

Python - cours intensif pour les scientifiques

Partie 1 : Présentation de Python

Par Rick Muller - [Raphaël Seban](#) (traducteur)

Date de publication : 23 juillet 2014

Pour de nombreux scientifiques, Python est LE langage de programmation par excellence, car il offre de grandes possibilités en analyse et modélisation de données scientifiques avec relativement peu de charge de travail en termes d'apprentissage, d'installation ou de temps de développement. C'est un langage que vous pouvez intégrer en un week-end puis utiliser une vie durant.

Les commentaires et les suggestions d'amélioration sont les bienvenus, alors, après votre lecture, n'hésitez pas.

En complément sur Developpez.com

- [Condensé Python pour les scientifiques - Partie 2](#)
- [Tutoriel Matplotlib](#)

I - Pourquoi Python ?.....	3
II - Ce qu'il faut installer.....	3
III - Présentation de Python.....	4
III-A - Utiliser Python comme une calculatrice.....	5
III-B - Chaînes de caractères.....	7
III-C - Listes.....	8
III-D - Itérations, indentations et blocs d'instructions.....	10
III-E - Échantillonnage.....	12
III-F - Booléens et valeurs de vérité.....	13
III-G - Exemple de code : la suite de Fibonacci.....	16
III-H - Fonctions.....	17
III-I - Récursivité et factorielles.....	18
III-J - Deux nouvelles structures de données : tuples et dictionnaires.....	19
III-K - Tracer des graphiques avec Matplotlib.....	21
IV - Synthèse de la première partie.....	23
V - Références.....	24
V-A - Ressources pédagogiques.....	24
V-B - Documents IPython Notebook pour les durs à cuire.....	24
V-C - Packages Python pour les scientifiques.....	25
V-D - Liens super cools.....	25
VI - Remerciements.....	25
VII - Remerciements Developpez.....	26

I - Pourquoi Python ?

Pour de nombreux scientifiques, Python est LE langage de programmation par excellence, car il offre de grandes possibilités en analyse et modélisation de données scientifiques avec relativement peu de charge de travail en termes d'apprentissage, d'installation ou de temps de développement. C'est un langage que vous pouvez intégrer en un week-end puis utiliser une vie durant.

Le **tutoriel Python** du site officiel est un bon point de départ pour se faire une idée du langage. En complément, j'avais enseigné voilà quelques années un **cours Python** à un groupe de chimistes théoriciens, à une époque où je m'inquiétais de la domination croissante des logiciels éditoriaux sur les solutions développées en interne. Je voulais mettre l'accent sur le fait que les scientifiques de laboratoire devaient être plus productifs : analyser les sorties résultant d'autres programmes, construire des modèles simples, expérimenter la POO (Programmation Orientée Objet), augmenter les possibilités de Python grâce au C et faire des interfaces utilisateur graphiques (GUI) simples.

J'essaie de faire quelque chose de très similaire ici, aller droit au but et me concentrer sur ce dont les scientifiques ont réellement besoin. L'année dernière, le **projet IPython** a mis en place une interface de publication de notes (IPython Notebook) particulièrement intéressante. Beaucoup de gens ont publié d'excellentes notes IPython et j'ai eu grand plaisir à les lire. Parmi mes notes préférées, vous trouverez :

- les **publications** de Rob Johansson, incluant **Calculs Scientifiques avec Python** et **Calculs en Physique Quantique avec QuTiP** ;
- **tracés « à main levée »** avec matplotlib ;
- une **collection de notes IPython** intéressantes.

Je trouve que le format d'échange Notebook de IPython est à la fois un moyen simple et productif au quotidien et à la fois un excellent moyen de partager ce que j'ai fait, comment je l'ai fait et pourquoi cela compte pour mes collaborateurs. Je me suis même surpris à guetter sans cesse le **Reddit rubrique IPython** dans l'espoir de voir paraître de nouvelles notes IPython Notebook. Toujours dans cette optique de publier un maximum de notes pour le bénéfice de tous, je me suis dit que je devrais essayer de réécrire quelques morceaux de ce que j'ai essayé de transmettre à l'époque dans mon cours Python, remis au goût du jour après 15 années de développement Python, Numpy, Scipy, Matplotlib et IPython, de même que de partager ma propre expérience dans l'utilisation quasi quotidienne de Python.

II - Ce qu'il faut installer

Le langage Python se subdivise en deux branches principales : Python 2 (ancienne syntaxe) et Python 3 (nouvelle syntaxe). Cette divergence est délibérée : certaines innovations de Python 3 ne pouvant pas assurer de rétrocompatibilité avec Python 2, l'équipe de développement Python a fait le choix d'une transition sur le moyen terme durant laquelle les nouvelles fonctionnalités seraient progressivement ajoutées au nouveau langage, tout en continuant la maintenance des anciennes versions de Python 2, ceci dans le but de faciliter cette transition entre Python 2 et Python 3. Nous avons à présent (2013) parcouru la moitié du chemin et pour la première fois, j'envisage de migrer vers Python 3.

Note du traducteur : l'équipe de développement Python a que Python 2 s'arrêterait à la version 2.7 et qu'il n'y aurait désormais plus aucune nouvelle version mineure de Python 2.

Toutefois, je rédigerai ces notes selon la syntaxe Python 2, d'une part parce que c'est la version du langage que j'utilise au quotidien et d'autre part parce que je maîtrise mieux cette version que la nouvelle. Maintenant, si ces notes s'avèrent populaires, je me ferais une joie de les réécrire pour Python 3.

En gardant ceci à l'esprit, ces notes présupposent que vous avez une distribution Python qui inclut :

- **Python** version 2.7 ;
- **Numpy**, la librairie d'extensions pour l'algèbre linéaire et les tableaux multidimensionnels ;
- **Scipy**, la librairie pour la programmation scientifique ;

- **Matplotlib**, la librairie de tracés et graphiques par excellence ;
- **IPython**, accompagné de ses librairies pour l'interface Notebook.

Il existe un moyen plus facile d'installer tous ces packages - et d'autres encore - sur Mac™, Windows™ et Linux™ : la distribution **Enthought Python** (EPD - Enthought Python Distribution), qui semble avoir migré vers une nouvelle mouture, Enthought Canopy. Enthought® est une entreprise commerciale très impliquée dans le domaine du développement Python scientifique et de ses nombreuses applications. Vous pouvez acheter une licence commerciale EPD ou essayer la **version gratuite**.

D'autres alternatives, si vous ne souhaitez pas opter pour EPD.

Linux™ : la plupart des distributions Linux™ disposent désormais de gestionnaires d'installation de paquets automatisés. Redhat™ a *yum*, Ubuntu™ a *apt-get* en console ou encore la « logithèque Ubuntu » en version graphique (GUI). À ma connaissance, tous les packages cités précédemment devraient être disponibles via ces gestionnaires d'installation.

Mac™ : j'utilise **Macports**, qui dispose généralement des toutes dernières versions de tous ces packages.

Windows™ : le package **PythonXY** a tout ce dont vous avez besoin ; installez ce package puis allez dans Démarrer > PythonXY > Invite de commandes > Serveur IPython Notebook.

Le « cloud » : le présent document n'est pas encore publié sur l'interface **IPython notebook viewer**, mais devrait l'être prochainement, du moins en lecture seule. Je garde un œil sur **Wakari™** de **Continuum Analytics®**, une interface IPython Notebook orientée « cloud ». Il semblerait que Wakari admette des comptes utilisateurs gratuits. Continuum® est une entreprise créée par des transfuges de chez Enthought®, Numpy, Scipy qui se sont focalisés sur le http://fr.wikipedia.org/wiki/Big_data (traitement de vastes champs de données).

Continuum® fournit aussi le package multiplateforme **Anaconda**, que je ne manque pas de surveiller, évidemment.

III - Présentation de Python

Ceci est une présentation rapide de Python. Il existe de très nombreuses ressources pour étudier ce langage plus en profondeur. J'ai regroupé des liens très utiles dans une liste en fin de ce document. Pour approfondir la question, vous avez le **tutoriel Python** officiel, ainsi que l'excellent **Learn Python the Hard Way** de Zed Shaw.

Les exemples qui vont suivre se basent sur l'interface IPython Notebook. Vous trouverez une présentation détaillée dans la **documentation IPython Notebook**, qui fournit même **une vidéo** de démonstration. Vous devriez aussi profiter de votre incommensurable temps libre pour vous plonger dans le **tutoriel IPython**.

Pour résumer, les « notebooks » sont composés de cellules de code - généralement suivies de cellules de résultat - et de cellules de texte. Les cellules de code commencent toujours par « In [nnn] : » avec nnn le numéro incrémental de l'entrée saisie. Si vous placez le curseur dans une cellule de code et que vous pressez <Shift><Enter> au clavier, le code contenu dans cette cellule sera envoyé à l'interpréteur Python et le résultat s'affichera dans une cellule résultat « Out [nnn] : », avec nnn le numéro incrémental de l'entrée qui a généré le résultat. Pour bien comprendre tout cela, le mieux reste encore d'expérimenter par soi-même, en éditant un document notebook préexistant et en s'aidant de la **documentation IPython Notebook**, de la **vidéo** de démonstration ainsi que du **tutoriel IPython**.

IP[y]: Notebook

La suite du chapitre est consultable également au format « notebook » en suivant le lien :
Notes : Partie 1 - Présentation de python

III-A - Utiliser Python comme une calculatrice


Avant, j'utilisais une calculatrice ; mais ça, c'était avant :

```
2+2
```

4

```
(50-5*6) / 4
```

5

 si vous entrez ces exemples dans un document IPython Notebook, pensez à presser [Shift] [Enter] au clavier pour exécuter le code Python contenu dans la cellule de code et obtenir ainsi l'affichage du résultat.

Il y a parfois quelques mauvaises surprises, à comparer avec une calculette.

```
7/3
```

2

La division entière Python - tout comme en C ou en Fortran - enlève le reste de la division et ne retourne qu'un nombre entier. Ceci est vrai en Python 2, mais pas en Python 3, qui lui retourne un nombre décimal à virgule flottante. Vous pouvez obtenir un aperçu du comportement Python 3 en Python 2 en important le module spécial « **future** » :

```
from __future__ import division
```

```
7/3.
```

2.3333333333333335

```
7/float(3)
```

2.3333333333333335

Dans les dernières lignes, nous sommes allés un peu vite et nous devrions peut-être nous y attarder quelque peu. Nous avons vu - bien que brièvement - deux nouveaux types de données : les entiers (*int*) et les nombres à virgule flottante (*float*), connus parfois - bien que ce soit incorrect - sous l'appellation : « nombres décimaux ».

Nous avons aussi vu pour la première fois, l'utilisation du mot-clé Python `import`. Python a un nombre considérable de bibliothèques d'extension fournies d'office lors de l'installation du langage. Pour faire simple, la plupart des fonctionnalités qui ne sont pas fondamentales au langage ne sont pas accessibles directement. Si vous voulez une ou plusieurs fonctionnalités données, vous devez importer dans la mémoire de l'interpréteur le module contenant ces fonctionnalités en citant son nom. Cela permet de ne pas encombrer inutilement le système : on ne prend que ce dont on a besoin. Par exemple, si vous voulez faire des opérations mathématiques un peu plus élaborées que la simple arithmétique, vous disposez du module `math` qui contient de nombreuses fonctions et constantes très utiles. Pour utiliser la fonction racine carrée (*square root* en anglais - `sqrt(x)`), vous devez tout d'abord réclamer :

```
from math import sqrt
```

et ensuite seulement saisir :

```
sqrt(81)
```

9.0

ou, si vous devez utiliser plusieurs fonctions, préférez appeler le module `math` en entier :

```
import math
math.sqrt(81)
```

9.0

Pour définir une variable, utilisez le signe égal (=) :

```
width = 20
length = 30
area = length * width
area
```

600

Si vous faites référence à une variable qui n'a pas encore été définie, vous obtiendrez un message d'erreur :

```
volume
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-9-0c7fc58f9268> in <module>()
----> 1 volume

NameError: name 'volume' is not defined
```

Il faut donc toujours définir une variable **avant** de s'y référer :

```
depth = 10
volume = area*depth
volume
```

6000

On peut nommer une variable presque comme bon nous semble. Le nom de la variable doit toutefois débiter par une lettre de l'alphabet ou le signe souligné '_' (underscore) ; les chiffres ne sont pas admis en début de nom de variable.

NdT : la <http://legacy.python.org/dev/peps/pep-0008/> de Python recommande malgré tout de suivre des <http://legacy.python.org/dev/peps/pep-0008/#naming-conventions> afin de faciliter la relecture et l'interchangeabilité du code entre les développeurs. En gros, un nom de variable ne devrait contenir que des lettres minuscules non accentuées allant de 'a' à 'z' (alphabet anglais), le signe souligné '_' (underscore) et des chiffres allant de '0' à '9'.

Certains noms sont en réalité des mots-clés réservés par le langage et ne doivent pas être utilisés pour nommer des variables :

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

Si vous tentez de définir un mot-clé réservé comme une variable, vous aurez un message d'erreur :

```
return = 0
```

```
File "<ipython-input-11-2b99136d4ec6>", line 1
```

```
return = 0
      ^
SyntaxError: invalid syntax
```

Vous trouverez plus d'information dans le [tutoriel Python](#) ainsi que dans le [tutoriel IPython](#) qui vient en complément.

III-B - Chaînes de caractères

Les chaînes de caractères (*str*) sont des suites de signes imprimables qui peuvent être définies soit à l'aide de guillemets simples (') placés de part et d'autre de la chaîne :

```
'Hello, World!'
```

'Hello, World!'

soit à l'aide de guillemets anglais (") :

```
"Hello, World!"
```

'Hello, World!'

Mais pas les deux en même temps, à moins que vous ne vouliez incorporer les uns dans les autres comme signes simplement imprimables (non déclaratifs).

```
"He's a Rebel"
```

"He's a Rebel"

```
'She asked, "How are you today?"'
```

'She asked, "How are you today?"'

Tout comme les deux types de données que nous avons vu précédemment (entiers *int* et nombres à virgule flottante *float*), il est possible d'assigner une chaîne de caractères (*str*) à une variable :

```
greeting = "Hello, World!"
```

Le mot-clé réservé `print` permet d'afficher des chaînes de caractères dans la console :

```
print greeting
```

Hello, World!

Plus globalement, ce mot-clé permet d'afficher toute sorte de résultats :

```
area = 600
print "The area is ", area
```

The area is 600

Dans l'exemple ci-dessus, le nombre 600 stocké dans la variable `area` est automatiquement converti en chaîne de caractères par `print` avant d'être affiché comme résultat dans la console.

Vous pouvez utiliser l'opérateur plus (+) pour concaténer (enchaîner) plusieurs chaînes de caractères entre elles :

```
statement = "Hello," + "World!"  
print statement
```

Hello,World!

Lors d'une concaténation, pensez à rajouter les **espaces blanches** qui pourraient manquer à l'intérieur même des déclarations de chaînes de caractères :

```
statement = "Hello, " + "World!"  
print statement
```

Hello, World!

Vous pouvez bien sûr utiliser l'opérateur plus (+) plusieurs fois dans une même expression :

```
print "This " + "is " + "a " + "longer " + "statement."
```

This is a longer statement.

Il existe d'autres moyens plus efficaces de concaténer de nombreux mots entre eux, mais l'opérateur plus (+) demeure très pratique pour assembler quelques chaînes de caractères entre elles, vite fait.

III-C - Listes

En programmation, il arrive fréquemment de vouloir regrouper des éléments similaires au sein d'une même structure. Python fournit une structure de ce type : la liste (type *list* ou []).

```
days_of_the_week = ["Sunday", "Monday",  
                    "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]
```

On accède aux éléments d'une liste par leur numéro d'**index** (position de l'élément dans la liste) :

```
days_of_the_week[2]
```

'Tuesday'

Le comptage des indices en Python - comme en C mais pas comme en Fortran - débute à zéro (0) pour le premier élément, 1 pour le second, 2 pour le troisième, etc. Dans notre exemple, l'élément d'index 0 est "Sunday", l'élément d'index 1 est "Monday" et ainsi de suite. Si vous voulez accéder au *nième* élément à partir de la fin de la liste, vous pouvez utiliser des indices négatifs. Par exemple, l'élément d'index -1 sera toujours le dernier élément de la liste, quelle que soit sa longueur, l'élément d'index -2 sera l'avant-dernier, etc. :

```
days_of_the_week[-1]
```

'Saturday'

On peut agrandir une liste référencée par une variable grâce à la méthode `append()` :

```
languages = ["Fortran", "C", "C++"]  
languages.append("Python")  
print languages
```

['Fortran', 'C', 'C++', 'Python']

La fonction `range()` permet de générer des listes séquentielles de nombres :


```
range(10)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Notez que `range(n)` génère une liste de valeurs entières allant de 0 à n-1 (n exclus). Si vous voulez démarrer votre liste à une autre valeur, utilisez plutôt `range(start, stop)` :

```
range(2, 8)
```

```
[2, 3, 4, 5, 6, 7]
```

La liste créée ci-dessus va de 2 (inclus) à 8 (exclus), donc de 2 à 7 avec un pas incrémental entier de 1 entre chaque élément. Vous pouvez également choisir le pas incrémental de votre liste grâce à `range(start, stop, step)` :

```
evens = range(0, 20, 2)
evens
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
evens[3]
```

```
6
```

Les listes Python autorisent le mix de plusieurs types de données parmi les éléments. Vous pouvez tout à fait écrire, par exemple :

```
["Today", 7, 99.3, ""]
```

```
['Today', 7, 99.3, ""]
```

Toutefois, il est recommandé - bien que pas du tout obligatoire - d'utiliser les listes pour regrouper des éléments de même type entre eux au sein d'une même structure. Si vous voulez regrouper des éléments de types hétérogènes au sein d'une structure, préférez utiliser les tuples - que nous aborderons un peu plus tard.

NdT : cette recommandation de l'auteur est plus que discutable : les tuples sont conçus pour être des structures fixes (immuables) et les listes des structures évolutives (mutables) - rien à voir donc, avec le fait que leurs éléments soient composites ou non.

Utilisez la fonction `len(liste)` pour obtenir le nombre d'éléments contenus dans une liste :

```
help(len)
```

```
Help on built-in function len in module __builtin__:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or mapping.
```

```
len(evens)
```

```
10
```

III-D - Itérations, indentations et blocs d'instructions

L'une des choses les plus utiles que vous puissiez faire avec des listes est de les parcourir avec une boucle d'itération, c'est-à-dire de passer en revue les éléments de la liste un à un, l'un après l'autre. En Python, nous utiliserons la syntaxe `for ... in ...` pour de telles itérations :

```
days_of_the_week = ["Sunday", "Monday",  
                    "Tuesday", "Wednesday",  
                    "Thursday", "Friday", "Saturday"]  
for day in days_of_the_week:  
    print day
```

Sunday

Monday

Tuesday

Wednesday

Thursday

Friday

Saturday

Ce petit bout de code parcourt chaque élément d'une liste référencée par la variable `days_of_the_week` et assigne cet élément - lorsque son tour vient - à la variable `day`. Vient ensuite l'exécution du bloc d'instructions indenté - ici, uniquement la ligne `print day` - qui se sert de la variable d'itération `day`. Lorsque le programme a fini de parcourir tous les éléments de la liste, un à un, l'un après l'autre, il sort du bloc d'instructions indenté et continue son chemin.

Presque tous les langages de programmation délimitent les blocs d'instructions d'une manière ou d'une autre. En Fortran, on utilise des déclarations `END` (`ENDDO`, `ENDIF`, etc). En C, C++ et Perl, on utilise des accolades `{instruction ; instruction ; instruction...}`.

Python, lui, utilise le signe deux-points (`:`) en fin de ligne suivi d'un retour chariot et d'une indentation fixe pour délimiter de tels blocs d'instructions. Pour rappel, une indentation est un décalage en colonne du code que l'on effectue en insérant soit des tabulations (signe `'\t'`) soit des **espaces blanches** (en général, quatre espaces par niveau d'indentation) et qui se répète d'une ligne sur l'autre. En Python, toutes les lignes de code qui se succèdent avec un même niveau d'indentation sont ainsi considérées comme appartenant au même bloc d'instructions indenté. Dans l'exemple précédent, nous avons une seule ligne de bloc indenté, mais nous pourrions en avoir plusieurs, exemple :

```
for day in days_of_the_week:  
    statement = "Today is " + day  
    print statement
```

Today is Sunday

Today is Monday

Today is Tuesday

Today is Wednesday

Today is Thursday

Today is Friday

Today is Saturday

La fonction `range()` s'avère particulièrement utile lorsqu'il s'agit de parcourir des intervalles numériques avec une boucle `for ... in ...` :

```
for i in range(20):  
    print "The square of ",i," is ",i*i
```

The square of 0 is 0

The square of 1 is 1

The square of 2 is 4

The square of 3 is 9

The square of 4 is 16

The square of 5 is 25

The square of 6 is 36

The square of 7 is 49

The square of 8 is 64

The square of 9 is 81

The square of 10 is 100

The square of 11 is 121

The square of 12 is 144

The square of 13 is 169

The square of 14 is 196

The square of 15 is 225

The square of 16 is 256

The square of 17 is 289

The square of 18 is 324

The square of 19 is 361

III-E - Échantillonnage

Les listes Python et les chaînes de caractères ont un point commun à peine soupçonnable : toutes deux peuvent être considérées comme itérables, des séquences à parcourir élément par élément. Vous savez déjà que l'on peut parcourir les éléments d'une liste un à un, l'un après l'autre. Vous pouvez faire la même chose avec les lettres d'une chaîne de caractères, lettres qui sont considérées comme les éléments d'une séquence itérable :

```
for letter in "Sunday":  
    print letter
```

S

u

n

d

a

y

Ce n'est que rarement utile. En revanche, l'échantillonnage d'une séquence, c'est-à-dire l'extraction d'une portion contiguë donnée d'une séquence, l'est nettement plus. Nous savons déjà extraire le premier élément d'une liste grâce à son numéro d'**index** :

```
days_of_the_week[0]
```

'Sunday'

Maintenant, si nous voulons extraire une portion de liste contenant les deux premiers éléments à partir de la liste référencée par la variable `days_of_the_week`, nous pouvons utiliser la notation :

```
days_of_the_week[0:2]
```

['Sunday', 'Monday']

ou plus simplement (notation abrégée) :

```
days_of_the_week[:2]
```

['Sunday', 'Monday']

Pour obtenir une portion de liste contenant les derniers éléments de la liste de référence, nous pouvons utiliser un échantillonnage négatif :

```
days_of_the_week[-2:]
```

['Friday', 'Saturday']

ce qui est en accord parfait avec la notation en indices négatifs adressant les derniers éléments de la liste.

Vous pouvez référencer une portion de liste par une variable :

```
workdays = days_of_the_week[1:6]
print workdays
```

`['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']`

Les chaînes de caractères étant elles-mêmes des séquences, vous pouvez tout à fait leur appliquer un échantillonnage, c'est-à-dire en extraire des portions de chaîne - que l'on nomme généralement des sous-chaînes de caractères :

```
day = "Sunday"
abbreviation = day[:3]
print abbreviation
```

`Sun`

Pour finir, nous pouvons passer un troisième argument à notre échantillonnage qui précisera le pas incrémental d'extraction (un peu comme le troisième argument de la fonction `range(start, stop, step)`) :

```
numbers = range(0,40)
evens = numbers[2::2]
evens
```

`[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]`

Notez que l'omission du second argument dans l'échantillonnage signifie « jusqu'à la fin » de la liste, ainsi la portion démarre à l'index 2, va jusqu'au dernier élément de liste et avance avec un pas incrémental de 2 en 2, d'où la génération d'une liste de naturels pairs inférieurs à 40.

III-F - Booléens et valeurs de vérité

Nous venons de découvrir quelques nouveaux types de données. Nous avons vu les entiers (*int*), les nombres à virgule flottante (*float*), les chaînes de caractères (*str*) et les listes (*list* ou `[]`), qui sont des structures capables de contenir des éléments. Nous avons aussi vu que les listes pouvaient contenir toute sorte de types de données (y compris d'autres listes). Nous avons appris à afficher des résultats dans la console avec `print` et à parcourir les éléments d'une séquence itérable un à un, les uns après les autres avec la boucle `for ... in ...`. Nous allons à présent découvrir le type booléen (*bool*) qui ne peut accepter que deux valeurs : `True` (vrai) et `False` (faux).

En programmation, nous avons toujours besoin de la notion de condition pour permettre à un programme de s'adapter à différents cas de figure. Si aujourd'hui est lundi alors je dois aller travailler, mais si c'est dimanche alors je peux rester faire la grasse matinée. Pour pouvoir accomplir ce genre de choses en Python, on fait appel à des expressions booléennes qui ne peuvent revêtir que deux possibilités - ou bien l'expression est vraie ou bien elle est fausse - et à la syntaxe `if condition` qui permet de contrôler le flux du programme grâce à ces valeurs booléennes.

Par exemple :

```
if day == "Sunday":
    print "Sleep in"
else:
    print "Go to work"
```

`Sleep in`

Devinette : pourquoi ce bout de code résulte par « Sleep in » ? Quelle est la valeur de la variable `day` au moment de son évaluation dans l'expression booléenne de `if` ?

Voyons tout cela plus en détail. Tout d'abord, notez l'expression booléenne :

```
day == "Sunday"
```

True

En l'évaluant seule - comme nous venons de le faire - nous nous apercevons que cette expression est vraie (**True**). L'opérateur double-égal (==) effectue un test d'égalité entre les opérandes placés de part et d'autre. Si ses opérandes sont égaux, l'expression a la valeur **True** (vrai) et sinon, l'expression a la valeur **False** (faux). Dans notre exemple, l'opérateur compare d'un côté le contenu de la variable `day` et de l'autre la valeur intrinsèque de la chaîne de caractères "Sunday". Comme visiblement la variable `day` doit contenir la chaîne de caractères "Sunday", le test d'égalité obtient logiquement la valeur de vérité **True** (vrai).

La déclaration `if condition:` se termine par un caractère deux-points (:) ce qui implique qu'un bloc d'instructions indenté devra se trouver à la ligne suivante. Si l'expression condition est évaluée à **True** (vrai), c'est ce bloc indenté qui sera exécuté. Dans le cas contraire, si l'expression est évaluée à **False** (faux), ce bloc d'instructions indenté sera tout bonnement ignoré. En reprenant notre exemple, comme l'expression est évaluée à **True**, c'est bien le bloc indenté `print "Sleep in"` qui est exécuté, d'où le résultat final.

Le premier bloc d'instructions indenté est suivi par la déclaration `else:` qui exécutera le bloc d'instructions indenté se trouvant à la ligne suivante si tout ce qui a été évalué auparavant s'est avéré faux (**False**). Dans notre exemple, le test d'égalité s'étant avéré **True** (vrai), le bloc d'instructions indenté qui suit `else:` sera tout bonnement ignoré.

En Python, vous pouvez comparer toute sorte d'expressions et toute sorte de types de données :

```
1 == 2
```

False

```
50 == 2*25
```

True

```
3 < 3.14159
```

True

```
1 == 1.0
```

True

```
1 != 0
```

True

```
1 <= 2
```

True

```
1 >= 1
```

True

Vous noterez la présence de nouveaux opérateurs de comparaison - tous booléens - comme strictement inférieur (<), inférieur ou égal (<=), supérieur ou égal (>=) ou encore différent de (!=).

Le test `1 == 1.0` évalué à `True` (vrai) est particulièrement intéressant : les opérandes `1` (*int*) et `1.0` (*float*) ne sont pas du même type de données ; en revanche, ils sont de même valeur numérique. Pour comparer le type de données d'un objet plutôt que sa valeur intrinsèque, il existe l'opérateur booléen `is` (et son pendant logique `is not`) :

```
1 is 1.0
```

`False`

On peut aussi comparer des listes entre elles :

```
[1, 2, 3] == [1, 2, 4]
```

`False`

```
[1, 2, 3] < [1, 2, 4]
```

`True`

Pour finir, vous pouvez assembler des opérateurs de comparaison ensemble dans une même expression, ce qui produit des notations intuitives :

```
hours = 5  
0 < hours < 24
```

`True`

La syntaxe `if condition: bloc else: bloc` n'est complète qu'avec l'adjonction des clauses alternatives `elif condition:` (`elif` est une abréviation de *else if*) entre *if* et *else*, clauses qui permettent d'évaluer plusieurs cas de figure en cascade. Par exemple :

```
if day == "Sunday":  
    print "Sleep in"  
elif day == "Saturday":  
    print "Do chores"  
else:  
    print "Go to work"
```

`Sleep in`

Bien sûr, nous pouvons mélanger des conditions `if ...` avec des boucles `for ... in ...` afin d'obtenir quelque chose de - presque - intéressant :

```
for day in days_of_the_week:  
    statement = "Today is " + day  
    print statement  
    if day == "Sunday":  
        print "    Sleep in"  
    elif day == "Saturday":  
        print "    Do chores"  
    else:  
        print "    Go to work"
```

`Today is Sunday`

`Sleep in`

`Today is Monday`

Go to work

Today is Tuesday

Go to work

Today is Wednesday

Go to work

Today is Thursday

Go to work

Today is Friday

Go to work

Today is Saturday

Do chores

Ceci relève peut-être d'un sujet pointu, mais les types de données natifs Python ont tous une valeur booléenne qui leur est associée et - à vrai dire - dans les toutes premières versions de Python, il n'y avait pas de type booléen (*bool*) à proprement parler. En gros, tout ce qui pouvait s'évaluer à zéro (0) était considéré faux (*False*) et le reste était considéré vrai (*True*). Vous pouvez extraire la valeur booléenne d'une expression ou d'un type de données (objet) grâce à la fonction `bool(x)`.

```
bool(1)
```

True

```
bool(0)
```

False

```
bool(["This ", " is ", " a ", " list"])
```

True

III-G - Exemple de code : la suite de Fibonacci

En mathématiques, **la suite de Fibonacci** est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle commence généralement par les termes 0 et 1 (parfois 1 et 1) et ses premiers termes sont 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, etc.

Dans les livres qui traitent de programmation, on rencontre fréquemment cet exercice qui consiste à coder la série de Fibonacci pour obtenir les valeurs de *n* termes. Voyons tout d'abord le code, nous discuterons par la suite de sa signification.

```
n = 10
sequence = [0,1]
# cette boucle pourrait poser problème pour n <= 2
for i in range(2,n):
    sequence.append(sequence[i-1]+sequence[i-2])
```



```
print sequence
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

On commence par initialiser une variable nommée `n` avec la valeur entière 10. Ici, la variable `n` représente le nombre de termes à calculer. Ensuite, on affecte une liste Python `[0, 1]` à une variable nommée `sequence`, ces deux valeurs initiales correspondant aux termes de rang 0 et 1 de la suite ($F_0 = 0$ et $F_1 = 1$). Les valeurs initiales doivent être déclarées manuellement car la formulation de la série de Fibonacci requiert la somme des deux termes précédant le terme calculé.

Le calcul commence avec la boucle itérative `for ... in ...` qui affecte à une variable nommée `j` des valeurs entières allant de 2 (inclus) à `n` (exclus). Notez en passant la présence d'un commentaire qui signale un éventuel problème si l'on initialise `n` à une valeur `n <= 2`. Un commentaire Python se caractérise par une phrase humainement intelligible commençant toujours par le signe dièse (`#`) et se terminant en bout de ligne par le retour chariot (saut à la ligne, CR, `\n`), tout simplement. Tout ce qui se trouve entre ce signe dièse (`#`) et la fin de la ligne est ignoré par l'interpréteur Python, vous pouvez donc y inscrire tout ce que vous souhaitez mentionner, y compris des portions de code à ne pas exécuter. Bien pratique pour désactiver temporairement du code. En programmation, un code source est appelé à être lu et relu bien plus souvent qu'écrit, il est donc nécessaire d'indiquer à certains points critiques ce que vous avez fait et d'y expliquer pourquoi vous l'avez fait, quels sont les risques, les problèmes, les indications pour les développeurs qui vous succéderont dans la maintenance de ce code, etc. Dans notre exemple, nous avons signalé un problème que nous aurions facilement pu résoudre avec une condition `if condition`, ce que nous verrons un peu plus tard.

Dans le bloc d'instructions indenté de la boucle `for ... in ...`, nous ajoutons à la liste Python référencée par la variable `sequence` ce qui correspond à la somme des deux termes immédiatement précédents, à savoir les termes d'indices `[i-1]` et `[i-2]`, ce qui est bien conforme à la formulation de la suite de Fibonacci.

En sortie de boucle, lorsque tous les calculs sont effectués, nous affichons le résultat final, c'est-à-dire la liste Python référencée par la variable `sequence` dans son intégralité. Et voilà le travail !

III-H - Fonctions

Nous aimerions à présent pouvoir utiliser ce bout de code pour générer des listes de valeurs de Fibonacci avec divers nombres d'éléments, à la demande. Créons une fonction paramétrique que nous appellerons `fibonacci()` - notez la présence de parenthèses qui permettent de distinguer une fonction d'une variable simple - et plaçons-y tout le code dont nous avons besoin. Pour créer une fonction Python, il suffit d'utiliser le mot-clé réservé `def` en début d'instruction selon la syntaxe `def nom_fonction (paramètres):`.

```
def fibonacci (sequence_length):
    """
        retourne une liste de valeurs calculées
        selon la suite de Fibonacci comprenant
        sequence_length éléments ;
    """
    sequence = [0,1]
    if sequence_length < 1:
        print "La suite de Fibonacci commence avec au moins 1 élément"
        return
    if 0 < sequence_length < 3:
        return sequence[:sequence_length]
    for i in range(2,sequence_length):
        sequence.append(sequence[i-1]+sequence[i-2])
    return sequence
```

Nous pouvons à présent faire appel à cette fonction `fibonacci()` avec différentes valeurs pour le paramètre `sequence_length`. Pour appeler une fonction, il suffit de la mentionner - sans le mot-clé `def` - avec ses parenthèses et ses valeurs de paramètres, s'il y a lieu.

```
fibonacci(2)
```

```
[0, 1]
```

```
fibonacci(12)
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Nous venons de découvrir plusieurs nouveautés. Tout d'abord, notez que la définition d'une fonction en Python s'écrit avec la même syntaxe que pour les blocs d'instructions indentés - la phrase `def nom_fonction(paramètres)` se termine bien par un signe deux-points (`:`) qui indique l'obligation d'écrire un bloc d'instructions indenté à la ligne suivante. On retrouve cette logique partout dans la syntaxe Python. Ensuite, vous remarquerez que la fonction commence par un texte délimité par des guillemets anglais triples (`"""`) : il s'agit d'un *docstring*, un texte explicatif spécial qui permet de documenter automatiquement un module, une classe ou une fonction lors de l'appel de la fonction Python `help()`, ce qui peut s'avérer fort pratique pour les utilisateurs de votre code.

```
help(fibonacci)
```

```
Help on function fibonacci in module __main__:
```

```
fibonacci(sequence_length)
    retourne une liste de valeurs calculées
    selon la suite de Fibonacci comprenant
    sequence_length éléments ;
```

Plus vous documentez vos modules, classes et fonctions avec leurs *docstring* respectifs - toujours placés **en début** de bloc d'instructions indenté - et plus vous rendez votre code exploitable pour d'autres utilisateurs.

Pour finir, notez aussi que dans ce code, nous avons préféré remplacer le commentaire qui signalait un problème par une série de tests *if condition* : avec leurs blocs d'instructions indentés qui traitent le problème et qui affichent un message à l'attention de l'utilisateur, le cas échéant.

NdT : une fonction produit un résultat avec le mot-clé réservé `return` qui oblige l'interpréteur à quitter le bloc de code définissant une fonction en retournant le résultat de l'expression mentionnée à la suite de ce mot-clé - exemple : `return (x+3)` - à l'instruction qui a appelé cette fonction (l'appelant). Notez aussi que l'on peut mentionner plusieurs fois le mot-clé `return` dans une définition de fonction, à divers endroits du code et selon les besoins de contrôle du flux du programme.

III-I - Récursivité et factorielles

Un aspect intéressant de la définition des fonctions est la capacité d'une fonction à s'appeler elle-même. On appelle ce concept la récursivité. Expérimentons la récursivité grâce à la fonction factorielle, qui se définit pour un entier

naturel n par $n! = n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1$ avec comme cas particulier $0! = 1$.

Tout d'abord, notez que nous n'avons pas réellement besoin d'écrire cette fonction en temps normal, puisqu'elle existe déjà dans la librairie `math` fournie d'office avec Python. Voyons ce que nous dit la fonction `help()` à ce sujet :

```
from math import factorial
help(factorial)
```

```
Help on built-in function factorial in module math:
```

```
factorial(...)
    factorial(x) -> Integral

    Find x!. Raise a ValueError if x is negative or non-integral.
```

C'est clairement ce que nous voulons.

```
factorial(20)
```

2432902008176640000

Toutefois, si nous voulions coder cette fonction par nous-mêmes, nous pourrions remarquer que $n! = n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1 \Leftrightarrow n! = n \times (n - 1)!$, ce qui reviendrait à écrire sous forme de code Python :

```
def fact (n):
    if n <= 0:
        return 1
    return n*fact(n-1)
```

```
fact(20)
```

2432902008176640000

La récursivité peut être mathématiquement très élégante et permet en outre de produire des programmes très simples.

NdT : la récursivité s'avère néanmoins particulièrement gourmande en ressources mémoire et en temps machine, notamment lorsqu'il s'agit d'effectuer un très grand nombre d'itérations récursives.

III-J - Deux nouvelles structures de données : tuples et dictionnaires

Avant de terminer notre présentation du langage Python, je voudrais signaler deux autres structures de données très pratiques - et donc très répandues - parmi les programmes Python : le tuple et le dictionnaire.

Un tuple (*tuple* ou *()*) est un objet itérable au même titre que le sont les listes Python (*list* ou *[]*) et les chaînes de caractères (*str*). Un tuple s'écrit en regroupant divers objets dans une liste séparée par des virgules et généralement délimitée par des parenthèses (certains cas autorisent une notation sans parenthèses, *NdT : mais cela reste peu recommandé*) :

```
t = (1, 2, 'hi', 9.0)
t
```

(1, 2, 'hi', 9.0)

Tout comme les listes Python, vous pouvez accéder aux éléments d'un tuple par la notation en indices :

```
t[1]
```

2

Toutefois, contrairement aux listes Python, les tuples sont dits immutables (immuable), ce qui signifie que vous ne pouvez ni les agrandir ni modifier la position des éléments ni même modifier les éléments eux-mêmes. En somme, un tuple est un objet fixe, non modifiable, qui s'utilise tel qu'il a été défini (en lecture seule) :

```
t.append(7)
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-73-50c7062b1d5f> in <module>()
----> 1 t.append(7)

AttributeError: 'tuple' object has no attribute 'append'
```

```
t[1]=77
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-74-03cc8ba9c07d> in <module>()
----> 1 t[1]=77

TypeError: 'tuple' object does not support item assignment
```

Les tuples sont utiles chaque fois que vous souhaitez regrouper des objets au sein d'une même structure sans être pour autant obligés de recourir à une définition de classe (voir plus bas). Par exemple, supposons que vous ayez besoin de coordonnées cartésiennes pour certains objets dans votre programme. Les tuples sont un bon moyen d'effectuer ce regroupement :

```
('Bob', 0.0, 21.0)
```

```
('Bob', 0.0, 21.0)
```

À nouveau, certes ce n'est pas obligatoire, mais une bonne façon de distinguer un tuple d'une liste Python reste de considérer qu'un tuple est un regroupement de données hétérogènes (ici, un nom et des coordonnées x et y), là où une liste Python devrait plutôt être une collection d'objets identiques, comme par exemple ceci :

NdT : je répète, ces considérations de l'auteur sont plus que discutables car elles ne caractérisent en rien les listes et les tuples selon les <https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range> qui en sont faites, ni même l'usage que l'on devrait supposément en faire.

```
positions = [
    ('Bob', 0.0, 21.0),
    ('Cat', 2.5, 13.1),
    ('Dog', 33.0, 1.2)
]
```

On peut utiliser les tuples lorsqu'une fonction a par exemple besoin de retourner plus d'une valeur à la fois. Supposons que nous voulions déterminer la plus petite valeur des coordonnées x et y présentes dans la liste positions définie précédemment. Nous pourrions écrire :

```
def minmax(objects):
    minx = 1e20 # ceci représente un très grand nombre
    miny = 1e20
    for obj in objects:
        name, x, y = obj
        if x < minx:
            minx = x
        if y < miny:
            miny = y
    return minx, miny

x, y = minmax(positions)
print x, y
```

0.0 1.2

Nous venons de faire deux choses que nous n'avons pas encore vues. Tout d'abord, nous avons déployé la valeur de retour d'une fonction dans plusieurs variables en une seule fois, en nous servant du mécanisme d'initialisation par tuple :

```
name, x, y = tuple_values
```

Ensuite, avec `return minx,miny` en fin de définition de fonction, nous retournons le tuple `(minx, miny)` à l'appelant `x, y = minmax(positions)`, ce qui revient à faire une initialisation par tuple `x, y = (minx, miny)` au bout du compte. Ceci aurait été bien compliqué à mettre en œuvre en C++, par exemple.

L'initialisation par tuple permet en outre d'interchanger les valeurs de deux variables en une seule opération (*swap*) :

```
x, y = 1, 2
x, y = y, x
x, y
```

(2, 1)

Les dictionnaires Python (*dict* ou `{}`) sont des structures de données que l'on retrouve dans d'autres langages sous des appellations comme « mappages » ou encore « tableaux associatifs ». Là où une liste Python accède à ses éléments avec un index numérique :

```
mylist = [1, 2, 9, 21]
mylist[2]
```

9

L'index d'un dictionnaire est appelé une clé, et l'entrée qui lui est associée une valeur. Un dictionnaire admet toute sorte d'objets en tant que clé d'indexation. Les dictionnaires suivent la notation `{clé : valeur, clé : valeur, ...}`, exemple :

```
ages = {"Rick": 46, "Bob": 86, "Fred": 21}
print "Rick's age is ", ages["Rick"]
```

Rick's age is 46

On peut aussi créer un dictionnaire grâce aux arguments nommés :

```
dict(Rick=46, Bob=86, Fred=20)
```

{'Bob': 86, 'Fred': 20, 'Rick': 46}

La fonction `len()` permet de compter les éléments de tout itérable ; cela comprend les listes (*list*), les chaînes de caractères (*str*), les tuples (*tuple*) et bien entendu, les dictionnaires (*dict*) :

```
len(t)
```

4

```
len(ages)
```

3

III-K - Tracer des graphiques avec Matplotlib

Nous pouvons généralement mieux appréhender les différentes tendances des séries de données en les visualisant avec un graphique. Python dispose d'une excellente librairie pour faciliter le tracé de graphiques : **Matplotlib**. L'interface IPython Notebook que nous utilisons actuellement intègre d'office cette librairie graphique.

Durant nos exemples, nous avons expérimenté deux fonctions : la suite de Fibonacci et la factorielle. Toutes deux croissent plus vite qu'un polynôme. Mais laquelle croît le plus vite ? Traçons-les. Pour commencer, générons une liste de 10 valeurs de Fibonacci :

```
fibs = fibonacci(10)
```

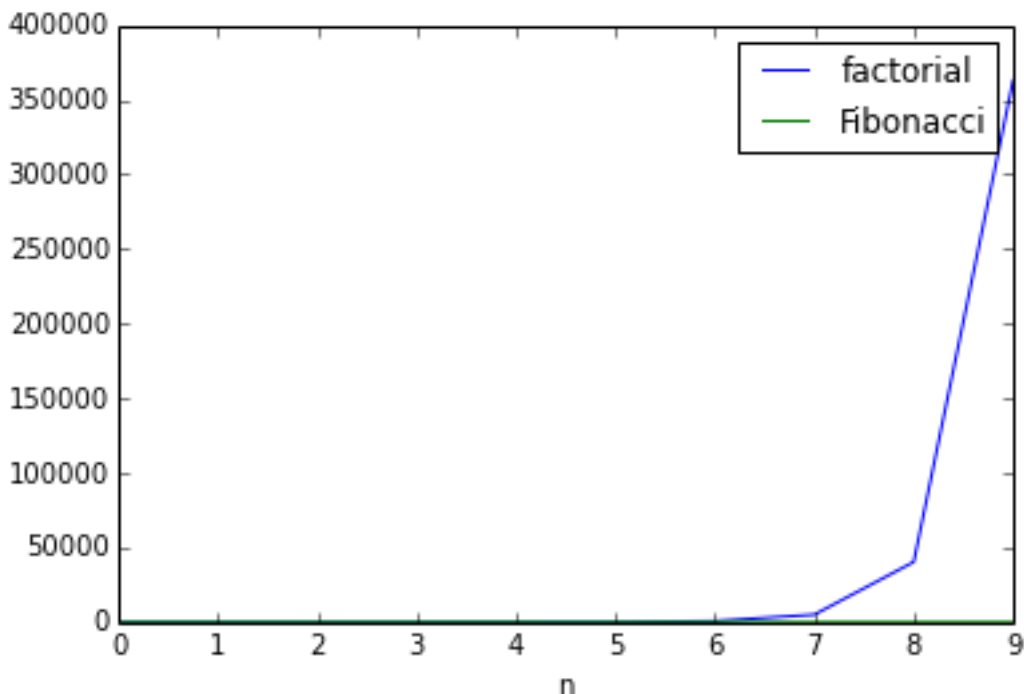
Ensuite, récupérons une liste de 10 valeurs de factorielle :

```
facts = []
for i in range(10):
    facts.append(factorial(i))
```

À présent, utilisons la fonction `plot()` de la librairie Matplotlib pour représenter toutes ces valeurs :

```
figsize=(8,6)
plot(facts,label="factorial")
plot(fibs,label="Fibonacci")
xlabel("n")
legend()
```

<matplotlib.legend.Legend at 0x7f7044612cd0>



La fonction factorielle croît nettement plus vite que la suite de Fibonacci. À vrai dire, on n'arrive même pas à visualiser la courbe des valeurs de Fibonacci sur le graphique. Ce n'est pas vraiment surprenant : une fonction qui multiplie par n chaque itération est appelée à croître plus rapidement qu'une fonction à laquelle l'on ajoute à peu de chose près n à chaque itération.

NdT : on remarquera toutefois que d'après la http://fr.wikipedia.org/wiki/Suite_de_Fibonacci#Avec_la_formule_de_Binet, la suite de Fibonacci semble se comporter comme une suite

$$\varphi = \frac{1 + \sqrt{5}}{2} \approx 1,618033988749$$

géométrique de raison

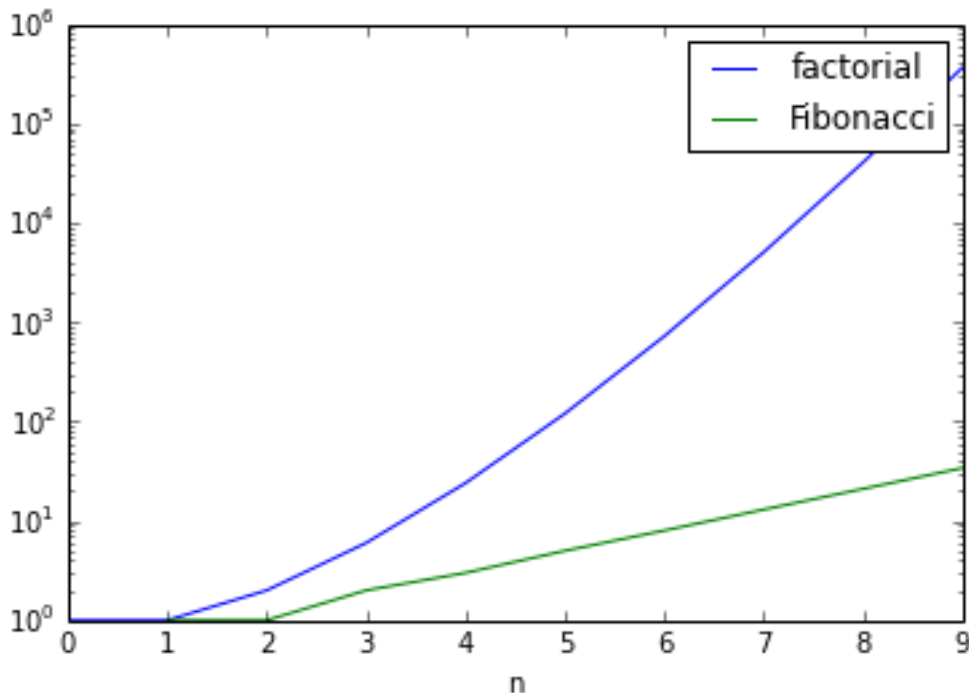
(http://fr.wikipedia.org/wiki/Nombre_d'or).

Utilisons une échelle semi-logarithmique pour visualiser ces courbes plus clairement :

```
semilogy(facts,label="factorial")
semilogy(fibs,label="Fibonacci")
```

```
xlabel("n")
legend()
```

<matplotlib.legend.Legend at 0x7f704441f910>



On peut faire énormément de choses avec Matplotlib. Nous verrons tout cela dans les prochaines rubriques. En attendant, si vous voulez vous faire une idée des capacités de Matplotlib, jetez un œil à la [galerie officielle](#). Le document IPython notebook [Introduction to Matplotlib](#) de Rob Johansson est lui aussi particulièrement intéressant.

IV - Synthèse de la première partie

Il y aurait beaucoup plus à dire sur ce langage que ce que j'ai tenté d'exposer ici. J'ai essayé de rester suffisamment bref pour que vous puissiez vous simplifier la vie (et le travail) en commençant à utiliser Python immédiatement. Par mon expérience personnelle dans l'apprentissage de choses nouvelles, je sais que l'information n'accroche jamais vraiment tant que l'on n'a pas expérimenté les concepts à travers des exemples concrets de la vie courante.

Il ne fait aucun doute que vous serez amenés à approfondir la question. J'ai recensé quelques références intéressantes, notamment le [tutoriel Python](#) officiel et [apprendre Python à la dure](#). Pour compléter, le moment est peut-être venu pour vous de vous familiariser avec la [documentation officielle Python](#) et plus particulièrement avec le [manuel de référence du langage Python](#).

Tim Peters, l'un des tous premiers contributeurs du langage Python - et sans doute l'un des plus prolifiques - a écrit le **Zen of Python** que l'on peut obtenir via la commande `import this` :


The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
```

```
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Quel que soit votre niveau d'expérience en programmation, ces mots valent toujours la peine d'être médités.

À suivre (en cours de traduction) :

-  • *Partie 2 - Modules Numpy et Scipy ;*
- *Partie 3 - Python avancé ;*
- *Partie 4 - Optimiser le code.*

V - Références

V-A - Ressources pédagogiques

- **Documentation Python** officielle, incluant :
 - **Tutoriel Python** ;
 - **Manuel de référence du langage Python**.
- Si vous êtes intéressé(e) par Python 3, la documentation officielle se trouve à **cet endroit**.
- **Tutoriel IPython**.
- **Apprendre Python à la dure**.
- **Plongez dans Python**, surtout si vous vous intéressez à Python 3.
- **Inventer avec Python**, sans doute le meilleur tutoriel pour apprendre à programmer en créant des jeux, pour les jeunes à partir de 10-12 ans.
- **Recettes de cuisine (how-to) en programmation fonctionnelle Python**.
- **Structure et interprétation de programmes informatiques**, écrit en se basant sur le langage Scheme, un dialecte Lisp, mais probablement l'un des meilleurs livres jamais écrits sur la programmation informatique.
- **De l'usage des générateurs** pour les administrateurs système, de David M. Beazley, exemples de code source et **diaporama** au format PDF pour découvrir ce que les fonctions de type « générateur » peuvent faire pour vous.
- **Le module Python de la semaine** est un périodique type blog (*NdT : et aujourd'hui un livre, aussi*) qui propose des analyses approfondies des modules de la librairie standard de Python dans un style facile à comprendre.

V-B - Documents IPython Notebook pour les durs à cuire

- Les **excellents notebooks** de Rob Johansson, incluant **calculs scientifiques avec Python** et **calculs en Physique Quantique avec QuTiP**.
- **Tracés « à main levée » avec matplotlib**.
- Une **galerie de documents IPython Notebook** très intéressante.
- **Analyses de données pluridisciplinaires**, documents IPython Notebook de Hadoop World 2012.
- **Utilisation du module Python Quantities**, unités et quantités en Physique.
- Un autre **module sur les unités** en Physique.

V-C - Packages Python pour les scientifiques

Librairies indispensables :

- **Python** version 2.7 ;
- **Numpy**, la librairie par excellence pour l'algèbre linéaire et les tableaux multidimensionnels ;
- **Scipy**, librairies additionnelles pour la programmation scientifique ;
- **Matplotlib**, une librairie majeure pour les tracés et les graphiques ;
- **IPython**, et ses librairies indispensables pour l'interface IPython Notebook ;
- **Sympy**, maths symboliques avec Python ;
- **Pandas**, librairie pour l'analyse de données et le *big data* en Python.

Autres packages intéressants :

- **PyQuante**, Python Quantum Chemistry, chimie quantique avec Python ;
- **QuTiP**, Quantum Toolbox in Python, la boîte à outils du domaine quantique pour Python ;
- **Python Scientifique**, de Konrad Hinsen ainsi que **MMTK** (Molecular Modelling ToolKit) ;
- **Atomic Simulation Environment (ASE)**, outils de manipulation, d'analyse, de simulation et de visualisation orientés monde de l'atome.

V-D - Liens super cools

- **Moin Moin**, un WikiEngine écrit en Python, ainsi que le **wiki Python** qui l'utilise.
- **Project Euler**, le site des problèmes en programmation qui auraient sans doute intéressé **Euler**. Python est l'un des langages les plus utilisés sur ce site.

VI - Remerciements

Un grand merci à Alex et Tess pour tout !

Remerciements chaleureux à Barbara Muller et Tom Tarman pour leurs précieuses suggestions.

Ce document (version originale :  **A Crash Course in Python for Scientists**) est publié sous licence **Creative Commons Paternité - Partage à l'identique 3.0 non transposé**. Ce document est publié gratuitement, avec l'espoir qu'il sera utile. Merci d'envisager un don au **Fonds de soutien en mémoire de John Hunter** - article en français à **cet endroit**.



Sandia est un laboratoire pluridisciplinaire géré par la Sandia Corporation®, une filiale de la Lockheed Martin Company®, pour le compte des États-Unis d'Amérique, département de l'Énergie, administration de la Sûreté Nucléaire, sous contrat n° DE-AC04-94AL85000.



VII - Remerciements Developpez

Nous remercions Rick Muller qui nous a aimablement autorisé à traduire son cours "**A Crash Course in Python for Scientists**".

Nos remerciements à Raphaël SEBAN (**tarball69**) pour la traduction et à Fabien (**f-leb**) pour la mise au gabarit. Merci aussi à Sébastien (**-Nikopol-**) pour son renfort sur les points mathématiques délicats dans cet article.

Nous remercions également Jacques (**jacques_jean**) pour sa relecture orthographique.