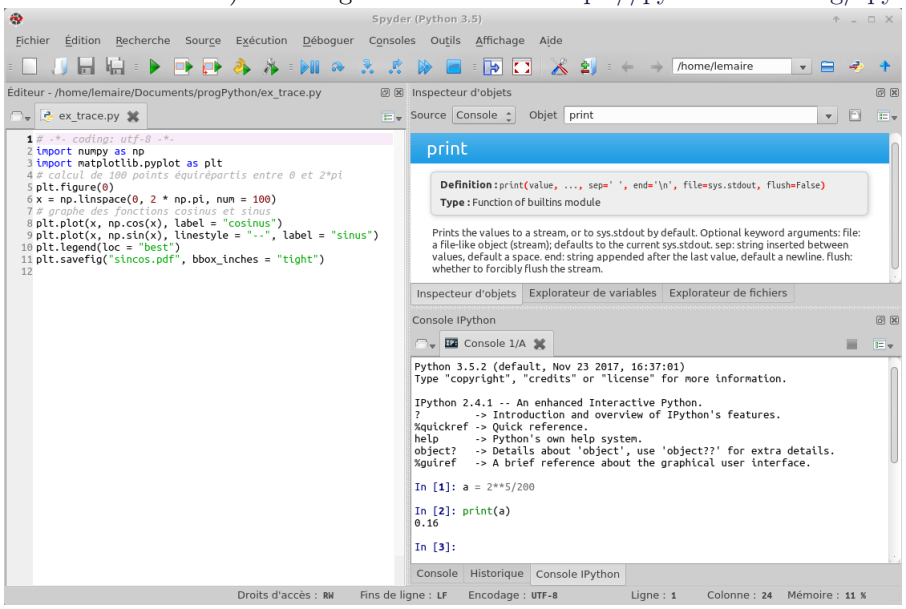
Introduction à Python 3 [1]

1 Les caractéristiques principales

Python est un langage conçu par Guido van Rossum. La première version est sortie en 1991. Son code est libre et gratuit (sous licence Open Source CNRI). De nombreuses « boites à outils » (appelées packages en Python) ont été développées notamment pour le calcul scientifique. Nous utiliserons les packages NumPy, SciPy et Matplotlib. Il existe actuellement deux versions différentes de Python avec quelques différences de syntaxes : versions 2.7 et 3.5. Nous utiliserons la version 3.5 avec l'environnement de programmation Spyder 3 (Scientific PYthon Development EnviRonment) téléchargeable à l'adresse https://pythonhosted.org/spyder.



Donnons rapidement quelques caractéristiques de Python pour le situer par rapport à d'autres langages.

— C'est un langage interprété : l'interpréteur exécute un programme en suivant les instructions une par une. (En fait, Python mélange à la fois du code interprété et du code compilé pour plus de rapidité. Les programmes sont compilés au moment de leur exécution en un code intermédiaire appelé *bytecode* qui est interprété par une machine virtuelle codée en C. Une grande partie du coeur du langage Python et de ses packages annexes sont écrits en C et compilé en langage natif).

— Tout est un objet dans Python dans le sens où tout peut être assigné à une variable ou passé comme argument à une fonction : les types de base comme les réels, les listes, les chaînes de caractères mais aussi les fichiers, les fonctions, les packages ...

— Tout objet manipulable par le programmeur possède un type bien défini à l'exécution, qui n'a pas besoin d'être déclaré à l'avance. Les variables servent à référencer les objets que l'on utilise ; on peut changer l'objet référencé par une variable au cours d'un programme.

— Python gère ses ressources (mémoire, descripteurs de fichiers...) sans intervention du programmeur, par un mécanisme de comptage de références.

— Python permet différents styles de programmation : programmation impérative (ce que nous ferons), programmation objet, mais aussi fonctionnelle.

— L'indentation du code est un élément de la syntaxe de Python : un bloc d'instructions sera défini comme composé de lignes d'instruction indentées de la même façon et précédées par le signe de ponctuation « : »

1.1 Environnement de programmation Spyder 3

Sur les ordinateurs des salles d'enseignement, on peut lancer l'environnement de programmation Spyder 3, dédié à la programmation en Python en tapant, dans un terminal, la commande

```
spyder3 &
```

1. Polycopié fait avec la version 3.5 de Python.
On trouvera les codes de certains exemples dans le répertoire `/commun/doc/lemaire/M322`

Le caractère & est optionnel : il permet de lancer un logiciel en tâche de fond et donc de pouvoir lancer d'autres commandes dans le terminal pendant que le logiciel fonctionne. Il permet de disposer :

- d'une fenêtre appelée *console* dans laquelle est lancé un interpréteur Python (zone en bas à droite sur l'image) permettant d'exécuter des lignes d'instructions. Le prompt, qui peut être `>>>` dans l'interpréteur de base ou `In[k]` pour l'interpréteur interactif IPython, invite à taper une instruction. On utilisera l'interpréteur IPython (mais pour simplifier la présentation du code dans ce polycopié on utilise `>>>` pour désigner le prompt).
- d'un éditeur de texte (zone à gauche sur l'image) qui permettra d'écrire un ensemble d'instructions Python et de les enregistrer dans un fichier avec l'extension `.py`. On pourra lancer l'exécution de ce fichier dans l'interpréteur Python à l'aide de la touche F5 ou de l'icône représentant une flèche verte (il est aussi possible de n'exécuter que quelques lignes de code du fichier. Passer la souris sur les différentes icônes du menu de Spyder pour voir apparaître leurs fonctions).
- d'une aide interactive en plus de l'aide de IPython [2] (fenêtre intitulée "Inspecteurs d'objet" en haut à droite sur l'image). Aller dans le menu "Outils" puis "préférences" pour configurer cette aide interactive afin par exemple de faire que la documentation sur une fonction apparaisse automatiquement lorsqu'on tape une parenthèse ouvrante.

Dans le menu Aide, se trouve un tutoriel expliquant le fonctionnement de Spyder.

2 Les variables

2.1 L'affectation

Les objets disposent d'une valeur que l'on peut faire afficher en utilisant la commande `print`(objet), d'un type que l'on obtient en tapant `type`(objet) et d'un identifiant `id`(objet) qui joue le rôle d'une adresse pour trouver l'objet dans l'espace mémoire de l'ordinateur.

Exemple 1.

```
>>> print(200)
200
>>> type(200)
int
>>> id(200)
10920736
```

Dans cet exemple, un objet de type int (pour « integer ») est crée dans l'espace de mémoire avec l'identifiant 10920736.

Les variables servent à référencer des objets (ce sont des noms servant à désigner des objets plus simplement que leur identifiant) . On utilise le symbole `=` pour effectuer une affectation d'un objet dans une variable :

```
variable = objet
```

Exemple 2.

```
>>> a = 200
>>> type(a), id(a)
(int, 10920736)
```

La variable `a` référence l'entier 200. Le type et l'identifiant de `a` sont ceux de l'objet que `a` référence.

NB : pour optimiser la place mémoire, certains objets comme les petits entiers ne sont créés qu'en un exemplaire dans la mémoire.

```
>>> b = a + 10
>>> id(a + 10), id(b)
(10921056, 10921056)
>>> a = b
>>> id(a)
10921056
```

2. L' annexe A.1 décrit quelques commandes d'aide utilisables dans la console.

A l'exécution de la commande `b = a + 10`, l'expression à droite du signe `=` est évalué, le résultat est un nouvel objet de type `int` qui est placé en mémoire avec l'identifiant 10921056. Cet objet est ensuite affecté à la variable `b`.
Lorsqu'on exécute ensuite la commande `a = b`, on obtient deux variables `a` et `b` qui référencent le même objet 210.

NB : Python libère de façon automatique de la place mémoire en supprimant les objets qui ne sont plus référencés par une variable.

On peut effectuer plusieurs affectations de variables simultanément : de chaque côté du signe "=", on sépare les noms des variables et les objets par des virgules.

Exemple 3.

```
a, b = 3, 10
```

La variable `a` référence l'entier 3 et la variable `b` l'entier 10.

Exercice 1. Affecter deux entiers distincts à des variables `x` et `y`. Puis exécuter l'instruction

```
x, y = y, x
```

Quel est l'objectif de cette instruction ?

2.2 Nom autorisé pour une variable

Le nom d'une variable ne peut contenir que des caractères alphanumériques (lettres de l'alphabet en minuscules ou majuscules, les chiffres) ou _ (l'underscore) ; les espaces, les autres caractères spéciaux tels que $, #, etc. sont interdits. Il ne peut pas commencer par un chiffre. Python distingue les minuscules des majuscules. Il est d'usage de commencer le nom d'une variable par une lettre minuscule. Il existe un certain nombre de mots réservés au langage (voir annexe A.2) et les variables commençant par un _ ont un statut particulier.

Le tableau suivant donne quelques caractères ayant une signification syntaxique dans Python (la signification sera précisée plus tard).

Caractères réservés	
#	pour écrire un commentaire sur une ligne
"""	pour marquer le début et la fin d'un commentaire écrit sur plusieurs lignes
" ou '	pour marquer le début et la fin d'une chaîne de caractères
;	pour séparer deux instructions écrites sur une même ligne
,	pour séparer deux éléments d'une séquence
\	pour une instruction écrite sur plusieurs lignes : indique, avant un passage à la ligne, que l'instruction se poursuit à la ligne suivante.
?	nom ? permet d'obtenir l'aide sur une commande ou un objet
:	pour marquer le début d'un bloc d'instructions
espace en début de ligne	pour distinguer les instructions appartenant à un bloc d'instructions

2.3 Les types de base prédéfinis dans Python

Contrairement à d'autres langages, la commande `a = b` ne provoque pas la duplication de l'objet référencé par `b`. On verra que cette différence est importante lorsque la valeur de l'objet référencé par `b` est modifiable (i.e. quand la valeur de l'objet peut changer en gardant le même identifiant). C'est pourquoi on distingue dans Python les objets dits *immuables*, des objets *mutables*.
Les tableaux ci-dessous décrivent les types des objets immuables et mutables de base (hors packages complémentaires).

Types immuables		exemple
`int`	entier de taille illimitée (limitée seulement par l'espace mémoire allouée à Python).	`200`
`float`	nombre à virgule codé en utilisant l'écriture en flottant sur 64 bites selon la norme IEEE 754 (voir annexe A.4)	`3.5`
`complex`	nombre complexe écrit en notation cartésienne : a + bj, a et b étant de type `float` et représentant les parties réelles et imaginaires respectivement (le nombre imaginaire `j` devra être précédé immédiatement d'un chiffre pour ne pas être vu comme le nom d'une variable : on l'écrira `1j` ou `1.j`)	`3.5 + 2j`
`bool`	booléen	`True` ou `False`
`str`	chaînes de caractères i.e. liste ordonnée de caractères délimitée soit par des apostrophes ', soit par des guillemets ", soit par 3 guillemets de suite pour des chaînes écrites sur plusieurs lignes.	`"figure 3"`
`tuple`	suite ordonnée d'objets de types hétérogènes	`(2,"Jean",3.5)`
`frozenset`	ensemble non ordonné d'objets distincts et immuables	`frozenset({1,'aa'})`
Types mutables		exemple
`list`	suite ordonnée d'objets de types hétérogènes (le parcours d'une liste est moins rapide que celui d'un tuple)	`['aa',2, [3]]`
`dict`	ensemble non ordonné de couples (clé : valeur), où clé est un objet immuable	`{'a' : 2, 'n' : 'Luc'}`
`set`	ensemble non ordonné d'objets distincts et immuables	`{1,'aa'}`

NB : Les chaînes de caractères, les listes et les ensembles sont des exemples d'objets appelés « *conteneurs* » car destinés à contenir plusieurs objets.

NB : `(3)` définit l'entier 3. Le tuple ayant un seul élément 3 s'écrit `(3,)`. Les parenthèses ne sont pas obligatoires, elles sont néanmoins utiles pour la lisibilité du code.

Exemple 4. Dans l'instruction suivante qui effectue une double affectation, les objets à droite et à gauche du signe « = » sont des tuples.

```
a, b = 1, 'xxx'
```

Aux objets du tuple de droite, sont affectés les noms `a` et `b` respectivement.

2.3.1 Opérateurs usuels

Opérations numériques	+ et -	* et /	// (quotient dans la division entière)	% (reste dans la division entière)	** (exposant)
Opérateurs relationnels	< et >	<= et >=	== (égalité de la valeur d'objets),	`is` (égalité des identificateurs d'objets)	! = (différent)
Opérateurs logiques	& `and`	| `or`	∧ (ou exclusif)	`not`	`in` (relation d'appartenance à un conteneur)

Certains de ces opérateurs sont aussi utilisables avec des conteneurs avec des significations adaptées. Par exemple l'opérateur `+` entre deux chaînes de caractères sert à définir la concaténation des deux chaînes.

Exemple 5.

```
>>> 5 % 2
1
>>> [2, 1] > [1, 4]     # comparaison pour l'ordre lexicographique de 2 listes
True
>>> [1, 3] < [1, 4]
True
>>> [1, 3] == (1, 3)
False                   # des objets de types différents ne peuvent être égaux
>>> ens = {1, 2, 5}
```

```
>>> (1 in ens) and  (0 not in ens) # les parenthèses sont ajoutées pour une meilleure lisibilité
True
```

Exercice 2. A l'aide des exemples suivants, étudier ce que font

1. les opérateurs `+` et `*` sur des listes ;
2. les opérateurs `-`, `>`, `|`, `&` sur des ensembles ;

```
l1 = [1, 2]; l2 = [8]
ens1 = {1, 2 ,4}; ens2 = {1, 7 ,4, 8};
l1 + l2
l1 * 3
ens3 = ens1 & ens2
print(ens3)
ens1 > ens3
ens1 - ens2
ens1 | ens2
```

L'exercice suivant présente quelques conséquences du codage des réels en utilisant la représentation en virgule flottante sur 64 bits (voir annexe A.4) selon la norme IEEE-754.

Exercice 3. Que retourne l'instruction suivante ?

```
0.3 - 0.1 == 0.2
```

Quelle est l'erreur relative dans le calcul suivant ?

```
>>> a = 1e-15 ; b = (1 + a) - 1
>>> b
 1.1102230246251565e-15
```

Quel est le résultat du calcul suivant ?

```
>>> x  = 1e+28;  y = 1e10
>>> x + y - x
```

2.4 Extraction de valeurs dans une séquence

Une séquence désigne un ensemble d'éléments ordonnés indexés par des entiers. Cela comprend les objets de type `list`, `tuple` et `str`. Le tableau indique comment extraire des valeurs d'une séquence notée `s`.

`s[0]`	1er élément de la séquence `s`
`s[k]`	$(k+1)$-ième élément de la séquence `s`
`s[-1]`	dernier élément de la séquence `s`
`s[i:j]`	extraction des éléments de `s` indexés par les entiers de `i` à `j-1`
`s[i:j:p]`	extraction des éléments de `s` indexés par les entiers de `i` à `j-1` de la forme `i` + k`p` où k est un entier

NB : dans la syntaxe `s[i:j:p]`

- si le 1er indice est omis, il est pris égal à 0.
- si le 2ème indice est omis, il est pris égal au nombre d'éléments de la séquence
- si le 3ème indice est omis, il est pris égal à 1. Sa valeur peut être négative.

Exemple 6.

```
>>> alp = 'abcdefgh'
>>> alp[:3]
'abc'
>>> alp[-2]
'g'
>>> alp[4:-1]
'efg'
```

```
>>> alp[4:]
'efgh'
>>> alp[4::-2]
'eca'
>>> l = [alp,[0,1,2,3]]
>>> l[0][2]
'c'
```

Exercice 4. On définit un tuple contenant les entiers de 10 à 20 :
`t = (10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)`
Ecrire une commande permettant d'extraire de `t` :

- le tuple (14, 15, 16)
- le tuple formé de tous les multiples de 3
- le tuple formé des nombres impairs entre 12 et 18 dans l'ordre croissant puis dans l'ordre décroissant.

2.5 Références multiples sur un objet mutable

Certaines opérations sur un objet mutable peuvent modifier la valeur de l'objet sans modifier l'identifiant. Les variables qui référençaient l'objet avant modification référencent toujours le même objet modifié. C'est le cas par exemple si on modifie un élément d'un objet de type `list` :

Exemple 7.

```
>>> a = [0, 2, 4]
>>> b = a
# comparaison de l'identifiant des objets
    référencés par a et b
>>> a is b
True
>>> b[1] = 'nom'
>>> a
[0, 'nom', 4]
```

```
>>> c = a[:]
>>> a is c
False
>>> c[1] = 'xx'
>>> a
[0, 'nom', 4]
>>> c
[0, 'xx', 4]
```

On remarque sur cet exemple que l'extraction de tous les éléments de `a` par l'instruction `a[:]` crée une copie de l'objet `a`. On parle de copie de surface car si les éléments sont eux-mêmes des conteneurs, cette copie ne duplique pas les éléments contenus dans ces conteneurs : ce qui est copié est `a[i]` qui est le nom de la variable qui référence un objet et non l'objet lui-même.

Exemple 8.

```
>>> m1 = [[1,2],[4,5]]
>>> m2 = m1[:]
>>> m1[1] = 'x'
>>> m1
[[1, 2], 'x']
>>> m2
```

```
[[1, 2], [4, 5]]
>>> m1[0][1] = 'y'
>>> m1
[[1, 'y'], 'x']
>>> m2
[[1, 'y'], [4, 5]]
```

3 Branchement et boucles

3.1 Branchement if [elif] [else]

L'instruction `if` permet d'exécuter ou non une portion d'un programme en fonction d'un test.

`if <condition>:` `  bloc d'instructions 1`	Le bloc d'instructions 1 n'est exécuté que si la valeur de <condition> correspond au booléen True.

Un bloc d'instructions est composé de lignes d'instruction indentées de la même façon (décalage de 4 espaces à droite) chaque instruction se termine simplement par un passage à la ligne suivi d'une indentation. Il est précédé par un « : ».

Exemple 9. Voici un exemple comprenant deux blocs d'instructions emboités.

```
chaine = input("Entrer une phrase :")
texte = 'phrase'
if chaine!="":
    if ' ' not in chaine:
        print("Votre phrase ne contient qu'un mot")
        texte = 'mot'
    print("votre", texte, "à l'envers : ", chaine[::-1])
```

NB : : la commande `input()` provoque l'arret de l'exécution du programme pour permettre à l'utilisateur de taper quelque chose au clavier. Le programme reprend lorsque l'utilisateur tape sur la touche <Enter> ; ce qu'il a tapé définit un objet de type `str` référencé par la variable `chaine`.

On peut compléter le branchement `if` pour exécuter des instructions seulement lorsque <condition> est faux :

`if <condition>:`
`  bloc d'instructions 1`
`else:`
`  bloc d'instructions 2`
	Ici, le bloc d'instructions 2 n'est exécuté que si <condition> correspond au booléen False.

Il est possible d'enchainer plusieurs branchements avec l'instruction `elif` qui est la contraction de `else` et de `if`. Par exemple,

`if <condition 1>:`
`  bloc d'instructions 1`
`elif <condition 2>:`
`  bloc d'instructions 2`
`else:`
`  bloc d'instructions 3`
	Si <condition 1> est satisfaite, le bloc d'instructions 1 est exécuté.
	Si <condition 1> n'est pas satisfaite et si <condition 2> est satisfaite alors le bloc d'instructions 2 est exécuté.
	Si ni <condition 1>, ni <condition 2> ne sont satisfaites, le bloc d'instructions 3 est exécuté.

Exemple 10.

```
a = 10
if a > 0:
    print("a est strictement positif")
elif a < 0:
    print("a est strictement négatif")
else:
    print("a est nul")
```

NB : Il existe une syntaxe compacte permettant de créer un objet à l'aide d'un branchement `if/else` en une instruction :

```
y = <objet 1> if <condition 1> else <objet 2>
```

Exemple 11. Supposons que `a` référence un réel et que l'on veuille définir `b` comme le signe de `a`.

```
b = 1 if a >= 0 else -1
```

NB : Lorsque c'est nécessaire les booléens True et False sont automatiquement transformés en les entiers 1 et 0 respectivement. On aurait donc pu définir `b` dans l'exemple précédent de la façon suivante :

```
b = 2*(a >= 0) - 1
```

NB : Un nombre égal à 0 (0, 0.0, 0+0j), un conteneur vide ((),[],"") et l'objet `None` (que l'on verra à la section 4) sont évalués comme le booléen False.

Exemple 12.

```
chaine = input("Entrer une phrase :")
if not chaine:
    print("Vous n'avez entré aucun caractère")
```

3.2 Boucle for

On utilise une boucle `for` pour exécuter des lignes d'instruction un nombre prédéfini de fois. Le bloc d'instructions que l'on veut exécuter plusieurs fois peut dépendre d'un paramètre `v` dont on souhaite faire varier la valeur à chaque passage dans la boucle.

`for v in obj_iterable:` `  <bloc d'instructions>`	`obj_iterable` doit être un objet itérable c'est-à-dire composé d'éléments que l'on peut parcourir successivement (comme les objets de type `list`, `tuple`, `str`, `set`). Le bloc d'instructions est alors exécuté avec une valeur de la variable `v` qui prend successivement chaque valeur des éléments composant l'objet `obj_iterable`.

NB : `obj_iterable` ne doit pas être modifié pendant l'exécution du bloc d'instructions. La variable `v` existe toujours une fois la boucle terminée.

Exercice 5.

1. Analyser l'exécution du code suivant.

```
>>> valeur = [3, -12, 1, 8]
>>> for v in valeur:
...     P = v**2 + 2*v - 3
...     if P == 0:
...         print(v, 'est une racine')
...     else:
...         print('le polynome en', v, 'vaut', P)
...
le polynome en 3 vaut 12
le polynome en -12 vaut 117
1 est une racine
le polynome en 8 vaut 77
```

2. Définir `valeur` comme l'ensemble contenant 3, -12, 1, 8 et exécuter de nouveau la boucle. L'affichage est-il le même ? Quelle est la valeur de v une fois la boucle exécutée ?

La fonction `range` permet de construire un *itérateur* qui permet d'itérer sur une liste d'entiers sans créer cette liste (pour gagner de la place mémoire) :

`range(n)`	pour itérer sur les entiers de 0 à `n-1`
`range(d, f)`	pour itérer sur les entiers de `d` à `f-1`
`range(d, f, p)`	pour itérer sur les entiers de la forme `d`+k`p` pour $k \in \{0, \dots, [\frac{\texttt{f}-1-d}{p}]\}$

Exemple 13. Ces quelques lignes calculent $n!$ pour $n = 1$ à 10.

```
n = 10
fact = 1
for k in  range(1,n+1):
    fact = fact*k
```

NB : On peut convertir l'itérateur créé par `range` en une liste en utilisant la fonction `list()` :

Exemple 14.

```
>>> list(range(2, 10, 3))
 [2, 5, 8]
```

Exercice 6. Le programme suivant calcule les premières valeurs d'une suite $(u_n)_n$ définie par récurrence telle que $u_0 = 1$. Donner la définition mathématique de la suite $(u_n)_n$. Quel est le plus grand entier j pour lequel u_j est affiché par ce programme ?

```
x = 1
for i in range(5):
    x = x/2 + 1/x
    print(x)
```

3.3 Définition par « compréhension » d'une séquence

Il existe une méthode concise pour créer une liste d'objets ou un ensemble d'objets à l'aide d'une boucle `for` :

`[f(x) for x in seq]`	liste dont les éléments sont les objets f(x) pour x une variable qui parcourt la séquence `seq`
`{f(x) for x in seq}`	ensemble dont les éléments sont les objets f(x) pour x une variable qui parcourt la séquence `seq`

Exemple 15.

```
>>> {(i-2)**2 for i in range(4)}
{0, 1, 4}
>>> [(i, (i-2)**2) for i in range(4)]
[(0, 4), (1, 1), (2, 0), (3, 1)]
>>> [i**2 for i in range(4) if i % 2 == 0]
[0, 4]
```

Exercice 7. Que fait l'instruction suivante ?

```
[[0]*k for k in range(1,4)]
```

3.4 La boucle conditionnelle `while`

La boucle conditionnelle « tant que » est utilisée pour exécuter une série de commandes tant qu'une expression logique est satisfaite.

`while <conditions>:` `  bloc d'instructions`	Si la valeur de <conditions> est False, le bloc d'instructions n'est pas exécuté. Si la valeur de <conditions> est True, le bloc d'instructions est répété tant que la valeur de <conditions> est True.

NB : il faut que dans le bloc d'instructions, il y ait une instruction qui change la valeur de <conditions> afin que le programme ne tourne pas indéfiniment.

Exercice 8. Analyser le programme suivant. Que représente la valeur affichée ? Quelle est sa valeur ?

```
x = 1
while 1 + x > 1:
    x = x / 2
print(x)
```

Instructions de rupture/prolongation	
`break`	termine l'exécution de la boucle la plus proche dans laquelle se trouve `break`
`continue`	passe directement à l'itération suivante de la boucle courante
Touches Ctr + C	interrompt l'exécution du programme

4 Les fonctions, modules et packages

Une fonction est un ensemble d'instructions que l'on regroupe sous un nom afin de pouvoir exécuter ces instructions facilement à plusieurs endroits d'un programme.

4.1 Fonctions existantes

On a déjà rencontré des fonctions `type`, `id` et `print`. La syntaxe pour appeler une fonction définie lors du chargement de Python est

```
nomDeLaFonction(listeArguments)
```

Exemple 16. Documentation de la fonction `sum` :

```
Return the sum of a 'start' value (default: 0) plus an iterable of numbers

When the iterable is empty, return the start value.
This function is intended specifically for use with numeric values and may
reject non-numeric types.
```

La documentation indique que le second argument de la fonction est optionnel ; voici des exemples d'utilisation :

```
>>> sum([1, 2, 4])
7
>>> sum([1, 2, 4], 5.5)
12.5
```

Exécuter la commande `sum(5)` donnerait un message d'erreur car le premier argument doit être un objet itérable.

4.2 Les fonctions attachées à un objet

Dans le vocabulaire de la programmation objet, une type de données comme `int`, `float`, ... sont des *classes*. Ce sont des programmes qui permettent de construire des objets et de définir

- les propriétés de ces objets, appelées *attributs*
- les opérations que l'on peut effectuer avec ces objets appelées *méthodes*.

Ce que l'on a appelé *objet* est un exemplaire particulier d'une classe : on parle d'*instance* de la classe. Les méthodes sont des fonctions définies dans une classe pour agir sur les instances de cette classe.

`obj.nomDeAttribut`	pour obtenir l'attribut `nomDeAttribut` de l'objet `obj`
`obj.methode`(listeArg)	pour appliquer la fonction `methode` sur l'objet `obj`

Exemple 17. Un objet de la classe `complex` possède :

- deux attributs : `imag` et `real` ;
- une méthode : `conjugate()`.
- des méthodes spéciales comme `__abs__`, `__add__`, `__mul__` ....

Les méthodes spéciales servent notamment à redéfinir des fonctions existantes pour les objets de type `float` afin de pouvoir les utiliser avec des nombres complexes. On parle de *surchage de méthodes*. Par exemple, `__abs__` permet d'utiliser `abs` pour calculer le module d'un nombre complexe. `__add__` et `__mul__` permettent de définir les opérations `+` et `*` entre deux nombres complexes.

```
>>> nb = 2 + 3.5j
>>> nb.imag
3.5
>>> nb.conjugate()
(2-3.5j)
>>> abs(nb)   #  même effet que nb.__abs__()
4.031128874149275
>>> nb * 2j   #  même effet que nb.__mul__(2j)
(-7+4j)
```

NB : Ne pas oublier les () après le nom d'une méthode si on l'appelle sans argument.

Exercice 9. La classe `dict` définit un type de données formées de couples (clé, valeur) permettant un accès rapide à une valeur :

- `d1 = dict()` crée un objet de type `dict` contenant aucune donnée avec comme nom `d1`.
- `d2 = dict(cle1 = val1, cle2 = val2)` crée un objet de type `dict` dont le nom est `d2` et qui s'écrit aussi : `{cle1: val1, cle2: val2}`.

NB : Les clés d'un dictionnaire doivent toutes être différentes. On accède à la valeur associée à la clé `cle1` du dictionnaire `d2` par `d2[cle1]`. A l'aide des exemples suivants, analyser le fonctionnement de quelques méthodes associées aux objets de type `dict`.

```
eng2fr = {'one': 'un', 'two' : 'deux', 'three' : 'trois'}
eng2fr.values()
eng2fr.keys()
for k, v in eng2fr.items():
    print(k, " --> ", v)
```

4.3 Les modules

Les fonctions intégrées au langage et disponibles au lancement de Python sont peu nombreuses. Il en existe beaucoup d'autres regroupées par thèmes dans des fichiers séparés, que l'on appelle des modules. Par exemple :

- Le module `math` est un module de la distribution standard de Python qui contient :
  - les définitions de nombreuses fonctions mathématiques : `acos`, `cos`, `exp`, `factorial`, `floor`, `gamma` ...
  - la définition de constantes : `e`, `pi`, `inf`, `nan`.
- Le module `copy` est un module de la distribution standard de Python qui contient deux fonctions permettant de faire des copies de conteneurs.
- Le module `time` est aussi un module de la distribution standard de Python qui contient notamment une fonction `time` qui affiche le nombre de secondes qui se sont écoulées depuis le 01/01/1970 et des fonctions permettant de convertir ce résultat en une date.

En pratique, un module est un fichier qui porte le nom du module suivi de l'extension `.py`. On importe les objets (fonctions, variables, classes) définis dans ce fichier à l'aide de l'instruction `import`.

`import` *nomModule*	pour importer l'ensemble des objets définis dans *nomModule*. L'objet *obj* de *nomModule* sera accessible sous le nom *nomModule.obj*.
`import` *nomModule* `as` *nomCourt*	pour importer l'ensemble des objets définis dans *nomModule* en renommant le module *nomCourt*.
`from` *nomModule* `import` *obj1*, *obj2*, ...	pour importer certains objets définis dans le module directement dans l'espace de noms courant : ils sont alors directement accessibles avec leur nom simple. Attention, si des objets ayant le même nom existaient avant l'importation, ceux-ci sont alors inaccessibles. On peut remplacer la liste des objets à importer par * si on veut tout importer (à éviter)

Exemple 18.

```
>>> import math as m
>>> from copy import deepcopy
>>> pas = m.pi/2
>>> v = [[i, m.cos(i * pas)] for i in range(3)]
>>> print(v)
[[0, 1.0], [1, 6.123233995736766e-17], [2, -1.0]]
>>> w = deepcopy(v)
>>> v2 = v
>>> v2[0][1] = 'xx'
>>> print(v)
[[0, 'xx'], [1, 6.123233995736766e-17], [2, -1.0]]
>>> print(v2)
[[0, 'xx'], [1, 6.123233995736766e-17], [2, -1.0]]
>>> print(w)
[[0, 1.0], [1, 6.123233995736766e-17], [2, -1.0]]
```

Lorsqu'on utilise l'instruction `import math as m`, le module `math` est importé avec son propre espace de noms ; la commande `dir(m)` permet d'afficher la liste des objets importés accessibles en faisant précédés leurs noms de `m.` (par exemple, `m.pi`). On peut utiliser l'autocomplétion dans IPython dès que l'on a tapé `m.`
Avec la commande `from copy import deepcopy`, on importe seulement la fonction `deepcopy` qui vient s'ajouter aux objets de l'espace courant.

NB : Le module chargé au lancement de Python est appelé ___main___.

Exercice 10. Exécuter le programme suivant qui permet de voir le temps d'exécution de trois codes différents pour créer la même liste.

```
import time
nb = 10000
deb1 = time.time()
carre1 = [i**2 for i in range(nb)]
print(time.time() - deb1)

deb2 = time.time()
carre2 = []
for i in range(nb):
    carre2 = carre2 +  [i**2]
fin2 = time.time()
print(time.time()  - deb2)

deb3 = time.time()
carre3 = []
for i in range(nb):
    carre3 +=  [i**2]
print(time.time() - deb3)
```

NB : Pour un réel `x`, l'instruction `x += 3` revient au même que l'opération `x = x + 3`. Pour une liste `x`, l'opérateur `+` correspond à l'opération de concaténation : `x = x + [3]` ajoute la liste `[3]` à `x` puis copie le résultat dans `x`, alors que `x += [3]` ajoute la liste `[3]` à `x` sans créer un nouvel objet.

4.4 Les packages

Un « package » correspond à un répertoire (ayant le nom du package) regroupant des modules et des fichiers de description de ce répertoire. La syntaxe pour importer un « package » est la même que pour un module. Un « package » peut contenir des « sub-packages » (correspondant à des sous-répertoires). Après l'import du package *nomPackage* par la commande :

`import` *nomPackage*

les objets d'un « sub-package » sont assessibles à l'aide de la syntaxe :

nomPackage.nomSubPackage.obj

On peut aussi importer simplement un module (ou un « sub-package ») d'un « package » :

1ère possibilité :

```
import nomPackage.nomModule
```

On accède à un objet de ce module par : `nomPackage.nomModule.obj`

2ème possibilité :

```
from  nom_package import nom_module
```

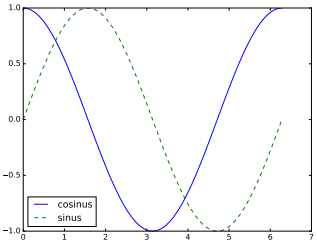
On accède à un objet de ce module par : `nomModule.obj`

NB : on peut bien sûr ajouter l'instruction `as` pour renommer l'espace de nom du module.

Exemple 19. Numpy est un « package » contenant des « subpackages » dont `linalg`, `random`, `polynomial`, `random`. matplotlib.pyplot est un module fournissant les outils permettant de faire facilement des graphiques

2D (ce module importe, dans le même espace de noms, un certain nombre de fonctions de plusieurs modules du « package » Matplotlib.

```
import numpy as np
import matplotlib.pyplot as plt
# calcul de 100 points équirépartis entre 0 et 2*pi
plt.figure(0)
x = np.linspace(0, 2 * np.pi, num = 100)
# graphe des fonctions cosinus et sinus
plt.plot(x, np.cos(x), label = "cosinus")
plt.plot(x, np.sin(x), linestyle = "--", label = "sinus")
plt.legend(loc = "best")
plt.savefig("sincos.pdf", bbox_inches = "tight")
```



4.5 Créer ses propres fonctions

Dans Python, la syntaxe pour créer une fonction (qui exécute un ensemble d'instructions et retourne un objet) ou une procédure (qui exécute un ensemble d'instructions sans rien renvoyer) est la même (on ne fera pas la distinction, on n'utilisera que le terme de fonction) :

```
def nomDeLaFonction(paramOblig,paramOpt):
    """
    description de ce que fait la
    fonction
    et des paramètres
    """
    bloc d'instructions
```

Les paramètres d'entrée de la fonction sont séparés par des virgules. La liste doit commencer par les paramètres obligatoires (ils sont désignés par un nom de variables) et se terminer par les paramètres optionnels pour lesquels on donne une valeur par défaut sous la forme `nomParam = valeur`. Une fonction doit obligatoirement avoir un bloc d'instructions.

Une fonction est un objet qui est créé au moment où on fait lire le code de la fonction à Python. Les valeurs par défaut des paramètres optionnels doivent être fixées à ce moment-là. Ensuite, la fonction créée peut être utilisée comme les fonctions déjà existantes.
Les instructions figurant à l'intérieur de la fonction seront exécutées seulement au moment où on appelle la fonction.
La description écrite entre deux """, juste après la définition d'une fonction, est appelée *docstring*. Elle apparait lorsqu'on exécute l'instruction `help(nomDeLaFonction)`.

Exemple 20. Exemple de fonction effectuant seulement un affichage.

```
>>> def table_multiplication(mult, deb = 1, fin = 3):
...     """affiche la liste k * mult pour tous les entiers k de deb à fin
...        mult : entier
...        deb, fin : paramètres optionnels par défaut deb = 1 et fin = 3
...     """
...     for i in range(deb,fin+1):
...         print(i, 'x', mult, '=', i*mult)
...
>>> table_multiplication(8)
1 x 8 = 8
2 x 8 = 16
3 x 8 = 24
>>> table_multiplication(8, 9, 10) # identique à table_multiplication(8, fin=10, deb=9)
9 x 8 = 72
10 x 8 = 80
```

NB : : A la place de `table_multiplication(8, 9, 10)`, on aurait pu taper :
`table_multiplication(8, fin=10, deb=9)` ou `table_multiplication(fin=10, deb=9, mult=8)` par exemple,

mais pas `table_multiplication(fin=10, deb=9, 8)` ; Python distingue deux types d'arguments suivant la façon dont on les écrit : « *keyword arguments* » ceux que l'on entre par la syntaxe `nomParam = valeur` et les autres appelés « *positional arguments* ». Ces derniers doivent apparaitre en premier dans la liste des arguments et dans l'ordre fixé par la définition de la fonction.

4.5.1 L'instruction `return`

Pour qu'une fonction retourne un objet, on utilise l'instruction `return` suivie de la liste des objets à retourner. Une fonction peut contenir plusieurs instructions `return`. L'exécution du bloc d'instruction s'arrête à la première instruction `return` rencontrée. Si lors de l'exécution de la fonction, aucune instruction `return` n'est rencontrée alors l'objet appelé `None` est renvoyé. C'est le cas de la fonction `table_multiplication`.

Exemple 21. Exemple d'une fonction retournant un tuple :

```
>>> def mesPave(long, larg = None, haut = None):
...     """ retourne un tuple (surf, volume) contenant  la surface
...         du pave de base le rectangle de  longueur long et de largeur larg et de hauteur haut.
...         Par défaut, larg = long et haut = long
...     """
...     h = long if haut == None else haut
...     if larg is None:
...         base = long ** 2
...         return 2 * base + 4 * h * long, base * h
...     else:
...         base = long * larg
...         return 2 * base + 2 * h * (long + larg), base * h
...
>>> surface, volume =  mesPave(10, 2)
>>> print("Sa surface  est", surface, "et son volume est", volume)
Sa surface  est 280 et son volume est 200
>>> surface, volume =  mesPave(10, 2, 3)
>>> print("Sa surface  est", surface, "et son volume est", volume)
Sa surface  est 112 et son volume est 60
```

Dans cet exemple, on ne peut pas définir `long` comme valeur par défaut à l'argument `larg`, car la valeur de `long` n'est pas connue au moment où l'objet de type `function` et de nom mesRectangle est créé.

NB : Les variables définies à l'intérieur d'une fonction sont des variables locales. Une fois l'exécution terminée, elles sont détruites. Par exemple, `h` et `base` sont des variables locales à la fonction `mesPave`.
Si une instruction dans une fonction utilise une variable `x`, la valeur de celle-ci est cherchée parmi les variables locales à la fonction. Si aucune variable locale à la fonction ne s'appelle `x`, Python cherche parmi les variables existantes dans l'espace mémoire au moment de l'appel de la fonction et affichera un message d'erreur, s'il n'en trouve pas.

Exercice 11. La fonction suivante calcule la valeur d'un polynome en un réel `x`. Donner l'expression de ce polynome. Compléter sa documentation en précisant ce qu'elle calcule et en ajoutant des exemples d'appels de cette fonction.

```
def calcul(x, listcoeff = [0]):
    """ calcule la valeur  en x ....
    x : réel
    listcoeff : liste de réels

    exemples :
    """
    result = 0
    for coef in listcoeff:
        result = result * x + coef
    return result
```

4.6 Compléments sur les arguments d'une fonction

Au moment de l'appel d'une fonction, ce sont les références aux objets qui sont passés comme arguments de la fonction et non une copie des objets. C'est pourquoi on utilise de préférence des arguments optionnels qui sont immuables (`int`, `float`, `str`, `bool`, `tuple`) car la modification d'un paramètre par un premier appel de la fonction est visible aux appels suivants. Si on a besoin d'un argument optionnel qui soit modifiable (`list`, `dict`), on utilise `None` comme valeur par défaut.

Exemple 22. Exemple d'effet de bord :

```
>>> def cumul(val, total=[0]):
...     total[0] = total[0] + val
...     print(total)
...
>>> cumul(3)
[3]
>>> cumul(3)
[6]
```

Passer une fonction comme argument d'une fonction Une fonction est un objet de type `function`. On peut donc les passer comme arguments de fonctions comme n'importe quel objet.

Exercice 12. Analyser la fonction suivante et la transformer afin qu'elle fonctionne avec un argument `x` qui soit une liste de réels : elle devra alors retourner la liste des valeurs de fun(x[i] + coef, param) pour i parcourant les indices des éléments de x.

```
def compositiontransl(fun, x, coef, param = None):
    """
    retourne la valeur de fun(x + coef, param)
    ou fun designe une fonction reelle qu'on appelle par  fun(x) ou fun(x, param)
    avec param la liste des paramètres de la fonction fun.

    Exemple :
    >>> compositiontransl(abs, -6, 3)
    3
    >>> def poly2(x,a):
            return a[0]*x**2 + a[1]*x + a[2]
    >>> compositiontransl(poly2, 1, -2, param = (3, 0, -1))
    2
    """
    if  param is None:
        return fun(x + coef)
    else:
        return fun(x + coef, param)
```

Exercice 13. Une façon d'approcher numériquement la dérivée d'une fonction f en un point x est de calculer le taux de variation de f entre $x + h$ et x pour un $h > 0$. Ecrire une fonction `tauxVariation` qui renvoie la valeur de ce taux de variation en un point x : elle aura comme paramètres x, h et f de sorte de pouvoir ensuite exécuter le programme suivant :

```
import math
for i in range(3,12):
    h = 10**(-i)
    m = 0
    for x in range(4):
```

```
        ecart = abs(tauxVariation(x, h, math.sin) - math.cos(x))
        if ecart > m:
            m = ecart
    print(m)
```

Expliquer ce qu'affiche le programme, l'exécuter et analyser les résultats affichés.

Distribution des arguments à l'aide d'une séquence. Il est aussi possible de passer une séquence *seq* (de type `tuple` ou `list`) qui sera *décompressée* en une liste d'arguments d'une fonction grâce à la notation `*seq`. De même, il est possible de passer un dictionnaire *dic* en utilisant la notation `**`*dic* pour définir les valeurs d'arguments optionnels : les clés du dictionnaire doivent alors correspondre aux noms des arguments optionnels.

Exemple 23. Voici différentes façons d'utiliser la fonction `mesPave` écrite à l'exemple 21 :

```
>>> cotes = (10, 2, 3)
>>> mesPave(*cotes)
(112, 60)
>>> larghaut = {'larg' : 2, 'haut' : 3}
>>> mesPave(10, **larghaut)
(112, 60)
>>> h = {'haut' : 3}
>>> mesPave(10, **h)
(320, 300)
```

Fonction avec un nombre arbitraire d'arguments Le passage d'un nombre arbitraire d'arguments sans nom est permis en utilisant la notation d'un argument final `*argsuppl`. Les paramètres surnuméraires sont alors transmis sous la forme d'un tuple affecté à cet argument (que l'on appelle généralement `argsuppl`).

Exercice 14. Analyser le fonctionnement de la fonction suivante :

```
>>> def sommePonderee(*val, poids = (1, )):
...     nw = len(poids)
...     nval = len(val)
...     s = 0
...     if nw == 1:
...         s = sum(val) * poids[0]
...     else:
...         for i in range(nw):
...             s = s + val[i] * poids[i]
...     return s
...
>>> print(sommePonderee(1, 2, 4, poids =  (2, )))
14
>>> print(sommePonderee(1, 2, 4, poids =  (2, 1, 3)))
16
>>> print(sommePonderee(1, 2, 4))
7
>>> data = [1, 2, 4]
>>> print(sommePonderee(*data, poids = (2, 1, 3)))
16
```

De la même façon, il est possible d'autoriser le passage d'un nombre arbitraire d'arguments optionnels nommés en plus de ceux prévus lors de la définition en utilisant la notation `**argOptsuppl`. Les paramètres optionnels surnuméraires nommés sont alors transmis sous la forme d'un dictionnaire affecté à cet argument.

Exemple 24.

```
>>> def unDict(**argOpt):
...     return argOpt
...
>>> unDict(a=3, b=3+2j, c='Paul')
{'c': 'Paul', 'b': (3+2j), 'a': 3}
```

Exercice 15. Analyser le fonctionnement de la fonction suivante :

```
def conversioneuro(*val, **monnaies):
    """ pour chaque valeur dans val, affiche la valeur dans les monnaies précisées dans monnaies.

    paramètres :
    val : liste de nombres
    monnaies : liste de la forme nom=cours en euros de la monnaie nom

    Exemple :
    >>> conversioneuro(2.5, 150, USDollar=1.05, CanadaDollar = 1.4)
    2.5 euros = 2.625 USDollar
    150 euros = 157.5 USDollar
    2.5 euros = 3.5 CanadaDollar
    150 euros = 210.0 CanadaDollar
    """
    for nom, cours in monnaies.items():
        for x in val:
            print(x, 'euros =', x * cours, nom)
```

5 Les tableaux du package Numpy

La convention est d'importer le package Numpy avec la commande :

```
import numpy as np
```

Ce package permet de manipuler un nouveau type d'objets mutables appelé `ndarray` correspondant à des tableaux multidimensionnels ; les principales caractéristiques des tableaux qui les distinguent des listes emboitées.

- les données du tableau doivent être toutes du même type ;
- le nombre de données dans le tableau est fixé à la création de l'objet ;
- si le tableau a plusieurs dimensions, chacune contient le même nombre d'éléments ;

Ces caractéristiques font qu'un objet de type `ndarray` a besoin de stocker moins d'informations sur les données qui le compose qu'un objet de type `list` et qu'il est possible de faire des opérations rapidement sur tous ses éléments sans utiliser de boucle. Pour ces raisons, on privilégiera l'utilisation des objets de type `ndarray` plutot que de type `list` lorsque cela est possible.
On se limitera dans la suite à décrire des tableaux de dimension 1 et 2.

5.1 Description d'un tableau

- le format (« *shape* ») d'un tableau est décrit par un tuple formé d'entiers strictement positifs. Par exemple, `(3,)` pour un tableau unidimensionnel à 3 éléments et `(3,2)`, pour un tableau bi-dimensionnel à 3 lignes et 2 colonnes.
- Les dimensions d'un tableau sont appelées (« *axis* ») et numérotées à partir de 0. Par exemple, pour un tableau bi-dimensionnel, l'axe 0 correspond aux lignes et l'axe 1 aux colonnes.

Quelques attributs d'un tableau numpy tab	
`tab.shape`	tuple d'entiers décrivant le nombre d'éléments pour chaque axe du tableau `tab`.
`tab.ndim`	nombre de dimensions (ou axes) du tableau `tab`.
`tab.size`	nombre total d'éléments du tableau `tab`.
`tab.dtype`	type des éléments du tableau `tab`.

Les tableaux ci-dessous décrivent quelques fonctions permettant la création et le redimensionnement des tableaux numériques 1D et 2D. Utiliser l'aide de Python pour avoir plus de détails sur l'utilisation de chaque fonction.

Création de tableaux 1D	
`np.array(v)`	les éléments sont les coefficients de la liste `v`
`np.linspace`(a,b,n)	les éléments sont n valeurs équiréparties entre `a` et `b` en commençant par a (par défaut b est le n-ième élément)
`np.arange`(a,b,p)	les éléments sont les premières valeurs d'une suite arithmétique de raison p commençant en a jusqu'à b exclus

Création de tableaux 1D (si dim est un entier) et 2D si (dim est un tuple contenant 2 entiers)	
`np.ones(dim)`	les éléments valent tous 1.
`np.zeros(dim)`	les éléments valent tous 0. (de type `float` par défaut)
`np.full(dim, a)`	les éléments valent tous `a`
`np.eye(dim)`	les éléments valent 1 sur la diagonale et 0 ailleurs
`np.fromfunction(f, dim)`	l'élément indexé par x vaut f(x)

Création d'un tableau 2D à partir d'une liste ou d'un tableau 1D appelé `v`	
`np.reshape`(v,(i,j))	création d'un tableau à i lignes et j colonnes en remplissant ligne par ligne avec les éléments de `v` (si ij est le nombre d'éléments de `v`)
`np.diag(v, k)`	matrice dont les éléments sur la `k`-ième diagonale sont donnés par la liste ou le tableau 1D `v` et dont les autres éléments sont nuls (k=0 par défaut).

Exemple 25.

```
>>> import numpy as np
>>> m = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(m, "dimension :",  m.ndim, "format :", m.shape)
[[1 2 3]
 [4 5 6]] dimension : 2 format : (2, 3)
>>> print("type des coef :", m.dtype, "nb d'elts :", m.size)
type des coef : int64 nb d'elts : 6
>>> x = np.linspace(1,9,10)
>>> m2 = np.reshape(x,(2,5))
>>> np.diag([1, 2, 4])
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 4]])
>>> np.full((2,3), 3.5)
array([[ 3.5,  3.5,  3.5],
       [ 3.5,  3.5,  3.5]])
>>> np.fromfunction(np.sin, (3,))
array([ 0.        ,  0.84147098,  0.90929743])
```

5.2 Extraction de valeurs

Si `m` est un tableau 1D alors l'extraction de valeurs se fait de la même façon que pour une liste.
Si `m` est un tableau 2D, `m[i,j]` permet d'extraire l'élément de la `i+1`-ème ligne et de la `j+1`-ème colonne.
Comme pour les listes, on peut extraire aussi une partie d'un tableau.

Exemple 26. Avec le tableau 2D `m` défini dans l'exemple précédent,
- `m[:, [0, 2]]` permet d'extraire les colonnes 1 et 3 de `m`.
- `m[-1, :2]` permet d'extraire les 2 derniers éléments de la dernière ligne de `m`.

NB : `m[i]` extrait la (i+1)-ème ligne du tableau 2D `m` tout comme `m[i ,:]`.

Exercice 16. Comment, en une seule commande, extraire les éléments d'indices $(0,0)$ et $(n-1,n-1)$ d'une matrice de taille $n \times n$ pour une faire un tableau 1D contenant ces éléments ?

Exercice 17.

1. si $x = [x_1, \ldots, x_m]$ et n est un entier strictement positif, que renvoie la fonction `tabpuissance`(x,n) suivante ?

```
import numpy as np

def tabpuissance(x,n):
    """x liste de nombres ou tableau 1D
       n entier positif
    """
    m = len(x)
    A = np.zeros((m,n+1))
    for j in range(n+1):
        A[:,j] = [val**j for val in x]
    return A
```

2. Vérifier que la fonction `tabpuissance2(x,n)` construit de façon différente le même objet que `tabpuissance(x,n)` :

```
def tabpuissance2(x,n):
    """x liste de nombres ou tableau 1D
       n entier positif
    """
    m = len(x)
    xt = np.array(x)
    A = np.zeros((m,n+1))
    for j in range(n+1):
        A[:,j] = xt ** j
    return A
```

3. Exécuter les instructions suivantes afin de comparer la vitesse d'exécution des deux fonctions :

```
import time
x = np. linspace(0, 1, 1000)
deb = time.time() ; tabpuissance(x,1000) ; print(time.time() - deb)
deb = time.time() ; tabpuissance2(x,1000) ; print(time.time() - deb)
```

5.3 Redimensionnement d'un tableau 2D

Le tableau suivant donne des exemples de fonctions permettant de construire à partir d'un tableau `m`, un nouveau tableau en lui ajoutant/supprimant des éléments.

np.append	pour ajouter une ligne ou une colonne à la fin d'un tableau 2D ou un élément à la fin d'un tableau 1D
np.insert	pour insérer des lignes ou colonnes dans un tableau 2D ou des éléments dans un tableau 1D
np.delete	pour supprimer des lignes ou colonnes dans un tableau 2D ou des éléments dans un tableau 1D

Exemple 27.

```
>>> m = np.array([[1, 2, 3],[4, 5, 6]])
>>> print(np.insert(m, 1, [[0,-1], [-2,-3]], axis=1))
[[ 1  0 -2  2  3]
 [ 4 -1 -3  5  6]]
>>> print(np.delete(m, 0, axis = 0))
[[4 5 6]]
```

5.4 Opérations sur chaque élément d'un tableau numérique

Les opérateurs `+, -, *, /, **` appliqués entre des tableaux de mêmes tailles ou entre un tableau et un scalaire applique l'opération sur chaque élément du/des tableaux.

Exemple 28.

```
>>> m = np.array([[1, 2, 3],[4, 5, 6]])
>>> print(m ** 2)
[[ 1  4  9]
 [16 25 36]]
>>> print(3 + m)
[[4 5 6]
 [7 8 9]]
```

Ces opérations sont beaucoup plus rapides que si on faisait la même chose avec les listes.

```
>>> import time
>>> a = np.arange(10000)
>>> deb = time.time(); a = a + 1; print(time.time() - deb)
4.673004150390625e-05
>>> b = range(10000)
>>> deb = time.time(); b = [i + 1 for i in b] ; print(time.time() - deb)
0.0009615421295166016
```

De même, les opérations de comparaisons entre deux tableaux de même taille effectue la comparaison élément par élément.

Exemple 29.

```
>>> np.array([1, 2, 3]) > np.array([-1, 5, 0])
array([ True, False,  True], dtype=bool)
```

NB : on peut utiliser une condition pour extraire les éléments d'un tableau ayant une certaine propriété.

Exemple 30.

```
>>> m = np.array([[1, 2, 3],[4, 5, 6]])
>>> np.extract(m >= 2, m) # ou bien de façon plus compacte m[ m >+2]
array([2, 3, 4, 5, 6])
```

La plupart des fonctions numériques programmées dans `Numpy` peuvent opérer directement à tous les éléments d'un tableau. C'est le cas par exemple de `np.cos` et `np.sin` utilisés dans l'exemple 19.

5.5 Opérations matricielles

En plus d'opérations éléments par éléments, on dispose de fonctions permettant d'effectuer du calcul matriciel. Le tableau suivant décrit quelques fonctions de base. Le module `linalg` de `scipy` contient un grand nombre de fonctions pour l'algèbre linéaire. Dans ce tableau, A et B désignent des tableaux numériques 1D ou 2D de tailles compatibles pour les opérations décrites.

Opérations matricielles	
`np.dot(A,B)` ou `A.dot(B)` (ou `A @ B` avec Python version ≥ 3.5)	produit matriciel `AB`
`np.power(A,n)`	puissance n-ème de la matrice A
`np.transpose(A)` ou `A.T`	transposée de la matrice A
`np.trace(A)`	trace de A
`scipy.linalg.norm(A)`	différentes normes (par défaut, norme de Frobénius si A est une matrice et norme 2 si A un vecteur)
`scipy.linalg.inv(A)`	inverse de A
`scipy.linalg.det(A)`	déterminant de A
`scipy.linalg.eigvals(A)`	valeurs propres de A
`scipy.linalg.eig(A)`	valeurs propres et base de vecteurs propres

5.6 Quelques fonctions opérant sur des tableaux numériques

Le tableau décrit le fonctionnement de quelques fonctions usuelles lorsqu'on les applique à un tableau 2D appelé M. On peut utiliser aussi ces fonctions avec des tableaux 1D, ou bien faire agir ces fonctions colonne par colonne ou ligne par ligne sur un tableau 2D en ajoutant l'argument optionnel `axis=0` ou `axis=1` :

`np.min`(M)	retourne la valeur minimale des coefficients de M
`np.max`(M)	retourne la valeur maximale des coefficients de M
`np.sum`(M)	retourne la somme des éléments de M
`np.cumsum`(M)	retourne un tableau 1D M contenant la somme cumulée des premiers éléments de M parcourus ligne par ligne.
`np.mean`(M)	retourne la moyenne de tous les éléments de M
`np.prod`(M)	retourne le produit des éléments de M
`np.cumprod`(M)	retourne un vecteur contenant le produit cumulé des premiers éléments de M.
`sort`(M)	tri par ordre croissant chaque ligne de M

Exemple 31.

```
>>> m=np.array([[2,-1,4],[5,-2,0]])
>>> np.sort(m)
array([[-1,  2,  4],
       [-2,  0,  5]])
>>> np.max(m, axis = 1)
array([4, 5])
>>> np.cumsum(m, axis = 0)
array([[ 2, -1,  4],
       [ 7, -3,  4]])
```

6 Représentation de points dans le plan

On décrit ici quelques fonctions du module pyplot de matplotlib que l'on importe avec la commande :

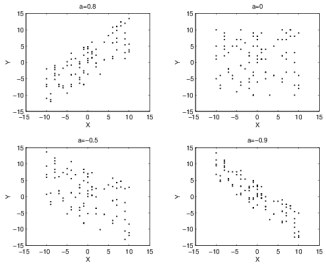
```
import matplotlib.pyplot as plt
```

Il existe plusieurs possibilités pour représenter un ensemble de points $(x(i), y(i))$, x et y etant des tableaux numériques 1D. Les plus utilisées sont énumérées ci-dessous :

`plt.plot(x,y)`	les points sont reliés par des segments de droites
`plt.step(x,y)`	tracé en escaliers : les points sont reliés par un segment vertical suivi d'un segment horizontal par défaut
`plt.stem(x,y)`	diagramme en bâtons
`plt.scatter(x,y)`	Tracé comm un nuage de points

Exemple 32. Le programme suivant trace 100 points dont les coordonnées sont choisis « aléatoirement » entre 0 et 1.

```
import matplotlib.pyplot as plt
plt.figure(1)
x = np.random.rand(1,100)
y = np.random.rand(1,100)
plt.scatter(x,y)
plt.savefig("nuage.png", bbox_inches = "tight")
```



Voici une liste de plusieurs fonctions générales permettant d'ouvrir les fenêtres qui contiennent les diagrammes, d'annoter les diagrammes...

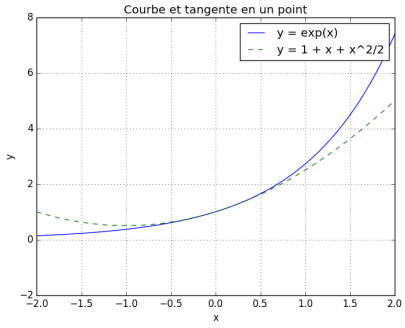
`plt.figure`	pour créer ou sélectionner une fenêtre graphique
`plt.show`	pour indiquer d'afficher à l'écran les tracés effectués
`plt.close('all')`	pour fermer toutes les fenêtes graphiques
`subplot`(n, m, p)	partage la fenêtre graphique en mxn espaces graphiques et sélectionne le p-i 'eme.
`plt.clf()`	pour effacer le contenu de la fenêtre graphique
`plt.title`	Donner un titre au diagramme
`plt.xlim` / `plt.ylim`	Définir les bornes des axes du diagramme
`plt.xlabel` / `plt.ylabel`	Annoter les axes du diagramme
`plt.legend`	Ajouter une légende (utile quand on superpose plusieurs diagrammes)
`plt.grid`	Ajouter une grille
`plt.savefig('nom.png')`	pour sauvegarder la figure en un fichier nom.png

Il est très important de prendre l'habitude d'annoter les figures en utilisant des fonctions `plt.title`, `plt.xlabel`, `plt.ylabel` et `plt.legend`.

NB : Les instructions de tracés ne sont effectuées qu'à l'appel d'une commande `plt.show` ou `plt.savefig` et si on ne précise rien, les tracés se superposent sur la même figure.

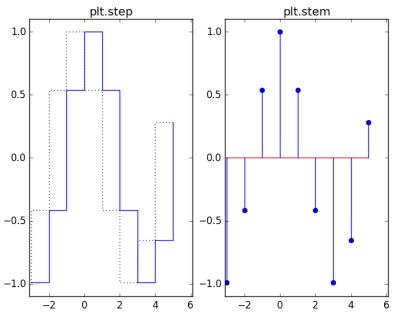
Exemple 33. Le programme suivant permet d'afficher la figure ci-contre :

```
plt.figure(2)
x = np.linspace(-2,2,1000)
y1 = np.exp(x)
y2 =  1 + x + x ** 2 / 2
plt.plot(x, y1)
plt.plot(x, y2, linestyle='--')
plt.grid()
plt.ylim(-2,8)
plt.xlabel('x')
plt.ylabel('y')
plt.legend(['y = exp(x)','y = 1 + x + x^2/2'])
plt.title('Courbe et tangente en un point')
plt.savefig("illust.png", bbox_inches = "tight")
```



Exemple 34. Exemple d'utilisation de `plt.subplot`.

```
plt.figure(3)
x = np.arange(-3,6,1)
y = np.cos(x)
bornes = [-3.1,6.1,-1.1,1.1]
plt.subplot(1,2,1)
plt.step(x, y, color = "black", linestyle = ':')
plt.step(x, y, where = "post")
plt.axis(bornes)
plt.title("plt.step")
plt.subplot(1,2,2)
plt.stem(x, y)
plt.title("plt.stem")
plt.axis(bornes)
plt.savefig("tracesubplot.png", bbox_inches = "tight")
```



A Annexe

A.1 Commandes pour obtenir de l'aide sur un objet dans l'interpréteur Python

`nom?`, `nom??` ou `help`(nom)	documentation de plus en plus détaillée sur un objet `nom`
`nom.*?` ou `dir(nom)`	liste des attributs et méthodes de l'objet `nom`
`numpy.lookfor("`*motcle*`")`	(avec le module Numpy) recherche tous les objets dont la documentation contient la chaîne de caractères *motcle*

NB : Lorsqu'on tape sur la touche de tabulation après avoir écrit le début d'une commande, une liste de complétions possibles apparait.

A.2 Liste des mots-réservés

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

A.3 Conseil pour la présentation du code

- Utiliser une indentation formée de 4 espaces pour définir un bloc d'instructions
- écrire une instruction par ligne
- Mettre un espace avant et après les opérateurs et le signe '=' d'affection
- Mettre un espace après chaque virgule séparant les arguments d'une fonction mais pas d'espaces autour des parenthèses
- Documenter les fonctions
- Suivant l'encodage des caractères d'un fichier `.py`, mettre sur la première ligne
  `# -*- coding: utf-8 -*-` ou `# -*- coding: latin-1 -*-`
  Ecrire ensuite les lignes d'instruction permettant importer les modules nécessaires au programme. Définir ensuite les fonctions que le programme utilisera et terminer par les lignes d'instructions qui appellent ces fonctions.

NB : si on veut pouvoir exécuter des « paquets » d'instructions les uns après les autres dans Spyder, il suffit de séparer chaque paquet d'instructions par une ligne commençant par `#%%` (on constitue ainsi ce que Spyder désigne par le terme de cellule).

A.4 Le type `float`

Python utilise la représentation des nombres réels en *virgule flottante* en suivant la norme IEEE-754 et travaille en double précision, ce qui signifie que les réels sont stockés en mémoire sur 64 bits : un nombre réel non nul est approché par un nombre ayant une représentation finie en base 2 que l'on écrit $(-1)^s 2^e (1 + \sum_{j=1}^{t} a_j 2^{-j})$ avec

- $s \in \{0, 1\}$ suivant le signe du réel ($s = 0$ pour un réel positif) ; il occupe 1 bit,
- $m = 1.a_1 \dots a_t$ la mantisse ; elle occupe 52 bits (le 1 initial n'est pas codé, $0.a_1 ... a_t$ correspond à la partie fractionnaire),
- e entier entre -1022 à 1023, appelé l'exposant ; $e + 1023$ dispose de 11 bits.

L'exposant $e = -1023$ est réservé pour représenter 0. L'exposant $e = 1024$ sert à représenter les nombres spéciaux $\pm$inf et NaN (not a number). Un tel codage permet de conserver des réels entre environ 2.10^{-308} et 2.10^{308} avec 16 chiffres significatifs.

Exemple 35. Le tableau présente des exemples de codage de nombres en virgule flottante normalisée :

Nombre	codage de l'exposant +1023	codage de la mantisse
2^{-1022}	000 0000 0001	que des bits 0
$(2 - 2^{-52})2^{1023}$	111 1111 1110	que des bits 1
30.0625	100 0000 0011	1110 0001 suivi de 43 zéros

Par exemple, 30.0625 s'écrit en binaire 11 110.0001. L'exposant est donc 4 et le codage en binaire de 1027 est 100 0000 0011.

NB :

- Les infinis `+inf` et `-inf` sont codés en utilisant le plus grand exposant et en mettant tous les bits de la mantisse à 0.
- Les nombres codés en utilisant le plus grand exposant sont appelés NaN (Not a Number) : ils servent à représenter le résultat d'une opération invalide (0/0 par exemple).
- Le nombre 0 est codé en mettant tous les bits de l'exposant et de la mantisse à 0. Le bit du signe donne 2 codages de 0 : +0 et -0. On a `1/-0=-inf` et `1/+0=+inf` et `+0=-0`.

Les opérations élémentaires $op \in \{+, -, \times, /\}$ entre 2 réels x et y se font alors de la façon suivante : fl(fl(x) op fl(y)).

Table des matières