MATH2010-1 Logiciels mathématiques

Notes de cours

Author: Sébastien Labbé, Université de Liège, slabbe@ulg.ac.be
URL: http://www.slabbe.org/Enseignements/MATH2010/notesdecours.pdf
Date: 2016-06-30

jupyter Untitled Last Checkpoint: 9 minutes ago (unsaved changes)

File Edit View Insert Cell Kernel Help

Code CellToolbar

In [1]:
```
from sympy import init_printing
init_printing(use_latex='mathjax')
```

In [2]:
```
from sympy.abc import x,y,z,alpha,epsilon
x ** 2 / (alpha + epsilon)
```

Out[2]: $\frac{x^2}{\alpha + \epsilon}$

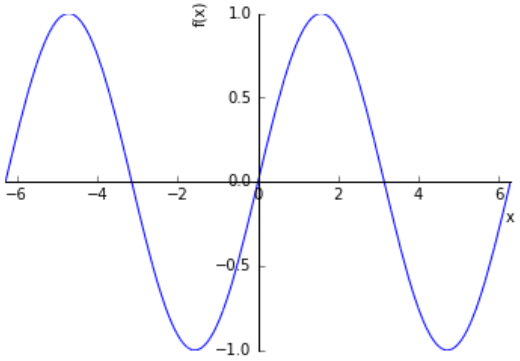
In [10]:
```
from sympy import Integral, integrate, sin, pi, Eq
A = Integral(sin(x), (x,0,pi))
B = integrate(sin(x), (x,0,pi))
Eq(A, B)
```

Out[10]: $\int_0^{\pi} \sin(x)\, dx = 2$

In [11]:
```
integrate(sin(x))
```

Out[11]: $-\cos(x)$

In [14]:
```
%matplotlib inline
from sympy import plot
plot(sin(x), (x,-2*pi, 2*pi))
```



Out[14]:
```
<sympy.plotting.plot.Plot object at 0x7f8daa3f2850>
```

Table des matières

1 Introduction

Ces notes de cours sont rédigés en fonction du nouveau cours de logiciels mathématiques du programme de bachelier en sciences mathématiques de l'Université de Liège. Selon la page du département,

> *le nouveau cours de logiciels mathématiques a pour but de familiariser les étudiants à l'informatique, outil omniprésent en sciences, en entreprises ou encore pour l'enseignement.*

Donné en première année du bachelier et totalisant 10 heures d'enseignement théorique et 20 heures de pratique, il s'agit avant tout de donner un aperçu des possibilités offertes par les logiciels pour faire des mathématiques.

L'environnement de travail proposé pour le cours de *Logiciel mathématiques* est l'environnement scientifique Python permettant d'atteindre les divers objectifs du cours. En effet, le langage Python est un langage *utilisé dans les entreprises* et sa connaissance est un atout pour les chercheurs d'emploi. De plus, les librairies scientifiques de l'environnement Python sont des logiciels libres permettant aux futurs enseignants d'utiliser ces outils *pour l'enseignement dans les écoles secondaires* sans avoir à payer des licences dispendieuses que les écoles n'ont pas les moyens de payer.

Nous utiliserons l'interface Jupyter développée par la communauté IPython. L'interface Jupyter supporte plus de 40 langages de programmation, incluant les langages populaires en sciences comme Python, R, Julia et Scala. Notons qu'il est possible de tester l'utilisation R ou Python à l'adresse try.jupyter.org. Nous nous concentrerons dans la première partie du cours sur la librairie de calcul formel SymPy. Tout ces outils font partie du logiciel de mathématiques Sage et nous couvrirons l'équivalent des quatres premiers chapitres de l'excellente référence libre en français *Calcul mathématique avec Sage* [Sage].

Jupyter et les outils que nous présenterons sont utilisés dans les grandes compagnies (Google, Microsoft, IBM, Nasa) et universités (Berkeley, Northwestern University, George Washington University). Elles sont aussi utilisées par les nouveaux médias tels que BuzzFeed qui a publié le 18 janvier 2016 une analyse de 26000 matchs de tennis professionnels pour identifier des joueurs soupçonnés de matchs truqués. Ou encore par des chercheurs qui ont étudié et modélisé le mouvement du pendule à 5 liens.

Dans le cours, nous aborderons aussi d'autres logiciels de mathématiques complémentaires tels que Mathematica et Geogebra. Finalement, dans la dernière partie, nous ferons une introduction à la programmation en Python.

2 Calculatrice et arithmétique avec Python

Dans cette section, nous assumons que la version de Python est la version 3.

2.1 Opérations de base en Python

Les opérations de base (addition, soustraction, multiplication, division) sur les nombres se font avec les opérateurs `+`, `-`, `*` et `/`:

```
>>> 13.14 + 1.2
14.34
>>> 14.34 - 1.2
13.14
>>> 6 * 9
54
>>> 54 / 9
6.0
```

La division d'un nombre par zéro retourne une erreur:

```
>>> 53 / 0
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

2.2 Exposant

Le calcul d'une puissance se fait avec la double astérisque `**`:

```
>>> 2 ** 8
256
```

Attention!

En Python, l'opérateur `^` ne calcule pas l'exposant, mais fait plutôt une opération sur la représentation binaire des nombres entiers (ou exclusif bit à bit):

```
>>> 5 ^ 3
6
```

2.3 Racine n-ième

Le calcul d'une puissance permet aussi de calculer la racine n-ième d'un nombre:

```
>>> 256 ** (1 / 8)
2.0
```

2.4 Reste et quotient de la division

Le *reste* de la division d'un nombre entier par un autre se fait avec le symbole `%`. Par exemple, on calcule le reste de la division du nombre 94 par 10:

```
>>> 94 % 10
4
```

L'opération `a//b` lorsque `a` et `b` sont des nombres entiers retourne le *quotient* de la divsion `a` par `b`:

```
>>> 94 // 10
9
```

2.5 Fonctions et constantes mathématiques en Python

Le module `math` de Python contient un certain nombre de fonctions et constantes mathématiques que l'on retrouve sur une calculatrice:

```
acos       atanh      e          factorial  hypot      log10      sin
acosh      ceil       erf        floor      isinf      log1p      sinh
asin       copysign   erfc       fmod       isnan      modf       sqrt
asinh      cos        exp        frexp      ldexp      pi         tan
atan       cosh       expm1      fsum       lgamma     pow        tanh
atan2      degrees    fabs       gamma      log        radians    trunc
```

On trouvera leur documentation sur https://docs.python.org/library/math.html

Pour importer `quelquechose` de ce module, il faut d'abord l'importer avec la syntaxe `from math import quelquechose`. Par exemple, pour importer la fonction `factorial`, on procède de la façon suivante:

```
>>> from math import factorial
>>> factorial(4)
24
```

Alternativement, on peut aussi importer le module `math` et procéder ainsi:

```
>>> import math
>>> math.factorial(4)
24
```

De même pour les fonctions trigonométriques, on les importe de la façon suivante:

```
>>> from math import sin, cos, tan, pi
```

Pour importer toutes les fonctions d'un `module`, il suffit d'écire `from module import *`. Par exemple, pour importer toutes les fonctions du module `math`, on écrit:

```
>>> from math import *
```

On vérifie que le sinus d'un angle de 90 degrés est bien égal à 1:

```
>>> sin(90)
0.8939966636005579
```

Oups, l'argument doit être écrit est en radians (90 degrés est égal à *pi/2* radians) et on obtient bien 1:

```
>>> sin(pi/2)
1.0
```

La constante *pi* du module `math` retourne une valeur approchée à une quinzaine de décimales:

```
>>> pi
3.141592653589793
```

Les fonctions `degrees` et `radians` permettent de passer d'une unité d'angle à l'autre:

```
>>> from math import degrees, radians
>>> degrees(pi)
180.0
```

```
>>> radians(180)
3.141592653589793
```

Extraction de la racine carrée avec la fonction `sqrt`:

```
>>> from math import sqrt
>>> sqrt(100)
10.0
```

Calcul des racines du polynôme `3x**2 + 7x + 2`:

```
>>> from math import sqrt
>>> (- 7 + sqrt(7**2 - 4 * 3 * 2) ) / (2 * 3)
-0.3333333333333333
>>> (- 7 - sqrt(7**2 - 4 * 3 * 2) ) / (2 * 3)
-2.0
```

2.6 Accéder à la documentation d'une fonction

En Python, pour obtenir de l'information sur une `fonction`, on peut écrire `help(fonction)`. Par exemple, si on ne sait pas à quoi peut bien servir la fonction `hypot`:

```
>>> from math import hypot
>>> help(hypot)
Help on built-in function hypot in module math:
hypot(...)
    hypot(x, y)
    Return the Euclidean distance, sqrt(x*x + y*y).
```

En IPython, on peut consulter la documentation d'une fonction en ajoutant un point d'interrogation avant ou après le nom de la fonction. Cela fonctionne aussi dans l'interface Jupiter, ce qui ouvre une fenêtre au bas de la page:

```
>>> ?hypot
Docstring:
hypot(x, y)
Return the Euclidean distance, sqrt(x*x + y*y).
Type:      builtin_function_or_method

>>> hypot?
Docstring:
hypot(x, y)
Return the Euclidean distance, sqrt(x*x + y*y).
Type:      builtin_function_or_method
```

2.7 Parenthèses et priorité des opérations

Les parenthèses permettent d'indiquer dans quelle ordre faire les opérations dans un calcul:

```
>>> 3 * (5 + 2)        # l'addition est calculée en premier
21
```

```
>>> (3 * 5) + 2        # la multiplication est calculée en premier
17
```

Sans les parenthèses, l'expression est évaluée selon l'ordre de priorité des opérations. En particulier, le comportement par défaut est que la multiplication est évaluée avant l'addition:

```
>>> 3 * 5 + 2          # la multiplication est calculée en premier
17
```

En général, les expressions non parenthésées utilisant les opérations de base sont évaluées en tenant compte de l'ordre décrit dans la table ci-bas.

Ordre de priorité des opérations de base (de la plus grande à la plus petite)

Opération	Description
`**`	Élévation à la puissance
`~ + -`	Complément, le plus et le moins unaire
`* / % //`	Multiplication, division, modulo et la division entière
`+ -`	Addition et soustraction

2.8 Variables et affectation

Supposons que l'on veut évaluer le polynôme `3*x**4 + 7*x**3 - 3*x**2 + x - 5` lorsque `x=1234567`. On peut procéder de la façon suivante:

```
>>> 3 * 1234567**4 + 7 * 1234567**3 - 3 * 123467**2 + 1234567 - 5
6969164759371928046905499
```

Cela nous oblige à écrire quatre fois le nombre `1234567` et on peut éviter cela au moyen d'une variable.

Une variable permet de mémoriser un nombre pour le réutiliser plus tard. Par exemple, on peut mémoriser le nombre `1234567` dans la variable `x`:

```
>>> x = 1234567
```

Le symbole = ne doit pas être vu comme une équation à résoudre, mais plutôt comme une *affectation* de la valeur `1234567` dans la variable `x`. On peut demander la valeur de `x`:

```
>>> x
1234567
```

Cela nous permet de faire des calculs avec `x`:

```
>>> x + 1
1234568
```

Finalement, on peut utiliser la variable `x` pour évaluer le polynôme au point `x=1234567`:

```
>>> 3*x**4 + 7*x**3 - 3*x**2 + x - 5
6969164759367401312173299
```

C'est curieux. On remarque que le résultat n'est pas le même que celui que l'on avait calculé plus haut. Pourquoi? En effet, on s'était trompé en écrivant `123467` plutôt que `1234567`. C'est aussi l'autre avantage d'utiliser une variable: ça permet d'éviter de se tromper lorsqu'on doit utiliser la même valeur plusieurs fois dans un calcul.

Ensuite, on peut changer la valeur de la variable `x` et évaluer le même polynôme lorsque `x` prend une autre valeur:

```
>>> x = 10
>>> 3*x**4 + 7*x**3 - 3*x**2 + x - 5
36705
```

2.9 Opérateurs de comparaison et d'égalités

Comme on l'a vu dans une section précédente, l'opérateur `=` est utilisé pour l'affectation de variable. Pour tester l'égalité de deux expressions, on utilise alors le l'opérateur `==` s'écrivant avec deux signes d'égalité:

```
>>> 5 * 9 == 40 + 5
True
```

La valeur retournée est un booléen: `True` pour vrai et `False` pour faux. Si l'égalité n'est pas vérifiée, alors c'est la valeur `False` qui est retournée:

```
>>> 5 * 9 == 40 + 6
False
```

Il existe d'autres opérateurs de comparaison dont la description se trouve dans la table ci-bas.

Opérateurs de comparaison et d'égalité

Opérateur	Description
<	strictement inférieur
>	strictement supérieur
<=	inférieur ou égal
>=	supérieur ou égal
==	égal
!=	différent

Par exemple:

```
>>> 5 * 9 < 1000
True
>>> 1 + 2 + 3 + 4 + 5 >= 15
True
>>> 2016 != 2016
False
```

3 Calculatrice et arithmétique avec SymPy

SymPy est une librairie Python de mathématiques symboliques et un outil de calcul formel. SymPy offre un ensemble riche de fonctions documentées qui facilite la modélisation de problèmes mathématiques (arithmétique, calcul symbolique, résolution d'équations, recherche de racines, dessin de fonctions, limites et séries, calcul différentiel et intégral, algèbre linéaire). La librairie offre aussi des fonctionalités de mathématiques plus avancées (géométrie différentielle et algébrique, intégration numérique, théorie des catégories) et pour la physique (optique, mécanique classique, mécanique et informatique quantique).

Le projet contient deux applications Web, SymPy Gamma et SymPy Live, permettant aux étudiants, enseignants et chercheurs d'effectuer des expériences mathématiques en ligne. SymPy Gamma est une interface d'apprentissage pour le calcul et la visualisation basée sur des exemples : algébrique, géométrique, trigonométrique et combinatoire. SymPy Live offre un terminal en ligne avec des fonctionnalités telle que l'écriture des équations en LaTeX, la manipulation symbolique avec le langage Python.

Pour plus d'informations sur SymPy:

- Tutoriel sur SymPy en français: http://docs.sympy.org/dev-py3k/tutorial/tutorial.fr.html
- Aperçu de fonctionalités de SymPy (anglais): http://www.sympy.org/en/features.html
- Documentation complète sur SymPy (anglais): http://docs.sympy.org/

Les exemples dans les sections qui suivent sont parfois nouveaux sinon inspirés de la documentation en ligne de SymPy.

3.1 Nombres rationels

Les nombres rationnels en SymPy doivent être construits à l'aide de la fonction `Rational`:

```
>>> from sympy import Rational
>>> Rational(53, 9)
53/9
>>> Rational(1,4) + Rational(1,3)
7/12
```

Une autre façon plus courte de construire un nombre rationnel en SymPy est d'utiliser la fonction raccourcie `S` qui traduit des types de Python comme une chaîne de caractères ou un entier en des nombres entiers ou rationnels de SymPy:

```
>>> from sympy import S
>>> S("5/2")    # traduction d'une chaîne de caractères en un rationel
5/2
```

La division d'un nombre entier de sympy par un nombre entier retourne un nombre rationnel plutôt qu'un nombre à virgule flottante:

```
>>> S(5)/2      # S(5) transforme 5 (type int) en un nombre entier de sympy
5/2
```

3.2 Nombres complexes

En SymPy, les nombres complexes s'écrivent avec la lettre majuscule `I` pour symboliser la partie imaginaire. Il faut d'abord l'importer:

```
>>> from sympy import I
```

Par exemple, le nombre complexe *2+5i* s'écrit `2+5*I`:

```
>>> 2 + 5*I
2 + 5*I
```

On vérifie que le carré du nombre imaginaire `I` retourne bien `-1`:

```
>>> I ** 2
-1
```

Les opérations de base sont définies sur les nombres complexes comme pour les nombres entiers et les nombres décimaux:

```
>>> (13 + 34*I) + (3 + 4*I)
16 + 38*I
```

Note

Le résultat n'est pas toujours simplifié. Pour ce faire, on peut utiliser la fonction `simplify` de SymPy:

```
>>> (13 + 34*I) / (3 + 4*I)
(13 + 34*I)/(3 + 4*I)
>>> from sympy import simplify
>>> simplify( (13 + 34*I) / (3 + 4*I) )
7 + 2*I
```

La division de nombre complexes entiers de partie imaginaire nulle retourne bien un nombre rationnel:

```
>>> (3+0*I) / (2+0*I)
3/2
```

Pour obtenir les parties réelles et imaginaires d'un nombre complexe, on peut utiliser les fonctions `re` et `im` de SymPy:

```
>>> from sympy import re,im
>>> re(3 + 7*I)
3
>>> im(3 + 7*I)
7
```

Pour obtenir le module et l'argument d'un nombre complexe, on utilse les fonctions `arg` et `abs` de SymPy. Notez qu'il n'est pas nécessaire d'importer `abs`, car cette fonction qui retourne la valeur absolue d'un nombre réel est déjà dans Python et fonctionne pour les nombres complexes:

```
>>> from sympy import arg
>>> arg(3 + 7*I)
atan(7/3)
>>> abs(3 + 7*I)
sqrt(58)
```

Quand c'est possible, SymPy procède à des simplifications:

```
>>> arg(1 + I)
pi/4
>>> abs(3 + 4*I)
5
```

Le conjugué d'un nombre complexe s'obtient avec la fonction `conjugate`:

```
>>> from sympy import conjugate
>>> conjugate(3 + 7*I)
3 - 7*I
```

On peut aussi obtenir le conjugué d'un nombre complexe en utilisant la méthode `conjugate` de la façon suivante (une *méthode* est une fonction définie dans la classe d'un objet, ici dans la classe des nombres complexes):

```
>>> a = 3 + 7*I
>>> a.conjugate()
3 - 7*I
```

Utiliser la deuxième façon (méthode `conjugate`) plutôt que la première (fonction globale `conjugate`) permet d'éviter d'importer la fonction et aussi permet d'utiliser la touche `TAB` (dans IPython ou Jupyter) pour choisir ou compléter l'écriture du nom de la méthode.

3.3 Calculer une valeur numérique

Calculer la valeur numérique d'un `nombre` se fait avec la méthode `evalf` ou de façon équivalente n avec la syntaxe `nombre.n(prec)` où `prec` est le nombre de chiffres à afficher:

```
>>> from sympy import pi
>>> pi.evalf(60)
3.14159265358979323846264338327950288419716939937510
>>> pi.n(60)
3.14159265358979323846264338327950288419716939937510
```

Le nombre de chiffres inclut les chiffres à gauche et à droite de la virgule:

```
>>> from sympy import exp, pi, sqrt
>>> exp(pi * sqrt(163)).evalf(50)
262537412640768743.99999999999925007259719818568888
```

3.4 Factoriser un nombre entier

Pour factoriser un nombre entier, il suffit d'utiliser la fonction `factorint`. La valeur retournée est un dictionnaire qui associe à chaque diviseur une valeur qui représente la multiplicité du diviseur:

```
>>> from sympy import factorint
>>> factorint(240)
{2: 4, 3: 1, 5: 1}
```

Il est possible d'afficher un résultat plus visuel de la factorisation au moyen de la fonction `pprint` et de l'option `visual=True`:

```
>>> from sympy import pprint
>>> pprint(factorint(240, visual=True))
 4  1  1
2 ·3 ·5
```

3.5 Accéder à la documentation et au code source d'une fonction

Comme on l'a déjà vu, pour obtenir de l'aide sur une fonction `f`, il suffit d'écrire `?f` ou `f?`. Par exemple:

```
>>> from sympy import Rational
>>> Rational?
```

Comme SymPy est un logiciel libre, on peut aussi accéder au **code source** en ajoutant un deuxième point d'interrogation:

```
>>> Rational??
```

4 Calcul symbolique

En Python, la fonction `sqrt` retourne un résultat approximé sur les nombres flottants:

```
>>> from math import sqrt
>>> sqrt(3)
1.73205080757
```

La racine de 3 ne peut pas être écrite exactement avec un nombre fini de décimales, car c'est un nombre irrationel. La façon la plus exacte d'exprimer la racine (positive) du nombre trois est de l'écrire tel quel. En SymPy, la racine carrée de 3 est exprimée *symboliquement*:

```
>>> from sympy import sqrt
>>> sqrt(3)
sqrt(3)
```

Quand c'est possible, SymPy procède à des simplifications:

```
>>> sqrt(4)
2
>>> sqrt(8)
2*sqrt(2)
```

4.1 Variable symbolique

La fonction `Symbol` de SymPy permet de créer une variable symbolique:

```
>>> from sympy import Symbol
>>> a = Symbol("a")     # le symbole a est stocké dans la variable a
>>> a
a
```

Cette variable peut être additionnée, soustraite, multipliée et divisée:

```
>>> a + a + a + a * a + 1/a - a
a**2 + 2*a + 1/a
```

Pour plus de commodité, on peut importer les variables les plus souvent utilisées du sous-module `abc`:

```
>>> from sympy.abc import a, epsilon
>>> a + a * a + epsilon
a**2 + a + epsilon
```

4.2 Définir les variables symboliques x_1, x_2, ..., x_n

En SymPy, on peut créer plusieurs variables *x_i* indicées pour *i=a,...,b-1* à l'aide de la fonction `symbols("xa:b")` où `a` et `b` sont remplacés par des nombres entiers:

```
>>> from sympy import symbols
>>> x4,x5,x6,x7,x8 = symbols("x4:9")
>>> x4 + x5 + x6 + x7 + x8
x4 + x5 + x6 + x7 + x8
```

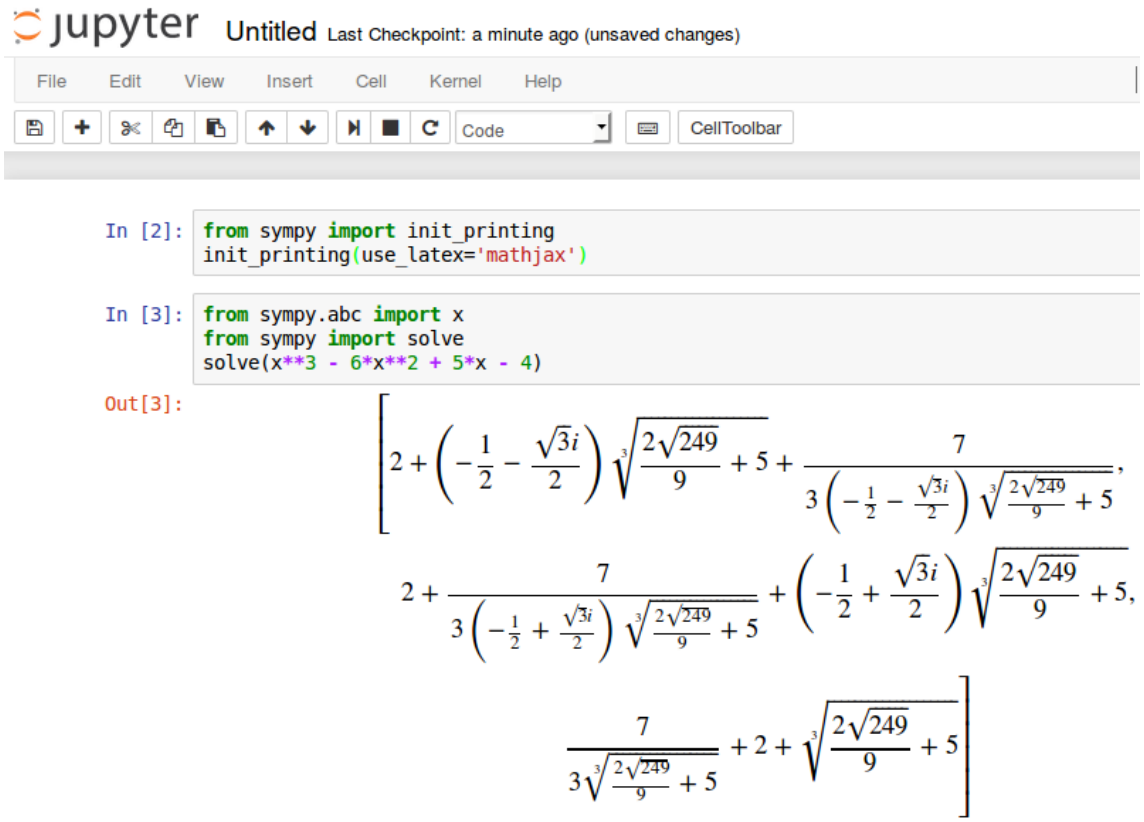
4.3 Affichage automatique des résultats en LaTeX

Pour que les résultats soient affichés en LaTeX automatiquement, il suffit d'utiliser la commande suivante une seule fois pour le fichier:

```
>>> from sympy import init_printing
>>> init_printing(pretty_print=True)
```

Parfois, dans le notebook Jupyter, la commande suivante qui utilise la librairie MathJax donne de meilleurs résultats (l'affichage en LaTeX est plus rapide):

```
>>> init_printing(use_latex='mathjax')
```



$$\left[2 + \left(-\frac{1}{2} - \frac{\sqrt{3}i}{2}\right)\sqrt[3]{\frac{2\sqrt{249}}{9} + 5} + \frac{7}{3\left(-\frac{1}{2} - \frac{\sqrt{3}i}{2}\right)\sqrt[3]{\frac{2\sqrt{249}}{9} + 5}},\right.$$

$$2 + \frac{7}{3\left(-\frac{1}{2} + \frac{\sqrt{3}i}{2}\right)\sqrt[3]{\frac{2\sqrt{249}}{9} + 5}} + \left(-\frac{1}{2} + \frac{\sqrt{3}i}{2}\right)\sqrt[3]{\frac{2\sqrt{249}}{9} + 5},$$

$$\left.\frac{7}{3\sqrt[3]{\frac{2\sqrt{249}}{9} + 5}} + 2 + \sqrt[3]{\frac{2\sqrt{249}}{9} + 5}\right]$$

Dans ces notes, on utilisera l'option `init_printing(pretty_print=True, use_unicode=False)` lorsque cela aide la lecture des formules. Pour un exemple de la section précédente, on obtient:

```
>>> from sympy.abc import a, epsilon
>>> a + a * a + epsilon
 2
a  + a + ε
```

4.4 Expressions symboliques

On peut faire des calculs impliquant plus d'une variable:

```
>>> from sympy.abc import a,b
>>> (a + b)**2
(a + b)**2
```

ou impliquant les fonctions de SymPy:

```
>>> from sympy import sin,cos
>>> sin(a)**2 + cos(a)**2 + b
b + sin(a)**2 + cos(a)**2
```

Les expressions symboliques peuvent combiner des rationels, des fonctions et des constantes de toutes sortes:

```
>>> from sympy import Rational,pi,exp,I
>>> from sympy.abc import x,y
```

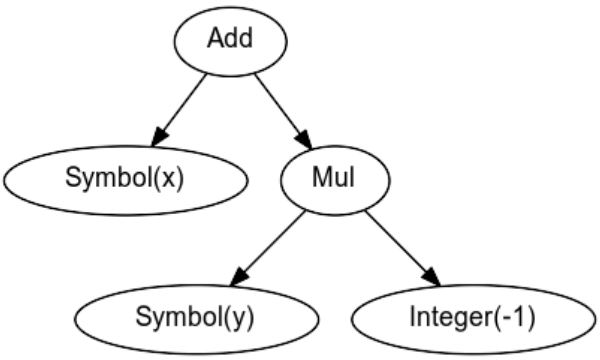
```
>>> Rational(3,2)*pi + exp(I*x) / (x**2 + y)
3*pi/2 + exp(I*x)/(x**2 + y)
```

4.5 Représentation interne

Pour voir comment une expression symbolique est représentée dans la machine, on peut utiliser la fonction `srepr`:

```
>>> from sympy import srepr
>>> expr = x - y
>>> srepr(x - y)
"Add(Symbol('x'), Mul(Integer(-1), Symbol('y')))"
```

L'expression symbolique est représentée par un arbre d'opérations.



Pour information, l'image a été créée avec Graphviz avec le resultat de la fonction `dotprint`:

```
>>> from sympy.printing.dot import dotprint
>>> s = dotprint(expr)
```

4.6 Substitutions

Pour substituer certaines variables dans une `expression`, on utilise la méthode `subs` qui s'écrit **après** l'expressions sous la forme `expressions.subs(<INPUT>)`. Par exemple:

```
>>> from sympy.abc import a,b,c
>>> expression = a + 2*b + 3*c
>>> expression.subs(a,9)
2*b + 3*c + 9
```

Pour faire plus d'une substitutions, on peut les indiquer dans un dictionaire (`{}`) comme ci-bas:

```
>>> expression.subs({a:9, b:4})
3*c + 17
>>> expression.subs({a:9, b:4, c:100})
317
```

On peut aussi substituer une variable symbolique par une expression symbolique:

```
>>> from sympy import log
>>> from sympy.abc import x,y,z
```

```
>>> expression.subs({a:x**2, b:log(y), c:z})
x**2 + 3*z + 2*log(y)
```

4.7 Constantes symboliques

Contrairement au module `math` de Python où le nombre pi est représenté par une approximation décimale, dans SymPy, le nombre pi est représenté symboliquement. C'est une **constante symbolique**:

```
>>> from sympy import pi
>>> pi
pi
```

Cela permet de faire des calculs exacts. Par exemple, le sinus d'un angle de `pi/3` est égal à la racine de trois sur deux:

```
>>> from sympy import sin, pi
>>> sin(pi/3)
  ___
\/ 3
-----
  2
```

La fonction inverse du sinus aussi appelée arc sinus et représentée par la fonction `asin` dans SymPy peut retourner des expressions symboliques impliquant des constantes symboliques telles que le nombre pi:

```
>>> from sympy import asin, Rational
>>> asin(1)
pi
--
2
>>> asin(Rational(1,2))
pi
--
6
```

SymPy sait que les fonctions sinus et arc sinus sont inverses une de l'autre:

```
>>> from sympy.abc import x
>>> sin(asin(x))
x
```

Attention!

La fonction `sin` du module `math` de Python ne peut pas être appelée sur des expressions symboliques, car elle assume que l'argument est un nombre réel (type float):

```
>>> from sympy.abc import
>>> from math import sin
```

```
>>> sin(x)
Traceback (most recent call last):
...
TypeError: can't convert expression to float
```

4.8 Simplifier une expression

Les expressions ne sont pas toujours simplifiées:

```
>>> from sympy import sin,cos
>>> from sympy.abc import a
>>> r = sin(a)**2 + cos(a)**2
>>> r
sin(a)**2 + cos(a)**2
```

Pour simplifier une expression, on utilise la commande `simplify`:

```
>>> from sympy import simplify
>>> simplify(r)
1
```

Voici un autre exemple:

```
>>> simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
x - 1
```

La fonction `simplify` performe une série de simplifications dans un ordre bien choisi. Les simplifications spécifiques incluent `besselsimp`, `combsimp`, `exptrigsimp`, `hypersimp`, `nsimplify`, `powsimp`, `radsimp`, `ratsimp`, `ratsimpmodprime`, `signsimp`, `simplify`, `simplify_logic`, `trigsimp` et d'autres encore. Il suffit de consulter le code `simplify??` pour voir ce qui se passe et dans quel ordre.

Lorsque l'on sait exactement ce qu'on veut faire sur l'expression symbolique (factoriser, mettre sur un dominateur commun, etc.), on peut utiliser directement la bonne fonction. Cela peut retourner un résultat plus rapidement. Les plus importantes fonctions de modification d'expressions symboliques sont décrites dans les sections qui suivent. On trouvera plus d'informations sur les façons de simplifier une expression dans le tutoriel de SymPy: http://docs.sympy.org/latest/tutorial/simplification.html

4.9 Développer une expression

Pour développer une expression, on utilise la fonction `expand`:

```
>>> from sympy import expand
>>> from sympy.abc import a,b
>>> (a + b)**2
(a + b)**2
>>> expand((a + b)**2)
a**2 + 2*a*b + b**2
```

Cela peut mener à des simplifications:

```
>>> (a + b)**2 - (a - b)**2
-(a - b)**2 + (a + b)**2
>>> expand(_)
4*a*b
```

Note

En IPython et Jupyter, la barre de soulignement (_) est une variable qui contient le dernier résultat calculé. Aussi, la double barre de soulignement (__) est une variable qui contient l'avant-dernier résultat calculé. Finalement, la triple barre de soulignement (___) est une variable qui contient l'avant-avant-dernier résultat calculé. La quadruple barre de soulignement ne contient rien.

4.10 Annuler les facteurs communs d'une fraction

Pour annuler les facteurs communs dans une fonction rationnelle, on utilise `cancel`:

```
>>> expr = (x**2 + x*y) / x
>>> expr
(x**2 + x*y)/x
>>> from sympy import cancel
>>> cancel(expr)
x + y
```

4.11 Factoriser un polynôme

La fonction `factor` de SymPy permet de factoriser un polynôme en un produit de facteurs irréductibles sur l'anneau des nombres rationnels:

```
>>> from sympy import factor
>>> factor(x**3 - x**2 + x - 1)
(x - 1)*(x**2 + 1)
```

Pour factoriser le polynôme sur les nombres de Gauss (nombres complexes à parties imaginaire et réelle entières), on ajoute l'option `gaussian=True`:

```
>>> factor(x**3 - x**2 + x - 1, gaussian=True)
(x - 1)*(x - I)*(x + I)
```

Pour faire la factorisation sur une extension algébrique des nombres rationnels, il suffit de spéficifier un ou des nombres algébriques qui engendrent l'extension:

```
>>> factor(x**2 - 5)
x**2 - 5
>>> factor(x**2 - 5, extension=sqrt(5))
(x - sqrt(5))*(x + sqrt(5))
```

Consulter la documentation `factor?` pour obtenir de l'aide sur la factorisation de polynômes sur d'autres domaines ou sur des extensions de corps.

4.12 Rassembler les termes d'une expression

La fonction `collect` rassemble les puissances communes d'un terme dans une expression. Par exemple:

```
>>> expr = x*z + x**2 + x + x*y + x**2 * w + 5 - x**3
>>> expr
w*x**2 - x**3 + x**2 + x*y + x*z + x + 5
```

On rassemble les termes selon les puissances de x:

```
>>> from sympy import collect
>>> collect(expr, x)
-x**3 + x**2*(w + 1) + x*(y + z + 1) + 5
```

4.13 Réduire au même dénominateur

Une somme de fonctions rationnelles reste sous forme de somme:

```
>>> from sympy.abc import x,y,z
>>> 1/(x+1) + 1/y + 1/z
  1     1   1
----- + - + -
x + 1   z   y
```

Pour la mettre au même dénominateur, on utilise `ratsimp`:

```
>>> from sympy import ratsimp
>>> ratsimp(1/(x+1) + 1/y + 1/z)
x*y + x*z + y*z + y + z
-----------------------
      x*y*z + y*z
```

Alternativement, on peut aussi utiliser la fonction `together`. À la différence de `ratsimp` la fonction `together` préserve le plus possible les termes sous la forme initiale:

```
>>> from sympy import together
>>> together(1/(x+1) + 1/y + 1/z)
y*z + y*(x + 1) + z*(x + 1)
---------------------------
        y*z*(x + 1)
```

4.14 Décomposition en fractions partielles

Soit un fraction rationnelle:

```
>>> expr = (3*x**2  + 52*x - 265) / ((x - 7)*(x - 1)*(x + 34))
>>> expr
      2
   3*x  + 52*x - 265
------------------------
(x - 7)*(x - 1)*(x + 34)
```

On peut la décomposer en somme de fractions rationnelles à l'aide de la fonction `apart` de SymPy:

```
>>> from sympy import apart
>>> apart(expr)
  1        1       1
------ + ----- + -----
x + 34   x - 1   x - 7
```

4.15 Rationalisation du dénominateur d'une expression

Pour rationaliser le dénominateur d'un expression, on utilise la fonction `radsimp` de SymPy:

```
>>> A = 1 / (1+sqrt(5))
>>> A
    1
---------
     ___
1 + \/ 5
>>> from sympy import radsimp
>>> radsimp(A)
       ___
-1 + \/ 5
----------
    4
```

5 Résolution d'équations

5.1 Définir une équation

La fonction `Eq` permet de définir une équation:

```
>>> from sympy import Eq
>>> from sympy.abc import x,y
>>> Eq(x, 3)
x == 3
>>> Eq(x + y, 3)
x + y == 3
```

5.2 Résoudre une équation

La fonction `solve` permet de résoudre une équation:

```
>>> from sympy import solve
>>> solve(Eq(x, 3), x)
[3]
>>>  solve(Eq(x + y, 3), x)
[-y + 3]
```

Dans le premier cas, indiquer la variable `x` n'est pas nécessaire:

```
>>> solve(Eq(x, 3))
[3]
```

Cela fonctionne aussi s'il y a plus d'une solution:

```
>>> solve(Eq(x**2, 3))
[-sqrt(3), sqrt(3)]
```

5.3 Résoudre un système d'équations

La fonction `solve` permet aussi de résoudre un système d'équations. Par exemple pour trouver deux variables `x` et `y` telle que leur somme vaut 47 et leur différence vaut 33:

```
>>> eq1 = Eq(x + y, 47)
>>> eq2 = Eq(x - y, 33)
>>> [eq1, eq2]
[x + y == 47, x - y == 33]
>>> solve([eq1, eq2])
{x: 40, y: 7}
```

Attention!

Pour résoudre un système d'équations, il faut absoluement mettre les équations dans une liste `[eq1, eq2]` entre crochets `[]`. Autrement, on obtient une erreur:

```
In [279]: solve(eq1, eq2)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of
x - y == 33
```

Cela fonctionne aussi pour des systèmes d'équations non linéaires. Par exemple pour trouver deux nombres dont la somme est 47 et dont le produit est 510:

```
>>> eq1 = Eq(x + y, 47)
>>> eq2 = Eq(x * y, 510)
>>> [eq1, eq2]
[x + y == 47, x*y == 510]
>>> solve([eq1, eq2])
[{x: 17, y: 30}, {x: 30, y: 17}]
```

5.4 Syntaxe abrégée

Souvent on désire résoudre des équations dont l'un des termes est zéro: `solve(Eq(expression,0))`. Pour ce cas, il est équivalent d'écrire `solve(expression)` et la fonction `solve` trouvera les valeurs des variables avec lesquelles expression est évaluée à zéro. Les exemples ci-haut s'écrivent de la façon suivante avec cette syntaxe abrégée:

```
>>> solve(x - 3)
[3]
>>> solve(x**2 + x - 6)
```

```
[-3, 2]
>>> solve([x + y - 47, x - y - 33])
{x: 40, y: 7}
>>> solve([x + y - 47, x * y - 510])
[{x: 17, y: 30}, {x: 30, y: 17}]
```

Un exemple sur un polynôme de degré 3:

```
>>> solve(x**3 + 2*x**2 - 1, x)
                ___     ___
      1   \/ 5     \/ 5    1
[-1, - - + -----, - ----- - -]
      2     2       2     2
```

La syntaxe abrégée peut aussi être utilisée pour résoudre un système d'équations. Dans l'exemple qui suit, on calcule les points d'intersection d'une ellipse et d'une droite:

```
>>> solve( [x**2 + 4*y**2 -2, -10*x + 2*y -15], [x, y])

                 ____                ____                ____                ____
    150   \/ 23 *I   15   5*\/ 23 *I      150   \/ 23 *I   15   5*\/ 23 *I
[(- --- - --------, --- - ----------), (- --- + --------, --- + ----------)]
    101     101     202      101          101     101     202      101
```

5.5 Trouver les racines d'une fonction

La fonction `roots` permet de calculer les racines d'une fonction polynomiale univariée

```
>>> from sympy import roots
>>> roots(x - 7)
{7: 1}
>>> roots(x**6)
{0: 6}
```

Le résultat est un dictionnaire (`{}`) qui associe à chaque racine sa multiplicité. La fonction `roots` trouve aussi les racines complexes:

```
>>> roots(x**5 - 7*x**4 + 2*x**3 - 14*x**2 + x - 7, x)
{7: 1, -I: 2, I: 2}
```

Les coefficients des polynômes peuvent être des variables symboliques:

```
>>> from sympy.abc import a,b,c
>>> roots(a*x + b, x)
{-b/a: 1}
```

Mais à ce moment-là, il faut absoluement spécifier par rapport à quelle variable on cherche les racines. Autrement, on obtient une erreur:

```
>>> roots(a*x + b)
Traceback (most recent call last)
```

```
...
PolynomialError: multivariate polynomials are not supported
```

La fonction `roots` trouve les formules qui expriment les racines d'un polynôme quadratique:

```
>>> roots(a*x**2 + b*x + c, x)
                ______________                ______________
              /              2              /              2
    b    \/  -4*a*c + b           b    \/  -4*a*c + b
{- --- - ----------------: 1, - --- + ----------------: 1}
   2*a          2*a              2*a          2*a
```

On trouvera d'autres exemples (résolution d'équations différentielles) et des explications plus détaillées dans la section *Solver* du tutoriel de SymPy: http://docs.sympy.org/latest/tutorial/solvers.html

6 Tracer une fonction

La librairie Sympy utilise matplotlib, une autre librairie de Python, pour faire des dessins. Pour activer l'affichage des graphiques dans Jupyter, on écrit d'abord ceci dans une cellule:

```
>>> %matplotlib inline
```

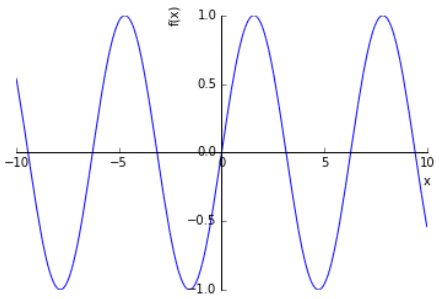
6.1 Tracer une fonction R -> R

On importe la fonction `plot` qui permet de dessiner des fonctions:

```
>>> from sympy import plot
```

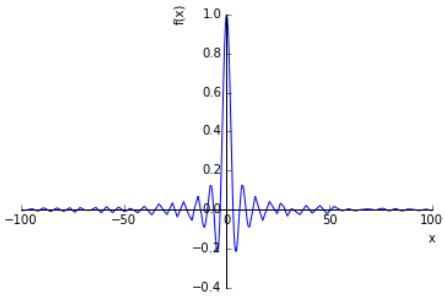
Un premier exemple. Par défaut, l'intervalle pour les `x` est [-10,10]:

```
>>> from sympy import sin
>>> from sympy.abc import x
>>> plot(sin(x))
```



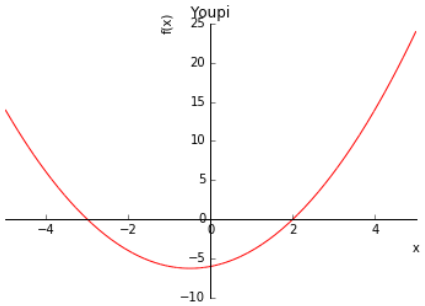
Un deuxième exemple sur l'intervalle [-100, 100]:

```
>>> plot(sin(x)/x, (x,-100,100))
```



On trace une parabole de couleur rouge dans l'intervale [-5, 5] avec un titre:

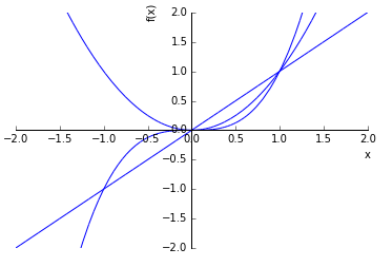
```
>>> plot(x**2+x-6, (x,-5,5), line_color='red', title='Youpi')
```



6.2 Tracer plusieurs fonctions R -> R

On trace plusieurs fonctions sur le même intervalle de la façon suivante. Dans cet exemple, on a aussi spécifier une limite inférieure et supérieure pour l'axe des y:

```
>>> plot(x, x**2, x**3, (x, -2, 2), ylim=(-2,2))
```



Pour dessiner les trois fonctions avec des couleurs différentes, il faut créer un dessin à la fois et ensuite les combiner. L'option `show=False` permet d'éviter d'afficher les dessins intermédiaires:

```
>>> p1 = plot(x,    (x, -1, 1), show=False, line_color='b')
>>> p2 = plot(x**2, (x, -1, 1), show=False, line_color='r')
>>> p3 = plot(x**3, (x, -1, 1), show=False, line_color='g')
```

On ajoute à `p1` les graphes `p2` et `p3`:

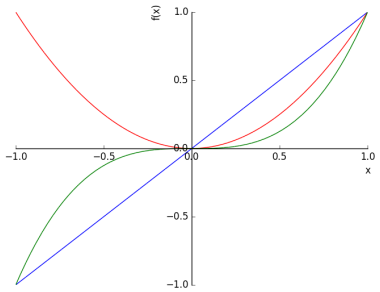
```
>>> p1.extend(p2)
>>> p1.extend(p3)
```

Maintenant `p1` contient les trois graphes:

```
>>> print(p1)
Plot object containing:
[0]: cartesian line: x for x over (-1.0, 1.0)
[1]: cartesian line: x**2 for x over (-1.0, 1.0)
[2]: cartesian line: x**3 for x over (-1.0, 1.0)
```

On affiche le graphe des trois fonctions:

```
>>> p1.show()
```



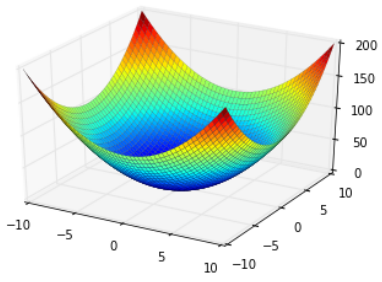
6.3 Tracer une fonction R^2 -> R

On importe la fonction `plot3d` du sous-module `sympy.plotting`:

```
>>> from sympy.plotting import plot3d
```

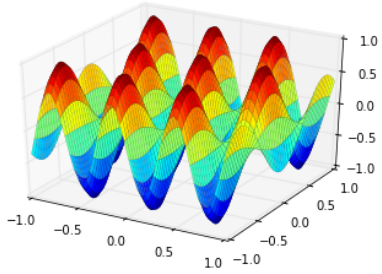
Un premier exemple:

```
>>> plot3d(x**2+y**2)
```



Un deuxième exemple:

```
>>> plot3d(sin(x*10)*cos(y*4), (x, -1, 1), (y, -1, 1))
```



On trouvera d'autres exemples en consultant la documentation de `plot?` et `plot3d?` ou dans la section Plotting du tutoriel de Sympy: http://docs.sympy.org/latest/modules/plotting.html

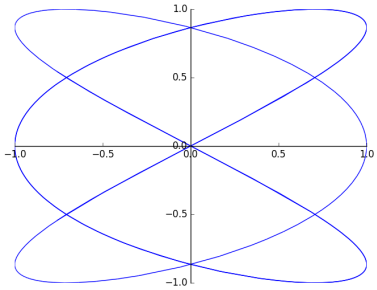
6.4 Dessiner une fonction R -> R^2

Dans cette section et les suivantes, on aura utilisera les fonctions et variables symboliques suivantes:

```
>>> from sympy import sin, cos
>>> from sympy.abc import u, v
```

La fonction `plot_parametric` permet de tracer des fonctions paramétrés R -> R^2. Par exemple, on trace la courbe de Lissajous lorsque a=3 et b=2:

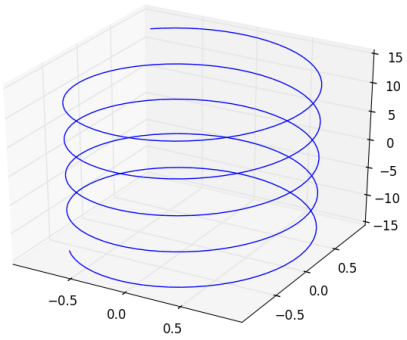
```
>>> from sympy.plotting import plot_parametric
>>> plot_parametric(cos(3*u), sin(2*u), (u, -5, 5))
```



6.5 Dessiner une fonction R -> R^3

La fonction `plot3d_parametric_line` permet de tracer des courbes dans l'espace 3d. Par exemple, on trace une hélice:

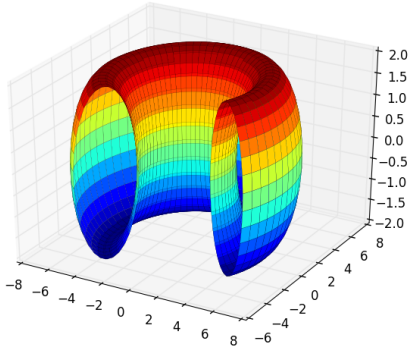
```
>>> from sympy.plotting import plot3d_parametric_line
>>> plot3d_parametric_line(cos(u), sin(u), u, (u, -15, 15))
```



6.6 Dessiner une fonction R^2 -> R^3

La fonction `plot3d_parametric_surface` permet de tracer des surfaces dans R^3. Par exemple, on trace un tore:

```
>>> from sympy.plotting import plot3d_parametric_surface
>>> X = cos(u)*(5+2*cos(v))
>>> Y = sin(u)*(5+2*cos(v))
>>> Z = 2*sin(v)
>>> plot3d_parametric_surface(X, Y, Z, (u, -.5, 4), (v, -5, 5))
```

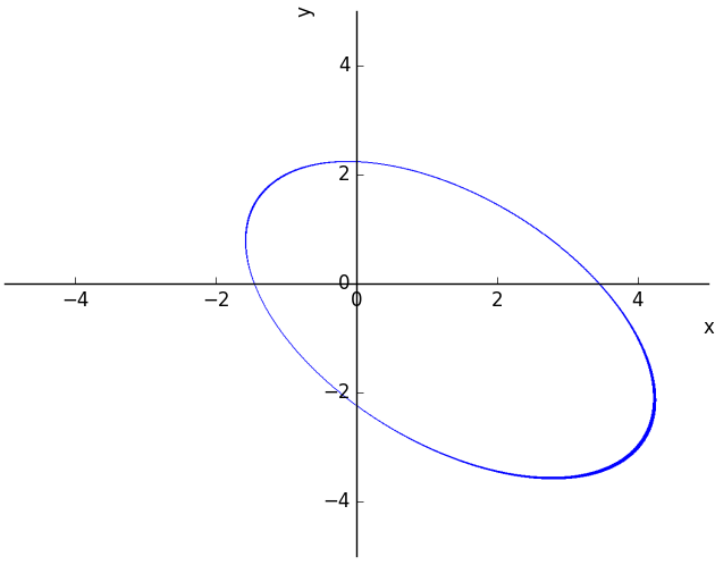


6.7 Dessiner les solutions d'une équation implicite

```
>>> from sympy import plot_implicit, Eq
>>> from sympy.abc import x, y
```

La fonction `plot_implicit` permet de tracer les solutions d'une équation implicite:

```
>>> eq = Eq(x**2+y**2+x*y-2*x, 5)
>>> eq
x**2 + x*y - 2*x + y**2 == 5
>>> plot_implicit(eq)
```



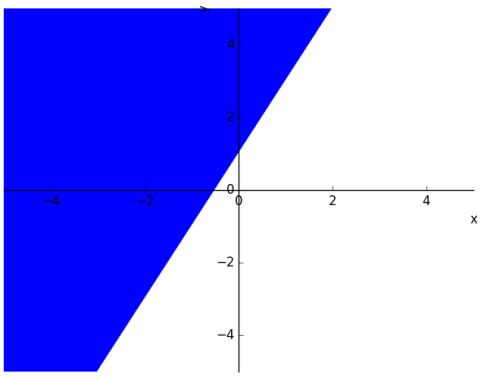
On peut modifier les étendues des variables `x` et `y` de la façon suivante (le dessin n'est pas affiché dans ces notes):

```
>>> plot_implicit(eq, (x,-2,5), (y,-5,3))
```

6.8 Tracer une région de R^2

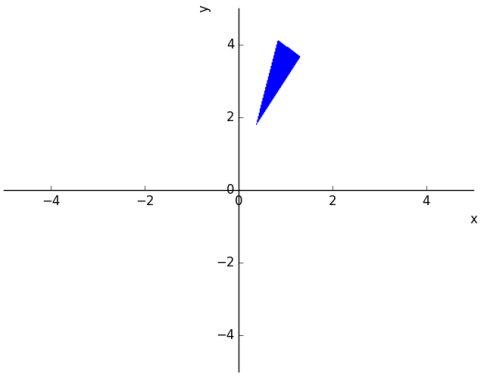
La fonction `plot_implicit` peut aussi servir à dessiner une région de points qui satisfont une inégalité:

```
>>> plot_implicit(y > 2*x+1)
```



Pour tracer la région définie par plusieurs inégalités, on utilise la fonction `And` de sympy:

```
>>> from sympy import And
>>> plot_implicit(And(y>2*x+1, y<5*x, x+y<5))
```



6.9 Dessiner une fonction complexe avec mpmath

mpmath est une librairie Python pour faire des calculs en précision arbitraire sur les nombres flottants. Elle permet aussi de faire des dessins de fonctions complexes.

La façon d'importer la librairie mpmath n'est pas exactement la même selon qu'on utilise une installation normale de SymPy ou qu'on utilise SageMath:

```
>>> from sympy import mpmath    # Sympy (installation normale)
>>> import mpmath               # SageMath
```

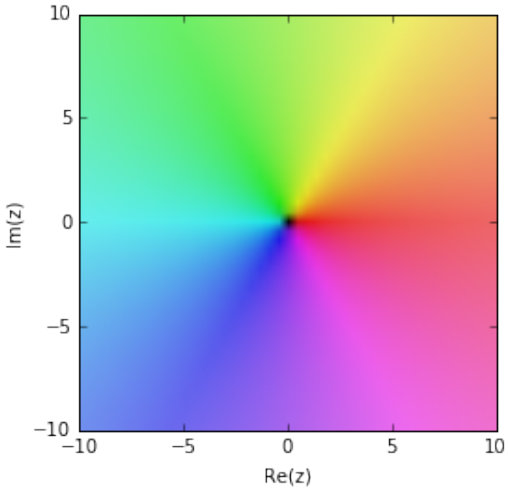
Rappelons que sans la ligne suivante, les dessins ne s'afficheront pas:

```
>>> %matplotlib inline
```

La syntaxe des arguments n'est pas exactement la même que pour la fonction `plot` de SymPy. Il faut définir une fonction Python avec la commande `def` ou encore sur une ligne avec `lambda`. Par exemple, la fonction identité peut s'écrire `lambda z:z` en Python.

On trace la fonction identité pour comprendre la signification de l'image obtenue:

```
>>> mpmath.cplot(lambda z: z, [-10, 10], [-10, 10])
```

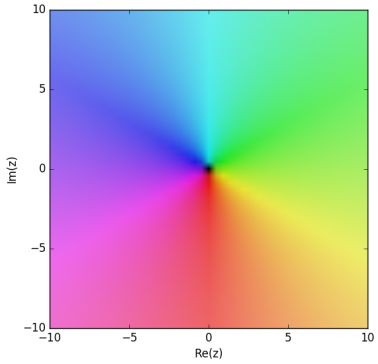


Les couleurs de l'arc en ciel doivent être interprétés comme l'argument d'un nombre complexe (rouge pour un nombre réel positif). Le module du nombre complexe est représenté par la transparence (0=noir opaque, oo=blanc transparent).

De la même façon, on ne peut pas utiliser le `I` de sympy avec mpmath, il faut utiliser les nombres complexes de Python. Le dessin suivant illustre la multiplication par le nombre complexe i, c'est-à-dire une rotation de 90 degrés:

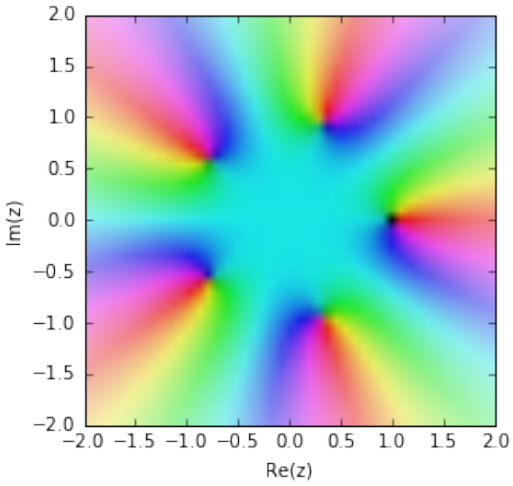
```
>>> I = complex(0,1)          # le nombre complexe I de Python
>>> mpmath.cplot(lambda z: I*z, [-10, 10], [-10, 10])
```

Les pixels en rouges sont envoyés sur la droite réelle positive par la fonction `lambda z: I*z`.



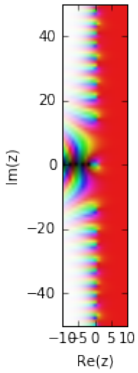
Le dessin suivant permet de voir les cinq racines cinquième de l'unité:

```
>>> mpmath.cplot(lambda z: z**5-1, [-2, 2], [-2, 2])
```



Cela permet aussi d'étudier les zéros de la fonction zeta de Riemann:

```
>>> from mpmath import zeta
>>> mpmath.cplot(zeta, [-10, 10], [-50, 50])
```



mpmath offre aussi sa propre fonction de dessin `mpmath.plot` ainsi qu'une fonction pour dessiner des surfaces en 3d `mpmath.splot`. On trouvera d'autres exemples dans la page suivante de la documentation de Sympy: http://docs.sympy.org/latest/modules/mpmath/plotting.html

7 Calcul différentiel et intégral

Cette section concerne le calcul différentiel et intégral. On trouvera d'autres exemples dans le tutoriel de Sympy sur le même sujet: http://docs.sympy.org/latest/tutorial/calculus.html

7.1 Limites

Pour calculer la limite d'une `expression` lorsqu'une `variable` tend vers une `valeur`, on utilise la fonction `limit` de sympy avec la syntaxe `limit(expression, variable, valeur)`.

```
>>> from sympy import limit, sin, S
>>> from sympy.abc import x
```

Par exemple, pour évaluer la limite lorsque `x` tend vers `0` de l'expression `(sin(x)-x)/x**3`, on écrit:

```
>>> limit((sin(x)-x)/x**3, x, 0)
-1/6
```

La limite de `f(x)=2*x+1` lorsque `x -> 5/2`:

```
>>> limit(2*x+1, x, S(5)/2)   # la fonction S permet de créer un nombre rationel
6
```

Pour calculer la limite **à gauche** en un point, on doit spécifier l'option `dir="-"`:

```
>>> limit(1/x, x, 0, dir="-")
-oo
```

Pour calculer la limite **à droite** en un point, on doit spécifier l'option `dir="+"`:

```
>>> limit(1/x, x, 0, dir="+")
oo
```

Lorsque la direction n'est pas spécifiée, c'est la limite à droite (`dir="+"`) qui est calculée par défaut:

```
>>> limit(1/x, x, 0)
oo
```

En sympy, tout comme dans SageMath, le symbole `oo` représente l'infini. Les deux `o` collés resemblent au symbole de l'infini `8` à l'horizontal. Les opérations d'addition, de soustraction, de multiplication, etc. sont possibles avec l'infini `oo` tant qu'elle soient bien définies. On doit l'importer pour l'utiliser:

```
>>> from sympy import oo
>>> oo
oo
>>> 5 - oo
-oo
>>> oo - oo           # nan signifie "Not A Number"
nan
```

On peut calculer la limite d'une expression lorsque `x` tend vers **plus l'infini**:

```
>>> limit(1/x, x, oo)
0
```

et aussi lorsque `x` tend vers **moins l'infini**:

```
>>> limit(4+x*exp(x), x, -oo)
4
```

Sympy procède à des simplifications lorsque possible:

```
>>> limit((1+1/x)**x, x, oo)
E
```

7.2 Sommes

En Python, il existe une fonction (`sum`) que l'on a pas besoin d'importer et qui permet de calculer la somme des valeurs d'une liste:

```
>>> sum([1,2,3,4,5])
15
```

Cette fonction `sum` permet aussi de calculer une somme impliquant des variables et expressions symboliques de SymPy:

```
>>> from sympy import tan
>>> from sympy.abc import x,z
>>> sum([1,2,3,4,5,x,tan(z)])
x + tan(z) + 15
```

Par contre, `sum` ne permet pas de calculer des sommes infinies ou encore des séries données par un terme général. En SymPy, il existe une autre fonction (`summation`) pour calculer des sommes possiblement infinies d'expressions symboliques:

```
>>> from sympy import summation
```

Pour calculer la somme d'une série dont le terme général est donné par une `expression` qui dépend de `n` pour toutes les valeurs entières de `n` entre `debut` et `fin` (`debut` et `fin` inclus), on utilise la syntaxe `summation(expression (n,debut,fin))`:

```
>>> from sympy.abc import n
>>> summation(n, (n,1,5))
15
```

Le début et la fin de l'intervalle des valeurs de `n` peut être donné par des variables symboliques:

```
>>> from sympy.abc import a,b
>>> summation(n, (n,1,b))
 2
b    b
-- + -
2    2
>>> summation(n, (n,a,b))
   2        2
  a    a   b    b
- -- + - + -- + -
  2    2   2    2
```

Pour faire la somme d'une série pour tous les nombres entiers de 1 à l'infini, on utilise le symbole `oo`:

```
>>> from sympy import oo
>>> summation(1/n**2, (n, 1, oo))
  2
pi
---
 6
```

Si la série est divergente, elle sera évaluée à `oo` ou encore elle restera non évaluée:

```
>>> summation(n, (n,1,oo))
oo
>>> summation((-1)**n, (n,1,oo))
  oo
 ___
 \  `
  \       n
  /   (-1)
 /__,
n = 1
```

Sympy peut aussi calculer une double somme. Il suffit de spéficier l'intervalle des valeurs pour chacune des variables en terminant avec la variable dont la somme est effectuée en dernier:

```
>>> from sympy.abc import m,n
>>> summation(n*m, (n,1,m), (m,1,10))
1705
```

Les doubles sommes fonctionnent aussi avec des intervalles infinis:

```
>>> summation(1/(n*m)**2, (n,1,oo), (m,1,oo))
  4
pi
---
 36
```

7.3 Produit

Comme pour la somme, le calcul d'un produit dont le terme général est donné par une `expression` qui dépend de `n` pour toutes les valeurs entières de `n` entre `debut` et `fin` (`debut` et `fin` inclus), on utilise la syntaxe `product(expression (n,debut,fin))`:

```
>>> from sympy import product
>>> from sympy.abc import n,b
>>> product(n, (n,1,5))
120
>>> product(n, (n,1,b))
b!
```

Voici un autre exemple:

```
>>> product(n*(n+1), (n, 1, b))
RisingFactorial(2, b)*b!
```

7.4 Calcul différentiel

Pour dériver une `fonction` par rapport à une variable `x`, on utilise la fonction `diff` de sympy avec la syntaxe `diff(fonction, x)`:

```
>>> from sympy import diff
```

Faisons quelques importations de fonctions et variables pour la suite:

```
>>> from sympy import sin,cos,tan,atan,pi
>>> from sympy.abc import x,y
```

On calcule la dérivée de `sin(x)`:

```
>>> diff(sin(x), x)
cos(x)
```

Voici quelques autres exemples:

```
>>> diff(cos(x**3), x)
    2    / 3\
-3*x *sin\x /
>>> diff(atan(2*x), x)
   2
--------
   2
4*x  + 1
```

```
>>> diff(1/tan(x), x)
     2
- tan (x) - 1
-------------
      2
   tan (x)
```

Pour calculer la i-ème dérivée d'une fonction, on ajoute autant de variables que nécessaire ou bien on spécifie le nombre de dérivées à faire:

```
>>> diff(sin(x), x, x, x)
-cos(x)
>>> diff(sin(x), x, 3)
-cos(x)
```

Cela fonctionne aussi avec des variables différentes:

```
>>> diff(x**2*y**3, x, y, y)
12*x*y
```

7.5 Calcul intégral

Le calcul d'une intégrale indéfinie se fait avec la fonction `integrate` avec la syntaxe `integrate(f, x)`:

```
>>> from sympy import integrate
```

Par exemple:

```
>>> integrate(1/x, x)
log(x)
```

Le calcul d'une intégrale définie se fait aussi avec la fonction `integrate` avec la syntaxe `integrate(f, (x, a, b))`:

```
>>> integrate(1/x, (x, 1, 57))
log(57)
```

Voici quelques autres exemples:

```
>>> from sympy import exp
>>> integrate(cos(x)*exp(x), x)
 x           x
e *sin(x)   e *cos(x)
--------- + ---------
    2           2
```

```
>>> integrate(x**2, (x,0,1))
1/3
```

L'intégrale d'une fonction rationnelle:

```
>>> integrate((x+1)/(x**2+4*x+4), x)
              1
log(x + 2) + -----
             x + 2
```

L'intégrale d'une fonction exponentielle polynomiale:

```
>>> integrate(5*x**2 * exp(x) * sin(x), x)
   2  x            2  x                              x             x
5*x *e *sin(x)   5*x *e *cos(x)        x          5*e *sin(x)   5*e *cos(x)
-------------- - -------------- + 5*x*e *cos(x) - ----------- - -----------
      2               2                              2             2
```

Deux intégrales non élémentaires:

```
>>> from sympy import erf
>>> integrate(exp(-x**2)*erf(x), x)
  ____    2
\/ pi *erf (x)
--------------
      4
```

Calculer l'intégrale de `x^2 \cos x` par rapport à `x`:

```
>>> integrate(x**2 * cos(x), x)
 2
x *sin(x) + 2*x*cos(x) - 2*sin(x)
```

Calculer l'intégrale définie de `x^2 \cos x` par rapport à `x` sur l'intervalle de `0` à `pi/2`:

```
>>> integrate(x**2 * cos(x), (x, 0, pi/2))
       2
     pi
-2 + ---
      4
```

7.6 Sommes, produits, dérivées et intégrales non évaluées

Les fonctions `summation`, `product`, `diff` et `integrate` ont tous un équivalent qui retourne un résultat non évalué. Elles s'utilisent avec la même syntaxe, mais portent un autre nom et commencent avec une majuscule: `Sum`, `Product`, `Derivative`, `Integral`.

```
>>> from sympy import Sum, Product, Derivative, Integral, sin, oo
>>> from sympy.abc import n, x
>>> Sum(1/n**2, (n, 1, oo))
  oo
____
\   `
 \    1
  \   --
  /    2
 /    n
```

```
/___,
n = 1
>>> Product(n, (n,1,10))
  10
_____
|   | n
|   |
n = 1
>>> Derivative(sin(x**2), x)
d /   / 2\\
--\sin\x //
dx
>>> Integral(1/x**2, (x,1,oo))
 oo
  /
 |
 |  1
 |  -- dx
 |   2
 |  x
 |
/
1
```

Pour les évaluer, on ajoute `.doit()`:

```
>>> Sum(1/n**2, (n, 1, oo)).doit()
  2
pi
---
 6
>>> Product(n, (n,1,10)).doit()
3628800
>>> Derivative(sin(x**2), x).doit()
       / 2\
2*x*cos\x /
>>> Integral(1/x**2, (x,1,oo)).doit()
1
```

Cela est utile pour écrire des équations:

```
>>> A = Sum(1/n**2, (n, 1, oo))
>>> B = Product(n, (n,1,10))
>>> C = Derivative(sin(x**2), x)
>>> D = Integral(1/x**2, (x,1,oo))
>>> from sympy import Eq
>>> Eq(A, A.doit())
  oo
____
\   `         2
 \    1    pi
  \   -- = ---
  /    2    6
 /    n
```

```
/___,
n = 1
>>> Eq(B, B.doit())
  10
_____
|   | n = 3628800
|   |
n = 1
>>> Eq(C, C.doit())
d /   / 2\\          / 2\
--\sin\x // = 2*x*cos\x /
dx
>>> Eq(D, D.doit())
 oo
  /
 |
 |  1
 |  -- dx = 1
 |   2
 |  x
 |
/
1
```

7.7 Intégrales multiples

Pour faire une intégrale double, on peut intégrer le résultat d'une première intégration comme ceci:

```
>>> from sympy.abc import x,y
>>> integrate(integrate(x**2+y**2, x), y)
 3        3
x *y   x*y
---- + ----
 3      3
```

Mais, il est plus commode d'utiliser une seule fois la commande `integrate` et sympy permet de le faire:

```
>>> integrate(x**2+y**2, x, y)
 3        3
x *y   x*y
---- + ----
 3      3
```

Pour les intégrales définies multiples, on spécifie les intervalles pour chaque variable entre parenthèses. Ici, on fait l'intégrale sur les valeurs de `x` dans l'intervalle `[0,y]`, puis pour les valeurs de `y` dans l'intervalle `[0,10]`:

```
>>> integrate(x**2+y**2, (x,0,y), (y,0,10))
10000/3
```

7.8 Développement en séries

On calcule la série de Taylor d'une `expression` qui dépend de `x` au point `x0` d'ordre `n` avec la syntaxe `series(expression, x, x0, n)`. Par exemple, la série de Maclaurin (une série de Maclaurin est une série de Taylor au point `x0=0`) de `cos(x)` d'ordre 14 est:

```
>>> from sympy import series, cos
>>> from sympy.abc import x
>>> series(cos(x), x, 0, 14)
     2    4     6      8       10         12
    x    x     x      x       x          x         / 14\
1 - -- + -- - --- + ----- - ------- + --------- + O\x  /
    2    24   720   40320   3628800   479001600
```

Par défaut, le développement est efféctuée en `0` et est d'ordre 6:

```
>>> series(cos(x), x)
     2    4
    x    x     / 6\
1 - -- + -- + O\x /
    2    24
```

De façon équivalente, on peut aussi utilise la syntaxe `expression.series(x, x0, n)`:

```
>>> (1/cos(x**2)).series(x, 0, 14)
     4      8       12
    x    5*x    61*x      / 14\
1 + -- + ---- + ------ + O\x  /
    2     24     720
```

Le développement de Taylor de `log` se fait en `x0=1`:

```
>>> from sympy import log
>>> series(log(x), x, 0)
log(x)
>>> series(log(x), x, 1)
            2          3          4          5
     (x - 1)    (x - 1)    (x - 1)    (x - 1)         /       6        \
-1 - -------- + -------- - -------- + -------- + x + O\(x - 1) ; x -> 1/
        2          3          4          5
```

7.9 Équations différentielles

Une équation différentielle est une relation entre une fonction inconnue et ses dérivées. Comme la fonction est inconnue, on doit la définir de façon abstraite comme ceci:

```
>>> from sympy import Function
>>> f = Function("f")
```

Déjà, cela permet d'écrire `f` et `f(x)`:

```
>>> f
f
```

```
>>> from sympy.abc import x
>>> f(x)
f(x)
```

On peut définir les dérivées de `f` à l'aide de la fonction `Derivative` de sympy:

```
>>> from sympy import Derivative
>>> Derivative(f(x), x)            # ordre 1
d
--(f(x))
dx
>>> Derivative(f(x), x, x)         # ordre 2
  2
 d
---(f(x))
  2
dx
```

En utilisant, `Eq` on peut définir une équation impliquant la fonction `f` et ses dérivées, c'est-à-dire une équation différentielle:

```
>>> Eq(f(x), Derivative(f(x),x))
       d
f(x) = --(f(x))
       dx
```

Puis, on peut la résoudre avec la fonction `dsolve` de sympy avec la syntaxe `dsolve(equation, f(x))` et trouver quelle fonction `f(x)` est égale à sa propre dérivée:

```
>>> from sympy import dsolve
>>> dsolve(Eq(f(x), Derivative(f(x),x)), f(x))
           x
f(x) = C1*e
```

Voici un autre exemple qui trouve une fonction égale à l'opposé de sa dérivée d'ordre 2:

```
>>> Eq(f(x), -Derivative(f(x),x,x))
           2
          d
f(x) = - ---(f(x))
           2
         dx
>>> dsolve(Eq(f(x), -Derivative(f(x),x,x)), f(x))
f(x) = C1*sin(x) + C2*cos(x)
```

Résoudre une équation différentielle ordinaire comme `f''(x) + 9 f(x) = 1`

```
>>> dsolve(Eq(Derivative(f(x),x,x) + 9*f(x), 1), f(x))
f(x) = C1*sin(3*x) + C2*cos(3*x) + 1/9
```

Pour définir la dérivée, on peut aussi utiliser `.diff()`. L'exemple précédent s'écrit:

```
>>> dsolve(Eq(f(x).diff(x, x) + 9*f(x), 1), f(x))
f(x) = C1*sin(3*x) + C2*cos(3*x) + 1/9
```

Finalement, voici un exemple impliquant deux équations:

```
>>> from sympy.abc import x,y,t
>>> eq1 = Eq(Derivative(x(t),t), x(t)*y(t)*sin(t))
>>> eq2 = Eq(Derivative(y(t),t), y(t)**2*sin(t))
>>> systeme = [eq1, eq2]
>>> systeme
 d                          d           2
[--(x(t)) = x(t)*y(t)*sin(t), --(y(t)) = y (t)*sin(t)]
 dt                         dt
>>> dsolve(systeme)
                  C1
                -e                     -1
set([x(t) = ---------------, y(t) = -----------])
               C1                  C1 - cos(t)
           C2*e   - cos(t)
```

8 Algèbre linéaire

Cette section concerne l'algèbre linéaire et les matrices. On trouvera d'autres exemples dans le tutoriel de Sympy sur le même sujet: http://docs.sympy.org/latest/tutorial/matrices.html

8.1 Définir une matrice

En SymPy, on peut créer une matrice avec la fonction `Matrix`:

```
>>> from sympy import Matrix
```

Il suffit d'écrire les entrées lignes par lignes avec la syntaxe suivante:

```
>>> Matrix([[2, 5, 6], [4, 7, 10], [1, 0, 3]])
[2  5  6 ]
[        ]
[4  7  10]
[        ]
[1  0  3 ]
```

Une autre façon équivalente est de spécifier le nombre de lignes, le nombre de colonnes et puis la liste des entrées:

```
>>> Matrix(2, 3, [1, 2, 3, 4, 5, 6])
[1  2  3]
[       ]
[4  5  6]
```

Par défaut, si on ne spécifie pas les dimensions de la matrice, un vecteur colonne est retourné:

```
>>> Matrix([1,2,3,4])
[1]
```

```
[ ]
[2]
[ ]
[3]
[ ]
[4]
```

8.2 Opérations de base

Les opérations de l'algèbre des matrices (addition, multiplication, multiplication par un scalaire) sont définies naturellement:

```
>>> M = Matrix([[5, 2], [-1, 7]])
>>> N = Matrix([[0, 4], [0, 5]])
>>> M + N
[5   6 ]
[      ]
[-1  12]
>>> M * N
[0  30]
[     ]
[0  31]
>>> 4 * M
[20  8 ]
[      ]
[-4  28]
>>> M ** 5
[-475   12242]
[             ]
[-6121  11767]
```

De même, on peut calculer l'inverse d'une matrice si elle est inversible:

```
>>> M**-1
[7/37  -2/37]
[           ]
[1/37  5/37 ]
>>> N**-1
Traceback (most recent call last):
...
ValueError: Matrix det == 0; not invertible.
```

La transposition d'une matrice se fait avec `.transpose()`:

```
>>> from sympy import I
>>> M = Matrix(( (1,2+I,5), (3,4,0) ))
>>> M
[1  2 + I  5]
[           ]
[3    4    0]
>>> M.transpose()
[  1    3]
[        ]
```

```
[2 + I  4]
[        ]
[  5    0]
```

8.3 Accéder aux coefficients

```
>>> from sympy import I
>>> M = Matrix(( (1,2+I,5), (3,4,0) ))
>>> M
[1  2 + I  5]
[           ]
[3    4    0]
```

On accède à l'élément en position `(i,j)` en écrivant `M[i,j]`:

```
>>> M[0,1]
2 + I
>>> M[1,1]
4
```

> ***Attention!***
>
> Les indices des positions commencent à zéro!!

On accède aux lignes et au colonnes d'une matrices avec les méthodes `row` et `col`:

```
>>> M.row(1)
[3  4  0]
>>> M.col(0)
[1]
[ ]
[3]
```

8.4 Construction de matrices particulières

Les fonctions `zeros` et `ones` permettent de créer des matrices de zéros et de uns:

```
>>> from sympy import ones,zeros
>>> ones(2)
[1, 1]
[1, 1]
>>> zeros((2, 4))
[0, 0, 0, 0]
[0, 0, 0, 0]
```

La fonction `eye` de sympy permet de créer une matrice identité:

```
>>> from sympy import eye
>>> eye(3)
[1, 0, 0]
[0, 1, 0]
[0, 0, 1]
```

La fonction `diag` permet de créer une matrice diagonale:

```
>>> from sympy import diag
>>> diag(1,2,3)
[1  0  0]
[       ]
[0  2  0]
[       ]
[0  0  3]
```

Les éléments de la diagonales peuvent être eux-mêmes des matrices:

```
>>> diag(1, 2, Matrix([[7,8],[2,3]]))
[1  0  0  0]
[          ]
[0  2  0  0]
[          ]
[0  0  7  8]
[          ]
[0  0  2  3]
```

8.5 Matrice échelonnée réduite

On calcule la forme échelonnée réduite d'une matrice avec la méthode `rref` (abbréviation de *reduced row echelon form* en anglais):

```
>>> M = Matrix([[1, 2, 0, 3], [2, 6, 5, 1], [-1, -4, -5, 2]])
>>> M.rref()
([1  0  -5    8  ], [0, 1])
 [                ]
 [0  1  5/2  -5/2]
 [                ]
 [0  0   0    0  ]
```

8.6 Noyau

On calcule le noyau d'une matrice avec `nullspace`:

```
>>> M = Matrix([[1, 2, 0, 3], [2, 6, 5, 1], [-1, -4, -5, 2]])
>>> M.nullspace()
[[ 5  ], [-8 ]]
 [    ]  [   ]
 [-5/2]  [5/2]
 [    ]  [   ]
 [ 1  ]  [ 0 ]
```

```
[    ]  [   ]
[ 0  ]  [ 1 ]
```

8.7 Déterminant

On calcule le déterminant avec la méthode `det`:

```
>>> M = Matrix([[2, 5, 6], [4, 7, 10], [1, 0, 3]])
>>> M.det()
-10
```

8.8 Polynôme caractéristique

La méthode `charpoly` permet de calculer le polynôme caractéristique d'une matrice carrée:

```
>>> M = Matrix([[3, -2,  4, -2], [5,  3, -3, -2], [5, -2,  2, -2], [5, -2, -3,  3]])
>>> from sympy.abc import x
>>> M.charpoly(x)
PurePoly(x**4 - 11*x**3 + 29*x**2 + 35*x - 150, x, domain='ZZ')
```

On ajoute `.as_expr()` pour obtenir l'expression symbolique du polynôme caractéristique:

```
>>> M.charpoly(x).as_expr()
 4       3       2
x  - 11*x  + 29*x  + 35*x - 150
>>> from sympy import factor
>>> factor(_)
       2
(x - 5) *(x - 3)*(x + 2)
```

8.9 Valeurs propres et vecteurs propres

Continuons avec la même matrice `M` définie précédemment:

```
>>>  M
[3  -2  4   -2]
[              ]
[5  3   -3  -2]
[              ]
[5  -2  2   -2]
[              ]
[5  -2  -3  3 ]
```

Soient les vecteurs colonnes `w` et `v` suivants:

```
>>> w = Matrix((1,2,3,4))
>>> v = Matrix((1,1,1,0))
>>> w
[1]
[ ]
[2]
[ ]
```

```
[3]
[ ]
[4]
>>> v
[1]
[ ]
[1]
[ ]
[1]
[ ]
[0]
```

En général, l'image par `M` d'un vecteur n'a rien à voir avec ce vecteur. Par exemple, l'image par `M` de `w` n'a rien à voir avec `w`:

```
>>> M * w
[3 ]
[  ]
[-6]
[  ]
[-1]
[  ]
[4 ]
```

Dans certains cas particuliers, l'image par `M` d'un vecteur retourne un multiple scalaire de ce vecteur. C'est ce qui se produit pour le vecteur `v`:

```
>>> M * v
[5]
[ ]
[5]
[ ]
[5]
[ ]
[0]
```

Le résultat précédent est égal à 5 fois le vecteur `v`:

```
>>> 5 * v
[5]
[ ]
[5]
[ ]
[5]
[ ]
[0]
```

Un vecteur `v` qui satisfait l'équation `M * v = lamda * v` pour un certain nombre réel (ou complexe) `lamda` est appelé *vecteur propre*. Le nombre `lamda` qui satisfait l'équation est appelé *valeur propre*. Il se trouve que les valeurs propres d'une matrice sont les racines de son polynôme caractéristique. Le calcul des valeurs et vecteurs propres d'une matrice est utile dans presque tous les domaines des mathématiques.

En sympy, on calcule les valeurs propres d'une matrice avec la méthode `eigenvals`. Le résultat est un dictionnaire qui associe à chaque valeur propre sa multiplicité algébrique (comme pour le calcul des racines):

```
>>> M.eigenvals()
{-2: 1, 3: 1, 5: 2}
```

Et on calcule les vecteurs propres d'une matrice avec la méthode `eigenvects`:

```
>>> M.eigenvects()
[(-2, 1, [[0]]), (3, 1, [[1]]), (5, 2, [[1], [0 ]])]
          [ ]            [ ]            [ ]  [  ]
          [1]            [1]            [1]  [-1]
          [ ]            [ ]            [ ]  [  ]
          [1]            [1]            [1]  [0 ]
          [ ]            [ ]            [ ]  [  ]
          [1]            [1]            [0]  [1 ]
```

Le calcul précédent montre bien que le vecteur colonne `v = [1, 1, 1, 0]^T` est bien un vecteur propre de la matrice `M` associé à la valeur propre `5` comme on l'avait vu plus tôt. Il permet aussi de réaliser qu'un autre vecteur colonne linéairement indépendant de `v` est aussi un vecteur propre associé à la valeur propre `5`. Finalement, il y a deux autres vecteurs propres associés aux valeurs propres `-2` et `3`.

9 Mathematica

Les thèmes des chapitres 1 à 8 représentent la base des fonctionalités offertes par les logiciels de calcul formel dont SymPy, Sage, Mathematica, Maple, Magma et d'autres font partie. Dans ces notes de cours, nous avons utilisé SymPy dans les 8 premiers chapitres, mais nous aurions pu utiliser un autre logiciel.

Entre deux logiciels qui font la même chose, on retrouve de nombreuses ressemblances, mais aussi des différences. Dans ce cours, nous n'aurons pas le temps d'apprendre à utiliser tous les logiciels de calcul formel ni d'étudier toutes les différences entre les uns et les autres. Toutefois, il est pertinent d'avoir une idée de ce que peuvent être ces ressemblances et ces différences afin d'être prêt à utiliser dans le futur un logiciel différent de celui que l'on connaît. Pour ce faire, il faut apprendre à utiliser au moins un deuxième logiciel de calcul formel.

Dans ce chapitre, nous présentons le logiciel commercial Mathematica. Nous présentons les différences et ressemblances principales avec SymPy. Puis, le chapitre se termine avec des tables qui traduisent en Mathematica l'ensemble des fonctions de SymPy que nous avons présentées dans les chapitre 1 à 8. La plupart du temps, les fonctions s'utilisent avec les mêmes arguments et dans le même ordre. Pour voir des exemples ou pour avoir plus de détails, le lecteur sera invité à consulter la documentation de Mathematica ou internet.

9.1 Résumé des différences entre SymPy et Mathematica

En Python, on doit importer les fonctions que l'on utilise. Une par une en écrivant `from sympy import factor,pi,sin` ou toutes à la fois avec `from sympy import *`. En Mathematica, on n'a **pas besoin d'importer les fonctions** avant de les utiliser.

Tout d'abord, la règle générale en Mathematica est que les fonctions commencent avec une **lettre majuscule** plutôt qu'avec des minuscules. On écrit donc `Sin`, `Pi`, `Factor` en Mathematica plutôt que `sin`, `pi` et `factor` en SymPy.

Les **noms des fonctions sont le plus souvent les mêmes qu'en SymPy**, mais on note quelques différences comme `FactorInteger`, `ArcSin` et `ParametricPlot` en Mathematica versus `factorint`, `asin` et `plot_parametric` en SymPy. On trouve quelques autres fonctions qui portent des noms différents dans les tables à la fin de ce chapitre.

Pour évaluer une fonction, on utilise les parenthèses en SymPy `sin(pi)` et les **crochets** `[]` en Mathematica `Sin[Pi]`.

Pour évaluer une cellule dans Mathematica comme dans Jupyter, c'est la même chose, on appuie sur les touches `MAJUSCULE + ENTRÉE`.

En SymPy comme en Python, la **multiplication implite** `3x` n'est pas possible et il faut toujours écrire `3*x` en spécifiant l'opération de multiplication (*). En Mathematica, les espaces vides sont interprétés comme des multiplications et `3x` sans espace est interprété comme le produit du nombre `3` par la variable `x`.

En Mathematica, l'**exponentiation** s'écrit avec le symbole chapeau `^` plutôt qu'avec `**` et la division de nombres entiers retourne bien un nombre rationnel.

Les listes comme `[x, 0, 2*pi]` et les tuples comme `(x, 0, 2*pi)` en Python et SymPy sont souvent utilisés comme arguments de fonctions. De la même façon en Mathematica, les listes sont beaucoup utilisées. Toutefois, en Mathematica, on utilise les **accolades** `{` **et** `}` **pour définir une liste**. On écrit alors `{x, 0, 2*Pi}`. Par exemple, l'intégrale de sin(x) s'écrit en SymPy:

```
>>> integrate(sin(x), (x, 0, 2*Pi))
```

En Mathematica, cela devient:

```
>>> Integrate[Sin[x], {x, 0, 2*Pi}]
```

En SymPy, on doit définir les symboles avec la fonction `Symbol` ou bien on peut les importer du sous-module `sympy.abc`. En Mathematica, **toute variable non définie est par défaut un symbole** et prend la couleur bleue (Mathematica v.10.0) pour les identifier.

En Mathematica, contrairement à SymPy, il n'y a pas de distinction entre la variable `x` et le symbole `x`. On peut remarquer cette différence en testant le code suivant qui ne retourne pas la même chose en Mathematica et en SymPy:

```
>>> Clear[x]                          # Mathematica
>>> from sympy.abc import x           # SymPy
>>> expr = x + 1
>>> x = 100
>>> expr
```

En Mathematica, la ligne `x = 100` change la valeur de la variable `x` et `x` n'est plus un symbole (sa couleur change et devient noir). Cela a un effet de bord sur l'expression `expr` qui retourne la valeur `101`. En SymPy, `expr` contient le symbole `x` plus `1`. La ligne `x = 100` change la valeur de la variable `x` mais n'a pas d'effet sur les expressions qui contiennent le symbole `x`. Donc, `expr` retourne bien `x + 1`.

La **complétion automatique** par la touche tabulation en Jupyter a son équivalent en Mathematica. Des suggestions de noms de fonctions apparraissent à l'écran en Mathematica pour compléter les premières lettres que l'on écrit.

Pour **accéder à la documentation d'une fonction** en Mathematica, il suffit d'ajouter un point d'interrogation *avant* le nom de la fonction et d'évaluer la cellule. En Jupyter, le point d'interrogation peut être placé avant ou après le nom de la fonction pour accéder à sa documentation.

Python et SymPy sont basés sur la programmation orientée objet ce qui explique que l'on peut écrire la fonction *après* l'objet comme par exemple `pi.n()` pour calculer les décimales de pi et `M.det()` pour calculer le déterminant d'une matrice. En Mathematica, cela ne se produit jamais. Les **fonctions s'écrivent toujours avant les objets** comme `N[Pi]` et `Det[M]`.

La plupart du temps, la **syntaxe** est exactement la même en SymPy et en Mathematica, mais il y a quelques exceptions. Par exemple, le calcul d'une limite en SymPy s'écrit:

```
>>> limit(sin(x)/x, x, 0)
```

alors, qu'en Mathematica, on utilise la flèche `->` pour indiquer qu'une variable tend vers une valeur:

```
>>> Limit[Sin[x]/x, x->0]
```

On trouvera une liste des particularités du langage SymPy ici: http://docs.sympy.org/dev/tutorial/gotchas.html. On trouvera le manuel de référence de Mathematica en ligne à l'adresse http://reference.wolfram.com/language/ et des vidéos d'introduction à l'adresse https://www.wolfram.com/broadcast/screencasts/handsonstart/.

9.2 Tables de traduction entre SymPy et Mathematica

Les tables ci-bas donnent les équivalents en Mathematica pour chacune des fonctionnalités de SymPy que l'on a vu dans les chapitres 1 à 8. La plupart du temps, la syntaxe est essentiellement la même. On doit simplement ajouter des majuscules et remplacer les parenthèses par des crochets `[]` ou des accolades `{}` selon que l'on évalue une fonction ou que l'on définit une liste.

En cas de problème, il ne faut pas hésiter à consulter la documentation de Mathematica en ajoutant un point d'interrogation avant le nom de la fonction, comme `?Factor` ou sinon chercher des exemples sur internet.

Chapitre 1: Interface

Thème	Jupyter, IPython, SymPy	Mathematica
Évaluer une cellule	MAJUSCULE + ENTRÉE	MAJUSCULE + ENTRÉE
Ne pas afficher le résultat	Ajouter un `";"`	Ajouter un `";"`
Règle générale	minuscules : `factor`	une majuscule: `Factor`
Aide	`?factor` ou `factor?`	`?Factor`
Évaluer une fonction	`factor(x)`	`Factor[x]`
Liste	`[x,0,2*pi]`	`{x,0,2*pi}`
n-uplet	`(x,0,2*pi)`	`{x,0,2*pi}`
Symboles	`x = Symbol('x')`	x, symbole par défaut (en bleu)
Variables et affectation	`k = 4`	`k = 4`
Les résultats précédents	`_, __, ___`	`%, %%, %%%`
Le 12e résultat	`Out[12]`	`Out[12]`
Temps de calcul	`%time factorint(100)`	`Timing[FactorInteger[100]]`
Approximation numérique	`pi.n()` ou `pi.evalf()` ou `N(pi)`	`N[Pi]`
Réinitialiser les variables	`%reset`	`?`

Chapitre 2 et 3: Calculatrice et arithmétique

SymPy	Mathematica
`2 + 3, 10 - 5`	`2 + 3, 10 - 5`
`2 * 3`	`2 * 3` ou `2 3`

`3 ** 10`	`3 ^ 10`
`1 / (1 + 4*5)**2`	`1 / (1 + 4*5)^2`
`Rational(3,4), S(3)/4`	`3/4`
`sin, cos, tan`	`Sin, Cos, Tan`
`asin, acos, atan`	`ArcSin, ArcCos, ArcTan`
`exp(2), log(2)`	`Exp[2], Log[2]`
`sqrt(2)`	`Sqrt[2]`
`E, I, pi, oo`	`E, I, Pi, Infinity`
`re, im, arg, conjugate`	`Re, Im, Arg, Conjugate`
`factorial(100)`	`Factorial[100]`
`factorint(100)`	`FactorInteger[100]`

Chapitre 4: Calcul symbolique

SymPy	Mathematica
`Symbol, symbols, sympy.abc`	les symboles sont définis automatiquement
`srepr`	`FullForm, TreeForm`
`3*x`	`3*x` ou `3x`
`(x+1).subs(x,0)`	`?`
`apart, together, cancel, collect`	`Apart, Together, Cancel, Collect`
`factor, expand`	`Factor, Expand`
`simplify`	`Simplify, FullSimplify`
`radsimp, ratsimp`	`?`

Chapitre 5: Résolution d'équations

SymPy	Mathematica
`Eq(x + y, 4)`	`x + y == 4`
`solve(x**2 - 3, x)`	`Solve[x^2-3 == 0, x]`
`roots, root, real_root, RootOf`	`Root`
`nsolve(x**2 - 3, x, 2)`	`NSolve[x^2-3 == 0, x]`

Chapitre 6: Tracer une fonction

SymPy	Mathematica
`plot(sin(x), (x,0,2*pi))`	`Plot[Sin[x], {x,0,2 Pi}]`
`plot3d`	`Plot3D`
`plot_parametric`	`ParametricPlot`
`plot3d_parametric_line`	`ParametricPlot3D`
`plot3d_parametric_surface`	`ParametricPlot3D`

`plot_implicit`	`RegionPlot[ImplicitRegion[...]]`
`mpmath.cplot`	`?`

Chapitre 7: Calcul différentiel et intégral

SymPy	Mathematica
`limit, diff, integrate, series`	`Limit, D, Integrate, Series`
`dsolve`	`DSolve`
`summation, product`	`Sum, Product`
`Function`	les symboles sont définis automatiquement
`Derivative, Integral, Sum, Product`	`?`

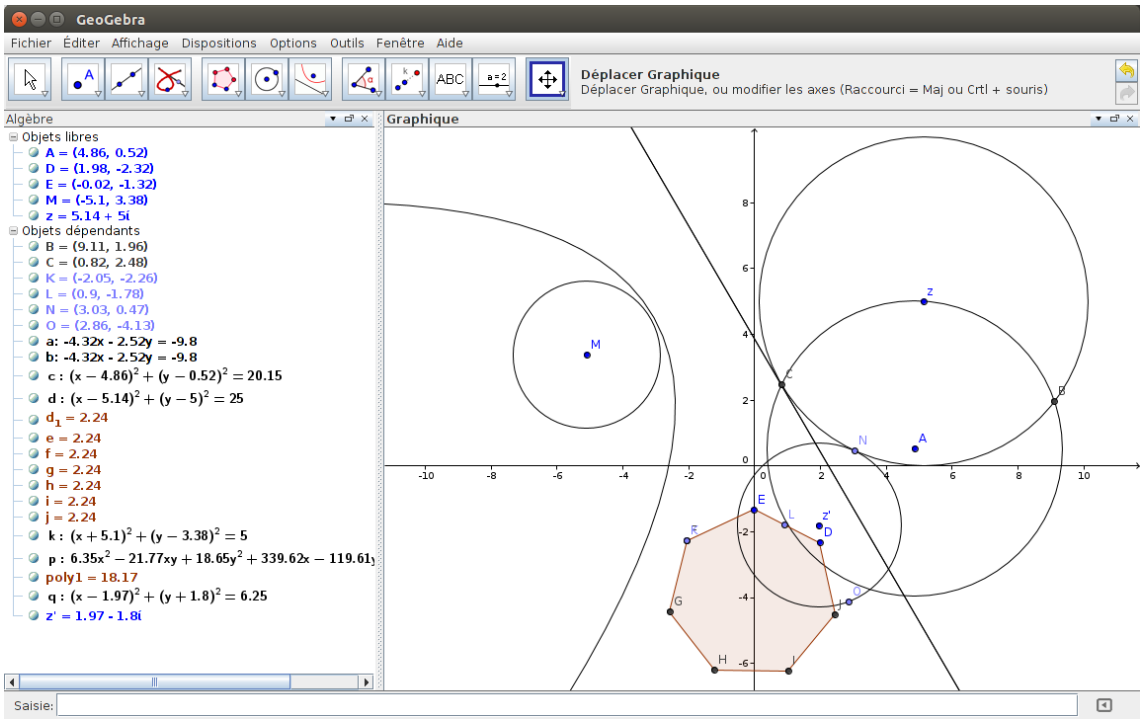
Chapitre 8: Algèbre linéaire

SymPy	Mathematica
`M = Matrix([[1,2],[3,4]])`	`M = {{1,2},{3,4}}`
`diag(2,3,4)`	`DiagonalMatrix[{2,3,4}]`
`eye(5)`	`IdentityMatrix[5]`
`zeros(3,5)`	`Table[0, {i,1,3}, {j,1,5}]`
`ones(3,5)`	`Table[1, {i,1,3}, {j,1,5}]`
`M + N`	`M + N`
`3 * M`	`3 * M`
`M * N`	`M . N`
`M ** 4`	`MatrixPower[M, 4]`
`M.transpose()`	`Transpose[M]`
`M.det(), M.rank(), M.nullspace()`	`Det[M], Rank[M], NullSpace[M]`
`M ** -1` ou `M.inverse()`	`Inverse[M]`
`M.rref()`	`RowReduce[M]`
`M.charpoly()`	`CharacteristicPolynomial[M]`
`M.eigenvals(), M.eigenvects()`	`Eigenvalues[M], Eigenvectors[M]`

10 GeoGebra

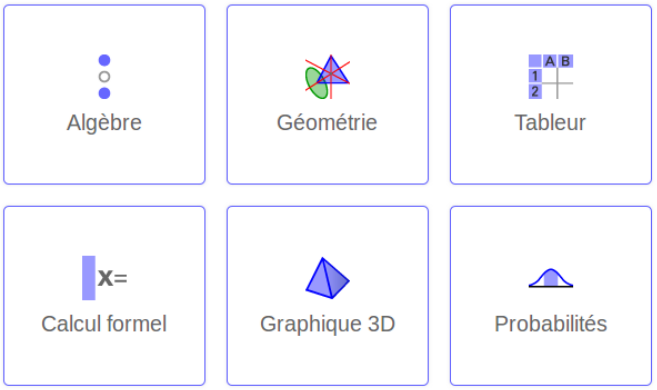
GeoGebra est un environnement mathématique dynamique qui allie géométrie, algèbre et calculs. Il a été développé pour apprendre et enseigner les mathématiques par Markus Hohenwarter et une équipe internationale de programmeurs. GeoGebra peut être installé sur tous les types d'ordinateurs, mais il y a aussi une version pour les tablettes et téléphones. Finalement, il est possible d'utiliser GeoGebra directement sur internet à l'adresse app.geogebra.org et de profiter de plus de 300 000 ressources d'enseignement et d'apprentissage interactif sur tube.geogebra.org.

Crédits: certains passages de ce chapitre proviennent de la page de wikipédia sur GeoGebra et du wiki de GeoGebra où on trouvera de plus amples informations.

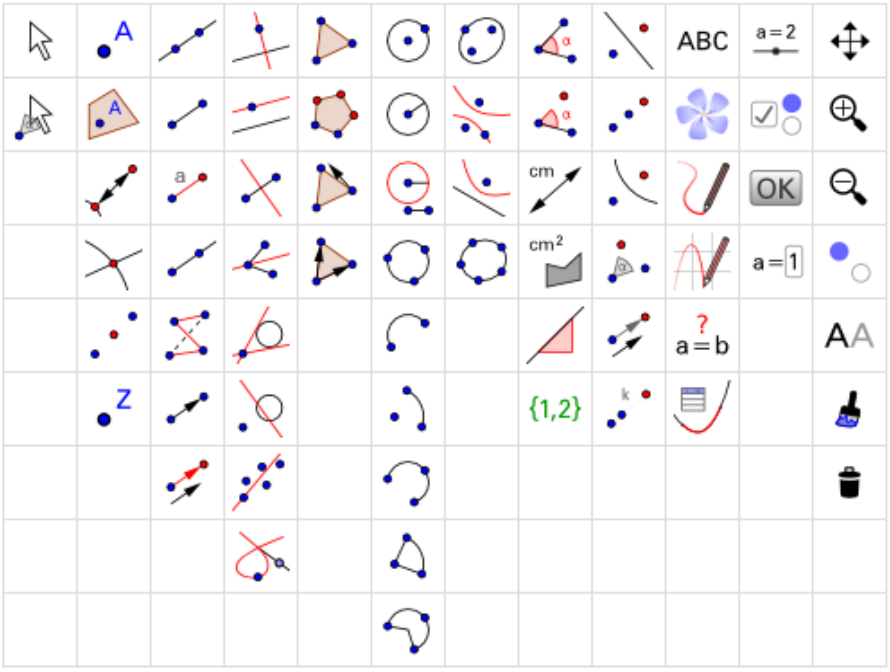


L'interface principale est divisée en fenêtres. Par défaut, la fenêtre *Algèbre* est affichée sur le côté gauche, la fenêtre *Graphique* sur le côté droit. Au dessus d'elles, il y a la Barre de menus et la Barre d'outils. Beaucoup de fonctionnalités de GeoGebra peuvent être appelées par des raccourcis clavier. GeoGebra contient aussi des fonctions d'accessibilité telle qu'un clavier virtuel.

Tout objet créé dans la fenêtre *Graphique* a aussi une représentation algébrique dans la fenêtre *Algèbre* et peut être modifié à partir de la fenêtre *Algèbre* ou de la fenêtre *Graphique*.



GeoGebra peut être utilisé pour faire du calcul symbolique (la fenêtre *Calcul formel*) comme SymPy et Mathematica ou pour manipuler des données (la fenêtre *Tableur*) comme Excel. Dans ce cours, nous nous concentrerons plutôt sur les fonctionnalités de géométrie et d'algèbre offertes par GeoGebra.



Les icônes de la barre d'outils permettent de faire un ensemble de constructions géométriques dans la fenêtre *Graphique*. Colonne par colonne et de gauche à droite dans l'image ci-haut, on retrouve différents outils pour:

- déplacer des points
- créer des points
- créer des droites, segments et vecteurs
- créer des droites perpendiculaires, parallèles, bissectrices, médiatrices, tangentes
- créer des polygones
- créer des cercles, arcs de cercle, de secteurs
- créer des ellipses, d'hyperboles, de paraboles et autres coniques
- calculer des angles, distances, aires, pentes
- calculer symmétries, d'inversions, de rotations, d'homothéties, de translations
- insérer du texte et des images
- créer des curseurs et des boutons
- déplacer, zoomer et afficher ou cacher des objets

La plupart du temps, les quelques mots d'aide indiqués dans la Barre d'outils (cette option doit être activée dans les préférences) sont suffisants pour comprendre comment utiliser l'outil sélectionné. Sinon, on se référera à la page wiki qui décrit comment utiliser chacun des icônes ci-haut ou sinon aux chapitres 1 et 2 (pages 1 à 30) du Manuel d'introduction à GeoGebra [GeoGebra].

Apprendre à utiliser GeoGebra sur Youtube

Comme GeoGebra est un outil très dynamique et interactif, il est parfois plus facile d'apprendre à l'utiliser en regardant comment les autres font. La chaîne Youtube de GeoGebra contient une multitude de vidéos qui permettent d'en apprendre sur toutes les fonctionalités de GeoGebra, sans compter les vidéos créés par les utilisateurs.

Ci-bas, on retrouve les vidéos qui couvrent les chapitres 1 et 2 du manuel d'introduction à Géogebra mentionné plus haut:

- Construction d'un rectangle, 58 s.
- Construction d'un triangle équilatéral, 1min.
- Construction d'un carré, 1min 21s.
- Construction d'un hexagone régulier, 1min 51s.
- Construction d'un cercle circonscrit à un triangle, 58s.
- Théorème du triangle inscrit dans un demi-cercle, 57s.
- Construction des tangentes à un cercle, 3min 11s.
- Explorer les paramètres d'un polynôme quadratique, 1min 18s.
- Utilisation des curseurs pour modifier des coefficients, 1min 15s.
- Visualiser la multiplication des nombres entiers, 5min 31s.
- Geogebra fun trick, 2min 40s.

11 Types de données de Python

Nous avons vu dans les chapitres précédents quelques types de données comme les entiers et les nombres flottants de Python ainsi que quelques structures de données (listes, tuples, dictionnaires) utilisées comme argument ou comme valeur de retour de certaines fonctions de SymPy. Dans ce chapitre, nous allons présenter avec plus de détails les types de données qui sont les plus souvent utilisées en programmation.

11.1 Le type d'un objet

Construisons le nombre entier 4 de deux façons différentes, avec Python puis avec SymPy:

```
>>> a = 4
>>> from sympy import S
>>> b = S(4)
```

Le nombre 4 est stocké dans la variable `a` et dans la variable `b`:

```
>>> a
4
>>> b
4
```

Bien qu'ils représentent tous deux le nombre 4 et qu'ils s'affichent de la même façon à l'écran, ils ne se comportent pas de la même façon:

```
>>> a/5
0.8
>>> b/5
4/5
```

C'est que les *objets* stockés dans les variables `a` et `b` ne sont pas du même *type*. La fonction `type` permet de connaître le type d'un `objet` avec la syntaxe `type(objet)`:

```
>>> type(a)
<type 'int'>
>>> type(b)
<class 'sympy.core.numbers.Integer'>
```

Cela explique le comportement différent de `a/5` qui retourne un nombre décimal et `b/5` qui retourne un nombre rationel de SymPy:

```
>>> type(a/5)
<type 'float'>
>>> type(b/5)
<class 'sympy.core.numbers.Rational'>
```

Comme le comportement d'un objet dépend de son type, il est souvent pertinent de vérifier le type d'un objet avant de l'utiliser. Les autres types très communs que nous allons voir dans les sections suivantes sont ci-dessous:

```
>>> type(4)
<type 'int'>
>>> type(4.0)
<type 'float'>
>>> type(True)
<type 'bool'>
>>> type('bonjour')
<type 'str'>
>>> type([3,4,5])
<type 'list'>
>>> type((3,4,5))
<type 'tuple'>
>>> type({2:3, 4:5})
<type 'dict'>
```

11.2 Nombres entiers (type `int`)

Les nombres entiers sont créés simplement en Python:

```
>>> 4
4
```

Ils sont aussi obtenus par le résultat d'opérations sur les nombres entiers comme l'addition, la multiplication, la soustraction, le modulo et le quotient:

```
>>> 4 + 6
10
>>> 7 * 9
63
>>> 4 - 6
-2
>>> 27 % 10
7
>>> 27 // 10
2
```

On peut vérifier que le résultat des opérations ci-haut est bel et bien un entier Python de type `int`:

```
>>> type(27 // 10)
<type 'int'>
```

En Python, la fonction `int` permet de créer un entier de type `int`:

```
>>> int()
0
>>> int(4)
4
```

Cette fonction permet aussi de traduire un objet d'un autre type en un nombre entier de Python de type `int`:

```
>>> int(4.02)
4
>>> int('41234')
41234
```

Pour stocker des nombres entiers un peu plus grand, Python utilise une autre structure de données appelé entier `long`. On peut tester à partir d'où cela se produit:

```
>>> type(2 ** 61)
<type 'int'>
>>> type(2 ** 62)
<type 'int'>
>>> type(2 ** 63)
<type 'long'>
>>> type(2 ** 64)
<type 'long'>
```

11.3 Nombres flottants (type `float`)

Les nombres décimaux aussi appelé nombre flottants ou nombre à virgule flottante sont créés simplement en Python:

```
>>> 4.
4.0
```

Ils sont aussi obtenus par le résultat d'opérations sur les nombres flottants comme l'addition, la multiplication, la soustraction, le modulo et le quotient:

```
>>> 4. * 3.41
13.64
```

On vérifie que le type du résultat précédent est bel et bien un nombre flottant de type `float`:

```
>>> type(_)
<type 'float'>
```

Les nombres flottants peuvent aussi être obtenus comme résultats d'opérations impliquant des nombres d'autres types comme la multiplication par un nombre entier ou la division de deux nombres entiers:

```
>>> 4. * 3
12.0
>>> 4 / 5
0.8
```

Finalement, les nombres flottants peuvent être créés avec la fonction `float` qui permet aussi de transformer un objet d'un autre type en nombre flottant:

```
>>> float()
0.0
>>> float(34)
34.0
>>> float('1234')
1234.0
>>> float('1234.56')
1234.56
```

11.4 Booléens (type `bool`)

Les booléens permettent de représenter les valeurs *vrai* et *faux*. On les écrit en anglais avec un majuscule:

```
>>> True
True
>>> False
False
```

Les valeurs `True` et `False` sont des objets de type `bool`:

```
>>> type(False)
<type 'bool'>
>>> type(True)
<type 'bool'>
```

Les opérations de base sur les booléens retournent aussi des booléens:

```
>>> True or False
True
>>> False and True
False
```

Si cela est nécessaire, voici toutes les possibilités de valeurs d'entrées pour le ET logique `and` qui retourne *vrai* lorsque les deux valeurs d'entrées sont vraies:

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
```

```
>>> False and False
False
```

Pareillement le OU logique (`or`) retourne `True` dès qu'une des deux valeurs est vraie:

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

La négation (`not`) retourne l'opposé d'une valeur booléenne:

```
>>> not True
False
>>> not False
True
```

Un booléen peut être retourné par des fonctions ou des tests de comparaison:

```
>>> 13 == 5 + 8
True
>>> 20 > 34
False
```

La fonction `bool` permet de transformer un objet en un booléen. En général, les valeurs zéro ou les listes vides sont transformées en `False` et les valeurs non nulles ou les listes non vides sont transformées en `True`:

```
>>> bool(113)
True
>>> bool(0)
False
>>> bool(1)
True
```

11.5 Chaînes de caractères (type `str`)

En Python, les chaînes de caractères sont définies par l'utilisation des simple guillemets (`'`) ou des doubles guillemets (`"`):

```
>>> 'bonjour'
'bonjour'
>>> "bonjour"
'bonjour'
```

Si on veut utiliser les simples guillemets à l'intérieur de la chaînes de caractères, on doit utiliser les doubles pour l'entourer et vice versa:

```
>>> "aujourd'hui"
"aujourd'hui"
>>> 'Je suis "ici"'
'Je suis "ici"'
```

Pour utiliser à la fois des simples et des doubles guillemets dans la chaîne de caractères, on utilise des triple double guillemets pour entourer la chaîne de caractères:

```
>>> """Je suis "ici" aujourd'hui"""
'Je suis "ici" aujourd\'hui'
```

On peut créer des chaînes de caractères à partir d'autres objets en utilisant la fonction `str`:

```
>>> str(12345)
'12345'
>>> str(12345.789)
'12345.789'
```

Pour accéder aux lettres d'une chaîne de caractères, on utilise les crochets après la variable de la façon suivante:

```
>>> w = 'bonjour'
>>> w[0]
'b'
>>> w[1]
'o'
```

Comme vous remarquez, l'indexation commence à zéro et non pas à un. C'est comme ça en Python. Ainsi la septième et dernière lettre du mot bonjour est à la position 6:

```
>>> w[6]
'r'
```

On peut aussi compter à partir de la fin avec des indices négatifs. La position `-1` retourne la dernière lettre:

```
>>> w[-1]
'r'
```

On peut accéder aux sous-chaînes de la position `i` à la position `j-1` avec la syntaxe `w[i:j]` de la façon suivante:

```
>>> w[2:5]
'njo'
```

Si on ne spécifie pas le début ou la fin, alors le comportement par défaut est d'aller jusqu'au bout:

```
>>> w[:4]
'bonj'
```

12 Listes

En Python, les listes sont beaucoup utilisées. Il est donc important de connaîtres les différentes façons de créer, modifier et utiliser les listes. Dans ce chapitre, nous allons voir les différentes opérations sur les listes.

Les listes sont créées avec l'utilisation des crochets et on peut mettre n'importe quel objet dans une liste:

```
>>> [2, 3, 4, 'bateau', 3.24]
[2, 3, 4, 'bateau', 3.24]
```

On accède aux éléments de la liste de la même façon qu'avec les chaînes de caractères, c'est-à-dire en utilisant les crochets et le premier élément est à la position zéro:

```
>>> L = [2, 3, 4, 'bateau', 3.24]
>>> L[0]
2
>>> L[3]
'bateau'
```

On peut modifier la liste en lui ajoutant des objets avec la méthode `append` de la façon suivante:

```
>>> L.append(789)
>>> L
[2, 3, 4, 'bateau', 3.24, 789]
```

On vérifie que le dernier élément de la liste est bien le nombre entier `789`:

```
>>> L[-1]
789
```

La fonction `list` permet de transformer un objet en liste pourvu qu'il soit itérable:

```
>>> list(w)
['b', 'o', 'n', 'j', 'o', 'u', 'r']
```

Plusieurs autres opérations sont possibles sur les listes. On peut consulter l'aide ou la documentation pour en savoir plus:

```
>>> L.<TAB>
>>> help(list)
```

12.1 Opérations sur les listes

D'abord, créons une liste:

```
>>> L = [1, 8, -4, 38, 8, 8, 4, 18]
```

Pour créer une sous-liste commençant à l'indice 2 jusqu'à l'indice 5-1=4:

```
>>> L[2:5]
[-4, 38, 8]
```

On peut créer une nouvelle liste avec l'opération d'addition (+) qui concatène deux listes. Ceci ne change pas les listes utilisées. Par exemple:

```
>>> L + [1,2,3]
[1, 8, -4, 38, 8, 8, 4, 18, 1, 2, 3]
>>> L
[1, 8, -4, 38, 8, 8, 4, 18]
```

La fonction `len` retourne la longueur d'une liste:

```
>>> len(L)
9
```

Les fonctions `min` et `max` retournent la valeur minimum et maximum d'une liste:

```
>>> min(L)
-4
>>> max(L)
38
```

Pour savoir si une `valeur` est dans une `liste`, on utilise `valeur in liste`. Cela retourne un booléen. Par exemple:

```
>>> 77 in L
False
>>> 38 in L
True
```

La méthode `.count()` permet de compter le nombre d'objets de la liste ayant une certaine valeur:

```
>>> L.count(38)
1
>>> L.count(8)
3
>>> L.count(77)
0
```

La méthode `.index()` retourne la position (ou indice) où un élément se retrouve dans la liste:

```
>>> L
[1, 8, -4, 38, 8, 8, 4, 18]
>>> L.index(38)
3
```

12.2 Modification de listes

Pour ajouter un élément à la liste, on utilise la méthode `.append()` qui ajoute un élément à la fin de la liste:

```
>>> L
[1, 8, -4, 38, 8, 8, 4, 18]
>>> L.append(15)
```

```
>>> L
[1, 8, -4, 38, 8, 8, 4, 18, 15]
```

La méthode `.remove()` permet d'enlever un élément de la liste:

```
>>> L.remove(4)
>>> L
[1, 8, -4, 38, 8, 8, 18, 15]
```

Si l'élément est là plus d'une fois, seule la première occurence de celle-ci est retirée:

```
>>> L.remove(8)
>>> L
[1, -4, 38, 8, 8, 18, 15]
```

La méthode `.reverse()` permet d'inverser l'ordre d'une liste:

```
>>> L.reverse()
>>> L
[15, 18, 8, 8, 38, -4, 1]
```

La méthode `.sort()` permet de trier les éléments d'une liste en ordre croissant:

```
>>> L.sort()
>>> L
[-4, 1, 8, 8, 15, 18, 38]
```

12.3 La fonction `range`

La fonction `range(n)` permet de créer la liste des entiers de `0` à `n-1`:

```
>>> range(15)                  # Python 2
>>> list(range(15))            # Python 3
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Avec deux arguments, la fonction `range(a, b)` crée la liste des entiers de `a` à `b-1`:

```
>>> range(3, 15)
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Avec trois arguments, la fonction `range(a, b, saut)` crée la liste des entiers de `a` à `b-1` par saut de `saut`:

```
>>> range(3,40,4)
[3, 7, 11, 15, 19, 23, 27, 31, 35, 39]
```

12.4 Compréhension de listes

Soit la liste des entiers de zéro à neuf:

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

Les *compréhensions de listes* (list comprehensions en anglais, certains auteurs écrivent *intentions de listes* en français) permettent de créer des listes facilement en une ligne. La syntaxe ressemble à la syntaxe qui permet de décrire un ensemble mathématique: `[expression_de_i for i in liste]`. Par exemple, l'ensemble des cubes des valeurs de la liste `L` s'écrit:

```
>>> [i**3 for i in L]
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

L'ensemble des cubes des valeurs impaires de la liste `L` se fait en ajoutant une condition à la fin de l'expression:

```
>>> [i**3 for i in L if i%2 == 1]
[1, 27, 125, 343, 729]
```

13 Boucle `for`

Dans ce chapitre et les suivants, nous traitons de la programmation en Python. Les notes ici présentent les grandes lignes et les éléments principaux de ce sujet. Le lecteur désirant en savoir plus sera invité à consulter les chapitres 1 à 7 du livre en français de G. Swinnen *Apprendre à programmer avec Python 3* [Swinnen], le syllabus du cours de programmation de Thierry Massart [Massart] ou encore les chapitre 1 à 11 du livre en anglais de Wentworth et al. *How to Think Like a Computer Scientist - Learning with Python* [Thinklike].

Une boucle permet de faire des tâches répétitives sur un ordinateur avec un moindre effort.


```
>>> for a in range(9):
...     print("I will not do anything bad ever again.")
I will not do anything bad ever again.
I will not do anything bad ever again.
I will not do anything bad ever again.
I will not do anything bad ever again.
I will not do anything bad ever again.
I will not do anything bad ever again.
I will not do anything bad ever again.
I will not do anything bad ever again.
I will not do anything bad ever again.
```

13.1 La boucle `for`

La boucle for permet aussi de parcourir les éléments d'une liste, une chaîne de caractères ou en général de tout objet itérable:

```
>>> for a in [1,2,3,4]:
...     print(a + 100)
101
102
103
104
>>> for a in 'bonjour':
...     print('A ' + a + ' Z')
A b Z
A o Z
A n Z
A j Z
A o Z
A u Z
A r Z
```

En Python, une boucle `for` est identifiée par une ligne d'en-tête commançant par `for` se terminant par un deux-points `:` et avec la syntaxe `for TRUC in MACHIN:`. La convention est de toujours utiliser 4 espaces pour indenter les lignes du bloc d'instructions qui appartient à la boucle:

```
for i in liste:                        # ligne d'en-tête
    <ligne 1 du bloc d'instruction>
    <ligne 2 du bloc d'instruction>
    ...
    <ligne n du bloc d'instruction>
<ligne exécutée après la boucle>
```

Le bloc d'instructions est exécuté autant de fois qu'il y a d'éléments dans la liste. Le bloc d'instruction est exécuté une fois pour chaque valeur de la variable `i` dans la liste.

13.2 Un exemple de boucle `for` avec Sympy

Supposons que l'on désire factoriser le polynôme `x**k-1` pour toutes les valeurs de `k=1..9`. En SymPy, il est possible d'écrire onze fois le même calcul où on change la valeur de l'exposant `k` à chaque fois:

```
>>> from sympy import factor
>>> from sympy.abc import x
>>> factor(x**1-1)
x - 1
>>> factor(x**2-1)
(x - 1)*(x + 1)
>>> factor(x**3-1)
(x - 1)*(x**2 + x + 1)
>>> factor(x**4-1)
(x - 1)*(x + 1)*(x**2 + 1)
>>> factor(x**5-1)
(x - 1)*(x**4 + x**3 + x**2 + x + 1)
>>> factor(x**6-1)
```

```
(x - 1)*(x + 1)*(x**2 - x + 1)*(x**2 + x + 1)
>>> factor(x**7-1)
(x - 1)*(x**6 + x**5 + x**4 + x**3 + x**2 + x + 1)
>>> factor(x**8-1)
(x - 1)*(x + 1)*(x**2 + 1)*(x**4 + 1)
>>> factor(x**9-1)
(x - 1)*(x**2 + x + 1)*(x**6 + x**3 + 1)
```

La boucle `for` permet répéter une action pour toutes les valeurs d'une liste. En utilisant une boucle `for`, l'exemple ci-haut peut se réécrire plus facilement:

```
>>> for k in range(1,12):
...     print(factor(x**k-1))
x - 1
(x - 1)*(x + 1)
(x - 1)*(x**2 + x + 1)
(x - 1)*(x + 1)*(x**2 + 1)
(x - 1)*(x**4 + x**3 + x**2 + x + 1)
(x - 1)*(x + 1)*(x**2 - x + 1)*(x**2 + x + 1)
(x - 1)*(x**6 + x**5 + x**4 + x**3 + x**2 + x + 1)
(x - 1)*(x + 1)*(x**2 + 1)*(x**4 + 1)
(x - 1)*(x**2 + x + 1)*(x**6 + x**3 + 1)
```

Pour différencier les lignes, il est possible d'afficher plus d'informations:

```
>>> from sympy import Eq
>>> for k in range(2, 10):
...     expr = x**k-1
...     eq = Eq(expr, factor(expr))
...     print(eq)
x**2 - 1 == (x - 1)*(x + 1)
x**3 - 1 == (x - 1)*(x**2 + x + 1)
x**4 - 1 == (x - 1)*(x + 1)*(x**2 + 1)
x**5 - 1 == (x - 1)*(x**4 + x**3 + x**2 + x + 1)
x**6 - 1 == (x - 1)*(x + 1)*(x**2 - x + 1)*(x**2 + x + 1)
x**7 - 1 == (x - 1)*(x**6 + x**5 + x**4 + x**3 + x**2 + x + 1)
x**8 - 1 == (x - 1)*(x + 1)*(x**2 + 1)*(x**4 + 1)
x**9 - 1 == (x - 1)*(x**2 + x + 1)*(x**6 + x**3 + 1)
```

13.3 Affectation d'une variable

Pour affecter une valeur dans une variable, on se rappelle que cela se fait en Python comme en C ou C++ ou Java avec la syntaxe:

```
>>> a = 5
```

La syntaxe `a == 5` est réservée pour le test d'égalité.

13.4 Mise à jour d'une variable

Quand une instruction d'affectation est exécutée, l'expression de droite (à savoir l'expression qui vient après le signe = d'affectation) est évaluée en premier. Cela produit une valeur. Ensuite, l'assignation est faite, de sorte que la variable sur le côté gauche se réfère maintenant à la nouvelle valeur.

L'une des formes les plus courantes de l'affectation est une mise à jour, lorsque la nouvelle valeur de la variable dépend de son ancienne valeur:

```
>>> n = 5
>>> n = 3 * n + 1
```

Ligne 2 signifie obtenir la valeur courante de n, la multiplier par trois et ajouter un, et affecter la réponse à n. Donc, après avoir exécuté les deux lignes ci-dessus, n va pointer / se référer à l'entier 16.

Si vous essayez d'obtenir la valeur d'une variable qui n'a jamais été attribuée, vous obtenez une erreur:

```
>>> W = x + 1
Traceback (most recent call last):
...
NameError: name 'x' is not defined
```

Avant de pouvoir mettre à jour une variable, vous devez l'initialiser à une valeur de départ, habituellement avec une valeur simple:

```
sous_total = 0
sous_total = sous_total + 1
```

La mise à jour d'une variable en lui ajoutant 1 à celle-ci est très commune. On appelle cela un **incrément** de la variable; soustraire 1 est appelé un **décrément**.

Le code `sous_total = sous_total + 1` calcule le résultat de la partie droite dans un nouvel espace en mémoire et ensuite cette nouvelle valeur est affectée à la variable `sous_total`. Une façon plus efficace d'incrémenter une variable est de la modifier sans avoir à garder en mémoire un résultat partiel. En Python (comme en C), on peut incrémenter une variable avec l'opérateur +=. Donc, il suffit d'écrire:

```
sous_total += 1
```

13.5 Quelques exemples

L'exemple suivant illustre comment calculer la somme des éléments d'une liste en utilisant une variable s initialisée à zéro avant la boucle:

```
>>> L = [134, 13614, 73467, 1451, 134, 88]
>>> s = 0
>>> for a in L:
...     s = s + a
>>> s
88888
```

On écrit la même chose en utilisant le signe += pour incrémenter la variable s:

```
>>> s = 0
>>> for a in L:
...     s += a
>>> s
88888
```

On vérifie que le calcul est bon:

```
>>> sum(L)
88888
```

L'exemple suivant double chacune des lettres d'une chaîne de caractères:

```
>>> s = 'gaston'
>>> t = ''
>>> for lettre in s:
...     t += lettre + lettre
...
>>> t
'ggaassttoonn'
```

Lorsque la variable de la boucle n'est pas utilisée dans le bloc d'instruction la convention est d'utiliser la barre de soulignement (_) pour l'indiquer. Ici, on calcule les puissances du nombre 3. On remarque que l'expression d'assignation `k *= 3` est équivalente à `k = k * 3`:

```
>>> k = 1
>>> for _ in range(10):
...     k *= 3
...     print k
...
3
9
27
81
243
729
2187
6561
19683
59049
```

14 Conditions `if`

Les conditions sont, avec les boucles, les éléments les plus importants de la programmation. Elles permettent de formaliser la prise de décisions selon l'information disposée. Faisons un exemple, supposons que nous sommes sur l'autoroute en voiture et que l'on peut faire le plein à la prochaine sortie. Est-ce qu'on prend la sortie ou est-ce qu'on reste sur l'autoroute? Supposons que la variable `reservoir` prend une valeur entre 0 et 1 et indique le pourcentage d'espace occupé par l'essence dans le réservoir. Une première façon de prendre la décision pourrait s'écrire en Python:

```
if reservoir < 0.15:
    print('Prendre la sortie')
else:
    print('Rester sur l'autoroute')
```

Bien sûr, on pourrait aussi améliorer la prise de décision en considérant la distance de la prochaine station d'essense et comparer avec la distance pouvant être parcourue avec ce qui reste d'essence, etc. Mais, pour le moment, l'important est de comprendre comment on écrit une condition simple en Python.

14.1 La forme `if - else`

La forme générale est la suivante:

```
if <condition>:
    <code exécuté si la condition est satisfaite, i.e., True>
else:
    <code exécuté si la condition n'est pas satisfaite, i.e., False>
```

La `<condition>` est une expression qui retourne une valeur booléenne `True` ou `False`. Elle s'écrit avec un `if` et la ligne doit se terminer avec les deux points `:`. Si la condition est vérifiée, c'est-à-dire si la condition est évaluée à `True`, alors le code indenté de 4 espaces sous la ligne `if` est exécuté. Sinon, c'est-à-dire si la condition est évaluée à `False`, alors c'est le code indenté de 4 espaces sous la ligne `else:` qui est exécuté.

14.2 La forme `if - elif - else`

Parfois, il y a plusieurs cas à tester. Il est possible d'emboîter les conditions:

```
if reservoir == 0:
    print("Panne d'essence")
else:
    if 0 < reservoir < 0.15:
        print('Prendre la sortie')
    else:
        print('Rester sur l'autoroute')
```

Mais en Python, ce qui n'est pas le cas dans tous les langages de programmation, le `else if` s'écrit de façon abrégée `elif` ce qui permet de limiter l'indentation du code. On préférera donc écrire en Python le code ci-haut de la façon suivante:

```
if reservoir == 0:
    print("Panne d'essence")
elif 0 < reservoir < 0.15:
    print('Prendre la sortie')
else:
    print('Rester sur l'autoroute')
```

14.3 La forme `if` seul

La ligne `elif` ou la ligne `else` n'est pas obligatoire car parfois on ne veut rien faire si la condition n'est pas satisfaite. Dans ce cas, on écrit simplement:

```
if reservoir == 0:
    print("Panne d'essence")
```

14.4 La forme `if - elif - ... - elif - else`

Il peut y avoir plusieurs lignes de `elif`:

```
if temperature < 0:
    print("L'eau est solide")
elif temperature == 0:
```

```
    print("L'eau est en transition de phase solide-liquide")
elif temperature < 100:
    print("L'eau est liquide")
elif temperature == 100:
    print("L'eau est en transition de phase liquide-gaz")
else:
    print("L'eau est un gaz")
```

Ci-haut, une seule des lignes `print` sera exécutée: celle qui est sous la première condition est qui satisfaite. Attention, ici comme les conditions ne sont pas mutuellement exclusives, l'ordre des conditions est important.

14.5 Syntaxe compacte d'une assignation conditionnelle

Parfois, on veut assigner à une variable une valeur qui dépend d'une condition. Par exemple, on veut calculer le minimum de deux valeurs. On peut utiliser une condition pour faire cette assignation:

```
>>> x,y = 10, 6
>>> if x < y:
...     minimum = x
... else:
...     minimum = y
...
>>> minimum
6
```

Python offre une syntaxe abrégée (inspirée du C) pour faire ceci:

```
>>> minimum = x if x < y else y
>>> minimum
6
```

15 Fonctions `def`

Une fonction rassemble un ensemble d'instructions qui permettent d'atteindre un certain objectif commun. Les fonctions permettent de séparer un programme en morceaux qui correspondent à la façon dont on pense à la résolution d'un problème.

La syntaxe pour la définition d'une fonction est:

```
def FONCTION( PARAMETRES ):
    INSTRUCTIONS
```

La ligne d'en-tête commence avec `def` et se termine par un deux-points. Le choix du nom de la fonction suit exactement les mêmes règles que pour le choix du nom d'une variable. Un bloc constitué d'une ou plusieurs instructions Python, chacune indentée du même nombre d'espace (la convention est d'utiliser 4 espaces) par rapport à la ligne d'en-tête. Nous avons déjà vu la boucle `for` qui suit ce modèle.

Le nom de la fonction est suivi par certains paramètres entre parenthèses. La liste des paramètres peut être vide, ou il peut contenir un certain nombre de paramètres séparés les uns des autres par des virgules. Dans les deux cas, les parenthèses sont nécessaires. Les paramètres spécifient les informations, le cas échéant, que nous devons fournir pour pouvoir utiliser la nouvelle fonction.

La ou les valeurs de retour d'une fonction sont retournées avec la commande `return`. Par exemple, la fonction qui retourne la somme de deux valeurs s'écrit:

```
def somme(a, b):
    return a + b

>>> somme(4,7)
11
```

La fonction qui calcule le volume d'un parallépipède rectangle s'écrit:

```
>>> def volume(largeur, hauteur, profondeur):
...     return largeur * hauteur * profondeur
...
>>> v = volume(2,3,4)
>>> v
24
```

On peut rassemble le code sur la température de l'eau que l'on a écrit plus au sein d'une fonction `etat_de_leau` qui dépend du paramètre `temperature`:

```
def etat_de_leau(temperature):
    if temperature < 0:
        print("L'eau est solide")
    elif temperature == 0:
        print("L'eau est en transition de phase solide-liquide")
    elif temperature < 100:
        print("L'eau est liquide")
    elif temperature == 100:
        print("L'eau est en transition de phase liquide-gaz")
    else:
        print("L'eau est un gaz")
```

Cette fonction permet de tester le code sur la température de l'eau plus facilement:

```
>>> etat_de_leau(23)
L'eau est liquide
>>> etat_de_leau(-23)
L'eau est solide
>>> etat_de_leau(0)
L'eau est en transition de phase solide-liquide
>>> etat_de_leau(0.1)
L'eau est liquide
>>> etat_de_leau(102)
L'eau est un gaz
```

16 Boucle `while`

Parfois, on ne sait pas à l'avance combien de fois on voudra exécuter un bloc d'instructions. Dans ce cas, il vaut mieux utiliser une boucle `while` dont la syntaxe est:

```
while CONDITION:
    INSTRUCTION 1
    INSTRUCTION 2
```

```
    ...
    INSTRUCTION n
```

Le bloc d'instruction est exécuté (au complet) tant que la condition est satisfaite. La condition est testée avant l'exécution du bloc, mais pas pendant. C'est donc toutes les instructions du bloc qui sont exécutées si la condition est vraie. Par exemple, on peut afficher les puissances de 5 inférieures à un million avec une boucle `while`:

```
>>> a = 1
>>> while a < 1000000:
...     print a
...     a = a * 5
...
1
5
25
125
625
3125
15625
78125
390625
```

L'exemple suivant est un autre exemple typique de la boucle tant que. Il consiste à rechercher, pour un nombre `x>=1`, l'unique valeur entière `n` vérifiant `2**(n-1) < x < 2**n`, c'est-à-dire le plus petit entier vérifiant `x < 2**n`.

```
>>> x = 10**4
>>> u = 1
>>> n = 0
>>> while u <= x:
...     n = n + 1
...     u = 2 * u
>>> n
14
```

On vérifie bien que `2**13 < 10**4 < 2**14`:

```
>>> 2 ** 13
8192
>>> 2 ** 14
16384
>>> 2**13 < 10**4 < 2**14
True
```

16.1 Interruptions de boucles avec `break` et `continue`

La commande `break` permet d'interrompre une boucle `for` ou `while` en cours:

```
>>> for i in range(10):
...     if i == 5:
...         break
...     print(i)
```

```
...
0
1
2
3
4
```

On remarque que les valeurs plus grandes que 4 n'ont pas été imprimées par la fonction `print`.

La commande `continue` permet de continuer le parcours d'une boucle à la valeur suivante:

```
>>> for i in range(10):
...     if i == 5:
...         continue
...     print(i)
...
0
1
2
3
4
6
7
8
9
```

On remarque que la valeur 5 n'a pas été imprimée par la fonction `print`.

> ***Note***
>
> Certains auteurs recommandent d'éviter l'utilisation des intructions `continue` et des `break`, car elles sont évitables et leur utilisation produit des programmes moins bien structurés.

17 Exemples (`def` + `while` + `for` + `if`)

On a vu dans les chapitres précédents comment définir des fonctions avec `def`, des boucles avec `while` et `for` et des tests avec `if` ainsi que quelques exemples sur chaque notion mais indépendants des autres. Très souvent en programmation, on a besoin d'utiliser plus tous ces outils à la fois. C'est leur utilisation simultanée qui permet de résoudre des problèmes très divers et de les exprimer en quelques lignes de code.

Dans ce chapitre, nous allons voir quelques exemples qui utilisent les fonctions, les boucles et les conditions dans un même programme.

17.1 Conjecture de Syracuse

La *suite de Syracuse* une suite d'entiers naturels définie de la manière suivante. On part d'un nombre entier plus grand que zéro ; s'il est pair, on le divise par 2 ; s'il est impair, on le multiplie par 3 et on ajoute 1. En répétant l'opération, on obtient une suite d'entiers positifs dont chacun ne dépend que de son prédécesseur. Par exemple, la suite de Syracuse du nombre 23 est:

23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...

Après que le nombre 1 a été atteint, la suite des valeurs (1, 4, 2, 1, 4, 2, ...) se répète indéfiniment en un cycle de longueur 3, appelé cycle trivial.

La conjecture de Syracuse est l'hypothèse selon laquelle la suite de Syracuse de n'importe quel entier strictement positif atteint 1. En dépit de la simplicité de son énoncé, cette conjecture défie depuis de nombreuses années les mathématiciens. Paul Erdos a dit à propos de la conjecture de Syracuse : "les mathématiques ne sont pas encore prêtes pour de tels problèmes".

```
def syracuse(n):
    while n != 1:
        print(n, end=' ')
        if n % 2 == 0:
            n = n//2
        else:
            n = 3*n+1
```

```
>>> syracuse(23)
23 70 35 106 53 160 80 40 20 10 5 16 8 4 2
>>> syracuse(245)
245 736 368 184 92 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2
>>> syracuse(245154)
245154 122577 367732 183866 91933 275800 137900 68950 34475 103426 51713
155140 77570 38785 116356 58178 29089 87268 43634 21817 65452 32726 16363
49090 24545 73636 36818 18409 55228 27614 13807 41422 20711 62134 31067
93202 46601 139804 69902 34951 104854 52427 157282 78641 235924 117962 58981
176944 88472 44236 22118 11059 33178 16589 49768 24884 12442 6221 18664 9332
4666 2333 7000 3500 1750 875 2626 1313 3940 1970 985 2956 1478 739 2218 1109
3328 1664 832 416 208 104 52 26 13 40 20 10 5 16 8 4 2
```

Pouvez-vous trouver un nombre n tel que la suite de Syracuse n'atteint pas le cycle 4-2-1?

17.2 Énumérer les diviseurs d'un nombre entier

Une fonction qui retourne la liste des diviseurs d'un nombre entiers peut s'écrire comme ceci en utilisant une boucle `for` et un test `if`:

```
def diviseurs(n):
    L = []
    for i in range(1, n+1):
        if n % i == 0:
            L.append(i)
    return L
```

On vérifie que la fonction marche bien:

```
>>> diviseurs(12)
[1, 2, 3, 4, 6, 12]
>>> diviseurs(13)
[1, 13]
>>> diviseurs(15)
[1, 3, 5, 15]
>>> diviseurs(24)
[1, 2, 3, 4, 6, 8, 12, 24]
```

17.3 Tester si un nombre est premier

Une fonction peut en utiliser une autre. Par exemple, en utilisant la fonction `diviseur` que l'on a définit plus haut, on peut tester si un nombre est premier:

```
def est_premier_1(n):
    L = diviseurs(n)
    return len(L) == 2
```

```
>>> est_premier_1(12)
False
>>> est_premier_1(13)
True
>>> [n for n in range(20) if est_premier_1(n)]
[2, 3, 5, 7, 11, 13, 17, 19]
```

On pourrait faire plus efficace, car il suffit de vérifier la non-existence de diviseurs inférieurs à la racine carrée de n.

```
from math import sqrt
def est_premier(n):
    sq = int(sqrt(n))
    for i in range(2, sq):
        if n % i == 0:
            return False
    return True
```

En utilisant cette fonciton, on trouve que la liste des premiers nombres premiers inférieurs à 20 est:

```
>>> [n for n in range(20) if est_premier(n)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 15, 17, 19]
```

Le résulat est erroné! Pourquoi?

La fonction `est_premier(8)` retourne `True` en ce moment, car la racine carrée de 8 vaut `2.828` et donc `sq=int(2.828)` est égal à `2` et la boucle ne teste pas la valeur `i=2`, car `range(2,2)` retourne une liste vide. On peut corriger de la façon suivante en ajoutant un `+1` au bon endroit:

```
from math import sqrt
def est_premier(n):
    sq = int(sqrt(n))
    for i in range(2, sq+1):
        if n % i == 0:
            return False
    return True
```

On vérifie que la fonction retourne bien que 4 et 8 ne sont pas des nombres premiers:

```
>>> [n for n in range(20) if est_premier(n)]
[0, 1, 2, 3, 5, 7, 11, 13, 17, 19]
```

Mais il y a encore une erreur, car 0 et 1 ne devraient pas faire partie de la liste. Une solution est de traiter ces deux cas de base à part:

```
from math import sqrt
def est_premier(n):
    if n == 0 or n == 1:
        return False
    sq = int(sqrt(n))
    for i in range(2, sq+1):
        if n % i == 0:
            return False
    return True
```

On vérifie que tout marche bien maintenant:

```
>>> [n for n in range(50) if est_premier(n)]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

18 Autres structures de données

18.1 Tuples (type `tuple`)

En Python, les tuples (ou n-uplets) sont créées avec l'utilisation des parenthèses et on peut mettre n'importe quel objet dans un tuple:

```
>>> t = (45, 'bonjour', 6.7, [3])
>>> t
(45, 'bonjour', 6.7, [3])
```

Comme pour les listes et les chaînes de caractères, on peut accéder aux éléments avec les crochets et zéro dénote la première position:

```
>>> t[0]
45
>>> t[1]
'bonjour'
```

On vérifie le type de cet objet:

```
>>> type(t)
<type 'tuple'>
```

Un tuple joue le même rôle qu'une liste à la différence principale qu'on ne peut pas modifier un tuple. On ne peut donc pas ajouter ou supprimer des objets d'un tuple.

La fonction `tuple` permet de transformer un objet en tuple pourvu qu'il soit itérable:

```
>>> tuple('bonjour')
('b', 'o', 'n', 'j', 'o', 'u', 'r')
>>> tuple([1,2,3])
(1, 2, 3)
```

18.2 Emballage et déballage d'un tuple

Les tuples peuvent être créés comme emballage de valeurs:

```
>>> b = ('Bob', 23, 'math')      # emballage d'un tuple
>>> b
(u'Bob', 23, u'math')
```

Mais, il peuvent aussi être déballés directement en faisant comme suit:

```
>>> (nom, age, etude) = b        # déballage d'un tuple
>>> nom
u'Bob'
>>> age
23
>>> etude
u'math'
```

18.3 Dictionnaires (type `dict`)

Les listes et tuples ont cela de contraignants que les positions sont des nombres de 0 à n-1 où n est la longueur de la liste. Parfois, il est pratique que les positions prennent d'autres valeurs ou d'autres types.

En Python, les dictionnaire sont créées avec l'utilisation des accolades avec la syntaxe `{cle1:valeur1, cle2:valeur2, cle3:valeur3}`. Par exemple:

```
>>> d = {'namur':813248, 'liege':441432, 'anvers':978756}
>>> d
{'liege': 441432, 'namur': 813248, 'anvers': 978756}
```

est un dictionnaire qui associe des noms de villes avec des nombres qui peuvent représenter le nombre d'habitants:

```
>>> type(d)
<type 'dict'>
```

On peut accéder à la valeur associée à une clé en utilisant les crochets:

```
>>> d['liege']
441432
```

Tenter d'accéder à une clée inexistante retourne une erreur:

```
>>> d['bruxelles']
Traceback (most recent call last):
...
KeyError: 'bruxelles'
```

Toutefois, on peut ajouter les données pour la ville de Bruxelles en faisant:

```
>>> d['bruxelles'] = 5000000
>>> d
{'bruxelles': 5000000, 'liege': 441432, 'namur': 813248, 'anvers': 978756}
```

La fonction `len` retourne la taille du dictionnaire:

```
>>> len(d)
4
```

Les méthodes `.keys()` et `.values()` retourne respectivement les clés et les valeurs d'un dictionnaire sous forme de liste:

```
>>> d.keys()
['bruxelles', 'liege', 'namur', 'anvers']
>>> d.values()
[5000000, 441432, 813248, 978756]
```

Finalement, la méthode `.items()` retourne la liste des paires clé-valeur d'un dictionnaire:

```
>>> d.items()
[('bruxelles', 5000000), ('liege', 441432), ('namur', 813248), ('anvers', 978756)]
```

En SymPy, on se rappelle que certaines fonctions retournent des dictionnaires telles que la fonction `factorint`:

```
>>> from sympy import factorint
>>> factorint(240)
{2: 4, 3: 1, 5: 1}
```

Les clés d'un dictionnaire doivent être des objets non modifiables (techniquement, des objets qui définissent une fonction de hachage `hash`). Comme les listes sont modifiables, une liste ne peut pas jouer le rôle d'une clé d'un dictionnaire. Si on le fait, on obtient l'erreur suivante:

```
>>> d = dict()
>>> cle = [2,3,4]
>>> d[cle] = 'valeur'
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

Comme les listes sont modifiables, elle ne sont pas hachable d'où l'erreur obtenue. Par contre, on peut utiliser un tuple comme clé d'un dictionnaire:

```
>>> cle = (2,3,4)
>>> d[cle] = 'valeur'
>>> d
{(2, 3, 4): 'valeur'}
```

18.4 Ensembles (type `set`)

Les listes peuvent contenir plusieurs fois le même objet:

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

En Python, le type `set` permet de créer un ensemble au sens mathématique où chaque élément apparaît au plus une fois:

```
>>> set('gauffredeliege')
set(['a', 'e', 'd', 'g', 'f', 'i', 'l', 'r', 'u'])
```

```
>>> set([1,2,2,3,3,3,4,4,4,4])
set([1, 2, 3, 4])
```

La méthode `.add()` permet d'ajouter un élément à l'ensemble:

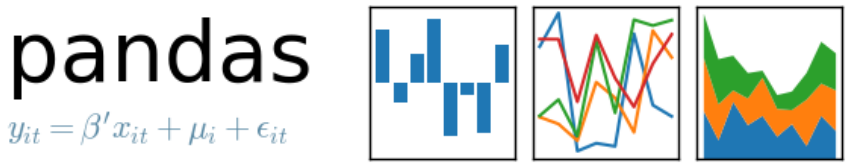
```
>>> s = set([1,2,3,4])
>>> s.add('bonjour')
>>> s
set([1, 2, 3, 4, 'bonjour'])
```

Comme pour les clés d'un dictionnaire, les éléments d'un ensemble doivent être hachables (non modifiables). Par exemple, on ne peut pas ajouter une liste à un ensemble, mais on peut ajouter un tuple:

```
>>> s.add([1,2,3])
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
>>> s.add((1,2,3))
>>> s
set([1, 2, 3, 4, (1, 2, 3), u'bonjour'])
```

19 Tableaux et analyse de données avec Pandas

Les données massives jouent et continueront de jouer un rôle important dans la société du 21e siècle. Dans ce chapitre, nous ferons une introduction à une librairie de l'environnement Python qui permet de représenter et analyser des données. Cette librairie s'appelle pandas, contraction des termes anglais "panel" et "data". Dans pandas, un tableau 3-dimensionnel de données est appelé un "panel". La librairie pandas joue le même rôle qu'un tableur comme Microsoft Excel, LibreOffice calc ou celui qu'on retrouve dans GeoGebra.



Les principales structures de données de pandas sont les `Series` (pour stocker des données selon une dimension) et les `DataFrame` (pour stocker des données selon 2 dimensions - lignes et colonnes). On peut aussi représenter des données selon trois dimensions ou plus avec `Panel` et `Panel4D`.

Dans ce chapitre nous décrivons les tableaux de données à une et deux dimensions. Nous verrons comment faire des calculs statistiques et créer des graphiques à partir de celles-ci. Nous verrons comment importer et exporter des données. Finalement, nous ferons un exemple basé sur le site de données de la Belgique: http://data.gov.be/. On trouvera plus d'informations dans la documentation en ligne de pandas incluant une introduction en 10 minutes, les notions de base et quelques tutoriels.

19.1 Tableau unidimensionnel de données

En utilisant sympy, construisons une liste de 0 et de 1 telle qu'un 1 est à la position `i` si et seulement si `i` est un nombre premier:

```
>>> from sympy import isprime
>>> L = [isprime(i) for i in range(15)]
>>> L
[False, False, True, True, False, True, False, True, False, False, False,
 True, False, True, False]
```

La librairie pandas permet de représenter les tableaux unidimensionnels de données appelés *séries*. Faisons un premier exemple. La liste Python ci-haut peut être transformée en une série de pandas en faisant comme suit:

```
>>> from pandas import Series
>>> s = Series(L)
>>> s
0    False
1    False
2     True
3     True
4    False
5     True
6    False
7     True
8    False
9    False
10   False
11    True
12   False
13    True
14   False
dtype: bool
```

Par défaut, les indices sont les nombres de `0` à `n-1` où `n` est la taille de la liste. On peut accéder aux éléments de la série de la même façon qu'on le fait pour les éléments d'une liste:

```
>>> s[0]
False
>>> s[7]
True
```

19.2 Afficher quelques statistiques

L'intérêt des séries de pandas par rapport aux listes Python de base est qu'un grand nombres de fonctions utiles sont disponibles sur les séries de pandas et qui retournent souvent d'autres séries. Par exemple, on peut obtenir une brève description statistique des éléments d'une série avec la méthode `describe()`:

```
>>> s.describe()
count        15
unique        2
top       False
freq          9
dtype: object
```

Ci-haut, cela nous indique qu'il y a deux valeurs distinctes dans la série et que `False` est la plus fréquence avec 9 apparitions sur 15. En effet, il y 6 nombres premiers inférieurs à 15.

On peut obtenir la séries des sommes cumulées d'une série avec la méthode `cumsum()`. Ici `False` vaut zéro et `True` vaut `1`:

```
>>> s.cumsum()
0     0
1     0
2     1
3     2
4     2
5     3
6     3
7     4
8     4
9     4
10    4
11    5
12    5
13    6
14    6
dtype: int64
```

Il suffit de faire `s.TOUCHE_TABULATION` pour voir les nombreuses possibilités offertes par pandas. On y reviendra.

19.3 Opérations sur une série

Les opérations arithmétiques sont définies sur les séries. Elle sont appliquées sur chaque terme:

```
>>> t = s.cumsum()
>>> t * 1000 + 43
0       43
1       43
2     1043
3     2043
4     2043
5     3043
6     3043
7     4043
8     4043
9     4043
10    4043
11    5043
12    5043
13    6043
14    6043
dtype: int64
```

On peut aussi appliquer une fonction aux éléments d'une série avec la méthode `apply`:

```
>>> def carre_plus_trois(x):
...     return x**2 + 3
>>> t.apply(carre_plus_trois)
```

```
0      3
1      3
2      4
3      7
4      7
5     12
6     12
7     19
8     19
9     19
10    19
11    28
12    28
13    39
14    39
dtype: int64
```

19.4 Concaténation de deux séries

Avec pandas, il est possible de construire un tableau comportant plus d'une colonne. Par exemple, les nombres premiers dans la première colonne et la somme cumulée dans la deuxième. Une première façon est avec la fonction `concat` qui concatène deux séries:

```
>>> from pandas import concat
>>> concat([s, s.cumsum()])
0     0
1     0
2     1
3     1
4     0
5     1
6     0
7     1
8     0
9     0
10    0
11    1
12    0
13    1
14    0
0     0
1     0
2     1
3     2
4     2
5     3
6     3
7     4
8     4
9     4
10    4
11    5
12    5
13    6
```

```
14    6
dtype: int64
```

La concaténation a été faite une en-dessous de l'autre et cela a aussi eu pour effet de transformer les valeurs booléennes en nombres entiers, car les données d'une même colonne doivent avoir le même type. Ce n'est pas exactement ce qu'on voulait. Pour spécifier que la concaténation doit être faite en colonnes, il faut spécifier dans quelle direction (axe) ou veut concaténer les données. On donne alors une valeur `1` à l'argument `axis` plutôt que `0` (la valeur par défaut) pour obtenir ce que l'on veut:

```
>>> concat([s, s.cumsum()], axis=1)
       0  1
0   False  0
1   False  0
2    True  1
3    True  2
4   False  2
5    True  3
6   False  3
7    True  4
8   False  4
9   False  4
10  False  4
11   True  5
12  False  5
13   True  6
14  False  6
```

Pour donner des titres plus parlant aux colonnes, il s'agit de spécifier une liste de titres via l'argument `keys`. Comme le nombre de nombres entiers inférieur à `x` est souvent dénoté `pi(x)`, on utilise `'pi_x'` pour le nom de la deuxième colonne:

```
>>> keys = ['isprime', 'pi_x']
>>> df = concat([s, s.cumsum()], axis=1, keys=keys)
>>> df
   isprime  pi_x
0    False     0
1    False     0
2     True     1
3     True     2
4    False     2
5     True     3
6    False     3
7     True     4
8    False     4
9    False     4
10   False     4
11    True     5
12   False     5
13    True     6
14   False     6
```

Le type du tableau ci-haut est `DataFrame` pour tableau de données:

```
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

19.5 Tableau 2-dimensionnel de données

Une autre façon de créer le même tableau est en utilisant la fonction `DataFrame` directement:

```
>>> from pandas import DataFrame
```

D'abord, on calcule en Python la liste des sommes cumulées de la liste `L`:

```
>>> L = [isprime(i) for i in range(15)]
>>> L_cumsum = [sum(L[:i]) for i in range(1,len(L)+1)]
>>> L_cumsum
[0, 0, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5, 6, 6]
```

On crée un dictionnaire qui associe des noms de colonnes à des valeurs:

```
>>> d = {'isprime':L, 'pi_x':L_cumsum}
>>> d
{'isprime': [False, False, True, True, False, True, False, True,
             False, False, False, True, False, True, False],
 'pi_x': [0, 0, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5, 6, 6]}
```

On crée un objet de type `DataFrame` à partir de ce dictionnaire:

```
>>> df = DataFrame(d)
>>> df
   isprime  pi_x
0    False     0
1    False     0
2     True     1
3     True     2
4    False     2
5     True     3
6    False     3
7     True     4
8    False     4
9    False     4
10   False     4
11    True     5
12   False     5
13    True     6
14   False     6
```

Comme pour les séries, on peut obtenir les statistiques simples pour les données de chaque colonne d'un tableau de données avec la méthode `describe()`:

```
>>> df.describe()
            pi_x
count  15.000000
mean    3.266667
```

```
std     1.944467
min     0.000000
25%     2.000000
50%     4.000000
75%     4.500000
max     6.000000
```

Il est aussi possible de créer des tableaux de données en dimensions supérieures, mais cela dépasse le cadre de ce cours:

```
>>> from pandas import Panel,Panel4D
```

19.6 Accéder à une colonne d'un tableau

Le nom des colonnes peut être utilisé pour accéder aux colonnes d'un tableau de la façon suivante sans parenthèse:

```
>>> df.pi_x
0     0
1     0
2     1
3     2
4     2
5     3
6     3
7     4
8     4
9     4
10    4
11    5
12    5
13    6
14    6
Name: pi_x, dtype: int64
```

Comme pour un dictionnaire, on peut aussi accéder à une colonne avec les crochets. Il faut alors spécifier le nom de la colonne entre guillemets:

```
>>> df['pi_x']
0     0
1     0
2     1
3     2
4     2
5     3
6     3
7     4
8     4
9     4
10    4
11    5
12    5
13    6
```

```
14    6
Name: pi_x, dtype: int64
```

Cela peut se combiner avec d'autres méthodes comme l'affichage de statistiques `df.pi_x.describe()` ou encore des calculs:

```
>>> df.pi_x * 100
0       0
1       0
2     100
3     200
4     200
5     300
6     300
7     400
8     400
9     400
10    400
11    500
12    500
13    600
14    600
Name: pi_x, dtype: int64
```

19.7 Afficher les premières/dernières lignes

Parfois, on travaille avec des tableaux de très grande taille et il n'est pas pratique d'afficher toutes les données à l'écran. On construit d'abord un tableau de 1000 lignes avec les mêmes colonnes que le précédent:

```
>>> L = [isprime(i) for i in range(1000)]
>>> s = Series(L)
>>> d = {'isprime':s, 'pi_x':s.cumsum()}
>>> df = DataFrame(d)
```

Pour afficher les cinq premières lignes d'un tableau de données, on utilise la méthode `head()`:

```
>>> df.head()
  isprime  pi_x
0   False     0
1   False     0
2    True     1
3    True     2
4   False     2
```

Pour afficher les cinq dernières lignes d'un tableau de données, on utilise la méthode `tail()`:

```
>>> df.tail()
    isprime  pi_x
995   False   167
996   False   167
997    True   168
```

```
998   False   168
999   False   168
```

Les deux méthodes `head` et `tail` peuvent prendre un nombre entier en argument pour indiquer le nombre de lignes à afficher si on veut en voir plus ou moins:

```
>>> df.tail(10)
    isprime  pi_x
990   False   166
991    True   167
992   False   167
993   False   167
994   False   167
995   False   167
996   False   167
997    True   168
998   False   168
999   False   168
```

19.8 Sous-tableau

Pour accéder à un sous-tableau de lignes consécutives, on utilise les crochets comme pour les listes Python. Ici, on affiche le sous-tableau des lignes 500 à 519. En fait, cela crée un nouveau tableau de 20 lignes:

```
>>> df[500:520]
    isprime  pi_x   x_logx
500   False    95  80.4556
501   False    95  80.5906
502   False    95  80.7256
503    True    96  80.8605
504   False    96  80.9954
505   False    96  81.1303
506   False    96  81.2651
507   False    96  81.3999
508   False    96  81.5346
509    True    97  81.6694
510   False    97   81.804
511   False    97  81.9387
512   False    97  82.0733
513   False    97  82.2079
514   False    97  82.3425
515   False    97   82.477
516   False    97  82.6115
517   False    97  82.7459
518   False    97  82.8803
519   False    97  83.0147
```

Pour accéder à une donnée particulière dans le tableau, on utilise la méthode `at` en spécifiant l'indice de la ligne puis le nom de la colonne entre crochets:

```
>>> df.at[510, 'x_logx']
81.804042504952918
```

```
>>> df.at[510, 'pi_x']
97
```

19.9 Ajouter une colonne dans un tableau

Supposons que l'on veuille ajouter une colonne à un tableau. Cela se fait avec la méthode `insert()`.

Johann Carl Friedrich Gauss avait deviné au 19e siècle que `pi(x)`, le nombre de nombres premiers inférieurs à `x`, était approximativement `x/log(x)`. Construisons une série qui calcule cette fonction pour les 1000 premiers nombres entiers:

```
>>> from math import log
>>> def x_sur_log_x(x):
...     if x > 1:
...         return x/log(x)
...     else:
...         return None
>>> t = Series(range(1000)).apply(x_sur_log_x)
```

On ajoute la nouvelle colonne avec la méthode `insert` en spécifiant la position où on veut l'insérer, le titre de la colonne et les données:

```
>>> df.insert(2, 'x_logx', t)
>>> df['x_logx'] = t        # equivalent, notation comme les dictionnaires Python
```

En 1838, Dirichlet a contacté Gauss pour lui dire qu'il avait trouvé une meilleure approximation de la fontion `pi(x)` en utilisant l'intégrale de l'inverse de la fonction `log(x)`, c'est-à-dire par la fonction `Li(x)=integrate(1/log(t), (t,2,x))`.

En utilisant sympy, calculons les 1000 premières valeurs de `Li(x)` et ajoutons cette colonne dans le tableau:

```
>>> from sympy import Li
>>> K = [Li(x).n() for x in range(1000)]
>>> df['Li_x'] = Series(K, dtype='float64')
```

On peut afficher les premières et dernières lignes du tableau à quatre colonnes:

```
>>> df.head()
  isprime  pi_x   x_logx              Li_x
0   False     0      NaN  -1.04516378011749
1   False     0      NaN              -inf
2    True     1  2.88539                 0
3    True     2  2.73072   1.11842481454970
4   False     2  2.88539   1.92242131492156
>>> df.tail()
    isprime  pi_x   x_logx             Li_x
995   False   167  144.146  175.840407548189
996   False   167  144.269  175.985266957056
997    True   168  144.393  176.130105300461
998   False   168  144.517  176.274922605648
999   False   168  144.641  176.419718899799
```

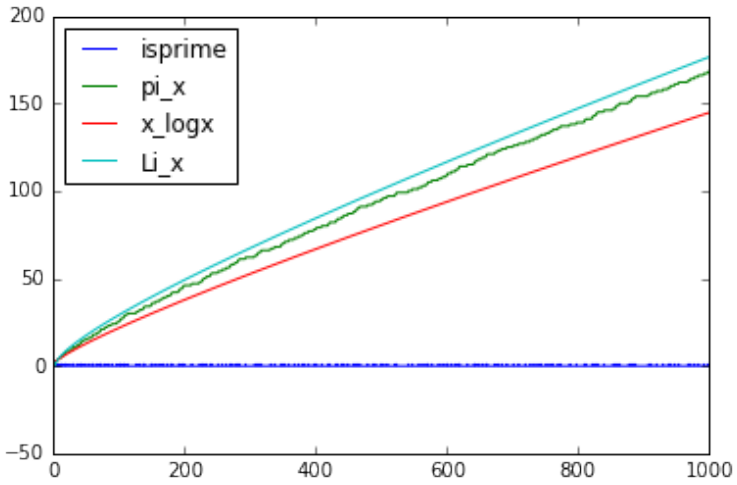
19.10 Visualiser les données

On active d'abord les dessins de matplotlib dans le notebook Jupyter:

```
%matplotlib inline
```

Pour visualiser les données, il suffit d'utiliser la commande `plot`:

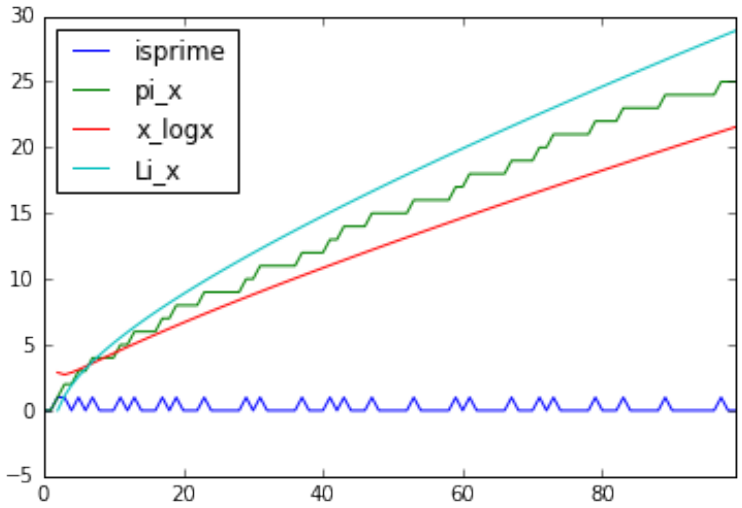
```
>>> df.plot()
```



On voit bien que `pi(x)`, le nombre de nombres premiers inférieurs à `x`, se trouve bien entre les fonctions `pi(x)` et `Li(x)` sur l'intervalle `[0,1000]`.

On peut visualiser qu'une partie par exemple l'intervalle `[0,100]` en choisissant d'abord un sous-tableau:

```
>>> df[:100].plot()
```



D'autres types de graphiques peuvent être plus adaptées dans d'autres situations (histogrammes, tartes, etc.). Voici la liste méthodes disponibles:

```
df.plot.area     df.plot.box      df.plot.hist     df.plot.pie
df.plot.bar      df.plot.density  df.plot.kde      df.plot.scatter
df.plot.barh     df.plot.hexbin   df.plot.line
```

On trouvera des exemples d'utilisation de ces méthodes de visualisation de données dans la documentation de pandas:

http://pandas.pydata.org/pandas-docs/stable/visualization.html#visualization

19.11 Exporter des données

Il est possible d'exporter un tableau de données de pandas vers plusieurs formats:

```
>>> df.to_[TOUCHE_TABULATION]
df.to_clipboard  df.to_excel      df.to_json       df.to_period     df.to_sql
df.to_csv        df.to_gbq        df.to_latex      df.to_pickle     df.to_stata
df.to_dense      df.to_hdf        df.to_msgpack    df.to_records    df.to_string
df.to_dict       df.to_html       df.to_panel      df.to_sparse     df.to_timestamp
df.to_wide       df.to_xarray
```

Pour exporter vers le format `.xlsx` on fait:

```
>>> from pandas import ExcelWriter
>>> writer = ExcelWriter('tableau.xlsx')
>>> df.to_excel(writer,'Feuille 1')
>>> writer.save()
```

On peut vérifier que Excel ouvre bien ce fichier qui se trouve dans le même répertoire que le notebook Jupyter (utiliser la commande pwd, abbréviation de "present working directory" en anglais, pour connaître ce répertoire en cas de doute).

Pour exporter vers le format `.csv` on fait:

```
>>> df.to_csv('tableau.csv')
```

> ***Note***
>
> L'importation et l'exportation vers le format excel .xls exige que les librairies Python `xlrd` et `openpyxl` soit installées. On peut les installer avec pip grâce à la commande `pip install xlrd openpyxl`.

19.12 Importer des données

Pour importer un fichier Excel dans pandas, on fait:

```
>>> import pandas as pd
>>> df = pd.read_excel('tableau.xlsx')
>>> df.head()
  isprime  pi_x      Li_x    x_logx
0   False     0 -1.045164       NaN
```

```
1   False     0      -inf       NaN
2    True     1  0.000000  2.885390
3    True     2  1.118425  2.730718
4   False     2  1.922421  2.885390
```

Parfois, un fichier Excel est corrompu et il vaut mieux passer par le format `.csv`. On procède alors ainsi:

```
>>> df = pandas.read_csv('tableau.csv')
>>> df.head()
   Unnamed: 0 isprime  pi_x      Li_x    x_logx
0           0   False     0 -1.045164       NaN
1           1   False     0      -inf       NaN
2           2    True     1  0.000000  2.885390
3           3    True     2  1.118425  2.730718
4           4   False     2  1.922421  2.885390
```

Parfois, la ligne de titre n'est pas sur la première ligne. À ce moment là, on peut spécifier la valeur de l'argument `header` pour dire où commencer la lecture du fichier en entrée:

```
>>> df = pandas.read_csv('tableau.csv', header=56)
>>> df.head()
   55  False  16  18.6860810929  13.7248383046
0  56  False  16      18.935063      13.911828
1  57  False  16      19.182942      14.098263
2  58  False  16      19.429748      14.284156
3  59   True  17      19.675508      14.469518
4  60  False  17      19.920249      14.654360
```

19.13 Exemple: analyser des données de data.gov.be

Le site web http://data.gov.be/ contient des centaines de données de toutes sortes de sujet sur la Belgique. Par exemple, à la page

http://data.gov.be/fr/dataset/4fd7a1cf-f959-46ff-83d0-807778fe3438

on retrouve des données météorologiques de Ostende depuis 2010. Sur cette page, on peut y télécharger le fichier `meteoostende.xls` au format excel. On peut l'importer dans pandas facilement:

```
>>> df = pandas.read_excel('meteoostende.xls')
```

Il est possible d'écrire l'URL directement ce qui évite d'avoir à télécharger le fichier:

```
>>> url = ("http://opendata.digitalwallonia.be/dataset/"
        "4fd7a1cf-f959-46ff-83d0-807778fe3438/resource/"
        "14306677-fb41-4472-9a23-2923f5e22d69/download/meteoostende.xls")
>>> df = pandas.read_excel(url)
```

Ce tableau de données comporte 1461 lignes:

```
>>> len(df)
1461
```

et 10 colonnes dont les titres sont:

```
>>> df.columns
Index([u'Période', u'Date', u'Température de l'air - moyenne (°C)',
       u'Température de l'air - minimum (°C)',
       u'Température de l'air - maximum (°C)', u'Humidité relative (%)',
       u'Rayonnement solaire quotidien - horizontal (kWh/m²/j)',
       u'Pression atmosphérique (kPa)', u'Vitesse du vent (m/s)',
       u'Température du sol (°C)'],
      dtype='object')
```

Les premières lignes permettent de se donner une idées des données. On peut aussi utiliser `df.describe()`:

```
>>> df.head()
   Période       Date  Température de l'air - moyenne (°C)  \
0        1 2010-01-01                                 3.90
1        2 2010-01-02                                 4.11
2        3 2010-01-03                                 3.24
3        4 2010-01-04                                 3.83
4        5 2010-01-05                                 3.88

   Température de l'air - minimum (°C)  Température de l'air - maximum (°C)  \
0                                 2.76                                 5.20
1                                 2.95                                 5.26
2                                 2.26                                 4.73
3                                 2.40                                 4.68
4                                 2.99                                 4.35

   Humidité relative (%)  \
0                 0.7465
1                 0.8288
2                 0.7919
3                 0.7825
4                 0.7757

   Rayonnement solaire quotidien - horizontal (kWh/m²/j)  \
0                                               1.08
1                                               0.65
2                                               1.04
3                                               0.68
4                                               0.72

   Pression atmosphérique (kPa)  Vitesse du vent (m/s)  \
0                        100.14                   7.70
1                        101.28                   6.13
2                        102.02                   5.46
3                        101.67                   3.45
4                        100.55                   4.86

   Température du sol (°C)
0                     6.15
1                     6.11
2                     5.94
3                     5.56
4                     5.42
```

Pour voir ce qu'il y a à la 100e ligne du tableau, on utilise la méthode `iloc`. Ce sont les données météo du 11 avril 2010:

```
>>> df.iloc[100]
Période                                                       101
Date                                              2010-04-11 00:00:00
Température de l'air - moyenne (°C)                          7.25
Température de l'air - minimum (°C)                          5.68
Température de l'air - maximum (°C)                          9.16
Humidité relative (%)                                      0.8023
Rayonnement solaire quotidien - horizontal (kWh/m²/j)        4.69
Pression atmosphérique (kPa)                               102.56
Vitesse du vent (m/s)                                        7.62
Température du sol (°C)                                      7.28
Name: 100, dtype: object
```

Pour afficher les moyennes par colonnes, on utilise la méthode `mean()`:

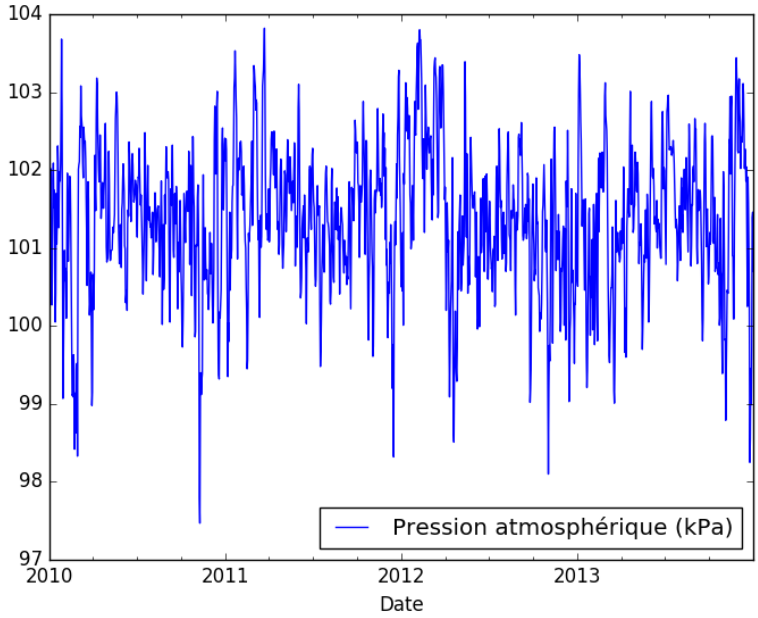
```
>>> df.mean()
Période                                             731.000000
Température de l'air - moyenne (°C)                  11.013005
Température de l'air - minimum (°C)                   9.289713
Température de l'air - maximum (°C)                  12.980171
Humidité relative (%)                                 0.796279
Rayonnement solaire quotidien - horizontal (kWh/m²/j)     3.283337
Pression atmosphérique (kPa)                        101.377502
Vitesse du vent (m/s)                                 6.117276
Température du sol (°C)                              11.255428
dtype: float64
```

Pour étudier une colonne en particulier, par exemple la pression atmosphérique, c'est-à-dire la septième colonne, on peut procéder ainsi:

```
>>> s = df.icol(7)
>>> s.head()
0    100.14
1    101.28
2    102.02
3    101.67
4    100.55
Name: Pression atmosphérique (kPa), dtype: float64
>>> s.describe()
count    1461.000000
mean      101.377502
std         0.932066
min        97.470000
25%       100.850000
50%       101.430000
75%       101.970000
max       103.820000
Name: Pression atmosphérique (kPa), dtype: float64
```

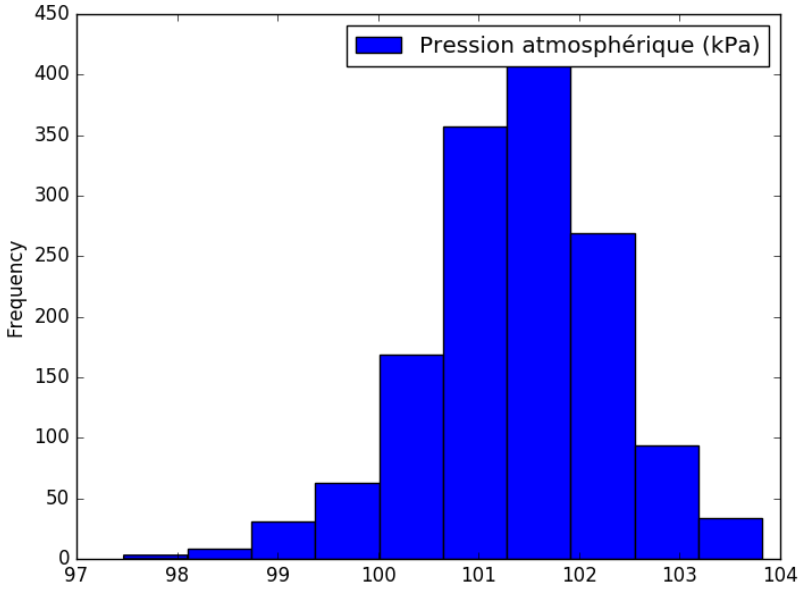
Finalement, on peut dessiner l'évolution de la pression atmosphérique en fonction de la date:

```
>>> date = df.columns[1]
>>> pression = df.columns[7]
>>> df.plot(x=date, y=pression)
```



Pour afficher un histogramme de la pression atmosphérique, il s'agit d'utiliser `df.plot.hist` avec les mêmes arguments:

```
>>> df.plot.hist(x=date, y=pression)
```



19.14 Filtrer les lignes d'un tableau

Parfois, il est pertinent de filtrer les lignes d'un tableau `df`. La façon de faire est d'abord de créer une série `s_vrai_faux` avec le même nombre de lignes contenant des valeurs booléennes en utilisant `True` pour les lignes que l'on veut garder et `False` sinon. La syntaxe est la suivante: `df[s_vrai_faux]` qui retourne un tableau filtré.

Voici un premier exemple facile où on veut afficher que les nombres multiples de 3 d'une série:

```
In [32]: s = Series(range(10))
In [31]: s
Out[31]:
0    0
1    1
2    2
3    3
4    4
5    5
6    6
7    7
8    8
9    9
dtype: int64
```

On crée une série de la même longueur qui teste si les entrées sont multiples de trois ou non:

```
In [29]: s % 3 == 0
Out[29]:
0     True
1    False
2    False
3     True
4    False
5    False
6     True
7    False
8    False
9     True
dtype: bool
```

On utilise la précédent série de booléen pour filtrer les lignes de la première série:

```
In [30]: s[s % 3 == 0]
Out[30]:
0    0
3    3
6    6
9    9
dtype: int64
```

Faisons maintenant un exemple au sujet de la météo de Ostende. Supposons qu'on s'intéresse à la température moyenne les jours de Noël à Ostende. D'abord, on crée une fonction qui teste si une date est bien le jour de Noël:

```
>>> est_noel = lambda date:date.day==25 and date.month==12
```

On applique cette fonction au tableau. On obtient une série de vrai ou faux:

```
>>> s_vrai_faux = df['Date'].apply(est_noel)
>>> s_vrai_faux.tail(10)
1451    False
1452    False
1453    False
1454     True
1455    False
1456    False
1457    False
1458    False
1459    False
1460    False
Name: Date, dtype: bool
```

Finalement, on filtre le tableau avec cette série. Et on affiche que les deux colonnes qui nous intéressent (la date et la température):

```
>>> df_noel = df[s_vrai_faux]
>>> df_noel.icol([1,2])
           Date  Température de l'air - moyenne (°C)
358  2010-12-25                                3.63
723  2011-12-25                               10.62
1089 2012-12-25                                9.22
1454 2013-12-25                                7.23
```

19.15 Conclusion

Les outils Python tels que la librairie pandas sont utilisés par les gens qui analysent des données comme le média alternatif BuzzFeedNews qui a mis au jour en janvier 2016 [TennisRacket] le fait que des matchs de tennis de l'ATP avaient été truqués. Les données ainsi que les notebook Jupyter réalisés par BuzzFeedNews sont disponibles sur github à l'adresse http://github.com/BuzzFeedNews/everything. On y trouvera d'autres analyses de données tels que les tremblements de terre reliés à l'exploitation des gaz de schiste aux États-Unis, les mouvements des donateurs de la campagne présidentielle américaine lorsqu'un candidat sort de la course, ou une analyse du placement des enfants dans les crèches.

Le lecteur désirant en savoir plus sur pandas est invité à lire les tutoriels en ligne sur pandas. La librairie pandas est utilisée par la librairie Python de statistiques StatsModels qui permet de faire encore plus comme des modèles statistiques, des estimations et des tests statistiques.

Sage — Zimmermann, Paul, Laurent Fousse, François Maltey, Matthias Meulien, Marc Mezzarobba, Clément Pernet, Nicolas M. Thiéry, et al. Calcul mathématique avec Sage. S. l.: CreateSpace Independent Publishing Platform, 2013. http://sagebook.gforge.inria.fr/

GeoGebra Introduction à GeoGebra, Version 4.4, traduction en français par Noël Lambert, novembre 2013, http://static.geogebra.org/book/intro-fr.pdf

Swinnen Gérard Swinnen, Apprendre à programmer avec Python 3, 2012. http://inforef.be/swi/download/apprendre_python3_5.pdf

Thinklike Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers, How to Think Like a Computer Scientist - Learning with Python, 3nd Edition, 2012. http://openbookproject.net/thinkcs/python/english3e/

Massart Thierry Massart, Syllabus du cours INFO-F-101 Programmation, Université libre de Bruxelles, 2015-2016, http://www.ulb.ac.be/di/verif/tmassart/Prog/html/

TennisRacket Methodology and code supporting the BuzzFeed News/BBC article, "The Tennis Racket," published Jan. 17, 2016. http://www.buzzfeed.com/heidiblake/the-tennis-racket