

Table des matières

1. Introduction à l'environnement de travail et à SageMath
 1. Environnement de travail
 2. Prise en main de SageMath
2. Anneaux, corps, polynômes, expressions
 1. Structures prédéfinies
 2. Variables et expressions symboliques
 3. Polynômes
 4. Idéaux et bases de Gröbner
 5. Résultants et élimination
3. Syntaxe Python/Sage
 1. Variables, assignement
 2. Syntaxe
 3. Structures de contrôle
 4. Listes
 5. Fonctions
4. La hiérarchie des types en Sage
 1. Parents et éléments
 2. Base
 3. Classes, types
5. Tracer des courbes et des surfaces en Sage
 1. Tracés implicites et paramétriques

Introduction à l'environnement de travail et à SageMath

Environnement de travail

- Creation des comptes utilisateurs, première connexion, lancement d'un terminal, changer son mot de passe (`passwd`).
- Accès distant à `bourbaki.math.uvsq.fr` via ssh/scp/sftp.

Prise en main de SageMath

- Présentation de SageMath : histoire, buts, modèle de développement, architecture.

Interfaces pour l'utilisation de SageMath

- Notebook *Jupyter* : espace de cours sur <https://bourbaki.math.uvsq.fr/>,
- SageMathCloud : <https://cocalc.com/> (SageMath Inc.).
- Dans un terminal : taper la commande `sage` dans un terminal de la salle.
- Installation locale sur machine personnelle (notebook *classique* et terminal) :
 - Instructions générales : <http://www.sagemath.org/download.html>
 - Linux :
 - Gestionnaire de paquets du système: sous Arch, Debian, Ubuntu.
 - Distribution précompilée.
 - Compiler à partir du code source.
 - MacOS, distribution précompilée (méthode conseillée).
 - Windows : Image virtualisée (pour VirtualBox).
 - Toute architecture (**méthode conseillée pour toute plateforme**) : Sage Debian Live (démarrage d'un système Linux préconfiguré depuis clef USB ou CD).
 - Toute architecture : système Linux complet virtualisé de l'UFR de Sciences (choisir la VM 2).

Premiers pas avec SageMath

Référence : chapitre 1 du Sagebook.

- Utilisation du notebook Jupyter : créer une feuille de calcul, cellules, insérer une cellule, insérer du texte riche (format Markdown).
- Affichage du résultat, instruction `print`, rappeler un résultat précédent, commandes *magiques* `%display latex` et `%display plain`.
- Syntaxe Python, objets, propriétés, méthodes.

- Aide, recherche et exploration : completion automatique (touche **tab**), aide en ligne (clef magique **?**), exploration du code source (clef magique **??**).

Anneaux, corps, polynômes, expressions

Structures prédéfinies

Sage prédéfinit pour l'utilisateur certaines structures algébriques de base, les noms s'expliquent tous seuls :

```
sage: NN
Non negative integer semiring
sage: ZZ
Integer Ring
sage: QQ
Rational Field
sage: QQbar
Algebraic Field
sage: RR
Real Field with 53 bits of precision
sage: RDF
Real Double Field
sage: CC
Complex Field with 53 bits of precision
sage: GF(7)
Finite Field of size 7
sage: Zmod(15)
Ring of integers modulo 15
sage: Zmod(7)
Ring of integers modulo 7
sage: Qp(5)
5-adic Field with capped relative precision 20
```

Variables et expressions symboliques

Voir le Sagebook, Section 1.2.

Sage peut utiliser toute la syntaxe du langage Python, à partir de la possibilité de définir des variables :

```
sage: a = 2+2
sage: a
4
sage: a+2
6
```

Une variable spéciale, `_` (tiret bas), permet de faire référence à la dernière valeur affichée :

```
sage: _
6
```

Cette variable est plus utile dans le terminal que dans le notebook, où on peut revenir sur une cellule précédemment saisie.

Sage inclut un moteur de calcul symbolique. Le nom `x` est une *variable symbolique* prédéfinie au lancement de Sage.

```
sage: a = (x^2 + 2) * (x^3 + 4)
sage: a
(x^3 + 4)*(x^2 + 2)
sage: a.expand()
x^5 + 2*x^3 + 4*x^2 + 8
```

On peut définir d'autres variables symboliques avec la fonction `var`. Ses effets sont une entorse aux conventions de bon usage des langages de programmation, mais Sage met la facilité d'utilisation en avant...

```
sage: var('y')
y
sage: (x+y)*(x-y)
(x + y)*(x - y)
```

Une façon équivalente, mais moins magique d'écrire la même chose :

```
sage: y = var('y')
sage: (x+y)*(x-y)
(x + y)*(x - y)
```

Mais attention !

```
sage: z = var('z')
sage: z + z
2*z
```

Beurk ! En programmation il est bon usage de faire une différence entre z (un *nom de variable*) et sa *valeur* z (une *expression symbolique*).

Beaucoup moins risqué : utiliser la méthode `SR.var` à la place de `var` :

```
sage: SR.var('t')
t
sage: t
-----
NameError                                 Traceback (most recent call last)
...
NameError: name 't' is not defined
```

Correct, le nom t ne souille pas l'espace des noms. Maintenant :

```
sage: T = SR.var('t')
sage: T + T
2*t
sage: T + t
-----
NameError                                 Traceback (most recent call last)
...
NameError: name 't' is not defined
```

Attention à ne pas confondre les *noms de variables* et les *variables symboliques*, même lorsqu'elles ont la même écriture.

```
sage: X = SR.var('x')
sage: X
x
sage: x = 6
sage: x
6
sage: X+x
x + 6
```

Dans l'exemple ci-dessus, X est un *nom de variable* qui pointe vers une *variable symbolique* représentée par le caractère x . Plus bas, x est encore un *nom de variable* qui pointe vers la valeur entière 6.

Polynômes

Voir le Sagebook, chapitre 7.

En plus des polynômes symboliques, Sage possède plusieurs classes spécialisées de polynômes à coefficients dans différents anneaux.

La syntaxe la plus simple pour définir un anneau de polynômes (par exemple sur le corps des rationnels) est

```
sage: A.<x> = QQ[]
sage: A
Univariate Polynomial Ring in x over Rational Field
```

Cette syntaxe associe le nom x au *générateur* de A . Il ne s'agit plus d'expressions symboliques, tout calcul est maintenant exécuté immédiatement :

```
sage: (x^2 + 2) * (x^3 + 4)
x^5 + 2*x^3 + 4*x^2 + 8
```

La syntaxe utilisée pour définir les polynômes n'est pas une syntaxe Python standard. Voici deux autres façons de définir A qui sont valides en Python.

```
sage: A = QQ['x']
sage: x = A.gen()
```

```
sage: A = PolynomialRing(QQ, 'x')
sage: x = A.gen()
```

Dans les deux cas, on voit bien la différence entre le *nom de variable* `x` et sa *représentation* (aussi `x`). Cela ne doit pas être nécessairement ainsi :

```
sage: A = QQ['x']
sage: toto = A.gen()
sage: toto^2 + 1
x^2 + 1
```

Polynômes multivariés

Avec la même syntaxe, on peut définir des anneaux de polynômes à plusieurs variables :

```
sage: B.<x,y,z> = RR[]; B
Multivariate Polynomial Ring in x, y, z over Real Field with 53 bits of precision
```

L'ordre monomial par défaut sur les anneaux multivariés est *degrevlex* :

```
sage: B.term_order()
Degree reverse lexicographic term order
sage: z < y < x < z^2 < y*z < x*z < y^2 < x*y < x^2
True
```

Il est possible de choisir l'ordre au moment de la création :

```
sage: C.<x,y,z> = PolynomialRing(QQ, order='lex')
sage: C
Multivariate Polynomial Ring in x, y, z over Rational Field
sage: C.term_order()
Lexicographic term order
sage: z < z^2 < x < x*y < x^2
True
```

Il est aussi possible d'obtenir une copie d'un anneau avec un ordre différent :

```
sage: D = B.change_ring(order='invlex')
sage: D.term_order()
Inverse lexicographic term order
sage: D == B
False
```

Sage supporter quelques ordres prédéfinis : `lex`, `invlex`, `degrevlex`, ... La liste complète peut être obtenue interactivement :

```
sage.rings.polynomial.term_order?
```

Il est possible de construire des ordres arbitraires grâce à la classe `TermOrder`. Il est par exemple possible de construire un ordre *avec poids* :

```
sage: t = TermOrder('wdegrevlex', (1,2,3)); t
Weighted degree reverse lexicographic term order with weights (1, 2, 3)
sage: E.<x,y,z> = PolynomialRing(QQ, order=t)
sage: x < y < x^2 < z < x*y < x^3 < y^2
True
```

Lire la documentation de `TermOrder` pour plus de détails.

Idéaux et bases de Gröbner

Sage sait calculer la division Euclidienne entre polynômes multivariés :

```
sage: A.<x,y> = QQ[]
sage: (x^2 - y) % (x + y)
x^2 + x
sage: (x^2 - y) // (x + y)
-1
sage: (x^2 - y).quo_rem(x + y)
(-1, x^2 + x)
```

Le résultat est bien évidemment dépendant de l'ordre choisi :

```
sage: (x^2 - y).quo_rem(x + y)
(-y + x, y^2 - y)
```

Le point de départ pour travailler avec les bases de Gröbner est la méthode `.ideal()` des anneaux de polynômes.

```
sage: A.<x,y,z> = QQ[]
sage: I = A.ideal(x^2 - y*z, x^3*y + 2*y*z - 4*z^2); I
Ideal (x^2 - y*z, x^3*y + 2*y*z - 4*z^2) of Multivariate Polynomial Ring in x, y, z
over Rational Field
```

Les idéaux ont une méthode `.groebner_basis()`, dont le résultat dépend bien évidemment de l'ordre choisi :

```
sage: I.groebner_basis()
[y^3*z^2 + 2*x*y*z - 4*x*z^2, x*y^2*z + 2*y*z - 4*z^2, x^2 - y*z]
```

C'est cette base qui est utilisée pour calculer les formes normales de polynômes modulo l'idéal. Les méthodes `.mod` (définie sur les polynômes) et `.reduce` (définie sur les idéaux et sur les polynômes), calculent la forme normale :

```
sage: (x^6*z - y*z^3 + x*z).mod(I)
-2*x*y*z^3 + 4*x*z^4 - y*z^3 + x*z
sage: I.reduce(x^6*z - y*z^3 + x*z)
-2*x*y*z^3 + 4*x*z^4 - y*z^3 + x*z
sage: (x^6*z - y*z^3 + x*z).reduce(I)
-2*x*y*z^3 + 4*x*z^4 - y*z^3 + x*z
```

Attention : les méthodes définies sur les polynômes se comportent différemment lorsqu'elles reçoivent en paramètre une liste de polynômes plutôt qu'un idéal.

```
sage: (x^6*z - y*z^3 + x*z).reduce([x^2 - y*z, x^3*y + 2*y*z - 4*z^2])
y^3*z^4 - y*z^3 + x*z
sage: (x^6*z - y*z^3 + x*z).mod([x^2 - y*z, x^3*y + 2*y*z - 4*z^2])
-2*x*y*z^3 + 4*x*z^4 - y*z^3 + x*z
```

De manière générale, préférez `I.reduce` en toute occasion.

Lorsque un polynôme appartient à l'idéal, la méthode `lift` permet de l'exprimer comme une combinaison linéaire de ses générateurs :

```
sage: p = 2*x^3*y^3 - x*y^4*z + 2*y^3*z - 4*y^2*z^2
sage: I.reduce(p)
0
sage: p.lift(I)
[x*y^3, y^2]
sage: _[0] * I.0 + _[1] * I.1 == p
True
```

Résultants et élimination

Le résultant de deux polynômes est calculé par la méthode `.resultant()`

```
sage: A.<x> = QQ[]
sage: (x^2+1).resultant(2*x)
4
```

Dans les anneaux à plusieurs variables, Sage élimine par défaut la première variable :

```
sage: A.<x,y,z> = QQ[]
sage: (x*y + z).resultant(x+z+y)
y^2 + y*z - z
```

Il est aussi possible d'indiquer la variable que l'on souhaite éliminer :

```
sage: (x*y + z).resultant(x+z+y, z)
-x*y + x + y
```

Des éliminations plus complexes peuvent être réalisées grâce aux bases de Gröbner, en effet on a déjà vu plus haut que Sage supporte le calcul de bases de Gröbner pour l'ordre *lex*, ce qui permet de calculer les idéaux d'élimination. Cette même fonctionnalité est encapsulée dans la méthode `.elimination_ideal()`, que voici à l'œuvre sur le même exemple :

```
sage: I = A.ideal(x*y + z, x+z+y)
sage: I.elimination_ideal(z)
Ideal (x*y - x - y) of Multivariate Polynomial Ring in x, y, z over Rational Field
```

`.elimination_ideal()` peut aussi prendre une liste de variables à éliminer :

```
sage: A.<x,y,z> = QQ[]
sage: I = A.ideal(x*y + w*z, x*z + w*y, x*w + y*z, x^2 + y^2 + z^2 + w^2)
sage: I.elimination_ideal([x,y]).gens()
[3*w^2*z + z^3, w^3 + 3*w*z^2, w*z^3, z^5]
```

Syntaxe Python/Sage

Variables, assignement

Python est un langage dynamiquement typé, les variables n'ont pas besoin d'être déclarées, et leur type peut changer au cours de l'exécution.

```
python: a = 3
python: type(a)
<type 'int'>
python: a = '3'
python: type(a)
<type 'str'>
python: a
'3'
python: int(a)
3
```

Syntaxe

L'indentation en python a une valeur syntaxique : elle sert à délimiter les blocs. Toutes les lignes d'un même bloc doivent être précédées du même nombre d'espaces blancs ; en général on conseille d'utiliser 4 espaces blancs.

Voici un exemple de bloc conditionnel mettant en évidence cette syntaxe.

```
if a == 0:
    print 'a vaut 0'
elif a > 0:
    print 'a est positif'
    print 'il vaut %d' % a
else:
    print 'a est négatif'
print 'encore des questions sur a?'
```

Les retours à la ligne sont interdits en Python, sauf entre parenthèses (), crochets [], et accolades {}. L'exemple suivant est incorrect :

```
if a == b
and c == d:
    print 'yes'
```

Alors que ceci est correct (et équivalent à la sémantique entendue):

```
if (a == b
    and c == d):
    print 'yes'
```

Structures de contrôle

Source : <https://docs.python.org/2/tutorial/controlflow.html>

if... else

La seule construction conditionnelle existante en Python est `if... elif... else...`. Toutes les branches sont optionnelles, à l'exception du `if`, il peut y avoir un nombre quelconque de `elif`, mais un seul `else` à la fin.

```
if a == b == c:
    print 'égaux'
elif a <= b <= c or c <= b <= a:
    print 'b au milieu'
elif b <= a <= c or c <= a <= b:
    print 'a au milieu'
else:
    print 'c au milieu'
```

Boucle `for... in`

Il existe deux types de boucles en Python. La plus couramment utilisée est le `for... in` qui permet de parcourir les éléments d'une liste.

```
for i in range(10):
    print i
```

La fonction `range`

La boucle `for` est souvent utilisée en conjonction avec la fonction `range`, dont la syntaxe générale est :

```
range(start, end, step)
```

Ainsi appelée, la fonction génère la liste des entiers entre `start` (inclus) et `end` (non inclus) avec pas de `step` :

```
python: range(0, 10, 2)
[0, 2, 4, 6, 8]
```

Les deux autres syntaxes admissibles sont `range(start, end)` (pas égal à 1) et `range(end)` (début égal à 0).

Note : À partir de Python 3.x, `range` ne renvoie plus une liste, mais un *générateur*. La différence réside exclusivement dans l'utilisation de la mémoire, beaucoup plus efficace avec la 3.x. Le même comportement est réalisé par la fonction `xrange` en Python 2.x.

Boucle `while`

La deuxième boucle est très similaire à la boucle `while` en C.

```
a = 0
while a < 10:
    a += 1
    print a
```

Pas étonnant qu'il soit alors beaucoup plus facile d'écrire une boucle infinie :

```
while True:
    print 'boucle toujours'
```

Interrompre les boucles: `break`, `continue` et `return`

Comme en C, l'instruction `break` sort de la boucle sans vérifier la condition :

```
for i in range(10):
    print i
    if i > 2:
        break
```

L'instruction `continue` passe à l'itération suivante en sautant le reste du corps :

```

for i in range(10):
    if i % 2 == 0:
        continue
    print i

```

Et le `return` sort immédiatement de toute boucle et de la fonction qui le contient.

Listes

Source : <https://docs.python.org/2/tutorial/datastructures.html#more-on-lists>

L'un des objets les plus utilisés en Python, ce sont les listes. On déclare une liste avec les crochets `[]`, et on accède à ses éléments comme on accède aux éléments d'un tableau en C :

```

python: l = [1, 2, 'a', True]
python: l
[1, 2, 'a', True]
python: l[0]
1
python: l[3]
True
python: l[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

```

La syntaxe pour accéder aux éléments d'une liste est plus puissante en Python qu'en C. Les indices négatifs accèdent aux éléments à partir du dernier :

```

python: l[-1]
True
python: l[-4]
1
python: l[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

```

Il est aussi possible d'obtenir les sous-listes d'une liste avec une syntaxe qui rappelle les paramètres de la fonction `range`. L'expression `l[start:end:step]` donne la sous-liste de `l` qui démarre à l'élément `start` (inclus), se termine à l'élément `end` (exclus) et saute tous les `step` éléments. Chacun des composants peut être omis, il prendra alors une valeur par défaut (0 pour `start`, la longueur de la liste pour `end`, 1 pour `step`).

```

python: l = range(10)
python: l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
python: l[0:4]
[0, 1, 2, 3]
python: l[0:4:2]
[0, 2]
python: l[2:]
[2, 3, 4, 5, 6, 7, 8, 9]
python: l[:-3]
[0, 1, 2, 3, 4, 5, 6]
python: l[0::3]
[0, 3, 6, 9]
python: l[4:-2]
[4, 5, 6, 7]
python: l[::]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
python: l[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```

La syntaxe `[:]` est un raccourci courant pour *copier* une liste :

```
python: l[:] == l
True
python: l[:] is l
False
```

Compréhensions

Python offre une syntaxe pour la création des listes qui devrait être familière aux mathématiciens. C'est un héritage du langage Lisp appelé *compréhensions de listes* :

```
python: [a + 0.5 for a in range(10)]
[0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5]
```

ce qui est sémantiquement équivalent à

```
l = []
for a in range(10):
    l.append(a + 0.5)
```

On peut ajouter un nombre arbitraire de `for` et de `if` (sans `else`) dans une compréhension, ils seront déroulés dans l'ordre :

```
[x*y for x in range(10)
    for y in range(x)
        if (x + y) % 2 == 0]
```

(les retours à la ligne sont optionnels) est équivalent à

```
l = []
for x in range(10):
    for y in range(x):
        if (x + y) % 2 == 0:
            l.append(x*y)
```

Opérateurs fonctionnels sur les listes

Python définit quelques fonctions typiques des langages fonctionnels pour traiter efficacement une liste. Les plus importantes sont `map`, `filter`, et `reduce` ; elles laissent inchangée la liste d'origine.

`map` applique une fonction à chaque élément d'une liste et renvoie la liste des résultats. Dans l'exemple ci-dessous, `hex` est la fonction qui renvoie la représentation hexadécimale d'un entier :

```
python: map(hex, range(10,20))
['0xa', '0xb', '0xc', '0xd', '0xe', '0xf', '0x10', '0x11', '0x12', '0x13']
```

`filter` applique une fonction à chaque élément d'une liste et renvoie la liste des éléments pour lesquels la fonction a donné `True`. L'exemple suivant utilise la fonction `is_prime` de Sage, qui teste la primalité de son paramètre :

```
sage: filter(is_prime, range(20))
[2, 3, 5, 7, 11, 13, 17, 19]
```

`reduce` parcourt la liste de gauche à droite. Le résultat de chaque itération est obtenu en appliquant une fonction bivariée au résultat de l'itération précédente et à l'élément courant. Dans l'exemple suivant, `reduce` calcule la somme des entiers de 0 à 9 ; la fonction standard `operator.add` est l'équivalent de l'opérateur `+` :

```
python: import operator
python: reduce(operator.add, range(10))
45
```

À partir de l'opérateur `reduce`, il est facile de définir quelques autres fonctions très utiles : `sum`, `all`, `any`, `max`, `min`, `join`. Vous êtes invités à en lire la documentation.

Fonctions

Source : <https://docs.python.org/2/tutorial/controlflow.html>

Les fonctions Python sont définies par le mot clef `def`. Elles peuvent prendre un nombre arbitraire de paramètres, et renvoient une valeur à l'aide du mot clef `return`. Toute fonction renvoie une valeur, les fonctions qui n'ont pas de `return` renvoient la valeur spéciale `None`.

```
def max(x, y):
    if x > y:
        return x
    else:
        return y
```

Certains paramètres peuvent prendre des valeurs par défaut. Si un paramètre prend une valeur par défaut, tous ceux qui le suivent doivent aussi en prendre.

```
python: def test(a, b, c=0, d=False):
.....
      return a, b, c, d

python: test(1, 2)
(1, 2, 0, False)
python: test(1, 2, 3)
(1, 2, 3, False)
python: test(1, 2, 3, 4)
(1, 2, 3, 4)
python: test(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: test() takes at least 2 arguments (1 given)
```

Les paramètres d'une fonction peuvent être assignés hors ordre avec la notation `paramètre=valeur`:

```
python: test(b=1, a=2)
(2, 1, 0, False)
python: test(1, 2, d=4)
(1, 2, 0, 4)
```

Python fournit deux opérateurs unaires pour transformer des objets en paramètres d'une fonction. L'opérateur `*` transforme une liste ou un tuple, tandis que l'opérateur `**` transforme un dictionnaire :

```
python: l = range(4)
python: test(*a)
(0, 1, 2, 3)
python: d = { 'a' : 3, 'b' : 5, 'd' : 1 }
python: test(**d)
(3, 5, 0, 1)
```

La hiérarchie des types en Sage

Parents et éléments

Beaucoup de structures algébriques en Sage sont organisées autour du concept de parent-élément.

Par exemple, un anneau de polynômes est un *parent*, à qui appartiennent des éléments. La majorité des valeurs en Sage possède un parent, qu'on peut interroger avec la méthode `parent()` :

```
sage: 1.parent()
Integer Ring
sage: (1/2).parent()
Rational Field
sage: (0.5).parent()
Real Field with 53 bits of precision
```

Attention, beaucoup de noms prédéfinis appartiennent à l'*anneau symbolique*, même si l'on pourrait imaginer qu'ils appartiennent à un parent plus spécifique (réels, complexes, ...).

```
sage: i.parent()
Symbolic Ring
sage: pi.parent()
Symbolic Ring
sage: x.parent()
Symbolic Ring
```

Sage connaît un bon nombre de *coercitions* automatiques, qui permettent de faire des opérations sur des éléments de parents différents.

```
sage: A.<x> = QQ[]
sage: x.parent()
Univariate Polynomial Ring in x over Rational Field
sage: 2.parent()
Integer Ring
sage: (2+x).parent()
Univariate Polynomial Ring in x over Rational Field
```

Parfois Sage ne sait pas trouver de *coercition canonique*, dans ce cas il donne une erreur :

```
sage: a = GF(7)(2); a
2
sage: a.parent()
Finite Field of size 7
sage: x + a
...
TypeError: unsupported operand parent(s) for '+': 'Univariate Polynomial Ring in
x over Rational Field' and 'Finite Field of size 7'
```

Il est possible dans ces cas de *convertir* un élément en l'associant à un nouveau parent, Sage appliquera alors d'autres règles, dites de *conversion*, qui ne doivent pas être canoniques. Toutes les conversions prennent la forme de `parent` (élément).

```
sage: x + QQ(a)
x + 2
sage: _.parent()
Univariate Polynomial Ring in x over Rational Field
```

Observons cela sur un exemple simple. Il existe un morphisme canonique de \mathbb{Z} vers $\mathbb{Z}/(10)$, Sage peut alors additionner deux éléments appartenant à ces deux parents, et le résultat sera un élément de $\mathbb{Z}/(10)$.

```
sage: a = Zmod(10)(2)
sage: a.parent()
Ring of integers modulo 10
sage: 11.parent()
Integer Ring
sage: 11 + a
3
sage: _.parent()
Ring of integers modulo 10
```

Le *lift* d'un élément de $\mathbb{Z}/(10)$ vers \mathbb{Z} n'a rien de canonique, mais il est tout de même convenable d'appliquer cette *conversion* lorsque elle est demandée par l'utilisateur :

```
sage: QQ(a).parent()
Rational Field
sage: 11 + QQ(a)
13
```

De façon similaire, il n'existe pas de morphisme naturel de \mathbb{Q} vers $\mathbb{Z}/(10)$. Sage donne donc correctement une erreur ici :

```
sage: 1/3 + a
...
TypeError: unsupported operand parent(s) for '+': 'Rational Field' and 'Ring of
integers modulo 10'
```

Mais, puisque 3 est inversible modulo 10, il est tout de même possible de donner un sens à cette conversion :

```
sage: Zmod(10)(1/3) + a
9
```

Base

Beaucoup de structures algébriques en Sage ont une *base*. Par exemple, les anneaux de polynômes ont pour base leur anneau de coefficients.

```
sage: A.<x> = GF(7) []
sage: A.base()
Finite Field of size 7
sage: A.base_ring()
Finite Field of size 7
```

Sage considère aussi les bases lorsque il cherche à trouver une *coercition* entre deux éléments. Dans cet exemple, x est un polynôme à coefficients dans \mathbb{F}_7 , alors que 10 est un entier. Au moment de la coercition, Sage transforme 10 en un élément de \mathbb{F}_7 , puis le en un élément de $\mathbb{F}_7[x]$.

```
sage: 10.parent()
Integer Ring
sage: x + 10
x + 3
sage: _.parent()
Univariate Polynomial Ring in x over Finite Field of size 7
```

Il est possible d'obtenir une copie d'un anneau avec une base différente grâce à la méthode `change_ring()` :

```
sage: B = A.change_ring(QQ); B
Univariate Polynomial Ring in x over Rational Field
sage: B(x)
x
sage: B(x).parent()
Univariate Polynomial Ring in x over Rational Field
sage: x.parent()
Univariate Polynomial Ring in x over Finite Field of size 7
```

Classes, types

Chaque objet mathématique est représenté par une *classe* Python. L'opérateur Python `type` permet d'interroger la classe d'un objet :

```
sage: type(1)
<type 'sage.rings.integer.Integer'>
```

Souvent, différentes classes Python réalisent le même objet mathématique, ce qui permet d'avoir plusieurs implantations pour un même objet. Cela est particulièrement évident pour les corps finis, où différents sous-systèmes sont choisis selon la cardinalité.

```
sage: A.<x> = GF(2) []
sage: B.<y> = GF(3) []
sage: C.<z> = GF(3^2, 'a') []
sage: type(x)
<type 'sage.rings.polynomial.polynomial_gf2x.Polynomial_GF2X'>
sage: type(y)
<type 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>
sage: type(z)
<type 'sage.rings.polynomial.polynomial_zz_pex.Polynomial_ZZ_pEX'>
```

Voici un autre exemple où deux classes différentes réalisent le même objet mathématique, mais avec un code différent :

```
sage: A = matrix([[1,1],[1,1]])
sage: B = matrix([[1,1],[1,1]]), sparse=True)
sage: A == B
True
sage: type(A)
<type 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
sage: type(B)
<type 'sage.matrix.matrix_integer_sparse.Matrix_integer_sparse'>
```

Tracer des courbes et des surfaces en Sage

Source: Sage Tutorial

Beaucoup d'objets en Sage peuvent être *tracés* :

```
sage: plot(cos)      # Affiche la fonction cosinus entre -1 et 1
sage: plot(cos, -5, 5)  # même fonction entre -5 et 5
```

La majorité des objets sont tracés par la fonction `plot()`, mais beaucoup disposent aussi d'une méthode plus spécifique `.plot()` attachée à l'objet, qui peut donner accès à plus d'options :

```
sage: A.<x,y,z> = QQ[]
sage: C = Curve(x^3 + y^3 + x*z^2)
sage: plot(C)      # Affiche la courbe C

sage: C.plot()    # Affiche la même courbe

sage: C.plot(patch=0)  # Affiche la même courbe dans une carte affine différente
```

Les *plots* sont des objets, qu'on peut stocker, copier...

```
sage: a = C.plot()
sage: a.set_aspect_ratio(2.0)
sage: a
```

et même additionner, le résultat étant la superposition de plusieurs *plots*.

```
sage: C.plot(patch=0) + C.plot(patch=1) + a
```

Tracés implicites et paramétriques

Le lieu d'annulation d'un polynôme à deux variables est tracé par la fonction `implicit_plot()`. Cette fonction prend en argument le polynôme à tracer et des bornes sur les axes sous forme de *tuples* :

```
sage: A.<x,y> = QQ[]
sage: implicit_plot(x*y - 1, (x, -10, 10), (y, -10, 10))
```

Il est important que le polynôme appartienne à un anneau bivarié, faute de quoi, Sage va donner une erreur :

```
sage: A.<x,y,z> = QQ[]
sage: implicit_plot(x*y - 1, (x, -10, 10), (y, -10, 10))
-----
ValueError                                Traceback (most recent call last)
...
ValueError: Variable 'z' not found
```

La fonction `implicit_plot3d()` a les mêmes conventions d'appel et permet de tracer des surfaces dans un espace à trois dimensions.

```
sage: implicit_plot3d(x*y*z - 1, (x,-10,10), (y,-10,10), (z,-10,10))
```

La fonction `parametric_plot()` permet de tracer des courbes ou des surfaces paramétriques dans un espace à deux ou trois dimensions. Dans ce sens, il s'agit d'une forme plus générale de `plot(cos)`.

```
sage: var('a,b')
(a, b)
sage: parametric_plot([cos(a), sin(a)], (a,-pi,pi))

sage: parametric_plot([cos(a), cos(a*b), sin(a)], (a,-pi,pi), (b,-pi,pi))
```

