>>> int('24')
24
>>> eval('24')
24
>>> int('024')
24
>>> eval('024')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
    024
      ^
SyntaxError: invalid token
```

Remarquez que pour évaluer des chaînes représentant des entiers, `int` et `eval` ne sont pas équivalents : `int` est défini de façon plus large, en incluant des cas particuliers non pris en compte par `eval`.
En revanche, `eval` est bien pratique lorsqu'on ne connaît pas a priori le type de l'objet défini par la chaîne.

IV Structuration d'un programme

IV.1 Notion de programme

Un programme est constitué d'une succession d'instructions, écrites dans un fichier, structurées de façon rigoureuse à l'aide de la syntaxe appropriée. Cette succession d'instructions peut être précédée par la définition d'un certain nombre de fonctions (dans le même fichier, ou dans un fichier annexe qu'il faut

alors importer) qui pourront alors être utilisées dans le programme.

Les instructions peuvent être :

- une affectation
- l'utilisation d'une fonction, ou d'une méthode sur un objet, ou un calcul
- une structure composée (conditionnelle, itérative, gestion d'exception...)

Les structures composées peuvent porter sur des blocs de plusieurs instructions. Il est important de pouvoir délimiter de façon très précise ces blocs. Contrairement à beaucoup de langages utilisant des délimiteurs de type `begin` et `end`, la délimitation des blocs en Python se fait uniquement par l'indentation. Par exemple, comparons le deux petits programmes suivants :

```
# Importance des indentations
x = y = 1
for i in range(5):
    x += 1
    y += 1
print(x,y)
```

et :

```
# Importance des indentations
x = y = 1
for i in range(5):
    x += 1
y += 1
print(x,y)
```

Le premier renvoie `6 6` tandis que le second renvoie 6 2. En effet, l'incrémentation de y n'est pas dans la boucle, et n'a donc lieu qu'une fois dans le second programme.

La taille de l'indentation n'a pas d'importance, mais dans un bloc donné, il faut être cohérent (la même indentation sur chaque ligne du bloc). Cependant, dans un soucis d'unification, on adopte généralement la règle suivante (respectée par les environnements adaptés *via* la tabulation) :

> **Un nouveau bloc sera déterminé par une indentation de 4 caractères de chacune de ses lignes.**

Évidemment, des blocs peuvent être imbriqués les uns dans les autres. Dans ce cas, on additionne les indentations (un bloc à l'intérieur d'un autre bloc principal sera donc indenté de 4 espaces supplémentaires, soit 8 caractères).

Un programme peut interagir avec l'utilisateur, soit en lui demandant d'entrer des données (entrée), soit en agissant sur les périphériques (sortie).

- La demande d'entrée de donnée par l'utilisateur se fait avec :

  ```
  input('texte')
  ```

  Cette instruction affiche 'texte' à l'écran, puis attend que l'utilisateur entre une donnée, validée par la touche « Entrée ». On associe souvent cette instruction à une affectation de sorte à conserver la donnée fournie :

  ```
  >>> x = input('Que voulez-vous me dire? ')
  Que voulez-vous me dire? ZUT!
  >>> x
  'ZUT!'
  ```

Ici, le texte `'Que voulez-vous me dire? '` a été affiché par l'ordinateur, et l'utilisateur a tapé 'ZUT !', valeur qui a été stockée dans la variable x. Évidemment, cette fonction `input` est plus intéressante en programme qu'en ligne de commande, afin de pouvoir donner la main à l'utilisateur.

Attention, toute donnée entrée par l'utilisateur l'est au format `str`, y compris les valeurs numériques. Si on veut les utiliser dans un calcul, il faut donc d'abord les convertir, à l'aide de la fonction `eval`, `int` ou `float` :

```
>>> x = input('Entrez une valeur : ')
Entrez une valeur : 2
>>> x
'2'
>>> x = eval(input('Entrez une valeur : '))
Entrez une valeur : 2
>>> x
2
```

- Nous ne ferons cette année que des sorties sur écran, à l'aide de la fonction `print`, ou des sorties sur des fichiers. Nous reparlerons plus loin de la lecture et l'écriture de fichiers en Python.

IV.2 Les fonctions

Une fonction est un morceau de programme que l'on isole, et qu'on peut faire dépendre de paramètres. Cela permet de :

- dégager le cœur même du programme de toute technique en isolant cette technique dans des fonctions (clarté et lisibilité du code)
- pouvoir répéter une certaine action à plusieurs endroits d'un même programme, ou même dans des programmes différents (en définissant des fonctions dans des fichiers séparés qu'on importe ensuite)
- rendre certains actions plus modulables en les faisant dépendre de paramètres.

Une fonction prend en paramètres un certain nombre de variables ou valeurs et retourne un objet (éventuellement `None`), calculé suivant l'algorithme donné dans sa définition. Il peut aussi agir sur les variables ou les périphériques.
La syntaxe générale de la définition d'une fonction est la suivante :

```
def nom_de_la_fonction(x,y):
    """Description"""
    instructions
    return résultat
```

La chaîne de caractères en raw string est facultative. Elle permet de définir la description qui sera donnée dans la page d'aide associée à cette fonction. Les instructions sont les différents étapes permettant d'arriver au résultat. L'instruction spécifique commençant par `return` permet de retourner la valeur de sortie de la fonction. Si cette ligne n'est pas présente, la valeur de sortie sera `None` (pas de sortie). Cela peut être le cas pour une fonction dont le but est simplement de faire un affichage.
Voici un exemple :

```
def truc(x,y):
    """Que peut bien calculer cette fonction?"""
    while x >= y:
        x -= y
    return x
```

Par exemple, en ajoutant la ligne `print(truc(26,7))` et en exécutant le programme, on obtient la réponse 5.

Si on veut savoir ce que calcule cette fonction, on peut rajouter la ligne `help(truc)`. L'exécution du programme nous ouvre alors la page d'aide de la fonction `truc` :

```
Help on function truc in module __main__:

truc(x, y)
    Que peut bien calculer cette fonction?
(END)
```

ce qui en l'occurrence ne va pas vous aider beaucoup. Il va falloir faire marcher vos neurones...

On peut rendre certains paramètres optionnels, en leur définissant une valeur par défaut, qui sera utilisée si l'utilisateur n'entre pas de valeur pour le paramètre considéré. Il suffit par exemple de remplacer l'entête de la fonction précédente par :

```
def truc(x=100,y=7):
```

On peut alors utiliser la fonction `truc` de la façon suivante :

```
>>> truc(200,9)    # x = 200, y = 9
2
>>> truc(200,7)    # x = 200, y = 7, entrée manuelle
4
>>> truc(200)      # x = 200, y = 7, entrée par défaut pour y
4
>>> truc()         # x = 100, y = 7, entrées par défaut pour x et y
2
>>> truc(y=9)      # x = 100, y = 9, entrée par défaut pour x
1
```

Enfin, notons qu'on peut décrire une fonction ponctuellement (sans la définir globalement) grâce à l'instruction `lambda` :

```
lambda x: expression en x    # désigne la fonction en la variable x définie
                             # par l'expression
```

Cela permet par exemple de donner une fonction en paramètre (clé de tri par exemple), sans avoir à la définir globalement par `def`

IV.3 Les structures conditionnelles

Une seule structure à connaître ici :

```
if test1:
    instructions1
elif test2:
    instructions2
elif test3:
    instructions3
...
else:
    instructions4
```

Les clauses `elif` (il peut y en avoir autant qu'on veut), ainsi que `else` sont facultatives.
La structure est à comprendre comme suit :

- Les `intructions1` sont effectuées uniquement si le `test1` est positif

- Les instructions `instructions2` sont effectuées uniquement si le `test1` est négatif et le `test2` positif
- Les instructions `instructions3` sont effectuées uniquement si le `test1` et le `test2` sont négalifs et le `test3` est positif
- ...
- Les instructions `instructions4` sont effectuées dans tous les autres cas.

Le mot `elif` est à comprendre comme une abréviation de `else if`. Ainsi, la structure précédente est en fait équivalente à une imbrication de structures conditionnelles simples :

```
if test1:
    instructions1
else:
    if test2:
        instructions2
    else:
        if test3:
            instructions3
        else:
            instructions4
```

Avertissement 1.4.1

N'oubliez pas les double-points, et faites attention à l'indentation !

Remarque 1.4.2

Les tests peuvent être remplacés par toute opération fournissant une valeur booléenne. On peut en particulier combiner plusieurs tests à l'aide des opérations sur les booléens. Par ailleurs, Python autorise les inégalités doubles (par exemple un test `2 < x < 4`).

IV.4 Les structures itératives

Il faut distinguer deux types de structures itératives ; celles où le nombre d'itérations est connu dès le début (itération sur un ensemble fixe de valeurs), et celles où l'arrêt de l'itération est déterminée par un test.

La structure itérative `for` permet d'itérer sur un nombre fini de valeurs. Elle s'utilise avec un objet itérable quel qu'il soit. La boucle est alors répétée pour toute valeur fournie par l'itérateur. L'objet itérable étant fixé, le nombre d'itérations est connu dès le départ.

```
# Boucle for, pour un nombre d'itérations connu d'avance
for i in objet_iterable:
    instructions        # qui peuvent dépendre de i
```

Par exemple :

```
for i in [1,2,5]:
    print(i)
```

va afficher les 3 valeurs 1, 2 et 5.
Le classique `for i = 1 to n` (ou similaire) qu'on trouve dans la plupart des langages se traduit alors par :

```
for i in range(1,n+1):
    instructions
```

Si les instructions ne dépendent par de i, on peut remplacer `range(1,n+1)` par `range(n)`.
Que fait par exemple le code suivant ?

```
x = 0
for i in range(1,1001):
    x += 1 / i
```

La boucle `while` permet d'obtenir un arrêt de l'itération par un test. Il s'agit en fait plutôt d'une condition de continuation, puisqu'on itère la boucle tant qu'une certaine condition est vérifiée.

```
# Boucle while, pour l'arrêt des itérations par un test
while cond:
    instructions
```

Tant que la condition `cond` est satisfaite, l'itération se poursuit.
Remarquez qu'une boucle `for` traditionnelle (de 1 à n) peut se traduire aussi par une boucle `while` :

```
i = 1
while i <= n:
    instructions
    i +=1
```

En revanche, ce n'est pas le cas de la boucle `for` sur d'autres objets itérables.

IV.5 La gestion des exceptions

L'exécution de certaines instructions peut fournir des erreurs. Il existe différents types d'erreur en Python, par exemple : `ZeroDivisionError`, `ValueError`, `NameError`, `TypeError`, `IOError`, etc.

Dans certaines conditions, on veut pouvoir continuer l'exécution du programme tout de même, éventuellement en adaptant légèrement l'algorithme au cas problématique détecté. On dispose pour cela de la structure `try`, qui dans sa version la plus simple, s'écrit :

```
try:
    instructions1
except:
    instructions2
else:
    instructions3
```

L'ordinateur essaie d'effectuer la série d'`instructions1`.

- S'il survient une erreur d'exécution, il effectue les `instructions2` et saute les `instructions3`
- Sinon, il saute les `instructions2` et exécute les `instructions3`.

On peut aussi rajouter une clause `finally:` qui sera exécutée dans les deux situations.

La clause `except` est obligatoire, la clause `else` est facultative. Si on n'a rien a faire dans la clause `except`, on peut utiliser l'instruction vide `pass`.

La gestion des exceptions peut différer suivant le type d'erreur qui intervient. Dans certains conditions, il faut donc pouvoir distinguer plusieurs démarches à suivre suivant l'erreur obtenue. En effet, considérons l'exemple suivant :

```
y = input("Entrez une valeur à inverser : ")
try:
    x = 1 / eval(y)
except:
    print("0 n'a pas d'inverse, Banane!")
else:
    print(x)
```

Lançons l'exécution du programme. Si on entre la valeur 2, on obtient en sortie 0.5. Si on entre la valeur 0, on se fait injurier. Mais si on entre une lettre (disons a), c'est maintenant l'évaluation de a qui pose problème (car il ne s'agit pas d'une valeur numérique). Ainsi, l'ordinateur va détecter une erreur et donc exécuter la clause `except`. On va donc se retouver face à un « 0 n'a pas d'inverse, Banane ! » assez inapproprié. Pour cela, il faut pouvoir ne considérer que le cas où l'erreur obtenue est une division par 0. En effectuant `1/0` dans la console, on récupère le nom de cette erreur, et on peut indiquer que la clause `except` ne doit porter que sur ce type d'erreur :

```
y = input("Entrez une valeur à inverser : ")
try:
    x = 1 / eval(y)
except ZeroDivisionError:
    print("0 n'a pas d'inverse, Banane!")
else:
    print(x)
```

Le comportement n'est pas changé si on entre les valeurs 2 ou 0. En revanche, avec la lettre a, on obtient :

```
Entrez une valeur à inverser : a
Traceback (most recent call last):
  File "try.py", line 3, in <module>
    x = 1 / eval(y)
  File "<string>", line 1, in <module>
NameError: name 'a' is not defined
```

Si on veut pouvoir gérer différemment plusieurs types d'exceptions, on peut mettre plusieurs clauses `except` :

```
y = input("Entrez une valeur à inverser : ")
try:
    x = 1 / eval(y)
except ZeroDivisionError:
    print("0 n'a pas d'inverse, Banane!")
except NameError:
    print("C'est ça que t'appelles une valeur, Banane?")
else:
    print(x)
```

Cette fois, entrer la lettre a provoque le second message d'insultes.

On peut aussi provoquer volontairement des erreurs (on dit « soulever des erreurs ou des exceptions »). Cela se fait avec l'instruction `raise` :

```
raise NomErreur('texte')
```

Le `NomErreur` doit être un des types d'erreur existant (on peut éventuellement en définir d'autres, mais nous n'étudierons pas cette année cette possibilité), le texte est la petite explication s'affichant lors du message d'erreur :

```
>>> raise TypeError('Quel est le rapport avec la choucroute?')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Quel est le rapport avec la choucroute?
```

On peut par exemple utiliser cela pour définir une fonction sur un intervalle donné. Supposons que nous voulions définir $f : x \mapsto x^2 - 2$ sur l'intervalle $[-1, 3]$ (allez savoir pourquoi !). On peut vouloir soulever une erreur si on cherche à évaluer f en un réel x hors de cet intervalle. Voici comment faire :

```
def f(x):
    if (x<-1) or (x>3):
        raise ValueError('valeur hors du domaine de définition de f')
    return x ** 2 - 2

print('f({})={}'.format(2,f(2)))
print('f({})={}'.format(4,f(4)))
```

L'exécution de ce programme retourne le résultat suivant :

```
f(2)=2
Traceback (most recent call last):
  File "raise.py", line 8, in <module>
    print('f({})={}'.format(4,f(4)))
  File "raise.py", line 4, in f
    raise ValueError('valeur hors du domaine de définition de f')
ValueError: valeur hors du domaine de définition de f
```

À noter que soulever une erreur arrête *de facto* le programme, à moins d'inclure cette erreur dans une clause `try` de gestion des exceptions.

Enfin, on peut utiliser ce qu'on vient de voir dans le simple but de redéfinir les messages d'erreurs. En reprenant le premier exemple, et en ne gérant que l'exception relative à la division par 0, on peut par exemple ecrire :

```
y = input("Entrez une valeur à inverser : ")
try:
    x = 1 / eval(y)
except ZeroDivisionError:
    raise ZeroDivisionError("0 n'a pas d'inverse, Banane!")
else:
    print(x)
```

Pour la valeur 0, on obtient alors :

```
Traceback (most recent call last):
  File "try.py", line 3, in <module>
    x = 1 / eval(y)
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "try.py", line 5, in <module>
    raise ZeroDivisionError("0 n'a pas d'inverse, Banane!")
ZeroDivisionError: 0 n'a pas d'inverse, Banane!
```

À noter que le comportement est totalement différent du premier exemple, puisqu'ici, le programme est interrompu s'il y a une division par 0.

V Modules complémentaires

Nous terminons ce chapitre par une description rapide d'un certain nombre de modules qui nous seront utiles en cours d'année.

V.1 Utilisation d'un module

De nombreuses fonctions sont définies dans des modules spécialisées ; cela permet de ne pas encombrer la mémoire de la définition de plein de fonctions dont on ne se servira pas : on peut se contenter de charger (importer) les modules contenant les fonctions utilisées (ou même selectionner les fonctions qu'on importe dans chaque module). De cette manière, on peut définir un très grand nombre de fonctions, sans pour autant alourdir l'ensemble.
Pour utiliser une fonction `fonct` d'un module `mod`, il faut importer cette fonction, ou le module entier avant l'utilisation de cette fonction. Cela peut se faire de 3 manières :

- **Import simple du module**

```
import mod
  #Cette instruction permet de faire savoir à l'ordinateur qu'on
  #utilise le module \texttt{mod}. On peut alors utiliser les fonctions
  #de ce module en les faisant précéder du préfixe \texttt{mod} (nom du
  #module):
mod.fonct()
```

  L'utilisation de préfixes permet d'utiliser des fonctions ayant éventuellement même nom et se situant dans des modules différents. Certains noms de module sont longs, ou seront utilisés très souvent. On peut, au moment de l'import du module, créer un alias, qui permettra d'appeler les fonctions de ce module en donnant comme préfixe l'alias au lieu du nom complet :

```
import mod as al
al.fonct()
```

  Un alias d'usage très courant est :

```
import numpy as np
```

- **Import d'une fonction d'un module**

```
from mod import fonct
  # Cette instruction permet d'importer la définition de la fonction
  # \texttt{fonct}, qui peut alors être utilisée sans préfixe:
fonct()
  # les autres fonctions du module ne peuvent pas être utilisées, même
  # avec le préfixe.
```

  Attention, si une fonction du même nom existe auparavant (de base dans Python, ou importée précédemment), elle sera écrasée.

- **Import de toutes les fonctions d'un module**

```
from mod import *
  # Cette instruction permet d'importer la définition de toutes les
  # fonctions du module
fonct()
  # les autres fonctions du module peuvent être utilisées de même.
```

Attention aussi aux écrasements possibles. Par ailleurs, l'import est plus long à effectuer, et d'une gestion plus lourde, surtout si le module est gros.

Nous passons en revue les modules qui nous seront les plus utiles. Il existe une foule d'autres modules (se renseigner sur internet en cas de besoin), dans des domaines très spécifiques.

V.2 Le module `math`

Ce module contient les fonctions et constantes mathématiques usuelles, dont nous citons les plus utiles (utiliser `help()` pour avoir la liste de toutes les fonctions du module)

```
### Exponentielle et logarithme ###
e             # constante e, base de l'exponentielle
exp(x)        # exponentielle de x
log(x)        # logarithme népérien
log10(x)      # logarithme en base 10

### Fonctions trigonométriques ###
pi            # le nombre pi
cos(x)        # cosinus
sin(x)        # sinus
tan(x)        # tangente
acos(x)       # arccos
asin(x)       # arcsin
atan(x)       # arctan

### Fonctions hyperboliques ###
cosh(x)       # cosinus hyperbolique
sinh(x)       # sinus hyperbolique
tanh(x)       # tangente hyperbolique

### parties entières ###
floor(x)      # partie entière
ceil(x)       # partie entière par excès

### Autres fonctions ###
sqrt(x)       # racine carrée
factorial(x)  # factorielle (pour x entier)
```

V.3 Le module `numpy` (calcul numérique)

Il est impossible de faire le tour de façon rapide de ce module définissant un certain nombre d'objets indispensables en calcul numérique, en particulier l'objet matriciel, et toutes les règles associées. Nous ne faisons qu'un survol rapide de cet aspect, en vous laissant découvrir le reste à l'aide de l'aide ou des guides d'utilisation.

Le module `numpy` contient lui-même des sous-modules, en particulier le sous-module `linalg`. Le module principal `numpy` est supposé importé sous l'alias`np`. Si on n'utilise qu'un sous-module particulier, on peut importer uniquement ce sous-module (par exemple `import numpy.linalg as al` permettra d'utiliser toutes les fonctions de `linalg`, en se contentant de préciser le suffixe `al`).

```
### La structure array (np.array): tableau multidimentionnel ###
array(liste) # convertisseur: transforme une liste en tableau
             # En imbriquant des listes dans des listes, on peut
```

```
            # obtenir des tableaux multidimentionnels

### Exemples ###
np.array([1,2,3])      # tableau à 3 entrées (matrice ligne)
np.array([[1,2],[3,4]]) # matrice 2x2 de première ligne (1,2) seconde ligne (3,4)
np.array([[1],[2],[3]]) # matrice colonne

### fonctions de structure sur les array ###
np.shape(A)    # format du tableau (nombre de lignes, de colonnes...,
            # sous forme d'un tuple
np.rank(A)     # profondeur d'imbrication (dimension spatiale)
            # ATTENTION, ce n'est pas le rang au sens mathématique!
```

Quelques différences entre les listes et les `np.array` :

- Homogénéité : toutes les entrées doivent être de même type
- Le format est immuable. La taille est définie à partie de la première affectation. Pour initialiser, il faut donc souvent créer un tableau préalablement rempli de 0 ou de 1.

```
### Création de tableaux particuliers ###
np.ones(n)         # Crée une matrice ligne de taille n constituée de 1.
np.ones((n1,n2))   # matrice de taille n1 x n2, constituée de 1
                # se généralise à des tuples
np.zeros(n)        # Crée une matrice ligne de taille n constituée de 0.
np.zeros((n1,n2))  # matrice de taille n1 x n2, constituée de 0
                # se généralise à des tuples
np.eye(n)          # matrice identité d'ordre n (des 1 sur la diagonale)
np.diag(d0,...,dn) # matrice diagonale de coefficients diagonaux d1,...,d_n
np.linspace(a,b,n) # matrice ligne constituée de n valeurs régulièrement réparties
                # entre $a$ et $b$
np.arange(a,b,p)   # matrice ligne des valeurs de a à b en progressant de pas p
np.fromfunction(f,(n,))         # matrice ligne remplie des valeurs f(k) pour
                              # k de 0 à n-1
np.fromfunction(f,(n1,...,nd)) # De même pour plus de dimension
                              # f dépend ici de d variables
```

Les `np.array` permettent le calcul matriciel. Les opérations matricielles usuelles sont définies :

```
### Opérations matricielles ###
A + B                   # Somme de deux matrices
a * M                   # Multiplication par un scalaire
A * B                   # Produit coefficient par coefficient (produit de Schur)
                        # ATTENTION, ce n'est pas le produit matriciel
np.dot(A,B) ou A.dot(B) # Produit matriciel
np.linalg.det(A)        # déterminant
np.trace(A)             # trace
np.transpose(A)         # transposée
np.power(A,n)           # A puissance n
np.linalg.inv(A)        # inverse de A
np.linalg.eigvals(A)    # valeurs propres de A
np.linalg.eig(A)        # valeurs propres et base de vecteurs propres
```

Remarquez que dans la syntaxe suffixe `A.dot(B)`, il n'est pas nécessaire de préciser qu'on utilise le module `numpy`, puisque cette notation suffixe signifie qu'on utiliser une méthode associée à la structure d l'objet `A`, donc à la structure de `np.array`. Il ne peut donc pas y avoir d'erreur d'aiguillage.

La résolution de systèmes de Cramer est programmée aussi :

```
### Résolution de systèmes de Cramer ###
np.linalg.solve(A,b) # Donne l'unique solution du système de Cramer Ax=b
```

Enfin, le sous-module `numpy.random` définit un certain nombre de fonctions de génération aléatoire de nombres, suivant la plupart des lois classiques. Nous n'en citons que la loi uniforme, utiliser l'aide `help(numpy.random)` pour découvrir les autres en cas de besoin. On suppose `numpy.random` importé sous le nom `npr`

```
npr.randint(a)     # génère aléatoirement un entier de 0 à a-1 inclus,
                   # de façon uniforme
npr.randint(a,b)   # génère aléatoirement un entier de a à b-1 inclus,
                   # de façon uniforme
npr.random(a,b,n)  # génère un tableau 1 x n d'entiers aléatoirement
                   # choisis uniformément entre a et b-1
npr.random()       # génère aléatoirement un réel de [0,1[ de façon uniforme
npr.sample(n)      # génère un np.array 1 ligne n colonnes rempli aléatoirement
                   # par des réels de [0,1[ choisis uniformément
npr.sample(tuple)  # de même pour un tableau multidimentionnel de
                   # taille définie par le tuple
```

V.4 Le module `scipy` (calcul scientifique)

Le module `scipy` regroupe un certain nombre de fonctions de calcul scientifique (algorithmes classiques de calcul numérique). Nous donnons celles qui sont incontournables dans le cadre de votre programme :

```
### Calcul numérique ###
scipy.integrate.quad(f,a,b)    # intégrale de f de a à b
scipy.optimize.newton(f,a,g)   # résolution d'équation par la méthode de Newton
                               # initialisée au point a
                               # g doit être égal à f'
scipy.optimize.newton(f,a)     # de même par la méthode de la sécante
scipy.integrate.odeint(f,y0,T) # Résolution de l'ED y'=f(y,t)
                               # y0 est la valeur initial (t minimal)
                               # T est un tableau des temps auquels
                               # calculer les valeurs de y
```

V.5 Le module `matplotlib` (tracé de courbes)

Ce module donne un certain nombre d'outils graphiques, notamment pour le tracé de courbes. Nous n'utiliserons que le sous-module `matplotlib.pyplot`, que nous supposerons importé sous l'alias `plt`. Notons que le préfixe `matplotlib` fait référence au mot « matrice » et non « mathématique ».

```
plt.plot(L1,L2, label='nom') # trace la ligne brisée entre les points
                             # dont les abscisse sont dans la liste L1
                             # et les ordonnées dans la liste L2
                             # Le label sert à identifier la courbe

plt.title("titre")           # Définit le titre du graphe (affiché en haut)

plt.grid()                   # Affiche un grille en pointillés pour
                             # une meilleure lisibilité
```

```
plt.legend(loc = ...)        # Affiche une légende (associant à chaque
                             # courbe son label)
                             # loc peut être un nombre (1,2,3,4,5,...)
                             # plaçant la légende à un endroit précis
                             # loc peut aussi être plus explicitement:
                             # 'upper left' etc.

plt.savefig('nom.ext')       # Sauvegarde le graphe dans un fichier de ce nom
                             # L'extention définit le format
                             # Voir l'aide pour savoir les formats acceptés

plt.show()                   # Affichage à l'écran de la figure

plt.matshow(T)               # Affiche une image constituée de points dont
                             # les couleurs sont définies par les valeurs du
                             # tableau T de dimension 2 (à  voir comme un
                             # tableau de pixels)
```

V.6 Autres modules (`random`, `time`, `sqlite3`,...)

Parmi les autres modules que nous utiliserons, citons `random`, qui proposent des fonctions de génération aléatoire, qui peuvent rentrer en conflit avec celles définies dans `numpy.random` :

```
### Module random ###
random.randint(a,b)   # entier aléatoire entre a et b inclus.
                      # Notez la différence avec la fonction de numpy
sample(l,k)           # Liste aléatoire d'éléments distincts de la liste l
                      # Deux occurrences d'une même valeur dans l sont
                      # considérées comme distinctes.
```

A part pour cette dernière fonction qui peut s'avérer utile, ce module est plutôt moins fourni que celui de `numpy`

Le module `time` permet d'accéder à l'horloge. On s'en servira essentiellement pour mesurer le temps d'exécution d'une fonction :

```
### Module time ###
time.perf_counter() ou time.time()  # donne une valeur d'horloge correspondant
                                    # à l'instant en cours en seconde.
```

Cette fonction sera utilisée comme suit :

```
import time
debut = perf_counter()
instructions
fin = perf_counter()
temps_execution = fin - debut
```

Enfin, le module `sqlite3` nous sera utile pour traiter des bases de données en fin d'année. Il n'est pas utile de retenir les instructions ci-dessous dès maintenant. Supposons ce module importé sous le nom `sql`.

```
### Module sqlite3 ###
connection = sql.connect('base.db')
    # crée un objet de connexion vers la base base.db
```

```
cur = connection.cursor()
    # crée un curseur qui permettra des modifications de la base
cur.execute("instruction SQL")
    # Exécute l'instruction donnée, dans la syntaxe SQL,
    # dans la base pointée par le curseur
connection.commit()
connection.close()
    # fermeture de la connection à la base
```

VI Lecture et écriture de fichiers

Nous voyons enfin comment il est possible d'accéder aux fichiers en Python (lecture, écriture). Sont définies dans le noyau initial les fonctions suivantes :

```
### Lecture, écriture dans un fichier ###
f = open('nom_fichier','r' ou 'w' ou 'a')
     # ouvre le fichier nom_fichier (accessible ensuite dans la variable f)
     # nom_fichier peut contenir un chemin d'accès, avec la syntaxe standard
     # pouvant différer suivant le système d'exploitation.
     # 'r' : ouverture en lecture seule
     # 'w' : ouverture en écriture (efface le contenu précédent)
     # 'a' : ouverture en ajout (écrit à la suite du contenu précédent)
f.readline()
     # lit la ligne suivante (en commançant à la première) (renvoie un str)
f.readlines()
     # renvoie une liste de str, chaque str étant une ligne du fichier
     # en commençant à la ligne en cours s'il y a déjà  eu des lectures
f.write('texte')
     # Écrit le texte à la suite de ce qui a été écrit depuis l'ouverture
f.close()
     # ferme le fichier f.
```

Dans le module `os`, et plus précisément le sous-module `os.path` on trouve des fonctions permettant de gérer la recherche de fichiers dans une arborescence, et la manipulation des chemins et noms de fichiers.

```
### Fonctions de manipulation de chemins dans os.path ###
abspath('chemin')   # transforme un chemin (relatif) en chemin absolu
basename('chemin')  # extrait la dernière composante du chemin
dirname('chemin')   # extrait le répertoire
isfile('chemin')    # teste si le chemin correspond à un fichier
isdir('chemin')     # teste si le chemin correspond à un répertoire
splitext('chemin')  # revoie (partie initiale, extension de fichier)
etc.
```