Introduction à la programmation avec Sage

1 Pour démarrer

A l'aide du navigateur de votre choix connectez vous à l'adresse `http://sage-math.univ-lyon1.fr`. Après vous être identifié cliquez sur le lien `New Worksheet`. Une feuille de travail vierge apparaît. Cliquez sur son nom `Untitled` et renommez–la par exemple `TD1`. En fin de séance, n'oubliez pas de sauvegarder votre travail. La feuille de travail se présente comme un *suite de cellules* dans lesquelles vous taperez vos *instructions* destinées à être *exécutées*

- Une même cellule peut contenir autant d'instructions que l'on veut, une par ligne, ou même, pour certaines instructions simples, plusieurs instructions sur une même ligne, séparées par des points-virgules.
- Pour créer une nouvelle cellule glissez le curseur un peu au dessus d'une cellule déja présente jusqu'à l'apparition d'une ligne horizontale bleue. Faites alors un *clic gauche*.
- Pour *lancer l'exécution des instructions* de la cellule contenant le curseur frappez (simultanément) les touches <MAJUSCULE><ENTREE>.

Sage et Python

Avant de commencer il faut savoir que `Sage` est un outil destiné à faire des calculs *numériques*, des calculs sur des *matrices* à éléments dans toutes sortes d'*anneaux*, y compris des anneaux de *polynômes*, tracer des *graphes de fonctions*, etc. Il est aussi capable, même s'il n'excelle pas encore dans ce domaine, de faire du *calcul formel*, c'est à dire du calcul sur des *expressions mathématiques dans lesquelles figurent des symboles de variables*. On pourrait utiliser `Sage` comme une calculette perfectionnée. Ce serait dommage, `Sage` est aussi un *langage de programmation* qui comprend et exécute les programmes écrits en *Python*. Pour la plupart, vous allez donc découvrir en même temps le système `Sage` et des rudiments de Python. Cela fait beaucoup de choses à la fois. Bien des langages de programmation ont précédé Python, bien d'autres suivront, nous allons donc essayer à l'occasion de cet apprentissage, de nous familiariser avec des notions générales communes à la plupart des langages de programmation, comme, par exemple, les notions d'*identifiant*, d'*expression*, de *syntaxe*, d'*instruction*, de *variable*, *variables locales* ou *variables globales* etc.

2 Les trois premières instructions

Nous découvrirons pas à pas une partie des instructions `Sage` ou Python. Dans cette section nous faisons connaissance avec quelques instructions très simples. Dans le texte composant une instruction figurent des *mots*. Certains de ces mots ont une fonction réservée que vous découvrirez au fur et à mesure de votre apprentissage. Par exemple les mots `if, for, while`.

Définition *Un* identifiant *est un mot dont le premier caractère est une lettre, et les suivants des lettres, des chiffres ou le caractère '_', (le* blanc souligné*).*

Par exemple, `x, y, a, somme, x1, a_plus_b` sont des identifiants valides.

Première instruction : Evaluer un identifiant. A certains identifiants sont associés *une valeur*. Pour obtenir la valeur d'un identifiant nous allons utiliser notre première instruction, c'est l'instruction dont la syntaxe est

<IDENTIFIANT>

Essayons

```
cos
```

```
cos
```

Bien. L'identifiant `cos` a une valeur, ce n'est pas nous qui l'avons définie, c'est donc une valeur *prédéfinie* par le système `Sage`. Un deuxième essai :

```
v
```

```
Traceback (click to the left for traceback) ...
NameError : name 'v' is not defined
```

`Sage` a bien compris notre demande mais nous fait part de son insatisfaction : il est bien en peine d'évaluer l'expression réduite à `v`. l'identifiant `v` n'a pas de valeur, il n'a pas été défini.

Définition *En informatique une* variable *est un identifiant dont la* valeur *est susceptible d'être définie, ou redéfinie par le programmeur.*

`cos` est un identifiant prédéfini, `v` un identifiant qui n'a pas reçu de valeur. Encore un essai

```
if
```

```
Syntax Error : if
```

`Sage` n'a pas pu interpréter correctement l'instruction que nous lui avons soumise. Nous n'avons pas respecté la grammaire de Python. Le mot `if` est réservé à la construction de ce qu'on appelle *une instruction conditionelle*, construction qui doit respecter une syntaxe bien précise, que nous ne connaissons pas encore. L'identifiant `if` n'est pas un nom de variable.

Deuxième instruction : évaluer une expression. La notion d'expression nous est familière. Les expressions *les plus élémentaires* sont les *expressions atomiques*. L'expression 12 est une expression atomique. Un autre type d'expression atomique est une expression qui est de la forme <IDENTIFIANT>. A partir d'expressions déja construites on en construit d'autres, par exemple à l'aide des *opérateurs* d'addition et de multiplication `+,-,...` ou encore par applications de fonctions. Notre deuxième type d'instruction est l'instruction d'évaluation d'expression, dont la syntaxe est

<EXPRESSION>

Par exemple

```
2*5 + log(2)
```

```
log(2) + 10
```

Pour Sage la valeur de cette expression est 10+log(2). Il ne pousse pas plus loin l'évaluation, afin de rendre une *valeur exacte*, pas une valeur numérique approchée.

Remarque : Notre première instruction l'instruction <IDENTIFIANT> est une instruction <EXPRESSION> dans laquelle l'expression est l'expression atomique <IDENTIFIANT>.

Troisième instruction : L'affectation. Nous avons défini une variable informatique comme un identifiant dont la valeur peut être modifiée par l'exécution de certaines instructions. Quelles sont ces instructions ? La plus naturelle est l'*affectation*. En voici un exemple.

```
toto = 2*5 + log(2)
```

La forme générale de l'instruction d'affectation est

$$\text{<IDENTIFIANT>} = \text{<EXPRESSION>}$$

Dans la cellule ci-dessus l'identifiant est `toto`, l'expression est `2*5+log(2)`.
L'effet de cette instruction est double : l'*expression* `2*5+log(2)` est évaluée, et sa valeur, $10 + \log(2)$ *affectée* à l'identifiant `toto` dont elle devient la nouvelle valeur.

```
toto
```

```
log(2) + 10
```

3 Expressions symboliques

Sage est (entre autres) un outil de *calcul formel*. Nous l'utiliserons non seulement pour faire des calculs *numériques*, mais pour faire des calculs sur des *expressions mathématiques* formées à partir de nombres et de *symboles de variables*, comme par exemple l'expression

$$a^2 + t^2.$$

Cet *objet mathématique* est une *expression symbolique*, dans laquelle figurent deux *symboles* a et t. Demandons à Sage d'évaluer :

```
a^2+t^2
```

```
Traceback (click to the left for traceback)
... NameError : name 'a' is not defined
```

Que c'est-il passé ? Cette expression est *syntactiquement correcte*, c'est à dire bien formée. Sage reconnait la somme de a^2 et de t^2. Il est conduit à évaluer a^2 et donc a. C'est ici que les choses se gâtent. Pour nous les choses sont claires, ce `a` est l'*expression mathématique* réduite à la *variable symbolique* `a`. Pour Python ce `a` n'est qu'une variable qui n'a pas été définie. Remarquez ici le télescopage entre *deux sens* du mot variable, variable au sens *des langages de programmation* et variable au sens symbole de variable figurant dans une *expression mathématique*.

Pour satisfaire Python, il faut donc affecter à la variable `a` une valeur, qui dans ce cas est l'*expression mathématique symbolique* réduite au symbole `a`. On écrit pour cela

```
a = var('a')
```

La fonction `var` est la fonction de *création de variable symbolique*. En utilisant le raccourci `a,t = var('a,t')` pour `a = var('a') ; t = var('t')` nous écrivons donc

```
a,t = var('a,t')
f = a^2+t^2)
print f
```

Python est un langage de programmation par objets. Le langage `Sage` manipule des *objets* au sens informatique du terme. Un objet a des *attributs* (i.e. données internes) et des *méthodes* (i.e. fonctions qui peuvent lui être appliquées). Pour faire appel à une méthode d'un objet, la syntaxe est de la forme :

`objet.methode`<*paramètres éventuels*>

qui peut parfois s'écrire sous une forme plus classique

`methode`(*objet, paramètres éventuels)*

Dans la cellule ci dessous on applique à l'objet `f` défini ci-dessus la méthode `integrate` avec le paramètre `x` qui renvoie une primitive de $x^2 + a^2$ considérée comme fonction de x.

```
f.integrate(x)
```

```
a^2 x + 1/3 x^3
```

Le langage Python *connait-il donc a priori* ces objets mathématiques qu'il est capable d'intégrer, de dériver, etc. ? Bien sûr que non. Mais Python, en tant que *langage de programmation*, permet à l'utilisateur de définir de *nouvelles classes d'objets*, que cet utilisateur pourra ensuite utiliser. Les concepteurs de `Sage` ont *écrit pour nous toutes les définitions de ces classes d'objets mathématiques*, qui sont *lues au lancement du système*, et viennnent ainsi s'*ajouter aux notions de base* de Python. La notion d'*expression symbolique*, la fonction `var` de création de variable symbolique sont des exemples de tels ajouts.

Quelques manipulations sur les expressions symboliques L'une des opérations de base du calcul formel est l'opération de *substitution*. Considérons l'expression symbolique $a^2 + t^2$. On peut souhaiter remplacer le symbole a par la valeur 2 ou remplacer x par $t+1$ ou faire ces deux substitutions. On utilise pour cela la méthode `subst` qui est l'une des méthodes s'appliquant aux expressions symboliques.

```
a,t,x = var'a,t,x')
f = a^2 + t^2
print f.subst(t=2), f.subs(a=x+1), f.subst(t=2,a=x+1)
```

```
a^2 + 4, (x + 1)^2 + a^2, (x + 1)^2 + 4.
```

Si on est insatisfait par le troisième résultat on applique la méthode `expand`

```
f.subst(t=2,a=x+1).expand()
```

```
x^2 + 2*x + 5
```

La notion de type. Chaque valeur a un *type* bien défini, par exemple **Integer, Rational, Polynom**, etc. La fonction *type* s'utilise selon la syntaxe

$$type(\texttt{<EXPRESSION>})$$

Elle évalue l'expression et renvoie le type de la valeur obtenue.

```
type(13)
```

```
<type 'sage.rings.integer.Integer'>
```

```
x,a = var('x,a')
f = x^2+a^2
type(f)
```

```
<type 'sage.symbolic.expression.Expression'>
```

4 Etude à l'aide de Sage de fonctions numériques définies par des expressions symboliques

Vous vous proposez d'étudier la fonction f définie sur $\mathbb{R} \setminus \{0\}$ par $f(x) = x^2 + \dfrac{1}{x}$, plus particulièrement ses limites aux bornes du domaine de définition.

Première variante : utiliser une expression symbolique ordinaire. Commencez par affecter à `x` la variable de même nom. Affectez ensuite à la variable `y` (par exemple) l'expression $x^2 + \dfrac{1}{x}$, puis appliquez à l'expression symbolique y la méthode `limit`

```
var('x')
y = x^2 + 1/x
print limit(y, x=-oo), limit(y, x=+oo)
print limit(y, x=0, dir='below'), limit(y, x=0, dir='above')
```

```
+Infinity +Infinity
-Infinity +Infinity
```

Deuxième variante : utiliser un expressions symbolique paramétrée. Une autre possibilité est d'*affecter à une variable* `f` ce que nous conviendrons d'appeler une *expression symbolique paramétrée*. La syntaxe de cette affectation diffère de celle que nous avons apprise un peu plus haut, c'est à dire <IDENTIFIANT> = <EXPRESSION>. La syntaxe utilisée ici est

$$\texttt{<IDENTIFIANT>}\ (\texttt{<IDENTIFIANT>}) = \texttt{<EXPRESSION SYMBOLIQUE>}$$

Dans le cas de la fonction f étudiée ci dessus on écrira `f(x) = x^2 + 1/x`. Et ceci est bien une instruction dont l'*effet est une affectation*. Vérifions

```
f(x) = x^2 + 1/x
print f, type(f)
```

```
x |-> x^2 + 1/x    <type 'sage.symbolic.expression.Expression'>
```

La commodité la plus agréable offerte par cette classe des *expressions symboliques paramétrées* est la suivante. En écrivant

$$f(\mathtt{E})$$

on obtient toutes les expressions de la forme `E^2 + 1/E` où `E` est une expression symbolique quelconque. Exemple

```
f(cos(t))
```

```
cos(t)^2 + 1/cos(t)
```

Cela est beaucoup plus plaisant à écrire que `y = x^2 + 1/x ; y.subst(x=cos(t))`.
Avec cette notion le calcul des limites de f s'écrit

```
var('x')
f(x) = x^2 + 1/x
print limit(f(x), x=-oo), limit(f(x), x=+oo)
print limit(f(x), x=0, dir='below'), limit(f(x), x=0, dir='above')
```

Tracés de graphes de fonctions Vous utiliserez la fonction `plot` et ses nombreux paramètres. Un exemple simple est,

```
var('x')
f(x) = x^2 + 1/x
plot(f(x),xmin=-5, xmax=5, ymin=-10, ymax=10, detect_poles = true)
```

Pour comprendre l'utilité du paramètre `detect_poles` réévaluez la cellulle ci-dessus après avoir supprimé ce paramètre.
La fonction `parametric_plot` trace des graphes de courbes paramétrées. Informez vous à l'aide de la commande `parametric_plot ?`.
La fonction `plot` renvoie une valeur qui peut être affectée à une variable. Des valeurs de type `plot` s'ajoutent, la somme étant la superposition des graphes. Exemple

```
t,x = var('t,x')
f(x) = x^2 / (2*x - 1)
p1 = plot(f(x),xmin=-3, xmax=3, ymin=-10, ymax=10, detect_poles = true)
p2 = plot(x/2+1/4,xmin=-3, xmax=3, ymin=-10, ymax=10, color= 'red')
p3 = parametric_plot((1/2,t),(t,-10,10),color='green')
p1+p2+p3
```

5 Un peu de Python : les listes, l'instruction for

5.1 Type liste et fonction range

L'un des types de base de Python (et donc de `Sage`) est le type *Liste* en français ce qui se dit `list` en Python. La première façon de construire une liste est d'en énumérer les éléments à la main en écrivant, par exemple

```
[1, 'a', 3, 20, 3.14]
```

Voici deux autres constructions de listes ; essayez[1]

$$[1, 3..8, 10, 12..15], \quad 3 * [1, 2, 4]$$

[1] le constructeur [a..b] n'est pas défini dans Python, c'est un ajout de `Sage`.

La fonction Python `range` construit des listes d'entiers en *progression arithmétique*. Pour tout triplet d'entiers `a,b,h`

1. `range(b)` rend la liste des entiers `[0,1,...,b-1]`.
2. `range(a,b)` rend la liste `[a,a+1,...,b-1]`
3. `range(a,b,h)` rend la liste `[a,a+h,a+2h,...]` en s'arretant au dernier terme qui précède *strictement* `b`. Notez que `a,b,h` ne sont pas nécessairement positifs.

Familiarisez vous avec cette fonction en testant divers exemples.

5.2 L'instruction `for` :

L'instruction `for` est notre première *instruction d'itération*, c'est à dire une instruction qui a pour effet de provoquer la *répétition d'un bloc d'instructions*. L'instruction `for` est la plus sûre des instructions d'itération. La plus sûre en ce sens que son utilisation prête moins à l'ereur que l'instruction *while* qu'on rencontrera plus loin.
Partons de ce petit problème : Nous voulons calculer la somme des nombres impairs plus petits que 20 et affecter le résultat à la variable `som1`, et calculer aussi la somme des carrés de ces nombres et affecter le résultat à la variable som2.
Soit liste la liste `[1,3,5,7,11,13,15,17,19]`
Utilisant `Sage` comme une calculette nous pouvons affecter à `som1` et `som2` la valeur 0, puis, pour toutes les valeurs `x` de la liste `liste` répéter l'addition `som1 = som1 + x` et l'addition `som2 = som2 + x^2`. C'est très fastidieux. La bonne solution est

```
liste = range(1,20,2)
som1, som2 = 0, 0
for x in liste :
    som1 = som1 + x
    som2 = som2 + x^2
print 'La somme des impairs < 20 est ', som1
print 'La somme de leurs carrés est ', som2
```

Remarques :

1. Notez bien la présence du symbole `:` à la fin de la ligne `for ...`. Ce symbole est *indispensable*.
2. Remarquez aussi que les lignes `som1 = som1 + ...` et `som2 = som2 + ...` sont *indentées*, c'est à dire que leur premier caractère est placé dans une colonne située plus à droite que la colonne de la ligne `for`. Ceci est obligatoire c'est une des particularités qui contribuent à l'élégance de *Python*, le langage de base de `Sage`. A quoi sert cette indentation ? C'est elle qui définit la *portée* de l'instruction `for` c'est à dire le *bloc d'instructions* qui sera répété pour `x` parcourant `liste`. Toutes les instructions de ce bloc *doivent impérativement commencer sur la même colonne*.
3. Pour faire encore un peu de grammaire, la syntaxe de l'instruction `for` est donc

for <IDENTIFIANT> in <IDENTIFIANT> do :
<BLOC>

La fonction prédéfinie `add`. Dans l'exemple précédent on a utlisé une boucle `for` pour en illustrer le fonctionnement sur un exemple tres simple. La fonction prédéfinie `add` est beaucoup plus expéditive. Essayez

```
som1 = add([x for x in range(1,20,2)])
```

Une variante très utile de l'instruction for : Application d'une fonction à tous les éléments d'une liste.

Si `liste` est une liste et `f` une fonction l'instruction `[f(x) for x in liste]` construit la liste des valeur `f(x)`, `x` parcourant `liste`. Essayez avec

```
liste = [1,3,5,7,9]
[x^2 for x in liste]
```

6 Définir une fonction en Python

Considérons la fonction H définie par $H(n) = \sum_{k=1}^{n} \frac{1}{n}$. On peut calculer $H(20)$ en écrivant

```
N=20
som = 0.
for k in [1..N] :
    som = som+1/k
print 'som = ', som
```

Pour calculer une autre valeur de H, par exemple $H(100)$, nous sommes obligés de faire un copié-collé de la cellule ci-dessus, et dans la nouvelle cellule, de remplacer `N=20` par `N=100`. ou, avec la souris, ramener le curseur dans la cellule précédente, et recommencer le calcul après avoir remplacé 20 par 100. Il est beaucoup simple définir une fois pour toutes un nouvelle fonction, la fonction H, de la manière suivante :

```
def H(N) :
   som = 0.
   for k in [1..N] :
       som = som+1/k
   return som
```

1. La ligne `def H(N)` est appelé *l'entête* de la définition. Sa signification est la suivante : *Dorénavant la valeur associée au nom H sera une fonction. Pour obtenir la valeur $H(N)$ à partir de celle de N voici les calculs à effectuer.* Le bloc d'instructions suivant l'entête, et décrivant ces calculs est appelé *le corps de la définition.* Le nom `N` qui figure dans l'entête s'appelle un *paramètre formel.* C'est un nom qui n'a pas besoin d'avoir été défini.
2. Une fois la cellule évaluée, `Sage` connait la fonction H et celle ci peut être utilisée comme n'importe quelle fonction prédéfinie. Par exemple

   ```
   z = 100
   print H(20), H(factorial(7)), H(z)
   print 'ln(1000.) - H(1000.) = ', H(1000.) - log(1000.)
   ```

   Les instructions `H(20)`, `factorial(7)` et `H(z)` sont des *appels de la fonction H* avec les *paramètres effectifs* respectifs `20`, `7!` et `z`.
3. Remarquez l'instruction `return som`. Sa signification est la suivante : *Le calcul est terminé, la valeur la valeur de H, calculée en N, est la valeur de* `som`. L'instruction `return` n'est pas nécessairement la dernière instruction du corps.

4. Testez la fonction H. Puis remplacez la première instruction dans le corps de sa définition par `som = 0`, sans le point décimal. Que constatez vous ?

Les variables locales et les variables globales

Il est important pour éviter des déboires de bien saisir les notions de *variable locale* et de *variable globale*.

1. Les noms auquels ont été attribués une valeur en dehors de la définition d'une fonction sont les *noms globaux*. La cellule

```
s=0
for k in range(10) :
    s = s + k
print s,k
```

   utilise deux variables globales `s,k`. Quelles que soient les valeurs antérieurse de ces 2 variables, après l'évaluation de cette cellule, leurs valeurs respectives sont 45 et 9.

2. Toutes les variables qui apparaissent dans la définition d'une fonction, soit dans l'entête (les paramètres formels), soit dans le corps de la fonction, sont des *variables locales*. Dans la défintion de `H` ci dessus il y a 3 variables locales : `N`, `som`, `k`. Lorsque `Sage` calcule `H(20)` *tout ce passe comme si une autre feuille de travail était ouverte, une feuille de brouillon en quelques sorte*, utilisant ses propres variables `N,som,k`, et c'est dans cette feuille temporaire que se font les calculs. Une fois le résultat obtenu la feuille de brouillon est détruite, et le calcul n'a laissé aucune trace. Il a simplement rendu son résultat.

Après avoir défini votre fonction `H` évaluez

```
som = 0
print 'H(20) = ', H(20)
print som
```

Vous constatez que `som` a conservé sa valeur 0. Pourquoi : Parce que la variable `som` de la cellule ci-dessus est une variable *globale*. Ce n'est pas la même que la variable locale `som` qui figure dans la définition de `H`. Comme on vient de l'expliquer le calcul de $H(20)$ a été effectué sur une feuille de brouillon avec ses propres variables `N,som,k` sans affecter la variable globale `som`.

Une expression symbolique paramétrée n'est pas une fonction

Ce paragraphe est destiné à ceux d'entre vous déja à l'aise avec `Sage` et Python. Les autres le réseveront pour une seconde ou troisième lecture de ce document. Quelle différence y a-t'il entre les deux objets `f` et `g` suivants ? Demandons le à `Sage`

```
f(x) = x^2
def g(x) : return x^2
print type(f),  type(g)
```

<type 'sage.symbolic.expression.Expression'> <type 'function'>

L'affichage ci-dessus met en évidence une première diférence. f est une *expression symbolique* `Sage`, tandis que g est une *fonction* Python. Allons plus loin : cela change t'il quelque chose pour l'utilisateur ? Soit M, matrice (2,2) sur l'anneau des entiers modulo 10

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Affectons cette matrice à une variable `M` et calculons `g(M)`

```
M = Matrix(IntegerModRing(10),2,2,[1,2,3,4])
print M, g(M)
```

```
[1,2]    [7 0]
[3,4]    [5 2]
```

Bien. Essayons maintenant `f(M)`.

```
f(M)
```

```
Traceback (click to the left for traceback) ... TypeError :
no canonical coercion from Full MatrixSpace of 2 by 2
dense matrices over Ring of integers modulo 10 to Callable
function ring with arguments (x,)
```

`f` est une expression symbolique paramétrée, l'expression `f(E)` n'est définie que lorsque `E` est une expression symbolique, ce qui n'est pas le cas de la matrice `M`.

Quelques précisions à propos de types et de variables

Réalisons une expérience

```
print type(cos), cos(3*pi)
cos = 100 ; print 1+cos
```

```
<class 'sage.functions.trig.Function_cos'> -1
101
```

Nous avons défini une variable comme un identifiant dont la valeur peut être modifiée. L'identifiant `cos` serait-il un identifiant de variable Python ? Oui, puisqu'il est modifiable. Autrement dit `cos` est une variable `Sage` prédefine. Au démarrage du sytème sa valeur est la fonction trigonométrique bien connue. Mais nous pouvons la redéfinir. C'est ce que nous venons de faire ci-dessus.

Dans certains langages, comme **C**, par exemple, avant d'affecter des valeurs à une variable, on doit au préalable faire une *déclaration de type*, comme par exemple `int x ;` dont la signification est *à l'identifiant* `x` ne peuvent être associées que des valeurs entières.

Retenons donc que, en Python, ont *peut attribuer successivement à une même variable des valeurs de types différents.* Pour terminer cette section, une dernière révision. Les quatres instructions suivantes

```
u = 17
x = var('x')
y = x^2+ log(x)
f(x) = x^2+ log(x)
def g(x) : return x^2
h = g
```

ont pour effets, dans l'ordre

1. D'affecter à la variable u la valeur *entière* 17
2. D'affecter à la variable x la *variable symbolique* de même nom.
3. D'affecter à la variable y l'*expression symbolique* `x^2+ log(x)`
4. D'affecter à la variable f l'*expression symbolique de paramètrée* `x - > x^2+ log(x)`
5. D'affecter à la variable `u` la fonction Python qui à x associe x^2 chaque fois que ceci a un sens. Nous avons constaté plus haut que par exemple, `g(M)` rend ce qu'on espère lorsque M est une matrice (2,2) sur l'anneau des entiers modulo 10.
6. D'affecter à la variable `h` la même fonction Python que celle associée à `g`, celle qui a un objet `x` associe l'objet `x^2` pourvu que le type de cet objet se prête à cette opération.

7 L'instruction conditionelle `If`

L'instruction `for` est un exemple d' instruction d'itération, l'instruction `if` est une *instruction conditionnelle*, qui permet, selon qu'une propriété est réalisée ou non, d'exécuter un bloc d'instructions ou un autre bloc d'instructions. Avant de définir cette instruction il faut savoir ce qu'est une `expression booléenne`. C'est une expression qui prend soit la valeur Vrai, soit la valeur Faux. En Python ce sont les constantes prédéfinies `true, false`. Les expressions booléennes suivantes

```
x < 10,    x == y+3,    x != 0,    x % 3 == 1
```

sont vraies respectivement si et seulement si $x < 10$, $x = y + 3$, $x \neq 0$ et si $x \equiv 1 \bmod 3$ (parce que `x % 3` est le reste de la division de x par 3).

L'instruction conditionnelle `if` Sa syntaxe prend deux formes :

1.
```
if ExpressionBooleenne :
   Bloc
```

et le sens de cette instruction est le suivant : L'expression booléenne est d'abord évaluée. Si sa valeur est true les instructions qui figurent dans le bloc suivant les 2 points sont éxécutées. Sinon le bloc n'est pas éxécuté et l'execution de `if` est terminée.

2.
```
if ExpressionBooleenne :
   Bloc1
else :
   Bloc2
```

et le sens de cette instruction est le suivant : L'expression booléenne est d'abord évaluée. Si sa valeur est true, l'exécution de cette instruction `if` se termine par l'évaluation du premier bloc d'instructions. Sinon c'est la second bloc qui est exécuté.

Par exemple, pour afficher, s'il en existe, les entiers n dont l'écriture décimale comporte 4 chiffres, et tels que les 4 derniers chiffres de l'écriture de n^2 soient les mêmes (dans le même ordre) que les 4 chiffres de n on évaluera

```
for n in range(1000,10000] :
    if n*n % 10000 == n :
         print 'n = ', n, ' n*n = ', n*n
```

Si l'on veut construire la liste de ces entiers on écrira,

```
[n for n in range(1000,10000) if n^2 % 10000 = n]
```

Remarque : Remarquez au passage cette forme plus élaborée de l'instruction `for` que celle donnée dans la section 5.

Exercice 1 La *fonction de Collatz* est la fonction collatz définie sur $\mathbb{N}$ par

$$\text{collatz}(n) = \begin{cases} \dfrac{n}{2} & \text{si } n \text{ est pair} \\ \dfrac{3n+1}{2} & \text{sinon.} \end{cases}$$

1. Définir la fonction collatz en `Sage` en complétant la cellule,

```
def collatz(n) :
    if n % 2 == 0 :
         return ...
    else :
```

2. Afficher les couples $(n, \text{collatz}(n))$ pour $n = 1, 2, \ldots, 15$.

8 L'instruction while

La boucle `for` suppose de connaitre à l'avance la liste des objets `x` sur laquelle se fera l'intération. Il est fréquent qu'on veuille répéter un même calcul un nombre de fois a priori inconnu, mais tant qu'une certaine condition est satisfaite. On utilise pour cela l'instruction `while` dont la syntaxe est

```
while UneExpressionBooleenne :
      Bloc
```

L'effet de cette instruction est :

1. L'expression booléenne est évaluée.
2. Si sa valeur est false l'instruction `while` se termine immédiatement.
3. Sinon le bloc constituant le corps du while est exécuté, on évalue à nouveau l'expression booléenne, et ainsi de suite, l'exécution de l'instruction `while` se terminant quand l'évaluation de l'expression booléenne donne la valeur false (peut être jamais donc).

La conjecture de Collatz, ou d'Ulam ou encore de Syracuse Soit $x \geq 1$, entier et la suite définie par $u_0 = x$, et $u_{n+1} = \text{collatz}(u_n)$. Est-il vrai qu'il existe une entier k tel que $u_k = 1$? Autrement dit : est-il vrai que pour tout $x \geq 1$, il existe un entier k tel que $\text{collatz}^k(x) = 1$? Ce problème n'est toujours pas résolu.

Exercice 2

1. Pour $x = 23$ aficher les 15 premiers termes de cette suite. Quelle est la plus petite valeur de k telle que collatz$^k(23) = 1$?
2. Ecrire une fonction `Ulam(x)` qui calcule les u_n lorsque $u_0 = x$, et se termine que lorsque la valeur 1 est atteinte. Si $x \leq 1$ il n'y a rien à faire, sinon tant que $x > 1$ on le remplace par collatz(x) et on recommence jusqu'à obtenir $x <= 1$. Il suffit don d'écrire

```
def Ulam(x) :
   while x > 1 :
         x = collatz(x)
         print 'x = ', x
```

3. La fonction `Ulam` définie ci-dessus ne renvoie pas de valeur. Elle se contente d'afficher des lignes à l'écran. Modifiez cette définition en ajoutant une variable locale `k` initialisée à 0 et incrémentée de 1 à chaque appel de `colatz`. La fonction `Ulam(x)` rendra cette fois la valeur terminale de la variable `k` c'est à dire le plus petit entier tel que collatz$^k(x) = 1$.

La fonction `Ulam` définie ci-dessus est très simple. Aujourd'hui personne n'est capable de prouver que son evaluation se termine pour toute valeur de x. Cet exemple montre que la correction des programmes est un problème difficile.

Une autre application de l'instruction while. Soit f une fonction réelle continue sur $I = [a, b]$ telle que $f(a)f(b) < 0$. Alors f admet au moins un zéro dans l'intervalle $]a, b[$ (pourquoi ?). On se donne un nombre $\varepsilon > 0$ et on se propose de déterminer un intervalle J, de longueur plus petite que ε, qui contient un zéro de f.

1. Si la longueur $b - a$ est plus petite que ε il n'y a rien à faire, la réponse est $[a, b]$.
2. Si $b - a > \varepsilon$. Soit c le milieu de $[a, b]$. Si par extraordinaire $f(c) = 0$ on renvoie l'intervalle fermé réduit à c et tout est terminé. Si $f(a)f(c) > 0$ c'est que $f(b)f(c) < 0$, on remplace l'intervalle $[a, b]$ par l'intervalle $[b, c]$, sinon on remplace $[a, b]$ par $[a, c]$. Et on recomence tant que $b - a > \varepsilon$.

```
def dichotomie(f,a,b,eps) :
    while b-a > eps :
          c = (a+b)/2
          if f(c) == 0
              return [c,c]
          if f(a)f(c) > 0 :
              a=c
          else :
              b=c
    return [a,b]
```

Exercice 3 Démontrez que l'équation $x = \cos x$ admet une unique racine réelle, et que cette racine appartient à $[0, 1]$. Résolvez par dichotomie à 10^{-8} près l'équation $x = \cos x$. Vérifiez votre résultat à l'aide de la fonction prédéfinie `find_root`.

Exercice 4 Démontrez que la suite (x_n) définie par $x_0 = 0$ et $x_{n+1} = \cos(x_n)$ est convergente et que sa limite est solution de l'équation $x = \cos x$. Démontrez de plus que x_n est une valeur approchée par défaut de x lorsque n est pair, et une valeur approchée par excès lorsque n est impair. Calculez au moyen d'une boucle `for` les 30 premiers termes de la suite (x_n) et retrouvez ainsi une valeur approchée de la solution de $x = \cos(x)$.

9 Une définition de fonction peut être récursive

Nous avons vu que la syntaxe de définiton d'une fonction est la suivante

```
def f(x) :
    corps_de_fonction
```

On pourrait penser qu'aucun appel de la fonction `f` ne peut figurer dans l'une des instructions constituant le corps de la fonction, parce que cela n'a pas de sens d'utiliser un mot pour définir ce mot. Et pourtant ceci vous est deja familier, dans le cas des suites par exemple. Vous avez deja rencontré la définition suivante : *Soit la suite* (u_n) *définie par* $u_0 = 1$ *et, pour* $n \geq 1$, $u_n = \cos(u_{n-1})$. Cette définition par récurrence de la fonction u s'écrit tout naturellement en `Sage`.

```
def u(n) :
    if n == 0 :
         return 1.0
    else :
         return cos(u(n-1))
```

Exercice 5

1. Affichez les 10 premiers termes de la suite $(u(n))$,

   ```
   [u(n) for n in [0..9]]
   ```

2. Que se passe t'il si dans la défintion de `u` vous remplacez `return 1.0` par `return 1` ?

Remarquons que pour obtenir la suite des u_k pour tous les k de 0 à n il est un peu ridicule d'utiliser la fonction u car, pour chaque valeur de k, le calcul de $u(k)$ nécessite les calcul des $u(j)$ pour $j \leq k$. La fonction u n'a d'intéret que pour calculer une valeur *isolée* $u(n)$.
En mathématique on parle de *définition par récurrence*, en informatique on dit plus volontiers *définition récursive*.
N'hésitez pas à utiliser les définitions récursives. C'est une technique qui permet d'écrire rapidement des programmes corrects.

Exercice 6 Écrivez une définition récursive de la fonction de Ulam de l'exercice 1

```
def UlamRec(N) :
    if N==1 :
       return 0
    else :
       return ...
```

10 La structure de dictionnaire

Le type *Dictionnaire*, parfois dénommé *table associative* est une notions essentielle en algorithmique. Cette structure modélise la notion d'application définie sur un ensemble fini. Un *dictionnaire* `dic` est une application **dic** : $K \longrightarrow E$ définie sur un ensemble fini K et à valeurs dans un ensemble quelconque E. Les éléments de K sont appelées les *clefs*, La valeur dic(k) est souvent appellée l'information associée à la clef k. Un couple (k, dic[k]) est une *entrée* du dictionnaire.
Un dictionnaire au sens usuel du terme est un dictionnaire au sens informatique. Les clefs sont les mots définis dans le dictionnaire et l'information associée à un mot est la définiton de ce mot donnée par le dictionnaire.

Voyons comment affecter à une variable une valeur de type dictionnaire. Affectons à la variable `Dic` un dictionnaire vide,

```
Dic = {}; type(Dic)
```

```
<type 'dict'>
```

Si k est une clef du dictionnaire, la valeur de `Dic[k]` est l'information associée.

```
Dic[10]
```

```
Traceback (click to the left for traceback)
...
KeyError : 10
```

C'était prévisible, `Dic` est un dictionnaire vide. Ajoutons deux entrées au dictionnaire `Dic`, de clefs respectives 10 et 100.

```
Dic[10] = 100; Dic[pi] = "Surface du cercle de rayon 1"
```

```
[10, pi]
```

Les méthodes les plus utiles sont `has_key`, `keys`, `items` et `values`.
L'appel `Dic.has_key(x)` rend true ou false selon que x est où n'est pas une clef. Les appels

```
Dic.keys(), Dic.values(), Dic.items()
```

renvoient respectivement la liste de clefs présentes dans le dictionnaire, la liste des informations associées à toutes les clefs et enfin la liste des entrées du dictionnaire `Dic`.
On est pas obligé d'initialiser un dictionnaire en lui affectant la valeur vide. L'instruction

```
f = {1 :10, 5 :100, pi : "Surface du cercle de rayon 1"}
```

affecte à la variable `f` le dictionnaire associant aux trois entrées 1, 10 et `pi` les valeurs respectives 10, 100, `"Surface du cercle de rayon 1"`.

Exercice 7 La *paradoxe des anniversaires* est un résultat combinatoire simple et amusant, qui joue un grand rôle en algorithmique, en particulier dans le domaine de la cryptographie. Soit E un ensemble de cardinal n. On se donne un entier k et on tire k fois de suite, avec remise, et avec la probabilité uniforme, un élément de E. On obtient ainsi une suite $(x_1, x_2, \ldots, x_k)$ d'éléments de E. On se propose de calculer la probabilité p_k de l'évenement *tous les x_i sont distincts.*

1. Quel est le cardinal de E^k ?

2. Combien y a-t-il de k uplets $(x_1, x_2, \ldots, x_k) \in E^k$ dont les composantes $x_1, x_2, \ldots, x_k$ sont deux à deux distinctes ?
3. En déduire que la probabilité p_k est donnée par $p_k = \prod_{i=1}^{k-1} \left(1 - \frac{i}{n}\right)$.
4. Démontrez que si 23 personnes sont réunies dans une salle la probabilité que deux au moins d'entre elles aient leur anniversaire le même jour, est plus grande que 1/2.
5. Réalisons l'expérience suivante : on choisit un entier N (par exemple $N = 365$) puis on tire au hasard des valeurs x dans l'ensemble $\{1, 2, \ldots, N\}$. L'expérience se terminera la première fois que le tirage donnera une valeur déja obtenue.

   La solution la plus naturelle est d'utiliser un *dictionnaire*, de nom `rang`. initialisé à vide. Un entier x est une clef du dictionnaire avec `rang[x] = k` si et seulement si la valeur x a été obtenue pour la première fois au k^{ieme} tirage.

   Cela conduit à l'écriture suivante :

```
def Paradox(N) :
 rang = {}; k = 1
 while true :
   x = randint(1,N)
   if rang.has_key(x) :
    print 'k=', k, ' donne', x, ' deja obtenu au rang ',rang[x]
    return
   rang[x] = k
   k = k+1
```

11 Exercices complémentaires

Exercice 8

1. Que pensez vous de la suite (x_n) définie par $x_0 = 1/3$, et $x_{n+1} = 4x_n - 1$?
2. Lancez les instructions

```
x=1/3
for n in range(30):
        x=4*x-1
print " x = " , x
```

3. Refaites le même calcul en remplaçant simplement la première ligne par

```
x=1.0/3
```

4. Que s'est il passé ?

Exercice 9 (Le jeu des tours de Hanoï.) Le matériel est composé de trois plots verticaux, nommés `A,B,C`, et de N disques percés en leur centre d'on orifice de diamètre égal au diamètre des plots `A,B,C`. Le disque numéro k est de diamètre k. Initialement les disques sont tous empilés sur le plot `A`, par ordre de taille décroissante, c'est à dire, de bas jusqu'en haut, $N, N-1, \ldots, 2, 1$. Le but du jeu est de déplacer tous les disques sur le plot `C`. La règle

du jeu est : On prend le disque le plus haut placé sur l'un des plots `A,B,C`, et on le poser au sommet de l'un des autres plots, pourvu qu'on ne pose jamais un disque sur un disque plus petit que lui. Nous appelerons `Hanoi(A,B,C,N)` la suite des tansferts à effectuer pour résoudre ce problème.
Soit $N \geq 2$. Supposons que l'on sache résoudre le problème à l'ordre $N-1$. Vu la règle du jeu, on ne pourra déplacer le plus gros disque de `A` vers `C` qu'après avoir déplacé les $N-1$ autres disques depuis `A` vers `B`.

1. On commence done par transférer les $N-1$ plus petits disques de `A` vers `B`.
2. Le plus gros disque, celui de numéro N est alors tout seul sur le plot `A`, tandis que le plot `C` est vide. On transfère donc le disque N de `A` vers `C`.
3. Le plot `A` est maintenant vide. Il ne reste plus qu'à transférer les $N-1$ plus petits disques de `B` vers `C`.

Autrement dit résoudre `Hanoi(A,B,C,N)` avec $N \geq 2$ c'est résoudre `Hanoi(A,C,B,N-1)`, puis transférez le disque N de `A` vers `C`, et enfin résoudre `Hanoi(B,A,C,N-1)`.

1. Complétez la cellule suivante pour définir la fonction `Hanoi(A,B,C,N)` qui ne revoie pas de résultats mais qui affiche la suite des déplacements pour transférer N disques de `A` vers `C`, en utilisant le plot intermédiaire `C`.

```
def Hanoi(A,B,C,N) :
  if N==1 :
       print 'De ', A, ' vers ', C
  else :
```

2. Combien de transferts faut il effectuer pour résoudre `Hanoi(A,B,C,N)` ?
