Python pour la physique

Version 2019-2020

Pierre Cladé

oct. 30, 2019

Table des matières

CHAPITRE 1

Prise en main

1.1 Qu'est ce que Python

L'objectif d'un langage informatique est de décrire un ensemble d'instructions que l'on souhaite voir exécuter par un ordinateur. Le langage que comprend l'ordinateur (l'assembleur) est un langage trop primitif pour être simple et rapide à utiliser. C'est pour cela qu'il existe des langages dits évolués.

Parmi ces langages on peut distinguer les langages compilés (par exemple, le C, le Fortran, le C++, ...) et les langages interprétés (basic, python, matlab, scilab, ...). Dans le premier cas, le programme est entièrement traduit (compilé) pour être exécuté directement par le processeur (on peut prendre l'analogie d'un livre traduit entièrement dans une autre langue). Les langages interprétés sont traduits au fur et à mesure par un interpréteur (pour continuer l'analogie, il s'agit d'un traducteur qui traduit les phrases au fur et à mesure). Les langages interprétés ont la réputations d'être plus lent, cependant, il sont bien plus facile à utiliser : on peut faire une modification sans avoir à recompiler, ou simplement voir le résultat de chaque instruction au fur et à mesure de l'exécution.

Le langage Python est un langage interprété. Il a été créé en 1995 avec pour objectif d'être un langage informatique général (par opposition aux langages spécialisés pour effectuer un type de tache bien précis). Python fait parti des dix langages informatiques les plus utilisés au monde. Python est devenu au cours des années un langage interprété de premier choix. Il est présent dans beaucoup de projet open source (par exemple LibreOffice).

1.2 Installer Python

1.2.1 L'interpréteur Python

Il existe plusieurs interpréteurs pour Python. Citons principalement CPython, qui est l'interpréteur de référence écrit en C, JPython (écrit en Java), pypy (un interpréteur plus rapide, partiellement écrit en Python !), IronPython (écrit en C#). Il existe aussi des interpréteurs Python pour programmer des microprocesseur (python-on-chip, pymite, MicroPython <https://micropython.org/>). Ces derniers ne peuvent interpréter qu'un partie du langage Python. Nous utiliserons CPython.

Python est un langage qui évolue au cours des années. La définition du langage est faite de facto par l'interpréteur standard CPython et on assimile la version de langage à celle de cette interpréteur. Actuellement (en 2019), il existe deux versions principales de Python couramment utilisées, la version 2 et la version 3. La version 3 apporte des changements incompatibles avec la version 2, c'est pour cela que les développeurs de Python on continué à développer la version 2 en parallèle de la version 3.

Nous avons choisi d'utiliser dans ce cours la version 3 de Python. La version 2 étant toujours très utilisées, nous essaierons autant que possible de préciser les situations où la différence est notable. Lors de l'installation de Python, il est donc important de bien choisir.

1.2.2 La distribution Python

Il existe plusieurs distributions de Python qui permettent d'installer facilement l'interpréteur CPython déjà compilé ainsi que les librairies supplémentaires. Dans ce cours, nous utiliserons la distribution Anaconda <https://www.anaconda.com/distribution/> qui offre plusieurs avantages : elle est disponible sous Windows, Mac et Linux. Elle offre aussi la possibilité d'avoir une installation indépendante du système (ce qui est particulièrement utile sous Linux où Python est souvent installé par défaut). En installant Anaconda, la plupart des logiciels permettant de travailles avec Python seront installés (notons IPython, jupyter notebook, spyder..). De même les librairies scientifiques de bases seront aussi installées.

Nous laissons le lecteur suivre en ligne les instructions d'installation propre à son système.

1.3 Utilisation de Python

Il existe plusieurs façons d'exécuter un code Python. On peut lancer un interpréteur dans une console, ce qui permet d'exécuter les instructions ligne par ligne. On peut aussi écrire le script dans un ficher que l'on exécute. Enfin, il existe des solutions intermédiaires avec lesquels on utilise à la foi un fichier et une console.

1.3.1 Console Python

La console Python se lance simplement à partir du programme `python`. Sous windows, on peut ouvrir un terminal depuis le menu `Anaconda`.

La console Python peut être utilisée comme une simple calculatrice :

```
>>> 1+13
14
```

On peut aussi y entrer des structures plus complexes directement :

```
>>> l = ['alpha', 'beta']
>>> for elm in l:
...     print(elm)
...
alpha
beta
```

1.3.2 Console IPython

La console IPython est beaucoup plus pratique à utiliser que la console Python. Elle remplace de fait la console standard. La page <https://ipython.org/ipython-doc/2/interactive/tutorial.html> indique les principales fonctions de IPython. Notons :

— Une console qui permet d'accéder au commandes précédentes (en utilisant la flèche du haut). Il est possible de limiter la recherche en commençant à taper les premiers caractères de la lignes que l'on veut retrouver

— La complétion automatique à l'aide de la touche de tabulation.

```
In [1]: une_variable_dont_le_nom_est_long = 3.2
```

On pourra ensuite simplement taper quelques lettres (par exemple `une_va`) puis la touche de tabulation. Si plusieurs choix sont possibles, ils s'afficheront. Ceci est valable pour les noms de variables (donc aussi pour les fonctions), mais aussi les attributs ou méthodes des objets.

— Les commandes magiques. Sera utilisé particulièrement la commande `%run` qui permet d'exécuter un fichier.

1.3.3 Utilisation d'un fichier

Dès que l'on veut exécuter plus que quelques instructions, il est préférable d'utiliser un fichier. Les fichiers contenant des instructions python ont comme extension `.py`. Écrivons à l'aide d'un éditeur de texte standard les quelques instructions suivantes :

```
nom = 'Pierre'
print('Bonjour', nom, '!')
```

Ce fichier peut être exécuté directement à partir d'un shell à l'aide de l'instruction `python nom_de_fichier.py`. Normalement, ce programme devrait afficher :

```
Bonjour Pierre !
```

1.3.4 Spyder

Spyder offre un environnement de travail pour Python qui se veut un clone de celui de Matlab. Il est donc adapté à l'utilisation pour les scientifiques. En ouvrant ce programme, on a d'un côté un éditeur de texte et de l'autre un fenêtre d'aide et une console IPython. Nous recommandons l'utilisation de cet environnement.

Exemple : lancer spyder et ouvrir le fichier créé précédemment. Exécuter le contenu de ce fichier (à partir des menus ou directement avec la touche F5).

1.3.5 Jupyter Notebook

Le jupyter notebook offre un environnement dans lequel on peut afficher à la foi les instructions (regroupés dans dans des cellules) et les résultats de ces instructions au fur et a mesure de leur exécutions. Il est par exemple possible d'afficher des graphiques. Cet environnement est particulièrement intéressant lorsque l'on veut pouvoir présenter directement des résultats (par exemple pour un TP, ou une simulation).

Jupyter se présente sous la forme d'un serveur http auquel on accède avec un navigateur web. Sous linux, lancer simplement le notebook à partir d'un terminal à l'aide de la commande

```
jupyter notebook
```

Normalement, le navigateur par défaut devrait s'ouvrir.

Sous windows, il existe un lien depuis le menu `anaconda`.

1.3.6 Exemple

Voici un premier exemple de code Python. L'objectif de ce code est de calculer la valeur de e^x. Nous allons utiliser le développement limité suivant :

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Pour cela, nous allons calculer la somme jusqu'à une valeur $n_{\max}$ de n telle que $x^n/n! < \epsilon$. La variable ϵ déterminera la précision du calcul (plus ϵ est petit, meilleure est la précision).

Le code Python permettant de faire ce calcul est le suivant :

```
x = 3.14
epsilon = 1E-6
resultat = 0
n = 1
terme = 1 # Valeur initiale de terme de la boucle
while terme>epsilon :
    resultat = resultat + terme
    terme  = terme * x/n
    n = n + 1
print(resultat)
```

Ce code ne devrait pas poser de problème au lecteur ayant déjà eu des cours de programmation. Une particularité du langage Python est présente : dans la boucle `while`, le bloc d'instruction qui est répété est déterminé par l'indentation (espaces au début de chaque ligne). Python est donc différent de la pluspart des langages qui utilisent une structure du type `begin-end` ou bien des accolades pour déterminer le bloc d'instruction qui est répété. En python un bloc d'instruction est repéré par les `:` et un ensemble de lignes indenté identiquement.

Pour effectuer l'indentation d'une ligne ou d'un bloc de ligne, le plus simple est d'utiliser la touche de tabulation (et shift + tabulation) pour supprimer l'indentation.

Exercice

Essayez d'exécuter ce code en utilisant les différentes méthodes proposées ci dessus :

- En recopiant le script dans un fichier `exponentielle.py` et en l'exécutant avec la commande `python exponentielle.py`
- En utilisant ipython
- En utilisant spyder
- En utilisant un notebook jupyter.

Pour créer une fonction, on utilise l'instruction `def`, ce qui donne :

```
def exp(x, epsilon=1e-6): # epsilon vaut par défaut 1e-6
    """ Renvoie e a la puissance x """
    resultat = 0
    n = 1
    terme = 1 # Valeur initiale du terme de la boucle
    while terme>epsilon :
        resultat = resultat + terme
        terme = terme * x/n
        n = n + 1
    return resultat
```

Exercice

Depuis un éditeur (par exemple spyder), modifiez le fichier pour en faire une fonction. Exécutez le fichier et la fonction.

CHAPITRE 2

Les types de données en Python

Il s'agit d'une introduction basique sur les types de données offerts par python. Cette introduction peut servir d'aide mémoire pour les utilisations les plus courantes, sachant que l'aide officiel de python est beaucoup plus complète.

2.1 Les nombres

C'est le type le plus simple. Il existe trois types de nombre :

- Les entier (type `int`). Par exemple `a = 5`, en binaire `a = 0b1001`, en hexadécimal `a = 0x23`.
- Les nombres à virgules flottante (type `float`). Par exemple `a = 1234.234` ou `a = 6.2E-34`. Les nombres sont enregistré en double précision (64 bits). La précision relative est de 52 bits, soit environ 10^{-16}. :

```
>>> a = 3.14
>>> a == a + 1E-15
False
>>> a == a + 1E-16
True
```

- Les nombres complexes (type `complex`). Il sont enregistrés sous la forme de deux nombres à virgule flottante. Par exemple `a = 1 + 3J`.

Les opérations sur les nombres sont les suivantes :

- somme : `+`
- produit : `*`
- différence ou négation : `-`
- division : `/`
- division entière : `//`
- modulo : `%` (par exemple `7%2`)
- puissance : `**` (par exemple `2**10`)

Avertissement : En Python 2, la division simple `/` entre deux entiers est la division euclidienne. Ainsi, `print 1/2` donne `0`. Dans la version 3, la division simple renvoie toujours un flottant. Comme souvent, il est possible d'anticiper ce comportement dans les dernières versions de Python 2 :

```
>>> from __future__ import division
>>> 1/2
0.5
```

Pour les nombres complexes, on peut facilement accéder à la partie réelle et imaginaire de la façon suivante : :

```
>>> a = 1 + 3J
>>> a.imag
3.0
>>> a.real
1.0
>>> print("Norme ", sqrt(a.real**2 + a.imag**2))
Norme  3.1622776601683795
```

Nous avons introduit ici des attributs d'objet. Il s'agit de la notation `objet.attribut` (`imag` est un attribut de l'objet `1 + 3J`). Nous verrons ceci plus en détail par la suite.

2.2 Les booléens et comparaison

Il existe deux valeurs : `True` et `False` (attention à la casse). Les opérations sont par ordre de priorité : `not`, `and` et `or`. Les comparaisons se font à l'aide des symboles <, <=, ==, > et >=. Pour savoir si deux valeurs sont différentes, on utilise `!=`.

Les symboles d'opérations que l'on a vu sont des opérateurs binaires, c'est a dire qu'il appellent implicitement une fonction prenant deux arguments. Ce n'est pas le cas de `and` et `or` qui effectuent en fait un test conditionnel. L'instruction `A and B` est interprété comme `B if not A else A`, de même `A or B` équivaut à `A if A else B`. Voir le chapitre *Expression en Python* pour cette syntaxe particulière.

Avertissement : Les symboles `&` et `|` sont des opérateurs binaires. Ils réalisent les opérations and et or sur les entiers bit par bit en binaire (par exemple `6 & 5` donne `4`). Il ne faut pas les utiliser pour les opérations sur des booléens.

2.3 Les chaînes de caractères (string)

2.3.1 Création d'une chaîne de caractères

Il existe trois façons de créer une chaîne de caractère : avec des `'`, des `"` et des `"""`. Ces caractères servent à délimiter les début et la fin du texte de la chaîne de caractère. Les guillemets simples `'` et doubles `"` sont équivalents. On pourra choisir l'un ou l'autre. Il sera cependant judicieux, si une chaîne de caractère doit contenir un de ces guillemets, d'utiliser l'autre pour le début et la fin de la chaîne. Les trois guillemets sont eux utilisés lorsque l'on veut qu'une chaîne de caractère soit sur plusieurs lignes.

Voici quelques exemples :

```
>>> s = 'Pierre'
>>> s = "Aujourd'hui" #Rq : s = 'Aujourd'hui' ne va pas fonctionner
>>> s = """Aujourd'hui, le petit enfant a dit:
...  "Faisons le clown!" """
```

2.3.2 Caractères spéciaux

Les **caractères spéciaux** sont les caractères qui ne sont pas affichables et en tant que tel. Par exemple, il existe un caractère pour le retour à la ligne. Il est possible d'utiliser ce caractère dans une chaîne en utilisant `\n`. L'antislash sert ici de caractère d'échappement pour indiquer que l'on va entrer un caractère spécial. La lettre `n` indique ici qu'il s'agit d'un retour à la ligne.

Voici un exemple :

```
>>> s = "Un\nDeux"
>>> print(s)
Un
Deux
>>> len(s) # \n compte pour un caractère
7
```

L'antislash sert aussi à insérer un guillemet dans une chaîne : `'Aujourd\'hui'`. Si on veut mettre un antislash dans un chaîne, il faut le précéder d'un autre antislash « \def ». Si on ne souhaite pas que Python interprète ces caractères spéciaux, il est possible de précéder la déclaration de la chaîne d'un `r` :

```
>>> a = r"\theta"
>>> print(a)
\theta
```

2.3.3 Manipulation des chaînes de caractères

Tout comme une liste, il est possible d'accéder à chaque caractère d'une chaîne ou à une partie d'une chaîne.

```
>>> s = "Pierre"
>>> s[0]
'P'
>>> s[1:3]
'ie'
```

La longueur de la chaîne s'obtient avec la fonction `len`. On peut aussi faire une boucle `for` sur chacun des éléments de la chaîne.

Cependant, il n'est pas possible de modifier une chaîne de caractères (l'opération `s[0]='p'` échoue).

La **concaténation** est l'opération qui consiste à créer une nouvelle chaîne en mettant à la suite deux chaînes de caractères. Elle se fait à l'aide du signe +. Par exemple :

```
>>> s1 = 'Un'
>>> s2 = 'Deux'
>>> s1+s2
'UnDeux'
```

Une autre opération est importante, il s'agit du **formatage** d'une chaîne de caractère. Cette opération consiste à insérer un élément variable dans une chaîne. Elle est souvent utilisée lorsque l'on veut afficher proprement un résultat. Voici une exemple :

```
>>> heure = 15
>>> minute = 30
>>> "Il est {0}h{1}".format(heure, minute)
'Il est 15h30'
```

Pour insérer un élément ou plusieurs éléments variables dans une chaîne de caractère, on crée d'abord cette chaîne en mettant à la place des ces éléments une accolade avec un numéro d'ordre `{i}`. En appliquant la méthode `format` sur cette chaîne, les accolades seront remplacées par le ième argument.

Il est possible de passer l'argument par nom dans ce cas la clé est le nom de l'argument.

```
>>> "Il est {heure}h{minute}".format(heure=heure, minute=minute)
'Il est 15h30'
```

Depuis la version 3.6 de Python, il est possible de demander à Python d'utiliser automatiquement les variables locales à l'aide du préfix `f`.

```
>>> f"Il est {heure}h{minute}"
'Il est 15h30'
```

Il est aussi possible de demander d'utiliser un attribut d'un objet :

```
>>> z = 1 + 2J
>>> print(f'Re(z) is {z.real}')
Re(z) is 1.0
```

2.3.4 Formatage de nombre

Nous avons déjà vu qu'il était possible de transformer un nombre en une chaîne de caractère à l'aide de la fonction `str`. En utilisant le formatage de chaîne de caractère, il est possible de spécifier en détail comment ce nombre doit s'afficher. Par exemple, si il s'agit d'un nombre à virgule flottante, combien de décimales faut-il afficher, faut il utiliser la notation scientifique, etc. Pour cela, on rajoute à l'intérieur des accolades un code particulier. Ce code est précédé du signe " :".

Voici quelques exemples :

```
>>> from math import pi
>>> '{0:.5f}'.format(pi)
'3.14159'
>>> c = 299792458. # Vitesse de la lumière en m/s
>>> 'c = {0:.3e} m/s'.format(c)
'c = 2.998e+08 m/s'
```

Le "f" indique que l'on veut une notation a virgule fixe, le "e" une notation scientifique. Le chiffre que l'on indique après le "." donne le nombre de chiffre après la virgule que l'on souhaite.

Note : L'aide en ligne de Python fournit d'autres exemples et des détails.

Il existe aussi une façon plus élémentaire de formater des chaîne de caractères avec Python et qui est obsolète (mais que l'on peut rencontrer). Pour formater le nombre `pi`, cette méthode écrira dans ce cas

`'%.6f'%pi`.

2.3.5 Les accents

Historiquement, l'informatique étant née dans des pays anglo-saxons, les caractères utilisés se limitent aux caractères latin simple (sans accent). En utilisant les lettres minuscules, majuscules, les chiffres et autres signes, une norme est apparu concernant 128 caractères. Chaque caractère se voyant attribué un nombre entre 0 et 127 (2^7). Dans un fichier informatique, chaque caractère correspond alors à un octet. Les 128 octets restant ont fait l'objet de normes locales (par exemple la norme Windows-1252 pour les ordinateurs vendu dans le pays latins occidentaux, ISO 8859-1 pour les unix/linux).

Un nouveau standard, l'unicode a été développé à partir des années 90. Ce standard associe un nombre unique à chaque caractère (dans plus de 90 langues), allant des hiéroglyphes égyptiens aux milliers de sinogrammes asiatiques. Reste alors le problème d'enregistrer dans un fichier un texte écrit à l'aide de cette norme. Le plus simple serait, au lieu d'utiliser un octet par caractère, d'utiliser plusieurs octets. Cependant, les fichiers contenant du texte simple seraient alors inutilement long. Et d'autre part, il n'y aurait pas de compatibilité vers le standard ASCII (un fichier ASCII serait mal interprété). Il existe une norme permettant de ne pas avoir ces inconvénients : il s'agit de l'UTF-8. Les caractères ASCII standard dont toujours enregistré sur un octet, et on utilise plusieurs octets pour les autres caractères.

Dans Python 3, que nous utilisons, les chaînes de caractères sont des chaînes de caractères unicode. Si on souhaite des chaînes d'octet ASCII, il faut alors préfixer la chaîne par une `b`, par exemple `b'abcde'`. Dans Python 2, le comportement est opposé, par défaut les chaînes sont des chaînes d'octet et pour créer une chaîne unicode, il faut préfixer par un `u`.

Il est possible de rentrer un caractère directement à partir de sa valeur unicode. Il y a deux façons : pour un simple caractère, on peut utiliser la fonction `chr` (unichr en Python 2), pour un caractères dans une chaîne on utilise la séquence suivante `'\uxxxx'`, où `xxxx` est la valeur en hexadécimal du caractère unicode.

Par exemple, la lettre grec α (alpha) a pour valeur 945. En hexadécimal, elle s'écrit 03B1. On pourra l'utiliser de la façon suivante :

```
>>> chr(945)
'α'
>>> print("La lettre \u03B1 est la première lettre de l'alphabet grec")
La lettre α est la première lettre de l'alphabet grec
```

Il y a enfin une notion supplémentaire qu'il faut introduire, il s'agit de la possibilité d'afficher ou non un caractère. La console que nous utilisons (sous linux ou spyder) permet d'afficher un très grand nombre de caractères (essayer par exemple `print u'\u4e2d\u570b'` ou `print u'\u0646\u0627\u0646\u0628\u0644'`). Elle ne permet cependant pas d'afficher les hiéroglyphes... Beaucoup de logiciels utilisent des polices de caractères qui n'affichent qu'une partie très restreinte de la table unicode.

2.4 Les listes (list)

— On peut créer et remplir une liste de plusieurs manières :

```
>>> l = [1, 2, 3, 4]
>>> l = [] # liste vide
>>> l.append(3)
>>> l
[3]
>>> l.append(4)
>>> l
[3, 4]
>>> l.insert(0,3.24+1j)
>>> l
[(3.24+1j), 3, 4]
>>> len(l) # Longueur de la liste
3
```

— On peut changer un élément de la liste :

```
>>> l[2] = 23
```

— On peut créer une nouvelle liste à partir d'une liste à l'aide de la commande `list` :

```
>>> l = [2, 3]
>>> m = list(l)
>>> l.append(45)
>>> l==m
False
```

— La commande `range(n)` peut être utilisée pour créer la liste `[0,1,2, .., n-2, n-1]`. Cette liste commence à `0` et contient `n` nombres (elle s'arrête donc à `n-1`).

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Techniquement, depuis la version 3 de Python, range ne renvoie plus une liste, mais un *générateur*. Ce générateur produit la suite de chiffre qui est ensuite utilisée par le constructeur de la liste.

— Une liste peut contenir des éléments de types différents.

— Souvent on a besoin de créer une liste dans une boucle. Par exemple la liste des `n` premiers nombres au carré peut se calculer de la façon suivante

```
l = []
for i in range(n):
    l.append(i**2)
```

Il existe cependant une façon plus directe de faire, en utilisant les `list comprehension`

```
l = [i**2 for i in range(n)]
```

Cette syntaxe se lit directement en français : calcule la liste des i au carré pour i allant de 0 à n. Il est aussi possible de rajouter une condition (pour i allant de 0 à n si i est pair)

l = [i**2 for i in range(n) if i%2==0]

— Si on souhaite appliquer une fonction à tous les éléments d'une liste pour recréer une liste, il est possible d'utiliser la fonction `map`

```
def mafonction(i):
    if i%2==0:
        return i**2
    else:
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return i**2-1

map(mafonction, range(5)) # [0,0,4,8,16]
```

— On peut parcourir une liste de plusieurs façons

```
l = [1, 3, 5, "Pierre"]
for elm in l:
    print(elm)

for i,elm in enumerate(l):
    print(elm, " est l'element numero ",i," de la liste)
```

Avec les listes, nous avons donc introduit la structure de contrôle `for`. Celle ci fonctionne sur tout type de séquence (par exemple les chaînes de caractères). Par cette syntaxe, elle est assez différente de ce qui existe dans d'autres langage informatique.

Note : Nous avons vu qu'il était possible de rajouter des éléments dans une liste. Pour cela, nous avons utilisé la syntaxe `l.append(elm)`. Il s'agit ici de programmation orientée objet. Il est utile de savoir reconnaître la syntaxe : la liste est suivi du nom de la fonction que l'on applique (on appelle cette fonction une *méthode*) avec ses arguments. Ici, donc, `append` rajoute `elm` à la liste `l`.

Notons de plus, que l'objet `l` est modifié. La variable `l` désigne toujours la même liste, mais cette liste est modifiée (si on ajoute une page à un classeur, il s'agit toujours du même classeur). Ce comportement est différent de celui que nous avons vu dans la paragraphe précédent avec la méthode `format` sur les chaînes de caractères. En effet dans ce cas, on crée un nouvelle chaîne de caractère qui est renvoyé par la méthode - la chaîne initiale n'étant pas modifiée. La méthode `append` appliquée à une liste, modifie la liste, mais ne renvoie rien.

— Pour trier une liste, on peut soit utiliser la méthode `.sort` qui *modifie* la liste soit la fonction `sorted` qui renvoie une nouvelle liste triée. Il est possible d'ajouter comme argument optionnel une fonction qui donne l'ordre. Par défaut, python utilise la fonction cmp (ordre croissant pour les nombre et alphabétique pour le chaîne)

```
l = ['Pour', 'trier', 'une', 'liste', 'on', 'peut']
print(sorted(l))

def compare(a,b):
    u""" Ordre alphabétique inversé sans tenir compte
    de la casse"""
    return cmp(b.lower(), a.lower())

sorted(l, compare)
```

— Fonction `zip` : lorsque l'on veut parcourir deux listes en même temps, il est possible d'utiliser la fonction zip qui crée alors une liste de n-uplets à partir de `n` listes. :

```
>>> liste_nom = ['Martin', 'Lefevre', 'Dubois', 'Durand']
>>> liste_prenom = ['Emma', 'Nathan', 'Lola', 'Lucas']
```

```
>>> for nom, prenom in zip(liste_nom, liste_prenom):
...     print(prenom, nom)
Emma Martin
```

(suite sur la page suivante)

(suite de la page précédente)

```
Nathan Lefevre
Lola Dubois
Lucas Durand
```

2.5 Les n-uplets (tuple)

Les n-uplets sont utilisés lorsque l'on veut regrouper des valeurs ensembles. On utilise une liste lorsque l'on a une longue séquence (dont la longueur est souvent variable) et un n-uplet pour regrouper quelques éléments. Par exemple :

```
>>> personne = ('Jean', 'Dupont', '13 juillet 1973', 38)
>>> print('Nom :',personne[1])
Nom : Dupont
```

Les n-uplets sont utilisés lorsqu'une fonction renvoie plusieurs éléments :

```
>>> def fonction(x):
...     return x**2,x**3
>>> a = fonction(4)
>>> a
(16, 64)
>>> a,b = fonction(4)
```

2.6 Les dictionnaires (dictionnary)

Contrairement aux listes ou aux n-uplets qui sont indexés par des nombres (`l[2]`), un dictionnaire est indexé par une clé. En général, la clé est une chaîne de caractère ou un nombre. On peut reprendre l'exemple précédent :

```
>>> personne = {"Prenom":"Jean", "Nom":"Dupont",
...    "date_naissance":"13 juillet 1973", "Age":38}
>>> print("Nom :",personne['Nom'])
Nom : Dupont
```

Tout comme pour une liste, on peut ajouter ou enlever des éléments à un dictionnaire :

```
>>> personne['telephone'] = "02 99 79 24 58"
>>> del personne['Age']
```

Il est possible de parcourir et accéder à un dictionnaire de plusieurs façons :

```
>>> releve_note = {'Jacques':15, 'Jean':16, 'Pierre':14}

# Par clef
>>> for nom in sorted(releve_note.keys()):
...     print('La note de {0} est {1}'.format(nom, releve_note[nom]))
La note de Jacques est 15
La note de Jean est 16
La note de Pierre est 14
```

(suite sur la page suivante)

(suite de la page précédente)

```
# Par valeurs
>>> notes = releve_note.values()
>>> print("Moyenne", sum(notes)/len(notes))
Moyenne 15.0
```

```
# Par clef et valeur
>>> for cle, val in releve_note.items():
...     print('La note de {0} est {1}'.format(cle, val))
La note de Pierre est 14
La note de Jean est 16
La note de Jacques est 15
```

La clé d'un dictionnaire doit être un objet non modifiable, c'est à dire un nombre, une chaîne de caractère ou un tuple contenant uniquement des objets non modifiables.

2.7 Les ensembles (set)

Utilisé pour représenter un ensemble non-ordonné d'éléments qui sont tous différents. Par exemple :

```
>>> a = set([1,2,3])
>>> b = set([3,5,6])
```

```
>>> c = a | b # union
>>> d = a & b # intersection
```

Un exemple d'application :

```
>>> mdp = raw_input('Entrez un mot de passe contenant \
...                          au moins un signe de ponctuation')
>>> ponctuation = set("?,.;:!")
>>> if (ponctuation & mdp == set()):
...     print("Le mot de passe ne contient pas de signe de ponctuation")
```

Les ensembles sont très pratiques lorsque l'on veut supprimer des doublons.

Exercice

Récupérez un texte quelconque en chinois (par exemple https://www.gutenberg.org/files/24051/24051-0.txt). Combien d'idéogrammes chinois différents sont utilisés dans ce texte ?

2.8 Tableaux numpy (numpy array)

Cette structure n'est pas native de Python, mais fait partie du package numpy. C'est la structure qu'il faut utiliser pour traiter des données ou faire des calculs numériques. A la différence d'une liste, la taille d'un tableau numpy n'est pas modifiable et l'ensemble des données doit être du même type. Ces conditions permettent un traitement beaucoup plus rapide des données. Cette structure sera vue en détail dans la *partie dédiée à numpy*.

Quelques exemples :

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

```
>>> a = np.array([[1.5,3.14,5.27],[2.17,0.69,1.414]])
>>> a.shape
(2, 3)
>>> print(a.dtype)
float64
```

```
>>> print(a[1,2]) # Attention, notation en partant de 0, comme en C
...               # mais pas comme en Fortran, scilab, matlab, ...
1.414
```

```
>>> a[1,2] = 8.56 # on peut changer un element
```

```
>>> a = np.arange(5) # comme range, sauf que l'on renvoie un tableau
```

```
a = np.loadtxt('data_file.txt')
            # data_file est une fichier texte contenant des
            # données numériques (type tableur)
```

Un grand nombre d'opération simple peut être réalisé sur des tableaux. Elle sont réalisées élément par élément :

```
>>> a = np.array([[1.5,3.14,5.27],[2.17,0.69,1.414]])
>>> b = 2*a + a**2 - a
```

Important : lors de l'utilisation de tableau contenant un grand nombre de données, cette méthode est **beaucoup** plus rapide (en plus d'être plus simple) que de faire l'opération élément par élément avec une liste. A titre d'exemple comparer la vitesse de

```
# N premiers carrés
a = [i**2 for i in range(1000000)]
b = np.arange(1000000)**2
```

Les tableaux numpy sont proche du C ou de Fortran. Ils offrent plus de type que Python. Par exemple les nombres à virgule flottante peuvent être enregistrés en 32, 64 ou 96 bits.

Avertissement : Attention, le type des éléments d'un tableau ne peut pas être modifié. Ainsi, si un tableau est défini comme un tableau d'entier, on ne peut pas mettre dedans un nombre flottant. Celui ci sera automatiquement converti en nombre entier. C'est un piège auquel il faut faire attention

```
>>> a = np.arange(5)
>>> a[1] = 3.141592
>>> a[1]
3
```

CHAPITRE 3

Eléments du langage Python

3.1 Expression en Python

Dans un langage de programmation, on distingue les expressions des commandes. Les expressions vont être évaluer par l'interpréteur pour renvoyer un objet.

Nous n'allons pas faire une description exhaustive de toutes les possibilités.

— Les parenthèses peuvent avoir plusieurs sens :

```
>>> sin(1 + 2) # appel d'une fonction
0.1411200080598672
>>> (1 + 2J)*3 # parenthèse logique
(3+6j)
>>> (1, 2J)*3 # n-uplet
(1, 2j, 1, 2j, 1, 2j)
```

— Nous vous rappelons la syntaxe dite de *list comprehension* :

```
>>> [i**2 for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [i//3 for i in range(10) if i%2==0]
[0, 0, 1, 2, 2]
```

De même il existe des *comprehension* pour les dictionnaires et les ensembles :

```
>>> s = {chr(97+i) for i in range(10) if i%2==0} # ensemble
>>> s == {'i', 'c', 'g', 'e', 'a'}
True
>>> d = {chr(97+i):i for i in range(10) if i%2==0} # dictionnaire
>>> d == {'e': 4, 'c': 2, 'g': 6, 'i': 8, 'a': 0}
True
```

— Il est possible d'utiliser une condition dans une expression sous la forme intuitive *renvoie A si B sinon C*. Ce type d'expression permet d'éviter de passer par des variables intermédiaires.

```
sqrt(x) if x>=0 else sqrt(-x)*1J
print("Bonjour" if lang=='fr' else 'Hello')
```

3.2 Créer des fonctions

3.2.1 Arguments d'une fonction

Voici un exemple général de la définition d'une fonction :

```
>>> def f(a, b, c=1, d=2, *args, **kwd):
...     print(a, b, c, d)
...     print(args)
...     print(kwd)
```

Cette fonction possède deux arguments obligatoires, deux arguments optionnels. Les variables args et kwd vont contenir les arguments supplémentaires (sans et avec mots-clé - keyword).

Lorsque l'on appelle une fonction, les arguments peuvent être passés anonymement (par ordre) ou avec un nom (*keyword argument*, `nom=valeur`). Il faut mettre d'abord les arguments anonymes puis les autres. Il n'y a pas de lien entre le fait qu'un argument ait une valeur par défaut et le fait qu'il soit utilisé avec son nom. Lorsque les arguments sont passés avec leur nom, l'ordre est indifférent :

```
>>> f(1, 2, 4)
1 2 4 2
()
{}
>>> f(b=2 , a=2)
2 2 1 2
()
{}
```

Les arguments en trop sont envoyés dans args ou kwd :

```
>>> f(1, 2, 3, 4, 5, 6)
1 2 3 4
(5, 6)
{}
```

```
>>> f(1, 2, 3, e=4)
1 2 3 2
()
{'e': 4}
```

```
>>> f(1, 2, 3, 4, 5, 6, e=7)
1 2 3 4
(5, 6)
{'e': 7}
```

Enfin, il est possible de séparer un groupe d'arguments à partir d'un itérable (list, tuple, ..) (séparation anonyme) ou à partir d'un dictionnaire (séparation avec mots-clés) :

```
>>> liste = list(range(1,3))
>>> dct = {'d':3, 'e':4}
>>> f(0, *liste, **dct)
... # equivaut à f(0, 1, 2, d=3, e=4)
0 1 2 3
()
{'e': 4}
```

On remarquera que les variables `args` et `kwd` à l'intérieur de la fonction `f` sont différentes de celles que l'on a séparées (liste et dct dans cet exemple).

Il est de toute façon possible de séparer plusieurs listes ou dictionnaires :

```
>>> print(*range(3), *range(3))
0 1 2 0 1 2
```

Quelques remarques :

— Il ne faut par hésiter à utiliser des arguments par défaut (et c'est mieux que des variables globales)

— Lorsque l'on appelle une fonction, il ne faut pas hésiter à nommer les arguments, même si c'est inutile et que c'est plus long à taper. Comparez

```
>>> scope.configure_channel(1, 0.01, 0.03, 50)
>>> scope.configure_channel(channel_name=1, scale=0.01, offset=0.03,␣
↪impedance=50)
```

3.2.2 Fonction anonyme

Lorsque l'on crée une fonction à l'aide de l'instruction `def ma_fonction()`, il y a en fait deux choses faites en parallèle : la création d'un objet `function` qui porte le nom `ma_fonction` ainsi que l'assignation de la variable `ma_fonction` à cette fonction.

Il est possible cependant de créer une fonction en python sans avoir à lui donner un nom, ni à l'assigner à une variable. C'est ce que appel une fonction anonyme. Elle est créer à l'aide de l'instruction lambda :

```
lambda args:resultat
```

où `args` représente tout ce que l'on peut mettre dans les arguments d'une fonction usuelle et `resultat` est une expression Python. Par exemple :

```
lambda x, a, b : a*x+b
```

Voici un exemple plus concret. La fonction `quad` du module `scipy.integrate` permet d'effectuer l'intégrale d'une fonction :

```
from scipy.integrate import quad
quad(lambda x:x**2, 0, 1, epsrel=1E-12)
```

3.2.3 Variable locale/globale

Les variables que l'on crée dans une fonction sont locale, c'est à dire indépendante d'une variable extérieure à la fonction et qui porte le même nom.

```
>>> def f():
...     x = 2
>>> x = 3
>>> f()
>>> x
3
```

A l'intérieur d'une fonction, une variable est soit locale soit globale

3.2.4 Décorateur

Il arrive fréquemment que l'on veuille modifier ou rajouter des fonctionnalités à une fonction. Pour cela, il est possible en Python de créer une fonction qui renvoie une nouvelle fonction. C'est ce que l'on appelle in *décorateur*. Prenons l'exemple de cette fonctions :

```
def trace_fonction(f):
    def wrapper(*args):
        name = f.__name__
        arg_str = ', '.join((str(elm) for elm in args))
        print(f'Appel de la fonction {name}({arg_str})')
        return f(*args)
    return wrapper

from math import sin
sin = trace_fonction(sin)
```

Ce qui donne :

```
>>> sin(4)
Appel de la fonction sin(4)
-0.7568024953079282
```

Notez que dans la fonction `wrapper`, le variable `f` est globale tout en étant locale dans `trace_fonction`.

Lorsque l'on définit la fonction soit même, on peut utiliser une syntaxe particulière : avant la ligne contenant mot clé `def`, on indique le nom du décorateur précédé de `@`. Par exemple :

```
@trace_fonction
def difference(x, y):
    return x - y
```

Ce qui permet de s'amuser, et savoir comment python trie une liste :

```
>>> from functools import cmp_to_key
>>> liste = [2, 4, 6, 3, 7, 3]
>>> liste.sort(key=cmp_to_key(difference))
Appel de la fonction difference(4, 2)
Appel de la fonction difference(6, 4)
Appel de la fonction difference(3, 6)
Appel de la fonction difference(3, 4)
Appel de la fonction difference(3, 2)
Appel de la fonction difference(7, 4)
Appel de la fonction difference(7, 6)
Appel de la fonction difference(3, 4)
Appel de la fonction difference(3, 3)
```

Un exemple de décorateur est la fonction `vectorize` du module `numpy`, qui permet d'adapter n'importe quelle fonction à un tableau :

```
import math
import numpy as np

@np.vectorize
def my_sqrt(x):
```

(suite sur la page suivante)

(suite de la page précédente)

```
    if x>=0:
        return math.sqrt(x) + 0J
    else:
        return math.sqrt(-x)*1J
```

Exercice

Créer une décorateur list_vectorize qui fait la même chose pour des listes

---

On peut aller un étape plus loin, ce que l'on met après l'arobase est n'importe quelle expression qui renvoie un décorateur. Par exemple, cela peut-être le résultat de l'appel d'une fonction. Par exemple :

```
def check_arg_type(arg_type):
    def decorateur(f):
        def wrapper(x):
            assert isinstance(x, arg_type), 'Argument should of type {}'.
↪format(str(arg_type))
            return f(x)
        return wrapper
    return decorateur

@check_arg_type(int)
def fibonacci(n):
    if n<=1: return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Ce qui donne :

```
>>> fibonacci(5)
5
>>> fibonacci(5.4)
AssertionError: Argument should of type <class 'int'>
```

On peut mettre plusieurs décorateurs :

```
@check_arg_type(int)
@trace_fonction
def fibonacci(n):
    if n<=1: return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Ce qui donne :

```
>>> fibonacci(5)
Appel de la fonction fibonacci(5)
Appel de la fonction fibonacci(4)
Appel de la fonction fibonacci(3)
Appel de la fonction fibonacci(2)
Appel de la fonction fibonacci(1)
Appel de la fonction fibonacci(0)
Appel de la fonction fibonacci(1)
Appel de la fonction fibonacci(2)
Appel de la fonction fibonacci(1)
Appel de la fonction fibonacci(0)
```

(suite sur la page suivante)

(suite de la page précédente)

```
Appel de la fonction fibonacci(3)
Appel de la fonction fibonacci(2)
Appel de la fonction fibonacci(1)
Appel de la fonction fibonacci(0)
Appel de la fonction fibonacci(1)
5
```

Évidement, on voit dans ce cas, que la façon de procéder n'est pas efficace, on peut alors décider de mettre en cache le résultat :

```
def cached(f):
    memory = {}
    def wrapper(x):
        return memory.get(x, None) or memory.setdefault(x, f(x))
    return wrapper

@cached
@check_arg_type(int)
@trace_fonction
def fibonacci(n):
    if n<=1: return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Ce qui donne :

```
>>> fibonacci(5)
Appel de la fonction fibonacci(5)
Appel de la fonction fibonacci(4)
Appel de la fonction fibonacci(3)
Appel de la fonction fibonacci(2)
Appel de la fonction fibonacci(1)
Appel de la fonction fibonacci(0)
5
```

Note : La fonction `functools.lru_cache` du module `functools` effectue cette tache.

Exercice

Modifiez le décorateur `check_arg_type` pour qu'il fonctionne avec un nombre arbitraire d'arguments.

3.2.5 Les annotations

Il s'agit d'une nouvelle fonctionnalité de Python 3. Chaque argument peut être précédé d'un annotation qui est un objet quelconque de Python. Cette modification de la syntaxe Python ne change rien à l'interprétation du code. Elle pourra servir par exemple à analyser automatiquement le code :

```
def ma_fonction(param1: "Une annotation", param2: 1 + 1, param3: int):
    print(param1, param2, param3)
```

(suite sur la page suivante)

(suite de la page précédente)

```
ma_fonction(1,2,3)
```

```
1 2 3
```

```
print(ma_fonction.__annotations__)
```

```
{'param1': 'Une annotation', 'param2': 2, 'param3': <class 'int'>}
```

Exercice

Ecrire un décorateur qui à partir d'une fonction annotée avec de types, renvoie une fonction qui vérifie le type de donnée automatiquement lors de son appel. On utilisera la fonction `isinstance(val, type)` pour tester si `val` est du type `type`.

3.3 Les boucles

Il existe des boucle for et des boucle while.

Pour sortir d'une boucle on peut utiliser l'instruction `break`, pour passer à l'itération suivante l'instruction `continue`. Si une boucle se finit normalement (sans break), il est alors possible d'exécuter un bloc d'instruction dans un `else`. Voici un examole :

```
def affiche_si_premier(n):
    i=2
    while i**2<=n:
        if n%i==0:
            print("{} n'est pas premier".format(n))
            break
        i = i+1
    else:
        print('{} est premier'.format(n))
```

Remarquons que en Python il est possible de quitter une fonction à n'importe quel moment à l'aide de l'instruction return. Lorsque dans une boucle on connaît le résultat de la fonction, il est alors préférable de quitter celle ci immédiatement :

```
def est_premier(n):
    i=2
    while i**2<=n:
        if n%i==0:
            return False
        i = i+1
    return True
```

3.3.1 Boucle for

Nous avons vu qu'il est possible de faire des boucles for sur des listes, des chaînes de caractères, des tuples. Il existe un concept général que l'on appelle itérateur. On va dire que les listes, chaînes de

caractères et les tuples sont des itérateurs. Un itérateur est un objet qui possède une méthode `__iter__` laquelle va renvoyer un objet (souvent le même) qui possède une méthode `__next__`. On le verra plus en détail dans la partie orientée objet. Par exemple

```
>>> l = [0,1,2]
>>> x = l.__iter__()
>>> x.__next__()
0
>>> x.__next__()
1
>>> x.__next__()
2
>>> x.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

3.3.2 Générateurs

Les générateurs sont utilisés pour faciliter la création d'itérateur. Il s'agit d'un fonction qui va renvoyer une série de résultat au lieu d'une simple valeur. On utilise la mot clé `yield`. Un générateur ne pourra utiliser de `return` :

```
>>> def my_range(n):
...     i = 0
...     while i<n:
...         yield i
...         i = i + 1
```

Une nouvelle valeur est renvoyée à chaque fois que yield est appelé :

```
>>> y = my_range(3)
>>> type(y)
<class 'generator'>
>>> y.__next__()
0
>>> y.__next__()
1
>>> y.__next__()
2
>>> y.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Dans le cas où l'on veut transformer une liste en une autre liste, cette méthode permet d'éviter de créer explicitement la liste. Par exemple

```
>>> def carre(l):
...     for elm in l:
...         yield elm**2
...
>>> for elm in carre(range(4)):
...     print(elm)
0
```

(suite sur la page suivante)

(suite de la page précédente)

```
1
4
9
```

Cette méthode permet aussi de parcourir les éléments d'une structure compliquée, comme un système 2D ou un arbre

```
def indices_tableau_2D(n, m):
    for i in range(n):
        for j in range(m):
            yield i,j
```

Et une structure en arbre. Affiche l'ensemble des fichiers d'un répertoire ayant une extension donnée :

```
import os

def parcourt_repertoire(repertoire, ext):
    for elm in os.listdir(repertoire):
        elm_abs_path = os.path.join(repertoire, elm)
        if os.path.isdir(elm_path):
            for filename in parcourt_repertoire(elm_abs_path, ext):
                yield filename
        elif elm_abs_path.endswith(ext):
            yield elm_abs_path

list(parcourt_repertoire('.', '.py'))
```

Exercice

Faire une fonction indices_tableau qui prend n argument et parcourt tous les indices de ce tableau nD. On utilisera des tuples pour les indices et on fera quelque chose de récursif (pour parcourir un tableau nD, on parcours un tableau (n-1)D le nombre de fois qu'il faut).

3.4 Les exceptions

Il arrive souvent lors de l'exécution d'un programme qu'une erreur apparaisse (par exemple prendre la racine carré d'un nombre négatif). Dans ce cas, Python arrête l'exécution du programme et affiche l'erreur. Ce comportement peut être modifié : il est en effet possible de stopper la propagation de l'erreur. Ceci se fait à l'aide la structure `try except`

```
from math import sqrt

a = -1
try:
    b = sqrt(a)
except ValueError:
    b = 0
```

Dans le cas où une fonction que l'on a créée ne peut pas être exécutée correctement, il est possible de créer et renvoyer une erreur.

```
def triangle_existe(a,b,c):
    u""" Teste si le triangle de côté a,b,c existe """
    pass

def aire_triangle(a,b,c):
    u""" Calcule l'aire d'un triangle de côté a,b,c """
    if not triangle_existe(a,b,c):
        raise ValueError(u"Le triangle de côté {0},{1},{2} n'existe pas".
↪format(a,b,c))
    pass
```

Il existe plusieurs type d'erreur générique (`ValueError`, `NameError`, `SyntaxError`). Dans la partie orientée objet, nous verrons comment créer son propre type d'erreur.

Un cas fréquent consiste à attraper une erreur et ensuite à renvoyer une nouvelle erreur. Par exemple, la formule de Héron permet de calculer l'aire d'un triangle en connaissant uniquement la longueur des trois côtés a,b et c.

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

où s est le demi-périmètre

$$s = \frac{1}{2}(a+b+c)$$

Dans le cas où le triangle n'existe par, l'argument de la racine carré est négatif. Le module `math` va renvoyer une `ValueError`. Il est possible de modifier cette erreur et de renvoyer un message plus explicite :

```
from math import sqrt

def aire_triangle(a, b, c):
    s = (a+b+c)/2
    try:
        return sqrt(s*(s-a)*(s-b)*(s-c))
    except ValueError:
        msg = "Le triangle de côtés {a}, {b} et {c} n'existe pas"
        raise ValueError(msg.format(a=a, b=b, c=c))
```

Ce qui donne :

```
>>> aire_triangle(3, 4, 5)
6.0
>>> aire_triangle(1, 1, 10)
ValueError: Le triangle de côtés 1, 1 et 10 n'existe pas
```

Note : Il est plus efficace d'utiliser un `try ... except` que de faire un test préalable (qui peut parfois être compliqué). C'est un comportement non-intuitif. En Python, il est plus efficace de traverser la route les yeux fermé et de voir ce qu'on fait si on a un accident que de regarder si il y a une voiture qui passe avant de traverser. Cette façon de procéder peut être indispensable dans certaine situation. Par exemple :

```
>>> try:
...     open('un_fichier_qui_n_existe_pas')
... except FileNotFoundError:
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     print('Erreur : entrez un nouveau fichier')
Erreur : entrez un nouveau fichier
```

En effet, si on teste l'existence du fichier, il est toujours possible (bien que peu probable) que le système d'exploitation supprime le fichier entre le test et l'ouverture du fichier.

3.5 Context Managers

Il s'agit de la syntaxe suivante

```
with open('fichier') as f:
    f.write('Bonjour')
```

Cette syntaxe est équivalente à la suivante :

```
f = open('fichier')
try:
    f.write('Bonjour')
finally:
    f.close()
```

Cette syntaxe permet donc de s'assurer que le fichier sera fermé même si il y a une erreur dans le bloc d'instruction. On retrouve principalement cette syntaxe dans le cas d'un fichier. On peut aussi l'utiliser pour une connexion à un serveur ou un instrument de mesure.

Le cas général suivant :

```
with something as t:
    BLOCK
```

est équivalent à :

```
t = something.__enter__()
try:
    BLOCK
finally:
    t.__exit__()
```

Les méthode __enter__ et __exit__ seront propre au type d'objet utilisé dans le context manager.

CHAPITRE 4

Programmation orientée objet

4.1 Les objets

Nous avons utilisé ce terme auparavant pour désigner de façon générique ce que contient une variable : un nombre, une liste, une chaîne de caractère, une fonction, etc. Il est possible en Python de créer des objets dont le type est personnalisé. Ces objets auront pour objectif de stocker des données et de pouvoir les traiter à l'aide de méthodes que l'on aura définies.

Nous allons illustrer dans ce chapitre l'utilisation d'objets pour réaliser un annuaire personnalisé.

4.2 Les attributs

Voici un exemple :

```
class Personne():
    nom = ""
```

Que l'on peut utiliser de la façon suivante :

```
>>> jean = Personne()
>>> jean.nom = "Dupont" # modification de l'attribut
>>> jean.prenom = "Jean" # On peut rajouter un attribut
>>> print(jean.nom)
Dupont
```

Nous avons tout d'abord créé un nouveau type d'objet. Il s'agit du type `Personne`. On appelle cela une *class*. Cette class contient pour l'instant un *attribut* : le nom. Pour créer un objet, on appelle le nom de la class. On peut lire ou écrire un attribut simplement. Cet objet est donc modifiable (*mutable*). On peut aussi rajouter un attribut (ici, le prénom) à l'objet.

Pour cet exemple, nous aurions pu utiliser une liste (ou un tuple) pour stocker le nom et le prénom dans une seule variable, mais l'objet rend le code beaucoup plus lisible (`jean.nom` est plus explicite que jean[0]). C'est un premier avantage.

4.3 Les méthodes

Nous avons vu que les attributs permettent de stocker les informations.

On peut alors imaginer la création de fonctions utilisant ces objets :

```
def bonjour(personne):
    print("Bonjour", personne.prenom, personne.nom, '!')
```

Plutôt que de définir une fonction indépendamment de l'objet, il est possible d'attacher cette fonction à l'objet. C'est ce qu'on appelle une méthode

```
class Personne():
    def bonjour(self):
        print("Bonjour", self.prenom, self.nom, '!')

jean = Personne()
jean.nom = "Dupont"
jean.prenom = "Jean"
jean.bonjour()
```

Toute fonction définie dans le corps de la class est une méthode. Le premier argument de la méthode est l'objet lui même que l'on nomme conventionnellement `self`. L'appel d'une méthode se fait de la façon suivante : `objet.nom_methode(tous_les_arguments_sauf_self)`.

On voit ici un deuxième avantage de l'utilisation d'objet : la variable contient des fonctions (les méthodes). L'utilisateur de la variable ne doit pas ce soucier de la façon de réaliser une fonction ni de l'importer. Il fait confiance à la variable qui sait comment faire. Ceci permet en particulier d'avoir pour deux variables différentes, deux méthodes différentes ayant le même nom.

4.4 Méthodes spéciales

Il existe plusieurs méthodes qui ont des rôles bien précis

4.4.1 Méthode __init__

Cette méthode sera appelée automatiquement lors de la création de l'objet. On peut l'utiliser pour initialiser des attributs :

```
class Personne():
    def __init__(self, nom="", prenom=""):
        self.nom = nom
        self.prenom = prenom
    def bonjour(self):
        print("Bonjour", self.prenom, self.nom, '!')
```

Ce qui donne :

```
>>> jean = Personne("Dupont", "Jean")
>>> jean.bonjour()
Bonjour Jean Dupont !
```

4.4.2 Méthode `__repr__`

Cette méthode sert a représenter l'objet sous forme d'une chaîne de caractère. Elle est utilisé par la fonction `print` :

```
class Personne():
    def __init__(self, nom="", prenom=""):
        self.nom = nom
        self.prenom = prenom
    def bonjour(self):
        print("Bonjour", self.prenom, self.nom, '!')
    def __repr__(self):
        return self.prenom + " " + self.nom
```

Ce qui donne :

```
>>> jean = Personne(prenom="Jean", nom="Dupont")
>>> print(jean)
Jean Dupont
```

Note : Il existe aussi la méthode `__str__` qui par défaut renverra la même chose que `__repr__`. Cette méthode peut être utlisée pour renvoyé une chaîne plus simple est lisible que `__repr__`. Par exemple, pour une chaîne de caractère, le `__repr__` rajoute des guillemets pour signifier qu'il n's'agit d'une chaîne de caractère, mais pas le `__str__` :

```
>>> s = 'Pierre'
>>> s.__repr__()=="'Pierre'"
True
>>> s.__str__()=='Pierre'
True
```

4.5 Exemple

On veut pouvoir créer des liens d'amitié (à la facebook) entre les personnes. Ainsi chaque personnes a un ensemble d'amis. Cet ensemble, initialement vide, est créé au moment de l'initialisation de l'objet. Il faut alors pouvoir rajouter un ami ou afficher la liste des amis :

```
class Personne():
    def __init__(self, nom="", prenom=""):
        self.nom = nom
        self.prenom = prenom
        self.liste_amis = set()
    def bonjour(self):
        print("Bonjour", self.prenom, self.nom, '!')
    def ajoute_ami(self, ami):
        self.liste_amis.add(ami)
    def affiche_amis(self):
        print("Les amis de", self.__repr__(), "sont :")
        for ami in self.liste_amis:
            print(ami)
    def __repr__(self):
        return self.prenom + " " + self.nom
```

Que l'on utilise de la façon suivante :

```
>>> jean = Personne(prenom="Jean", nom="Dupont")
>>> jacques = Personne(prenom="Jacques", nom="Dupond")
>>> pierre = Personne(prenom="Pierre", nom="Martin")
>>> jean.ajoute_ami(jacques)
>>> jean.ajoute_ami(pierre)
```

```
>>> jean.affiche_amis()
Les amis de Jean Dupont sont :
Jacques Dupond
Pierre Martin
```

Exercice

Les amis de mes amis sont mes amis ! Écrire (en trois lignes) une méthode qui ajoute automatiquement les amis de mes amis à mes amis.

4.6 Héritage

C'est le dernier point important concernant les objets en Python. Nous avons vu comment à l'aide du mot clé `class` définir une classe d'objet. Dans cette classe, nous avons défini l'ensemble des attributs et des méthodes. Il est cependant possible d'hériter des méthodes d'une autre classe - ce qui permet d'avoir plusieurs classes possédant des méthodes identiques.

Continuons notre exemple : suivant la nationalité, je souhaite pouvoir écrire un message de bonjour différent. Pour cela, je vais créer une classe PersonneFrancaise et PersonneAnglaise :

```
class Personne():
    def __init__(self, nom="", prenom=""):
        self.nom = nom
        self.prenom = prenom
        self.liste_ami = set()
    def ajoute_ami(self, ami):
        self.liste_ami.add(ami)
    def affiche_amis(self):
        print("Les amis de", self.__repr__(), "sont :")
        for ami in self.liste_amis:
            print(ami)
    def __repr__(self):
        return self.prenom + " " + self.nom

class PersonneAnglaise(Personne):
    def bonjour(self):
        print("Hello", self.prenom, self.nom, '!')

class PersonneFrancaise(Personne):
    def bonjour(self):
        print("Bonjour", self.prenom, self.nom, '!')
```

Ce qui donne :

```
>>> john = PersonneAnglaise(prenom="John", nom="Dupont")
>>> john.bonjour()
Hello John Dupont !
```

Du côté de l'utilisateur, celui qui reçoit la variable `john` ne veut pas savoir si `john` est français ou anglais, il ne veut même pas savoir comment dire bonjour à un anglais. Par contre, il sait que pour dire bonjour à john, il suffit de faire `john.bonjour()`.

Du côté du programmeur, on applique ici le principe DRY (Don't Repeat Yourself), ou le principe de ne pas faire du copier-coller dans son code. Les méthodes qui sont communes ne sont pas dupliquées.

4.7 Autres méthodes spéciales

4.7.1 Opérateurs binaires

Les méthodes __add__, __mul__, __sub__, __truediv__ sont utilisées pour implémenter le `+`, `*`, `-` et `/` entre deux objets. Lorsque l'on fait par exemple `a+b`, alors l'interpréteur va appeler `a.__add__(b)`. Ceci nous permet donc de donner le sens que l'on souhaite à n'importe quel opérateur binaire.

D'autres méthodes spéciales existe pour tous les opérateurs binaires (c.f. https://docs.python.org/3/reference/datamodel.html).

Ceci permet, par exemple d'imiter un type numérique - mais peut être utilisé pour n'importe quel autre objet pour simplifier la création d'un objet. C'est par exemple ce que l'on voit implicitement lorsque l'on fait un + entre deux listes ou deux chaînes de caractères.

Notons enfin que pour toutes méthodes, il existe une version réciproque (__radd__, __rmul__, etc.). Cette méthode est appelée lorsque la première méthode échoue (concrètement en revoyant NotImplemented). Par exemple `a+b` appelle d'abord `a.__add__(b)`. Si cette méthode renvoie NotImplemented alors `b.__radd__(a)` est appelé. Ceci permet donc à un nouveau type de donnée de fonctionner avec un autre type, sans que celui-ci soit modifié. Par exemple la syntaxe suivante fonctionne `2*'pa'`, on se doute bien que ce n'est pas la type int qui implémente sa multiplication par une chaîne de caractère, mais bien la chaîne de caractère qui sait quoi faire lorsqu'elle est multipliée par un entier.

4.7.2 Émulation des conteneur

Il existe plusieurs type de conteneur en Python, par exemple les listes, les tuples, les dictionnaires, etc. Ils on commun que l'on peut extraire un élément à l'aide de la syntaxe `a[key]`. On peut aussi vouloir changer un élément `a[key] = val` ou l'effacer `del a[key]`. Toutes ces syntaxes correspondent à des méthode de l'objet `a`. Ainsi :

- `a[key]` correspond à a.__getitem__(self, key)
- `a[key] = val` correspond à a.__setitem__(self, key, val)
- `del a[key]` correspond à a.__delitem__(self, key)
- `len(a)` correspond à a.__len__(self)
- `for elm in a` correspond à `for elm in a.__iter__()`

4.8 Attribut et property

Il est important de distinguer les attributs de classe et le attribut d'objet. Lorsque l'on définit une classe, tous les attributs sont des attributs de la classe. Lorsque l'on assigne un attribut à un objet, on crée

alors un attribut pour l'objet. Concrètement, cet attribut est stocké dans un dictionnaire créé avec l'objet. Lorsque l'on veut obtenir un attribut, l'interpréteur renvoie l'attribut de l'objet si celui ci existe et l'attribut de la classe de l'objet sinon.

```
class Test():
    a = 1
    b = []
    c = []
    def __init__(self, val):
        self.b = [val]
        self.c.append(val)

test1 = Test(10)
test1.c.append(3)
test1.b.append(3)
test2 = Test(20)
```

Ce qui donne :

```
>>> test2.b
[20]
>>> test2.c
[10, 3, 20]
```

Dans ce cas, `test1.a` : attribut de la classe ; `test1.b` attribut de l'objet après le `__init__` ; `test1.c` attribut de la classe.

Nous avons vu ici deux mécanismes différents lorsque l'on fait `obj.attr` : si `obj.__dict__` possède la clé `"attr"` alors on renvoie `obj.__dict__["attr"]`, sinon on appelle `type(obj).attr`. Il existe un troisième mécanisme : si aucune des deux méthodes en fonctionne, et si l'objet possède une méthode __getattr__ alors cette méthode est appelée :

```
class Test():
    a = 1
    def __init__(self):
        self.b = 2

    def __getattr__(self, key):
        if key=='c':
            return 3
        raise AttributeError

t = Test()
```

Ce qui donne :

```
>>> t.a
1
>>> t.b
2
>>> t.c
3
```

Un quatrième mécanisme existe, il s'agit des property. Une poperty est une fonction qui est appelé lorsque l'on accède à un attribut. Lorsqu'une classe possède un attribut qui est une property, alors c'est ce résultat qui est renvoyé à la place de l'attribut. Par exemple :

```
class Test():
    def __init__(self, nom, prenom=""):
        self.prenom = prenom
        self.nom = nom

    @property
    def nom_complet(self):
        if self.prenom:
            return self.prenom + ' '+ self.nom
        return self.nom

t = Test('Dupond', 'Jean')
```

Ce qui donne :

```
>>> t.nom_complet
'Jean Dupond'
```

De façon similaire, l'affectation d'un objet à un attribut peut être spécifié à l'aide de la méthode `__setattr__`, ou d'un `setter` dans le cas d'une property. On peut par exemple utiliser ce mécanisme pour protéger un attribut :

```
class Test():
    def __init__(self, percent):
        self.percent = percent

    @property
    def percent(self):
        return self._percent

    @percent.setter
    def percent(self, val):
        if val<0 or val>100:
            raise Exception('Percent should be between 0 and 100')
        self._percent = val

t = Test(10)
```

Note : Il existe un convention en Python qui est que les attributs commençant par une underscore sont privés, c'est à dire qu'ils ne doivent pas être utilisés en dehors de la définition de la classe. A la différence d'autre langage où des attributs privés sont contraint par le compilateur, il s'agit ici uniquement d'un convention. Dans l'exemple suivant, on peut toujours appeler `t._percent` et même faire librement `t._percent = 200`.

4.9 Exemple

Voici un exemple de simulation d'un nombre complexe

```
from math import atan2

class Complex(object):
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def __init__(self, partie_reelle, partie_imaginaire):
        self.real = partie_reelle
        self.ima = partie_imaginaire

    def display(self):
        print('{}+{}i'.format(self.real, self.ima))

    def __str__(self):
        if self.ima>0:
            return '{}+{}i'.format(self.real, self.ima)
        else:
            return '{}-{}i'.format(self.real, -self.ima)

    def __repr__(self):
        return "Complexe({}, {})".format(self.real, self.ima)

    def __add__(self, other):
        if isinstance(other, Complex):
            return Complex(self.real+other.real,
                           self.ima+other.ima)
        else:
            return Complex(self.real+other,
                           self.ima)

    def __radd__(self, other):
        return self + other

    # idem pour __mul__, __truediv__, __sub__, __pow__, __neg__
    # et leur reciproque

    def conj(self):
        return Complex(self.real, -self.ima)

    @property
    def theta(self):
        return atan2(self.ima, self.real)

class ImaginairePur(Complex):
    def __init__(self, val):
        self.real = 0
        self.ima = val

    def __str__(self):
        return '{}i'.format(self.ima)
```

4.10 Quand faut-il utiliser des objets ?

L'idée générale de l'utilisation d'objet est de supprimer la difficulté pour l'utilisateur et de la déplacer au niveau de la définition des méthodes de la classe. Dans cette méthode, on aura accès à la plupart des paramètres nécessaires pour exécuter une action.

La difficulté n'est généralement pas de savoir quand il faut utiliser une objet (le réponse est tout le temps !), mais de réussir à identifier dans un problème quels sont les objets à définir.

Les objets permettent souvent de décrire des objets réel dont la classe est le concept. Voici quelques exemples :

— Un livre : C'est un objet qui possède comme attribut : un titre, auteur, ... Mais aussi une liste de chapitres, lesquels ont un titre et des parties, ... On peut alors imaginer une méthode qui renvoie un sommaire du livre (méthode avec comme argument la profondeur que l'on souhaite du sommaire). Une méthode qui renvoie le nombre de chapitre, ...

— Un circuit électronique : il y a des composants électronique et des connexions entre les composants. Pour les composants, on aura plusieurs classes qui définiront chacune un type de composant (dipôle linéaire passif, transistor, amplificateur opérationnel, ...). Chaque composant aura des pattes (un autre type d'objet). Une connexion rassemblera les pattes des composants. L'objectif de ces objets est de décrire complètement le circuit électronique. On pourra alors imaginer des méthodes plus ou moins complexes : faire une liste triée des composants que l'on devra acheter ou calculer la réponse impulsionnel du circuit...

— Un instrument de mesure : par exemple un oscilloscope, il aura des méthodes pour changer l'échelle verticale ou horizontale, récupérer la courbe, etc.

— On souhaite modéliser le système solaire : on pourra créer une classe Planete et une classe Systeme. Une planète contient un nom et une masse. Le système enregistrera pour chaque planète qu'on lui ajoute sa position et sa vitesse. Le système sera capable d'être représenté graphiquement. Il pourra créer l'ensemble d'équation différentiel qui sera ensuite utilisé par un solveur afin de connaître la position des planète à un autre instant. Dans cet exemple, l'utilisation d'objet est d'aucune utilité pour la résolution en elle même du problème (qui est un exercice classique). Par contre, elle permet de fournir une interface avec l'utilisateur qui va facilité l'entrée des paramètres ou leur visualisation.

— Les objets sont utilisés pour décrire des structures complexes dans un ordinateur : par exemple la fenêtre de l'interface graphique d'une application est un objet (qui contient d'autres objets comme les boutons, les menus, ...). En python, Matplotlib représente une figure par un objet.

— En physique, on pourra utiliser un objet pour représenter l'ensemble des paramètres d'un problème ou les résultats d'une mesure. Par exemple, l'image prise par un télescope sera un objet qui contiendra l'image, mais aussi le moment de la prise de vue, sa durée, l'orientation du télescope. Cet objet devra pouvoir être enregistré sur un fichier et créé à partir de ce fichier. Une autre classe permettra de faire une analyse de cette image. Elle s'initialisera avec une image et aura des méthodes pour faire l'analyse (appliquer des filtres, mesurer des distances entre étoiles, ...).

CHAPITRE 5

Gérer un projet en python

Ce chapitre à pour but d'expliquer quelques concepts et outils utils pour créer un projet en Python. La première partie explique la notion de modules et de package en Python. La seconde comment faire une documentation pour un projet et la dernière partie parle des tests unittaires.

5.1 Modules and Package

5.1.1 Modules

Creating a module

Let'us look to the following example. In the file exponential.py we have written the following function :

```
def exp(x, epsilon=1E-6):
    """ calculate e to the power x """
    result = 0
    n = 1
    term = 1 # Initial value
    while abs(term)>epsilon :
        result = result + term
        term  = term * x/n
        n = n+1
```

How to use this `exp` from another file ? A solution could be the execute the file from the script with the `exec(open('exponentielle.py').read())` command. However, there is a much more convenient technique in python. The function be can imported using the command

```
from exponential import exp
```

We do not put the `.py` file extension. Using modules has many advantages :

- If the module was already imported, python do not execute the file a second time.
- One can choose what we want to import in the module.
- Python can automatically looks for different pathes to find the file.

Importing from a module

Python is not a general purpose language. All the function that are specific are defined in a module. Mathematical function (and constants) are in the math module.

They can be imported in different ways :

— Import of the module

```
import math
print math.sin(math.pi/3)
```

— Import of specific objects in the module

```
from math import sin, pi
print sin(pi/3)
```

— Import of all the objects defined in the module

```
from math import *
print sin(pi/3)
```

The first method is more verbose because we have to specify the module name every time. However, we have to use this method in our example if we want to compare the exp function from two different modules.

In general, the second method should be preferred over the third one. Indeed, when all the functions are imported it is not easy then to know from which module the function was imported. Furthermore, this import can override existing function (for example, there is an `open` function is the module `os`. Therefore the command `from os import *` will override the standard `open` function).

The `import *` can be used for module with **well known functions** that are used **many times** in a program. This is the case for the `math` module.

5.1.2 Package

A package is a group of module (in other language it is called a library).

Installation of a package

Many packages are available on the Pypy web site. The `pip` shell command can be used to download and install the package on your computer. For example if you want to install the PyDAQmx package, simply run the command :

```
pip install PyDAQmx
```

If you don't have root permission, then use the `--user` option.

If the package is not available on Pypy, then the usual way consist in downloading the package and then install it by running the setup script :

```
wget https://pypi.python.org/packages/source/P/PyDAQmx/PyDAQmx-1.3.2.tar.gz
tar -xvzf PyDAQmx-1.3.2.tar.gz
cd PyDAQmx-1.3.2
python setup.py install --user
```

Create your package

A package is a collection of modules and sub-packages. In order to create a package, you simply have to put the files into the same directory (called for example `my_package`). Python will know that this directory is a package if the there is a file called `__init__.py` in the directory. This file can be empty.

In order to import the modules, you can then do `import my_package.mod1` or `from my_package import mod1,...`

The `__init__.py` is a python module whose content is imported by `from my_package import ...`.

Python will be able to import « my_package » :

— If your current working directory is the directory containing the « my_package » directory
— If this directory is in your search path. You can dynamically a directory into the search path using the following :

```
import sys
sys.path.insert(1, 'directory/containing/my_package')
```

— If your directory is in the the « PYTHONPATH » environment variable.
— If you have installed your package (see instruction bellow).

Local import

Inside a package, it is common to import modules or sub-packages, for example in the `__init__.py` file. This is a local import. You should indicate to python that an import is local by prefixing the module name with a dot. For example, in the `__init__.py` of my_package

```
from .mod1 import my_function

from . import mod1
mod1.my_function()
```

If you are in a sub-package, then you can import from the parent package using two (or more) dots.

Distribute your package

It is easy to create you package and distribute it. You have to write a setup.py file containing some informations about your package. Then Python will take care of everything. The setup.py file should be in the same directory as the « my_package » directory. A minimal example of a setup.py file is the following :

```
#!/usr/bin/env python
# -*- coding: utf_8 -*-

from distutils.core import setup
__version__ = "alpha"

long_description="""This is a very nice package

"""

setup(name='my_package',
```

(suite sur la page suivante)

(suite de la page précédente)

```
      version=__version__,
      description='A very nice package',
      author=u'François Pignon',
      author_email='francois.pignon@trucmuch.fr',
      url='',
      packages=['my_package'],
     )
```

Then you can execute the following commands :

```
python setup.py install # Install the package
python setup.py sdist # create a tar.gz or zip of your package

python setup.py register # register on Pypy
python seupt.py sdist upload # Upload your package on Pypy

pip install -e . --user
```

The last command is very useful for developers. It installs the project in editable mode, which means that modification of the project files will be taken into account (a simple install will copy the files to another place).

5.2 Créer une documentation

La librairie `Sphinx` est devenue le logiciel standard permettant d'écrire une documentation python. Le fait qu'elle soit écrite en python lui permet d'effectuer automatiquement un certain nombre de taches (comme récupérer des chaînes de documentation, effectuer des test, ...).

Sphinx permet d'exporter la documentation en format html mais aussi en Latex (et donc en PDF) ou en ODF (Libre office). Ce cours est écrite grâce à Sphinx.

5.2.1 Utilisation

Pour l'utiliser on lance le script `sphinx-quickstart`. On répond ensuite aux questions !

Il y alors deux fichiers importants : le fichier de configuration (`conf.py`). Ce fichier python peut être modifié pour adapter le projet (et changer ce qui a été créé suite aux questions de `sphinx-quickstart`).

Le deuxième fichier est `index.rst`. Il s'agit du fichier racine de la documentation. Sphinx utilise le reStructuredText comme langage de balisage.

5.2.2 Exemple

Voici un exemple de fichier `index.rst`

```
Voici un exemple de documentation
=================================

Voici un super projet.
```

(suite sur la page suivante)

(suite de la page précédente)

```
Installation
------------

Il suffit de lancer la commande ::

    python setup.py

Utilisation
-----------

Voici un exemple d'utilisation ::

    from truc.fichier import UneClasse

    ma_classe = UneClasse()
    ma_classe.une_methode()

Documentation
-------------

On peut facilement extraire la documentation d'un module grâce à la␣
↪directive ``automodule``.

Module fichier
++++++++++++++

.. automodule:: truc.fichier
     :members:


.. toctree::
   :maxdepth: 2
```

Dans cet exemple, nous avons utilisé la directive automodule qui importe un module (il existe aussi autoclass et autofunction) et utilise sa chaîne de documentation. Le paramètre `:members:` permet d'afficher automatiquement les membres du modules récursivement (classe, méthodes, fonctions).

La directive toctree permet de créer une arborescence à partir de fichiers `.rst`. Par exemple

```
.. toctree::
   :maxdepth: 1

   introduction
   part_1
   part_2
```

5.2.3 Pour aller plus loin

Regarder la documentation officielle et en particulier la page www.sphinx-doc.org/en/stable/rest.html

5.3 Test unitaire

5.3.1 Exemple

En informatique, il est important de toujours tester son code. Il existe plusieurs librairie en Python permettant d'automatiser l'exécution des tests. Nous avons choisi de présenter `unittest`.

Avec cette librairie, chaque test est représenté par une classe. On peut ensuite choisir les tests que l'on veut exécuter.

Voici un exemple (que l'on mettra dans le fichier test.py)

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

Pour exécuter le test, on peut simplement exécuter le fichier. On peut aussi utiliser une commande

```
python -m unittest test # test est le nom du module (fichier test.py)
python -m unittest discover
```

L'option discover est utilisée lorsqu'il y a plusieurs fichiers de test. Dans ce cas, tous les fichiers commençant par test seront considérés comme des fichiers de test.

On pourra regarder la documentation pour plus de détail. Il existe un grand nombre de méthode assertXXX (par exemple assertIsInstance, assertAlmostEqual, assertLess).

5.3.2 Utilisation dans une librairie

Lorsque l'on utilise unittest dans une librairie, il faut importer le module que l'on teste. Un choix consiste à écrire le test comme un sous-module de la librairie et à importer le module que l'on teste comme un module local. Il faudra alors exécuter le test depuis la racine du package.

Par exemple

```
my_package/
    __init__.py
    pkg1/
```

(suite sur la page suivante)

(suite de la page précédente)

```
        __init__.py
        test/
            __init__.py
            test_pkg1.py
    mod1.py
    test/
        __init__.py
        test_mod1.py
```

Dans le fichier `test_mod1.py`, on écrira :

```
from .. import mod1
# ou bien
from ..mod1 import xxx
```

Pour effectuer le test, on pourra alors utiliser l'une des commandes :

```
python -m unittest discover
python -m unittest my_package.test.test_mod1
python -m unittest my_package/test/test_mod1.py
```

5.4 Conclusion

Voici donc typiquement l'arborescence des fichiers d'un projet :

```
mon_projet/
    setup.py
    README.rst
    my_package/
        __init__.py
        pkg1/
            __init__.py
            test/
                __init__.py
                test_pkg1.py
        mod1.py
        test/
            __init__.py
            test_mod1.py
    doc/
        index.rst
        conf.py
        .....
    exemple/
        exemple1.py
    autre_chose/
```

Il est important de garder les répertoire my_package et doc aussi propre que possible. On mettre les exemples en dehors de my_package. Les autres choses (notes personnelles, tests temporaires, ...) seront mis dans d'autres répertoires afin de ne laisser que le setup.py et le README dans le répertoire principal.

CHAPITRE 6

Outils pour la science

6.1 Numpy Array

Numpy array is not a native Python datatype. It is part of the `numpy` package. This is the datatype that should be used for large numeric data analysis or numerical calculation. They differ from list : the size of a numpy array cannot be modified and all the data inside an array are of the same type. Those constraints allow much faster computation. We will mainly use 1D and 2D array. They can however be of any dimension. Numpy arrays allow users to perform calculation in a way similar to Matlab or Scilab.

Some examples :

```
from numpy import *

a = array([4,23,3]) # 1D array
print a[2]
a[2] = 13

b = array([[1,2,3],[2,3,4]])
print b
```

In order to use array, we should first import the numpy package. The array command can be used to create array from any iterable (for example a `list`). A list of number will create a 1D array. A list of lists having the same size will create a 2D array.

One can create an array containing the 0, 1, 2, 3, ... sequence using the `arange` command. This command has the same arguments than the `range` command (`arange(end)`, `arange(start, end)`, `arange(start, end, step)`). Items of an array can be changed using the same syntax as `list`. The size of an array can be obtained using the `shape` property. This property returns a tuple.

```
a = arange(10)
print a[2]
a[3] = 2
print a.shape
```

6.1.1 Mathematical operation on array

One of the main advantage of using numpy array is that it is possible to perform usual operation, item by item, without using a loop. This holds for binary operation (sum, product, difference, ...) and also for usual mathematical operation (sin, cos, tan, exp, log, ...). In order to work with array, those function should be imported from the `numpy` module (and not the `math` module). Global functions on array, such as `sum`, `prod`, `max`, `min`, `mean`, `std` can be used either as method of the array or as function of the numpy module.

For example :

```
from numpy import *
a = arange(10)
print a**2 - sin(a)

even_numbers = arange(2,10000,2)
print 2*prod(even_numbers**2)/prod(even_numbers**2-1)

x = linspace(0,5, 1000)
# using function
print min(x+2/x)
#using methods
a = x + 2/x
a.min()
```

In this example, we have used the function `linspace(start, end, nb_steps)`. This function creates an array with `nb_steps` values between `start` and `end`.

Note also that the `sin` function is imported from the numpy library. This function takes an array and returns an array of the same size.

Note that the builtins functions `sum`, `max` and `min` are different than the one imported from `numpy`. If you use the bultin ones, you loose the advantage of using numpy array.

It is possible to create a function that works with an array from any function using the `vectorize` function of the `numpy` package :

```
def myfunction(x):
    ....
    return ....

myfunction = vectorize(myfunction)
print myfunction(linspace(0,1))
```

6.1.2 Data type

In order to get the data type of the array, one can use the dtype attribute. It is possible to set the data type of an array when it is created using the optional `dtype` parameter.

For example :

```
a = arange(10)
print a.dtype
a = arange(10, dtype='float64')
print a.dtype
```

Numpy array have more data type than Python. For example, floating point number can be saved as 32, 64 or 128 bits (`float32`, `float64`, `float128`), integer are usually limited to 64 bits (this depends on you system).

The data type of an array cannot be modified. If an array is defined as an integer array one cannot put inside a float number. This number will be converted to an integer (without warning).

```
a = arange(5)
a[1] = 3.141592
print a[1] # a[1]==3
```

Of course, you can create a new array from an existing one with the data type you want.

6.1.3 Array indexing

One can access to array items the same way than for list (item number starts at 0, one can extract an array using slices : `a[start:end:step]`, negative numbers start from the end). Numpy array can be modified. This can be done element by element or on a group of elements.

For example :

```
a = arange(10)
a[3] = 34
a[::2] = 0 #This is not possible for a list
```

Be aware that when an array is extracted using slices, it share the same memory space than the initial array. The following code will modify the array `a`.

```
a = arange(10)
b = a[::2]
b[:] = 0
print a
```

It is possible to extract an array from an array using a boolean array. For example :

```
a = array([23,25,10])
condition = array([False, True, False])
print a[condition]
```

This is very useful because it is possible to create a boolean array using comparison and logical operators.

For example :

```
a = arange(10)
cond = a>=5
print cond

a[cond] = 5

b = arange(100)
cond = (b%2==0) & ~(b%5==2)
print b[cond]
```

Logical operators on array are `'&'` for and, `'|'` for or and `'~'` for not. The operators `and`, `or` and `not` do not work with array (indeed they are not operator !).

6.1.4 2D array and linear algebra

Here are some examples of 2D array

```
from numpy import *

a = array([[1,2],[1,9]])

print a[:,0] # Column number 0
print a[1,:] # Line number 1
```

2D numpy array can be used as matrices. The matrix product **is not** the `*` operator (which makes the product element by element). In order to perform true product one can use the `dot` function of the numpy package. The linalg package contains also many function of linear algebra (see the documentation).

```
import numpy as np
from numpy import linalg

a = np.array([[1,2],[1,9]])
b = np.array([[3,1.],[3.14,6.02]])
c = np.dot(a,b)

print linalg.eig(c) # eigen vectors and eigen values
```

We briefly mention here the `matrix` package of `numpy`. Using the `matrix` type, one can create matrices or vectors. The product is done conveniently using the `*` operator.

6.1.5 Useful functions and methods

Array methods :

- `.sum()`, `.prod()`, `.mean()`, `.std()`. Those method have an `axis` option that perform the operation only on one `axis`.
- `.reshape(shape)` : change the shape. The shape argument is a tuple. The total number of items should be unchanged
- `.flatten()` : makes a 1D array
- `.transpose()`

Array creation :

- `arange` : similar to the range command.
- `linspace` : the `linspace(start, stop, num=50, endpoint=True)`. The `endpoint` option set weather or not `stop` is the last sample (or the next one).
- `zeros(tpl)`, where `tpl=(n1,n2,...)` is a tuple containing the shape of the array.
- `ones(tpl)` : similar to zeros.
- `rand(nx, ny, ...)` : random number between 0 and 1. The shape will be (nx, ny, ...).
- `X,Y = meshgrid(x,y)` : created 2D array X and Y such that `X[i,j] = x[j]` and `Y[i,j] = y[j]`

6.1.6 Saving numpy array

Ascii format

One can save and read 2D numpy array in a text file using the `savetxt(fname, tableau)` and `loadtxt` commands. This is a universal method (ascii file can be read by many applications). However

it is slow and this method can lead to rounding error when numbers are converted to text.

The `loadtxt` can be used to read almost any file produced by another application. Look to the function documentation for the different options. For example one can set the separator in order to read a CSV (comma separated values) file.

```
loadtxt(filename, delimiter=',')
```

Binary format

The `load` and `save` function of the `numpy` module can be used to store directly numpy array in a file. This method is a direct copy of the memory. It is fast and uses less memory space. However if does not provide direct compatibility with other application.

It is possible to convert binary data of other application to array. If the data is written into a file, one can use the `fromfile` function. If the data comes directly from a function (for example, if it comes from a CCD camera or a digital scope), one can use the `frombuffer` function. They are universal functions that can be parametrized to read any format (signed/unsigned integer, big or little endian, 16 or 32 bits, ...).

6.1.7 Understanding numpy array

It's useful to understand how numpy arrays are stored in the memory. Because the size and the data type of the array are not mutable, the total amount of memory is fixed. The place where the data are stored is a contiguous block of memory, in a way similar to array in C or Fortran. One can access to this memory block using the `data` attribute of an array : this will return a buffer (basically a string of bytes). For example

```
from numpy import *
a = arange(5, dtype=int16)

print len(a.data) # 5*2 = 10 bytes
assert a.data[0]==chr(0) # The first byte is 0
```

The numpy array does not only contain a pointer to the starting address in the block of memory, but also informations on how to retrieve elements of the array. Each element of the array starts at a specific position in the block of memory. This position is a linear combination of the indexes in the array. For example, in a 10x20 2D array of int32, the position of element (i,j) will be $20 \times 4 \times i + 4 \times j$. The two coefficients $(80, 4)$ are the only parameters you need in order to read the array. The are obtained from the `strides` property of the array.

It is important to know that you can create two arrays that share the same memory block, but with a different starting address and a different `strides`. This is the mechanism used when slicing an array or reshaping the array.

```
from numpy import *
a = arange(10, dtype=int16)
print a.strides # (2,)
b = a[::2]
print b.strides # (4,)

a = array([1,2],[3,4])
```

(suite sur la page suivante)

(suite de la page précédente)

```
I,J = a.strides
a.strides = (J,I) # transposition
```

This mechanism is very fast and efficient because no copy of the data is performed. However, we should remember that modifying the content of one array will modify the content of all arrays that share the same memory block.

We have seen in the previous section that we can use operation on array (like +, ⋆, /, ...). They will perform the operatio element by element and return a new array. Those operation can also be used between an array and a scale (`2*a`). Numpy has a mechanism called broadcasting that makes possible, under certain circumstance (see docs.scipy.org/doc/numpy/user/basics.broadcasting.html), to work with two arrays of different sizes. For example an array of size $m \times 1$ can operate with an array of size $1 \times n$ to form an array of size $m \times n$. The arrays are broadcast to have the correct dimension by duplicating the column or row. Indeed, there is no real dupliction, simply numpy set the strides of the broadcast dimension to 0.

6.1.8 Beyond numpy

As we have seen, numpy array should be used when dealing with large data set. **You should not use** loops with numpy array. When there is a way to not use loop, execution can then be almost as fast as with a compiled language.

If you really have to use a loop and if this loop is the bottle neck of you program, then you can consider using alternative solution based on compilaton. There are two ways : either by building your library directely in the langage you choose, and then link it to python. In C, you can use the ctypes module and in Fortran, the F2PY module. The other, and more convenient, alternative is to use a tool to convert your function to a compiled langage. This a how the Cython module work.

Cython

The Cython package is used to convert a Python function to a C function which is compiled.

Let us see a simple example. We want to calculate the value of π using the following formula :

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

This formula is chosen because it is probably the slowest formula that can be used to calculate π. We will calculate the sum up to $k = 10^6$. The `%timeit` magic function of IPython was used.

```
# Using python
%timeit sum([(-1)**k/float(2*k+1) for k in range(10**6)])
# result : 493ms

%timeit sum([(1-2*(k%2))/float(2*k+1) for k in range(10**6)])
# result : 253 ms

def calc_pi(N):
    out = 0
    sgn = 1.
    for k in range(N):
        out += sgn/(2*k+1)
```

(suite sur la page suivante)

(suite de la page précédente)

```
        sgn = -sgn
    return out

%timeit calc_pi(10**6)
# result 112ms

import numpy as np
k = np.arange(10**6, dtype=np.float64)
%timeit np.sum((1-(k%2))/(2*k+1))
# result : 17ms
```

Below is a simple example using cython. The online documentation (http://docs.cython.org/) will give you more informations on how to use cython and to compile the code below. We have created a calc_pi.pyx file from the pure python function. The pyx file contains regular Python code with declaration of the type of variable.

```
def calc_pi(int N):
    cdef double out = 0
    cdef int i
    cdef double sgn = 1
    for i in xrange(N):
        out += sgn/(2*i+1)
        sgn = -sgn
    return out
```

To use the code below, run the following script

```
import pyximport; pyximport.install()
import calc_pi

print calc_pi(1E6)
```

Once compiled, the calc_pi function can be used as any python function. The duration is 4.9ms, four times faster than numpy. Even in the situation where numpy can be used, cython can be faster, the main reason is that in this example we don't need to store an array in memory.

ctypes

Ctypes is a library that allow to use C functions in Python. This library allow you to link any functions. You don't need the source code of the function but only a compiled shared library (.so for linux and .dll for windows). For this example, I will show you how to compile your shared library from a C file, but ctypes can be used even if you get a library from a third party (for example the driver of an instrument).

The C file to calculate pi is the following (named pi.c) :

```
#include <stdio.h>
#include <stdlib.h>

int calc_pi(int N, double * out){
    int i;
    double sgn = 1;
    *out = 0;
    for(i=0; i<N; i++){
```

(suite sur la page suivante)

(suite de la page précédente)

```
        *out += sgn/(2*i+1);
        sgn = -sgn;
        }
    }
```

Which is compiled (under linux) using :

```
gcc -shared -o libpi.so -fPIC pi.c  -Wno-pointer-to-int-cast
```

The `libpi.so` is now linked using ctypes :

```
import ctypes
from ctypes import byref
lib = ctypes.cdll.LoadLibrary('./libpi.so')

calc_pi = lib.calc_pi
out = ctypes.c_double(0) # creation of the output variable
calc_pi(10**6, byref(out))
print out.value
```

In this example, the memory for the output variable is created in the python script and then passed by reference to the C function. The execution time is similar to the Cython example.

Numba

Numba is a compiler for python. It is even simpler to use than cython, because no modification of the code is required. A simple decorator is used to accelerate a Python function.

```
import numba

@numba.jit(numba.float64(numba.int32), nogil=True)
def calc_pi_numba(N):
    out = 0
    sgn = 1.
    for k in range(N):
        out += sgn/(2*k+1)
        sgn = -sgn
    return out
```

Here the signature of the function is specified explicitly. You can also omit it, and numba will infer it for you the first time you use the function (and therefore compile it then). To specify the signature, use the following syntax : `out_type(arg1_type, arg2_type, ...)`. Usual types are `int64` (or `uint64` : unsigned int), `float64` or `complex128`. If your input or output is an array, then use `type[:]`. For example

```
from numba import jit, float64
@jit(float64[:](float64[:], float64[:])
def sum(x, y):
    return x + y
```

Another very useful function of numba is `vectorize`. It allows to create a function compatible with numpy using a function written for scalar

```
from numba import vectorize, float64

@vectorize([float64(float64)])
def my_abs(x):
    if x>0:
        return x
    else:
        return -x
```

Using numba.vectorize is a convenient and fast way to make function compatible with numpy. It is much faster than the numpy.vectorize and much more readable than full numpy implementation of the function (using numpy.where or array mask).

6.2 Graphics in Python

There is a very complete library in Python that can be used to plot curves. This the `matplotlib` library. It relies on the `numpy` library because curves are always plotted using arrays.

6.2.1 The plot command

Let us start with an example :

```
from pylab import *
ion()

x = linspace(0,2*pi)
y = sin(x) + cos(y)**2
plot(x, y)
```

The `pylab` modules will import all the function that will make Python suitable for numerical and graphical computation. It will import the matplotlib and numpy modules. To use pylab, one can either import it using the `from pylab import *` command or using an interactive python shell with the command `ipython -pylab`.

The `plot` command takes two parameters : the X and Y axis coordinates (if only one parameter is given, the x axis will be equal to arange(len(Y)))

Optional parameters are used to modify the way the graph is plotted. The two main parameters are the color and the shape. For example, the command `plot(x,y, 'k^')`, will plot triangle (signe `'^'`) in black (`'k'` is for black - the `'b'` is for blue). Look to the documentation for more details.

6.2.2 Other commands

To clear a graph, we use the function `clf()` (clear figure)

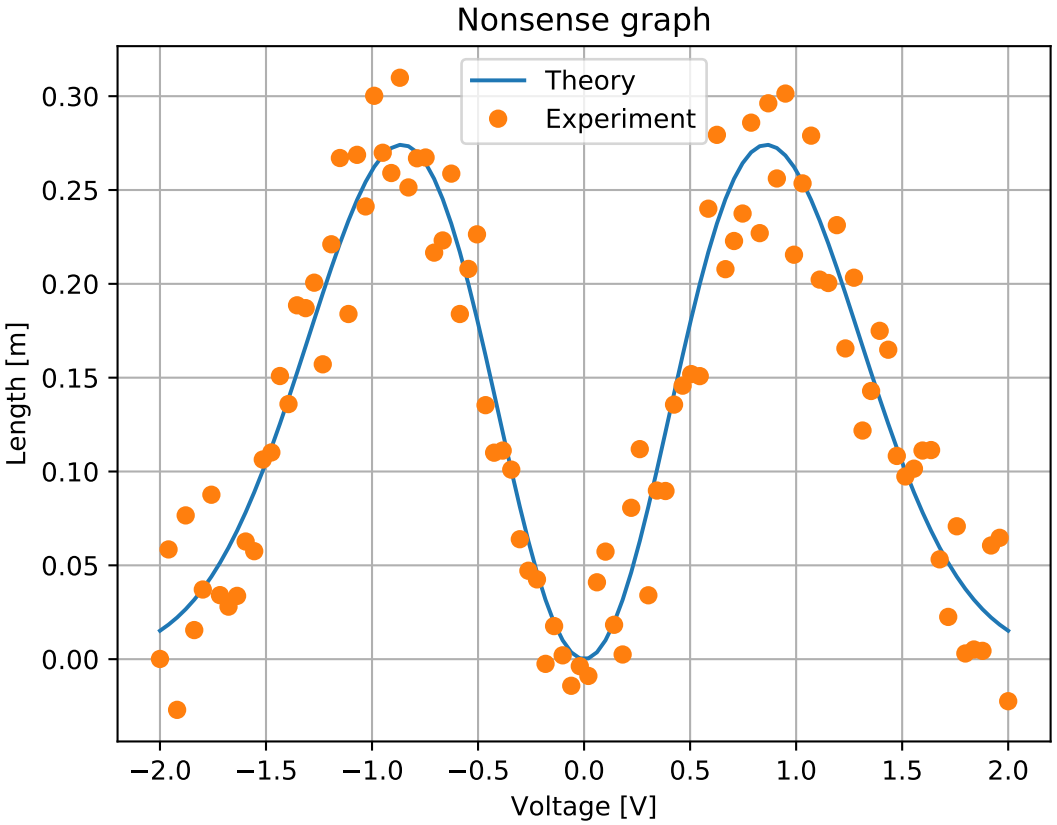
On can add a title (`title("Le titre")`), labels for axis (`xlabel('x-axis')` et `ylabel`). The optional `label` parameter is used to make a legend on a graph with many plots :

```
from pylab import *
X = linspace(-2,2, 100)
Y = sin(X)**2*exp(-X**2)
```

(suite sur la page suivante)

(suite de la page précédente)

```
Y_noise = Y + .1*(rand(len(X))-0.5)
plot(X,Y, label=u"Theory")
plot(X,Y_noise,'o', label=u"Experiment")
xlabel(u'Voltage [V]')
ylabel(u'Length [m]')
title("Nonsense graph")
legend()
grid(True)
```



6.2.3 Latex formula

In recent version of Matplotlib, it is possible to automatically insert Latex formula in graphs. They will be nicely converted

```
ylabel(u'Noise [$V/\sqrt{\mathrm{Hz}}$]')
```

6.2.4 Main graphical commands

- `plot(X,Y)`
- `loglog(X,Y)`, `semilogx(X,Y)`, `semilogy(X,Y)`
- `clf()` to clear the graph
- `xlabel('blabla')`, `ylabel('blabla')`, `title('blabla')`
- `xlim((x_inf, x_sup))`, `ylim((y_inf, y_sup))` to zoom onto a part of the graph

— `grid(True)` to plot the grid. The command `grid(True, which='both')` will plot a thin grid.
— `figure` to open a new figure (a new window). Figures are automatically numbered. You can swith to an existing figure using the `figure(n)` command.
— `subplot(nx,ny,m)` to make many plots on one figure.
— `imshow(image)` to plot a matrix using false color and `colorbar()` pour plot color scale.
— `text(x,y,s)` plot the text `s` at the `x, y` position. Optional parameters can be used to center correctly the text.
— `savefig(nom_fichier)`. Save the figure. The file format is determined by the file extension. We advise to use `pdf` for output that cannot be modified and svg if one want to modify it (using Inkscape- for example).

Screenshots are available on the `matplotlib` web site and can be used to quickly find the best way to plot a specific graph.

6.2.5 Using OO programmation

In the example above, we have used function to create a graph. However, a graph is a Python object and those function are simply shortcut the current graph.

More precisely, to create a graph, you need first to create a `figure` and then add to this figures `axes` (and axes will represent a plot). Then the usual usual command are methods of this `axes`. Methodes that read or write a parameter of the `axes` is prefixed with `set` or `get`. Functions that add an element to the plot are the same.

The example above will then be :

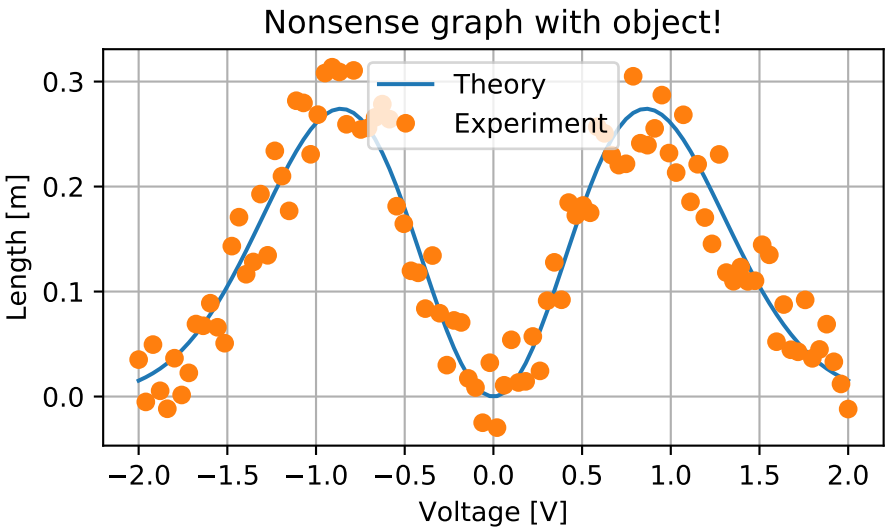
```
from matplotlib.pyplot import figure
import numpy as np
X = np.linspace(-2,2, 100)
Y = np.sin(X)**2*np.exp(-X**2)
Y_noise = Y + .1*(np.random.rand(len(X))-0.5)
f = figure(figsize=(5,3)) # The size can be specified
ax = f.subplots(1, 1)
ax.plot(X, Y, label=u"Theory")
ax.plot(X, Y_noise,'o', label=u"Experiment")
ax.set_xlabel(u'Voltage [V]')
ax.set_ylabel(u'Length [m]')
ax.set_title("Nonsense graph with object!")
ax.legend()
ax.grid(True)
f.tight_layout()
```

Even if using object is slightly more cumbersome, it has advantages. You don't need to import all the functions that you will use. You can also pass the axes as a parameter for a function that will do the plot. This allow to factorize the code. For example you can think of the following function :

```
from matplotlib.pyplot import figure
import numpy as np
import scipy.optimize

def plot_fit(data_x, data_y, fit_function, ax):
    ax.plot(data_x, data_y, 'o', label='data')
    popt, pcov = scipy.optimize.curve_fit(fit_function, data_x, data_y)
```

(suite sur la page suivante)



(suite de la page précédente)

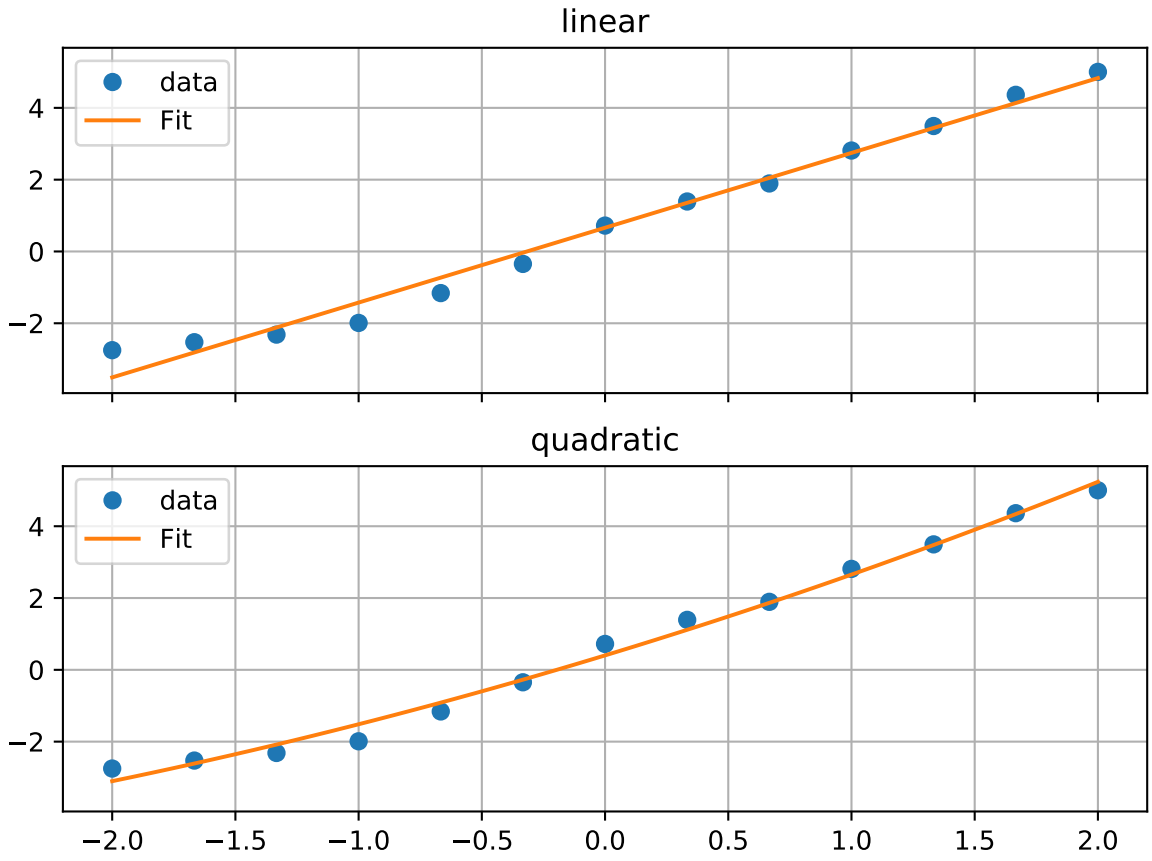
```
    x_plot = np.linspace(data_x.min(), data_x.max())
    ax.plot(x_plot, fit_function(x_plot, *popt), label='Fit')
    ax.grid()
    ax.legend()

f = figure()
X = np.linspace(-2,2, 13)
Y = 0.2*X**2 + 2*X + 0.3
Y_noise = Y + 1*(np.random.rand(len(X))-0.5)
ax1, ax2 = f.subplots(2, 1, sharex=True, sharey=True)
plot_fit(X, Y_noise, lambda x, a, b:a*x + b, ax1)
ax1.set_title("linear")
plot_fit(X, Y_noise, lambda x, a, b, c:a*x**2 + b*x +c, ax2)
ax2.set_title("quadratic")
f.tight_layout()
```

With object, you can also deal with the figure without displaying the figure. This can be usefull if you want to automatically create a PDF of the figure. This is perforned using specific backend. The backend specify how and where to dusplay figure. Obviously the backend by default in jupyter notebook, which display the figure in the HTML, is different from backend of python, that display the figure in a new windows. The `Agg` backend allows you to not display the figure. The following example will create directly a pdf file

```
import matplotlib as mpl
mpl.use('Agg')
from matplotlib.pyplot import figure

fig = plt.figure()
ax = fig.subplots(1, 1)
ax.plot(range(10))
fig.savefig('temp.pdf')
```



6.3 Autres librairies scientifiques

6.3.1 Scipy

La librairie scipy regroupe un grand nombre d'algorithme utilisé pour résoudre des problèmes numériques. Elle vient en complément de numpy. Numpy va contenir uniquement des algorithme de base alors que scipy se veut être exhaustive. On trouvera dans scipy, en autre :

— Toutes les fonctions spéciales (bessel, elliptique, ...)
— Des algorithmes d'optimisation (minimum, moindre carré, ...)
— Des algorithmes pour résoudre des équations différentielles (méthodes avec pas adaptatif, ...)
— Algèbre linéaire (diagonalisation, ...)
— Traitement du signal (densité spectrale de puissance, filtre numérique, ...)

Beaucoup d'algorithme de scipy (et de numpy) sont issus de librairie open source écrite en C ou Fortran et optimisées depuis plusieurs années. Il est illusoire d'essayer de faire mieux pour résoudre un problème générique.

6.3.2 Sympy

Sympy est la librairie de calcul formel de référence pour Python. Elle permet de manipuler des expressions algébriques et de faire des opération dessus. Elle n'est pas au niveau de Mathematica, mais fonctionne bien.

Voici quelques exemples :

```
from sympy import * init_printing() # Pour un affichage graphique
```

x = Symbol("x") a = exp(-x**2) pprint(a.series(x)) pprint(a.diff(x))

On peut facilement utiliser la syntaxe de Python pour générer des expressions sympy :

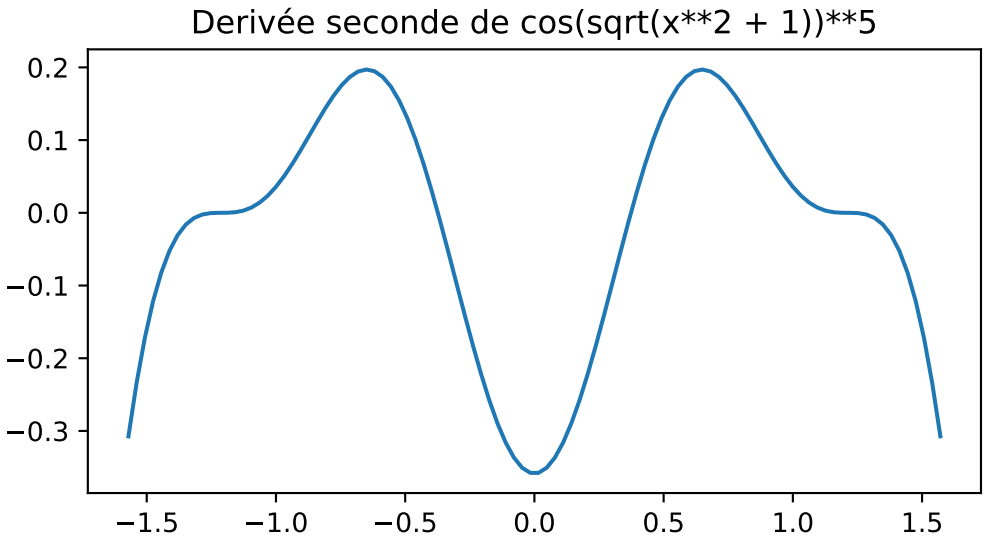
```
>>> from sympy import *
>>> # polynome d'ordre 10
>>> res = 0
>>> x = Symbol('x')
>>> for i in range(8):
...     coef = Symbol('a_{}'.format(i))
...     res += coef*x**i
>>> pprint(res)
                    2         3         4         5         6         7
a_0 + a_1*x + a_2*x  + a_3*x  + a_4*x  + a_5*x  + a_6*x  + a_7*x
```

Il est possible de convertir une expression en une fonction numpy, ce qui permet de la tracer facilement ou de faire des calculs dessus :

```
from matplotlib.pyplot import figure
from numpy import linspace
from sympy import Symbol, cos, sqrt, diff, lambdify

x = Symbol('x')
f = cos(sqrt(1+x**2))**5
f_seconde = diff(f, x, x)
f_seconde_numpy = lambdify(x, f_seconde, 'numpy')


X = linspace(-np.pi/2, np.pi/2, 100)
Y = f_seconde_numpy(X)
fig = figure(figsize=(6, 3))
ax = fig.subplots(1, 1)
ax.plot(X,Y)
ax.set_title('Derivée seconde de {}'.format(f))
```



Enfin, notons que l'on peut définir n'importe quelle expression formelle avec sympy. La package `sympy.physics.quantum` permet par exemple de manipuler des opérateurs ou des états quantique.