Institut National Polytechnique

Félix HOUPHOUET-BOIGNY

Cours de programmation

Initiation à Python 3

Initiation

python™

CPGE / P1

Auteur

KPO Loua

Enseignants-Chercheurs en informatique à l'INP-HB

Département : Maths – Info

Janvier 2021

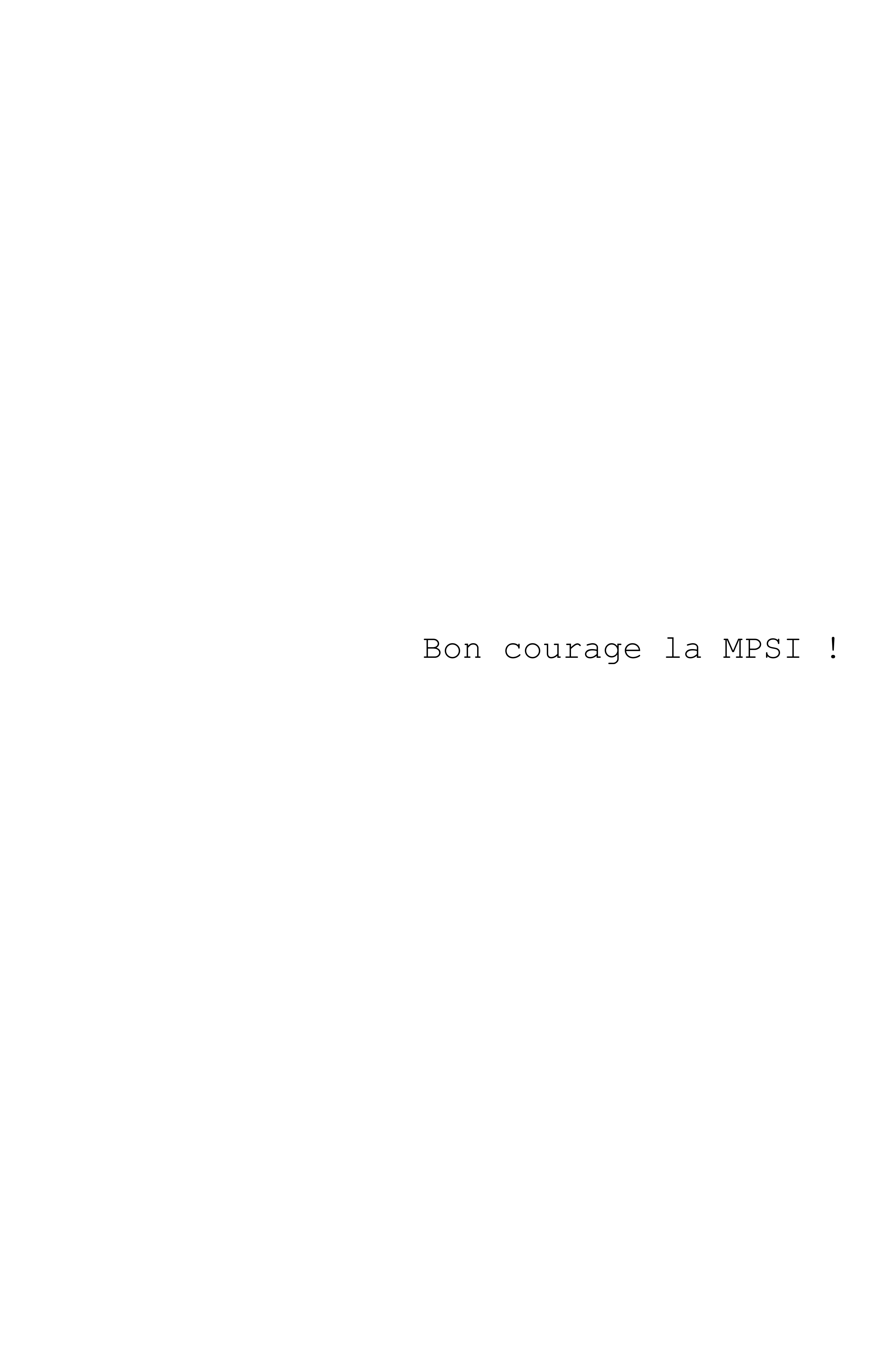

Bon courage la MPSI !

Table des matières

Savoir-faire : 4

CHAPITRE 6 : Introduction à python 5

Introduction....6

6.1. Principales caractéristiques du langage Python....6

6.1.1. Genèse....6

6.1.2. Evolution....6

6.1.3. Les avantages de python....6

6.2. Environnements matériels et logiciel....7

6.2.1. Environnement matériel....7

6.2.2. Environnement logiciel....7

6.3. Fonctionnement de python....9

6.4. TP 2 : installation winPython....10

CHAPITRE 7 : Base du langage Python....13

7.1 Calculs et opérations....14

7.2. Les affectations....14

7.3. Les entrées et les sorties standards....15

7.3.1. Les entrées....15

7.3.2. Les sorties....15

7.4. Notion de variables....16

7.4.1. Définition....16

7.4.2. Déclaration....16

7.4.3. Nommer une variable....16

7.4.4. Les types de variables....17

7.5. Types de base....17

7.5.1. Types immuables (ou immutable)....17

7.5.2. Type « rien » ou None....18

7.5.3. Les Entiers et les réels....18

7.5.4. Les complexes....19

7.5.5. Les booléens....19

7.5.6. Les chaînes de caractère....19

7.5.6. Les tuples....24

7.5.7. Les complexes....26

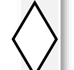
7.5.8. Les bytes....26

7.6. Notion de liste en Python 27
7.6.1. Les listes 27
7.6.2. Les dictionnaires 32
7.6.3. Les fonctions des dictionnaires 34
7.7. Structures conditionnelles 35
7.7.1. La condition if (Si …. Alors) 35
7.7.2. La condition if … else (Si … Alors ….. Sinon) 36
7.7.3. Les conditions if … else … elif (Si … Alors … Sinon Si) 36
7.7.4. Comment fonctionne les structures conditionnelles ? 36
7.8. Structures de boucle 38
7.8.1. La boucle while (TantQue … Faire) 38
7.8.2. La boucle for (Pour … Faire) 39
7.8.3. Structure d'un programme en python 40
7.9. Fonctions et procédure 40
7.9.1. Les paramètres 41
7.9.2. Variable local, variable global : 42
7.9.3. Exemple de fonction 43
7.9.4. Fiche TD 4 : les bases du langage Python 44
7.9.5. Fiche TD 5 : Fonctions en Python, notion de paramètre 44
7.9.6. Fiche TP 3 : Base du langage, Fonctions 44
7.10. Compréhension des listes ! 44
7.10.1. TD 5 : Manipulation des conteneurs 45
7.10.2. TP 3 : Manipulation des conteneurs 45
CHAPITRE 8 : Python et les calculs scientifiques 46
8.1. Les modules et packages 47
8.2. Numpy (`Numerical Python`) 48
8.2.1. Variables prédéfinies 48
8.2.2. Tableaux - numpy.array() 49
8.2.3. La fonction numpy.arange() 49
8.2.4. La fonction numpy.linspace() 50
8.2.5. Action d'une fonction mathématique sur un tableau 50
8.2.6. Calcul sur les matrices 50
8.2.7. Fiche TP 4 : Ecriture de modules, manipulation de NumPy 54
8.3. Le module matplotlib 54
8.3.1. Définition 54
8.3.2. Des détails 56

8.3.3. Quelques exemples 57
8.4. Scientific Python (Scipy) 59
8.4.1. Fonctions Spéciales 60
8.4.2. Intégration 62
8.4.3. Equations différentielles ordinaires (EDO) 63
8.4.4. Optimisation 65
CHAPITRE 9 : Approfondissements en Python 67
9.1. Les exceptions 68
9.1.1. Try except 68
9.1.2. Cibler les erreurs 68
9.1.3. Finally 69
9.2. Edition d'un fichier 69
9.2.1. Editer un fichier 70
9.2.2. La fonction open 70
9.2.3. Les types d'ouverture 70
9.2.4. Fermeture d'un fichier 70
9.2.5. Lire le contenu d'un fichier 71
9.2.6. Ecrire dans un fichier 71
9.2.7. Le mot clé with 71
9.3. Application 71
9.3.1. Concernant le répertoire 71
9.3.2. Application 1 72
9.3.3. Application 2 72
9.3.4. Les exceptons et les fichiers 73
9.3.5. Application 3 73
9.3.6. TDn°7 – Gestion des exceptions, manipulation des fichiers textes 74
9.3.7. TPn°8 – Gestion des exceptions, manipulation des fichiers textes 74

Savoir-faire :

- Choisir un type de données en fonction d'un problème à résoudre,
- Traduire un algorithme dans un langage de programmation
- Concevoir l'en-tête (spécification des paramètres éventuels) d'une fonction, puis la fonction elle-même (utilisation des paramètres éventuels),
- Rechercher une information au sein d'une documentation en ligne, analyser des exemples fournis dans cette documentation,
- Réaliser un programme complet structuré allant de la prise en compte de données expérimentales à la mise en forme des résultats permettant de résoudre un problème scientifique donné,
- Utiliser les bibliothèques de calcul standard pour résoudre un problème scientifique mis en équation lors des enseignements de chimie, physique, mathématiques, sciences industrielles et de l'ingénieur
- Étudier l'effet d'une variation des paramètres ou de méthodes sur le temps de calcul, sur la précision des résultats, sur la forme des solutions pour des programmes d'ingénierie numérique choisis,
- Utiliser les bibliothèques standard pour afficher les résultats sous forme graphique,
- Tenir compte des aspects pratiques comme l'impact des erreurs d'arrondi sur les résultats
- Fournir le résultat à l'utilisateur par diverses voies (sortie standard, fichiers),



CHAPITRE 6 : Introduction à Python

Introduction

Programmer consiste à demander à un ordinateur d'effectuer des taches, appelée aussi instructions ou commandes. Or, l'être humain et l'ordinateur ne parlent pas le même langage : l'ordinateur ne sait reconnaitre qu'une succession de 0 et de 1 (le langage machine), et il est bien entendu illusoire de penser qu'un homme puisse apprendre un tel langage. Pour permettre a l'un et à l'autre de communiquer, on fait appel à Un langage de programmation : basic, pascal, fortran, C++, caml... ou encore python.

1. La rédaction des programmes dans le langage choisi se fait alors avec un logiciel appelé interface graphique
2. Lors de l'exécution des commandes ou du programme, un autre logiciel appelé compilateur, interpréteur ou noyau, traduit le langage informatique en langage machine et effectue les taches demandées.

Python est un langage parmi beaucoup d'autres, permettant le développement de programmes informatiques.

6.1. Principales caractéristiques du langage Python

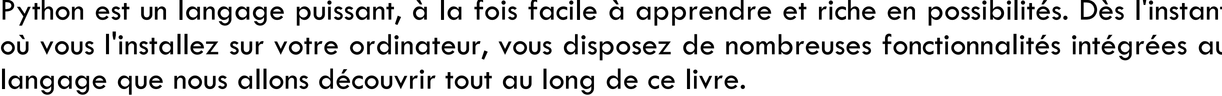
6.1.1. Genèse

Python est un langage de programmation, dont la première version est sortie en 1991. Créé par Guido van Rossum, il a voyagé du Macintosh de son créateur, qui travaillait à cette époque au Centrum voor Wiskunde en Informatica aux Pays-Bas, jusqu'à se voir associer une organisation à but non lucratif particulièrement dévouée, la Python Software Foundation, créée en 2001. Ce langage a été baptisé ainsi en hommage à la troupe de comiques les « Monty Python ».

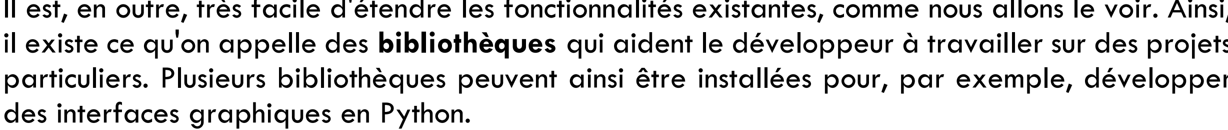
6.1.2. Evolution

Lors de la création de la Python Software Foundation, en 2001, et durant les années qui ont suivi, le langage Python est passé par une suite de versions que l'on a englobées dans l'appellation Python 2.x (2.3, 2.5, 2.6…). Depuis le 13 février 2009, la version 3.0.1 est disponible. Cette version casse la compatibilité ascendante qui prévalait lors des dernières versions. A présent python existe sous la version 3.x

6.1.3. Les avantages de python



Python est un langage puissant, à la fois facile à apprendre et riche en possibilités. Dès l'instant où vous l'installez sur votre ordinateur, vous disposez de nombreuses fonctionnalités intégrées au langage que nous allons découvrir tout au long de ce livre.



Il est, en outre, très facile d'étendre les fonctionnalités existantes, comme nous allons le voir. Ainsi, il existe ce qu'on appelle des **bibliothèques** qui aident le développeur à travailler sur des projets particuliers. Plusieurs bibliothèques peuvent ainsi être installées pour, par exemple, développer des interfaces graphiques en Python.

Concrètement, voilà ce qu'on peut faire avec Python :

- de petits programmes très simples, appelés scripts, chargés d'une mission très précise sur votre ordinateur ;
- des programmes complets, comme des jeux, des suites bureautiques, des logiciels multimédias, des clients de messagerie…

- des projets très complexes, comme des progiciels (ensemble de plusieurs logiciels pouvant fonctionner ensemble, principalement utilisés dans le monde professionnel).

Voici quelques-unes des fonctionnalités offertes par Python et ses bibliothèques :

- créer des interfaces graphiques ;
- faire circuler des informations au travers d'un réseau ;
- dialoguer d'une façon avancée avec votre système d'exploitation ;
- … et j'en passe…

Bien entendu, vous n'allez pas apprendre à faire tout cela en quelques minutes. Mais ce cours vous donnera des bases suffisamment larges pour développer des projets qui pourront devenir, par la suite, assez importants.

6.2. Environnements matériel et logiciel

6.2.1. Environnement matériel

Pour exécuter un programme python sur un ordinateur, il (l'ordinateur) doit avoir les caractéristiques minima suivant :

- Un dual core ;
- Un disque dure d'au moins 10 Gio
- Un processeur de 1.0 Ghz

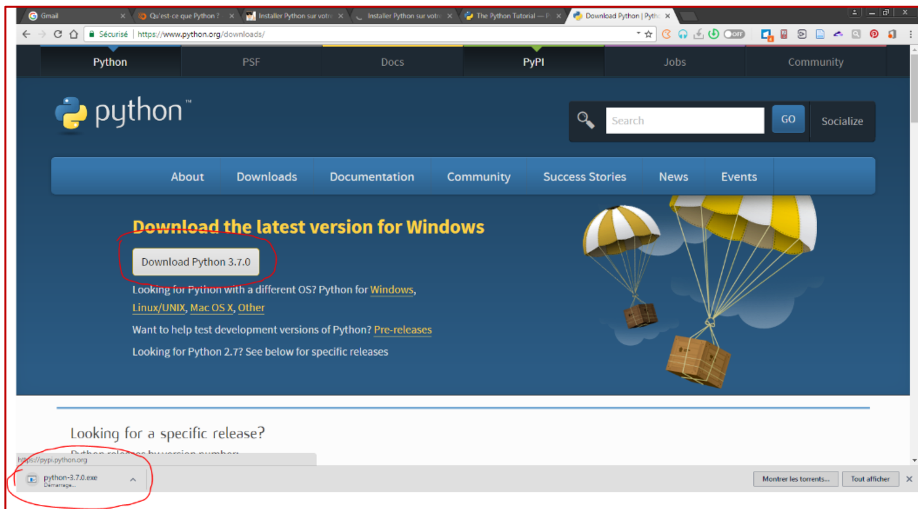
La performance des ordinateurs de nos jours est très favorable pour l'exécution des petits programme écrit en python. Certain programme aura besoin d'une machine très particulière pour son exécution. C'est l'exemple des programmes en big data où l'ordinateur doit avoir au moins 7 cœurs pour le bon fonctionnement du programme.

6.2.2. Environnement logiciel

Python est présent sur les systèmes d'exploitation Windows, Linux, Mac OS …

Environnement de développement intégré (IDE)

- *Pour installer python, rendez-vous sur le site suivant :* https://www.python.org/ftp/python/3.7.0/python-3.7.0.exe



- *Lancer l'exécutable ainsi téléchargé et suivez les instructions d'installation.*

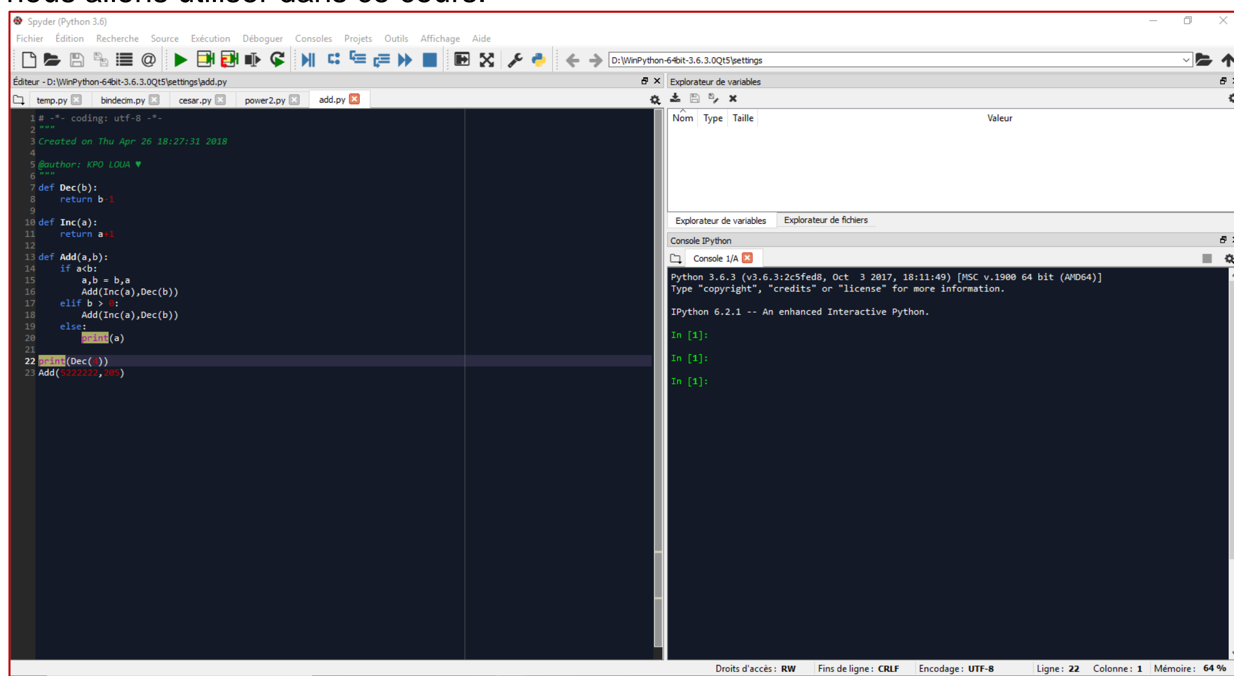
Un environnement de développement intégré est un outil qui va vous faciliter la programmation dans un langage de programmation. En python il en existe plusieurs dont nous allons présenter quelques-uns.

Sublime Text

Sublime text possède tout une panoplie de plugins dont vous serez vite accroc ! Sa version de base est gratuite, une petite alerte vous demandera de temps en temps si vous voulez acheter une licence pour soutenir le projet mais rien ne vous oblige à le faire.

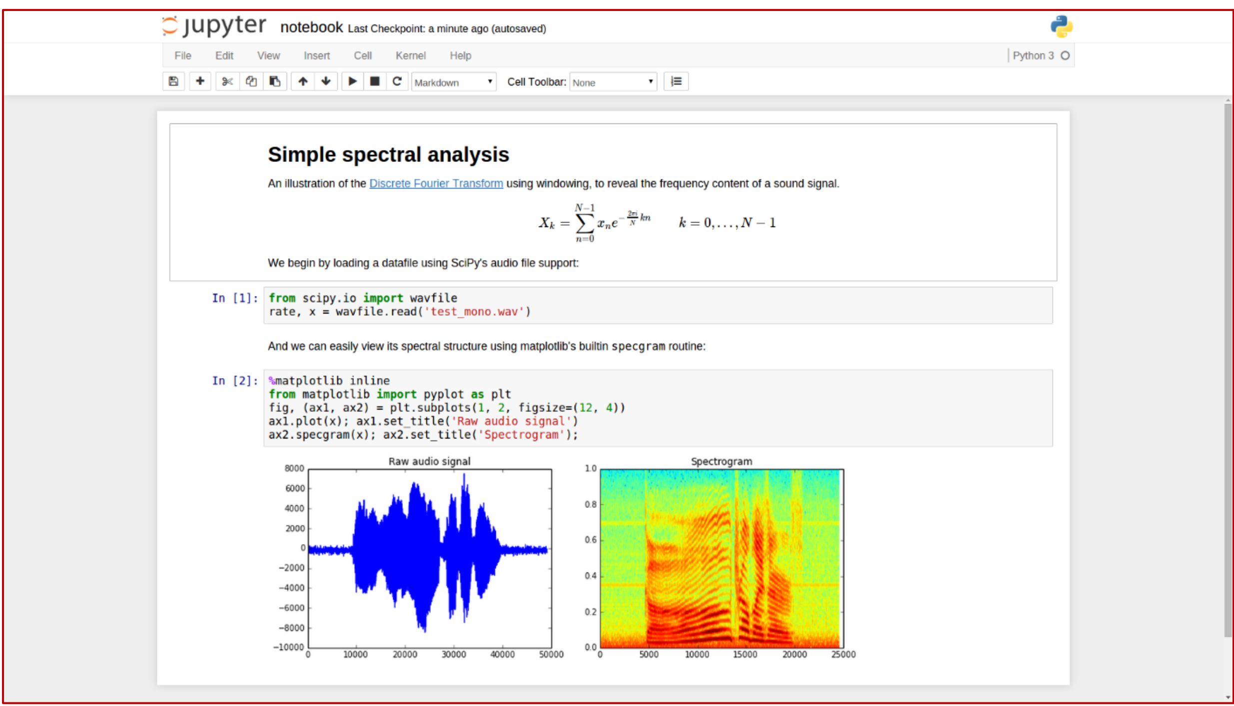
Spyder

Créé et développé par Pierre Raybaut en 2008, Spyder est maintenu, depuis 2012, par une communauté de développeurs qui ont pour point commun d'appartenir à la communauté Python scientifique. En comparaison avec d'autres IDE pour le développement scientifique, Spyder a un ensemble unique de fonctionnalités - multiplateforme, open-source, écrit en Python et disponible sous une licence non-copyleft. Spyder est extensible avec des plugins, comprend le support d'outils interactifs pour l'inspection des données et incorpore des instruments d'assurance de la qualité et d'introspection spécifiques au code Python, tels que Pyflakes, Pylint et Rope. C'est celui que nous allons utiliser dans ce cours.



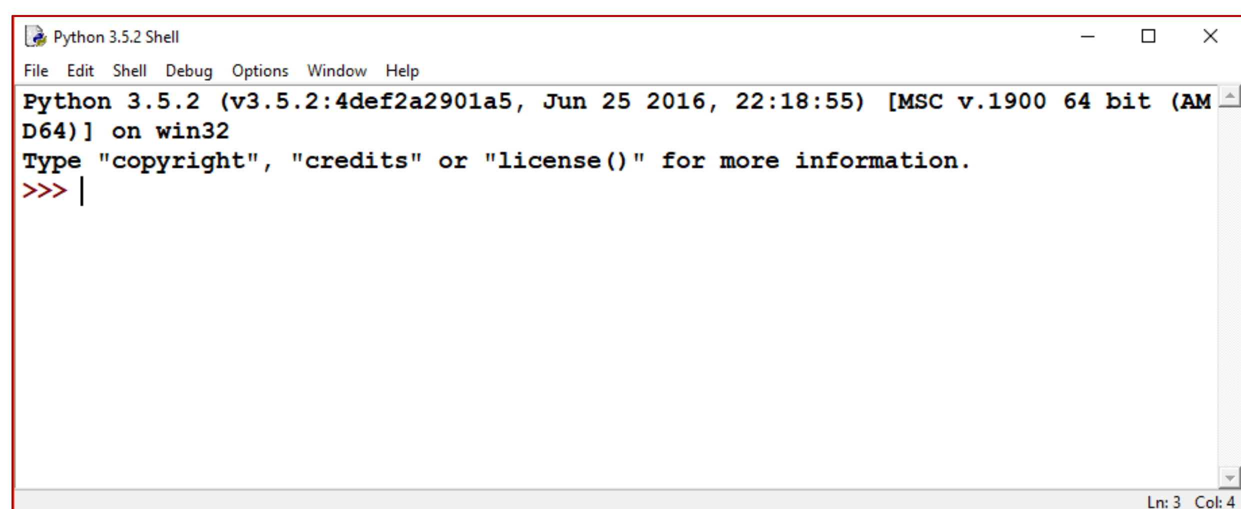
Jupyter notebook

Jupyter est une application web utilisée pour programmer dans plus de 40 langages de programmation, dont Julia, Python, R, Ruby ou encore Scala2. Jupyter est une évolution du projet IPython. Jupyter permet de réaliser des calepins ou notebooks, c'est-à-dire des programmes contenant à la fois du texte en markdown et du code en Julia, Python, R... Ces notebooks sont utilisés en science des données pour explorer et analyser des données.



La console de python

Au démarrage de Python, la console affiche la version Python qui est utilisée ainsi que les différentes bibliothèques automatiquement importées. Le curseur qui clignote après le " >>>" indique l'endroit où sont entrées les commandes.



6.3. Fonctionnement de python

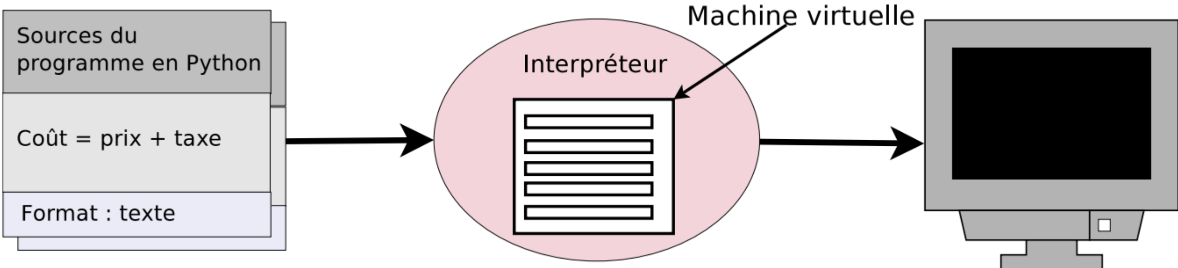
Eh oui, vous allez devoir patienter encore un peu car il me reste deux ou trois choses à vous expliquer, et je suis persuadé qu'il est important de connaître un minimum ces détails qui peuvent sembler peu pratiques de prime abord.

Python est un langage de programmation interprété, c'est-à-dire que les instructions que vous lui envoyez sont « transcrites » en langage machine au fur et à mesure de leur lecture. D'autres

langages (comme le C / C++) sont appelés « langages compilés » car, avant de pouvoir les exécuter, un logiciel spécialisé se charge de transformer le code du programme en langage machine. On appelle cette étape la « compilation ». À chaque modification du code, il faut rappeler une étape de compilation.

Les avantages d'un langage interprété sont la simplicité (on ne passe pas par une étape de compilation avant d'exécuter son programme) et la portabilité (un langage tel que Python est censé fonctionner aussi bien sous Windows que sous Linux ou Mac OS, et on ne devrait avoir à effectuer aucun changement dans le code pour le passer d'un système à l'autre). Cela ne veut pas dire que les langages compilés ne sont pas portables, loin de là ! Mais on doit utiliser des compilateurs différents et, d'un système à l'autre, certaines instructions ne sont pas compatibles, voire se comportent différemment.

En contrepartie, un langage compilé se révélera bien plus rapide qu'un langage interprété (la traduction à la volée de votre programme ralentit l'exécution), bien que cette différence tende à se faire de moins en moins sentir au fil des améliorations. De plus, il faudra installer Python sur le système d'exploitation que vous utilisez pour que l'ordinateur puisse comprendre votre code.

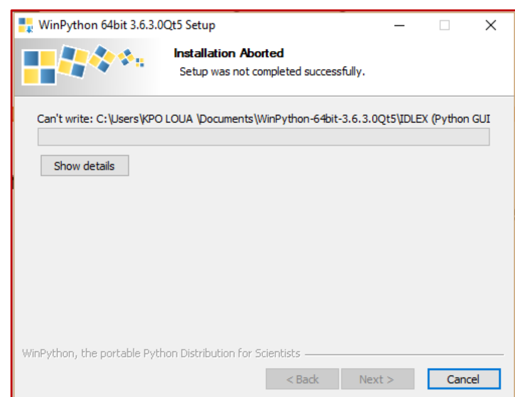


6.4. TP 2 : installation winPython

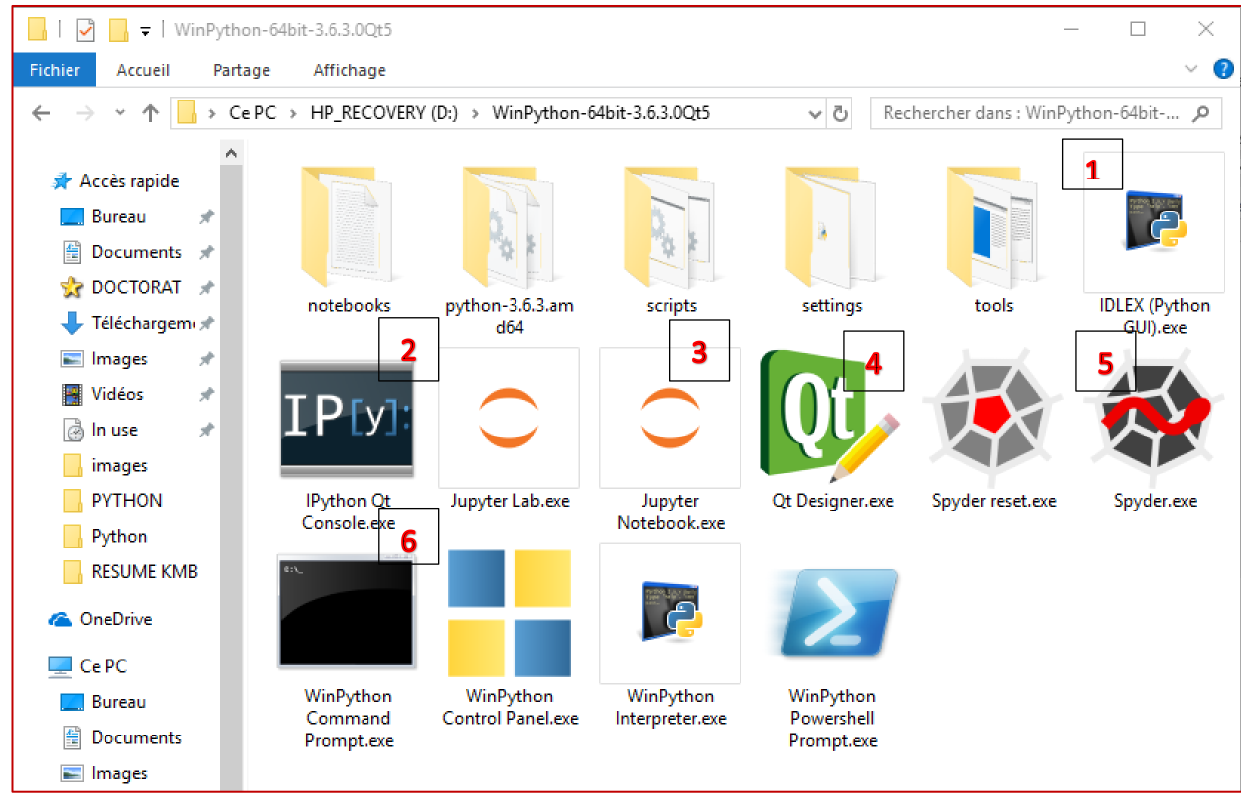
Rassurez-vous que vous n'ayez pas le logiciel smartdev sur votre ordinateur, dans le cas contraire désinstallez-le rapidement de l'ordinateur et redémarrez la machine.

Téléchargement : https://sourceforge.net/projects/winpython/postdownload

- Selon le codage de votre système, extraire le contenu du compressé WinPython64.exe ou WinPython32.exe que votre enseignant vous a remis. L'extraction est très simple, il faut simplement exécuter le fichier. L'extraction dure environ 30 minutes, et oui il faudrait être très patient.

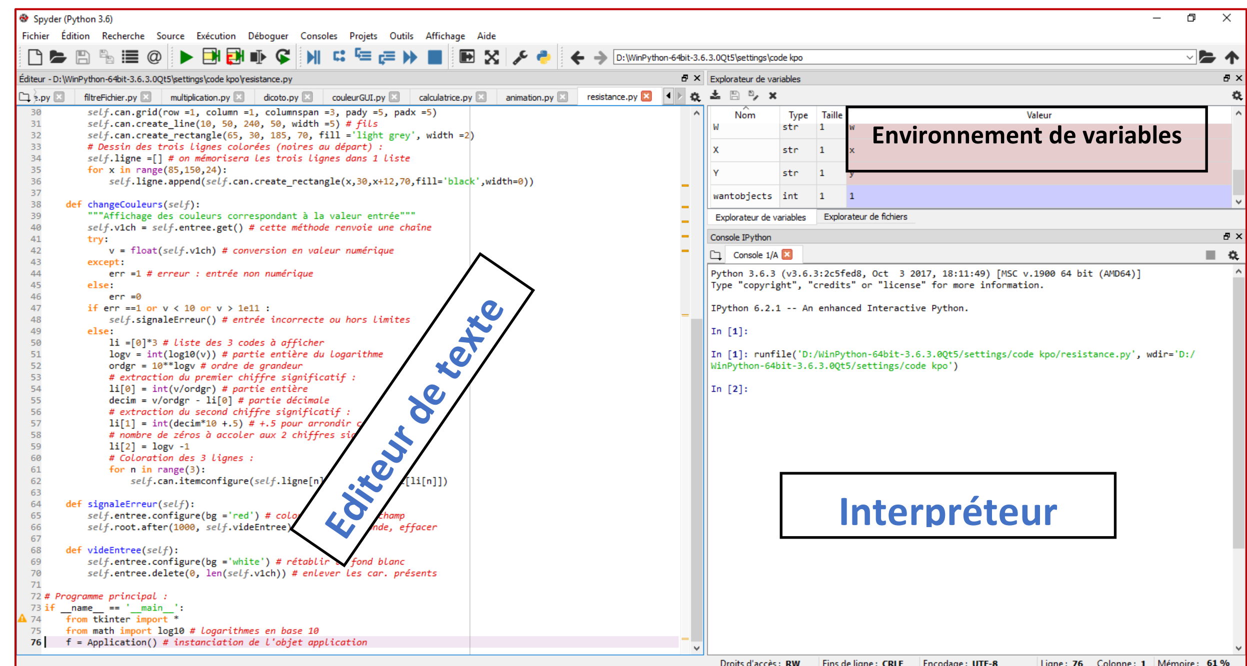


- Après extraction vous obtenez un dossier du nom de ***WinPython-64bit-3.6.3.0Qt5*** Ouvrez-le.



Ce dossier contient plusieurs éditeurs de script python. Je les ai numérotés de 1à 6.
Dans ce cours, nous allons utiliser deux éditeurs de texte à savoir Spyder (4) et Jupyter (3).

Présentation de Spyder



Spyder est constitué de 4 fenêtres :

1. L'éditeur qui permet de rédiger des programmes.
2. La console qui permet de tester des commandes (onglet : Console IPython) et qui renvoie les résultats des programmes rédiges dans l'éditeur (autres onglets)

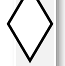
3. L'explorateur avec pour onglets :
   (a) L'inspecteur d'objets qui donne des informations sur l'utilisation des fonctions activées
   (b) L'explorateur de variables qui donne la liste et les valeurs de toutes les variables qui ont été crée
   (c) L'explorateur de fichiers qui donne accès au disque dur
4. L'environnement des variables : répertorie toutes les variables du code python tout en donnant les informations telles que le type de la variable et sa valeur.

L'´éditeur

Nous utiliserons la fenêtre "éditeur" lors de la création de programmes... Cet environnement facilite la rédaction d'un programme en :

1. Imposant des indentations lorsque c'est nécessaire ;
2. Mettant en couleur les fonctions, les mots clés et les chaines de caractères (entre guillemets)
3. proposant une aide pour la gestion des parenthèses
4. indiquant par un panneau ⚠ les erreurs éventuelles de syntaxe ;
5. affichant une aide contextuelle pour l'utilisation des fonctions Python.

Une fois le programme rédige, on l'enregistre dans un fichier *.py puis on peut l'exécuter à l'aide de la touche **F5**. Les résultats s'affichent alors dans la console interactive ou une console spécifique d´Eddie au fichier contenant le programme.



Remarque : *Dans l'´éditeur, il est possible d'ouvrir plusieurs fichiers correspondants chacun à des programmes différents*

CHAPITRE 7 : Bases du langage Python

7.1 Calculs et opérations

Les opérateurs en python peuvent se regrouper dans le tableau suivant :

	Nom	Symbole
Opérations mathématique	Addition	+
	Multiplication	*
	Exponentielle	**
	Division	/
	Division entière	//
	Modulo	%
Logique	Comparaison	< >
	Différent	!=
	Test égalité	==
	ET – OU	and - or
Affectation	Affectation	=

- Exemple de calcul

```
Type "copyright", "credits" or "license()" for more information.
>>> 7/5
1.4
>>> 7%5
2
>>> 7*5
35
>>> 7**5
16807
>>> |
```

7.2. Les affectations

Python est un langage typer dynamiquement. C'est-à-dire que les variables prennent le type de la variable que l'on leur affecte. En python, il n'y pas de déclaration de variables.

```
>>> var = 5
>>> var
5
>>> var = "Bonjour la MPSI et la Bio 1"
>>> var
'Bonjour la MPSI et la Bio 1'
>>> var = 7.6
>>> var
7.6
>>> |
```

L'une des choses qui fait la force de python est l'affectation multiple :

```
>>> Longueur, largeur, hauteur = 12, 5, 25.5
>>> largeur
5
>>> x = y = 15
>>> x
15
>>> y
15
>>> |
```

7.3. Les entrées et les sorties standards

7.3.1. Les entrées

Une entrée permet de saisir et stocker une valeur dans une variable par un utilisateur ! En LDA nous l'avons noté **LIRE()**. En programmation Python on écrira **input().**

```
>>> # Lire(a)se traduit par
>>> a = input()
```

Toutes les entrées en python sont de type string c'est-à-dire des chaines de caractère.

```
>>> # Lire(a)se traduit par
>>> a = input()
15
>>> a
'15'
>>> |
```

La valeur de a vaut '15 ' entre quotte donc chaine de caractère. Si l'on veut utiliser la valeur de a comme un entier (integer), il va falloir convertir sa valeur en int.

```
>>> a
'15'
>>> a = int(a)
>>> a
15
>>> |
```

Et c'est fait !

7.3.2. Les sorties

Les sorties (**ECRIRE()** en LDA) en python se font avec la fonction **print()**. Cette fonction permet d'afficher un texte à l'écran.

```
>>> print("Entrez votre nom svp !")
Entrez votre nom svp !
>>> age = 18
>>> print("votre âge est : ",age)
votre âge est :  18
>>> |
```

J'espère que vous comprenez maintenant pourquoi il fallait bien maitriser le LDA. Il n'est pas encore tard pour y retourner très chèr(e) étudiant(e).

7.4. Notion de variables

7.4.1. Définition

Une variable est une sorte de boite virtuelle dans laquelle on peut mettre une (ou plusieurs) donnée(s). L'idée est de stocker temporairement une donnée pour travailler avec. Pour votre machine une variable est une adresse qui indique l'emplacement de la mémoire vive où sont stockées les informations que nous avons liées avec. Affectons une valeur à la variable *age* que nous allons ensuite afficher :

7.4.2. Déclaration

En python, il n'est pas nécessaire d'écrire VAR avant la déclaration d'une variable. Cette déclaration peut se faire à n'importe quel niveau dans le programme, mais faite attention ! une variable qui n'est pas déclarée ne peut pas être utilisé.

- **Avec des INT**

```
>>> age = 18
>>> print(age)
18
>>> # Dans 10 ans tu auras
>>> age = age + 10
>>> print(age)
28
>>> # Dans 50 ans l'âge sera
>>> age += 40
>>> print(age)
68
>>> |
```

- **Avec les STR**

```
>>> SMS = "Quand viendras-Tu ?"
>>> print(SMS)
Quand viendras-Tu ?
>>> SMS = SMS + " Nous tendons vers la fin"
>>> print(SMS) # c'est la concaténation des str
Quand viendras-Tu ? Nous tendons vers la fin
>>> |
```

7.4.3. Nommer une variable

Vous ne pouvez pas nommer les variables comme bon vous semble, puisqu'il existe déjà des mots utilisés par Python. Voici la liste des mots réservé par python :

Print	in	and	or	if	del
For	is	raise	assert	elif	from
lambda	return	break	else	global	not
try	class	except	while	continue	exec
import	pass	yield	def	finally	with

Pourquoi ces mots sont-ils réservés ? Parce qu'ils servent à faire autre chose. Nous verrons cela plus en détail dans les prochains chapitres. Pour nommer une variable vous devez obligatoirement utiliser les lettres de l'alphabet, les chiffres et le caractère "_" et "-". N'utilisez pas les accents, ni les signe de ponctuation ou le signe @. De plus les chiffres ne doivent jamais se trouver en première position dans votre variable :

```
>>> 5var = 15
SyntaxError: invalid syntax
>>> |
```

Voilà que l'interprète n'est pas du tout ravi ! Comme vous le remarquez, python refuse ce genre de syntaxe, mais il acceptera var5 = 1

7.4.4. Les types de variables

En python une variable est typée, c'est à dire qu'en plus d'une valeur, une variable possède une sorte d'étiquette qui indique ce que contient cette boite virtuelle. Voici une liste de type de variable : Les **integer (int)** ou nombres entiers : comme son nom l'indique un entier est un chiffre sans décimales. Les **float** ou nombre à virgules : exemple : 1.5 Les **strings (str)** ou chaine de caractères : pour faire simple tout ce qui n'est pas chiffre.

Il en existe plein d'autres mais il est peut-être encore un peu trop tôt pour vous en parler. Pour connaitre le type d'une variable, vous pouvez utiliser la fonction "type()"

```
>>> varX = 8
>>> type(varX)
<class 'int'>
>>> varX = 8.0
>>> type(varX)
<class 'float'>
>>> varX = "Je programme en python"
>>> type(varX)
<class 'str'>
>>> |
```

7.5. Types de base

7.5.1. Types immuables (ou immutable)

- Définition : type immuable (ou immutable)

Une variable de type immuable ne peut être modifiée. Une opération sur une variable de ce type entraîne nécessairement la création d'une autre variable du même type, même si cette dernière est temporaire.

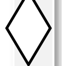
Autrement dit, la simple instruction x+=3 qui consiste à ajouter à la variable x la valeur 3 crée une seconde variable dont la valeur est celle de x augmentée de 3 puis à en recopier le contenu dans celui de la variable x. Les nombres sont des types immuables tout comme les chaînes de caractères et les tuple qui sont des tableaux d'objets. Il n'est pas possible de modifier une variable de ce type, il faut en recréer une autre du même type qui intègrera la modification.

7.5.2. Type « rien » ou None

Python propose un type **None** pour signifier qu'une variable ne contient rien. La variable est de type None et est égale à None.

```
>>> s = None
>>> print(s)
None
>>> type(s)
<class 'NoneType'>
>>> |
```

Certaines fonctions utilisent cette convention lorsqu'il leur est impossible de retourner un résultat. Ce n'est pas la seule option pour gérer cette impossibilité : il est possible de générer une exception, de retourner une valeur par défaut ou encore de retourner None. Il n'y a pas de choix meilleur, il suffit juste de préciser la convention choisie. Les fonctions sont définies au paragraphe Fonctions, plus simplement, ce sont des mini programmes : elles permettent de découper un programme long en tâches plus petites. On les distingue des variables car leur nom est suivi d'une liste de constantes ou variables comprises entre parenthèses et séparées par une virgule.



7.5.3. Les Entiers et les réels

Il existe deux types de nombres en python, les nombres réels *float* et les nombres entiers int. L'instruction x=3 crée une variable de type int initialisée à 3 tandis que y=3.0 crée une variable de type float initialisée à 3.0. Le programme suivant permet de vérifier cela en affichant pour les variables x et y, leurs valeurs et leurs types respectifs grâce à la fonction type

```
>>> x = 3
>>> y = 3.0
>>> print("x = ",x, type(x))
x =  3 <class 'int'>
>>> print("y = ",y, type(y))
y =  3.0 <class 'float'>
>>> |
```

Les fonctions int et float permettent de convertir un nombre quelconque ou une chaîne de caractères respectivement en un entier (arrondi) et en un nombre réel.

```
>>> x = int(3.5)
>>> y = float(3)
>>> z = int("3")
>>> print("x:", type(x), " y:", type(y), " z:", type(z))
x: <class 'int'>  y: <class 'float'>  z: <class 'int'>
>>> |
```

Il peut arriver que la conversion en un nombre entier ne soit pas directe. Dans l'exemple qui suit, on cherche à convertir une chaîne de caractères (voir Chaîne de de caractère) en entier mais cette chaîne représente un réel. Il faut d'abord la convertir en réel puis en entier, c'est à ce moment que l'arrondi sera effectué.

```
>>> i = int ("3.5")
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    i = int ("3.5")
ValueError: invalid literal for int() with base 10: '3.5'
>>> i = int (float ("3.5"))
>>> |
```

7.5.4. Les complexes

Le nombre imaginaire en python est noté « j ». En mathématique soit z = 4 + 5i ; en python on écrira Z = 4 +5j.

```
>>> z = 4+5j
>>> print(z)
(4+5j)
>>> type(z)
<class 'complex'>
>>> |
```

7.5.5. Les booléens

Les booléens sont le résultat d'opérations logiques et ont deux valeurs possibles : True ou False. Voici la liste des opérateurs qui s'appliquent aux booléens : and, or et not

```
>>> x = 4 < 10
>>> x
True
>>> y = not(x)
>>> y
False
>>> z = x and y
>>> z
False
>>> |
```

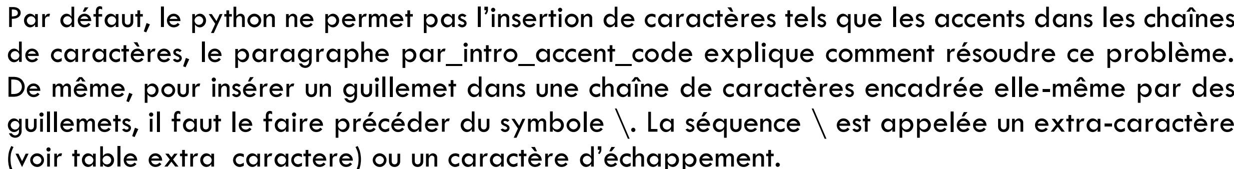
7.5.6. Les chaînes de caractères

Définition : chaîne de caractères

Le terme « chaîne de caractères » ou string en anglais signifie une suite finie de caractères, autrement dit, du texte. Ce texte est compris entre deux guillemets ou deux apostrophes, ces deux symboles sont interchangeables. Le type python est str. L'exemple suivant montre comment créer une chaîne de caractères. Il ne faut pas confondre la partie entre guillemets ou apostrophes, qui est une constante, de la variable qui la contient.

```
>>> t = "string = texte"
>>> print(type(t),t)
<class 'str'> string = texte
>>> t = 'string = texte, initialisation avec apostrophes'
>>> print(type(t),t)
<class 'str'> string = texte, initialisation avec apostrophes
>>> t = "morceau 1"\
    "morceau 2" # second moreceau ajouté au premier par l'ajout du symbole\
>>> #il ne doit rien y avoir après le symbole\,
>>> # pas d'espace ni de commentaire
>>> print(t)
morceau 1morceau 2
>>> t = """Première ligne
seconde ligne""" #chaîne de caractères qui s'étend sur deux lignes
>>> print(t)
Première ligne
```

La troisième chaîne de caractères créée lors de ce programme s'étend sur deux lignes. Il est parfois plus commode d'écrire du texte sur deux lignes plutôt que de le laisser cacher par les limites de fenêtres d'affichage. Python offre la possibilité de couper le texte en deux chaînes de caractères recollées à l'aide du symbole \ à condition que ce symbole soit le dernier de la ligne sur laquelle il apparaît. De même, lorsque le texte contient plusieurs lignes, il suffit de les encadrer entre deux symboles """ ou ''' pour que l'interpréteur python considère l'ensemble comme une chaîne de caractères et non comme une série d'instructions.



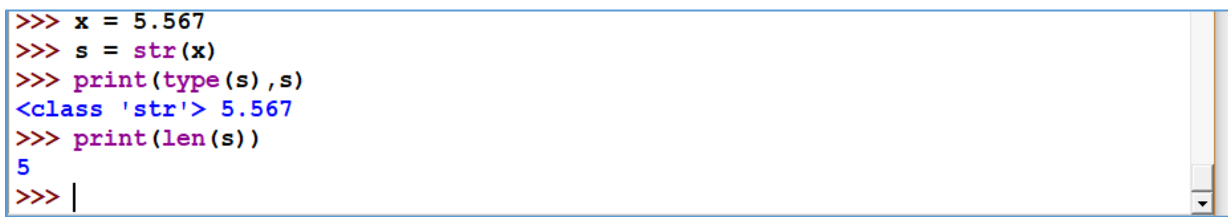
Par défaut, le python ne permet pas l'insertion de caractères tels que les accents dans les chaînes de caractères, le paragraphe par_intro_accent_code explique comment résoudre ce problème. De même, pour insérer un guillemet dans une chaîne de caractères encadrée elle-même par des guillemets, il faut le faire précéder du symbole \. La séquence \ est appelée un extra-caractère (voir table extra_caractere) ou un caractère d'échappement.

"	guillemet
'	apostrophe
\n	passage à la ligne
\\	insertion du symbole \
\%	pourcentage, ce symbole est aussi un caractère spécial
\t	tabulation
\r	retour à la ligne, peu usité, il a surtout son importance lorsqu'on passe d'un système Windows à Linux car Windows l'ajoute automatiquement à tous ses fichiers textes

Manipulation d'une chaîne de caractères

Une chaîne de caractères est semblable à un tableau et certains opérateurs qui s'appliquent aux tableaux s'appliquent également aux chaînes de caractères. Ceux-ci sont regroupés dans la table operation_string. La fonction str permet de convertir un nombre, un

tableau, un objet (voir chapitre Classes) en chaîne de caractères afin de pouvoir l'afficher. La fonction len retourne la longueur de la chaîne de caractères.



opérateur	signification	exemple
+	concaténation de chaînes de caractères	t = "abc" + "def"
+=	concaténation puis affectation	t += "abc"
in, not in	une chaîne en contient-elle une autre ?	"ed" in "med"
*	répétition d'une chaîne de caractères	t = "abc" * 4
[n]	obtention du enième caractère, le premier caractère a pour indice 0	t = "abc";print(t[0]) # donne a
[i:j]	obtention des caractères compris entre les indices iet j-1 inclus, le premier caractère a pour indice 0	t = "abc"; print(t[0:2]) # donne ab

Il existe d'autres fonctions qui permettent de manipuler les chaînes de caractères.

```
res = s.fonction (...)
```

Où s est une chaîne de caractères, fonction est le nom de l'opération que l'on veut appliquer à s, res est le résultat de cette manipulation.
La table string_method présente une liste non exhaustive des fonctions disponibles dont un exemple d'utilisation suit. Cette syntaxe variable.fonction(arguments) est celle des classes.

count(sub[, start[, end]])	Retourne le nombre d'occurences de la chaîne de caractères sub, les paramètres par défaut start et end permettent de réduire la recherche entre les caractères d'indice start et end exclu. Par défaut, start est nul tandis que end correspond à la fin de la chaîne de caractères.
find(sub[, start[, end]])	Recherche une chaîne de caractères sub, les paramètres par défaut start et end ont la même signification que ceux de la fonction count. Cette fonction retourne -1 si la recherche n'a pas abouti.
isalpha()	Retourne True si tous les caractères sont des lettres, Falsesinon.

isdigit()	Retourne True si tous les caractères sont des chiffres, Falsesinon.
replace(old, new[, count])	Retourne une copie de la chaîne de caractères en remplaçant toutes les occurrences de la chaîne old par new. Si le paramètre optionnel count est renseigné, alors seules les count premières occurrences seront remplacées.
split([sep [,maxsplit]])	Découpe la chaîne de caractères en se servant de la chaîne sepcomme délimiteur. Si le paramètre maxsplit est renseigné, au plus maxsplit coupures seront effectuées.
upper()	Remplace les minuscules par des majuscules.
lower()	Remplace les majuscules par des minuscules.
join (li)	li est une liste, cette fonction agglutine tous les éléments d'une liste séparés par sep dans l'expression sep.join (["un","deux"]).
startswith(prefix[, start[,end]])	Teste si la chaîne commence par prefix.
endswith(suffix[, start[,end]])	Teste si la chaîne se termine par suffix.

- **Exemple :**

```
>>> st = "langage python"
>>> st = st.upper()
>>> st
'LANGAGE PYTHON'
>>> i = st.find("PYTHON")
>>> print(i)
8
>>> st.count("PYTHON")
1
>>> st.count("PYTHON",9)
0
>>> |
```

```
>>> s = "un;deux;trois"
>>> mots = s.split(";")
>>> mots
['un', 'deux', 'trois']
>>> mots.reverse()
>>> s2 = ";".join(mots)
>>> s2
'trois;deux;un'
>>> |
```

- **Formatage d'une chaine de caractère**

Syntaxe %

Python (printf-style String Formatting) offre une manière plus concise de former une chaîne de caractères à l'aide de plusieurs types d'informations en évitant la conversion explicite de ces informations (type str) et leur concaténation. Il est particulièrement intéressant pour les nombres réels qu'il est possible d'écrire en imposant un nombre de décimales fixe. Le format est le suivant :

```
".... %c1   ....  %c2 " % (v1,v2)
```

c1 est un code choisi parmi ceux de la table format_print. Il indique le format dans lequel la variable v1 devra être transcrite. Il en est de même pour le code c2 associé à la variable v2. Les codes insérés dans la chaîne de caractères seront remplacés par les variables citées entre parenthèses après le symbole % suivant la fin de la chaîne de caractères. Il doit y avoir autant de codes que de variables, qui peuvent aussi être des constantes.
Voici concrètement l'utilisation de cette syntaxe :

```
>>> x = 5.5
>>> y = 8
>>> s = "caractères"
>>> res = "un nombre réel %f et un entier %d, une chaine de %s,\n"\
     "un réel d'abord converti en chaîne de caractères %s"%(x,y,s,str(x+4))
>>> print(res)
un nombre réel 5.500000 et un entier 8, une chaine de caractères,
un réel d'abord converti en chaîne de caractères 9.5
>>> |
```

Il existe d'autres formats regroupés dans la table format_print. L'aide reste encore le meilleur réflexe car le langage python est susceptible d'évoluer et d'ajouter de nouveaux formats.

d	entier relatif
e	nombre réel au format exponentiel
f	nombre réel au format décimal
g	nombre réel, format décimal ou exponentiel si la puissance est trop grande ou trop petite
s	chaîne de caractères

- **Méthode format**

La méthode format propose plus d'options pour formatter le texte et son usage est de plus en plus fréquent. La méthode interprète les accolades {} comme des codes qu'elle remplace avec les valeurs passées en argument. Le type n'importe plus. Quelques exemples :

```
>>> print("{0},{1},{2}".format('a','b','c'))#Le format le plus simple
a,b,c
>>> print("{},{},{}".format('a','b','c')) #Sans numéro
a,b,c
>>> print("{2},{1},{0}".format('a','b','c')) #Ordre changé
c,b,a
>>> print("{0}{1}{0}".format('avo','cat')) # Répétition
avocatavo
>>> |
```

La méthode accepte aussi les paramètres nommés et des expressions

```
>>> print('Coordinates: {latitude}, {longitude}'.format(
latitude='37.24N', longitude='-115.81W'))
Coordinates: 37.24N, -115.81W
>>> coord = (3, 5)
>>> print('X: {0[0]}; Y: {0[1]}'.format(coord))
X: 3; Y: 5
>>> |
```

L'alignement est plus simple :

```
>>> print('A{:<30}B'.format('left aligned'))
Aleft aligned                  B
>>> print('A{:>30}B'.format('right aligned'))
A                 right alignedB
>>> print('A{:^30}B'.format('centered'))
A           centered           B
>>> print('A{:*^30}B'.format('centered'))
A***********centered***********B
>>> |
```

Dates :

```
>>> import datetime
>>> d = datetime.datetime.now()
>>> print('{:%Y-%m-%d %H:%M:%S}'.format(d))
2019-01-04 17:33:55
>>> |
```

7.5.6. Les tuples

- **Définition : tuple**

Les tuple sont un tableau d'objets qui peuvent être de tout type. Ils ne sont pas modifiables (les tuple sont immuables ou immutable). Un tuple apparaît comme une liste d'objets comprise entre parenthèses et séparés par des virgules. Leur création reprend le même format :

```
>>> x = (4,5)
>>> x = (4,5) # Création d'un tuple composé de 2 entiers
>>> x
(4, 5)
>>> x = ("un",1,"deux",2)#création d'un tuple composé de 2 chaînes de caractères
>>> x
('un', 1, 'deux', 2)
>>> y = (3,) # Création d'un tuple d'un élément, sans la virgule
>>> y
(3,)
>>> |
```

Ces objets sont des vecteurs d'objets. Etant donné que les chaînes de caractères sont également des tableaux, ces opérations reprennent en partie celles des _string_paragraphe_chaine et décrites par le paragraphe Common Sequence Operations.

x in s	vrai si x est un des éléments de s

x not in s	réciproque de la ligne précédente
s + t	concaténation de s et t
s * n	concatène n copies de s les unes à la suite des autres
s[i]	retourne le ième élément de s
s[i:j]	retourne un tuple contenant une copie des éléments de sd'indices i à j exclu
s[i:j:k]	retourne un tuple contenant une copie des éléments de s dont les indices sont compris entre i et j exclu, ces indices sont espacés de k :
len(s)	nombre d'éléments de s
min(s)	plus petit élément de s, résultat difficile à prévoir lorsque les types des éléments sont différents
max(s)	plus grand élément de s
sum(s)	retourne la somme de tous les éléments

Les tuples ne sont pas modifiables (ou mutable), cela signifie qu'il est impossible de modifier un de leurs éléments. Par conséquent, la ligne d'affectation suivante n'est pas correcte :

```
>>> a = (4,5)
>>> a[0] = 3
Traceback (most recent call last):
  File "<pyshell#169>", line 1, in <module>
    a[0] = 3
TypeError: 'tuple' object does not support item assignment
>>> |
```

Pour changer cet élément, il est possible de s'y prendre de la manière suivante :

```
>>> a = (4,5)
>>> a = (3,)+a[1:2]#Crée un tuple d'un élément concaténé
>>> #avec la partie inchangée de a
>>> a
(3, 5)
>>> |
```

- **A quoi sert un tuple alors ?**
  - Le tuple permet une affectation multiple :

```
>>> v1, v2 = 11,12
>>> v1
11
>>> v2
12
>>> 
```

- Il permet également de renvoyer plusieurs valeurs lors d'un appel d'une fonction : nous verrons un exemple au moment opportun.

7.5.7. Les complexes

Il existe d'autres types comme le type complex permettant de représenter les nombres complexes. Ce type numérique suit les mêmes règles et fonctionne avec les mêmes opérateurs (excepté les opérateurs de comparaisons) que ceux présentés au paragraphe type_nombre et décrivant les nombres.

```
>>> print(complex(1,1))
(1+1j)
>>> c = complex(4,5)
>>> print(c*c)
(-9+40j)
>>> 
```

Le langage python offre la possibilité de créer ses propres types immuables (voir classe_slots_att) mais ils seront définis à partir des types immuables présentés jusqu'ici.

7.5.8. Les bytes

Le type bytes représente un tableau d'octets. Il fonctionne quasiment pareil que le type str. Les opérations qu'on peut faire dessus sont quasiment identiques :

count(sub[, start[, end]])	Retourne le nombre d'occurences de la séquence d'octets sub, les paramètres par défaut start et end permettent de réduire la recherche entre les octets d'indice start et end exclu. Par défaut, start est nul tandis que end correspond à la fin de la séquence d'octets.
find(sub[, start[, end]])	Recherche une séquence d'octets sub, les paramètres par défaut start et end ont la même signification que ceux de la fonction count. Cette fonction retourne -1 si la recherche n'a pas abouti.
replace(old, new[, count])	Retourne une copie de la séquence d'octets en remplaçant toutes les occurrences de la séquence old par new. Si le paramètre optionnel count est renseigné, alors seules les count premières occurrences seront remplacées.

partition([sep[,maxsplit]])	Découpe la séquence d'octets en se servant de la séquence sepcomme délimiteur. Si le paramètre maxsplit est renseigné, au plus maxsplit coupures seront effectuées.
join (li)	li est une liste, cette fonction agglutine tous les éléments d'une liste séparés par sep dans l'expression sep.join (["un","deux"]).
startswith(prefix[, start[,end]])	Teste si la chaîne commence par prefix.
endswith(suffix[, start[,end]])	Teste si la chaîne se termine par suffix.

Pour déclarer un tableau de bytes, il faut préfixer une chaîne de caractères par b :

```
>>> byte = b"245"
>>> print(byte,type(byte))
b'245' <class 'bytes'>
>>> b = bytes.fromhex("2Ef0 F1f2")
>>> print(b,type(b))
b'.\xf0\xf1\xf2' <class 'bytes'>
>>> |
```

Le type bytes est très utilisé quand il s'agit de convertit une chaîne de caractères d'un encoding à l'autre.

```
>>> b = "abc".encode("utf-8")
>>> s =  b.decode("ascii")
>>> print(b,s)
b'abc' abc
>>> print(type(b),type(s))
<class 'bytes'> <class 'str'>
>>> |
```

Les encoding sont utiles dès qu'une chaîne de caractères contient un caractère non anglais (accent, sigle…). Les bytes sont aussi très utilisées pour sérialiser un objet.

7.6. Notion de liste en Python

7.6.1. Les listes

Une liste en python est une collection de donne de tout type dans une seul variable.

- **Création de liste :** Pour créer une **liste**, rien de plus simple :

```
>>> MaListe = []
>>> Liste = list()
>>> MaListe
[]
>>> Liste
[]
>>> |
```

- **Ajout des éléments à mon ma liste :**

Après la création de la liste avec la méthode append (qui signifie "ajouter" en anglais):

```
>>> Liste
[]
>>> Liste.append(1)
>>> Liste
[1]
>>> Liste.append("Poupoue")
>>> Liste
[1, 'Poupoue']
>>> |
```

- **Afficher un item d'une liste**

Pour lire une liste, on peut demander à voir l'index de la valeur qui nous intéresse :

```
>>> liste = ["a","d","m"]
>>> liste[0]
'a'
>>> liste[2]
'm'
>>> |
```

Le premier item commence toujours avec l'index 0. Pour lire le premier item on utilise la valeur 0, le deuxième on utilise la valeur 1, etc.
Il est d'ailleurs possible de modifier une valeur avec son index

```
>>> liste = ["a","d","m"]
>>> liste[0]
'a'
>>> liste[2]
'm'
>>> liste[2] = "z"
>>> liste
['a', 'd', 'z']
>>> |
```

- **Supprimer une entrée avec un index**

Il est parfois nécessaire de supprimer une entrée de la liste. Pour cela vous pouvez utiliser la fonction `del`.

```
>>> del liste[1]
>>> liste
['a', 'z']
>>> |
```

- **Supprimer une entrée avec sa valeur**

Il est possible de supprimer une entrée d'une liste avec sa valeur avec la méthode `remove`.

```
>>> liste.remove("a")
>>> liste
['z']
>>> |
```

- **Compter le nombre d'items d'une liste**

Il est possible de compter le nombre d'items d'une liste avec la fonction `len`.

```
>>> liste = [1,2,3,5,10]
>>> len(liste)
5
>>> |
```

- **Compter le nombre d'occurrences d'une valeur**

Pour connaitre le nombre d'occurrences d'une valeur dans une liste, vous pouvez utiliser la méthode `count`.

```
>>> liste = ["a","a","a","b","c","c"]
>>> liste.count("a")
3
>>> liste.count("c")
2
>>> |
```

- **Trouver l'index d'une valeur**

La méthode `index` vous permet de connaitre la position de l'item cherché.

```
>>> liste = ["a","a","a","b","c","c"]
>>> liste.index("b")
3
>>> |
```

- **Manipuler une liste**

Voici quelques astuces pour manipuler des listes :

```
>>> liste = [1, 10, 100, 250, 500]
>>> liste[0]
1
>>> liste[-1] # Cherche la dernière occurence
500
>>> liste[-4:] # Affiche les 4 dernières occurrences
[10, 100, 250, 500]
>>> liste[:] # Affiche toutes les occurences
[1, 10, 100, 250, 500]
>>> liste[2:4] = [69, 70]
>>> liste[:] = [] # vide la liste
>>> liste
[]
>>> |
```

- **Transformer un string en liste**

Parfois il peut être utile de transformer une chaine de caractère en liste. Cela est possible avec la méthode split.

```
>>> ma_chaine = "Kaba:Diakité:Amadou:Paris"
>>> ma_chaine.split(":")
['Kaba', 'Diakité', 'Amadou', 'Paris']
>>> |
```

- **Transformer une liste en string**

L'inverse est possible avec la méthode "**join**".

```
>>> ma_chaine = "Kaba:Diakité:Amadou:Paris"
>>> ma_chaine.split(":")
['Kaba', 'Diakité', 'Amadou', 'Paris']
>>> "-".join(ma_chaine)
'K-a-b-a-:-D-i-a-k-i-t-é-:-A-m-a-d-o-u-:-P-a-r-i-s'
>>> |
```

- **Trouver un item dans une liste**

Pour savoir si un élément est dans une liste, vous pouvez utiliser le mot clé **in** de cette manière :

```
>>> liste = [1,2,3,5,10]
>>> 3 in liste
True
>>> 11 in liste
False
>>> |
```

Ces objets sont des listes chaînées d'autres objets de type quelconque (immuable ou modifiable). Il est possible d'effectuer les opérations qui suivent. Ces opérations reprennent celles des tuple (voir opération tuple) et incluent d'autres fonctionnalités puisque les listes sont modifiables. Il est donc possible d'insérer, de supprimer des éléments, de les trier. La syntaxe des opérations sur les listes est similaire à celle des opérations qui s'appliquent sur les chaînes de caractères, elles sont présentées par la table suivante.

x in l	vrai si x est un des éléments de l
x not in l	réciproque de la ligne précédente
l + t	concaténation de l et t
l * n	concatène n copies de l les unes à la suite des autres
l[i]	retourne l'élément i$^text{ème}$ élément de l, à la différence des tuples, l'instruction l [i] = "a" est valide, elle remplace l'élément i par "a". Un indice négatif correspond à la position len(l)+i.

l[i:j]	retourne une liste contenant les éléments de l d'indices i à jexclu. Il est possible de remplacer cette sous-liste par une autre en utilisant l'affectation l[i:j] = l2 où l2 est une autre liste (ou un tuple) de dimension différente ou égale.
l[i:j:k]	retourne une liste contenant les éléments de l dont les indices sont compris entre i et j exclu, ces indices sont espacés de k : Ici encore, il est possible d'écrire l'affectation suivante : l[i:j:k] = l2 mais l2 doit être une liste (ou un tuple) de même dimension que l[i:j:k].
len(l)	nombre d'éléments de l
min(l)	plus petit élément de l, résultat difficile à prévoir lorsque les types des éléments sont différents
max(l)	plus grand élément de l
sum(l)	retourne la somme de tous les éléments
del l [i:j]	supprime les éléments d'indices entre i et j exclu. Cette instruction est équivalente à l [i:j] = [].
list (x)	convertit x en une liste quand cela est possible
l.count (x)	Retourne le nombre d'occurrences de l'élément x. Cette notation suit la syntaxe des classes développée au chapitre Classes. count est une méthode de la classe list.
l.index (x)	Retourne l'indice de la première occurrence de l'élément xdans la liste l. Si celle-ci n'existe pas, une exception est déclenchée (voir le paragraphe Exceptions)
l.append (x)	Ajoute l'élément x à la fin de la liste l. Si x est une liste, cette fonction ajoute la liste x en tant qu'élément, au final, la liste lne contiendra qu'un élément de plus.
l.extend (k)	Ajoute tous les éléments de la liste k à la liste l. La liste laura autant d'éléments supplémentaires qu'il y en a dans la liste k.
l.insert(i,x)	Insère l'élément x à la position i dans la liste l.
l.remove (x)	Supprime la première occurrence de l'élément x dans la liste l. S'il n'y a aucune occurrence de x, cette méthode déclenche une exception.
l.pop ([i])	Retourne l'élément l[i] et le supprime de la liste. Le paramètre i est facultatif, s'il n'est pas précisé, c'est le dernier élément qui est retourné puis supprimé de la liste.

l.reverse (x)	Retourne la liste, le premier et dernier élément échange leurs places, le second et l'avant dernier, et ainsi de suite.
l.sort ([key=None,reverse=False])	Cette fonction trie la liste par ordre croissant. Le paramètre key est facultatif, il permet de préciser la fonction qui précise clé de comparaison qui doit être utilisée lors du tri. Si reverseest True, alors le tri est décroissant. Lire Sorting HOW TO.

7.6.2. Les dictionnaires

Un **dictionnaire** en **python** est une sorte de **liste** mais au lieu d'utiliser des **index**, on utilise des **clés**, c'est à dire des valeurs autres que numériques.

- **Comment créer un dictionnaire ?**

Pour initialiser un **dictionnaire**, on utile la syntaxe suivante :

```
>>> a = {}
>>> b = dict()
>>> |
```

- **Comment ajouter des valeurs dans un** dictionnaire **?**

Pour ajouter des valeurs à un dictionnaire il faut indiquer une clé ainsi qu'une valeur :

```
>>> a = {}
>>> a["nom"] = "engel"
>>> a["prenom"] = "olivier"
>>> a
{'nom': 'engel', 'prenom': 'olivier'}
>>> |
```

Vous pouvez utiliser des clés numériques comme dans la logique des listes.

- **Récupérer une valeur dans un dictionnaire**

La méthode get vous permet de récupérer une valeur dans un dictionnaire et si la clé est introuvable, vous pouvez donner une valeur à retourner par défaut :

```
>>> data = {"name": "Olivier", "age": 30}
>>> data.get("name")
'Olivier'
>>> data.get("adresse", "Adresse inconnue")
'Adresse inconnue'
>>> |
```

- **Supprimer une entrée de dictionnaire**

Il est possible de supprimer une entrée en indiquant sa clé, comme pour les listes :

```
>>> del a["nom"]
>>> a
{'prenom': 'olivier'}
>>>
```

- **Récupérer les clés par une boucle**

Pour récupérer les clés on utilise la méthode keys

```
>>> for cle in fiche.keys():
        print(cle)


nom
prenom
>>>
```

- **Récupérer les valeurs par une boucle**

Pour cela on utilise la méthode values

```
>>> for valeur in fiche.values():
        print(valeur)


engel
olivier
>>>
```

- **Récupérer les clés et les valeurs par une boucle**

Pour récupérer les clés et les valeurs en même temps, on utilise la méthode items qui retourne un tuple.

```
>>> for cle,valeur in fiche.items():
        print(cle,valeur)


nom engel
prenom olivier
```

- **Utiliser des tuples comme clé**

Une des forces de python est la combinaison tuple/dictionnaire qui fait des merveilles dans certains cas comme lors de l'utilisation de coordonnées.

```
>>> b = {}
>>> b[(3,2)]=12
>>> b[(4,5)]=13
>>> b
{(4, 5): 13, (3, 2): 12}
>>>
```

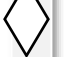
- **Créer une copie indépendante d'un dictionnaire**

Comme pour toute variable, vous ne pouvez pas copier un dictionnaire en faisant dic1 = dic2:

```
>>> d = {"k1":"olivier", "k2":"engel"}
>>> e = d
>>> e
{'k1': 'olivier', 'k2': 'engel'}
>>> del e["k1"]
>>> d
{'k2': 'engel'}
>>> |
```

Toutes les actions menées sur e affectent directement d. Ce n'est donc pas une copie que vous créez en faisant e = d. Solution :
Pour créer une copie indépendante vous pouvez utiliser la méthode **copy**:

```
>>> d = {"k1":"olivier", "k2":"engel"}
>>> e = d.copy()
>>> del e["k1"]
>>> d
{'k1': 'olivier', 'k2': 'engel'}
>>> e
{'k2': 'engel'}
>>> # Voilà ! C'est fait !!!
```



7.6.3. Les fonctions des dictionnaires

La plupart des fonctions disponibles pour les listes sont interdites pour les dictionnaires comme la concaténation ou l'opération de multiplication (*). Il n'existe plus non plus d'indices entiers pour repérer les éléments, le seul repère est leur clé. La table suivante dresse la liste des opérations sur les dictionnaires.

x in d	vrai si x est une des clés de d
x not in d	réciproque de la ligne précédente
d[i]	retourne l'élément associé à la clé i
len(d)	nombre d'éléments de d
min(d)	plus petite clé
max(d)	plus grande clé
del d [i]	supprime l'élément associé à la clé i
list (d)	retourne une liste contenant toutes les clés du dictionnaire d
dict (x)	convertit x en un dictionnaire si cela est possible, d est alors égal à dict (d.items ())
d.copy ()	Retourne une copie de d

d.items ()	Retourne un itérateur sur tous les couples (clé, valeur) inclus dans le dictionnaire.
d.keys ()	Retourne un itérateur sur toutes les clés du dictionnaire d
d.values ()	Retourne un itérateur sur toutes les valeurs du dictionnaire d
d.get (k[,x])	Retourne d[k], si la clé k est manquante, alors la valeur Noneest retournée à moins que le paramètre optionnel x soit renseigné, auquel cas, ce sera cette valeur qui sera retourné.
d.clear ()	Supprime tous les éléments du dictionnaire.
d.update(d2)	Le dictionnaire d reçoit le contenu de d2.
d.setdefault(k[,x])	Définit d[k] si la clé k existe, sinon, affecte x à d[k]
d.pop()	Retourne un élément et le supprime du dictionnaire.

7.7. Structures conditionnelles

7.7.1. La condition if (Si .... Alors)

Cette notion est l'une des plus importante en programmation. L'idée est de dire que **si** telle variable a telle valeur **alors** faire cela. Prenons un exemple, on va donner une valeur à une variable et si cette valeur est supérieure à 5, alors on va incrémenter la valeur de 1.

```
>>> a = 10
>>> if a > 5:
        a = a + 1


>>> a
11
>>> |
```

Que se passe-t-il si la valeur était inférieure à 5 ?

```
>>> a = 3
>>> if a > 5 :
        a = a + 1


>>> a
3
>>> |
```

On remarque que si la condition n'est pas remplie, les instructions dans la structure conditionnelle sont ignorées.

7.7.2. La condition if ... else (Si ... Alors ..... Sinon)

Il est possible de donner des instructions quel que soit les choix possibles avec le mot clé else :

```
>>> x = 20
>>> if x > 5:
        x = x + 10
else:
        x = x+16


>>> x
30
>>> |
```

7.7.3. Les conditions if ... elif ... else... (Si ... Alors ... Sinon Si)

Il est possible d'ajouter autant de conditions précises que l'on souhaite en ajoutant le mot clé elif , contraction de "else" et "if", qu'on pourrait traduire par "sinon Si". C'est l'imbrication des Si en LDA.

```
>>> if a > 5:
        a = a + 1
elif a == 5:
        a = a + 1000
else:
        a = a - 1


>>> a
1005
>>> |
```

Dans cet exemple, on a repris le même que les précédent mais nous avons ajouté la conditions "Si la valeur est égale à 5" que se passe-t-il ? Et bien on ajoute 1000.

7.7.4. Comment fonctionnent les structures conditionnelles ?

Les mots clé **if, elif** et **else** cherchent à savoir si ce qu'on leur soumet est True. En anglais True signifie "Vrai". Donc si c'est la valeur est True, les instructions concernant la condition seront exécutées.

- **Les comparaisons possibles**

Il est possible de comparer des éléments :

==	égal à
!=	différent de (fonctionne aussi avec)
>	strictement supérieur à
>=	supérieur ou égal à
<	strictement inférieur à
<=	inférieur ou égal à

Comment savoir si la valeur qu'on soumet à l'interpréteur est True ? Il est possible de le voir directement dans l'interpréteur. Demandons à python si 3 est égal à 4 :

```
>>> 3 == 4
False
>>> |
```

Il vous répondra gentiment que c'est False, c'est à dire que c'est faux . Maintenant on va donner une valeur à une variable est on va lui demander si la valeur correspond bien à ce que l'on attend.

```
>>> y = 5 # affectation
>>> y == 5 # Test
True
>>> |
```

- **AND / OR**

Il est possible d'affiner une condition avec les mots clé AND qui signifie " ET " et OR qui signifie "OU". On veut par exemple savoir si une valeur est plus grande que 5 mais aussi plus petite que 10 :

```
>>> v = 15
>>> v > 5 and v < 10
False
>>> |
```

Essayons avec la valeur 7 :

```
>>> v = 7
>>> v > 5 and v < 10
True
>>> |
```

Pour que le résultat soit TRUE, il faut que les deux conditions soient remplies Testons maintenant la condition OR

```
>>> v = 11
>>> v > 5 or v < 100
True
>>> |
```

Le résultat est TRUE parce qu'au moins une des deux conditions est respectée.

```
>>> v = 1
>>> v > 5 or v > 100
False
>>> |
```

Dans ce cas aucune condition n'est respectée, le résultat est donc FALSE.

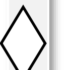
- **Chainer les comparateurs**

Il est également possible de chainer les comparateurs :

```
>>> a, b, c = 1, 10, 100
>>> a < b < c
True
>>> a > b < c
False
>>> |
```

Il est également possible de comparer les caractères alphabétiques :

```
>>> a ="a"
>>> b = "b"
>>> a < b
True
>>> a < "z"
True
>>> "f" > "k"
False
>>> "A" > "a"
False
>>> |
```



Le classement par ordre croissant donne ceci :
`ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz`

7.8. Structures de boucle

Une boucle (*ou loop*) vous permet de répéter à l'infini ou à un nombre fini, des instructions selon vos besoins.

7.8.1. La boucle while (TantQue ... Faire)

En anglais " *while* " signifie "Tant que". Pour créer une **boucle**, il faut donc utiliser ce mot clé suivi d'une indication qui dit quand la boucle s'arrête. Cette boucle prend la place de TantQue … Faire et Repeter …. jusqu'à (condition initiale vraie).

Un exemple sera plus parlant : On désire écrire 10 fois cette phrase : " *Je dis étudier mo cours de Python* " Ecrire à la main prend beaucoup de temps et beaucoup de temps x 10 c'est vraiment beaucoup de temps, et peu fiable, même pour les chanceux qui connaissent le copier/coller. Et un bon programmeur est toujours un peu *fainéant* perfectionniste, il cherchera la manière la plus élégante de ne pas répéter du code.

```
>>> i = 0
>>> while i < 10 :
        print("Je dois étudier mon cours de Python")
        i = i + 1 # i += 1 est possible


Je dois étudier mon cours de Python
Je dois étudier mon cours de Python
Je dois étudier mon cours de Python
Je dois étudier mon cours de Python
Je dois étudier mon cours de Python
Je dois étudier mon cours de Python
Je dois étudier mon cours de Python
Je dois étudier mon cours de Python
Je dois étudier mon cours de Python
Je dois étudier mon cours de Python
>>>
```

7.8.2. La boucle for (Pour … Faire)

La boucle for permet de faire des itérations sur un élément, comme une chaine de caractères par exemple ou une liste.
Syntaxe : *For element in Itérateur: "Pour ….. Faire"*

- Range()

Il est possible de créer une boucle facilement avec range qui est un itérateur ; range(a,b, pas)
For i in range(a, b, pas): correspond à Pour I ← a, b – 1, pas Faire

```
>>> for i in range(1,11):
        print(i)


1
2
3
4
5
6
7
8
9
10
>>> 
```

- Chaine de caractère comme itérateur

```
>>> phrase = "ZRAN"
>>> for car in phrase:
        print(car)


Z
R
A
N
>>> 
```

- **Stopper une boucle avec break**

Pour stopper immédiatement une boucle on peut utiliser le mot clé break. Cette instruction est très utile lorsque que l'on construit une boucle infinie.

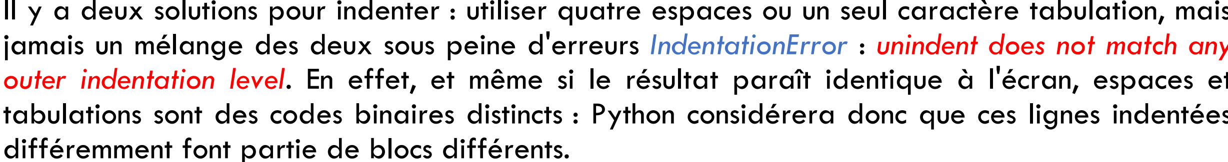
```
>>> for k in range(5):
        if k == 2:
                break
        else:
                print(k)


0
1
>>> |
```

7.8.3. Structure d'un programme en python

La programmation en python est organisée en bloc d'instruction. Un bloc d'instructions est une suite d'instructions qui est alignée sur la même tabulation. Les blocs d'instructions sont créés par les instructions de contrôles comme *if*, *while* et *for*, ainsi que par les instructions permettant de déclarer des fonctions.

Sous Python, toutes les instructions composées ont toujours la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs instructions indentées sous cette ligne d'en-tête. L'indentation est obligatoire en python.

```
Ligne d'en-tête :
    première instruction du bloc
    ... ...
    ... ...
    dernière instruction du bloc
Autre instruction
Une autre par ici
```



Il y a deux solutions pour indenter : utiliser quatre espaces ou un seul caractère tabulation, mais jamais un mélange des deux sous peine d'erreurs *IndentationError* : *unindent does not match any outer indentation level*. En effet, et même si le résultat paraît identique à l'écran, espaces et tabulations sont des codes binaires distincts : Python considérera donc que ces lignes indentées différemment font partie de blocs différents.

7.9. Fonctions et procédures

Une fonction est un programme portant un nom et réutilisable dans un ou plusieurs autres programmes. Son utilisation nous permet de coder très vite rendant la vie du programmeur très douce. Elle exécute une tâche unique !
Une fonction n'est pas obligée de renvoyer une valeur, on parlera alors dans ce cas plutôt de procédure.

```
def maFonction():

        Mes_instructions

        return resultat
```

```
def maProcedure():

        Mes_instructions
```

Les fonctions et les procédures sont utilisées à cause de leur capacité à avoir des paramètres à l'entrée. Elles ont la capacité d'avoir assez d'argument possible.

7.9.1. Les paramètres

Une peut de maths : En mathématiques, une fonction est une relation entre un ensemble d'entrées (variables ou paramètres en informatique) et un ensemble de sorties (image ou retour en informatique), avec la propriété que **chaque entrée est liée au plus à une unique sortie.**

Exemple : programmons la fonction suivante : $f(x) = \mathrm{x}^2$ avec x un réel

```
>>> augment_moi(5)
7
>>> def f(x):
        return x**2

>>> f(5)
25
>>> |
```

Un autre exemple :

```
>>> def augment_moi(a):
        return a + 2

>>> augment_moi(5)
7
>>> |
```

Cette fonction incrémente de 2 une valeur que l'on passe en paramètre.
Il est d'ailleurs possible d'utiliser plusieurs paramètres :

```
>>> def addition(a,b):
        return a + b

>>> addition(45,78)
123
>>> valeur = addition(4,9)
>>> valeur
13
>>> |
```

Exemple : Procédure : une procédure qui permet de voir si vous êtes adulte.

```
>>> def Adulte(age):
        if age < 18:
                print("Vous êtes mineur")
        else :
                print("vous êtes bien un adulte")


>>> a = 15
>>> Adulte(a)
Vous êtes mineur
```

Si vous avez compris les principes des fonctions, vous avez compris 80% de ce qu'est la programmation.

7.9.2.Variable locale, variable globale :

On parle de la portée des variables

Une variable déclarée à la racine d'un module est visible dans tout ce module. On parle alors de **variable globale**.

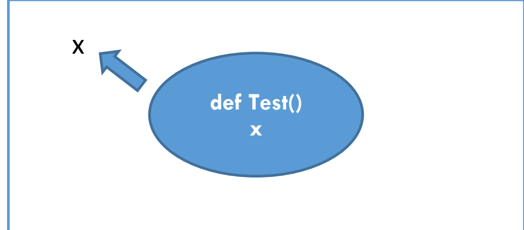
```
>>> salut = "Hello"
>>> def test():
        print(salut)


>>> test()
Hello
>>> |
```

La variable *salut* est une variable globale.

Et une variable déclarée dans une fonction ne sera visible que dans cette fonction. Sa durée de vie part de l'appel de la fonction et la fin de l'exécution de cette fonction. On parle alors de variable locale.

```
>>> x = False # On déclare ici une varible globale
>>> def Test():
        x = True # On essaie de modifier le contenu de x à True


>>> # Que ce passe t-il ?
>>> Test() # essayons de modifier x
>>> x # on affiche x
False
>>> # la valeur de x (global) s'affiche
```

On récapitule :



Le x dans l'ellipse (fonction) n'est accessible que dans son environnement qui l'ellipse : c'est une variable locale. L'autre x qui se trouve en dehors de l'ellipse est visible par la fonction : c'est une variable globale.

7.9.3. Exemple de fonction

- **abs(x)**

Retourne une valeur absolue

```
>>> x = -5.3
>>> abs(x)
5.3
>>> |
```

- **str.capitalize()**

La méthode capitalize permet de mettre une chaine de caractères au format Xxxxx

```
>>> "julien".capitalize()
'Julien'
>>> |
```

- **eval**(expression,globals=None,locals=None)

Exécute une chaine de caractères.

```
>>> val = "58+69"
>>> eval(val)
127
>>> |
```

- **len(s)**

Retourne le nombre d'items d'un objet.

```
>>> len("Le monde court à sa fin")
23
>>> |
```

- **max() / min()**

Retourne la valeur la plus élevée pour max() et la plus basse pour min()

```
>>> max([1,3,2,6,99,1])
99
>>> min([1,3,2,6,99,1])
1
>>> |
```

- **help(element)**

Cette fonction vous retourne des informations sur l'utilisation de l'élément qui vous intéresse.

```
>>> help(str)
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
```

7.9.4. Fiche TD 4 : les bases du langage Python

7.9.5. Fiche TD 5 : Fonctions en Python, notion de paramètres

7.9.6. Fiche TP 3 : Base du langage, Fonctions

7.10. Compréhension des listes !

Un bon développeur cherchera toujours à augmenter sa productivité avec le temps. Il existe des astuces python qui permettent d'optimiser le code.
L'idée est simple : simplifier le code pour le rendre plus lisible et donc plus rapide à écrire et plus simple à maintenir.

- **Syntaxe**

```
new_list = [function(item) for item in list if condition(item)]
```

- **Filtrer une liste**

Prenons un exemple d'une liste :

```
>>> ma_liste = [1,7,8,0,1,55,4,8,9,0,3,5,7,6]
>>> #Nous voulons filtrer ma_liste et ne garder que les valeurs qui
>>> #sont supérieures à 5. Voici un bout de code
>>> b = list()
>>> for item in ma_liste:
        if item > 5:
                b.append(item)


>>> b
[7, 8, 55, 8, 9, 7, 6]
>>>
```

Il est possible de faire exactement ce que fait ce bloc de code en une seule ligne :

```
>>> b = [item for item in ma_liste if item> 5]
>>> b
[7, 8, 55, 8, 9, 7, 6]
>>>
```

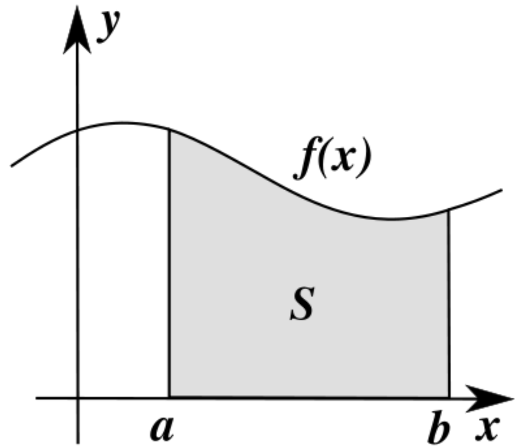
Exécuter une fonction sur chaque item d'une liste

Prenons l'exemple d'une conversion de string en integer de plusieurs items :

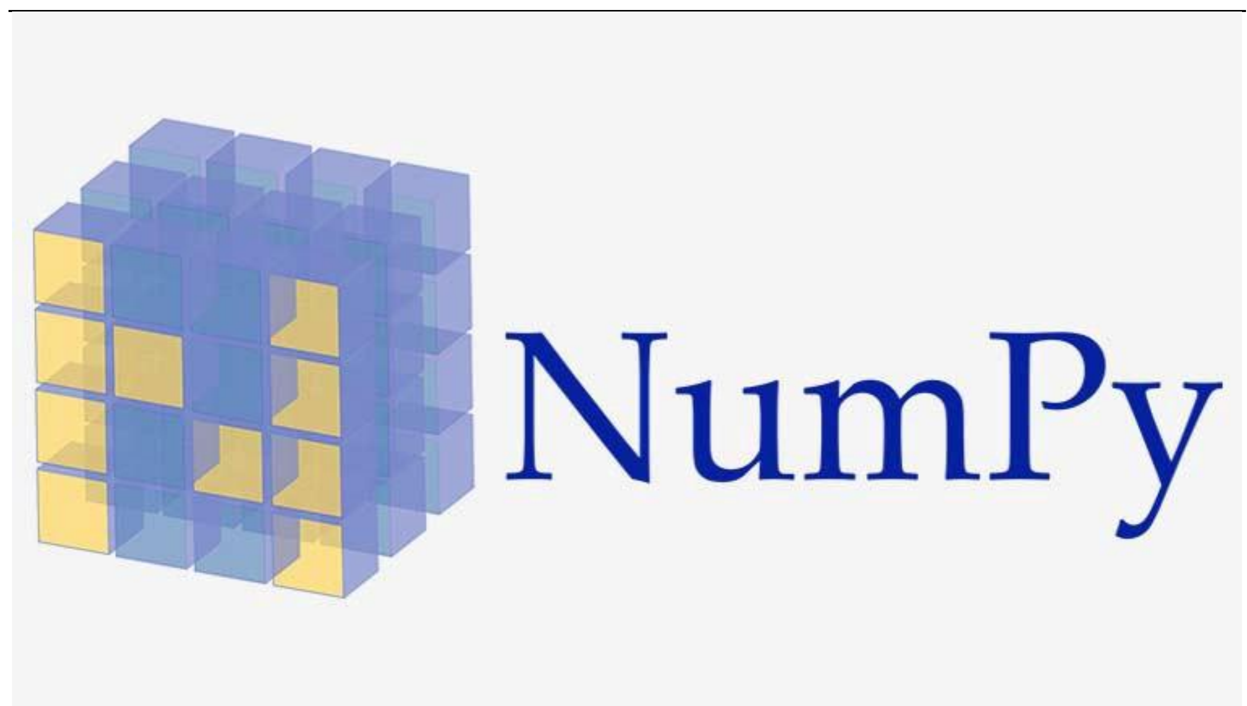
```
>>> items = ["5", "10", "15"]
>>> items = [int(x) for x in items]
>>> items
[5, 10, 15]
>>>
```

7.10.1. TD 5 : Manipulation des conteneurs

7.10.2. TP 3 : Manipulation des conteneurs



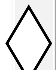
CHAPITRE 8 : Python et les calculs scientifiques



8.1. Les modules et packages

Un module est un ensemble de fonctions écrites qui ont un lien précis dans un seul ficher .py. Ces fonctions pourront être utilisées dans un autre programme juste par importation avec la commande (module.py) :

```
>>> import module # pour importer toutes les fonctions de mon module
>>>Module.fonction1() # pour faire appel à la fonction1 de mon module module


>>>from module import fonction1, fonction2
>>>fonction1()#dans ce cas d'importation on utilise la fonction1 directement sa
ns précéder du nom du module.


>>>from module import* # Pour importer aussi toutes les fonctions du module
>>>fonction1()#dans ce cas d'importation on utilise la fonction1 directement sa
>>>fonction2()
>>>fonction3()

>>>from module import fonction1, fonction2

>>>from module import* # Pour importer aussi toutes les fonctions du module
```



Le package c'est un dossier qui contient plusieurs modules. Son importation est similaire à celui des modules.

```
>>>from package.module import fonction1, fonction2

>>>from package.module import* # Pour importer aussi toutes les fonctions du module
```

Le renommage du module

Il arrive parfois que le nom du module soit trop compliqué pour la lecture ou trop long à écrire. Il existe un moyen de changer son nom : c'est le renommage avec la clause as.

```
>>>import module as mod
Mod.fonction1()
```

Le package c'est un dossier qui contient plusieurs modules. Son importation est similaire à celui des modules.

```
>>>from package.module import fonction1, fonction2
>>>from package.module import* # Pour importer aussi toutes les fonctions du module
```

Il est aussi possible de renommer les packages :

```
>>>import package.module as pacmod
```

8.2. **Numpy** (Numerical Python)

La bibliothèque NumPy (http://www.numpy.org/) permet d'effectuer des calculs numériques avec Python. Elle introduit une gestion facilitée des tableaux de nombres.

Le package package python n'est pas installer avec l'environnement de travail standard. Pour l'installer suivez les instructions suivantes :

1. Rassurez d'être connecté à l'internet
2. Ouvrez une invite de commande en tapant « cmd » dans la barre de recherche windows puis appuyez sur « entrée ».
3. Rendez vous dans le répertoire où vous avez installé python dans mon cas je tape dans l'invite de commande « d : » puis je valide.
4. Placez vous dans le répertoire suivante « cd D:/Python3/Scripts »
5. Tapez la commande suivante « pip install numpy » puis appuyez sur « entrée ».

```
D:\>cd Python3

D:\Python3>cd Scripts

D:\Python3\Scripts>pip install numpy
Collecting numpy
  Downloading https://files.pythonhosted.org/packages/a1/1a/d3491298c548870dd9c31d40f0234ff71a1f337d98581c978338d6b83d00
/numpy-1.15.4-cp35-none-win_amd64.whl (13.5MB)
    100% |################################| 13.5MB 95kB/s
Installing collected packages: numpy
Successfully installed numpy-1.15.4
You are using pip version 8.1.1, however version 18.1 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

Installez par la même occasion la bibliothèque matplotlib que nous utiliserons plus tard !

```
D:\Python3\Scripts>pip install matplotlib
Collecting matplotlib
  Downloading https://files.pythonhosted.org/packages/3b/29/b2b657b4cbb306c6cfe82227c6f61d9939b0f3930c61e38766f3e98a91a2/matplotlib-3.0.2-cp35-cp35m-win_amd64.whl (8.9MB)
    100% |################################| 8.9MB 143kB/s
Collecting python-dateutil>=2.1 (from matplotlib)
  Downloading https://files.pythonhosted.org/packages/74/68/d87d9b36af36f44254a8d512cbfc48369103a3b9e474be9bdfe536abfc45/python_dateutil-2.7.5-py2.py3-none-any.whl (225kB)
    100% |################################| 235kB 3.2MB/s
Collecting pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 (from matplotlib)
  Downloading https://files.pythonhosted.org/packages/71/e8/6777f6624681c8b9701a8a0a5654f3eb56919a01a78e12bf3c73f5a3c714/pyparsing-2.3.0-py2.py3-none-any.whl (59kB)
    100% |################################| 61kB 2.8MB/s
Requirement already satisfied (use --upgrade to upgrade): numpy>=1.10.0 in d:\python3\lib\site-packages (from matplotlib)
Collecting cycler>=0.10 (from matplotlib)
  Downloading https://files.pythonhosted.org/packages/f7/d2/e07d3ebb2bd7af696440ce7e754c59dd546ffe1bbe732c8ab68b9c834e61/cycler-0.10.0-py2.py3-none-any.whl
Collecting kiwisolver>=1.0.1 (from matplotlib)
  Downloading https://files.pythonhosted.org/packages/67/57/834881c80cd1361792a18b467ac8c1638c224a484956582e51d2f9e16e30/kiwisolver-1.0.1-cp35-none-win_amd64.whl (57kB)
    100% |################################| 61kB 2.7MB/s
Collecting six>=1.5 (from python-dateutil>=2.1->matplotlib)
  Downloading https://files.pythonhosted.org/packages/73/fb/00a976f728d0d1fecfe898238ce23f502a721c0ac0ecfedb80e0d88c64e9/six-1.12.0-py2.py3-none-any.whl
Requirement already satisfied (use --upgrade to upgrade): setuptools in d:\python3\lib\site-packages (from kiwisolver>=1.0.1->matplotlib)
Installing collected packages: six, python-dateutil, pyparsing, cycler, kiwisolver, matplotlib
Successfully installed cycler-0.10.0 kiwisolver-1.0.1 matplotlib-3.0.2 pyparsing-2.3.0 python-dateutil-2.7.5 six-1.12.0
You are using pip version 8.1.1, however version 18.1 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

Le module numpy vient d'être installe. Il reste à l'importer pour continuer le cours.

```
>>> import numpy as np
>>> |
```

8.2.1. Variables prédéfinies

Variable **pi** et **e** NumPy définit par défaut la valeur de pi.

```
>>> import numpy as np
>>> np.pi
3.141592653589793
>>> np.e
2.718281828459045
```

8.2.2. Tableaux - numpy.array()

- **Création**

Les tableaux (en anglais, array) peuvent être créés avec numpy.array(). On utilise des crochets pour délimiter les listes d'éléments dans les tableaux.

```
>>> import numpy as np
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
```

- **Affichage**

```
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> type(a)
<class 'numpy.ndarray'>
```

On voit que l'on a obtenu un objet de type numpy.ndarray

- **Accès aux éléments d'un tableau**

Avertissement Comme pour les listes, les indices des éléments commencent à zéro.

```
>>> a[0,1]
2
>>> a[1,2]
6
```

8.2.3. La fonction numpy.arange()

```
>>> m = np.arange(3, 15, 2)
>>> m
array([ 3,  5,  7,  9, 11, 13])
>>> type(m)
<class 'numpy.ndarray'>
```

Noter la différence entre numpy.arange() et range() :

- numpy.arange() retourne un objet de type numpy.ndarray.
- range() retourne un objet de type range.

```
>>> n = range(3, 15, 2)
>>> n
range(3, 15, 2)
>>> type(n)
<class 'range'>
```

Ceci est également à distinguer d'une liste.

```
>>> u = [3, 7, 10]
>>> type(u)
<class 'list'>
```

Il est possible d'obtenir des listes en combinant list et range().

```
>>> list(range(3, 15, 2))
[3, 5, 7, 9, 11, 13]
```

numpy.arange() accepte des arguments qui ne sont pas entiers

```
>>> np.arange(0, 11*np.pi, np.pi)
array([ 0.        ,  3.14159265,  6.28318531,  9.42477796, 12.56637061,
       15.70796327, 18.84955592, 21.99114858, 25.13274123, 28.27433388,
       31.41592654])
```

8.2.4. La fonction numpy.linspace()

numpy.linspace() permet d'obtenir un tableau 1D allant d'une valeur de départ à une valeur de fin avec un nombre donné d'éléments.

```
>>> np.linspace(3, 9, 10)
array([3.        , 3.66666667, 4.33333333, 5.        , 5.66666667,
       6.33333333, 7.        , 7.66666667, 8.33333333, 9.        ])
```

8.2.5. Action d'une fonction mathématique sur un tableau

NumPy dispose d'un grand nombre de fonctions mathématiques qui peuvent être appliquées directement à un tableau. Dans ce cas, la fonction est appliquée à chacun des éléments du tableau.

```
>>> x = np.linspace(-np.pi/2, np.pi/2, 3)
>>> x
array([-1.57079633,  0.        ,  1.57079633])
>>> y = np.sin(x)
>>> y
array([-1.,  0.,  1.])
>>> 
```

8.2.6. Calcul sur les matrices

- **Quelques méthodes des matrices : soit** $A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$

```
>>> A = np.array([[1,1,1],[0,1,1],[0,0,1]])
>>> 
```

Le type d'une matrice s'obtient grâce à la fonction shape(), son nombre d'élément grâce à la fonction size().
La fonction reshape() ou flatten() permet de changer la forme (=type) d'une matrice (transformer un array bi-dimensionnel en un array uni-dimensionnel .

- L'instruction v.flatten() renvoie une copie de v (où v est une matrice)
- la syntaxe de reshape np.reshape(array,dimension)

✓ Le produit matriciel s'obtient à l'aide de la fonction **dot()**.
Si le deuxième argument est une matrice ligne, dot() la traitera si besoin est comme une matrice colonne, ou vecteur, en la transposant.

- Pour effectuer le produit scalaire de 2 "vecteurs" utiliser la fonction **vdot()**
- La fonction **transpose()** permet d'obtenir la transposée
- le sous-module **linalg** contient, entre autres :
  - la fonction **inv()** permettant d'obtenir l'inversion de matrice.
  - la fonction **det()** retourne le déterminant d'une matrice.
  - La fonction **solve(A,b)** résout le système linéaire de matrice A et de second membre b (vecteur ou ligne)

- **Exemple 1 : Accès aux éléments de la matrice A**

```
>>> import numpy as np
>>> A = np.array([[1,1,1],[0,1,1],[0,0,1]])
>>> A
array([[1, 1, 1],
       [0, 1, 1],
       [0, 0, 1]])
>>> A[0]
array([1, 1, 1])
>>> A[2][1]
0
>>> print(A[2][2])
1
```

```
>>> np.reshape(A,9)
array([1, 1, 1, 0, 1, 1, 0, 0, 1])
>>> 
```

- **Exemple 2: Slicing, et nombre d'éléments d'une matrice**

```
>>> print(A[0:1])
[[1 1 1]]
>>> print(A[::-1])
[[0 0 1]
 [0 1 1]
 [1 1 1]]
>>> print(np.size(A))
9
>>> 
```

Ce tableau récapitule les **Slicing :**

Syntaxe	Valeur de retour
A[: , i] où A est une matrice, i un entier	Extrait la i+1-ème colonne de A sous forme de matrice ligne
A[: , :i] où A est une matrice, i un entier	Extrait les i premières colonnes de A sous forme de matrice
A[: , i:] où A est une matrice, i un entier	Extrait les i+1 dernières colonnes de A sous forme de matrice
A[: , i:j+1] où A est une matrice, i et j deux entiers	Extrait de la i-ème à la j-ème colonne de A sous forme de matrice
A[i , :] où A est une matrice, i un entier	Extrait la i+1-ème ligne de A sous forme de matrice ligne
A[i][j] où A est une matrice, i et j deux entiers	Extrait le coefficient d'indice (i+1, j+1) de A

- **Exemple 3 :** Une autre méthode d'extraction consiste à écrire v[b] où b est un tableau d'entiers qui correspondra aux indices à extraire et v un vecteur. Pour les matrices c'est B[b,c] et on prend les indices de ligne dans b, les indices de colonnes dans c, successivement

```
>>> import numpy as np
>>> A = np.array([[1,1,1],[0,1,1],[0,0,1]])
>>> c = np.array([0,1,1],int)
>>> b = np.array([0,0,1],int)
>>> B = A+6
>>> B
array([[7, 7, 7],
       [6, 7, 7],
       [6, 6, 7]])
```

```
>>> B[b,c]
array([7, 7, 7])
>>> |
```

- **Exemple 4 : dot() et linal** Pour élever une matrice carrée A à une puissance N , on pourrait coder comme ceci: Dans cet exemple, à la sortie, B contient A^N

```
>>> import numpy as np
>>> A = np.array([[1,1,1],[0,1,1],[0,0,1]])
>>> A
array([[1, 1, 1],
       [0, 1, 1],
       [0, 0, 1]])
>>> B = A
>>> N = 10
>>> for k in range(1,N):
        B = np.dot(A,B)


>>> B
array([[ 1, 10, 55],
       [ 0,  1, 10],
       [ 0,  0,  1]])
>>> |
```

De même, A** -1 se fait terme à terme . Cela produit une erreur ici (division par 0) alors que la matrice est inversible.
Le sous module linalg de numpy contient des fonctions spécifiques telles que la fonction inv() permettant de réaliser l'inversion de matrice, la fonction matrix_power() pour réaliser A^N

- **Autres fonctions de numpy sur des objets de type ndarray**
  - shape indique le format du tableau, sous la forme du tuple du nombre d'éléments dans chaque direction
  - size donne le nombre total d'éléments. np.size(A,0) et np.size(A,1) donnent respectivement le nombre de lignes et le nombre de colonnes d'une matrice A
  - ndim renvoie le nombre d'indices nécessaires au parcours du tableau (usuellement : 1 pour un vecteur, 2 pour une matrice)
  - alen donne la première dimension d'un tableau (la taille pour un vecteur, le nombre de lignes pour une matrice)

```
>>> # Pour l'inverse de A
>>> np.linalg.inv(A)
array([[ 1., -1.,  0.],
       [ 0.,  1., -1.],
       [ 0.,  0.,  1.]])
>>> # Pour élever A à la puissance 10
>>> np.linalg.matrix_power(A,10)
array([[ 1, 10, 55],
       [ 0,  1, 10],
       [ 0,  0,  1]])
>>> # pour déterminer le rang de A
>>> np.linalg.matrix_rank(A)
3
>>> |
```

- **Exemple 5 : Résoudre le système linéaire :**

$$\begin{cases} x + y + 2z = 5 \\ x - y - z = 1 \\ x + z = 3 \end{cases} \quad \text{A.X = B soit} \quad \begin{pmatrix} 1 & 1 & 2 \\ 1 & -1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ 3 \end{pmatrix}$$

```
>>> import numpy as np
>>> A = np.array([[1,1,2],[1,-1,-1],[1,0,1]])
>>> B = np.array([5,1,3])
>>> print(np.linalg.solve(A,B))
[3. 2. 0.]
>>> # On pourrait aussi faire :
>>> print(np.dot(np.linalg.inv(A),B))# En inversant A
[3. 2. 0.]
>>> |
```

- **Matrices particulières**
  - **np.zeros(n)** , respectivement np.zeros((n,p)): donne un vecteur nul de taille n, resp. une matrice nulle de taille n, p
  - **np.eye(n)** ou identity(n) (respectivement np.eye(n,p)): matrice de taille n (respectivement une matrice de taille n,p) avec des 1 sur la diagonale et des zéros ailleurs
  - **np.ones(n)** (respectivement np.ones((n,p))) : vecteur de taille n, (respectivement matrice de taille n,p) rempli(e) de 1

- **np.diag(v)** : matrice diagonale dont la diagonale est constituée des coordonnées du vecteur v
- **np.diag(v,k):** matrice dont la « diagonale » décalée de k, un entier relatif, est constituée des coordonnées du vecteur v (k est un entier relatif)
- **np.random.rand(n)** (respectivement np.random.rand(n,p)): vecteur (taille n) (respectivement matrice (taille n,p)) à coefficients aléatoires uniformes sur [0,1[
- **fromfunction:** permet de construire un tableau dont le terme général obéit à une formule donnée. La syntaxe est fromfunction(function, shape, [dtype]) où function est une fonction dont le nombre de paramètres est déterminé par le format de la matrice indiqué par shape

- **Voici une liste de quelques méthodes supplémentaires.**

Fonctions	Rôle
array(l)	Crée un tableau à partir d'une liste l
arange(a,b,c)	Crée un vecteur dont les coefficients sont les a+k entre a (inclus) et b (inclus)
linspace(a,b,n)	Crée un vecteur de n valeurs régulièrement espacées entre a et b (inclus)
zeros(p)	Crée un tableau de taille p rempli de zéros
zeros((p,q))	Crée un tableau de taille (p,q) rempli de zéros
ones(p)	Crée un tableau de taille p rempli de uns
ones((p,q))	Crée un tableau de taille (p,q) rempli de uns
shape()	Pour obtyenir la taille d'un tableau. C'est le type d'une matirce
reshape()	Pour redimensionner un tableau
dot()	Pour effectuer un produit matriciel de deux matrices
vdot()	Pour effectuer un produit scalaire de deux vecteurs
transpose()	Pour transposer une matrice
rank()	Rang d'une matrice
mean()	Valeur moyen d'un tableau
odeint()	Pour intégrer une équation différentielle
linalg	Pour les opérations algébriques sur les matrices
inv()	Inversion d'une matrice
det()	Déterminant d'une matrice
solve(a,b)	Résoudre le système linéaire AX = b

8.2.7. Fiche TP 4 : Ecriture de modules, manipulation de NumPy

8.3. Le module matplotlib

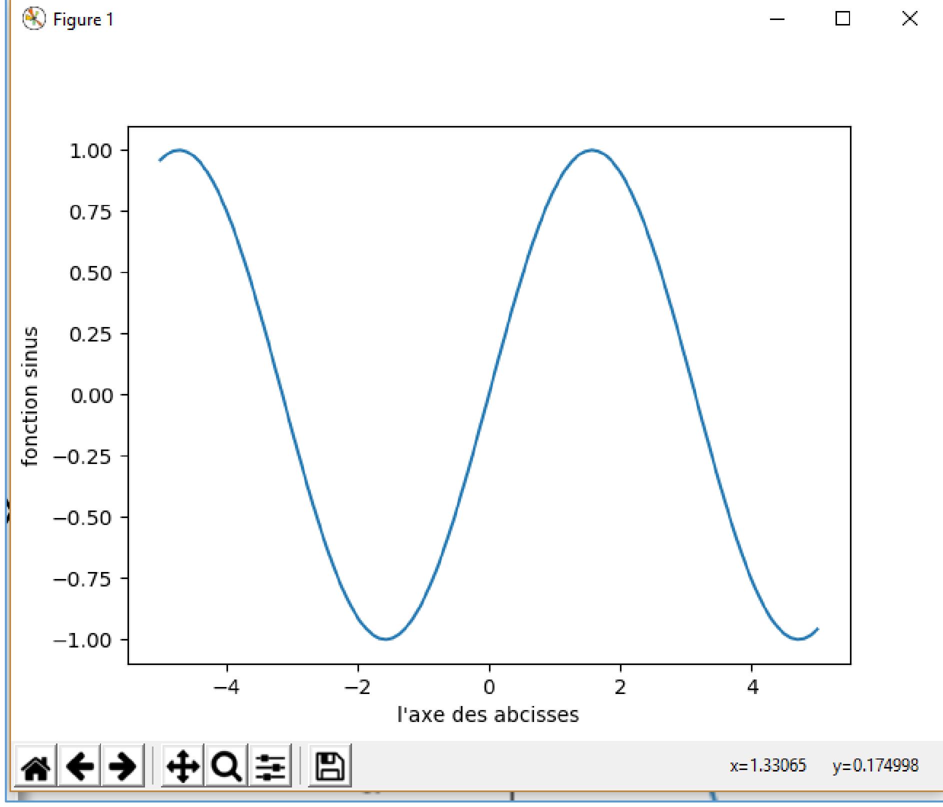
8.3.1. Définition

Le module matplotlib est chargé de tracer les courbes :

```
>>> import matplotlib.pyplot as plt
>>> |
```

D'une manière générale les fonctions plt.plot attendent des vecteur/matrice, bref des tableaux de points du plan. Selon les options, ces points du plan sont reliés entre eux de façon ordonnée par des segments : le résultat est une courbe. Commençons par la fonction sinus.

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> x=np.linspace(-5,5,100)
>>> plt.plot(x,np.sin(x)) # on utilise la fonction sinus de Numpy
[<matplotlib.lines.Line2D object at 0x000001CAD0DE75C0>]
>>> plt.ylabel('fonction sinus')
Text(0, 0.5, 'fonction sinus')
>>> plt.xlabel("l'axe des abcisses")
Text(0.5, 0, "l'axe des abcisses")
>>> plt.show()
```



Si tout se passe bien, une fenêtre doit s'ouvrir avec la figure ci-dessus. Il est possible de jouer avec les menus dans le bas de cette fenêtre : zoomer, déplacer la figure, etc et surtout sauvegarder dans un format PNG, PDF, EPS, etc.
plt.clf() : effacе la fenêtre graphique

plt.savefig()
Sauvegarde le graphique. Par exemple plt.savefig("mongraphe.png") sauve sous le nom "mongraphe.png" le graphique. Par défaut le format est PNG. Il est possible d'augmenter la résolution, la couleur de fond, l'orientation, la taille (a0, a1, lettertype, etc) et aussi le format de l'image. Si aucun format n'est spécifié, le format est celui de l'extension dans "nomfigure.ext" (où "ext" est "eps", "png", "pdf", "ps" ou "svg"). Il est toujours conseillé de mettre une extension aux noms de fichier ; si vous y tenez plt.savefig('toto',format='pdf') sauvegarder l'image sous le nom "toto" (sans extension !) au format "pdf".

8.3.2. Des détails

Pour connaître toutes les options, le mieux est de se référer à la documentation de matplotlib. Voyons ici quelques-unes d'entre elles

- **Bornes :** spécifier un rectangle de représentation, ce qui permet un zoom, d'éviter les grandes valeurs des fonctions par exemple, se fait via la commande `plt.axis([xmin,xmax,ymin,ymax])`

- **Couleur du trait :** pour changer la couleur du tracé une lettre g vert (green), r rouge (red), k noir, b bleu, c cyan, m magenta, y jaune (yellow), wblanc (white). `plt.plot(np.sin(x),'r')` tracera notre courbe sinus en rouge. Les amateurs de gris sont servis via `color='(un flottant entre 0 et1)'`. Enfin pour avoir encore plus de couleurs, comme en HTML la séquence `color='#eeefff'` donnera la couleur attendu et les amateurs de RGB sont servis par `color=( R, G, B)` avec trois paramètres compris entre 0 et 1 (RGBA est possible aussi).

- **Symboles :** mettre des symboles aux points tracés se fait via l'option `marker`. Les possibilités sont nombreuses parmi
  ['+' | '*' | ',' | '.' |'1' | '2' | '3' | '4' | '<' | '>' | 'D' | 'H' | '^' | '_' | 'd' | 'h' | 'o' | 'p' | 's' | 'v' | 'x' | '|' | TICKUP | TICKDOWN | TICKLEFT |TICKRIGHT | 'None' | ' ' | ''].

- **Style du trait :** pointillés, absences de trait, etc se décident avec `linestyle`. Au choix '-' ligne continue, '--' tirets, '-.' points-tirets, ':'pointillés, sachant que 'None', '', ' ' donnent "rien-du-tout". Plutôt que `linestyle`, `ls` (plus court) fait le même travail.

- **Épaisseur du trait :** `linewidth=flottant (comme linewidth=2)` donne un trait, pointillé (tout ce qui est défini par style du trait) d'épaiseur "flottant" en points. Il est possible d'utiliser *lw* en lieu et place de *linewidth*.

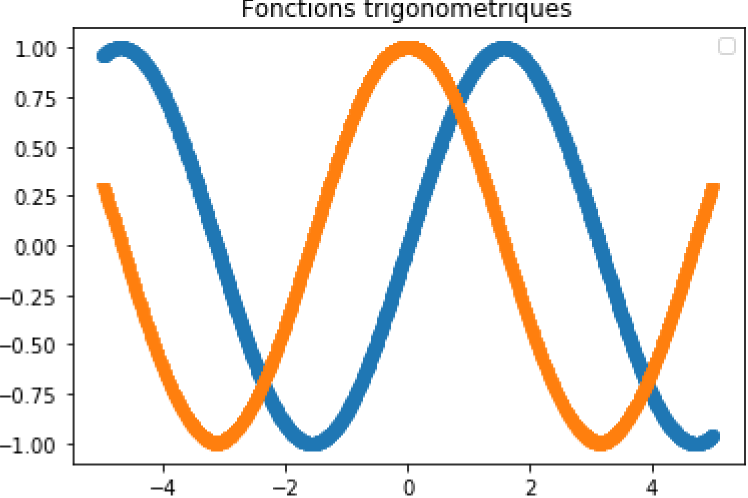
- **Taille des symboles (markers) :** *markersize=flottant* comme pour l'épaisseur du trait. D'autres paramètres sont modifiables *markeredgecolor* la couleur du trait du pourtour du *marker*, markerfacecolor la couleur de l'intérieur (si le marker possède un intérieur comme 'o'), *markeredgsize=flottant* l'épaisseur du trait du pourtour du marker. Remarquez que si la couleur n'est pas spécifiée pour chaque nouvel appel la couleur des "markers" change de façon cyclique.

- **Étiquettes** sur l'axe des abscisses/ordonnées : Matplotlib décide tout seul des graduations sur les axes. Tout ceci se modifie via `plt.xticks(tf)`, `plt.yticks(tl)` où tf est un vecteur de flottants ordonnés de façon croissante.

- **Ajouter un titre :** plt.title("Mon titre")

- **Légendes :** c'est un peu plus compliqué. D'après ce que j'ai compris il faut assigner à des variables le tracé, via *g1=plt.plot()*, etc. Enfin *plt.legend((g1, g2),* ("ligne 2","ligne 1")) fait le boulot. Par exemple

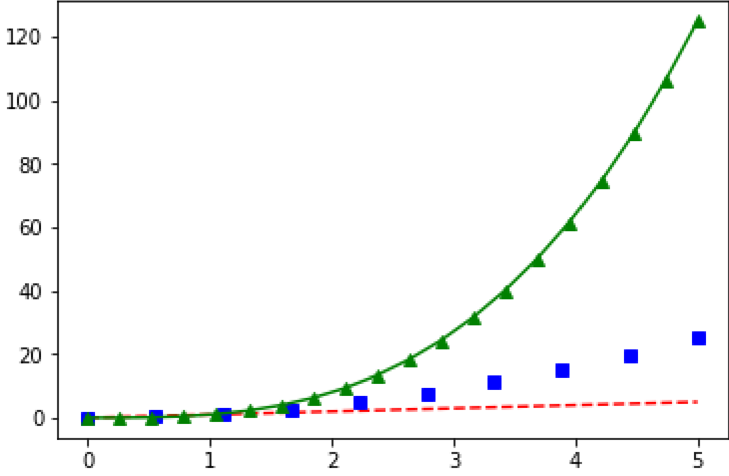
```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 x=np.linspace(-5,5,100)
4 p1=plt.plot(x,np.sin(x),marker='o')
5 p2=plt.plot(x,np.cos(x),marker='v')
6 plt.title("Fonctions trigonometriques")#Problemes avec accents (plot_directive)
7 plt.legend()
8 plt.show()
```



8.3.3. Quelques exemples

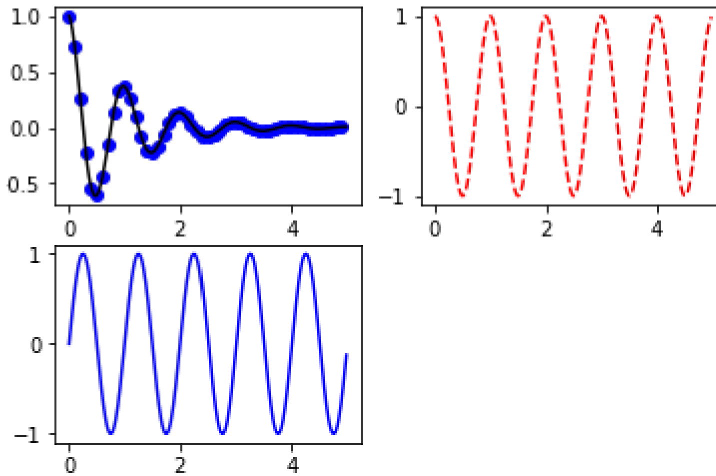
Pour superposer plusieurs graphes de fonctions, il est possible de faire une succession de commandes `plt.plot` ou encore en une seule commande. Remarquez aussi que pour des choses simples il est possible de se passer des `ls`, `color` et `marker`.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 t1=np.linspace(0,5,10)
4 t2=np.linspace(0,5,20)
5 plt.plot(t1, t1, 'r--', t1, t1**2, 'bs', t2, t2**3, 'g^-')
6
```



Enfin un système de sous-figures permet de juxtaposer différents graphiques

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(t):
5     return np.exp(-t) * np.cos(2*np.pi*t)
6
7 t1 = np.arange(0.0, 5.0, 0.1)
8 t2 = np.arange(0.0, 5.0, 0.02)
9 plt.figure(1)
10 plt.subplot(221)
11 plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
12 plt.subplot(222)
13 plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
14 plt.subplot(223)
15 plt.plot(t2, np.sin(2*np.pi*t2), 'b-')
16
```

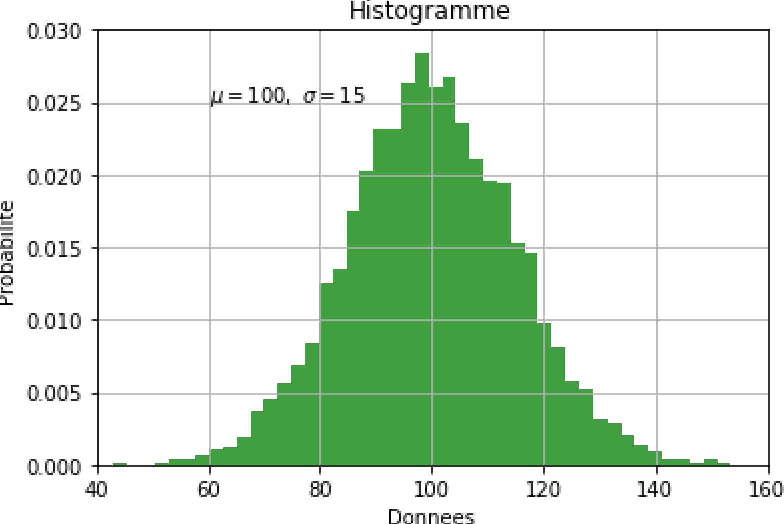


Dans la commande plt.subplot l'argument est nbre de lignes, nbre de colonnes, numéro de la figure. Il y a une condition à respecter : le nombre de lignes multiplié par le nombre de colonnes est supérieur ou égal au nombre de figure. Ensuite Matplotlib place les figures au fur et à mesure dans le sens des lignes.

Dans le même registre, ouvrir plusieurs fenêtres graphiques est possible. Si vous avez déjà une fenêtre graphique, la commande `plt.figure(2)` en ouvre une seconde et les instructions `plt.plot` qui suivent s'adresseront à cette seconde figure. Pour revenir et modifier la première fenêtre graphique, `plt.figure(1)` suffit.

Pour terminer, un histogramme et un affichage de texte sur le graphique

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 mu, sigma = 100, 15
5 x = mu + sigma * np.random.randn(10000)
6 #  histogramme des donn\'es
7 n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)
8 plt.xlabel('Donn\'ees')
9 plt.ylabel('Probabilite')
10 plt.title('Histogramme')
11 plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
12 plt.axis([40, 160, 0, 0.03])
13 plt.grid(True)
14
```



Source: http://math.mad.free.fr/depot/numpy/courbe.html

8.4. Scientific Python (Scipy)

SciPy est un projet visant à unifier et fédérer un ensemble de bibliothèques Python à usage scientifique. Scipy utilise les tableaux et matrices du module NumPy.

il contient par exemple des modules pour l'optimisation, l'algèbre linéaire, les statistiques, le traitement du signal ou encore le traitement d'images.
Il offre également des possibilités avancées de visualisation grâce au module matplotlib.

Voici quelques liens très utiles :

- Fonctions Spéciales (scipy.special (http://docs.scipy.org/doc/scipy/reference/special.html))
- Intégration (scipy.integrate (http://docs.scipy.org/doc/scipy/reference/integrate.html))
- Optimisation (scipy.optimize (http://docs.scipy.org/doc/scipy/reference/optimize.html))
- Interpolation (scipy.interpolate (http://docs.scipy.org/doc/scipy/reference/interpolate.html))
- Transformées de Fourier (scipy.fftpack (http://docs.scipy.org/doc/scipy/reference/fftpack.html))

- Traitement du Signal (scipy.signal (http://docs.scipy.org/doc/scipy/reference/signal.html))
- Algèbre Linéaire (scipy.linalg (http://docs.scipy.org/doc/scipy/reference/linalg.html))
- Matrices Sparses et Algèbre Linéaire Sparse (scipy.sparse (http://docs.scipy.org/doc/scipy/reference/sparse.html))
- Statistiques (scipy.stats (http://docs.scipy.org/doc/scipy/reference/stats.html))
- Traitement d'images N-dimensionelles (scipy.ndimage (http://docs.scipy.org/doc/scipy/reference/ndimage.html))
- Lecture/Ecriture Fichiers IO (scipy.io (http://docs.scipy.org/doc/scipy/reference/io.html))

Durant ce cours on abordera certains de ces modules.
Pour utiliser un module de SciPy dans un programme Python il faut commencer par l'importer. Si vous utilisez Spyder pas de problème ; dans le cas contraire il faut l'installer d'abord dans votre environnement de travail.
Voici un exemple avec le module linalg

```
>>> from scipy import linalg
>>> |
```

On aura besoin de NumPy:

```
>>> import numpy as np
>>> |
```

Et de matplotlib/pylab:

```
>>> import matplotlib.pyplot as plt
>>> |
```

8.4.1. Fonctions Spéciales

Un grand nombre de fonctions importantes, notamment en physique, sont disponibles dans le module scipy.special
Pour plus de détails: http://docs.scipy.org/doc/scipy/reference/special.html#module scipy.special (http://docs.scipy.org/doc/scipy/reference/special.html#module-scipy.special).

Un exemple avec les fonctions de Bessel:

En mathématiques, et plus précisément en analyse, les fonctions de Bessel, découvertes par le mathématicien suisse Daniel Bernoulli, portent le nom du mathématicien allemand Friedrich Wilhelm Bessel. Bessel développa l'analyse de ces fonctions en 1816 dans le cadre de ses études du mouvement des planètes induit par l'interaction gravitationnelle, généralisant les découvertes antérieures de Bernoulli. Ces fonctions sont des solutions canoniques y(x) de l'équation différentielle de Bessel :

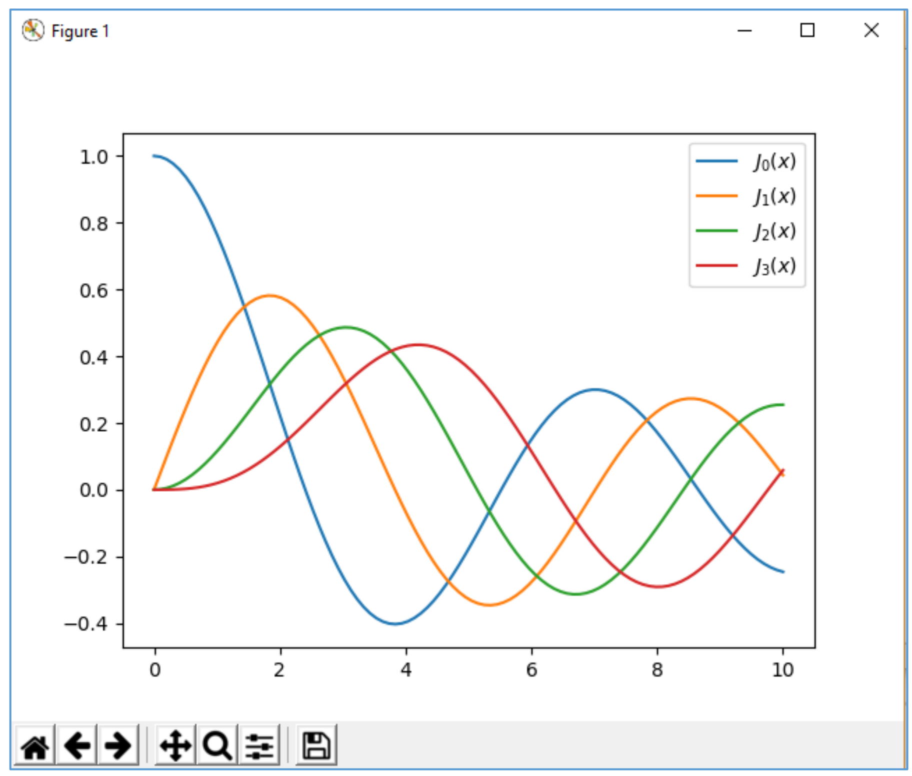
$$x^2 \frac{\mathrm{d}^2 y}{\mathrm{d}x^2} + x \frac{\mathrm{d}y}{\mathrm{d}x} + (x^2 - \alpha^2) y = 0$$

Pour les valeurs entières de α=n, les fonctions de Bessel de première espèce J_n sont définies par la série entière (de rayon de convergence infini) suivante :

$$J_n(x) = \sum_{p=0}^{\infty} \frac{(-1)^p}{p!(n+p)!} \left(\frac{x}{2}\right)^{2p+n}$$

```
>>> # jn : Bessel de premier type
>>> # yn : Bessel de deuxième type
>>> from scipy.special import jn, yn
>>> n = 0 # ordre
>>> x = 0.0
>>> # Bessel de premier type
>>> print("J_%d(%s) = %f" % (n, x, jn(n, x)))
J_0(0.0) = 1.000000
>>> x = 1.0
>>> # Bessel de deuxième type
>>> print("Y_%d(%s) = %f" % (n, x, yn(n, x)))
Y_0(1.0) = 0.088257
>>>
```

```
>>> x = np.linspace(0, 10, 100)
>>> for n in range(4):
        plt.plot(x, jn(n, x), label=r"$J_%d(x)$" % n)


[<matplotlib.lines.Line2D object at 0x000001B01C290D30>]
[<matplotlib.lines.Line2D object at 0x000001B01C290EF0>]
[<matplotlib.lines.Line2D object at 0x000001B01C2982B0>]
[<matplotlib.lines.Line2D object at 0x000001B01C2985C0>]
>>> plt.legend()
<matplotlib.legend.Legend object at 0x000001B01C298B38>
>>> plt.show()
```



8.4.2. Intégration

- **Intégration numérique**

L'évaluation numérique de : $\int_a^b f(x)dx$

est nommée *quadrature (abbr. quad)*. SciPy fournit différentes fonctions : par exemple `quad`, `dblquad` et `tplquad` pour les intégrales simples, doubles ou triples.

```
>>> from scipy.integrate import quad, dblquad, tplquad
>>> # soit une fonction f
>>> def f(x):
        return x

>>> a, b = 1, 2 # intégrale entre a et b
>>> val, abserr = quad(f, a, b)
>>> print("intégrale =", val, ", erreur =", abserr)
intégrale = 1.5 , erreur = 1.6653345369377348e-14
>>> |
```

- **EXERCICE :** Intégrer la fonction de Bessel j_n d'ordre 3 entre 0 et 10

```
>>> def J3(x):
        return jn(3,x)

>>> val, abserr = quad(J3, 0, 10)
>>> print("intégrale =", val, ", erreur =", abserr)
intégrale = 0.7366751370811073 , erreur = 9.389126882496413e-13
>>> |
```

- **Exemple intégrale double :**

$$\int_{x=1}^{2} \int_{y=1}^{x} (x + y^2)dxdy$$

```
>>> def f(x,y):
        return x + y**2

>>> def gfun(x):
        return 1

>>> def hfun(x):
        return x

>>> print(dblquad(f, 1, 2, gfun, hfun))
(2.083333333333333, 6.082147827281036e-14)
>>> |
```

8.4.3. Equations différentielles ordinaires (EDO)

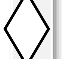
SciPy fournit deux façons de résoudre les EDO: Une API basée sur la fonction odeint, et une API orientée-objet basée sur la classe ode. odeint est plus simple pour commencer.
Commençons par l'importer :

```
>>> from scipy.integrate import odeint
>>> |
```

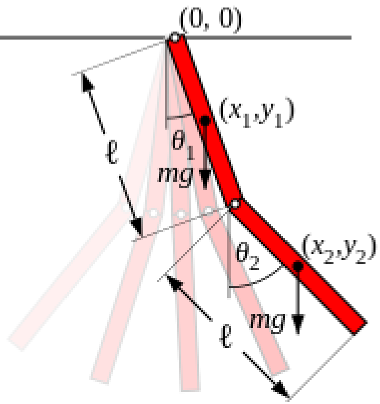
Un système d'EDO se formule de la façon standard :
$y' = f(y, t)$ avec $y = [\,(y_1(t), (y_2(t), \ldots, (y_n(t)]$
et est une fonction qui fournit les dérivées des fonctions . Pour résoudre une EDO il faut spécifier et les conditions initiales, .

Une fois définies, on peut utiliser odeint: y_t = odeint(f, y_0, t) où t est un NumPy array des coordonnées en temps où résoudre l'EDO. y_t est un array avec une ligne pour chaque point du temps t, et chaque colonne correspond à la solution y_i(t) à chaque point du temps.

- **Exemple : double pendule**



Description: http://en.wikipedia.org/wiki/Double_pendulum
(http://en.wikipedia.org/wiki/Double_pendulum)



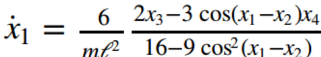
Les équations du mouvement du pendule sont données sur la page Wikipédia :

$$\dot{\theta}_1 = \frac{6}{ml^2}\frac{2p_{\theta_1} - 3\cos(\theta_1-\theta_2)p_{\theta_2}}{16-9\cos^2(\theta_1-\theta_2)}$$
$$\dot{\theta}_2 = \frac{6}{ml^2}\frac{8p_{\theta_2} - 3\cos(\theta_1-\theta_2)p_{\theta_1}}{16-9\cos^2(\theta_1-\theta_2)}.$$

Les équations de mouvement restantes sont écrites comme suit :

$$\dot{p}_{\theta_1} = \frac{\partial L}{\partial \theta_1} = -\tfrac{1}{2}ml^2\left(\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2) + 3\frac{g}{l}\sin\theta_1\right)$$
$$\dot{p}_{\theta_2} = \frac{\partial L}{\partial \theta_2} = -\tfrac{1}{2}ml^2\left(-\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2) + \frac{g}{l}\sin\theta_2\right)$$

où les p_{θ_i} sont les moments d'inertie. Pour simplifier le code Python, on peut introduire la variable $x = [\theta_1, \theta_2, p_{\theta_1}, p_{\theta_2}]$



$$\dot{x}_1 = \frac{6}{m\ell^2}\frac{2x_3 - 3\cos(x_1 - x_2)x_4}{16 - 9\cos^2(x_1 - x_2)}$$

$$\dot{x}_2 = \frac{6}{m\ell^2}\frac{8x_4 - 3\cos(x_1 - x_2)x_3}{16 - 9\cos^2(x_1 - x_2)}$$

$$\dot{x}_3 = -\frac{1}{2}m\ell^2\left[\dot{x}_1\dot{x}_2\sin(x_1 - x_2) + 3\frac{g}{\ell}\sin x_1\right]$$

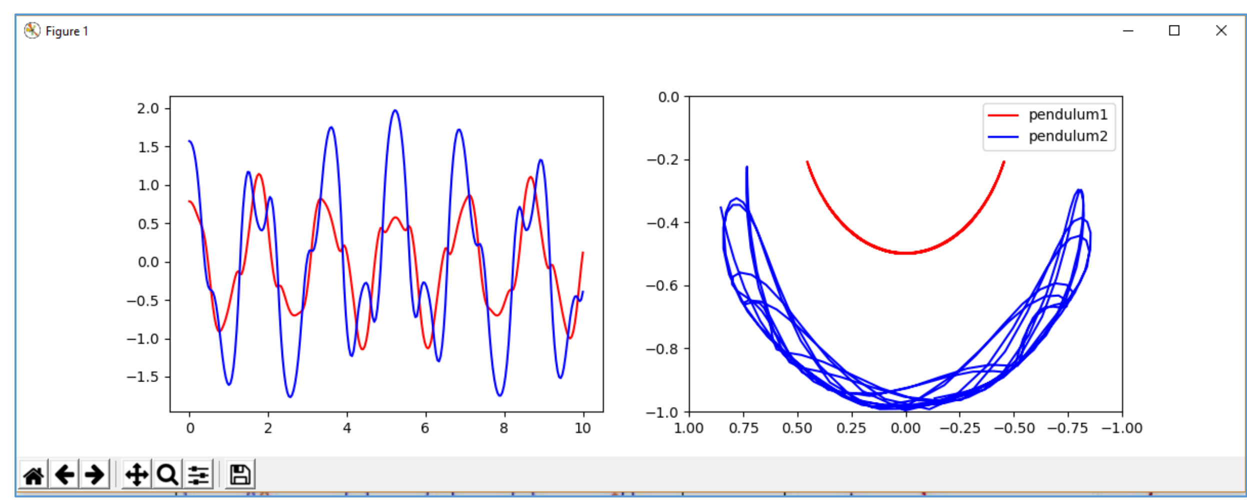
$$\dot{x}_4 = -\frac{1}{2}m\ell^2\left[-\dot{x}_1\dot{x}_2\sin(x_1 - x_2) + \frac{g}{\ell}\sin x_2\right]$$

```
>>> g = 9.82
>>> L = 0.5
>>> m = 0.1
>>> def dx(x, t):
        "" "Le côté droit du pendule ODE" ""
        x1, x2, x3, x4 = x[0], x[1], x[2], x[3]
        dx1 = 6.0/(m*L**2) * (2 * x3 - 3 * np.cos(x1-x2) * x4)/(16 - 9* np.cos(x1-x2)**2)
        dx2 = 6.0/(m*L**2) * (8 * x4 - 3 * np.cos(x1-x2) * x3)/(16 - 9* np.cos(x1-x2)**2)
        dx3 = -0.5 * m * L**2 * ( dx1 * dx2 * np.sin(x1-x2) + 3 * (g/L)* np.sin(x1))
        dx4 = -0.5 * m * L**2 * (-dx1 * dx2 * np.sin(x1-x2) + (g/L) * np.sin(x2))
        return [dx1, dx2, dx3, dx4]

>>> # on choisit une condition initiale
>>> x0 = [np.pi/4, np.pi/2, 0, 0]
>>> # les instants du temps: de 0 à 10 secondes
>>> t = np.linspace(0, 10, 250)
>>> 
```

```
>>> # On résout
>>> x = odeint(dx, x0, t)
```

```
>>> print(x.shape)
(250, 4)
```

```
>>> # affichage des angles en fonction du temps
>>> fig, axes = plt.subplots(1,2, figsize=(12,4))
>>> axes[0].plot(t, x[:, 0], 'r', label="theta1")
[<matplotlib.lines.Line2D object at 0x000001B01D2015C0>]
>>> axes[0].plot(t, x[:, 1], 'b', label="theta2")
[<matplotlib.lines.Line2D object at 0x000001B01D201588>]
>>> x1 = + L * np.sin(x[:, 0])
>>> y1 = - L * np.cos(x[:, 0])
>>> x2 = x1 + L * np.sin(x[:, 1])
>>> y2 = y1 - L * np.cos(x[:, 1])
>>> axes[1].plot(x1, y1, 'r', label="pendulum1")
[<matplotlib.lines.Line2D object at 0x000001B01D201710>]
>>> axes[1].plot(x2, y2, 'b', label="pendulum2")
[<matplotlib.lines.Line2D object at 0x000001B01D201A20>]
>>> axes[1].set_ylim([-1, 0])
(-1, 0)
>>> axes[1].set_xlim([1, -1])
(1, -1)
>>> plt.legend()
<matplotlib.legend.Legend object at 0x000001B01D201F28>
>>> plt.show()
```



8.4.4. Optimisation

- **Objectif :** trouver les minima ou maxima d'une fonction

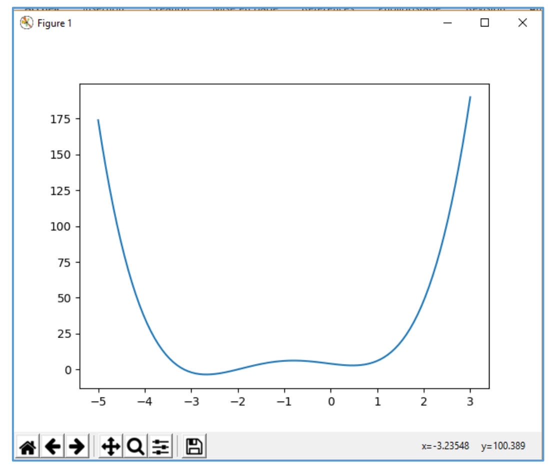
Doc : http://scipy-lectures.github.com/advanced/mathematical_optimization/index.html (http://scipylectures.github.com/advanced/mathematical_optimization/index.html)

On commence par l'import

```
>>> from scipy import optimize
>>> |
```

- **Trouver un minimum**

```
>>> from scipy import optimize
>>> def f(x):
        return 4*x**3 + (x-2)**2 + x**4

>>> x = np.linspace(-5, 3, 100)
>>> plt.plot(x, f(x))
[<matplotlib.lines.Line2D object at 0x000001B01C335DA0>]
>>> plt.show()
```



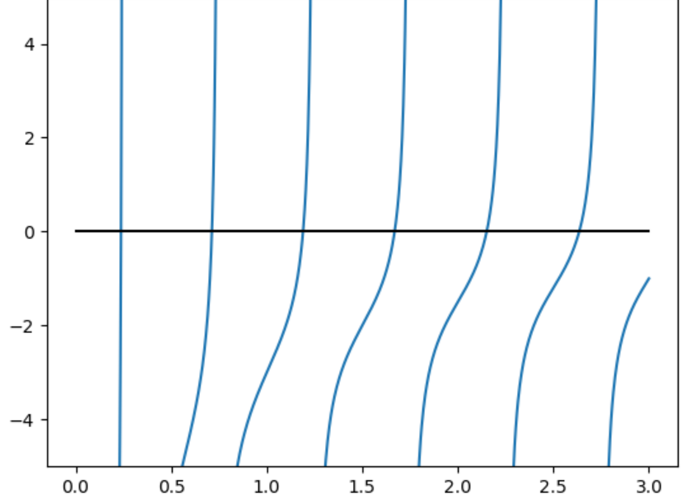
Nous allons utiliser la fonction `fmin_bfgs`:

```
>>> x_min = optimize.fmin_bfgs(f, x0=-3)
Optimization terminated successfully.
         Current function value: -3.506641
         Iterations: 4
         Function evaluations: 18
         Gradient evaluations: 6
>>> x_min
array([-2.67298161])
>>> |
```

- **Trouver les zéros d'une fonction**

Trouver tel que f(x) = 0. On va utiliser `fsolve`.

```
>>> np.unique(
(optimize.fsolve(f, np.linspace(0.2, 3, 40))*1000).astype(int)
) / 1000
array([0.237, 0.712, 1.189, 1.669, 2.15 , 2.635, 3.121])
>>> optimize.fsolve(f, 0.72)
array([0.71286972])
>>> optimize.fsolve(f, 1.1)
array([1.18990285])
>>> |
```



Pour aller plus loin

- http://www.scipy.org (http://www.scipy.org) - The official web page for the SciPy project
- http://docs.scipy.org/doc/scipy/reference/tutorial/index.html
- (http://docs.scipy.org/doc/scipy/reference/tutorial/index.html) - A tutorial on how to get started using SciPy.
- https://github.com/scipy/scipy/ (https://github.com/scipy/scipy/) - The SciPy source code.
- http://scipy-lectures.github.io (http://scipy-lectures.github.io)

CHAPITRE 9 : Approfondissements en Python

9.1. Les exceptions

Exécuter ce script ci-dessous :

```
>>> 9/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    9/0
ZeroDivisionError: division by zero
```

On remarque tout d'abord que python est certes sévère mais juste ; il nous dit pourquoi il n'est pas content: `ZeroDivisionError`.
Cet exemple est sans intérêt mais il est tout à fait possible d'utiliser une variable comme dénominateur et à ce moment-là comment éviter cette erreur ?
Une solution serait de vérifier la valeur de la variable et si elle est égale à 0, on annule tout.
Une autre solution serait d'anticiper qu'il serait possible qu'il y ait une erreur et en cas d'erreur prévoir des instructions spécifiques.

9.1.1. Try except

Try signifie "essayer" en anglais, ce mot clé permet d'essayer une action et si l'action échoue on peut lui donner d'autres instructions dans un bloc except.
Alors pourquoi utiliser try ?
Et bien ce genre d'erreur est bloquante, c'est-à-dire que si les instructions sont exécutées dans un script, le script s'arrête et cela devient un bug.

```
>>> try:
        w/v
        print("Ok pas d'erreur")
except:
        print("Erreur")


Erreur
>>> |
```

9.1.2. Cibler les erreurs

La syntaxe exposée plus haut répond à tout type d'erreur, mais il est possible d'affiner la gestion d'erreur.
Par exemple que se passe-t-il si nous divisons un nombre par des lettres ?

```
>>> 45/"MPSI"
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    45/"MPSI"
TypeError: unsupported operand type(s) for /: 'int' and 'str'
>>> |
```

On remarque que python nous affiche une erreur mais elle est différente de celle provoquée par la division par 0. Ici l'erreur est `TypeError`.

Il est possible en python d'exécuter des instructions en fonction du type d'erreur. Par exemple si la valeur est 0 on aimerait afficher un message et si le dénominateur est du texte on aimerait pouvoir afficher un autre message à l'utilisateur.

```
>>> w,v = 5,0
>>> try:
	w/v
	print("Ok pas d'erreur")
except TypeError:
	print("Merci d'utiliser des chiffres")
except ZeroDivisionError:
	print("Merci de ne pas utiliser le 0")


Merci de ne pas utiliser le 0
>>> |
```

Et dans le cas où la variable v vaut "Bio 1":

```
>>> w,v = 5,"Bio 1"
>>> try:
	w/v
	print("Ok pas d'erreur")
except TypeError:
	print("Merci d'utiliser des chiffres")
except ZeroDivisionError:
	print("Merci de ne pas utiliser le 0")


Merci d'utiliser des chiffres
>>> |
```

9.1.3. Finally

On utilise le mot clé finally pour exécuter des instructions quelque soit les erreurs générées ou non (et même s'il y a présence d'un return). Dans tous les cas les instructions placées dans finally seront exécutées.

```
>>> try:
	pass
except:
	pass
finally:
	print("Execution")


Execution
>>> |
```

9.2. Edition d'un fichier

Une manière de stocker des données de manière pérenne est de les stocker dans des fichiers.

9.2.1. Editer un fichier

Pour éditer un fichier en python on utilise la fonction `open`.
Cette fonction prend en premier paramètre le chemin du fichier (relatif ou absolu) et en second paramètre le type d'ouverture.

- **Chemin relatif / chemin absolu**

Un chemin relatif en informatique est un chemin qui prend en compte l'emplacement de lecture.
Un chemin absolu est un chemin complet qui peut être lu quel que soit l'emplacement de lecture.

9.2.2. La fonction open()

Voici la syntaxe pour lire un fichier : open("chemin absolu du fichier", "type d'ouverture")

```
>>> fichier = open("data.txt", "r")
>>> print(fichier)
<_io.TextIOWrapper name='data.txt' mode='r' encoding='cp1252'>
>>> |
```

On remarque que le deuxième paramètre est renseigné par un r, ce paramètre indique une ouverture de fichier en lecture.

9.2.3. Les types d'ouverture

Il existe plusieurs modes d'ouverture :

```
r, pour une ouverture en lecture (READ).

w, pour une ouverture en écriture (WRITE), à chaque ouverture le contenu du fichier
est écrasé. Si le fichier n'existe pas python le crée.

a, pour une ouverture en mode ajout à la fin du fichier (APPEND). Si le fichier n'e
xiste pas python le crée.

b, pour une ouverture en mode binaire.

t, pour une ouverture en mode texte.

x, crée un nouveau fichier et l'ouvre pour écriture
```

9.2.4. Fermeture d'un fichier

Comme tout élément ouvert, il faut le refermer une fois les instructions terminées. Pour cela on utilise la méthode `close()`.

```
>>> fichier.close()
>>> |
```

9.2.5. Lire le contenu d'un fichier

Pour afficher tout le contenu d'un fichier, vous pouvez utiliser la méthode read sur l'objet-fichier.

```
>>> fichier = open("data.txt", "r")
>>> print(fichier.read())
Tous les hommes vont comparaitre en ce jour-là devant le grand trône !
>>> fichier.close()
>>> |
```

9.2.6. Ecrire dans un fichier

Voici la syntaxe pour écrire dans un fichier :

```
>>> fichier = open("data.txt", "a")
>>> fichier.write("\nLes pécheurs non repentis ne veront pas le Seigneur !")
54
>>> fichier.close()
>>> |
```

A noter que pour le monde d'ouverture **a**, si vous voulez écrire à la ligne, vous pouvez utiliser le saut de ligne **\n**

9.2.7. Le mot clé with

Il existe une autre syntaxe plus courte qui permet de s'émanciper du problème de fermeture du fichier : le mot clé with.
Voici la syntaxe :

```
>>> with open("data.txt", "r") as fichier:
        print(fichier.read())


Tous les hommes vont comparaitre en ce jour-là devant le grand trône !
Les pécheurs non repentis ne veront pas le Seigneur !
>>> |
```

9.3. Application

9.3.1. Concernant le répertoire

```
>>> from os import chdir
>>> chdir("C:")
>>> # chdir = change directory quand le chemin existe
>>> rep_cour = getcwd()
>>> print(rep_cour)
C:\
>>> |
```

getcwd() permet de récupérer le chemin courant.

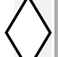
9.3.2. Application 1

Ecrire une fonction `copieFichier` qui permet de copier le contenu d'un fichier source dans le un fichier destinataire. `copieFichier(fsource, fdestination)`

```
>>> def copieFichier(fsource, fdestination):
        "copie intégrale d'un fichier"
        fs = open(fsource, 'r')
        fd = open(fdestination, 'w')
        while 1:
                txt = fs.read(50)
                if txt =="":
                        break
                fd.write(txt)
        fs.close()
        fd.close()
        return None
```

Tester sur le fichier discours.txt

9.3.3. Application 2

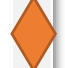


Le script qui suit vous montre comment créer une fonction destinée à effectuer un certain traitement sur un fichier texte. En l'occurrence, il s'agit ici de recopier un fichier texte, en omettant toutes les lignes qui commencent par un caractère `'#'`. Ecrivez cette fonction.

NB : au niveau de la méthode `read()` on peut également utiliser `readline()` est une méthode qui renvoie une chaîne de caractères, alors que la méthode `readlines()` renvoie une liste. À la fin du fichier, `readline()` renvoie une chaîne vide, tandis que `readlines()` renvoie une liste vide.

```
1 def filtre(source,destination):
2     "recopier un fichier en éliminant les lignes de remarques"
3     fs = open(source, 'r')
4     fd = open(destination, 'w')
5     while 1:
6         txt = fs.readline()
7         if txt =='':
8             break
9         if txt[0] != '#':
10            fd.write(txt)
11    fs.close()
12    fd.close()
```

Tester sur le fichier electionPr2018.txt



9.3.4. Les exceptons et les fichiers

Considérons par exemple un script qui demande à l'utilisateur d'entrer un nom de fichier, lequel fichier étant destiné à être ouvert en lecture. Si le fichier n'existe pas, nous ne voulons pas que le programme se « plante ». Nous voulons qu'un avertissement soit affiché, et éventuellement que l'utilisateur puisse essayer d'entrer un autre nom.

```
1 filename = input("Veuillez entrer un nom de fichier : ")
2 try:
3     f = open(filename, "r")
4 except:
5     print("Le fichier", filename, "est introuvable")
6
```

Si nous estimons que ce genre de test est susceptible de rendre service à plusieurs endroits d'un programme, nous pouvons aussi l'inclure dans une fonction :

```
 1 def existe(fname):
 2     try:
 3         f = open(fname,'r')
 4         f.close()
 5         return 1
 6     except:
 7         return 0
 8
 9
10 filename = input("Veuillez entrer le nom du fichier : ")
11 if existe(filename):
12     print("Ce fichier existe bel et bien.")
13 else:
14     print("Le fichier", filename, "est introuvable.")
15
```

9.3.5. Application 3

Écrivez un script qui permette de créer et de relire aisément un fichier texte. Votre programme demandera d'abord à l'utilisateur d'entrer le nom du fichier. Ensuite il lui proposera le choix, soit d'enregistrer de nouvelles lignes de texte, soit d'afficher le contenu du fichier. L'utilisateur devra pouvoir entrer ses lignes de texte successives en utilisant simplement la touche <Enter> pour les séparer les unes des autres. Pour terminer les entrées, il lui suffira d'entrer une ligne vide (c'est-à-dire utiliser la touche <Enter> seule). L'affichage du contenu devra montrer les lignes du fichier séparées les unes des autres de la manière la plus naturelle (les codes de fin de ligne ne doivent pas apparaître)

```
1  def sansDC(ch):
2      "cette fonction renvoie la chaîne ch amputée de son dernier caractère"
3      nouv = ""
4      i, j = 0, len(ch) -1
5      while i < j:
6          nouv = nouv + ch[i]
7          i = i + 1
8      return nouv
9
10 def ecrireDansFichier():
11     of = open(nomF, 'a')
12     while 1:
13         ligne = input("entrez une ligne de texte (ou <Enter>) : ")
14         if ligne == '':
15             break
16         else:
17             of.write(ligne + '\n')
18     of.close()
19
20 def lireDansFichier():
21     of = open(nomF, 'r')
22     while 1:
23         ligne = of.readline()
24         if ligne == "":
25             break
26         # afficher en omettant le dernier caractère (= fin de ligne) :
27         print(sansDC(ligne))
28     of.close()
29
30 nomF = input('Nom du fichier à traiter : ')
31 choix = input('Entrez "e" pour écrire, "c" pour consulter les données : ')
32 if choix =='e':
33     ecrireDansFichier()
34 else:
35     lireDansFichier()
```

9.3.6. TDn°7 – Gestion des exceptions, manipulation des fichiers textes

9.3.7. TPn°8 – Gestion des exceptions, manipulation des fichiers textes