INSTITUTO TECNOLOGICO DE SONORA

Manual de SAGE para principiantes

Traducción por Héctor Yanajara Parra

Instituto Tecnológico de Sonora
Uso y Aprendizaje de SAGE

Traduccion desarrollada por Héctor Yanajara Parra
Estudiante de la carrera Ingeniero en Electrónica
Contacto a: Yanajarahp@hotmail.com

Tabla de contenidos

1 Prefacio

1.1 Dedicatoria

Este libro esta dedicado a Steve Yegge y su entrada de blog “Math Every Day” (http://steve.yegge.googlepages.com/math-every-day).

1.2 Agradecimientos

La siguientes personas han proporcionado retroalimentación para este libro (si olvide incluir su nombre en esta lista, por favor mándenme un correo a ted.kosan en gmail.com):

Dave Dobbs
David Joyner
Greg Landweber
Jeremy Pedersen
William Stein
Steve Vonn
Joe Wetherell

*Un agradecimiento especial a Ted Kosan por permitirme traducir este libro

1.2 Grupo de soporte

El grupo de soporte para este libro es llamado **sage-support** y puede ser contactado en: http://groups.google.com./groups/sage-support . Por favor coloquen “[Newbies Book]” en el titulo de su email cuando posteen en este grupo.

2 Introducción

SAGE es un entorno de cálculos matemáticos (MCE – Mathematics computing enviroment) de código abierto para llevar a cabo cálculos algebraicos, simbólicos y numéricos. Los entornos de cálculos matemáticos son complejos y requieren una gran cantidad de tiempo y esfuerzo para volverse hábil utilizando uno. Sin embargo, la cantidad de poder que este tipo de software proporciona al usuario vale muy bien el esfuerzo requerido para aprenderlo. A un principiante le tomara un rato volverse experto en el uso de SAGE, pero afortunadamente uno no necesita ser un experto en SAGE para comenzar a utilizarlo en la resolución de problemas.

2.1 ¿Qué es un entorno de cálculos matemáticos?

Un entorno de cálculos matemáticos es un grupo de programas computacionales capaces de llevar a cabo automáticamente un amplio rango de algoritmos de cálculo matemáticos. Los algoritmos de cálculo existen para casi todas las áreas de las matemáticas, y nuevos algoritmos son desarrollados todo el tiempo.

Un gran número de entornos de cálculos matemáticos han sido creados desde los 60's y la siguiente lista contiene algunos de los más populares:

http://en.wikipedia.org/wiki/Comparison_of_computer_algebra_systems

Algunos entornos están altamente especializados y otros son de propósito general. Algunos permiten que los datos matemáticos sean introducidos en la forma tradicional (que es como los encontramos en la mayoría de los libros de texto), otros son capaces de desplegar datos matemáticos en la forma tradicional pero necesitan que estos datos sean introducidos como texto, y otros solamente son capaces de mostrar y leer los datos como texto.

Como ejemplo de la diferencia entre la forma matemática normal y la forma textual, aquí se muestra una fórmula en la forma tradicional:

$$A = x^2 + 4.h.x$$

y esta es la misma fórmula en forma de texto:

```
A = = x^2 + 4*h*x
```

La mayoría de los entornos de cálculo matemático contienen algún tipo de lenguaje de programación de alto nivel orientado a las matemáticas. Esto permite que los programas de cómputo sean desarrollados para tener acceso a los algoritmos matemáticos que están incluidos en el entorno. Algunos de estos lenguajes de programación orientados a las matemáticas fueron creados específicamente para el entorno en el que trabajan, mientras que otros son construidos en torno a un lenguaje de programación existente.

Algunos entornos de cálculos matemáticos son de marca registrada y necesitan ser comprados mientras que otros son de código libre, y gratuitos. Ambos tipos de entornos poseen esencialmente capacidades similares, pero usualmente difieren en otras áreas.

Los entornos de marca registrada tienden a ser más detallados que los de código abierto y comúnmente tienen interfaces de usuario que hacen relativamente fácil la introducción y manipulación de datos matemáticos en forma tradicional. Sin embargo, estos entornos también tienen sus desventajas. Una es que siempre esta la posibilidad de que la compañía que lo posee salga del negocio y esto puede ocasionar que el entorno no este disponible para futuro uso. Otra desventaja es que los usuarios no pueden aumentar un entorno de marca registrada debido a que el código fuente del entorno no esta disponible para los usuarios.

Los entornos de cálculos matemáticos de código libre usualmente no tienen interfaces de usuario graficas (GUI), pero sus interfaces de usuario son adecuadas para la mayoría de los propósitos y el código fuente del entorno siempre estará disponible para cualquier persona que lo quiera. Esto significa que la gente puede usar el entorno por tanto tiempo como haya interés en el y también pueden mejorarlo a su gusto.

2.2 ¿Qué es SAGE?

SAGE (iniciales de Software for Algebra and Geometry Experimentation – Software para Experimentación de Algebra y Geometría) es un entorno de cálculos matemáticos que introduce datos matemáticos en forma textual y los despliega en forma textual o tradicional. Mientras que la mayor parte de los entornos de cálculo matemático son entidades independientes, SAGE provee algunos algoritmos por si mismo y otros los toma de otros entornos de cálculo matemático. Esta estrategia le permite a SAGE proveer el poder de múltiples entornos de cálculo matemáticos dentro de una arquitectura capaz de evolucionar para satisfacer futuras necesidades.

SAGE esta escrito en el poderoso y muy popular lenguaje de programación Python y el lenguaje de programación orientado a las matemáticas que SAGE hace disponible a los usuarios es una extensión de Python. Esto significa que los usuarios expertos en SAGE deben ser también expertos programadores en Python. Algo del conocimiento del lenguaje de programación Python es tan decisivo para utilizar con éxito SAGE que el nivel de conocimiento de Python de un usuario puede ser utilizado para ayudar a determinar su nivel o habilidad en SAGE. (ver tabla 1)

Nivel	Conocimiento
Experto en SAGE	Conoce muy bien Python y SAGE.
Novato en SAGE	Conoce Python pero solo ha usado SAGE por un corto tiempo.
Principiante en SAGE	No conoce Python pero ha conocido al menos 1 lenguaje de programación
Programador principiante	No sabe como funciona una computadora y nunca ha programado antes.

Tabla 1: Niveles de experiencia en SAGE

Este libro es para principiantes en SAGE. Asume que el lector ha conocido por lo menos 1 lenguaje de programación, pero nunca ha programado en Python (si su conocimiento acerca de cómo funciona la programación computacional necesita refrescarse, podría querer leer la sección de Fundamentos de la Computación de este libro). Este libro le enseñara suficiente lenguaje de programación Python para comenzar a resolver problemas con SAGE. Le ayudara a convertirse en un Novato en SAGE, pero necesitará aprender Python de libros que estén dedicados a el antes de convertirse en un experto en SAGE.

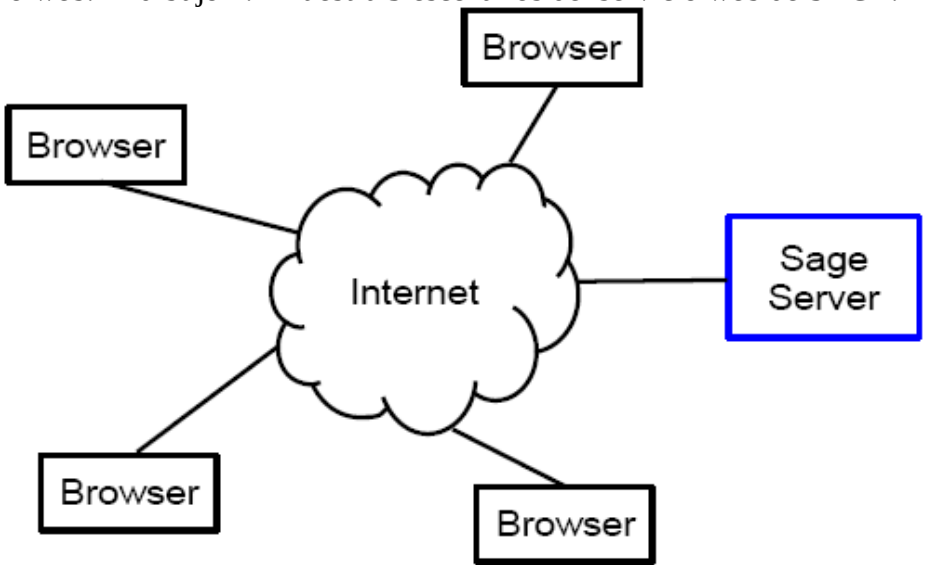
Su usted es un Programador principiante, este libro probablemente sea muy avanzado para usted. He escrito una serie de libros gratuitos llamados *Las Series de Programación del Profesor y Pat* (http://professorandpat.org) y están diseñados para programadores principiantes. Si usted es un programador principiante y esta interesado en aprender a utilizar SAGE, podría estar interesado en trabajar con los libros de programación del Profesor y Pat primero y después volver a este libro cuando haya terminado con ellos.

El sitio web de SAGE (www.sagemath.org) contiene más información sobre SAGE junto con otros recursos de SAGE.

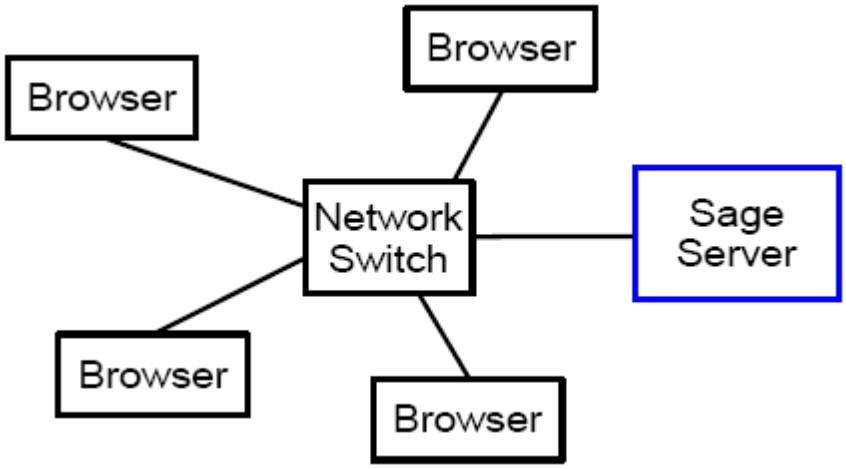
2.3 Acceso a SAGE como un servicio web

Las formas en que SAGE puede ser utilizado son tan flexibles como su arquitectura. La mayoría de los que inician con SAGE, sin embargo, primero lo usaran como un servicio web donde ingresan mediante un navegador de Internet. Cualquier copia de SAGE puede ser configurada para proveer este servicio web. El dibujo 2.1 muestra 3 escenarios del servicio web de SAGE.

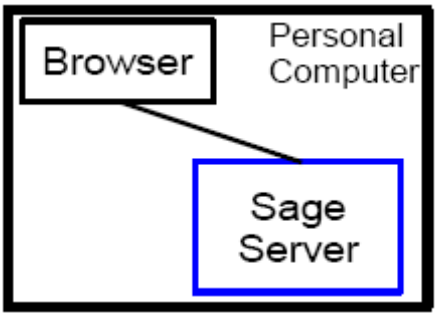
Escenario1:
Servicio web SAGE
Disponible en
Internet.



Escenario 2:
Servicio web SAGE
disponible en una
red de área local



Escenario 3:
Servicio web SAGE
disponible en la misma
computadora en la que
corre el navegador.



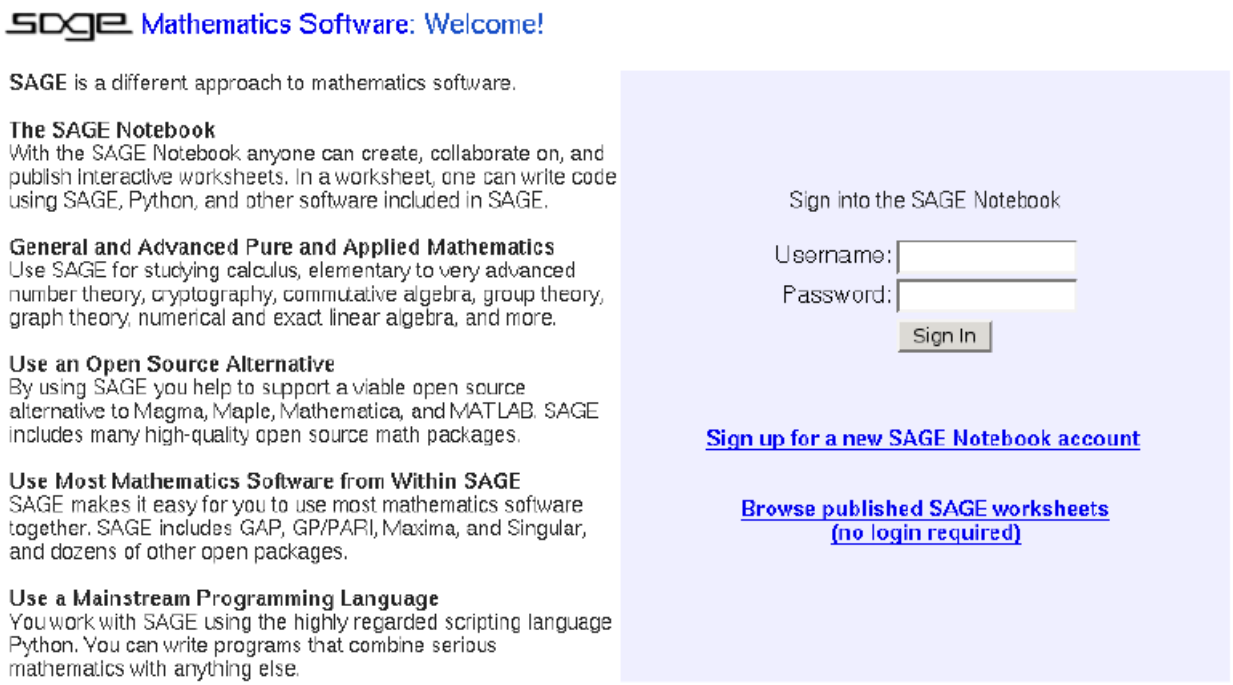
Dibujo 2.1: Tres escenarios del servicio web de SAGE.

2.3.1 Acceso a SAGE como un servicio web usando el escenario 1

SAGE actualmente funciona mejor con el navegador de red Firefox y su usted todavía no lo tiene instalado en su computadora, puede obtenerlo en http://mozilla.com/firefox.

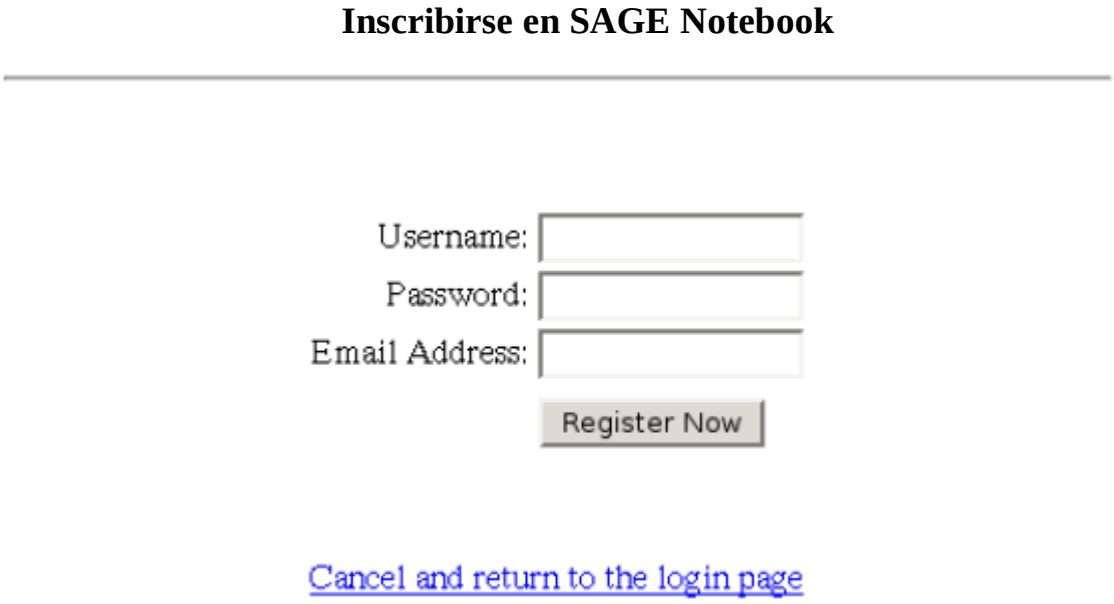
El equipo de desarrollo de SAGE provee un servicio de red público en (http://sagenb.com) y este servicio también puede ser accesado desde la parte superior de la página principal de SAGE. Ahora veremos los pasos necesarios para abrir una cuenta en este servicio de red público de SAGE.

Abrir una ventana del explorador Firefox e introducir la siguiente dirección en la barra de URL:
http://sagenb.com
El servicio entonces mostrara una pagina de bienvenida (ver dibujo 2.2)



Dibujo 2.2: Página de bienvenida de SAGE

El servicio web de SAGE es llamado SAGE **Notebook (libro de apuntes, cuaderno)** por que simula el tipo de cuaderno que los matemáticos tradicionalmente usan para llevar a cabo cálculos matemáticos. Antes de acceder al servicio, primero se debe registrar para una cuenta. Seleccione el enlace de **Sign up for a new SAGE Notebook account** (registrarse para una nueva cuenta de SAGE Notebook). (ver dibujo 2.3)



Dibujo 2.3: Pagina de registro.

Introduzca un nombre de usuario (username) y contraseña (password) en los cuadros de texto y posteriormente presione el botón **Register Now.** Una página será entonces mostrada que indica que la información de registro fue recibida y que un mensaje de confirmación fue enviado a la dirección de correo suministrada.

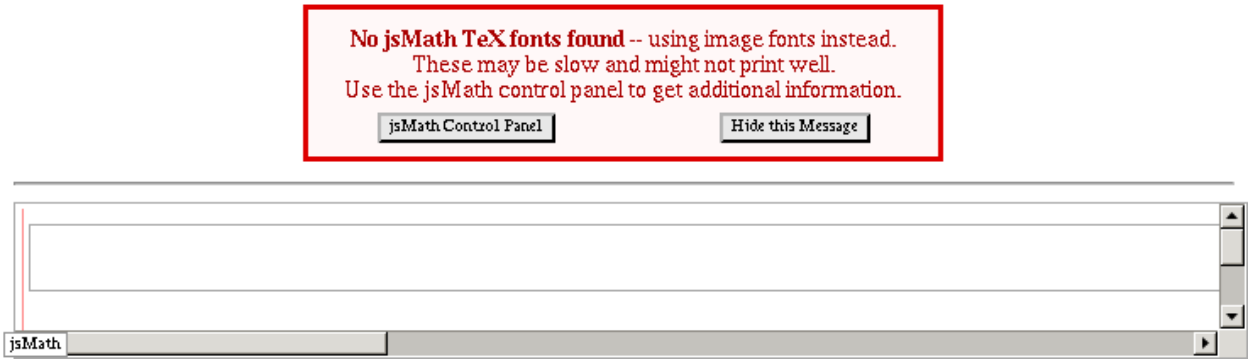
Abra este correo y seleccione el enlace que contiene. Esto completara el proceso de registro y entonces podrá volver a la pagina de bienvenida e ingresar.

Después de ingresar exitosamente a su cuenta Notebook, una página de manejo de hojas de trabajo será mostrada. (ver dibujo 2.4)

Dibujo 2.4: Pagina de manejo de hojas de trabajo.

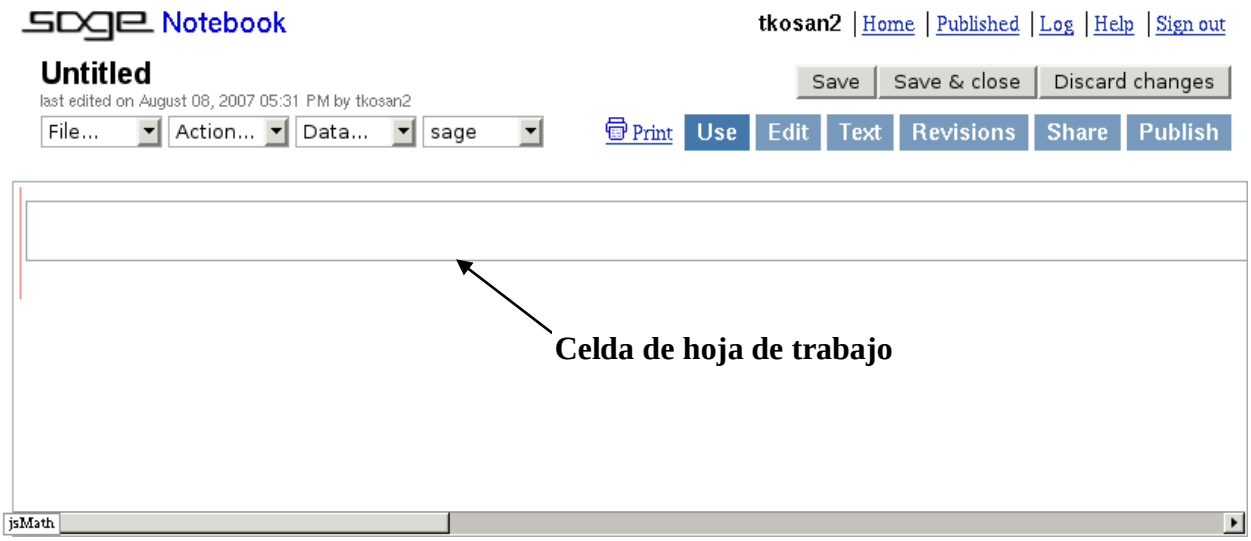
Los cuadernos de matemáticas físicos contienen hojas de trabajo y por lo tanto el cuaderno virtual de SAGE también. La página de manejo de hojas de trabajo permite crearlas, borrarlas, publicarlas en Internet, etc. Como nuestra cuenta acaba de ser creada, no contiene ninguna hoja de trabajo todavía.

Creamos una nueva hoja de trabajo seleccionando el enlace de **New Worksheet**. Una hoja de trabajo puede usar tanto fuentes especiales de matemáticas para desplegar los datos en la forma matemática tradicional o puede utilizar imágenes de estas fuentes. Si la computadora en la que usted esta trabajando no tiene instaladas fuentes de matemáticas, la hoja de trabajo desplegara un mensaje que indica que utilizara sus fuentes de imagen incorporadas como alternativa. (Ver dibujo 2.5)



Dibujo 2.5: Alerta de falta de fuentes de jsMath

Las fuentes de imagen no son tan claras como las fuentes normales de matemáticas, pero son adecuadas para la mayoría de los propósitos. Mas adelante usted puede instalar fuentes de matemáticas en su computadora si así lo desea, pero por el momento solo presione el botón **Hide this message** (esconder este mensaje) y una página con una hoja de trabajo en blanco será mostrada. (ver dibujo 2.6)



Dibujo 2.6: Hoja de trabajo en blanco.

Las hojas de trabajo contienen 1 o mas celdas las cuales son usadas para introducir el código fuente que será ejecutado por SAGE. Las celdas tienen rectángulos dibujados alrededor de ellas como se muestra en el dibujo 2.6 y son capaces de aumentar su tamaño conforme se introduzca más texto en ellas. Cuando se crea una hoja de trabajo, una celda inicial se coloca en la parte superior de su área de trabajo y aquí es donde usted normalmente comenzará a introducir texto.

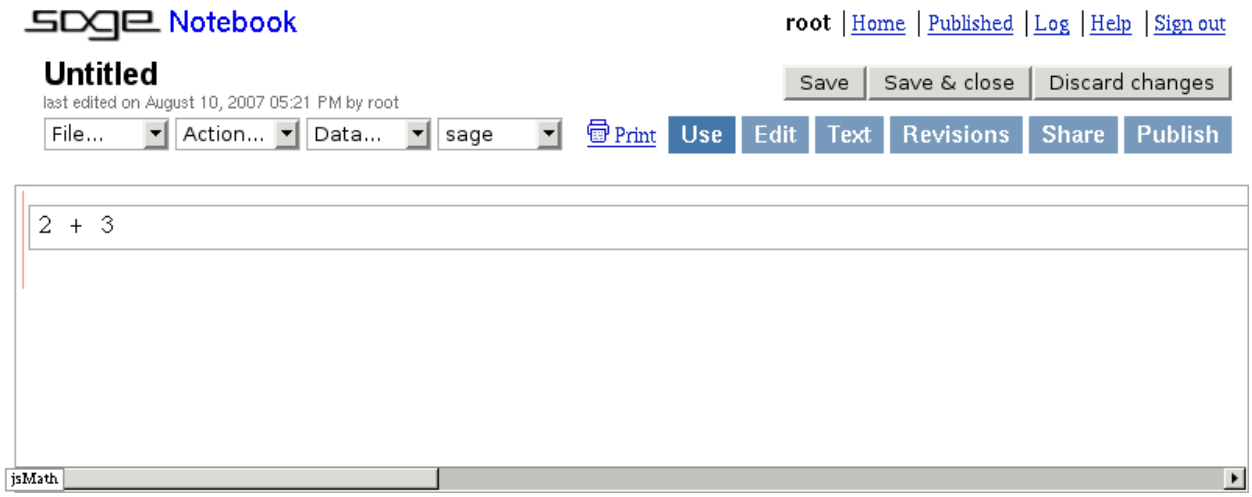
2.4 Introduciendo código fuente en una celda de SAGE

Comencemos a explorar SAGE utilizándolo como una calculadora sencilla. Coloque el cursor del mouse dentro de la celda que esta en la parte superior de su hoja de trabajo. Note que el cursor automáticamente es colocado contra el lado izquierdo de una celda nueva. Usted debe siempre iniciar cada línea de código fuente de SAGE hacia el lado izquierdo de la celda sin dejar espacios (a menos que se le pida hacerlo de otra manera).

Escriba el siguiente texto, pero no presione la tecla enter:

2+3

Su hoja de trabajo deberá verse ahora como el dibujo 2.7.



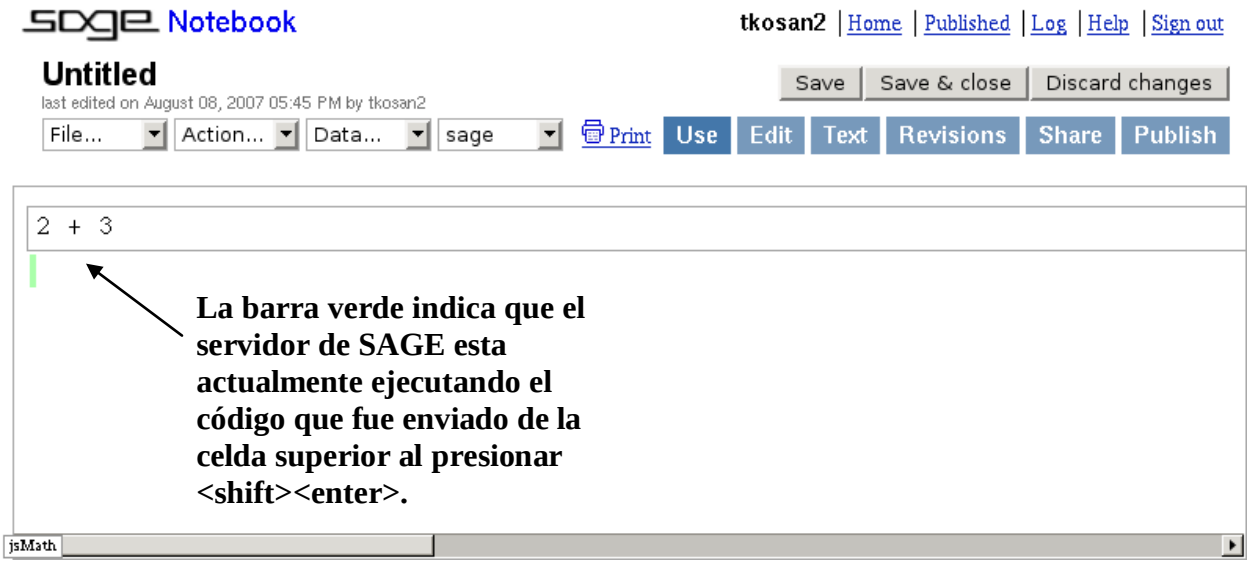
Dibujo 2.7: Introduciendo texto en una celda.

Ahora, tiene 2 opciones. Puede presionar la tecla enter o puede sostener la tecla shift y presionar enter. Si solamente presiona enter, la celda se expandirá y el cursor saltara a la línea siguiente para que usted pueda seguir introduciendo código fuente.

Sin embargo, si presiona shift y enter, la hoja de trabajo tomará todo el código fuente que ha sido introducido dentro de la celda y lo enviará al servidor SAGE por medio de la red para que el servidor pueda ejecutar el código. Cuando SAGE recibe un código fuente para ser ejecutado, primero lo procesará utilizando un software llamado **SAGE preprocessor (preprocesador).** Este convierte el código fuente de SAGE a un código fuente de Python, para que de esta manera pueda ser ejecutado utilizando el entorno Python en el cual está desarrollado SAGE.

El código fuente convertido se pasa entonces al entorno Python donde es compilado a una forma especial de lenguaje máquina llamada **Python bytecode** (código byte Python). El código byte es entonces ejecutado por un programa que emula el hardware del CPU y este programa es llamado **Python interpreter** (interprete Python).

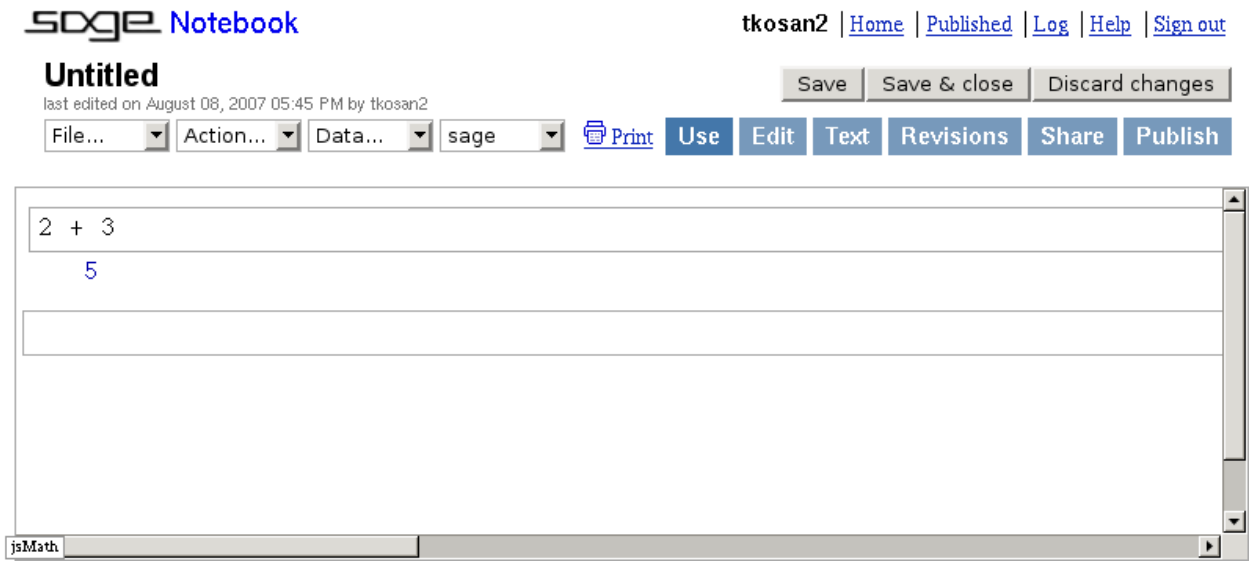
Algunas veces el servidor es capaz de ejecutar el código rápidamente y otras veces tomará tiempo. Mientras el código es ejecutado por el servidor, la hoja de trabajo desplegará una pequeña barra vertical verde debajo de la celda hacia el lado izquierdo de la ventana como se muestra en el dibujo 2.8.



Dibujo 2.8: Ejecutando el texto en una celda.

Cuando el servidor ha terminado de ejecutar el código fuente, la barra verde desaparecerá. Si se generó un resultado que pueda ser mostrado, el resultado es enviado de regreso a la hoja de trabajo y entonces la hoja de trabajo lo muestra en el área que esta directamente debajo de la celda de donde se envió la petición.

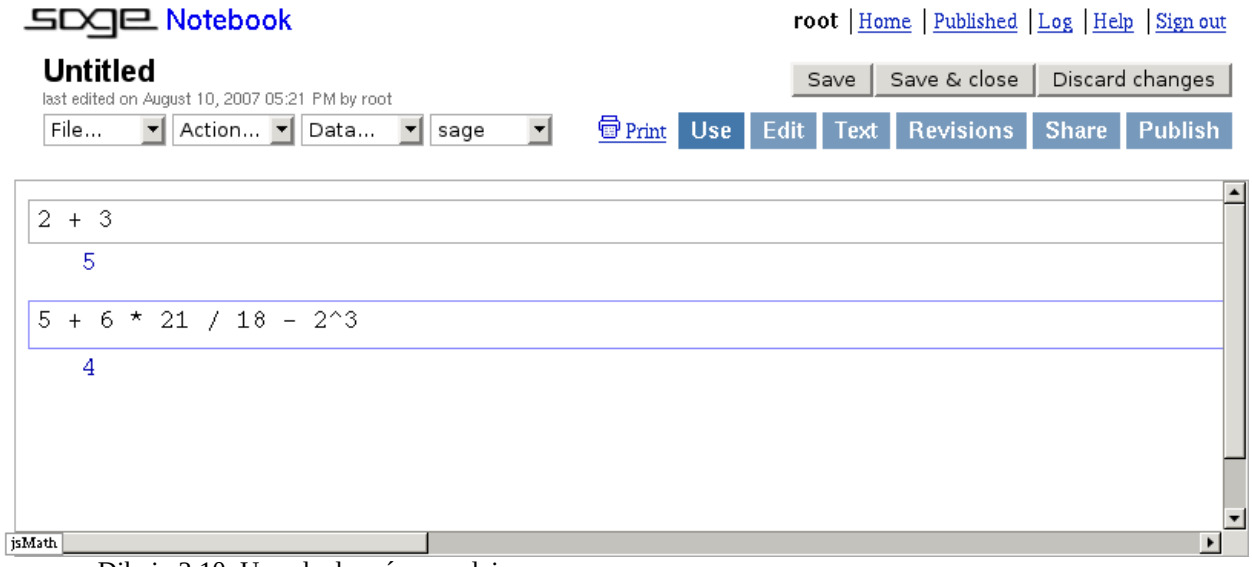
Presione shift y enter en su celda ahora y en unos momentos usted deberá ver un resultado como el del dibujo2.9.



Dibujo 2.9: Los resultados de la ejecución son mostrados.

Si el código fue enviado para su ejecución desde la celda de abajo en el cuaderno, una celda en blanco es automáticamente agregada debajo de esta celda cuando el servidor ha terminado de ejecutar el código.

Ahora introduzca el código fuente que se muestra en la segunda celda en el dibujo 2.10 y ejecútelo.



Dibujo 2.10: Un calculo más complejo

3 Fundamentos de programación en SAGE

3.1 Objetos, Valores y Expresiones.

Las líneas de código fuente

2 + 3 y 5 + 6*21/18 - 2^3

son ambas llamadas expresiones y lo siguiente es una definición de lo que es una expresión:

Una expresión en lenguaje de programación es una combinación de valores, variables, operadores, y funciones que son interpretadas (evaluadas) de acuerdo a las reglas particulares de precedencia y asociación para un lenguaje de programación particular, el cual calcula y posteriormente produce otro valor. La expresión se dice que evalúa a ese valor. Como en matemáticas, la expresión es (o puede decirse que tiene) su valor evaluado; la expresión es una representación de ese valor.
(http://es.wikipedia.org/wiki/Expresi%C3%B3n_%28programaci%C3%B3n%29)

En una computadora, un valor es una secuencia de bits en una o más localidades de la memoria que significan algo cuando son interpretados empleando un contexto dado. En SAGE, las secuencias de bits en la memoria que tienen significado son llamados **objects** (objetos). El mismo SAGE esta construido con objetos y la información que los programas de SAGE procesan también están representados como objetos. Los objetos son explicados a mayor detalle en el capítulo 4.

En las expresiones anteriores, 2, 3, 5, 6, 21 y 18 son objetos que se interpretan usando un contexto llamado contexto **sage.rings.integer.Integer.** Los contextos que pueden ser asociados con objetos son llamados **types** (tipos) y un objeto que es de tipo sage.rings.integer.Integer es usado para representar enteros.

Hay un comando en SAGE llamado **type()** que muestra el tipo de cualquier objeto que es pasado a el. Hagamos que el comando type() nos diga cual es el tipo de los objetos 3 y 21 ejecutando el siguiente código: (Nota: de este punto en adelante, el código fuente que será introducido en una celda y cualquier resultado que necesite ser mostrado, se dará sin usar la captura de pantalla de la hoja de trabajo).

```
type(3)
|
        <type ‘sage.rings.integer.Integer’>
type(21)
|
        <type ‘sage.rings.integer.Integer’>
```

La forma en que una persona le dice al comando type() de que objeto quieren ver la información de tipo, es colocando el objeto entre los paréntesis que están al lado derecho del nombre ‘type’.

3.2 Operadores

En las expresiones de arriba, los caracteres +, -, *, /, ^ son llamados operadores y su propósito es decirle a SAGE que operaciones realizar en los objetos en una expresión. Por ejemplo, en las expresiones 2 + 3, el operador de adición + le dice a SAGE que sume el entero 2 con el entero 3 y regrese el resultado. Como ambos objetos 2 y 3 son del tipo sage.rings.integer.Integer, el resultado que se obtiene con la suma de ellos 2 también será un objeto del tipo sage.rings.integer.Integer.

El operador de substracción es -, el operador de multiplicación es *, / es el operador de división, % es el operador para obtener el residuo de una división, y ^es el operador de exponente. SAGE tiene más operadores además de estos y se puede conseguir más información de estos en la documentación de Python.

Los siguientes ejemplos muestran el uso de los operadores -, *, /, % y ^:

```
5-2
|
        3
3*4
|
        12
30/3
|
        10
8%5
|
        3
2^3
|
        8
```

El signo – también puede ser utilizado para indicar un número negativo:

```
-3
|
        -3
```

Restar un número negativo nos da como resultado un número positivo:

```
- - 3
|
        3
```

3.3 Precedencia de los operadores

Cuando las expresiones contienen más de 1 operador, SAGE usa un set de reglas llamadas precedencia de los operadores para determinar el orden en el cual los operadores son aplicados a los objetos en la expresión. La precedencia de los operadores también es conocida como el orden de las operaciones. Los operadores con mayor prioridad son evaluados antes de los operadores con menor prioridad. La siguiente tabla muestra un sub grupo de reglas de prioridad de operadores en SAGE con los operadores de mayor prioridad situados en posición más alta en la tabla.

^	Los exponentes son evaluados de derecha a izquierda.
*, %, /	Posteriormente multiplicación, porcentaje restante y operadores de división son evaluados de izquierda a derecha.
+, -	Finalmente suma y resta son evaluados de izquierda a derecha.

Apliquemos manualmente estas reglas de prioridad a la expresión de multi-operadores que empleamos anteriormente. Aquí esta la expresión en código fuente:

```
5 + 6*21/18 - 2^3
```

Y aquí en forma tradicional:

$$5 + \frac{6 \cdot 21}{18} - 2^3$$

De acuerdo a las reglas de precedencia, este es el orden en el cual SAGE evalúa las operaciones en la expresión:

```
5 + 6*21/18 - 2^3
5 + 6*21/18 - 8
5 + 126/18 - 8
5 + 7 - 8
12 - 8
4
```

Comenzando por la primera expresión, SAGE evalúa el operador ^ primero, el cual resulta en el 8 de la expresión de abajo. En la segunda expresión, el operador * es el siguiente en ser ejecutado y así sucesivamente. La última expresión muestra que el resultado final después de haber evaluado todos los operadores es 4.

3.4 Cambiando el orden de los operadores en una expresión

El orden predispuesto de las operaciones para una expresión puede ser cambiado agrupando varias partes de la expresión con paréntesis. Los paréntesis obligan al código que esta dentro de ellos a ser evaluados antes que cualquier otro operador. Por ejemplo, la expresión 2+4*5 evaluada nos da como resultado 22 usando las reglas normales de precedencia:

```
2 + 4*5
|
22
```

Sin embargo, si los paréntesis se colocan alrededor de 4 + 5, la suma es obligada a ser evaluada antes que la multiplicación y el resultado es 30:

```
(2 + 4)*5
|
30
```

Los paréntesis también pueden anidarse, y estos son evaluados desde dentro hacia afuera:

```
((2 + 4)*3)5
|
90
```

Como los paréntesis son evaluados antes que cualquier otro operador, son colocados al principio de la tabla de precedencia:

()	Los paréntesis son evaluados de dentro hacia afuera.
^	Los exponentes son evaluados de derecha a izquierda.
*, %, /	Posteriormente multiplicación, porcentaje restante y operadores de división son evaluados de izquierda a derecha.
+, -	Finalmente suma y resta son evaluados de izquierda a derecha.

3.5 Variables

Una variable es un **nombre** que puede ser asociado con una dirección de memoria para que así las personas puedan hacer referencia a símbolos de patrones de bits en la memoria usando un **nombre** en vez de un **número**. Una forma de crear variables en SAGE es por **asignación** y consiste en colocar el nombre de una variable que le gustaría crear en el lado izquierdo de un signo igual '=' y una expresión en el lado derecho del signo igual. Cuando la expresión regresa un objeto, el objeto es asignado a la variable.

En el siguiente ejemplo, una variable llamada **box** es creada y se le asigna el número 7:

```
box = 7
```

Note que a diferencia de ejemplos anteriores, no se muestra un resultado en la hoja de trabajo debido a que el resultado fue colocado en la variable **box**. Si se desea ver el contenido en la variable, escriba su nombre en una celda en blanco y posteriormente evalúela:

```
box
```

```
7
```

Como puede verse en el ejemplo, las variables que son creadas en una celda dada en una hoja de trabajo también están disponibles para las otras celdas en la misma hoja de trabajo. Las variables existen en una hoja de trabajo, siempre y cuando esta este abierta, pero cuando se cierra la hoja de trabajo las variables se pierden. Cuando se abre de nuevo la hoja de trabajo, las variables necesitarán ser creadas de nuevo evaluando las celdas en las que están asignadas. Las variables pueden guardarse antes de que se cierre la hoja de trabajo y después cargada cuando la hoja de trabajo se abra de nuevo, pero este es un tópico avanzado que se cubrirá mas adelante.

SAGE también distingue entre mayúsculas y minúsculas. Esto significa que SAGE toma en cuenta si cada letra de una palabra es mayúscula o minúscula cuando decide si 2 o mas nombres de variables son iguales o no. Por ejemplo, las variables **Box** y **box** no son las mismas variables debido a que la primera inicia con mayúscula 'B' y la segunda variable inicia con una minúscula 'b'.

Los programas son capaces de tener más de 1 variable y aquí hay un ejemplo mas sofisticado que emplea 3 variables:

```
a = 2
```

```
b = 3
```

```
a + b
```

```
5
```

```
respuesta = a + b
|
respuesta
|
        5
```

La parte de una expresión que se ubica en el lado derecho de un signo igual '=' siempre es evaluada primero y el resultado es entonces asignado a la variable que esta en el lado izquierdo del signo igual.

Cuando una variable se pasa al comando type(), el tipo de objeto que se le asigna a la variable es mostrado:

```
a = 4
type (a)
    |
        <type 'sage.rings.integer.Integer'>
```

Los tipos de datos y el comando type se cubrirán más a fondo mas adelante.

3.6 Instrucciones

Las instrucciones son parte de un lenguaje de programación que es usado para codificar lógica algorítmica. A diferencia de las expresiones, las instrucciones no regresan objetos y son usadas debido a los varios efectos que son capaces de lograr. Las instrucciones pueden contener ambas expresiones y variables, y los programas son construidos usando una secuencia de declaraciones.

3.6.1 La instrucción Print

Si más de una expresión en una celda genera un resultado desplegable, la celda solo desplegara el resultado de la última expresión. Por ejemplo, este programa crea 3 variables y posteriormente intenta mostrar los contenidos de estas variables:

```
a = 1
b = 2
c = 3
a
b
c
|
        3
```

En SAGE, los programas son ejecutados una línea a la vez, comenzando por la primer línea de la parte superior del código y trabajando hacia abajo de ahí. En este ejemplo, la línea a = 1 es ejecutada primero, después la línea b = 2, y así sucesivamente. Note, sin embargo, que aunque hayamos querido ver lo que había en las 3 variables, solo el contenido de la última variable fue mostrado.

SAGE tiene una instrucción llamada **print** que permite mostrar los resultados de las expresiones sin importar donde se encuentren en la celda. Este ejemplo es similar al anterior excepto que las instrucciones print son usadas para desplegar el contenido de las 3 variables:

```
a = 1
b = 2
c = 3
print a
print b
print c
```

```
1
2
3
```

La instrucción print también imprime (en pantalla) múltiples resultados en la misma línea si se colocan comas entre las expresiones que se desean:

```
a = 1
b = 2
c = 3*6
print a,b,c
```

```
1  2  18
```

Cuando una coma es colocada después de una variable u objeto que es pasado a la instrucción print, le dice a la instrucción que no baje el cursor al siguiente renglón después de haber finalizado de imprimir. Por lo tanto, la próxima vez que una instrucción print sea ejecutada, colocara su salida en la misma línea que en la de la salida de la instrucción anterior.

Otra forma de desplegar múltiples resultados en una celda es usando punto y coma ‘;’. En SAGE, el punto y coma puede ser colocado después de las instrucciones como terminadores opcionales, pero la mayoría del tiempo uno solo los vera usados para colocar múltiples instrucciones en la misma línea. El siguiente ejemplo muestra el uso del punto y coma para permitir que las variables a, b y c sean inicializadas en la misma línea:

```
a=1;b=2;c=3
print a,b,c
```

```
1 2 3
```

El siguiente ejemplo muestra como el punto y coma también puede ser utilizado para desplegar múltiples resultados de una celda:

```
a = 1
b = 2
c = 3*6
a;b;c
```

```
1
2
18
```

3.7 Secuencias (strings)

Una secuencia es un tipo de objeto que es usado para retener información de tipo texto. La expresión típica que es empleada para crear un objeto de secuencia consiste en texto el cual esta encerrado entre comillas, dobles o sencillas. Las secuencias pueden ser referenciadas por variables igual que con los números y las secuencias también pueden ser mostradas por la instrucción print. El siguiente ejemplo asigna un objeto de secuencia a la variable ‘a’, imprime el objeto de secuencia al cual ‘a’ hace referencia, y posteriormente despliega su tipo:

```
a = "Hola, soy una secuencia."
print a
type(a)
```

```
Hola, soy una secuencia.
<type 'str'>
```

3.8 Comentarios

El código fuente frecuentemente puede ser difícil de entender y por lo tanto todos los lenguajes de programación proporcionan la habilidad para incluir comentarios en el código. Los comentarios son usados para explicar lo que el código cerca de ellos esta haciendo y usualmente están hechos para ser leídos por una persona que esta viendo el código fuente. Los comentarios son ignorados cuando se ejecuta el programa.

Existen 2 formas en que SAGE permite agregar comentarios al código fuente. La primer forma es agregando un símbolo ‘#’ a la izquierda de cualquier texto que se desee como comentario. El texto desde el símbolo # hasta el final de la línea donde se encuentra este símbolo será tratado como comentario. Aquí esta un programa que contiene un comentario empleando el símbolo #.

```
#Esto es un comentario.
x = 2 #Inicializa la variable x a 2.
print x
```

```
2
```

Cuando este programa es ejecutado, el texto que inicia con el símbolo # es ignorado.

La segunda forma de agregar comentarios a un programa en SAGE es encerrándolos en un set de 3 comillas. Esta opción es útil cuando el comentario es demasiado largo para caber en la línea. El siguiente programa muestra un comentario de esta manera:

```
"""
Este es un comentario largo y emplea más de una línea.
El siguiente código asigna el número 3 a la variable
x y luego la imprime.
"""
x = 3
print x
```

```
3
```

3.9 Operadores condicionales

Un operador condicional es un operador que es usado para comparar 2 objetos. Las expresiones que contienen operadores condicionales regresan un objeto **booleano** y un objeto **booleano** es uno que solo puede ser verdadero o falso. La tabla muestra el operador condicional que utiliza SAGE:

Operador	Descripción
x = = y	**Verdadero (True)** si los dos objetos son iguales y **falso (False)** si no son iguales. Note que = = realiza una comparación y no una asignación como lo hace =.
x < > y	**Verdadero** si los objetos no son iguales y **Falso** si son iguales.
x! = y	**Verdadero** si los objetos no son iguales y **Falso** si son iguales.
x < y	**Verdadero** si el objeto izquierdo es menor que el objeto izquierdo y **Falso** si el objeto izquierdo no es menor que el derecho.
x < = y	**Verdadero** si el objeto izquierdo es menor o igual que el objeto derecho y **Falso** si el objeto izquierdo no es menor o igual que el objeto derecho.
x > y	**Verdadero** si el objeto izquierdo es mayor que el objeto derecho y **Falso** si el objeto izquierdo no es mayor que el objeto derecho.
x > = y	**Verdadero** si el objeto izquierdo es mayor o igual que el derecho y **Falso** si el objeto izquierdo no es mayor o igual que el objeto derecho.

Tabla 2: Operadores condicionales

Los siguientes ejemplos muestran el uso de cada operador condicional de la tabla 2, para comparar objetos que fueron colocados en las variables x y y:

```
# Ejemplo 1.
x = 2
y = 3
print x, "==", y, ":", x == y
print x, "<>", y, ":", x <> y
print x, "!=", y, ":", x != y
print x, "<", y, ":", x < y
print x, "<=", y, ":", x <= y
print x, ">", y, ":", x > y
print x, ">=", y, ":", x >= y
```

```
2 == 3 : False
2 <> 3 : True
2 != 3 : True
2 < 3 : True
2 <= 3 : True
2 > 3 : False
2 >= 3 : False
```

```
# Ejemplo 2.
x = 2
y = 2
print x, "==", y, ":", x == y
print x, "<>", y, ":", x <> y
print x, "!=", y, ":", x != y
print x, "<", y, ":", x < y
print x, "<=", y, ":", x <= y
print x, ">", y, ":", x > y
print x, ">=", y, ":", x >= y
```

```
2 == 2 : True
2 <> 2 : False
2 != 2 : False
2 < 2 : False
2 <= 2 : True
2 > 2 : False
2 >= 2 : True
```

```
# Ejemplo 3.
x = 3
y = 2
print x, "==", y, ":", x == y
print x, "<>", y, ":", x <> y
print x, "!=", y, ":", x != y
print x, "<", y, ":", x < y
print x, "<=", y, ":", x <= y
print x, ">", y, ":", x > y
print x, ">=", y, ":", x >= y
```

```
3 == 2 : False
3 <> 2 : True
3 != 2 : True
3 < 2 : False
3 <= 2 : False
3 > 2 : True
3 >= 2 : True
```

Los operadores condicionales son ubicados en el menor nivel de prioridad que los otros operadores que hemos cubierto hasta este punto:

()	Los paréntesis son evaluados de dentro hacia afuera.
^	Los exponentes son evaluados de derecha a izquierda.
*, %, /	Posteriormente multiplicación, porcentaje restante y operadores de división son evaluados de izquierda a derecha.
+, -	Después suma y resta son evaluados de izquierda a derecha.
= =,<>,!=,<,<=,>,>=	Después, son evaluados los operadores condicionales.

3.10 Realizando decisiones con la instrucción If

Todos los lenguajes de programación proveen la habilidad para realizar decisiones y la instrucción mas usada comúnmente para hacer decisiones en SAGE es la instrucción **if.**

Una forma simplificada de sintaxis para la instrucción **if** es la siguiente:

```
if < expresión >:
< Instrucción>
< Instrucción>
< Instrucción>
.
.
```

La forma en que la instrucción **if** funciona es evaluando la expresión a su derecha inmediata y posteriormente examina el objeto que es regresado. Si este objeto es "verdadero", las instrucciones dentro de la instrucción **if** serán ejecutadas. Si el objeto es "falso", las instrucciones dentro del **if** no se ejecutaran.
En SAGE, un objeto es verdadero "true" si no es cero o no este vacio y es falso "false" si es cero o esta vacío. Una expresión que contiene uno o más operadores condicionales regresará un objeto **booleano** que podrá ser verdadero (**True)** o falto (**False**).

La forma en que las instrucciones son colocadas dentro del ciclo es colocando un punto y coma ';' al final de la cabecera de la instrucción y después colocando una o mas instrucciones debajo de ella. Las instrucciones que son colocadas por debajo de esa instrucción deben estar separadas cada una uno o mas espacios del lado izquierdo de la instrucción principal. Sin embargo, todas las instrucciones separadas del margen izquierdo deben ser separadas de la misma manera y el mismo espacio. Una o más instrucciones que sean separadas de esta manera son llamadas bloques de código.

El siguiente programa usa una instrucción **if** para determinar si el número en la variable x es mayor que 5. Si x es mayor que 5, el programa escribirá "Mayor" y después "Fin del programa".

```
x = 6
print x > 5

if x > 5:
        print x
        print "Mayor"

print "Fin del programa"
```

```
True
6
Mayor
Fin del programa
```

En este programa, x ha sido inicializado a 6 y por lo tanto la expresión x > 5 es verdadera. Cuando esta expresión es escrita, escribe el objeto booleano **True** (verdadero) dado que 6 es mayor que 5.

Cuando la instrucción **if** evalúa la expresión y determina que es verdadera, entonces ejecuta las instrucciones print que están dentro de ella y los contenidos de la variable x son escritos junto con la secuencia "Mayor". Si se necesitan colocar instrucciones adicionales dentro de la instrucción **if**, estas hubieran sido agregadas debajo de las instrucciones print al mismo nivel de espaciamiento.

Finalmente, la ultima instrucción print, escribe la secuencia "Fin del programa" sin importar lo que haga la declaración **if**.

Aquí esta el mismo programa excepto que la x ha sido inicializada a 4 en vez de 6:

```
x = 4
print x > 5

if x > 5:
        print x
        print "Mayor."

print "Fin del programa."
|
        False
        Fin del programa.
```

Esta vez la expresión x > 4 regresa un objeto **false** (falso) lo que ocasiona que la declaración **if** no ejecute las instrucciones que están dentro de ella.

3.11 Los operadores Booleanos and, or y not.

Algunas veces uno desea checar si 2 o más expresiones son todas verdaderas y la forma de hacer esto es con el operador **and**:

```
a = 7
b = 9
print a < 5 and b < 10
print a > 5 and b > 10
print a < 5 and b > 10
print a > 5 and b < 10
if a > 5 and b < 10:
        print "Estas 2 expresiones son verdaderas."
|
    False
    False
    False
    True
    Estas 2 expresiones son verdaderas.
```

En otras ocasiones uno desea determinar si por lo menos una expresión en un grupo es verdadera y esto es hecho con el operador **or**:

```
a = 7
b = 9
print a < 5 or b < 10
print a > 5 or b > 10
print a > 5 or b < 10
print a < 5 or b > 10
if a < 5 or b < 10:
        print "Por lo menos una de estas expresiones es verdadera."
|
        True
        True
        True
        False
        Por lo menos una de estas expresiones es verdadera.
```

Finalmente, el operador **not** puede ser usado para cambiar o negar un resultado, es decir, cambiar verdadero por falso, y viceversa:

```
a = 7
print a > 5
print not a > 5
|
        True
        False
```

Los operadores Booleanos son colocados en un nivel menor de prioridad que todos los demás operadores que hemos visto hasta ahora:

()	Los paréntesis son evaluados de dentro hacia afuera.
^	Los exponentes son evaluados de derecha a izquierda.
*, %, /	Posteriormente multiplicación, porcentaje restante y operadores de división son evaluados de izquierda a derecha.
+, -	Después suma y resta son evaluados de izquierda a derecha.
= =,<>,!=,<,<=,>,>=	Después, son evaluados los operadores condicionales.
not, and, or	Los operadores booleanos son evaluados hasta el final.

3.12 Creando lazos con la instrucción while

Muchos tipos de máquinas, incluyendo computadoras, derivan mucho de su poder del principio de los ciclos repetitivos. SAGE provee varias formas para implementar ciclos repetitivos en un programa y estas maneras van desde directas, hasta sutiles. Comenzaremos tratando los lazos en SAGE iniciando con la instrucción **while** directa.

La especificación de sintaxis para la instrucción **while** es la siguiente:

```
while <expresión>:
        <Instrucción>
        <Instrucción>
        <Instrucción>
        .
        .
```

La instrucción **while** es parecida a la instrucción **if** excepto que ejecutará repetidamente las instrucciones que contenga siempre y cuando la expresión a la derecha de su cabecera sea cierta. Tan pronto como la expresión regrese un objeto falso, la instrucción **while** se salta las instrucciones y la ejecución continúa con la instrucción que sigue inmediatamente a la instrucción **while** (si es que hay una). De igual forma que en la instrucción **if**, para las líneas dentro de la instrucción **while** se debe dejar un espaciado (sangría).

El siguiente programa de ejemplo utiliza un lazo **while** para imprimir los enteros de 1 al 10:

```
# Imprimir los enteros del 1 al 10.
x = 1 #Inicializar la variable de conteo a 1 fuera del lazo.
while x <= 10:
        print x
        x = x + 1 #Incrementar x en 1.
```

```
1
2
3
4
5
6
7
8
9
10
```

En este programa, una variable es creada. Es usada para decirle a la instrucción **print** que entero escribir y también es usada en la expresión que determina si el lazo **while** debe continuar el lazo o no. Cuando el programa es ejecutado, se guarda un 1 en x y posteriormente se entra a la instrucción **while**. La expresión x < = 10 se vuelve 1 < = 10 y, como 1 es menor que o igual a 10, un objeto booleano verdadero es regresado por la expresión.

La instrucción **while** ve que la expresión regresa un objeto verdadero y por lo tanto ejecuta todas las instrucciones dentro de ella desde la parte superior hasta la inferior.

La instrucción print, escribe los contenidos actuales de x (que es 1) después se ejecuta x = x + 1.

La expresión x = x + 1 es una expresión estándar que es utilizada en muchos lenguajes de programación. Cada vez que una expresión en esta forma es evaluada, incrementa en 1 la variable que contiene. Otra forma de describir el efecto que tiene esta expresión en x es decir que incrementa x en 1.

En este caso x vale 1 y, después de que la expresión es evaluada, x vale 2.

Después de que la última instrucción dentro del ciclo **while** es ejecutada, la instrucción **while** reevalúa la expresión a la derecha de su cabecera para determinar si debe continuar con el lazo o no. Dado que x es 2 en este punto, la expresión regresa un valor verdadero y el código dentro de la instrucción **while** es ejecutado de nuevo. Este lazo se repetirá hasta que x se incremente a 11 y la expresión regrese un valor falso.

El programa anterior puede ser ajustado en varias formas para conseguir diferentes resultados. Por ejemplo, el siguiente programa escribe los enteros del 1 al 100 incrementando el 10 de la expresión que esta al lado derecho de la cabecera **while** a 100. Se ha colocado una coma después de la instrucción print, para que la salida sea desplegada en la misma línea hasta que llegue al tope del lado derecho de la ventana.

```
# Imprimir los enteros del 1 al 100.
x = 1
while x <= 100:
        print x,
        x = x + 1 #Incrementar x en 1.
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
100
```

El siguiente programa escribe los enteros impares del 1 al 99 cambiando el valor a incrementar en la expresión de 1 a 2:

```
# Imprimir los enteros impares del 1 al 99.
x = 1
while x <= 100:
        print x,
        x = x + 2 #Incrementar x en 2.
```

```
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51
53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99
```

Finalmente, este programa escribe los números del 1 al 100 en orden inverso:

```
# Imprimir los enteros del 1 al 100 en orden inverso.
x = 100
while x >= 1:
        print x,
        x = x - 1 #Reducir a x en 1.
```

```
100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77
76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53
52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29
28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2
1
```

Para lograr este resultado, este programa tuvo que inicializar a x en 100, checar si x era mas grande o igual a 1 (x > = 1) para continuar el lazo, y reducir a x restándole 1 en vez de sumándoselo.

3.13 Lazos de larga duración, lazos infinitos e interrupción de la ejecución.

Es fácil crear un lazo que se va a ejecutar una gran cantidad de veces, o incluso una cantidad infinita de veces, ya sea intencionalmente o por error. Cuando ejecutas un programa que contiene un lazo infinito, este correrá hasta que se le indique a SAGE que interrumpa su ejecución. Esto es hecho seleccionando el menú **Action** que esta cerca de la parte superior izquierda de la hoja de trabajo y después seleccionando el elemento del menú **Interrupt**. Los programas con lazos de larga duración también pueden ser interrumpidos de esta manera. En ambos casos, la barra vertical verde de ejecución indicará que el programa esta actualmente en ejecución y la barra verde desaparecerá después de que el programa haya sido interrumpido.

Este programa contiene un lazo infinito:

```
#Programa de ejemplo con lazo infinito.
x = 1
while x < 10:
respuesta = x + 1
```

Como los contenidos en x nunca cambiaron dentro del lazo, la expresión x < 10 siempre evalúa verdadero lo que causa que el lazo siga sin detenerse.

Ejecute este programa ahora y posteriormente interrúmpalo utilizando el comando de interrupción de la hoja de trabajo. Algunas veces no es suficiente con interrumpir la hoja de trabajo y entonces necesitaras seleccionar **Action - > Restart worksheet (reiniciar hoja de trabajo).** Sin embargo, cuando una hoja de trabajo es reiniciada, todas sus variables son regresadas a su condición inicial así que las celdas que asignaron valores a estas variables necesitan ser ejecutadas de nuevo.

3.14 Insertando y borrando celdas de hojas de trabajo

Si se necesita insertar una nueva celda de hoja de trabajo entre 2 celdas ya existentes, hay que mover el cursor del ratón entre las 2 celdas justo sobre la de abajo y una **barra horizontal azu**l aparecerá. Hacer click en esta barra azul y una nueva celda será insertada en la hoja de trabajo.

Si se desea borrar una celda, se debe borrar todo el texto que esta celda contiene para que quede vacía. Es necesario asegurarse que el cursor este en la celda ahora vacía y presionar la tecla de retroceso en el teclado. La celda entonces será borrada.

3.15 Introducción a tipos de objetos más avanzados

Hasta este punto, solo hemos usado objetos del tipo 'sage.rings.integer.Integer' y del tipo 'str'. Sin embargo, SAGE incluye un gran número de tipos de objetos matemáticos y no matemáticos que pueden ser usados para una amplia variedad de propósitos. Las siguientes secciones introducen dos tipos de objetos matemáticos adicionales y dos tipos de objetos no matemáticos.

3.15.1 Números racionales

Los números racionales son contenidos en objetos del tipo **sage.rings.rational.Rational**. El siguiente ejemplo escribe el tipo del número racional ½, asigna ½ a la variable x, la escribe y posteriormente despliega el tipo de objeto al cual x hace referencia:

```
print type(1/2)
x = 1/2
print x
type(x)
```

```
<type 'sage.rings.rational.Rational'>
1/2
<type 'sage.rings.rational.Rational'>
```

El siguiente código fue introducido en una celda separada en la hoja de trabajo después de que el código anterior fue ejecutado. Muestra la suma de 2 números racionales y el resultado, el cual también es un número racional, siendo asignado a la variable y:

```
y = x + 3/4
print y
type(y)
```

```
5/4
<type 'sage.rings.rational.Rational'>
```

Si un número racional es sumado a un número entero, el resultado es guardado en un objeto del tipo sage.rings.rational.Rational:

```
x = 1 + 1/2
print x
type(x)
```

```
3/2
<type 'sage.rings.rational.Rational'>
```

3.15.2 Números reales

Los números reales son guardados en objetos del tipo **sage.rings.real_mpfr.RealNumber**. El siguiente ejemplo escribe el tipo del número real .5, lo asigna a la variable x, escribe x y posteriormente despliega el tipo de objeto al cual x hace referencia:

```
print type(.5)
x = .5
print x
type(x)
```

```
<type 'sage.rings.real_mpfr.RealNumber'>
0.500000000000000
<type 'sage.rings.real_mpfr.RealNumber'>
```

El siguiente código fue introducido en una celda separada en la hoja de trabajo después de que el código anterior fue ejecutado. Muestra la suma de 2 números reales y el resultado, el cual es también un número real, siendo asignado a la variable y:

```
y = x + .75
print y
type(y)
```

```
1.25000000000000
<type 'sage.rings.real_mpfr.RealNumber'>
```

Si se suma un número real con uno racional, el resultado es guardado en un objeto de tipo sage.rings.real_mpfr.RealNumber:

```
x = 1/2 + .75
print x
type(x)
```

```
1.25000000000000
<type 'sage.rings.real_mpfr.RealNumber'>
```

3.15.3 Objetos que guardan secuencias de otros objetos: Listas y registros

El objeto de tipo list (lista) esta diseñado para guardar otros objetos en una colección ordenada o secuencia. Las listas son muy flexibles y son de los tipos de objetos mas fuertemente utilizados en SAGE. Las listas pueden contener objetos de cualquier tipo, pueden crecer y encogerse como sea necesario, y pueden ser anidadas. Los objetos en una lista pueden ser accesados por su posición en la lista y también pueden ser reemplazados por otros objetos. La habilidad de una lista para crecer, encogerse y cambiar sus contenidos la hace un tipo de objeto mutable.

Una forma de crear una lista es colocando 0 o más objetos o expresiones dentro de un par de corchetes [ ].

El programa siguiente inicia imprimiendo el tipo de una lista. De ahí crea una lista que contenga los números 50, 51, 52 y 53, los asigna a la variable x y la escribe. A continuación, escribe los objetos que están en las posiciones 0 y 3, reemplaza el 53 en la posición 3 por 100, escribe x nuevamente y finalmente escribe el tipo de objeto al cual se refiere x:

```
print type([])
x = [50,51,52,53]
print x
print x[0]
print x[3]
x[3] = 100
print x
type(x)
```

```
<type 'list'>
[50, 51, 52, 53]
50
53
[50, 51, 52, 100]
<type 'list'>
```

Note que el primer objeto en una lista es colocado en la posición 0 en vez de la posición 1 u que esto hace la posición del último objeto en la lista menor en 1 que la longitud de la lista. Además note que un objeto en una lista es accesado colocando un par de corchetes [ ], la cual contiene su número de posición, a la derecha de una variable que referencia a la lista.

El siguiente ejemplo muestra que pueden colocarse en una lista diferentes tipos de objetos:

```
x = [1, 1/2, .75, 'Hola', [50,51,52,53]]
print x
```

```
[1, 1/2, 0.750000000000000, 'Hola', [50, 51, 52, 53]]
```

Los registros también son secuencias y son similares a las listas excepto que son inmutables. Son creados usando un par de paréntesis en vez de un par de corchetes y que sean inmutables significa que una vez que uno de estos objetos ha sido creado, no puede crecer, encogerse o cambiar los objetos que contiene.

El siguiente programa es similar al primer programa de ejemplo de lista, excepto que usa un registro en vez de una lista, no intenta cambiar el objeto en la posición 4, y usa la técnica del punto y coma para desplegar múltiples resultados en vez de instrucciones print:

```
print type(())
x = (50,51,52,53)
x;x[0];x[3];x;type(x)
|
        <type 'tuple'>
        (50, 51, 52, 53)
        50
        53
        (50, 51, 52, 53)
        <type 'tuple'>
```

3.15.3.1 Empaquetado y desempaquetado de registros

Cando múltiples valores separados por comas son asignados a una simple variable, los valores son automáticamente colocados en un registro y este es llamado **empaquetado de registros**:

```
t = 1,2
t
|
        (1, 2)
```

Cuando un registro es asignado a múltiples variables las cuales son separadas por comas, esto es llamado **desempaquetado de registros**:

```
a,b,c = (1,2,3)
a;b;c
|
        1
        2
        3
```

Un requerimiento para el desempaquetado de registros es que el número de objetos en el registro deben coincidir con el número de variables en el lado izquierdo del signo igual.

3.16 Usando lazos de while con listas y registros

Las instrucciones que enlazan pueden ser utilizadas para seleccionar a cada objeto en una lista o registro en turno de tal manera que pueda realizarse una operación en estos objetos. El siguiente programa usa el lazo **while** para escribir cada uno de los objetos en una lista:

```
#Imprimir cada objeto en la lista
x = [50, 51, 52, 53, 54, 55, 56, 57, 58, 59]
y = 0
while y <= 9:
        print x[y]
        y = y + 1
```

```
50
51
52
53
54
55
56
57
58
59
```

Un lazo también puede ser usado para buscar a través de una lista. El siguiente programa utiliza el lazo **while** y una instrucción **if** para buscar a través de una lista si contiene el número 53. Si se encuentra dicho número en la lista, un mensaje es escrito.

```
#Determinar si 53 esta en la lista.
x = [50,51,52,53,54,55,56,57,58,59]
y = 0
while y <= 9:
    if x[y] == 53:
            print "53 fue encontrado en la lista en la posición", y
    y = y + 1
```

```
53 fue encontrado en la lista en la posicion 3
```

Nota: No intente usar acentos en este programa, como es en el caso de la palabra posición, ya que el sistema marca error al querer utilizar instrucciones o imprimir textos con acento. En los comentarios no hay problema, pues todo comentario es ignorado, pues solo es para el usuario que esta desarrollando el programa.

3.17 El operador in

Crear lazos es una capacidad tan útil que incluso SAGE tiene un operador llamado **in** que crea lazos internamente. El operador **in** es capaz de buscar automáticamente en una lista para determinar si contiene un objeto dado. Si encuentra el objeto, regresará un valor verdadero y si no encuentra el objeto, regresará un valor falso. Los siguientes programas muestran ambos casos:

```
print 53 in [50,51,52,53,54,55,56,57,58,59]
print 75 in [50,51,52,53,54,55,56,57,58,59]
```

```
True
False
```

El operador **not** puede ser utilizado también con el operador **in** para cambiar su resultado:

```
print 53 not in [50,51,52,53,54,55,56,57,58,59]
print 75 not in [50,51,52,53,54,55,56,57,58,59]
```

```
False
True
```

3.18 Creando lazos con la instrucción for

La instrucción **for** usa un lazo para indexar a través de una lista o registro como lo hace la instrucción **while**, pero es mas flexible y automático. Aquí esta la especificación simplificada de la sintaxis para la instrucción **for**:

```
for <objetivo> in <objeto>:
<Instrucción>
<Instrucción>
<Instrucción>
.
.
.
```

En esta sintaxis, <objetivo> es usualmente una variable y <objeto> es usualmente un objeto que contiene otros objetos. En el resto de esta sección, asumamos que <objeto> es una lista. La instrucción **for** seleccionará cada objeto en la lista en turno, lo asignará a <objetivo>, y posteriormente ejecutará las instrucciones que están dentro de su bloque de código anexo.

El siguiente programa muestra una instrucción **for** siendo usada para escribir todos los elementos en una lista:

```
for x in [50,51,52,53,54,55,56,57,58,59]:
        print x
```

```
50
51
52
53
54
55
56
57
58
59
```

3.19 Funciones

Las funciones de programación son instrucciones que consisten en bloques de código nombrados que pueden ser ejecutados una o más veces siendo llamadas de otras partes del programa. Las funciones pueden tener objetos pasados a ellos desde el código de llamante y también pueden regresar objetos hacia el. Un ejemplo de una función es el comando type ( ) que hemos estado utilizando para determinar los tipos de objetos.

Las funciones son una manera en la que SAGE habilita el código para ser reutilizado. La mayoría de los lenguajes de programación permiten que el código se reutilice de esta manera, aunque en otros lenguajes este tipo de instrucciones de reutilización de código algunas veces son llamadas subrutinas o procedimientos.

Los nombres de función utilizan letras minúsculas. Si el nombre de una función contiene mas de una palabra (como calcularsuma) puede colocarse un guion bajo entre las palabras para mejorar la lectura (calcular_suma).

3.20 Las funciones son definidas usando la instrucción def

La instrucción que es usada para definir una función es llamada **def** y su sintaxis es la siguiente:
def <function name>(arg1, arg2, ... argN):
<statement>
<statement>
<statement>
.
.
.

La instrucción **def** contiene una cabecera que incluye el nombre de la función junto con los argumentos que pueden ser pasados a ella. Una función puede tener 0 o más argumentos y estos argumentos son colocados entre los paréntesis. Las instrucciones que son ejecutadas cuando es llamada la función son colocadas dentro de la función utilizando un bloque de código anexado.

El siguiente programa define una función llamada **sumnums** la cual toma 2 números como argumentos, los suma y regresa el resultado al código de llamada utilizando una instrucción **return**:

```
def sumnums(num1, num2):
        """
        Realiza la suma de num1 y num2.
        """
        resp = num1 + num2
        return resp

#Llama la función y suma 2 a 3.
a = sumnums(2, 3)
print a

#Llama la función y suma 4 a 5.
b = sumnums(4, 5)
print b
|
        5
        9
```

La primera vez que esta función es llamada, se pasan los números 2 y 3 y estos números son asignados a las variables **num1** y **num2** respectivamente. Las variables de argumento que tienen objetos pasados a ellos durante una llamada de función pueden ser usadas dentro de la función según sea necesario.

Note que cuando la función regresa al llamante, el objeto que fue colocado a la derecha de la instrucción **return** se vuelve disponible para el código llamante. Es casi como si la propia función es remplazada con el objeto que regresa. Otra forma de pensar en un objeto regresado es que es enviado hacia afuera del lado izquierdo del nombre de la función en el código llamante, a través del signo igual, y es asignado a la variable. En la primera llamada de función, el objeto que regresa la función esta siendo asignado a la variable ‘a’ y posteriormente este objeto es escrito.

La segunda llamada a la función es similar a la primera, excepto que pasa números diferentes (4,5) a la función.

3.21 Un sub conjunto de funciones incluidas en SAGE

SAGE incluye un número largo de funciones pre-escritas que pueden ser utilizadas para una amplia variedad de propósitos. La tabla 3 contiene un sub conjunto de estas funciones y una lista mas larga de funciones que pueden ser encontradas en la documentación de SAGE. Una lista mas completa de funciones puede ser encontrada en el **Manual de referencia de SAGE** (SAGE Reference Manual).

Nombre de la función	**Descripción**
abs	Regresa el valor absoluto del argumento.
acos	La función de arcocoseno.
add	Regresa la suma de una secuencia de números (no cadenas) mas el valor del parámetro ‘’inicio’. Cuando la secuencia se encuentra vacía, regresa inicio.
additive_order	Regresa el orden aditivo de x.
asin	La función arcoseno.
atan	La función arcotangente.
binomial	Regresa el coeficiente binomial.
ceil	La función ceiling. Convierte un número real arbitrario al entero mas pequeño cercano no menor que x.
combinations	Una combinación de un multi-grupos (una lista de objetos que puedan contener el mismo objeto varias veces) mset es una selección sin orden sin repeticiones y esta representada por una sub-lista ordenada de mset. Regresa el conjunto de todas las combinaciones del multi-grupo mset con k elementos.
complex	Crea un número complejo de una parte real y una parte imaginaria opcional. Esto es equivalente a (real + imag*1j) donde imag es 0 por defecto.
cos	La función coseno.
cosh	La función coseno hiperbólico.
coth	La función cotangente hiperbólica.
csch	La función cosecante hiperbólica.
denominator	Regresa el denominador de x.
derivative	La derivada de f.
det	Regresa la determinante de la matriz x.
diff	La derivada de f.
dir	Regresa una lista ordenada alfabéticamente de nombres que consta (algunas) de los atributos del objeto dado y atributos alcanzables.
divisors	Regresa una lista de todos los divisores enteros positivos del entero n.
dumps	Bota el objeto a una secuencia s. Para recuperar el objeto, usar load (s).
e	La base del logaritmo natural.
eratosthenes	Regresa una lista de los números primos < = n.
exists	Si s contiene un elemento x tal que P(x) sea verdadero, esta función regresa un valor verdadero y el elemento x. Por lo contrario regresa un valor falso solamente.

exp	La función exponencial, exp(x) = e^x.
expand	Regresa la forma expandida de un polinomio.
factor	Regresa la factorización del entero n como una lista ordenada de registros (p,e).
factorial	Calcula el factorial de n, el cual es el producto de 1*2*3…(n-1)n.
fibonacci	Regresa el n-ésimo número Fibonacci.
fibonacci_sequence	Muestra la secuencia Fibonacci para todos los números Fibonacci f_n desde n = inicio hasta (pero sin incluir) n = fin.
fibonacci_xrange	Regresa un iterador de todos los números Fibonacci en el rango dado, incluyendo f_n = inicio, pero sin incluir, f_n = fin.
find_root	Númericamente encuentra una raíz para f en el intervalo cerrado [a, b] o [b, a] si es posible, donde f es una función en la variable.
floor	La función floor. Convierte un número real arbitrario al entero mas grande cercano menor o igual a x.
forall	Si P(x) es verdadera cada x en S, regresa solo un valor verdadero. Si hay algún elemento x en S tal que P no sea verdadera, regresa un valor falso y el valor de x.
forget	Olvidar la suposición dada, o llamar sin argumentos para olvidar todas las suposiciones. Aquí la suposición es una especie de restricción simbólica.
function	Crea una función simbólica formal con el nombre dado.
gaussian_binomial	Regresa el binomio Gaussiano.
gcd	El mas grande común divisor de a y b.
get_memory_usage	Regresa el uso de la memoria(solo para Linux).
hex	Regresa la representación hexadecimal de un entero.
imag	Regresa la parte imaginaria de x.
imaginary	Regresa la parte imaginaria de un número complejo.
integer_ceil	Regresa el entero mas grande mayor o igual a x.
integer_floor	Regresa el entero mas chico < = x.
integral	Regresa una integral indefinida de un objeto x.
integrate	La integral de f.
interval	Define un intervalo de números desde a hasta b.
is_AlgebraElement	Regresa un valor verdadero si x es del tipo AlgebraElement (elemento algebraico).
is_commutative	Regresa un valor verdadero si x es conmutativa.
is_ComplexNumber	Regresa un valor verdadero si x es un número complejo.
is_even	Indica si el entero x es o no par, es decir, divisible entre 2.
is_Infinite	Indica si x es infinita.
is_Integer	Indica si x es del tipo SAGE integer.
is_odd	Indica si el entero x es o no impar. Esto es por definición el complemento de is_even.
is_power_of_two	Esta función regresa un valor verdadero si, y solo si n es potencia de 2.
is_prime	Regresa valor positivo si x es primo, de lo contrario es falso.
is_prime_power	Regresa un valor positivo si x es potencia prima, de lo contrario es falso.

is_pseudoprime	Regresa un valor verdadero si x es un pseudo-primo, falso de ser contrario.
is_RealNumber	Regresa un valor verdadero si x es un número real.
is_Set	Regresa un valor verdadero si x es un conjunto de SAGE.
is_square	Analiza si n es o no un cuadrado y si n es un cuadrado muestra la raíz cuadrada. Si n no es cuadrado, no muestra nada.
is_SymbolicExpression	Indica si x es del tipo SAGE SymbolicExpression (expresión sibolica).
isqrt	Muestra la raíz cuadrada más cercana de un entero.
laplace	Intenta calcular y muestra la transformada de Laplace con respecto a l la variable t y el parámetro s. Si la función no encuentra la solución se muestra una función de retardo.
latex	Use latex (…) para la composición tipográfica de un objeto de SAGE
lcm	El minimo común múltiplo de a y b,
len	Muestra el número de elementos de una secuencia o mapeo.
lim	Muestra el límite conforme la variable v se acerca desde la dirección dada.
limit	Muestra el límite conforme la variable v se acerca desde la dirección dada.
list	Crea una lista nueva.
list_plot	Toma una lista de información, creando una lista de registros (i,di) donde i va desde 0 hasta { len}(data)-1 y di es el i-nesimo valor de información y coloca puntos en esos valores de registros.
load	Carga un objeto de SAGE desde el archivo el cual tendrá una extensión .sobj si no tiene ninguna asignada ya.
loads	Recupera un objeto x que ha sido eliminado a una secuencia s utilizando s = dumps(x).
log	Muestra el logaritmo de x a la base dada.
matrix	Crea una matriz.
max	Con un solo argumento iterable, regresa su elemento más grande. Con 2 o mas argumentos muestra el argumento mas grande.
min	Con un solo argumento iterable, regresa su elemento más chico. Con 2 o mas argumentos muestra el argumento mas chico.
minimal_polynomial	Muestra el polinomio mínimo de x.
mod	Muestra la clase equivalente de n modulo m como elemento de z/mz.
mrange	Muestra la lista multirango con sus tamaños y tipo dados.
mul	Muestra el producto de los elementos de la lista x. Si no se da argumento opcional z, se inicia el producto con el primer elemento de la lista, de otro modo se utiliza z.
next_prime	Muestra el siguiente numero primo mayor al entero n.

number_of_arrangements	Muestra la cantidad de acomodos k para mset.
number_of_combinations	Muestra la cantidad de combinaciones k para mset.
number_of_derangements	Muestra el número de permutaciones donde ningún elemento del grupo aparece en su posición original.
number_of_divisors	Muestra la cantidad de divisores para el entero n.
number_of_permutations	Muestra el número de permutaciones. Nota, no utilizar por que será removida de futuras versiones de SAGE, es mejor utilizar Permutations.
numerator	Muestra el numerador de x.
numerical_integral	Muestra la integral numérica de la función en el intervalo desde a hasta b.
numerical_sqrt	Muestra la raíz cuadrada de x.
oct	Muestra la representación octal de un entero.
order	Muestra el orden de x.
parametric_plot	Toma 2 o 3 funciones como una lista o registro y realiza una grafica con la primer función como las coordenadas x, la segunda función como las coordenadas y y la tercer función (si la hay) como las coordenadas z.
permutations	Muestra el grupo de todas las permutaciones de mset (multigrupo).
pi	La relación de la circunferencia de un círculo con su diámetro.
plot	Muestra una grafica de la función.
power_mod	La n-ésima potencia de un modulo en el entero m.
prange	Muestra una lista de todos los números primos entre start y stop. Si el segundo argumento es omitido, muestra los números primos hasta el primer argumento.
previous_prime	Muestra el número primo mas grande < n.
prime_divisors	Muestra los divisores primos del entero n, ordenados en orden ascendente. Si n es negativo, no se incluye el signo – entre los divisores primos por que -1 no es un número primo.
prime_powers	Muestra una lista de todas las potencias primas entre el inicio (start) y final-1 (stop-1). Si el segundo argumento es omitido, muestra los números primos hasta el primer argumento.
primes	Muestra todos los números primos entre el punto de inicio y final-1. Este es más lento que prime_range, pero utiliza menos memoria.
primes_first_n	Muestra los n primeros números primos.
prod	Muestra el producto de los elementos en la lista x.
random	Muestra un numero aleatorio en el rango de [0.0, 1.0]
random_prime	Muestra un numero aleatorio primo entre 2 y n.
randrange	Escoge un elemento aleatorio del rango dado.
range	Muestra una lista con el progreso aritmético de los enteros, cuando se da el tercer valor (step) indica la forma de incremento o decremento.
real	Regresa la parte real de x.
reset	Elimina todas las variables creadas por el usuario y reinicializa las variables globales a su estado inicial. Si se especifica la variable solo se restaura el valor de dicha variable y las demás se dejan intactas.

restore	Restaura las variables globales predefinidas, si se especifica la variable, solo restaura la variable dada.
round	Redondea un numero a una precisión dada en decimales (0 si no se especifica nada).
sample	
save	Guarda un objeto al archivo con el nombre del archivo, el cual tendrá una extensión .sobj si no se especifica una.
save_session	Guarda todas las variables que pueden ser guardadas al nombre de archivo dado. Las variables serán guardadas en un diccionario el cual puede ser cargado utilizando load(nombre) o load_session.
search_doc	Búsqueda completa de texto en la documentación de SAGE HTML de líneas conteniendo s. La búsqueda no es sensible a mayúsculas y minúsculas.
search_src	Busca en la librería del código fuente de SAGE. La búsqueda no es sensible a mayúsculas y minúsculas.
sec	La función secante.
sech	La función de secante hiperbólica.
seq	Una lista mutable de elementos con un universo común garantizado el cual se puede establecer como inmutable. (Un universo es un objeto que soporta coerción o una categoría).
set	Construye una colección sin orden de objetos únicos.
show	Muestra un objeto grafico x.
show_default	Establece los parámetros para mostrar graficas utilizando cualquier comando de graficas.
sigma	Realiza la suma de las k potencias de los divisores de n.
simplify	Simplifica la expresión f.
sin	La función seno.
sinh	La función seno hiperbólico.
sleep	Retarda una ejecución por un tiempo dado en segundos.
slice	Crea un objeto del tipo slice.
solve	Resuelve algebraicamente una ecuación de un sistema de ecuaciones para las variables dadas.
sqrt	La función de raíz cuadrada.
srange	Crea una lista de números desde start (inicio) a end (fin) con un rango de incremento. Si el rango de incremento es negativo se toma orden descendente.
str	Muestra una representación en cadena (string) del objeto.
subfactorial	Numero de permutaciones de n elementos sin puntos arreglados.
sum	Muestra la suma de una secuencia de números, mas el valor del parámetro start (inicio), cuando la secuencia esta vacía regresa el valor start.
symbolic_expression	Crea una expresión simbólica de x.
sys	Este modulo provee acceso a algunos objetos utilizados o mantenidos por el interprete y a funciones que interactúan fuertemente con el interprete.
tan	La función tangente.

tanh	La función tangente hiperbólica.
taylor	Expande la ecuación en forma de las series de Taylor o Laurent en la variable v alrededor del punto a, conteniendo términos a través de (x – a)^n.
transpose	La transpuesta de la matriz x.
trial_division	Muestra el divisor primo mas pequeño de n.
two_squares	Escribe el entero n como una suma de los cuadrados de 2 enteros si es posible, de lo contrario despliega un mensaje de error 'ValueError'.
type	Muestra el tipo de la variable nombrada.
union	Combina x y y en una lista, la lista resultante no necesita ser ordenada y puede cambiar de llamada en llamada.
uniq	Muestra la sublista de todos los elementos en la lista x que esta ordenada y tal que las entradas en la sublista son únicas.
valuation	La potencia exacta de p que divide m. m debe ser un entero o racional.
var	Crea una variable simbólica.
vector	Crea un vector con las entradas dadas.
version	Muestra la versión de SAGE.
view	Calcula una representación en latex de cada objeto en objects, compile y display utilizando xdvi (requiere que la computadora tenga instalados latex y xdvi).
walltime	Muestra el tiempo de pared en segundos, o con argumento optimo t, muestra el tiempo de pared desde el tiempo t. "Tiempo de pared" significa el tiempo en un reloj de pared, por ejemplo la hora actual.
xrange	Parecida a range(), pero en lugar de mostrar una lista, muestra un objeto que genera los números del rango deseado. Para crear lazos este es ligeramente mas rápido que range() y mas eficiente en memoria.
zip	La función zip toma secuencias múltiples y agrupa miembros paralelos dentro de registros en una lista de salida.

Tabla 3: Sub conjunto de funciones de SAGE

3.22 Obteniendo información de funciones de SAGE

La tabla 3 incluye una lista de funciones junto con una descripción corta de lo que cada una hace. Sin embargo, esta no es información suficiente para mostrar como se usan estas funciones. Una forma de obtener información adicional en cualquier función es escribiendo su nombre seguido de un signo de interrogación '?' en una celda de la hoja de trabajo y posteriormente presionar la tecla <tab>:

```
is_even?<tab>
```

```
File: /opt/sage-2.7.1-debian-32bit-i686-
Linux/local/lib/python2.5/site-packages/sage/misc/functional.py
Type:           <type 'function'>
Definition:     is_even(x)
Docstring:

    Return whether or not an integer x is even, e.g., divisible by 2.

    EXAMPLES:
        sage: is_even(-1)
        False
        sage: is_even(4)
        True
        sage: is_even(-2)
        True
```

Una ventana gris se mostrará con la siguiente información acerca de la función:

File: Muestra el nombre del archivo que contiene el código fuente que implementa la función. Esto es útil si quieres localizar el archivo para ver como es implementada la función o editarla.

Type: Indica el tipo del objeto al cual se refiere el nombre pasado al servicio de información.

Definition: Muestra como es llamada la función.

Docstring: Despliega la secuencia de documentación que ha sido colocada en el código fuente de esta función.

Se puede obtener información en cualquiera de las funciones enlistadas en la tabla 3, o en el manual de referencia de SAGE usando esta técnica. También, si se colocan 2 signos de interrogación '??' después de el nombre de la función y se presiona la tecla <tab>, se mostrará el código fuente de la función.

3.23 La información también esta disponible en funciones creadas por el usuario

El servicio de información también puede ser usado para obtener información de funciones creadas por el usuario, y se puede lograr un mejor entendimiento de cómo trabaja el servicio de información intentando esto por lo menos una vez.

Si no lo ha hecho en la hoja de trabajo actual, escriba la función sumnums otra vez y ejecútela:

```
def sumnums(num1, num2):
        """
        Realiza la suma de num1 y num2.
        """
        resp = num1 + num2
        return resp
#Llama la función y suma 2 al 3.
a = sumnums(2, 3)
print a
```

```
        5
```

Posteriormente se obtiene información de esta nueva función utilizando la técnica de la sección anterior:

```
sumnums?<tab>
```

```
File: /home/sage/sage_notebook/worksheets/root/9/code/8.py
Type: <type 'function'>
Definition: sumnums(num1, num2)
Docstring:

Realiza la suma de num1 y num2.
```

Esto muestra que la información que es mostrada sobre una función es obtenida del código fuente de la función.

3.24 Ejemplos que utilizan funciones incluidas con SAGE

Los siguientes programas cortos muestran como son utilizadas algunas de las funciones mostradas en la tabla 3.

```
#Determinar la suma de los números del 1 al 10.
add([1,2,3,4,5,6,7,8,9,10])
|
        55

#Coseno de 1 radian.
cos(1.0)
|
        0.540302305868140

#Determinar el denominador de 15/64.
denominator(15/64)
|
        64

#Obtener una lista que contenga todos los enteros positivos
# divisores de 20.
divisors(20)
|
        [1, 2, 4, 5, 10, 20]

#Determinar el mayor común divisor de 40 y 132.
gcd(40,132)
|
        4

#Determinar el producto de 2, 3 y 4.
mul([2,3,4])
|
        24

#Determinar la longitud de una lista.
a = [1,2,3,4,5,6,7]
len(a)
|
        7
```

```
#Crear una lista que contenga los enteros del 0 al 10.
a = srange(11)
a
|
        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]


#Crear una lista que contenga números reales entre
# 0.0 y 10.5 de .5 en .5.
a = srange(11,step=.5)
a
|
      [0.0000000, 0.5000000, 1.000000, 1.500000, 2.000000, 2.500000,
      3.000000, 3.500000, 4.000000, 4.500000, 5.000000, 5.500000,
      6.000000, 6.500000, 7.000000, 7.500000, 8.000000, 8.500000,
      9.000000, 9.500000, 10.00000, 10.50000]


#Crear una lista que contenga enteros del -5 al 5.
a = srange(-5,6)
a
|
        [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]


#La función zip toma secuencias múltiples y agrupa
#miembros paralelos dentro de registros en una lista de salida. Esta
#aplicación es útil para crear puntos de tablas de datos
#para que puedan ser graficados.
a = [1,2,3,4,5]
b = [6,7,8,9,10]
c = zip(a,b)
c
|
        [(1, 6), (2, 7), (3, 8), (4, 9), (5, 10)]
```

3.25 Utilizando srange() y zip() con la instrucción for

En vez de crear una secuencia manualmente para el uso de la instrucción **for**, srange() puede ser utilizado para crear la secuencia automáticamente:

```
for t in srange(6):
print t,
```

```
0 1 2 3 4 5
```

La instrucción **for** también puede ser utilizada para enlazar múltiples secuencias en paralelo utilizando la función zip():

```
t1 = (0,1,2,3,4)
t2 = (5,6,7,8,9)
for (a,b) in zip(t1,t2):
print a,b
```

```
0 5
1 6
2 7
3 8
4 9
```

3.26 List Comprehensions (Compresiones de lista)

Hasta este punto hemos visto que las **instrucciones if, lazos for, listas** y **funciones** son extremadamente poderosas cuando son utilizadas ya sea individualmente o juntas. Sin embargo, lo que es inclusive más poderoso es una instrucción especial llamada **list comprehension** la cual les permite ser utilizadas en conjunto con un mínimo de sintaxis.

Aquí esta la sintaxis simplificada:
[*Expresión* **for** *variable* **in** *secuencia* [**if** *condición*] **]**

Lo que hace una comprensión de lista es crear un lazo a través de una secuencia colocando a cada miembro de la secuencia dentro de la variable especificada en turno. La expresión también contiene la variable y, como cada miembro es colocado dentro de la variable, la expresión es evaluada y el resultado es colocado en una nueva lista. Cuando todos los miembros en la secuencia han sido procesados, la nueva lista es mostrada.

En el siguiente ejemplo, **t** es la variable, **2*t** es la expresión y **[1, 2, 3, 4, 5]** es la secuencia:

```
a = [2*t for t in [0,1,2,3,4,5]]
a
|
        [0, 2, 4, 6, 8, 10]
```

En vez de crear manualmente la secuencia, la función **srange()** es utilizada comúnmente para crearla automáticamente:

```
a = [2*t for t in srange(6)]
a
|
        [0, 2, 4, 6, 8, 10]
```

Una instrucción **if** también puede ser utilizada opcionalmente en una comprensión de lista para filtrar los resultados que son colocados en la nueva lista:

```
a = [b^2 for b in range(20) if b % 2 == 0]
a
|
        [0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

En este caso, solamente los resultados que son divisibles equitativamente entre 2 (partes iguales) son colocados en la lista de salida.

4 Programación orientada a objetos

El propósito de este capitulo es de introducir los conceptos principales detrás de cómo funciona el código orientado a objetos de SAGE y como se emplea para resolver problemas. Se asume que se tiene poca o ninguna experiencia con la programación orientada a objetos (OOP – Object Oriented Programming) y proporcionara el entendimiento suficiente de la OOP para que pueda utilizar los objetos de SAGE más efectivamente para resolver problemas.

No hay que preocuparse demasiado si esto de la OOP no es comprendido inmediatamente dado que se pueden usar los objetos de SAGE para resolver problemas sin tener la habilidad requerida para programar objetos. Habiendo dicho esto, este capitulo muestra como programar un objeto desde cero, así se puede comprender mejor como funcionan los objetos pre-construidos de SAGE.

4.1 Reacomodo mental orientado a objetos

En mi opinión, una de las cosas más difíciles de lograr en el área de la programación es realizar el cambio mental del paradigma de la programación por procedimiento al paradigma de la programación orientada a objetos. El problema no es la programación orientada a objetos sea necesariamente mas difícil que la programación por procedimiento. El problema es que es tan diferente en su forma de resolver problemas de programación que tendrá que suceder algún reacomodo mental antes de que realmente se entienda. Este reacomodo mental es un proceso que ocurre muy lentamente conforme uno escribe programas orientados a objetos y escarba profundo en los libros orientados a objetos en un esfuerzo para realmente entender lo que es la OOP.

Justo desde el principio se vera que hay algo muy especial y poderoso sucediendo, pero eludirá sus esfuerzos para comprenderlo firmemente. Cuando finalmente se entienda no vendrá todo de golpe como al encender una luz brillante. Es mas como una luz tenue que se puede sentir brillando en la parte trasera del cerebro que va aumentando su brillo lentamente. Por cada nuevo problema de programación que se encuentre, la parte delantera del cerebro todavía desarrollara un plan de procedimientos para resolverlo. Sin embargo, uno notará que este brillo en la parte trasera del cerebro presentará estrategias orientadas a objetos (tenue al principio, pero lentamente incrementándose en claridad) que también resolverán el problema y estas estrategias orientadas a objetos son tan interesantes que con el paso del tiempo uno se encontrara prestando mucha mayor atención a ellas. Eventualmente llegara el tiempo cuando varios problemas de programación detonarán la producción de excelentes estrategias orientadas a objetos para resolverlos desde la ahora brillante parte orientada a los objetos de tu mente.

4.2 Atributos y comportamientos

La programación orientada a objetos es una filosofía de diseño de software donde el software esta hecho para trabajar de manera similar a como los objetos en el mundo similar trabajarían. Todos los objetos físicos tienen atributos y comportamientos. Un ejemplo es una típica silla de oficina que tiene un **color, número de ruedas** y **tipo de material** como atributos y sus comportamientos son que **gira, rueda** y **se ajusta su altura.**

Los objetos de software son hechos para trabajar como objetos físicos y por lo tanto también tienen atributos y comportamientos. Los **atributos** de un objeto de software son guardados en variables especiales llamadas **variables de instancia** y sus **comportamientos** son determinados por **código** el cual esta guardado en **métodos** (los cuales son llamados también **funciones de miembro)**. Los **Métodos** son similares a las funciones estándar excepto que son asociados a objetos en vez de estar flotando libremente. En SAGE, las variables de instancia y los métodos son comúnmente referidos como simplemente **atributos.**

Después de que un objeto es creado, es utilizado enviándole **mensajes**, lo que significa **llamar** o **invocar** sus métodos. En el caso de la silla, podríamos imaginar enviándole un mensaje de **silla.gira(3)** el cual le diría a la silla que gire 3 veces, o un mensaje de **silla.ajustaraltura(32)** que le diría a la silla que ajuste su altura a 32 centímetros.

4.3 Clases (Planos que son usados para crear objetos)

Una **clase** puede ser pensada como un **plano** que es usado para construir objetos y es conceptualmente similar a un plano de una casa. Un arquitecto utiliza un plano para definir precisamente como una casa debe ser construida, que materiales deberán ser usados, como deberán ser sus varias dimensiones, etc. Después de que el plano esta terminado, puede ser utilizado para construir una o varias casas dado que el plano contiene la información que describe como crear una casa, pero no es la casa en si. Un programador creando una **clase** es muy similar a un arquitecto creando un **plano de una casa** excepto que el arquitecto utiliza una mesa de dibujo o un sistema CAD para desarrollar el plano, mientras que el programador utiliza un editor de texto o una IDE (Integrated Development Environment – Ambiente Integrado de Desarrollo) para desarrollar una **clase.**

4.4 Programas orientados a objetos, crear y destruir objetos según sea necesario

La siguiente analogía describe como los objetos de software son creados y destruidos según sea necesario en la programación orientada a objetos. El acto de destruir un objeto y recuperar la memoria y otros recursos que estaba utilizando se llama **garbage collection** (recolección de basura)**.**

Imagine que un jet de pasajeros dado puede operar de una manera similar a un programa orientado a objetos y que el jet esta siendo preparado para volar a través del océano Atlántico de Nueva York a Londres. Justo antes de despegar, los planos para cada parte del avión son llevados a la pista y dados a un grupo de trabajadores que los van a utilizar para rápidamente construir todos los componentes necesarios para construir el avión. Conforme cada componente es construido, es agregado al lugar indicado en el avión y en poco tiempo el avión esta completo y listo para ser usado. Los pasajeros se suben al jet y despega.

Después de que el avión deja el suelo, el tren de aterrizaje es desintegrado (basura colectada) por que no es necesario durante el vuelo y llevarlos a través del océano Atlántico solo gastaría combustible costoso. Sin embargo, no hay necesidad de preocuparse, por que el tren de aterrizaje será reconstruido utilizando los planos adecuados (clases) justo antes de aterrizar en Londres.

Unos minutos después del despegue el piloto recibe una notificación de que la compañía que manufacturo las turbinas del jet acaba de sacar a la venta un nuevo modelo que es 15% mas eficiente en consumo de combustible que los que actualmente utiliza el avión y que la aerolínea va a actualizar las turbinas del avión mientras este vuela. La aerolínea manda los planos de las nuevas turbinas a través de la red hacia el avión y estos son utilizados para construir 3 de las nuevas turbinas. Después de que las nuevas turbinas son construidas, las 3 turbinas viejas son apagadas una a la vez, reemplazadas con las turbinas nuevas y desintegradas. La actualización de las turbinas corre suavemente y los pasajeros ni siquiera se enteran de que sucedió la actualización.

El vuelo lleva abordo a un personaje muy importante mundialmente y a mitad del vuelo un avión hostil aparece, el cual le ordena al piloto que cambie su curso. Sin embargo, en lugar de cumplir con su demanda, el piloto recupera de la librería un grupo de planos para una torrecilla de ametralladora de 50mm, hace que se construyan 4 de estas y que se agreguen a las secciones superior, inferior, nariz y cola del avión. Unos cuantos disparos de una de estas armas es suficiente para disuadir al avión hostil y rápidamente se aleja, desapareciendo eventualmente de la pantalla del radar. El resto del viaje es tranquilo. Conforme el avión se acerca a Londres, las armas son desintegradas, un nuevo set de tren de aterrizaje es construido utilizando los planos del tren de aterrizaje y el avión aterriza seguramente. Una vez que los pasajeros están en la terminal, el avión completo es desintegrado.

4.5 Ejemplo de programa orientado a objetos

Las siguientes 2 secciones cubren un simple programa orientado a objetos llamado **Holas**. La primera sección presenta una versión del programa que no contiene ningún comentario así que el código mismo es fácil de verse. La segunda sección contiene una versión del programa llena de comentarios junto con una descripción detallada de cómo funciona el trabajo.

4.5.1 Ejemplo de programa Holas orientado a objetos (sin comentarios)

```
class Holas:
  def  __init__(self, mess):
    self.message = mess
  def print_message(self):
    print"El mensaje es: ", self.message
  def decir_adios(self):
    print '"¡Adios!"
  def print_holas(self, total):
    conteo = 1
    while conteo <= total:
      print"Hola ", conteo
      conteo = conteo + 1
    print " "
obj1 = Holas("Todavia te diviertes?")
obj2 = Holas("Si!")
obj1.print_message()
obj2.print_message()
print " "
obj1.print_holas(3)
obj2.print_holas(5)
obj1.decir_adios()
obj2.decir_adios()
```

```
El mensaje es:  Todavia te diviertes?
El mensaje es:  Si!
Hola  1
Hola  2
Hola  3
Hola  1
Hola  2
Hola  3
Hola  4
Hola  5
Adios!
Adios!
```

4.5.2 Ejemplo de programa Holas orientado a objetos (con comentarios)

Ahora veremos el programa **Holas** de forma detallada. Esta versión del programa tiene comentarios agregados. Los números de línea y dos puntos a la izquierda del programa no son parte del programa en si y han sido agregados para hacer referencia a diferentes partes del programa más fácilmente.

```
1:class Holas:
2:      """
3:      Holas es una clase( 'class') y una clase es un plano para crear
4:      objetos. Las Clases consisten en variables de instancia (atributos)
5:      y métodos (comportamientos).
6:      """
7:
8:      def __init__(self, mess):
9:              """
10:             __init__ es un tipo especial de método integrado llamado
11:             constructor. Un método constructor solo es llamado una vez
12:             cuando un objeto es creado y su trabajo es completar
13:             la construcción del objeto. Después de que el objeto ha
14:             sido creado sus constructores no son utilizados mas. El
15:             propósito de este constructor es crear una variable de
16:             instancia llamada 'message' e inicializarla con una
17:             cadena.
18:             """
19:
20:              """
21:             Este código crea una variable de instancia. Cada objeto
22:             creado de este plano de clase tendrá
23:              variables que contengan atributos del objeto (o estado).
25:             La variable self aquí guarda la referencia al objeto
26:             actual.
27:             """
28:             self.message = mess;
29:
30:
31:
32:     def print_message(self):
33:             """
34:             print_message es un método de instancia que les da a los objetos
35:             creados utilizando esta clase su comportamiento de  'print message' .
36:             """
37:             print"El mensaje es: ", self.message
38:
39:
40:
41:     def decir_adios(self):
42              """
43:             decir_adios es un método de instancia que le da a los objetos
```

```
44:             creados utilizando esta clase su comportamiento de ‘decir adios’.
45:              """
46:              print "Adios!"
47:
48:
49:
50:     def print_holas(self, total):
51:              """
52:             print_holas es un método de instancia que toma el número
53:             de holas para imprimir un argumento e imprime todos estos
54:             Holas en la pantalla.
55:             """
56:             conteo = 1
57:             while conteo <= total:
58:                     print"Hola ", conteo
59:                     conteo = conteo + 1
60:
61:             print " "
62:
63:
64:"""
65:El siguiente código crea dos objetos Holas separados
66:los cuales son referenciados por las variables obj1 y obj2 respectivamente.
67:Un parámetro de cadena única es pasado a cada objeto cuando es
68:ejecutado y esta cadena es utilizada para inicializar el estado del
69:objeto.
70:
71:Después de que los objetos son creados, se envían mensajes a ellos
72:llamando sus métodos para que realicen sus comportamientos.
73:Esto se logra ‘Tomando un objeto’ por su referencia (digamos
74:obj1) colocando un punto después de esta referencia y después escribiendo
75:el nombre del método de un objeto que se desea invocar.
76:"""
77:
78:obj1 = Holas("Todavia te diviertes?")
79:obj2 = Holas("Si!")
80:
81:obj1.print_message()
82:obj2.print_message()
83:print " "
84:
85:obj1.print_holas(3)
86:obj2.print_holas(5)
87:
88:obj1.decir_adios()
89:obj2.decir_adios()
```

En la línea 1 la clase Holas esta definida utilizando una instrucción de **clase** y por convención los nombres de clases inician con mayúscula. Si el nombre de clase consiste en palabras múltiples, entonces la primera letra de cada palabra es mayúscula y todas las demás son escritas en minúsculas (por ejemplo, HolaMundo). La clase inicia en la línea 1 y termina en la línea 61, la cual es la última línea de código con sangría. Todos los **métodos** y **variables de instancia** que son parte de una clase necesitan estar dentro del bloque de código con sangría.

La clase de Holas contiene un método **constructor** en la línea 8, una **variable de instancia** la cual es creada en la línea 28, y 3 **métodos de instancia** en las líneas 32, 41 y 50 respectivamente. El propósito de las **variables de instancia** es darle a un objeto **atributos** únicos que lo diferencien de otros objetos que son creados de una clase dada. El propósito de los **métodos de instancia** es darle a cada objeto sus **comportamientos.** Todos los métodos en un objeto tienen acceso a esas variables de instancia del objeto y estas pueden ser accedidas por el código en estos métodos. Los nombres de las variables de instancia siguen la misma convención que los nombres de funciones.

El método en la línea 8 es un método especial llamado **constructor.** Un método **constructor** solo es llamado cuando un objeto esta siendo creado y su propósito es completar la construcción de ese objeto. Después de que el objeto ha sido creado, su constructor ya no es usado. El propósito del constructor en la línea 8 es inicializar cada variable de instancia de mensaje de los objetos Holas con una cadena que es pasada cuando un objeto nuevo del tipo Holas es creado (ver líneas 78 y 79).

Todos los métodos de instancia tienen un argumento pasado a ellos que contiene una referencia para el objeto específico del cual fue llamado el método. Este argumento siempre es colocado lo más a la izquierda de la posición de argumento y por convención, la variable que es colocada en esta posición es llamada **self**. La variable **self** entonces utilizada para crear y acceder a esas variables de instancia de objeto específicas.
En la línea 28, el código **self.message = mess** toma el objeto que fue pasado a la variable del constructor **mess** y la asigna a una variable de instancia llamada **message.** Una variable de instancia es creada mediante asignación, justo como las variables normales. El operador punto "." Es utilizado para acceder a las variables de instancia de un objeto colocándolo entre la variable que tiene la referencia al objeto y el nombre de la variable de instancia (como self.message o obj1.message).

Los métodos en las líneas 32, 41 y 50 le da sus comportamientos a los objetos creados utilizando la clase Holas. El método print_message() provee el comportamiento de imprimir la cadena que esta presente en la variable de instancia del mensaje de objeto y el método say_goodbye() provee el comportamiento de imprimir la cadena "Goodbye!". El método print_hellos() toma un número entero como parámetro e imprime la palabra 'Hello' esa cantidad de veces. La convención de nombres para los métodos es la misma que la usada para los nombres de las funciones.

El código debajo de la clase Hellos crea dos objetos separados los cuales son posteriormente asignados a las variables **obj1** y **obj2** respectivamente. Un objeto es creado escribiendo su nombre de clase seguido de un par de paréntesis. Cualquier argumento que sea colocado entre los paréntesis será pasado al método constructor.

Cuando la clase Holas es llamada, una secuencia es pasada al método constructor y la secuencia es utilizada para inicializar el **estado** del objeto. El estado del objeto es determinado por los contenidos de sus variables de instancia. Si cualquiera de las variables de instancia de un objeto son cambiadas, entonces el estado del objeto ha sido cambiado también. Como los objetos Holas solo tienen una variable de instancia llamada **message**, su estado esta determinado por esta variable.

Después de que son creados los objetos, sus comportamientos son requeridos llamando sus métodos. Esto es hecho "Tomando un objeto" por una variable que lo referencia (digamos obj1), colocando un punto después de esta variable, y posteriormente escribiendo el nombre de uno de los métodos del objeto que se desee invocar, seguido de sus argumentos en paréntesis.

4.6 Clases y Objetos en SAGE

Mientras que las funciones de SAGE contienen muchas capacidades, la mayor parte de las capacidades de SAGE están contenidas en clases y los objetos que son cargados desde estas clases. Las clases y objetos en SAGE representan una cantidad de información significativa que tomaría cierto tiempo explicar. Sin embargo, el material más fácil será presentado primero para que se pueda comenzar a trabajar con los objetos de SAGE lo más pronto posible.

4.7 Obteniendo información de los objetos de SAGE

Escriba el siguiente código en una celda y ejecútelo:

```
x = 5
print type(x)
```

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

Ya hemos usado la función type() para determinar el tipo de un entero, pero ahora podemos explicar mas detalladamente lo que es un tipo. Introducir sage.rings.integer.Integer seguido de un signo de interrogación " ? " en una celda nueva y posteriormente presione la tecla <tab>:

sage.rings.integer.Integer?<tab>

```
File:/opt/sage-2.7.1-debian-32bit-i686-
Linux/local/lib/python2.5/site-packages/sage/rings/integer.so
Type:   <type 'sage.rings.integer.Integer'>
Definition: sage.rings.integer.Integer([noargspec])

Docstring:

        The class{Integer} class represents arbitrary precision
        integers.  It derives from the class{Element} class, so
        integers can be used as ring elements anywhere in SAGE.

        begin{notice}
        The class class{Integer} is implemented in Pyrex,
        as a wrapper of the GMP mpz_t integer type.
        end{notice}
```

Esta información indica que sage.rings.integer.Integer es realmente una clase que es capaz de crear objetos enteros. También, si se colocan 2 signos de interrogación '??' después de un nombre de clase y de ahí se presiona la tecla tab, el código fuente de la clase será mostrado.

Ahora en una celda separada escriba x y presione la tecla tab:

```
x.<tab>
```

```
x.additive_order          x.gcd                     x.numerator
x.base_base_extend        x.inverse_mod             x.ord
x.inverse_of_unit         x.order                   x.parent
x.base_extend             x.is_nilpotent            x.plot
x.base_extend_canonical   x.is_one                  x.powermodm_ui
x.is_perfect_power        x.powermod                x.quo_rem
x.base_extend_recursive   x.is_power                x.rename
x.base_ring               x.is_power_of             x.reset_name
x.binary                  x.is_prime                x.save
x.category                x.is_prime_power          x.set_si
x.ceil                    x.is_pseudoprime          x.set_str
x.coprime_integers        x.is_square               x.sqrt
x.crt                     x.is_squarefree           x.sqrt_approx
x.db                      x.is_unit                 x.square_free_part
x.degree                  x.is_zero                 x.str
x.denominator             x.isqrt                   x.substitute
x.digits                  x.jacobi                  x.test_bit
x.div                     x.kronecker               x.val_unit
x.lcm                     x.subs                    x.valuation
x.divides                 x.leading_coefficient     x.version
x.dump                    x.list                    x.xgcd
x.dumps                   x.mod                     x.parent
```

```
x.exact_log               x.multiplicative_order   x.plot
x.factor                  x.next_prime             x.rename
x.factorial               x.next_probable_prime    x.reset_name
x.floor                   x.nth_root               x.powermodm_ui
```

Una ventana gris será mostrada, la cual contiene todos los métodos que contiene el objeto. Si cualquiera de estos métodos es seleccionado con el cursor del ratón, su nombre será colocado en la celda después del operador punto como comodidad. Por ahora, seleccione el método **is_prime**. Cuando su nombre sea colocado en la celda, escriba un signo de interrogación y presione la tecla tab para obtener información de este método:

```
x.is_prime?
```

```
File:       /opt/sage-2.7.1-debian-32bit-i686-Linux/local/lib/python/
site-packages/sage/rings/integer/pyx
Type:       <type 'builtin_function_or_method '>
Definition: x.is_prime()

Docstring:

            Retuns True if self is prime

            EXAMPLES:
                sage: z = 2^31 - 1
                sage: z.is_prime()
                True
                sage: z = 2^31
                sage: z.is_prime()
                False
```

La sección Definition indica que el método is_prime() es llamado sin pasar ningún argumento a el y la sección Docstring indica que el método será verdadero si el objeto es primo. El siguiente código muestra la variable x (que todavía vale 5) siendo usada para llamar al método is_prime:

```
x.is_prime()
```

```
True
```

4.8 Los métodos de los objetos de la lista

Las listas son objetos y por lo tanto contienen métodos que proveen capacidades útiles:

```
a = []
a.<tab>
```

```
a.append   a.extend   a.insert   a.remove   a.sort
a.count    a.index    a.pop      a.reverse
```

Los siguientes programas demuestran algunos de los métodos de los objetos de una lista:

```
#  Añadir un objeto al final de una lista.
a = [1,2,3,4,5,6]
print a
a.append(7)
print a
```

```
        [1, 2, 3, 4, 5, 6]
        [1, 2, 3, 4, 5, 6, 7]
```

```
# Insertar un objeto a una lista.
a = [1,2,4,5]
print a
a.insert(2,3)
print a
```

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

```
# Clasificar los contenidos de una lista.
a = [8,2,7,1,6,4]
print a
a.sort()
print a
```

```
        [8, 2, 7, 1, 6, 4]
        [1, 2, 4, 6, 7, 8]
```

4.9 Extendiendo las clases con herencias.

Las tecnologías de objetos son sutiles y poderosas. Poseen un gran número de mecanismos para tratar con casos complejos y las herencias de clase es uno de ellos. La **herencia de clase** es la habilidad de una clase de obtener o heredar todas las variables de instancia y métodos de otra clase (llamada **clase parental, super clase, o clase base)** utilizando una cantidad mínima de código. Una clase que hereda o sucede de una clase parental es llamada **clase hija** o **sub clase.** Esto significa que una clase hija puede hacer todo lo que hace su clase padre junto con otra funcionalidad adicional que es programada en la hija.

El siguiente programa demuestra la herencia de clases dejando que una clase **Persona** herede información de la clase construida en la clase **objeto** y a la vez que la clase **ArmyPrivate** lo haga de la clase Persona:

```
class Persona(object):
        def __init__(self):
                rank.prop = "Solo soy una persona, no tengo rango."
        def __str__(self):
                return "str: " + rank.prop
        def __repr__(self):
                return "repr: " + rank.prop
class Soldado(Persona):
        def __init__(self):
                rank.prop = "Soldado."
a = object()
print type(a)
b = Persona()
print type(b)
c = Soldado()
print type(c)
```

```
        <type 'object'>
        <class '__main__.Person'>
        <class '__main__.ArmyPrivate'>
```

Después de que las clases han sido creadas, este programa ejecuta un objeto del tipo **objeto** el cual es asignado a la variable ‘a’, un objeto del tipo **Persona** que es asignado a la variable ‘b’, y un objeto del tipo **ArmyPrivate** el cual es asignado a la variable ‘c’.

El siguiente código puede ser utilizado para desplegar la jerarquía de ejecución de cada objeto. Si es ejecutado en una celda separada después de que el programa de arriba ha sido ejecutado, se muestra la jerarquía de la clase **ArmyPrivate** (Por el momento no es necesario preocuparse por entender como funciona este código. Solo usémoslo.)

```
#Desplegar la jerarquía de la herencia de un objeto. Nota: no te preocupes
#por entender como funciona este programa. Solo úsalo por
#ahora
def class_hierarchy(cls, indent):
        print '.'*indent, cls
        for supercls in cls.__bases__:
                class_hierarchy(supercls, indent+1)
def instance_hierarchy(inst):
        print 'Inheritance hierarchy of', inst
        class_hierarchy(inst.__class__, 3)
z = Soldado()
instance_hierarchy(z)
```

```
Inheritance hierarchy of str: Soldado
... <class '__main__.Soldado'>
.... <class '__main__.Persona'>
..... <type 'object'>
```

La función instance_hierarchy mostrará la jerarquía de herencia de cualquier objeto que sea pasado a ella. En este caso, un objeto ArmyPrivate fue ejecutado y pasado a la función instance_hierarchy y la jerarquía de herencia del objeto fue mostrada. Note que la clase de la parte mas alta de la jerarquía, que es la clase **objeto**, fue escrita al final y que **Persona** hereda de **objeto** y **ArmyPrivate** hereda de **Persona.**

4.10 La clase object, la función dir() y los métodos incorporados.

La clase **object** esta incorporada en SAGE y contiene un pequeño numero de métodos útiles. Estos métodos son tan útiles que varias clases en SAGE los heredan de la clase **object** ya sea 1) Directamente o 2) Indirectamente heredándolo de una clase que lo heredo de la clase object. Comencemos nuestra discusión del programa de herencias observando los métodos que están incluidos en la clase **object**. La función **dir()** enlista todos los atributos de un objeto (que son sus variables de ejecución y sus métodos) y podemos usarla para ver cuales métodos contiene un objeto del tipo **object**:

```
dir(a)
|
        ['__class__', '__delattr__', '__doc__',
        '__getattribute__','__hash__','__init__', '__new__', '__reduce__',
        '__reduce_ex__', '__repr__','__setattr__', '__str__']
```

Los nombres que inician y terminan con doble guion bajo '__' son parte de SAGE y los guiones ayudan a evitar que estos nombres hagan conflicto con los nombres definidos por el programador. La clase Persona hereda todos estos atributos de la clase **object**, pero solo utiliza algunos de ellos. Cuando un método es heredado de una clase parental, la clase hija puede utilizar la implementación de ese método como la clase parental o puede redefinirla para que se comporte diferente a la versión original.

Como se menciono anteriormente, el método __init__ es un constructor y ayuda a completar la construcción de cada objeto nuevo que es creado utilizando la clase en la que esta. La clase Persona redefine el método __init__ para que cree una variable de ejecución llamada **rango** y le asigna la cadena "Solo soy una persona, no tengo rango" a ella.

Los métodos **__repr__** y **__str__** son también redefinidos en la clase Persona. El método __repr__ muestra una representación de la cadena del objeto del cual es parte:

```
b
|
        repr: Solo soy una persona, no tengo rango.
```

La función __str__ también muestra una representación de la cadena del objeto del cual es parte, pero solo cuando se pasa a instrucciones como print:

```
print b
|
        str: I am just a Person, I have no rank.
```

El método __str__ es usualmente empleado para proveer una cadena mas amigable al usuario que el método __repr__ pero en este ejemplo, son mostradas cadenas muy parecidas.

4.11 La jerarquía de herencia de la clase sage.ring.integer.Integer

El siguiente código muestra la jerarquía de herencia de la clase sage.rings.integer.Integer:

```
#Despliegue de la jerarquía de herencia de un objeto. Nota: no te preocupes
#por tratar de entender como funciona el programa. Solo úsalo por
#ahora.
def class_hierarchy(cls, indent):
        print '.'*indent, cls
        for supercls in cls.__bases__:
                class_hierarchy(supercls, indent+1)
def instance_hierarchy(inst):
        print 'Jerarquía de herencia de', inst
        class_hierarchy(inst.__class__, 3)
instance_hierarchy(1)
|
        Jerarquía de herencia de 1
        ... <type 'sage.rings.integer.Integer'>
        .... <type 'sage.structure.element.EuclideanDomainElement'>
        ..... <type 'sage.structure.element.PrincipalIdealDomainElement'>
        ...... <type 'sage.structure.element.DedekindDomainElement'>
        ....... <type 'sage.structure.element.IntegralDomainElement'>
        ........ <type 'sage.structure.element.CommutativeRingElement'>
        ......... <type 'sage.structure.element.RingElement'>
        .......... <type 'sage.structure.element.ModuleElement'>
        ........... <type 'sage.structure.element.Element'>
        ............ <type 'sage.structure.sage_object.SAGEObject'>
        ............. <type 'object'>
```

En la siguiente explicación, se harán a un lado la parte inicial “sage.xxx.xxx…” de los nombres de clase para ahorrar espacio. La salida de la función **instance_hierarchy** indica que el número 1 es un objeto del tipo **Integer (entero).** Posteriormente muestra que Integer es heredero de **EuclideanDomainElement**, el cual es heredero de **PrincipalIdealDomainElement**, etc. En la cima de la jerarquía (que es hasta el fondo de la lista) SAGEObject es heredero de object.

Aquí esta una jerarquía de herencia de otros 2 objetos de SAGE comúnmente usados:

```
instancehierarchy(1/2)
|
        Inheritance hierarchy of 1/2
        ... <type 'sage.rings.rational.Rational'>
        .... <type 'sage.structure.element.FieldElement'>
        ..... <type 'sage.structure.element.CommutativeRingElement'>
        ...... <type 'sage.structure.element.RingElement'>
        ....... <type 'sage.structure.element.ModuleElement'>
        ........ <type 'sage.structure.element.Element'>
        ......... <type 'sage.structure.sage_object.SAGEObject'>
        .......... <type 'object'>

instancehierarchy(1.2)
|
        Inheritance hierarchy of 1.20000000000000
        ... <type 'sage.rings.real_mpfr.RealNumber'>
        .... <type 'sage.structure.element.RingElement'>
        ..... <type 'sage.structure.element.ModuleElement'>
        ...... <type 'sage.structure.element.Element'>
        ....... <type 'sage.structure.sage_object.SAGEObject'>
        ........ <type 'object'>
```

4.12 La relación “Is A”- (Es)

Otro aspecto del concepto de herencia es que, puesto que una clase hija puede hacer todo lo que su padre puede hacer, puede ser usada en cualquier lugar en el cual su objeto padre puede ser usada. Observando la jerarquía de herencia de la clase Integer. Esta jerarquía indica que **Integer** es **EuclideanDomainElement** y **EuclideanDomainElement** es **PrincipalIdealDomainElement** y **PrincipalIdealDomainElement** es **DedekindDomainElement** etc. hasta que finalmente **SAGEObject** es object (Justo como si casi todas las otras clases están en SAGE puesto que el la clase object es la clase raíz de la cual todas las demás descienden). Una forma más general de verlo es diciendo que una clase hija puede ser utilizada en cualquier lugar en donde sus clases antepasadas se usan.

4.13 ¿Confundido?

Este capitulo probablemente fue confuso pero nuevamente, no hay que preocuparse. El resto de este libro contendrá ejemplos que muestran como los objetos son usados en SAGE y mientas mas objetos siendo usados se vean, mas fácil será entenderlos.

5 Temas variados

5.1 Referenciando el resultado de la operación anterior

Cuando se trabaja en un problema que despliega múltiples celdas en una hoja de trabajo, comúnmente es deseable referenciar el resultado de la operación anterior. El símbolo de guión bajo '_' es usado para este propósito como se muestra en el siguiente ejemplo:

```
2 + 3
|
        5
_
|
        5
_ + 6
|
        11
a = _ * 2
a
|
        22
```

5.2 Excepciones

Con el fin de asegurar que los programas de SAGE tengan una forma uniforme para manejar condiciones excepcionales que podrían ocurrir mientras estos se están ejecutando, un despliegue de excepciones y un mecanismo para manejarlas esta integrado en la plataforma SAGE. Esta sección cubre solamente las excepciones mostradas debido a que su manejo es un tópico avanzado que esta más allá del alcance de este documento.

El siguiente código provoca una excepción y es mostrada la información de esta:

```
1/0
|
Exception (click to the left for traceback):
...
ZeroDivisionError: Rational division by zero
```

Como 1/0 es una operación matemática indefinida, SAGE es incapaz de realizar el cálculo. Detiene la ejecución del programa y genera una excepción para informar a otras aéreas del programa o al usuario sobre este problema. Si ninguna otra parte del programa maneja la excepción, se mostrará una explicación textual de la excepción. En este caso la excepción informa al usuario que ha ocurrido una excepción del tipo ZeroDivisionError, y esta fue causada por intentar realizar una división racional entre cero.

La mayor parte del tiempo, esta es información suficiente para que el usuario localice el problema en el código fuente y lo arregle. Sin embargo, algunas veces, el usuario necesita mas información para localizar el problema y por eso la excepción indica que si se realiza un click con el puntero del mouse al lado izquierdo del texto de la excepción mostrada, se desplegara información adicional:

```
Traceback (most recent call last):
File "", line 1, in
File "/home/sage/sage_notebook/worksheets/tkosan/2/code/2.py",
line 4, in
Integer(1)/Integer(0)
File "/opt/sage-2.8.3-linux-32bit-debian-4.0-i686-
Linux/data/extcode/sage/", line 1, in
File "element.pyx", line 1471, in element.RingElement.__div__
File "element.pyx", line 1485, in element.RingElement._div_c
File "integer.pyx", line 735, in integer.Integer._div_c_impl
File "integer_ring.pyx", line 185, in
integer_ring.IntegerRing_class._div
ZeroDivisionError: Rational division by zero
```

Esta información adicional muestra una pista de todo el código en la librería de SAGE que estaba en uso cuando ocurrió la excepción junto con los nombres de los archivos que tenían el código. A un usuario experto de SAGE le permite ver el código fuente si es necesario con el fin de determinar si la excepción fue causada por un error (BUG) en SAGE o un error en el código introducido.

5.3 Obteniendo resultados numéricos

Algunas veces uno necesita obtener la aproximación numérica de un objeto y SAGE provee varias formas para lograr esto. Una manera es utilizar la **función n()** y otra forma es utilizando el **método n()**. El siguiente ejemplo muestra a ambos de estos en uso:

```
a = 3/4
print a
print n(a)
print a.n()
```

```
3/4
0.750000000000000
0.750000000000000
```

El número de dígitos mostrados puede ser ajustado utilizando el parámetro **digits**:

```
a = 3/4
print a.n(digits=30)
|
        0.750000000000000000000000000000
```

Y el número de bits de precisión puede ser ajustado utilizando el parámetro **prec**:

```
a = 4/3
print a.n(prec=2)
print a.n(prec=3)
print a.n(prec=4)
print a.n(prec=10)
print a.n(prec=20)
|
        1.5
        1.2
        1.4
        1.3
        1.3333
```

5.4 Guía de estilos para expresiones

Siempre hay que rodear los siguientes operadores binarios con un espacio sencillo en cualquiera de los 2 lados: asignación '=', asignación aumentada (+ =, - =, etc), comparaciónes(= =, <, >, ¡ =, < >, < =, > =, in, not in, is, is not), Booleanas (and, or, not).

Usar espacios alrededor de los operadores aritméticos + y -, y sin espacios en los operadores *, /, % y ^:

```
x = x + 1
x = x*3 − 5%2
c = (a + b)/(a − b)
```

No utilizar espacios alrededor del signo '=' cuando este es usado para indicar un valor predefinido de parámetro:

```
a.n(digits=5)
```

5.5 Constantes integradas

SAGE tiene varias constantes matemáticas integradas y la siguiente es una lista de las más comunes:

Pi, pi: La proporción de la circunferencia del diámetro de un círculo.
E, e: Base del logaritmo natural.
I, i: El número imaginario.
log2: El logaritmo natural del número real 2.
Infinity, infinity: Puede tener un + o – antes de el para indicar infinito positivo o negativo.

Los siguientes ejemplos muestran a las constantes en uso:

```
a = pi.n()
b = e.n()
c = i.n()
a,b,c
```

```
(3.14159265358979, 2.71828182845905, 1.00000000000000*I)
```

```
r = 4
a = 2*pi*r
a,a.n()
```

```
(8*pi, 25.1327412287183)
```

Las constantes en SAGE son definidas como variables globales y una **variable global** es una variable accesible por la mayoría del código de SAGE, incluyendo dentro de funciones y métodos. Como las constantes son simplemente variables que tienen un objeto constante asignado a ellas, las variables pueden ser reasignadas si es necesario pero el objeto constante se pierde. Si uno necesita reasignar una constante a la variable que esta normalmente asociada, la función **restore()** puede ser utilizada. El siguiente programa muestra como la variable **pi** puede tener el objeto 7 asignado a ella y después tener su constante por defecto asignada a ella de nuevo colocando el nombre entre comillas en la función **restore()**:

```
print pi.n()
pi = 7
print pi
restore('pi')
print pi.n()
```

```
3.14159265358979
7
3.14159265358979
```

Si se utiliza la función restore() sin parámetros, todas las constantes reasignadas son restauradas a sus valores originales.

5.6 Raíces

La función sqrt() puede ser utilizada para obtener la raíz cuadrada de un valor, pero una técnica más general es utilizada para obtener las raíces de un valor. Por ejemplo, si uno desea obtener la raíz cúbica de 8:

$\sqrt[3]{8}$

8 sería elevado a la potencia 1/3:

```
8^(1/3)
```

```
2
```

Dado el orden de operaciones, el número racional 1/3 necesita ser colocado entre paréntesis para que pueda ser evaluado como exponente.

5.7 Variables simbólicas

Hasta este punto, todas las variables que hemos utilizado han sido creadas por medio de asignación. Por ejemplo, en el siguiente código la variable **w** es creada y posteriormente se le asigna el valor 7:

```
w = 7
w
```

```
7
```

Pero, ¿Qué tal si se necesitara trabajar con variables que no tengan asignados ningún valor específico? El siguiente código intenta imprimir el valor de la variable z, pero a z no se le ha asignado ningún valor todavía, así que ocurre una excepción:

```
print z
```

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

En matemáticas, las “Varibles sin asignar” son utilizadas todo el tiempo. Dado que SAGE es un software orientado a las matemáticas, tiene la capacidad de trabajar con variables sin asignar. En SAGE, las variables sin asignar son llamadas **variables simbólicas** y son definidas utilizando la función **var()**. Cuando una hoja de trabajo se abre por primera vez, la variable **x** es definida automáticamente para ser una variable simbólica y permanecerá así a menos que se le asigne otro valor en el código.

El siguiente código fue ejecutado en una hoja de trabajo nueva:

```
print x
type(x)
```

```
        x
        <class 'sage.calculus.calculus.SymbolicVariable'>
```

Noten que la variable **x** tenía un objeto del tipo **SymbolicVariable** automáticamente asignado a ella por el entorno SAGE.

Si se necesitaran utilizar también **y** y **z** como variables simbólicas, la función **var()** es necesaria para esto. Puede introducirse ya sea como **var('x,y')** o **var('x y').** La función **var()** está diseñada para aceptar uno o más nombres de variables dentro de una secuencia y los nombres pueden estar separados ya sea por comas o espacios.

El siguiente programa muestra **var()** siendo usada para inicializar **y** y **z** para ser variables simbólicas:

```
var('y,z')
y,z
```

```
        (y, z)
```

Después de que una o más variables simbólicas han sido definidas, la función **reset()** puede ser utilizada para deshacer esto:

```
reset('y,z')
y,z
```

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

5.8 Expresiones simbólicas

Las expresiones que contengan variables simbólicas son llamadas **expresiones simbólicas**. En el siguiente ejemplo, **b** es definida como una variable simbólica y posteriormente es utilizada para crear la expresión simbólica **2*b:**

```
var('b')
type(2*b)
```

```
        <class 'sage.calculus.calculus.SymbolicArithmetic'>
```

Como se puede ver en este ejemplo, la expresión simbólica **2*b** fue colocada en un objeto del tipo **SymbolicArithmetic**. La expresión también puede ser asignada a una variable:

```
m = 2*b
type(m)
|
        <class 'sage.calculus.calculus.SymbolicArithmetic'>
```

El siguiente programa crea 2 expresiones simbólicas, las asigna a variables y posteriormente realiza operaciones en ellas:

```
m = 2*b
n = 3*b
m+n, m-n, m*n, m/n
|
        (5*b, -b, 6*b^2, 2/3)
```

Aquí está otro ejemplo que multiplica 2 expresiones simbólicas entre ellas:

```
m = 5 + b
n = 8 + b
y = m*n
y
|
        (b + 5)*(b + 8)
```

5.9 Expandiendo y factorizando

Si es necesitada la forma expandida de la expresión de la sección anterior, se obtiene fácilmente llamando el método **expand()** (este ejemplo asume que las celdas en la sección anterior fueron ejecutadas):

```
z = y.expand()
z
|
        b^2 + 13*b + 40
```

5.10 Ejemplos variados de expresiones simbólicas.

```
var('a,b,c')
(5*a + b + 4*c) + (2*a + 3*b + c)
|
        5*c + 4*b + 7*a

(a + b) - (x + 2*b)
|
        -x - b + a

3*a^2 - a*(a -5)
|
        3*a^2 - (a - 5)*a

_.factor()
|
        a*(2*a + 5)
```

5.11 Pasando valores a las expresiones simbólicas

Si se le dan valores a las expresiones simbólicas, estas serán evaluadas y se dará un resultado. Si la expresión solo tiene una variable, entonces el valor puede simplemente pasado de la siguiente manera:

```
a = x^2
a(5)
|
        25
```

Sin embargo, si la expresión tiene 2 o más variables, cada variable necesita tener un valor asignado a ella por nombre:

```
var('y')
a = x^2 + y
a(x=2, y=3)
|
        7
```

5.12 Ecuaciones simbólicas y la función solve()

Además de trabajar con expresiones simbólicas, SAGE también es capaz de trabajar con **ecuaciones simbólicas**:

```
var('a')
type(x^2 == 16*a^2)
```

```
<class 'sage.calculus.equations.SymbolicEquation'>
```

Como se puede ver en el ejemplo, la ecuación simbólica **x^2 = = 16*a^2** fue colocada en un objeto del tipo **SymbolicEquation**. Una ecuación simbólica necesita utilizar doble signo igual '= =' para que pueda ser asignada a una variable utilizando un solo signo igual '=' como esta:

```
m = x^2 == 16*a^2
m, type(m)
```

```
(x^2 == 16*a^2, <class 'sage.calculus.equations.SymbolicEquation'>)
```

Muchas ecuaciones simbólicas pueden ser resueltas algebraicamente utilizando la función **solve()**:

```
solve(m, a)
```

```
[a == -x/4, a == x/4]
```

El primer parámetro en la función solve() acepta una ecuación simbólica y el segundo parámetro acepta la variable simbólica a resolver.
La función **solve()** también puede resolver ecuaciones simultaneas:

```
var('i1,i2,i3,v0')
a = (i1 - i3)*2 + (i1 - i2)*5 + 10 - 25 == 0
b = (i2 - i3)*3 + i2*1 - 10 + (i2 - i1)*5 == 0
c = i3*14 + (i3 - i2)*3 + (i3 - i1)*2 - (-3*v0) == 0
d = v0 == (i2 - i3)*3
solve([a,b,c,d], i1,i2,i3,v0)
```

```
[[i1 == 4, i2 == 3, i3 == -1, v0 == 12]]
```

Nota: cuando se pasan mas de 1 ecuación a la función solve(), estas necesitan ser colocadas en una lista.

5.13 Funciones matemáticas simbólicas

SAGE tiene la habilidad de definir funciones utilizando la sintaxis matemática. El siguiente ejemplo muestra una función **f** siendo definida que utiliza **x** como variable:

```
f(x) = x^2
f, type(f)
```

```
(x |--> x^2,
<class'sage.calculus.calculus.CallableSymbolicExpression'>)
```

Los objetos de esta manera son del tipo CallableSymbolicExpression, lo que significa que pueden ser llamadas como se muestra en el siguiente ejemplo:

```
f(4), f(50), f(.2)
```

```
(16, 2500, 0.040000000000000010)
```

Este es un ejemplo que utiliza la expresión de arriba (CallableSymbolicExpresion) dentro de un lazo:

```
a = 0
while a <= 9:
f(a)
a = a + 1
```

```
0
1
4
9
16
25
36
49
64
81
```

El siguiente ejemplo logra el mismo trabajo que el ejemplo anterior, pero utiliza características de lenguaje mas avanzadas:

```
a = srange(10)
a
|
        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

for num in a:
f(num)
|
        0
        1
        4
        9
        16
        25
        36
        49
        64
        81
```

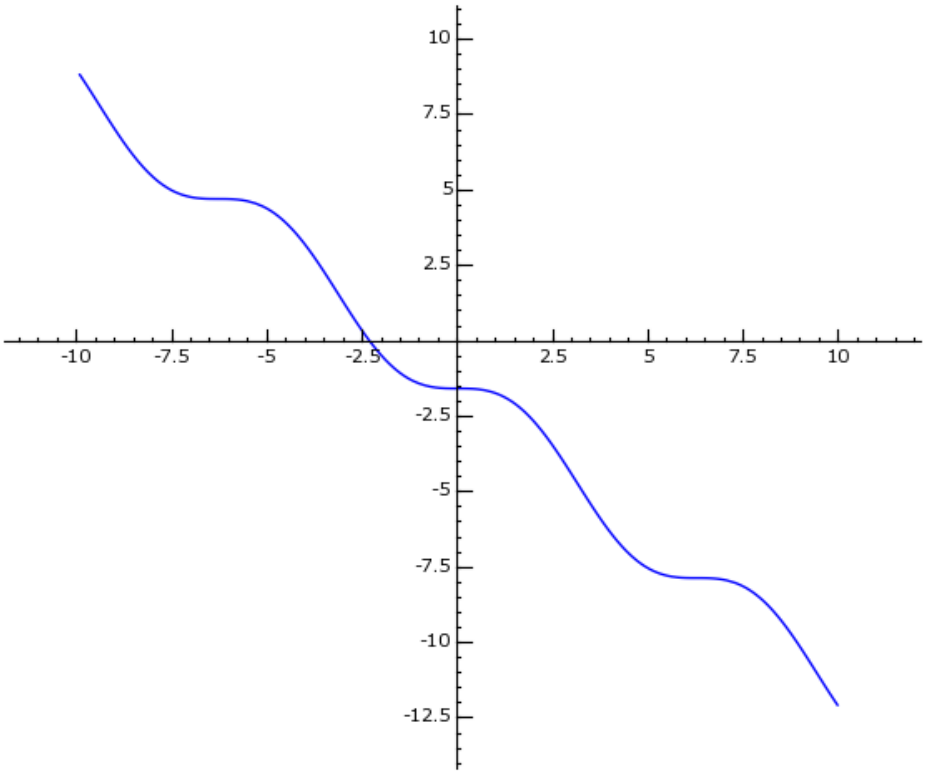
5.14 Encontrando raíces gráfica y numéricamente con el método find_root()

Algunas veces, las ecuaciones no pueden ser resueltas algebraicamente y la función solve() indica esto mostrando una copia de la ecuación que fue ingresada. Esto es mostrado en el siguiente ejemplo:

```
f(x) = sin(x) - x - pi/2
eqn = (f == 0)
solve(eqn, x)
|
        [x == (2*sin(x) - pi)/2]
```

Sin embargo, las ecuaciones que no pueden ser resueltas algebraicamente se pueden resolver tanto grafica como numéricamente. El siguiente ejemplo muestra la ecuación anterior siendo resuelta gráficamente:

```
show(plot(f,-10,10))
```



Esta gráfica indica que la raíz de esta ecuación es un poco mayor que -2.5.

El siguiente ejemplo muestra la ecuación siendo resuelta de forma más precisa con el método **find_root()**:

```
f.find_root(-10,10)
```

```
-2.309881460010057
```

El -10 y +10 que son pasados al método **find_root()** le dicen el intervalo en el cual debe buscar por las raíces.

5.15 Mostrando objetos matemáticos en la forma tradicional

Al principio se indico que SAGE es capaz de mostrar objetos matemáticos tanto en **forma tradicional** como **forma textual**. Hasta este punto, solo hemos estado usando forma textual, la cual esta por defecto. Si uno desea mostrar el objeto matemático en la forma tradicional, puede utilizarse la función **show()**. El siguiente ejemplo crea una expresión matemática y posteriormente las muestra en forma tradicional y textual.

```
var('y,b,c')
z = (3*y^(2*b))/(4*x^c)^2
#Mostrar la expresión en forma textual.
z
```

```
3*y^(2*b)/(16*x^(2*c))
```

```
#Mostrar la expression en forma tradicional.
show(z)
```

$$\frac{3 \cdot y^{2 \cdot b}}{16 \cdot x^{2 \cdot c}}$$

5.15.1 LaTeX es utilizado para mostrar objetos en la forma tradicional de matemáticas

LaTex (escrito LaTeX en texto plano, http://es.wikipedia.org/wiki/LaTeX) es un lenguaje de marcado para documentos que es capaz de trabajar con un amplio rango de símbolos matemáticos. Los objetos de SAGE proveerán descripciones en LaTeX de ellos mismos cuando los métodos latex() son llamados. La descripción LaTeX de un objeto también puede ser obtenida pasándola a la función **latex()**:

```
a = (2*x^2)/7
latex(a)
```

```
\frac{{2 \cdot {x}^{2} }}{7}
```

Cuando este resultado se le proporciona al software de despliegue LaTeX, generará salidas de formas matemáticas tradicionales similares a la siguiente:

$$\frac{2x^2}{7}$$

El paquete jsMath en el cual se hace referencia en el dibujo 2.5 es el software que SAGE Notebook utiliza para traducilr las entradas LaTeX a la forma de matemáticas tradicional como salida.

5.16 Grupos

El siguiente ejemplo muestra operaciones que SAGE puede desarrollar en grupos:

```
a = Set([0,1,2,3,4])
b = Set([5,6,7,8,9,0])
a,b
```

```
({0, 1, 2, 3, 4}, {0, 5, 6, 7, 8, 9})
```

```
a.cardinality()
```

```
5
```

```
3 in a
```

```
True
```

```
3 in b
```

```
False
```

```
a.union(b)
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
a.intersection(b)
```

```
{0}
```

6 Gráficas en 2D

6.1 Las funciones plot() y show()

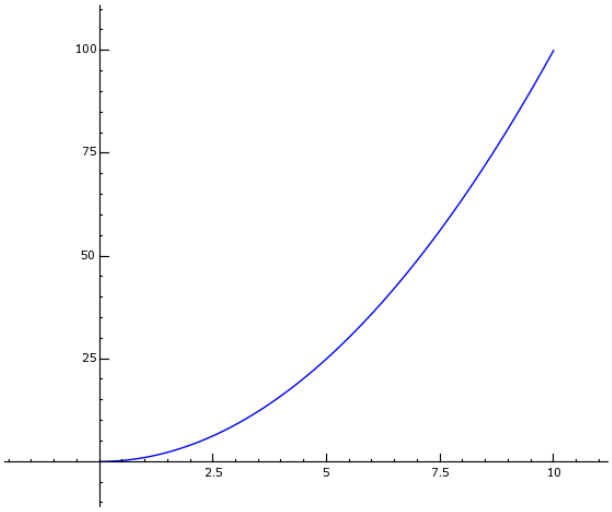
SAGE provee varias formas para generar gráficas en 2D de funciones matemáticas y una de estas maneras es utilizando la función **plot()** en conjunto con la función **plot()**. El siguiente ejemplo muestra una expresión simbólica siendo pasada a la función plot() como su primer parámetro. El segundo parámetro indica donde debe iniciar el trazo en el eje X y el tercer parámetro indica donde debe finalizar el trazo:

```
a = x^2
b = plot(a, 0, 10)
type(b)
```

```
<class 'sage.plot.plot.Graphics'>
```

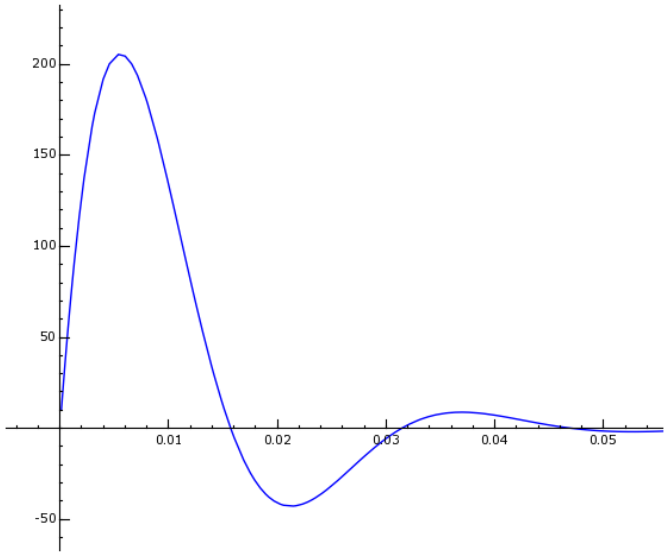
Note que la función **plot()** no muestra la gráfica. En lugar de esto, crea un objeto del tipo sage.plot.plot.Graphics y este objeto contiene la información de la gráfica. La función **show()** puede entonces ser utilizada para mostrar la gráfica:

```
show(b)
```

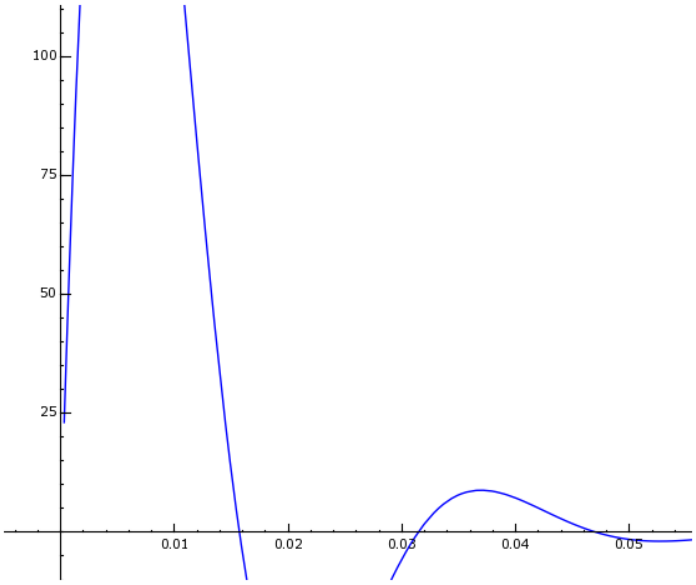


La función **show()** tiene 4 parámetros llamados **xmin**, **xmax**, **ymin** y **ymax** que pueden ser utilizados para ajustar que parte de la gráfica será mostrada. También tiene un parámetro **figsize** que determina que tan grande será la imagen. El siguiente ejemplo muestra el uso de **xmin** y **xmax** para mostrar la gráfica entre **0** y **.05** en el eje **x**. Note que la función **plot()** puede ser utilizada como el primer parámetro para la función **show()** para evitar escribir tanto (Nota: si otra variable simbólica en vez de x es utilizada, debe usarse la función var() primero para declararla):

```
v = 400*e^(-100*x)*sin(200*x)
show(plot(v,0,.1),xmin=0, xmax=.05, figsize=[3,3])
```



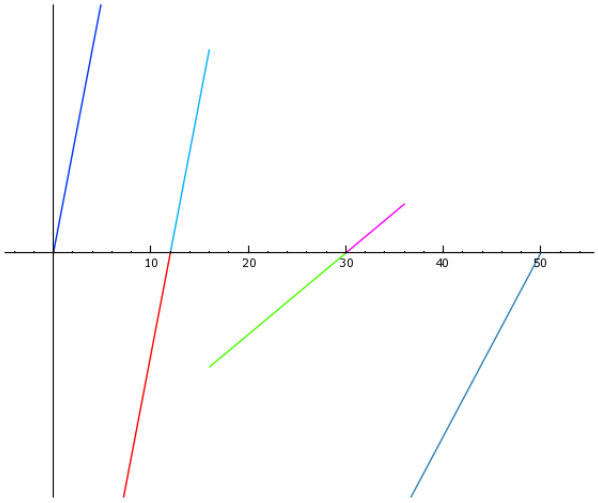
Los parámetros de **ymin** y **ymax** pueden ser utilizados para ajustar que tanto del eje y será mostrado en la gráfica:



6.1.1 Combinando gráficas y cambiando el color de la gráfica

Algunas veces es necesario combinar una o más gráficas en una sola. El siguiente ejemplo combina 6 gráficas utilizando la función **show()**:

```
p1 = t/4E5
p2 = (5*(t - 8)/2 - 10)/1000000
p3 = (t - 12)/400000
p4 = 0.0000004*(t - 30)
p5 = 0.0000004*(t - 30)
p6 = -0.0000006*(6 - 3*(t - 46)/2)

g1 = plot(p1,0,6,rgbcolor=(0,.2,1))
g2 = plot(p2,6,12,rgbcolor=(1,0,0))
g3 = plot(p3,12,16,rgbcolor=(0,.7,1))
g4 = plot(p4,16,30,rgbcolor=(.3,1,0))
g5 = plot(p5,30,36,rgbcolor=(1,0,1))
g6 = plot(p6,36,50,rgbcolor=(.2,.5,.7))

show(g1+g2+g3+g4+g5+g6,xmin=0, xmax=50, ymin=-.00001, ymax=.00001)
```

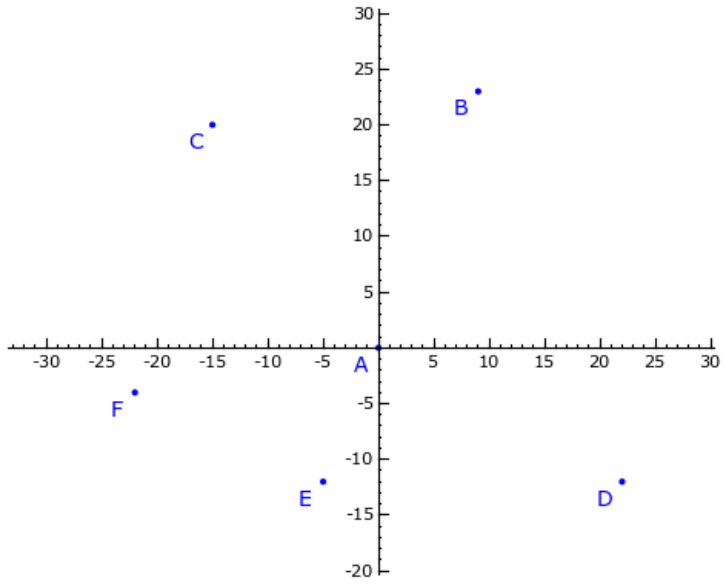


Noten que el color de cada línea puede ser cambiado utilizando el parámetro rgbcolor. RGB es por Red, Green, Blue (Rojo, Verde, Azul) y al parámetro **rgbcolor** se le asignan 3 valores entre el 0 y 1. El primer valor especifica que tanto rojo debe tener la gráfica (entre 0 y 100%), el segundo especifica el verde y el tercero indica el color azul.

6.1.2 Combinando graficas con un objeto de gráficas

Comúnmente es útil combinar varios tipos de gráficas en una imagen. En el siguiente ejemplo, 6 puntos son graficados junto con una etiqueta de texto para cada línea:

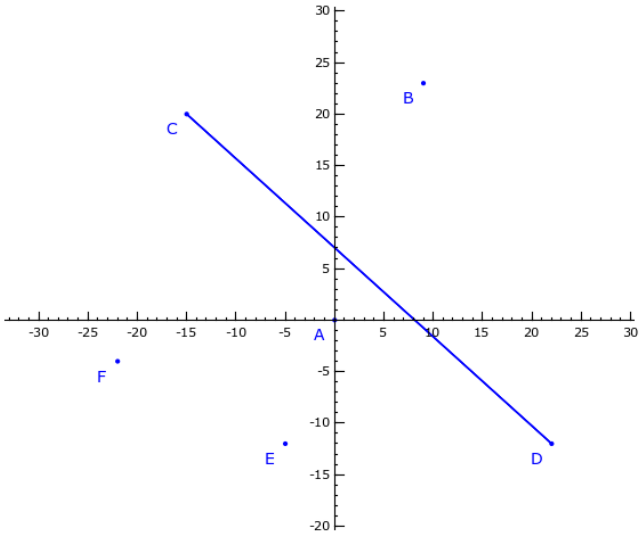
```
"""
Graficar los siguientes puntos en una gráfica:
A (0,0)
B (9,23)
C (-15,20)
D (22,-12)
E (-5,-12)
F (-22,-4)
"""
#Crear un objeto gráfico que será utilizado para contener varios
# objetos gráficos. Estos objetos gráficos serán mostrados en
# la misma imagen.
g = Graphics()
#Crear una lista de puntos y agregarlos al objeto gráfico.
points=[(0,0), (9,23), (-15,20), (22,-12), (-5,-12), (-22,-4)]
g += point(points)
#Agregar etiquetas a los puntos para el objeto gráfico.
for (pnt,letter) in zip(points,['A','B','C','D','E','F']):
        g += text(letter,(pnt[0]-1.5, pnt[1]-1.5))
#Mostrar los objetos gráficos combinados.
show(g,figsize=[5,4])
```



Primero se ejecuta un objeto gráfico vacio y una lista de puntos graficados es creada utilizando la función point(). Estos puntos son después agregados al objeto gráfico utilizando el operador +=. A continuación, se agrega una etiqueta para cada punto al objeto gráfico utilizando un lazo **for**. Finalmente, el objeto gráfico es mostrado en la hoja de trabajo utilizando la función show().

Incluso después de ser mostrado, el objeto gráfico todavía contiene todas las gráficas que han sido colocadas en el y más gráficas pueden ser agregadas a el según sea necesario. Por ejemplo, si se necesita dibujar una línea entre los puntos C y D, el siguiente código puede ser ejecutado en una celda separada:

```
g += line([(-15,20), (22,-12)])
show(g)
```

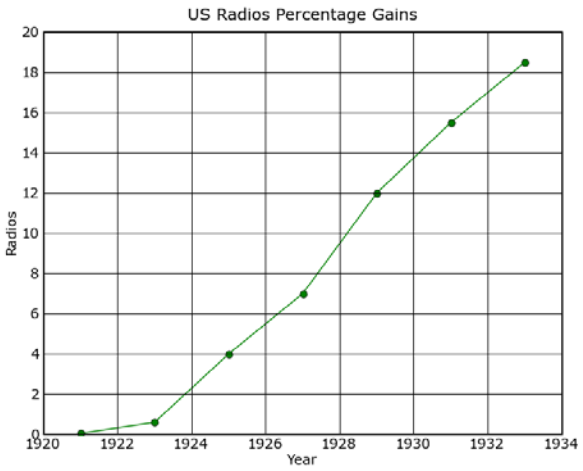


6.2 Graficas avanzadas con matplotlib

SAGE utiliza la librería **matplotlib** (http://matplotlib.sourceforge.net) para sus necesidades en gráficas y si una requiere más control al graficar que el que ofrece la función **plot(),** las capacidades de matplotlib pueden ser utilizadas directamente. Dado que una explicación completa de cómo funciona matplotlib esta fuera de la meta de este libro, esta sección provee ejemplos que ayudaran a iniciar en el uso de esta.

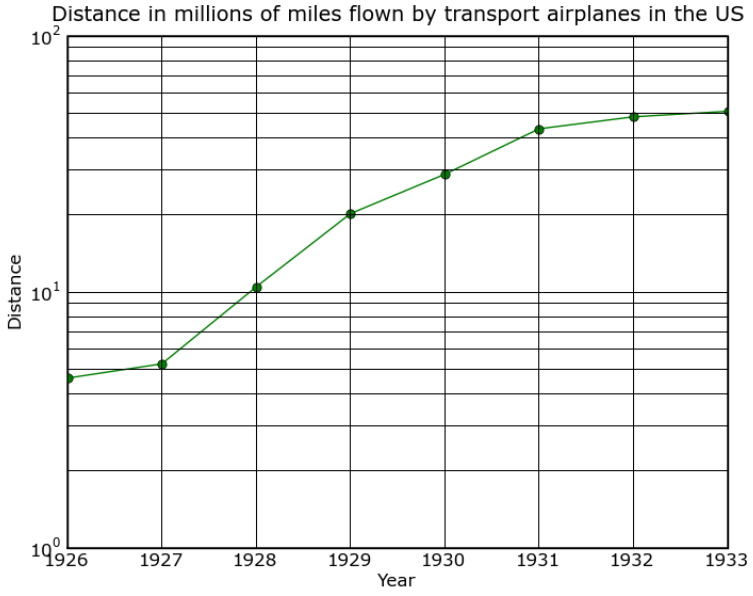
6.2.1 Graficando información de listas con líneas cuadriculadas y etiquetas de eje

```
x = [1921, 1923, 1925, 1927, 1929, 1931, 1933]
y = [ .05, .6, 4.0, 7.0, 12.0, 15.5, 18.5]

from matplotlib.backends.backend_agg import FigureCanvasAgg as \
FigureCanvas
from matplotlib.figure import Figure
from matplotlib.ticker import *
fig = Figure()
canvas = FigureCanvas(fig)
ax = fig.add_subplot(111)
ax.xaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.set_major_locator( MaxNLocator(10) )
ax.yaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.grid(True, linestyle='-', which='minor')
ax.grid(True, linestyle='-', linewidth=.5)
ax.set_title('US Radios Percentage Gains')
ax.set_xlabel('Year')
ax.set_ylabel('Radios')
ax.plot(x,y, 'go-', linewidth=1.0 )
canvas.print_figure('ex1_linear.png')
```



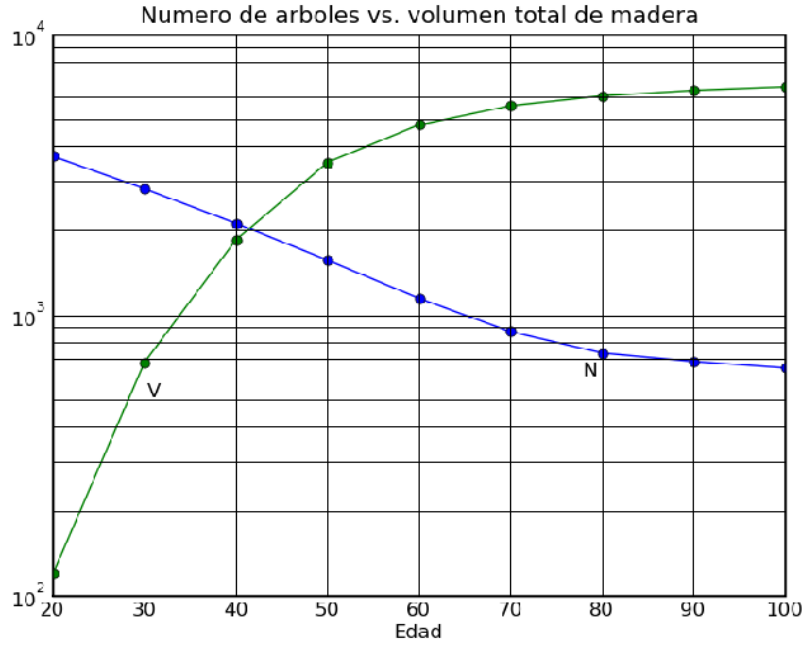
6.2.2 Graficando con un eje Y logarítmico

```
x = [1926, 1927, 1928, 1929, 1930, 1931, 1932, 1933]
y = [ 4.61,5.24, 10.47, 20.24, 28.83, 43.40, 48.34, 50.80]

from matplotlib.backends.backend_agg import FigureCanvasAgg as \
FigureCanvas
from matplotlib.figure import Figure
from matplotlib.ticker import *
fig = Figure()
canvas = FigureCanvas(fig)
ax = fig.add_subplot(111)
ax.xaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.set_major_locator( MaxNLocator(10) )
ax.yaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.grid(True, linestyle='-', which='minor')
ax.grid(True, linestyle='-', linewidth=.5)
ax.set_title('Distance in millions of miles flown by transport
airplanes in the US')
ax.set_xlabel('Year')
ax.set_ylabel('Distance')
ax.semilogy(x,y, 'go-', linewidth=1.0 )
canvas.print_figure('ex2_log.png')
```



6.2.3 Dos gráficas con etiquetas dentro de la gráfica

```
x = [20,30,40,50,60,70,80,90,100]
y = [3690,2830,2130,1575,1150,875,735,686,650]
z = [120,680,1860,3510,4780,5590,6060,6340,6520]

from matplotlib.backends.backend_agg import FigureCanvasAgg as \
FigureCanvas
from matplotlib.figure import Figure
from matplotlib.ticker import *
from matplotlib.dates import *
fig = Figure()
canvas = FigureCanvas(fig)
ax = fig.add_subplot(111)
ax.xaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.set_major_locator( MaxNLocator(10) )
ax.yaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.grid(True, linestyle='-', which='minor')
ax.grid(True, linestyle='-', linewidth=.5)
ax.set_title('Numero de arboles vs. volumen total de madera')
ax.set_xlabel('Edad')
ax.set_ylabel('')
ax.semilogy(x,y, 'bo-', linewidth=1.0 )
ax.semilogy(x,z, 'go-', linewidth=1.0 )
ax.annotate('N', xy=(550, 248), xycoords='figure pixels')
ax.annotate('V', xy=(180, 230), xycoords='figure pixels')
canvas.print_figure('ex5_log.png')
```



7 Ejemplos prácticos

7.1 Expresando una fracción a su mínima expresión

"""
Problema:
Reducir 90/105 a su mínima expresión.

Solución:
Una forma de resolver este problema es factorizando tanto el numerador como el denominador en factores primos, encontrar los factores comunes y dividir al numerador y denominador por estos factores.

"""

```
n = 90
d = 105
print n,n.factor()
print d,d.factor()
```

```
        Numerator: 2 * 3^2 * 5
        Denominator: 3 * 5 * 7
"""
```

Se puede ver que los factores 3 y 5 aparecen tanto en el numerador como el denominador, así que dividimos a ambos (numerador y denominador) entre 3*5:

"""

```
n2 = n/(3*5)
d2 = d/(3*5)
print "Numerador2:",n2
print "Denominador2:",d2
```

```
        Numerador2: 6
        Denominador2: 7
"""
```

Entonces, 6/7 es 90/105 reducido a su mínima expresión.

Este problema también pudo haberse resuelto de forma más directa simplemente introduciendo el valor 90/105 en una celda por que los objetos de números racionales son automáticamente reducidos a su mínima expresión:

"""

```
90/105
```

```
        6/7
```

7.2 Reduciendo una fracción simbólica a su mínima expresión

"""
Problema:
Expresar (6*x^2 - b) / (b - 6*a*b) en su mínima expresión, donde a y b
representan enteros positivos.
Solución:
"""

```
var('a,b')
n = 6*a^2 - a
d = b - 6 * a * b
print n
print " ---------"
print d
```

```
        6 a^2 - a
        ---------
        b - 6 a b
```

"""
Comenzamos por factorizar tanto el numerador como el denominador y después buscamos factores comunes:
"""

```
n2 = n.factor()
d2 = d.factor()
print "Numerador factorizado:",n2.__repr__()
print "Denominador factorizado:",d2.__repr__()
```

```
        Numerador factorizado: a*(6*a - 1)
        Denominador factorizado: -(6*a - 1)*b
```

"""
Al principio, no parece que el numerador y denominador algun factor común. Sin embargo, si se analiza con mas calma el denominador, se puede ver que si (1 - 6 a) es multiplicado por -1, (6 a - 1) es el resultado y este factor también esta presente en el numerador. Por lo tanto, nuestro siguiente paso es multiplicar al numerador y denominador por -1:
"""

```
n3 = n2 * -1
d3 = d2 * -1
print "Numerador * -1:",n3.__repr__()
print "Denominador * -1:",d3.__repr__()
```

```
        Numerator * -1: -a*(6*a - 1)
        Denominator * -1: (6*a - 1)*b
```

"""
Ahora, tanto el numerador como el denominador pueden ser divididos entre (6*a - 1) para reducir a ambos a su mínima expresión:
"""

```
factor_común = 6*a - 1
n4 = n3 / factor_común
d4 = d3 / factor_común
print n4
print " ---"
print d4
```

```
        - a
        ---
         b
```

"""

El problema también puede resolverse de forma mas directa utilizando un objeto de Aritmética Simbólica:

"""

```
z = n/d
z.simplify_rational()
```

```
        -a/b
```

7.3 Determinar el producto de dos fracciones simbólicas

Realizar la siguiente operación:

$$(\frac{x}{2y})^2 .(\frac{4y^2}{3x})^3$$

"""

Dado que las expresiones simbólicas son simplificadas automáticamente, todo lo que se tiene que hacer con este problema es introducir la expresión y asignársela a una variable:

"""

```
var('y')
a = (x/(2*y))^2 * ((4*y^2)/(3*x))^3
#Mostrar la expression en forma de texto:
a
```

```
    16*y^4/(27*x)
```

```
#Mostrar la expresión en forma tradicional:
show(a)
```

$$\frac{16 \cdot y^4}{27 \cdot x}$$

7.4 Resolver una ecuación lineal para x

Resolver $3x + 2x – 8 = 5x − 3x + 7$

"""

Como los términos serán automáticamente combinados cuando esta ecuación sea colocada en un objeto del tipo SymbolicEquation:

"""

```
a = 5*x + 2*x - 8 == 5*x - 3*x + 7
a
```

```
        7*x - 8 == 2*x + 7
```

"""

Primero, movamos los términos con x al lado izquierdo de la ecuación restándole 2x a cada lado. (Nota: Recuerda que el guión bajo '_' guarda el resultado de la última celda ejecutada:

"""

```
_- 2*x
```

```
        5*x - 8 == 7
```

"""

Ahora le agregamos 8 a ambos lados:

"""

```
_+8
```

```
        5*x == 15
```

"""

Finalmente, dividimos ambos lados entre 5 para obtener la solución:

"""

```
_/5
```

```
        x == 3
```

"""

Este problema también puede resolverse automáticamente utilizando la funciónn solve():

"""

```
solve(a,x)
```

```
        [x == 3]
```

7.5 Resolver una ecuación lineal que tiene fracciones

Resolver

$$\frac{16x-13}{6}=\frac{3x+5}{2}-\frac{4-x}{3}$$

"""
El primer paso es colocar la ecuación en un objeto del tipo SymbolicEquation. Es buena idea mostrar la ecuación posteriormente para verificar que haya sido ingresada correctamente.
"""

```
a = (16*x - 13)/6 == (3*x + 5)/2 - (4 - x)/3
a
```

```
(16*x - 13)/6 == (3*x + 5)/2 - (4 - x)/3
```

"""
En este caso, es dificil ver si la ecuación fue ingresada correctamente cuando se muestra en forma textual, asi que hay que mostrarla también en forma tradicional:
"""

```
show(a)
```

$$\frac{16\cdot x-13}{6} = \frac{3\cdot x+5}{2} - \frac{4-x}{3}$$

"""
El siguiente paso es determinar el mínimo común divisor (MCD) de las fracciones en esta ecuación para que las fracciones puedan ser removidas:
"""

```
lcm([6,2,3])
```

```
6
```

"""
El MCD de esta ecuación es 6 así que multiplicándola por 6 remueve las fracciones:
"""

```
b = a*6
b
```

```
16*x - 13 == 6*((3*x + 5)/2 - (4 - x)/3)
```

"""
El lado derecho de esta ecuación sigue en forma factorizado, así que es necesario expandirlo:
"""

```
c = b.expand()
c
```

```
16*x - 13 == 11*x + 7
```

"""

Transponemos el 11x hacia el lado izquierdo del signo igual restando 11x de la ecuación simbólica:

```
"""
d = c - 11*x
d
|
        5*x - 13 == 7
"""
```

Transponemos el -13 hacia el lado derecho del signo igual sumándole 13 a la ecuación simbólica:

```
"""
e = d + 13
e
|
        5*x == 20
"""
```

Finalmente, dividiendo la ecuación simbólica entre 5 dejará a la x sola en el lado izquierdo del signo igual dándonos la solución:

```
"""
f = e / 5
f
|
        x == 4
"""
```

Este problema también se puede resolver de la manera fácil utilizando la función solve():

```
"""
solve(a,x)
|
        [x == 4]
```

7.6 Uso de matrices

SAGE provee comandos estándar de algebra lineal, como lo son polinomio característico, forma de escalón, rastro, descomposición, etc. de una matriz.

Creación y multiplicación de matrices

```
A = Matrix([[1,2,3],[3,2,1],[1,1,1]])
w = vector([1,1,-4])
print A
|
        [1 2 3]
        [3 2 1]
        [1 1 1]
```

```
B = transpose(A)
print B
```

```
[1 3 1]
[2 2 1]
[3 1 1]
```

```
w*A
```

```
(0, 0, 0)
```

```
A*w
```

```
(-9, 1, -2)
```

7.7 Derivadas, integrales, fracciones parciales y transformada de Laplace

Para calcular la derivada de

$$\frac{d^4 \sin(x^2)}{dx^4}:$$

```
diff(sin(x^2), x, 4)
```

```
16*x^4*sin(x^2) - 12*sin(x^2) - 48*x^2*cos(x^2)
```

Ahora calculamos la derivada de

$$\frac{\partial(x^2+17y^2)}{\partial x}, \frac{\partial(x^2+17y^2)}{\partial y}:$$

```
x, y = var('x,y')
f = x^2 + 17*y^2
f.diff(x)
```

```
2*x
```

```
sage: f.diff(y)
```

```
34*y
```

Si queremos calcular una integral como $\int x\sin(x^2)\,dx$ o $\int_0^1 \frac{x}{x^2+1}\,dx$:

```
integral(x*sin(x^2), x)
```

```
-cos(x^2)/2
```

```
integral(x/(x^2+1), x, 0, 1)
```

```
log(2)/2
```

$$\frac{1}{x^2-1}:$$

Para calcular la descomposición de la fracción parcial

```
f = 1/((1+x)*(x-1))
f.partial_fraction(x)
```

```
        1/(2*(x - 1)) - 1/(2*(x + 1))
```

```
print f.partial_fraction(x)
```

```
          1            1
        ---------  -   ---------
        2 (x - 1)   2 (x + 1)
```

Finalmente, si queremos calcular la transformada de Laplace de $t^2e^t - \sin(t)$:

```
s = var("s")
t = var("t")
f = t^2*exp(t) - sin(t)
f.laplace(t,s)
```

```
        2/(s - 1)^3 - 1/(s^2 + 1)
```

7.8 Sistemas de ecuaciones no lineales

Los siguientes ejemplos fueron proporcionados por Jason Grout: Para resolver un sistema de ecuaciones simbólicamente uno puede utilizar:

```
var('x y p q')
```

```
        (x, y, p, q)
```

```
eq1 = p+q==9
eq2 = q*y+p*x==-6
eq3 = q*y^2+p*x^2==24
solve([eq1,eq2,eq3,p==1],p,q,x,y)
```

```
        [[p == 1, q == 8, x == (-4*sqrt(10) - 2)/3, y == (sqrt(2)*sqrt(5) - 4)/6],
        [p == 1, q == 8, x == (4*sqrt(10) - 2)/3, y == (-sqrt(2)*sqrt(5) - 4)/6]]
```

Si queremos ver la solución sin ecuaciones podemos intentar:

```
var('x y p q')
```

```
        (x, y, p, q)
```

```
eq1 = p+q==9
eq2 = q*y+p*x==-6
eq3 = q*y^2+p*x^2==24
solns = solve([eq1,eq2,eq3,p==1],p,q,x,y, solution_dict=True)
[[soln[p].n(), soln[q].n(),soln[x].n(), soln[y].n()] for soln in solns]
```

[[1.00000000000000, 8.00000000000000, -4.88303688022451, -0.139620389971937],
[1.00000000000000, 8.00000000000000, 3.54970354689117, -1.19371294336140]]