RHG DOR

Laboratorio

Madrid 2017

Índice general

Prólogo

La idea detrás de la existencia de un curso como este es fácil de expresar: *es muy conveniente que alumnos de un Grado en Ciencias Matemáticas tengan la oportunidad, desde el comienzo de sus estudios, de conocer y practicar una aproximación experimental al estudio de las matemáticas.*

La idea de las matemáticas como una ciencia experimental aparece después de que se generalizara el uso de los ordenadores personales, en nuestro país a partir de los años 90, y, sobre todo, con la aparición de ordenadores personales con suficiente potencia de cálculo ya en este siglo.

Sin embargo, algunos matemáticos de siglos anteriores demostraron una extraordinaria capacidad de cálculo mental y realizaron sin duda experimentos que les convencieron de que las ideas que pudieran tener sobre un determinado problema eran correctas, o bien de que eran incorrectas. Un ejemplo claro es el de Gauss en sus trabajos sobre teoría de números: se sabe, ya que se conservan los cuadernos que utilizaba, que fundamentaba sus afirmaciones en cálculos muy extensos de ejemplos concretos.

Los ordenadores personales extienden nuestra capacidad de cálculo, en estos tiempos bastante limitada, y permiten realizar búsquedas exhaustivas de ejemplos o contraejemplos. Podemos decir, hablando en general, que los ordenadores realizan para nosotros, de forma muy eficiente, tareas repetitivas como la ejecución, miles de veces, de un bloque de instrucciones que por algún motivo nos interesa.

Es cierto que el ordenador no va a demostrar un teorema por nosotros, pero puede encontrar un contraejemplo, probando así que el resultado propuesto es falso, y también puede, en ocasiones, ayudarnos a lo largo del desarrollo de una demostración. Sobre todo nos convencen de que una cierta afirmación es plausible y, por tanto, puede merecer la pena pensar sobre ella.

Descripción del curso

En este curso usamos Sage como instrumento para calcular y programar. Puedes ver una descripción de la forma de acceder a su uso en el primer capítulo de estas notas.

Comenzamos estudiando el uso de Sage como calculadora avanzada, es decir, su uso para obtener respuesta inmediata a nuestras preguntas mediante programas ya incluidos dentro del sistema. Por ejemplo, mediante `factor(n)` podemos obtener la descomposición como producto de factores primos de un entero `n`.

Sage tiene miles de tales instrucciones preprogramadas que resuelven muchos de los problemas con los que nos podemos encontrar. En particular, revisaremos las instrucciones para resolver problemas de cálculo y álgebra lineal, así como las instrucciones para obtener representaciones gráficas en 2 y 3 dimensiones.

A continuación, después describir las estructuras de datos y su manipulación, de trataremos los aspectos básicos de la programación usando el lenguaje Python, que es el lenguaje utilizado en una gran parte de Sage, y es el que usaremos a lo largo del curso. Concretamente, veremos los bucles `for` y `while`, el control del flujo usando `if` y `else`, y la recursión. Con estos pocos mimbres haremos todos los cestos que podamos durante este curso.

Por último, el curso contiene cuatro bloques, aproximación, aritmética, criptografía y teoría de la probabilidad, que desarrollamos todos los cursos junto con uno más, que llamamos miscelánea, cuyo contenido puede variar de unos cursos a otros. En esta parte usamos los rudimentos de programación que hemos visto antes para resolver problemas concretos dentro de estas áreas.

Aunque puede parecer que no hay conexión entre estos bloques, veremos que la hay bastante fuerte:

1. La aritmética, el estudio de los números enteros, es la base de muchos de los sistemas criptográficos que usamos para transmitir información de forma segura. En particular, estudiaremos el sistema RSA, uno de los más utilizados actualmente, cuya seguridad se basa en la enorme dificultad de factorizar un entero muy grande cuyos únicos factores son dos números primos enormes y distantes entre sí.

2. Para elegir los factores primos en el sistema RSA, cada usuario del sistema debe tener su par de primos, se utilizan generadores de números (pseudo-)aleatorios. Este será uno de los asuntos que trataremos en el bloque de probabilidad.

3. En el capítulo de ampliación de teoría de números discutiremos cómo encontrar números primos muy grandes y cómo intentar factorizar, de manera eficiente, números grandes. Son dos asuntos muy relacionados con la criptografía.

4. En el bloque sobre la aproximación de números reales, dedicaremos cierta atención al cálculo de los dígitos de algunas constantes matemáticas, en particular π y e. También estudiaremos la resolución aproximada de ecuaciones y la aproximación de funciones.

5. Como aplicación de este bloque sobre aproximación de números reales veremos un par de ejemplos en que usaremos logaritmos para estudiar potencias a^n con n muy grande. Es decir, usaremos los reales (*el continuo*) para estudiar un problema discreto.

De la lista anterior se deduce que el tema central del curso es la criptografía, con varios de los otros temas ayudando a entender y aplicar correctamente los sistemas criptográficos.

Los temas que tratamos en la segunda parte del curso vuelven a aparecer en asignaturas de la carrera, en algún caso optativas como la criptografía, y pueden verse como pequeñas introducciones *experimentales* a ellos. Creemos entonces que la asignatura es una buena muestra, por supuesto incompleta, de lo que os encontraréis durante los próximos aõs. Es como catar *el melón* antes de abrirlo.

El prototipo de este curso fué desarrollado, en su totalidad, por Pablo Angulo, y puedes todavía consultar el excelente resultado de su trabajo en este enlace. Los que lo hemos impartido después aprendimos casi todo lo que sabemos en sus notas y, por supuesto, se lo agradecemos aquí.

Parte I

Fundamentos

Capítulo 1

Introducción

Sage es un sistema de álgebra computacional (CAS, del inglés *computer algebra system*). El programa es libre, lo que nos permite copiarlo, modificarlo y redistribuirlo libremente. Sage consta de un buen número de librerías para ejecutar cálculos matemáticos y para generar gráficas. Para llamar a estas librerías se usa el lenguaje de programación `Python`.

Sage está desarrollado por el proyecto de software libre Sagemath. Se encuentra disponible para GNU/Linux y MacOS, y para Windows bajo máquina virtual (*virtualbox*)[1]. Reúne y compatibiliza bajo una única interfaz y un único entorno, distintos sistemas algebraicos de software libre.

Python es un lenguaje de propósito general de muy alto nivel, que permite representar conceptos abstractos de forma natural y, en general, hacer más con menos código. Buena parte de las librerías que componen Sage se pueden usar directamente desde Python, sin necesidad de acarrear todo el entorno de Sage.

Existen varias formas de interactuar con Sage: desde la consola, desde ciertos programas como TeXmacs o Cantor, y desde el *navegador de internet*. Para este último uso, Sage crea un *servidor web* que escucha las peticiones del cliente (un navegador), realiza los cálculos que le pide el cliente, y le devuelve los resultados. En esta asignatura sólo usaremos el interfaz web (*notebook*).

1.1. Iniciar sesión

1. En una máquina local (no se necesita conexión a internet)

   a) de nuestro laboratorio: abrir una terminal (la encontramos en el menú `Aplicaciones/Herramientas del Sistema/Terminal de MATE`) y ejecutar (escribir el texto indicado y pulsar `Intro`)

   arrancar−jupyter.sh

   Lo único que hace este `script` es cambiar de directorio a

   `/Desktop/SAGE-noteb/IPYNB/`

   y ejecutar `sage -notebook=jupyter --ip=127.0.0.1 --port=8888` que arranca Sage con la interfaz de `Jupyter`.

   Si en la terminal ejecutamos `sage -notebook=sagenb` accederíamos al `notebook` antiguo de Sage. En este curso usaremos siempre el nuevo, es decir `Jupyter`.

   b) exterior a nuestro laboratorio: generalmente se ha de instalar previamente el software. En el sitio de Sagemath se encontrarán las

[1] Recientemente se ha presentado una versión de Sage para Windows que no requiere la máquina virtual, es decir, puede instalarse directamente. Pueden verse los detalles en esta página.

instrucciones precisas para cada sistema operativo[2]. Una vez instalado, el inicio de sesión será similar al indicado en el punto anterior.

En este caso el programa se ejecuta en la máquina en la que estamos sentados (*máquina local=localhost*) y el navegador web se conecta a la máquina local mediante la dirección

`http://localhost:8888/`.

El número del puerto, por defecto `8888`, se puede utilizar si no hay otro proceso en la máquina que lo haya reservado. Si el puerto está ocupado `Jupyter` utiliza otro y los enlaces a hojas (Apéndice A-4*c*) de Sage no funcionarán. Si se trata, como ocurrirá casi siempre, de que ya hemos arrancado `Jupyter` en nuestra sesión, podemos ejecutar en una terminal `killall python` para poder arrancar `Jupyter` en el puerto `8888`.

2. En el servidor del Departamento de Matemáticas (se necesita conexión a internet). Al teclear, en un navegador, la dirección

   `https://jupyter.mat.uam.es/`,

   se nos muestra la siguiente ventana

El usuario es del tipo `nombre-apellido-jup`, nombre y apellido como en vuestra dirección de correo de la UAM, y contraseña que debe ser la misma que se os da para las cuentas en el Laboratorio.

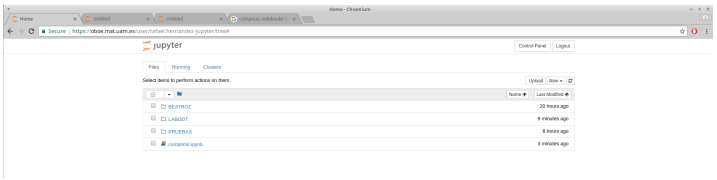
Cuando conectamos el navegador a una dirección remota, es decir, no a `localhost`, el código de Sage se ejecuta en la máquina remota y la máquina en la que estemos trabajando únicamente ejecutará el navegador. Si el servidor tiene que atender peticiones de cálculos complicados de muchos usuarios simultáneamente puede ralentizarse su funcionamiento, o incluso puede colgarse. En consecuencia, para trabajar durante las clases debería usarse la máquina local, como explicado en 1.1-1*a*, en lugar del servidor del Departamento.

1.2. El *Notebook* de Jupyter

Una vez que conectamos, a través del navegador, con una sesión de `Jupyter`, accedemos a una página en la que podemos ver el que va a ser nuestro *espacio de trabajo* básico.

[2] En este momento todavía no aparecen las instrucciones para la instalación en Windows sin máquina virtual mencionada en la nota anterior.

1.2.1. Sistema de archivos



1. Si estamos trabajando en el Laboratorio, y por tanto hemos ejecutado `arrancar-jupyter.sh` en una terminal para iniciar la sesión, las carpetas y archivos que vemos son los que hay en el directorio ∼/Desktop/SAGE-noteb/ (el símbolo ∼ representa el directorio `HOME` del usuario), que es el que contiene los archivos que os damos para el curso (ver Apéndice A-1).

   La terminal desde la que arrancamos queda abierta y en ella van apareciendo mensajes que se refieren a la ejecución de `Jupyter`. Esta terminal se puede mandar al panel, pinchando en el pequeño guión que hay en la parte de arriba a la derecha de la ventana, pero debe permanecer viva.

   Al terminar de trabajar, es IMPORTANTE cerrar bien `jupyter` pulsando `control-c` con la ventana de la terminal activa. Es decir, la buscamos en el panel, pinchamos en ella para activarla y entonces pulsamos las teclas `control` y `c` al mismo tiempo dos veces seguidas.

2. Si, en cambio, hemos iniciado sesión en el servidor del Departamento, las carpetas y archivos que vemos son las que teníamos en el servidor, tal como quedaron en la sesión anterior.

   Cuando terminamos de trabajar, pinchamos en la pestaña `Running` para ver las hojas que todavía están activas, aparecen en color verde, y las paramos todas marcándolas en la casilla de la izquierda y pulsando `Shutdown`. Finalmente, pinchamos en la pestaña `Files` y en la línea superior de la ventana pinchamos en `Logout`.

   Aquí también es IMPORTANTE cerrar, como se acaba de indicar, los procesos en el servidor que nos han permitido el acceso.

El resto de las indicaciones se aplican a los dos casos: Laboratorio y servidor del Departamento.

1. Cuando pinchamos en el cuadrado a la izquierda de un nombre de archivo se activan cuatro nuevas posibilidades: *Duplicate, Rename, Move, Download* que no necesitan grandes explicaciones. *Move* permite mover el archivo marcado a cualquiera de las carpetas que existen en nuestro directorio.

   En el menú *New* se puede crear una carpeta nueva dentro de la que está activa en ese momento, y podemos navegar en el sistema de archivos pinchando en los nombres de las carpetas.

2. Al seleccionar un archivo o carpeta también aparece un icono de papelera de color rojo. Permite destruirlos y, aunque pide confirmación, es BASTANTE PELIGROSO ya que puede haber archivos o carpetas seleccionados pero que no vemos en la ventana del navegador. Debemos usarlo poco y asegurarnos de que únicamente estamos destruyendo lo que realmente queremos.

   Una opción razonable es crear una carpeta de nombre BASURA en la página de inicio y mover los archivos a ella en lugar de destruirlos. Al cabo de un tiempo deberemos limpiar esta carpeta de archivos que realmente no son necesarios.

3. A la derecha de la ventana, encima de la lista de archivos y carpetas, también hay un botón *Upload* para subir archivos desde el sistema de archivos del ordenador en el que estamos trabajando al sistema de archivos del servidor del Departamento.

En el caso de que estemos trabajando en *local*, este botón NO ES NECESARIO ya que podemos ver, desde `Jupyter`, todos los archivos que hay por debajo de la carpeta en la que hemos arrancado (1.2.1-1).

4. Cuando marcamos dos o más archivos desaparece la posibilidad de descargarlos *clicando* una sóla vez. Hay que descargarlos uno a uno, o bien comprimirlos en un sólo archivo antes de descargarlo. De la misma forma, se puede comprimir una carpeta con archivos, que queremos subir al servidor, para poder subirlos todos de golpe. Se explica la forma más adelante (1.2.3).

5. Debe estar claro que conviene usar estas funciones de la interfaz de `Jupyter` para mantener nuestro ESPACIO DE TRABAJO ORDENADO. La forma concreta de hacerlo dependerá de cada persona.

1.2.2. Hojas de trabajo

1. Creamos una hoja de trabajo vacía en el menú *New* eligiendo el tipo de hoja en el deplegable. En el Laboratorio los únicos tipos disponibles serán `Python` y `Sagemath`, pero en el servidor del Departamento hay algunos más: `R` para Estadística, `Matlab`, `Julia` y `Octave` para Cálculo Numérico, `C++11` para programar, de forma interactiva, en `C++`.

2. Cada uno de estos tipos de hoja está asociado a un `núcleo` (`kernel`) que ejecuta los cálculos que le indicamos. En ocasiones hay que interrumpir la ejecución o reiniciar el `núcleo`. Cuando se reinicia se pierden todas las funciones y variables que teníamos.

   Todo esto se hace bajo el menú desplegable `Kernel`. La última entrada del menú sirve para cambiar de núcleo, y que usaremos (raramente) cuando una hoja de Sage sea entendida por el sistema como si fuera de Python.

3. Las celdas que vemos en la hoja son, por defecto, **celdas de cálculo**. Es decir, en ellas escribimos el código que queremos ejecutar. Encima de la zona de celdas hay un menú con pequeños iconos, y al pasar el cursor por encima de ellos se autoexplican. En particular, el triangulito con el segmento vertical en un vértice sirve para ejecutar la celda activa y el cuadrado para parar la ejecución. La celda activa tiene su borde de color verde.

   También es posible ejecutar la celda activa pulsando `control+Intro` (las dos teclas al mismo tiempo).

4. En ocasiones es necesario reiniciar totalmente una hoja que se ha colgado. Se puede utilizar el último, de izquierda a derecha, de los iconos mencionados en el punto anterior, que muestra una flecha circular. Al hacer esto se borran de la memoria todos los cálculos anteriores y hay que volver a ejecutar las celdas que nos convenga.

5. En el submenú `File/Download as` podemos elegir `PDF via Latex (.pdf)` para convertir la hoja a formato `PDF`, por ejemplo para imprimirla bien formateada. Las líneas de código que son demasiado largas y salen de la página, en el `PDF`, se pueden cortar usando el carácter `'\'` para terminarlas, es decir, varias líneas consecutivas terminadas cada una, menos la última, en `'\'` se interpretan como una única línea.

6. En las celdas de cálculo, en hojas `Sagemath`, hay un sistema para completar comandos pulsando el tabulador, y un sistema completo de ayuda que se obtiene completando el nombre de un comando con un interrogante de cierre (?) y ejecutando la celda.

7. Convertimos una **celda** de cálculo en una de **texto** en el submenú *Cell/Cell Type*, eligiendo como tipo *Markdown*.

En estas celdas se puede escribir código LaTeX(ver Capítulo **??**-Apéndice B) entre dólares, para las fórmulas matemáticas, y usar el lenguaje `markdown` para formatear el texto normal: secciones, subsecciones, listas, etc. Las celdas de texto también hay que ejecutarlas, en la misma forma que las de cálculo, para ver el resultado.

Puedes ver un resumen del uso de *Markdown* en esta hoja[3].

8. Es fácil convertir las hojas de trabajo antiguas, con extensión `.sws` a hojas nuevas (hojas de `Jupyter`) con extensión `.ipynb`. Las instrucciones y programas necesarios están en la carpeta `SAGE-noteb/bin/sws2ipynb/` en tu escritorio.

9. Hay otras opciones en la interfaz de `Jupyter` pero, aunque quizá útiles, no es muy probable que las usemos.

1.2.3. Interacción con el sistema operativo

1. Una hoja de trabajo de tipo `Python` permite interaccionar con el sistema operativo de la máquina. En el caso de que estemos trabajando en `local` no es muy útil porque podemos hacer lo mismo en una terminal, pero en el caso de que estemos trabajando en `remoto` sí lo es.

2. En una celda de cálculo, de una hoja de `Python`, se pueden escribir comandos del sistema operativo sin más que comenzar la línea con el símbolo de admiración (!). El comportamiento que se obtiene es el mismo que en una terminal, y depende de los permisos de ejecución que tenga el usuario.

   Puedes ver ejemplos de esta interacción, por jemplo comprimir y descomprimir archivos, en esta otra hoja.

3. Esta posibilidad es intrínsecamente peligrosa ya que cualquier pequeño error en la configuración del sistema podría permitir actividades no deseadas, como, por ejemplo, borrar el disco duro.

   En el caso en que encontréis un error de este tipo, en las máquinas del Laboratorio o en el servidor, rogamos que lo comuniquéis cuanto antes a la dirección de correo

   `informatica.matematicas@uam.es`.

1.3. Notación

En estas notas, representaremos los cuadros de código por una

caja con fondo de color .

En ocasiones aparecerán numeradas las líneas de código, en la parte exterior de la caja. Esta numeración no es parte del código y aparece para facilitar la referencia a líneas individuales. Además, las palabras clave del lenguaje de programación aparecerán resaltadas, para distinguirlas de las demás. Las respuestas del intérprete, en caso de querer mostrarlas, aparecerán indentadas, y en otro color, bajo las cajas con el código.

Así, por ejemplo, trascribiremos la celda

```
## Suma de los 150 primeros impares positivos
m=150
'Los %d primeros impares positivos suman %d.'%(m,sum([2*k+1 for k in range(m)]))
        'Los 150 primeros impares positivos suman 22500.'
```

```
'Los 150 primeros impares positivos suman 22500.'
```

con alguno de los siguientes aspectos

- numerado

[3] El enlace funciona si se ha arrancado `Jupyter` como se indica en 1.1-1*a*.

```
## Suma de los 150 primeros impares positivos
m=150
'Los %d primeros impares positivos suman %d.' %(m,sum([2*k+1 for k in range(m)]))
```

- sin numerar

```
## Suma de los 150 primeros impares positivos
m=150
'Los %d primeros impares positivos suman %d.' %(m,sum([2*k+1 for k in range(m)]))
```

- con la respuesta del intérprete

```
## Suma de los 150 primeros impares positivos
m=150
'Los %d primeros impares positivos suman %d.' %(m,sum([2*k+1 for k in range(m)]))
```

```
'Los 150 primeros impares positivos suman 22500.'
```

Cuando copiamos código desde este PDF a una celda de Sage podemos encontrarnos con errores de sintaxis debidos simplemente a que en la celda se entienden algunos caracteres, que estaban en el PDF, de forma incompatible con las reglas sintácticas de Sage. Un ejemplo típico aparece con el guión que usamos para la resta, que al pegarlo en la celda aparece como un guión largo, que Sage no interpreta como el símbolo de la resta.

1.4. En resumen

Cuando trabajamos en `local` los recursos de la máquina, `RAM, cores, red, etc.`, son nuestros, pero ese no es el caso al trabajar con el servidor de cálculo del Departamento, ya que estamos compartiendo los recursos con el resto de los usuarios.

1. El servidor dispone de 500 GB para almacenar los archivos de trabajo de los usuarios. Aunque no se ha fijado, de momento, un límite de espacio por usuario esperamos que ningún usuario supere unos 2 GB de espacio en disco.

   En el caso de las máquinas del Laboratorio también debe haber límites, ya que todos los archivos de todos los usuarios se almacenan en un único disco duro de 256 GB, lo que nos permite sentarnos en cualquier puesto y ver los archivos de nuestra cuenta. En este caso un límite razonable puede ser de 1 GB.

2. En el servidor hay un límite de 3 GB de `RAM` por usuario conectado. Es NECESARIO cerrar las hojas que no estemos usando, para trabajar basta con tener una hoja abierta y puntualmente abrir otra para copiar algún código, y no dejar procesos vivos en la máquina, como se explica en la página 5, cuando hemos dejado de trabajar.

3. En este momento todavía no sabemos cuántos usuarios simultáneos soporta, sin ralentizarse, el servidor. Entonces, no parece conveniente usar el servidor durante las clases ya que podría colapsarse y hacernos perder tiempo.

4. Aunque se hace un *backup* cada noche del servidor de cálculo y de las cuentas del Laboratorio, no deberíais fiaros completamente de esto. Es conveniente mantener un *backup* personal en un lápiz de memoria.

Apéndice A

Uso de este documento

Hemos preparado estas notas con la intención de que faciliten el mantenimiento organizado de toda la información que genera el curso. Pueden cambiar un poco a lo largo del curso, ya que corregimos las erratas, y errores, que se detecten, y añadiremos nuevas secciones o temas si nos parece útil. Por otra parte, también pretendemos que sigan creciendo en cursos sucesivos aunque entonces será probablemente imposible cubrir todo el material y habrá que seleccionar.

1. Encontrarás una carpeta `SAGE-noteb` en el escritorio de tu cuenta en el Laboratorio. Esta carpeta contendrá los materiales que os vayamos dando, y, por tanto, su contenido puede variar de una semana a otra. Se recomienda mantener una copia actualizada de esta carpeta en un *pendrive*.

2. En cada examen encontrarás una copia de la carpeta `SAGE-noteb`, tal como estaba en tu cuenta habitual justo antes del examen, en el escritorio de la cuenta en la que debes hacer el examen. Esto quiere decir que puedes colocar archivos que quieres ver durante el examen en ciertas subcarpetas, se concreta un poco más adelante, de la carpeta `SAGE-noteb`.

3. El documento básico es este, `laboratorio.pdf`, que tiene un montón de enlaces a otras páginas del documento, a páginas *web* y a otros documentos, lecturas opcionales, que están situados en la carpeta `PDFs` dentro de la que contiene todo el material. Se trata entonces de un documento *navegable*.

   Recomendamos abrirlo con el navegador *web*, en nuestro caso usamos `chromium-browser` por defecto. Se abre escribiendo en la barra de direcciones del navegador

   ```
   file:///alumnos/<curso>/<usuario>/Desktop/SAGE-noteb/laboratorio.pdf
   ```

   con `<curso>` el nombre que tiene en el Laboratorio (`labodt, labot,` etc.) y `<usuario>` el de vuestra cuenta en el Laboratorio.

4. Enlaces: navegamos en el documento y fuera de él usando diversos tipos de enlaces:

   a) Por ejemplo, este es un enlace a un documento PDF situado en uno de nuestros servidores y debe abrirse en el navegador que usamos por defecto, mientras que este otro es un enlace a una página web externa.

   b) También hay enlaces a otras zonas de este mismo documento. Debemos recordar, aproximadamente, la página del documento en la que estábamos para poder volver cómodamente a ella. Puede ayudar recordar la zona de la barra de la derecha de la ventana (barra

de *scroll*) en la que estaba la página de origen ya que *clickando* en esa zona volvemos a ella.

c) Por último, hay enlaces que nos llevan directamente a hojas de trabajo de Sage que, como hemos visto se abren dentro del navegador (en nuestro caso `Chromium`). Para que estos enlaces funcionen hay que arrancar Sage como se indica en 1.1-1*a*. Además conviene abrir esos enlaces en pestañas nuevas del navegador *clickando* en el enlace con el botón medio, es decir, con la rueda. Alternativamente, se puede *clickar* en el enlace con el botón derecho y seleccionar en el menú que aparece *"Open link in new tab"*.

5. Puedes añadir tus propias notas para completar o clarificar el contenido de nuestro documento. Es importante entonces tener en cuenta que puedes, y debes, *p*ersonalizar nuestro `laboratorio.pdf` con tus aportaciones o las de tus compañeros. Para esto

   a) Para cada capítulo, por ejemplo el 4, hay un documento, en la carpeta `SAGE-noteb/INPUTS/NOTAS`, con nombre `notas-cap4.tex` en el que puedes escribir y no desaparecerá cuando modifiquemos la carpeta. Si escribes en cualquiera de los otros documentos a la semana siguiente puede haber desaparecido lo que hayas añadido.

   b) En esos documentos `notas-capn.tex` se puede escribir texto simple, pero para obtener un mínimo de legibilidad hay que escribir en LaTeX, que no es sino texto formateado, como se explica en el apéndice siguiente.

   c) Para generar el PDF resultante, incluyendo las notas añadidas, hay que compilar los archivos tex (ver Apéndice B-refTS).

6. Las hojas de trabajo de Sage que hayas creado o modificado y quieras ver durante un examen debes guardarlas en la subcarpeta `IPYNB-mios` dentro de `SAGE-noteb/IPYNB`. Esta subcarpeta puede contener, a su vez, diversas subcarpetas que organicen su contenido.

7. Conviene mantener la información acerca de nuestras hojas de Sage, las que hayamos elaborado o modificado nosotros, de manera que sea fácilmente accesible durante los exámenes: por ejemplo, para una hoja que se refiere al Capítulo 4 de estas notas podrías incluir en el archivo `notas-cap4.tex` un `\item` indicando el nombre y localización del archivo, debería estar en la subcarpeta `IPYNB/IPYNB-mios` de la carpeta principal, y una descripción de su contenido. Esto es importante para facilitar la búsqueda de una hoja concreta sobre la que quizá trabajamos hace cuatro meses y de la que podemos haber olvidado casi todo.

8. En el apéndice **B** se describe la manera de generar enlaces de nuestro PDF, `laboratorio.pdf`, a páginas *web*, a otros documentos PDF o a hojas de trabajo de Sage.

Apéndice B

LaTeX básico

1. Aunque este pequeño resumen puede servir para escribir en LaTeX las notas que queráis añadir al texto, una introducción bastante completa y clara se puede encontrar en este enlace.

2. Como editor de LaTeX usamos el programa `texstudio` que está instalado en las máquinas del Laboratorio.

   Una vez que hemos abierto el programa, su lanzador está en el menú *Aplicaciones/Oficina/*, debemos abrir, usando el botón *Open*, los archivos con código LaTeX que vamos a editar.

3. Siempre hay que abrir, dentro de `texstudio`, el archivo

   `SAGE-noteb/laboratorio.tex`

   que es el documento raíz y el que hay que procesar para obtener el `PDF`. Se procesa pinchando en el botón `Build&view`, el botón con dos puntas de flecha verdes en la barra superior de `texstudio`. El `PDF` resultante aparece en el lado derecho de la ventana.

4. Los documentos `notas-capn.tex`, que deberían están en la subcarpeta `SAGE-noteb/INPUTS/NOTAS/` de la carpeta principal, inicialmente contienen únicamente

   ```
   \begin{enumerate}
    \item
   \end{enumerate}
   ```

   que es un entorno de listas numeradas. Debes abrirlos en el editor, usando el menú `Open`, para añadirles materia.

5. Las líneas, dentro de un documento de código LaTeX, que comienzan con un % son comentarios que no aparecen en el PDF resultante. Así, por ejemplo, para ver en el PDF uno de los archivos `notas-capn.tex` que has editado, por ejemplo el `notas-cap2.tex`, debes quitar el símbolo % al comienzo de dos líneas en `SAGE-noteb/laboratorio.tex` cuyo contenido es

   ```
   %\section{Notas personales}
   %\montan|notas-cap2|
   ```

6. Cada nota que quieras incluir debe comenzar con un nuevo `\item` y a continuación el texto que quieras.

7. Para escribir matemáticas dentro de una línea de texto basta escribir el código adecuado entre símbolos de dólar ($..$). Para escribir matemáticas en *display*, es decir ocupando las fórmulas toda la línea se puede encerrar el código entre dobles dólares ($$..$$), o, mucho mejor, abrir la zona de código con `\[` y cerrarla con `\]`.

8. Por ejemplo, podemos mostrar una ecuación cuadrática en *display* mediante

   ```
   \[ax^2+bx+c=0\]
   ```

   que produce

   $$ax^2 + bx + c = 0$$

   y su solución mediante

   ```
   \[ x=\frac{-b\pm \sqrt{b^2-4ac}}{2a}\]
   ```

   que ahora produce

   $$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

9. Si observas con cuidado el código LaTeX anterior verás que la forma en que se escribe el código coincide bastante con la forma en que leemos la expresión. Una diferencia es que ante ciertos operadores con dos argumentos, como la fracción que tiene numerador y denominador, debemos avisar a LaTeX de que debe esperar dos argumentos mientras que cuando leemos la fórmula hasta que no llegamos a `partido por 2a` no sabemos que se trata de una fracción.

   Esto es lo que hace que aprender a escribir código LaTeX sea muy sencillo para personas acostumbradas a leer texto matemático.

10. Para cambiar de párrafo en LaTeX basta dejar una línea completamente en blanco.

11. Los subíndices se consiguen con la barra baja, `x_n` da x_n, y los superíndices con el acento circunflejo, `x^n` da x^n.

12. Como se ve en el ejemplo anterior, `\frac{numerador}{denominador}` es la forma de obtener una fracción.

13. Conjuntos:

    a) $A\times B$ produce $A \times B$.
    b) $A\cap B$ produce $A \cap B$.
    c) $A\cup B$ produce $A \cup B$.
    d) $a\in B$ produce $a \in B$.
    e) $a\notin B$ produce $a \notin B$.
    f) $A\subset B$ produce $A \subset B$.
    g) $A\to B$ produce $A \to B$.
    h) $a\mapsto f(a)$ produce $a \mapsto f(a)$.
    i) $A=\{a,b,c\}$ produce $A = \{a, b, c\}$.

14. Cálculo:

    a) `\[\lim_{x\to \infty} f(x)=a\]` produce

    $$\lim_{x\to\infty} f(x) = a.$$

b) `\[\lim_{h\to 0} \frac{f(x+h)-f(x)}{h}=:f^{\prime}(x)\]` produce

$$\lim_{h\to 0} \frac{f(x+h)-f(x)}{h} =: f'(x).$$

c) `\[\sum_{i=0}^{i=\infty}\frac{x^n}{n!}=:e^x\]` produce

$$\sum_{i=0}^{i=\infty} \frac{x^n}{n!} =: e^x.$$

d) `\[\int_a^b f(x)dx\]` produce

$$\int_a^b f(x)dx.$$

15. También es conveniente saber componer matrices. Por ejemplo,

```
\begin{equation}
\begin{pmatrix}
    1&0&0\\
    0&1&0\\
    0&0&1
\end{pmatrix}
\end{equation}
```

produce la matriz identidad

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{B.1}$$

16. Puedes encontrar una lista más completa de los códigos que producen diversos símbolos matemáticos en este archivo, mientras que la lista completa, que es enorme, se encuentra en este otro.

17. Podemos cambiar el color de un trozo de texto en el PDF sin más que incluir el correspondiente texto, en el archivo con el código LaTeX, entre llaves indicando el color en la forma `{\color{green}...texto...}`, que veríamos como ...texto....

18. Para incluir en el PDF un enlace a otra zona del mismo documento

    a) En la zona a la que queremos que lleve el enlace debemos incluir una línea con el contenido `\label{nombre}`, donde `nombre` es el nombre arbitrario que damos al enlace y que no debe ser igual a ningún otro *label* en el documento.

    b) Donde queremos que aparezca el enlace usamos

    `\hyperref[nombre]{texto},`

    con `nombre` el del enlace de acuerdo al punto anterior, y `texto` el que queramos que aparezca como enlace, es decir coloreado, y que pinchamos para movernos al otro lugar en el documento.

    Si incluyes enlaces de estos en la copia de la carpeta `SAGE-noteb` en el ordenador del Laboratorio, y que lleven a zonas del PDF fuera de tus notas personales, *esos enlaces desaparecerán cuando actualicemos* la carpeta.

19. Para incluir en el PDF resultante un enlace a una página *web* basta escribir, en el lugar adecuado del texto, algo como

`\href{http://...URL...}{enlace}`,

donde `...URL...` es la dirección completa de la página y `enlace` es el texto que va a aparecer en el PDF como el enlace pinchable.

20. Para incluir en el PDF un enlace a otro PDF, por ejemplo situado en la subcarpeta `PDFs-mios` de la carpeta `SAGE-noteb`, basta escribir, en el lugar adecuado del texto, algo como

    `\href{run:PDFs-mios/<nombre del PDF>.pdf}{enlace}`.

    El contenido de esta carpeta `PDFs-mios` no desaparecerá al actualizar la carpeta `SAGE-noteb`.

21. En este archivo `laboratorio.pdf` hay también enlaces que llevan directamente a hojas de trabajo de Sage. En las secciones de notas personales puedes crear esa clase de enlaces incluyendo en el código LaTeX algo como

    ```
    \href{http://localhost:8888/notebooks/<camino>}{texto}
    ```

    con `<camino>` que debería ser

    ```
    ~/Desktop/SAGE-noteb/IPYNB/IPYNB-mios/<nombre del archivo>
    ```

    Estos enlaces funcionarán si hemos arrancado Sage como se indica en 1.1-1*a*.

Capítulo 2

SAGE como calculadora avanzada

Usamos Sage como una "calculadora avanzada" escribiendo en una celda de cálculo una instrucción preprogramada junto con un número correcto de parámetros.

Por ejemplo, existe una instrucción para factorizar números enteros, **factor**, que admite como parámetro un número entero, de forma que debemos evaluar una celda que contenga **factor**(2^137+1) para calcular los factores primos del número $2^{137}+1$.

Muchas instrucciones de SAGE tienen, además de parámetros obligatorios, algunos parámetros opcionales. Obtenemos la información sobre los parámetros de una función escribiendo, en una celda de cálculo, el nombre de la función seguido de un paréntesis abierto y evaluando, o, más cómodo, clicando la tecla de tabulación. Por ejemplo, tabulando tras escribir **factor**(obtenemos la información sobre la función que factoriza enteros.

Además, si sólo conocemos, o sospechamos, algunas letras del nombre de la función podemos escribirlas en la celda de cálculo y pulsar el tabulador, lo que produce un desplegable con los nombres de todas las instrucciones que comienzan por esas letras. En el desplegable podemos elegir la que nos parezca más próxima a lo que buscamos y vemos que se completan las letras que habíamos escrito con el nombre completo que hemos elegido. Si necesitamos información sobre los parámetros de la instrucción podemos añadir el interrogante al final y evaluar.

Una instrucción como **factor**(2^137+1) decimos que es una función, o que está en forma funcional, ya que se parece a una función matemática que se aplica a un número entero y produce sus factores. Hay otra clase de instrucciones a las que llamamos *métodos* y que tienen una sintaxis diferente

```
(objeto).metodo()
```

Por ejemplo, también podemos factorizar un entero mediante

(2^137+1).**factor**()

donde $2^{137}+1$ es un objeto de clase *número entero* al que aplicamos el método *factor*. Esta sintaxis procede de la programación *orientada a objetos* (OOP), y como Python es un lenguaje orientado a objetos es natural que en SAGE aparezca esta sintaxis.

Algunas instrucciones, como **factor**, admiten las dos formas, funciones y métodos, pero otras sólo tienen uno de las dos. Veremos ejemplos a lo largo del curso.

Como SAGE tiene miles de instrucciones preprogramadas, para toda clase de tareas en matemáticas, es muy útil tener a mano un *chuletario* con las

instrucciones más comunes[1]. Hay varios y aquí puedes encontrar enlaces a algunos:

1. En castellano.
2. Preparada por el desarrollador principal de SAGE William Stein (en inglés).
3. Cálculo.
4. Álgebra lineal.
5. Aritmética.
6. Estructuras algebraicas.
7. Grafos.

2.1. Aritmética elemental

Las operaciones de *la escuela* son sencillas de invocar. Sobre el resultado a esperar de cada operación (o sucesión de operaciones), ha de tenerse en cuenta que la aritmética es exacta, así:

- Suma (y diferencia): 3+56+23−75 devuelve 7.
- Producto: 3∗56∗23, 3864
- Potencia: 3∗∗2, 9; y también lo hace 3^2.[2]
- Cociente: 3864/56 es 23. Pero, 7/3 (o 14/6) devolverá, 7/3.
  También la respuesta a 7/5 se muestra en notación racional[3], 7/5, en detrimento de la notación decimal[4], 1.4.
- División entera: 7//2 devuelve 3, el cociente en la división entera[5]; y 7 %2, el resto, 1.

Al encontrar varias operaciones en una misma expresión, se siguen las usuales reglas de precedencia, rotas por la presencia de paréntesis.

- 8∗(5−3)+9^(1/2)+6 da 25, mientras que (8∗(5−3)+9)^(1/2)+6 es 11.

Estas operaciones aritméticas se efectúan en algún conjunto, generalmente un anillo o cuerpo, de números, y Sage dispone de unos cuantos predefinidos:

Símbolo	Descripción
ZZ	anillo de los números enteros, $\mathbb{Z}$
Integers(6)	anillo de enteros módulo 6, $\mathbb{Z}_6$
QQ	cuerpo de los números racionales, $\mathbb{Q}$
RR	cuerpo de los números reales (53 bits de precisión), $\mathbb{R}$
CC	cuerpo de los números complejos (53 bits de precisión), $\mathbb{C}$
RDF	cuerpo de números reales con doble precisión
CDF	cuerpo de números complejos con doble precisión
RealField(400)	reales con 400-bit de precisión
ComplexField(400)	complejos con 400-bit de precisión
ZZ[I]	anillo de enteros Gaussianos
AA, QQbar	cuerpo de números algebraicos, Q
FiniteField(7)	cuerpo finito de 7 elementos, $\mathbb{Z}_7$ o $\mathbb{F}_7$
SR	anillo de expresiones simbólicas

[1] Estos *chuletarios* pertenecen a versiones anteriores de Sage y no han sido actualizados. Aunque la sintaxis de las instrucciones de Sage no varía mucho, en ocasiones hay pequeños cambios en la versi'on actual respecto a la información que aparece en el chuletario.

[2] La segunda expresión, con ser más sencilla, obliga a hacer aparecer el símbolo ^ en escena, lo que, en la mayoría de teclados, requiere pulsar la barra espaciadora tras él.

[3] El cuerpo de los racionales, $\mathbb{Q}$, es suficiente para contener cocientes de números enteros.

[4] Se reserva la notación decimal para números que, con cierta precisión (finita), servirán para aproximar reales, complejos, ...

[5] $Dividendo = cociente \times divisor + resto$, con $0 < resto < divisor$.

2.2. Listas y tuplas

Las listas y tuplas son *estructuras de datos* y se tratarán más ampliamente en el capítulo siguiente. Para cubrir las necesidades de este capítulo baste decir que

1. Se crea una lista de nombre L escribiendo en una celda de cálculo algo como

   L=[1,7,2,5,27,−5]

2. Los elementos de la lista están ordenados y se accede a ellos mediante instrucciones como a=L[1], que asigna a la variable a el valor del segundo elemento de la lista, es decir a vale 7. Eso quiere decir que el primer elemento de la lista es L[0] y NO L[1] como podríamos esperar.

3. Las tuplas son muy parecidas a las listas, pero hay ciertas diferencias que veremos más adelante. Una tupla t se crea con la instrucción

   t=(1,7,2,5,27,−5)

   con paréntesis en lugar de corchetes. Se accede a sus elementos de la misma forma que para las listas, y, por ejemplo, t[1] vale 7. Mencionamos aquí las tuplas porque en Sage las coordenadas de un punto no forman una lista sino una tupla.

2.3. Funciones

Necesitamos definir funciones, en sentido matemático, para poder calcular con ellas, o bien representarlas gráficamente. Muchas funciones están ya definidas en Sage y lo único que haremos es asignarles un nombre cómodo para poder referirnos a ellas. Así tenemos la exponencial, el logaritmo y las funciones trigonométricas. También podemos construir nuevas funciones usando las conocidas y las operaciones aritméticas.

Podemos definir una función, por ejemplo de la variable x, mediante una expresión como `f(x)=sin(x)`, que asigna el nombre `f` a la función seno. A la derecha de la igualdad podemos escribir cualquier expresión definida usando las operaciones aritméticas y las funciones definidas en Sage. De éstas, las de uso más común son

Símbolo	Descripción
$f1(x) = exp(x)$	que define la función exponencial, e^x
$f2(x) = log(x)$	que define la función logaritmo neperiano, $\ln(x)$
$f3(x) = sin(x)$	que define la función seno, $\text{sen}(x)$
$f4(x) = cos(x)$	que define la función coseno, $\cos(x)$
$f5(x) = tan(x)$	que define la función tangente, $\tan(x)$
$f6(x) = sqrt(x)$	que define la función raíz cuadrada, $\sqrt{x}$

Usando estas funciones y las operaciones aritméticas podemos definir funciones como

F(x,y,z)=**sin**(xˆ2+yˆ2+zˆ2)+**sqrt**(**cos**(x)+**cos**(y)+**cos**(z)).

Como definimos la función indicando explícitamente el orden de sus argumentos no existe ambigüedad sobre qué variables se deben sustituir por qué argumentos:

```
f(a,b,c)  = a + 2*b + 3*c
f(1,2,1)
```

```
s = a + 2*b + 3*c
s(1,2,1)
```

```
__main__:4: DeprecationWarning: Substitution using function-call syntax
and unnamed arguments is deprecated and will be removed from a future
release of Sage; you can use named arguments instead, like EXPR(x=...,
y=...)
See http://trac.sagemath.org/5930 for details.
8
```

En este ejemplo vemos que podemos sustituir en una función, pero la sustitución en una expresión simbólica será imposible, dentro de Sage, en un futuro próximo.

Puedes encontrar más información acerca de las funciones predefinidas en Sage en esta página.

La otra manera de definir funciones matemáticas es usando la misma sintaxis que utilizamos, en Python, para definir programas o trozos de programas. Esto aparecerá en detalle más adelante, pero de momento podemos ver un ejemplo:

```
def f(x):
    return x*x
```

define la función *elevar al cuadrado.*

2.3.1. Variables y expresiones simbólicas

Una **variable simbólica** es un objeto en Python que representa una *variable*, en sentido matemático, que puede tomar cualquier valor en un cierto dominio. Casi siempre, ese dominio es un cuerpo de números, como los racionales, los reales o los números complejos. En el caso de los números racionales la representación es exacta, mientras que los reales, o los complejos, se representan usando decimales con un número dado de dígitos en la parte decimal.

1. La variable simbólica x está predefinida, y para cualquier otra que utilicemos debemos avisar al intérprete:

   ```
   var('a b c')
   ```

2. La igualdad a = 2 es una asignación que crea una variable a y le da el valor 2. Hasta que no se evalúe una celda que le asigne otro valor, por ejemplo a = 3, el valor de a será 2. Los valores de las variables, una vez asignados, se mantienen dentro de la hoja mientras no se cambien explícitamente.

3. Una operación que involucra una o más variables simbólicas no devuelve un valor numérico, sino una **expresión simbólica** que involucra números, operaciones, funciones y variables simbólicas.

   ```
   s = a + b + c
   s2 = a^2 + b^2 + c^2
   s3 = a^3 + b^3 + c^3
   s4 = a + b + c + 2*(a + b + c)
   p = a*b*c
   ```

4. Una igualdad como s = a + b + c es, de hecho, una asignación: después de ejecutar esa línea el valor de `s` es a + b + c y, por tanto, el de s^2 es $(a+b+c)^2$, etc.

5. Podemos imprimirlas como código

   ```
   print s; s2; s3; s4; p
   ```

```
a + b + c
a^2 + b^2 + c^2
a^3 + b^3 + c^3
3*a + 3*b + 3*c
a*b*c
```

o mostrarlas en un formato matemático más habitual

```
show([s, s2, s3,  s4,  p])
```

$$\left[a+b+c,\, a^2+b^2+c^2,\, a^3+b^3+c^3,\, 3\,a+3\,b+3\,c,\, abc\right]$$

6. Si en algún momento sustituimos las variables simbólicas por números (o elementos de un anillo), podremos realizar las operaciones y obtener un número (o un elemento de un anillo).

```
print s(a=1, b=1, c=1), s(a=1, b=2, c=3)
s(a=1, b=1, c=1)+ s(a=1, b=2, c=3)
```

```
3 6
9
```

La sustitución no cambia el valor de s, que sigue siendo una expresión simbólica, y sólo nos da el valor que se obtiene al sustituir. Obsérvese, en cambio, qué se obtiene si ejecutamos

```
a=1;b=1;c=1
print s
```

7. Si operamos expresiones simbólicas, obtenemos otras expresiones simbólicas, aunque pocas veces estarán simplificadas

```
ex = (1/6)*(s^3 - 3*s*s2 +2*s3 )
show(ex)
```

$$\frac{1}{6}\,(a+b+c)^3+\frac{1}{3}\,a^3+\frac{1}{3}\,b^3+\frac{1}{3}\,c^3-\frac{1}{2}\,(a+b+c)\left(a^2+b^2+c^2\right)$$

2.3.1.1. Simplificar expresiones

Observamos que al crear la expresión se han realizado "*de oficio*" algunas simplificaciones triviales. En ejemplos como el de arriba, nos puede interesar simplificar la expresión todavía más, pero es necesario decir qué queremos exactamente.

Existen varias estrategias para intentar simplificar una expresión, y cada estrategia puede tener más o menos éxito dependiendo del tipo de expresión simbólica. Algunas dan lugar a una expresión más sencilla en algunos casos, pero no en otros, y con expresiones complicadas pueden consumir bastante tiempo de proceso. Para la expresión anterior, como tenemos un polinomio, es buena idea expandirla en monomios que se puedan comparar unos con otros, usando el método .**expand**().

```
show(ex.expand())
```

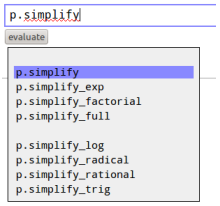
$$abc$$

A menudo nos interesa lo contrario: factorizar la expresión usando .**factor**()

```
p = a^3 + a^2*b + a^2*c + a*b^2 + a*c^2 + b^3 + b^2*c + b*c^2 + c^3
show(p)
show(p.factor())
```

$$a^3 + a^2b + a^2c + ab^2 + ac^2 + b^3 + b^2c + bc^2 + c^3$$
$$(a + b + c)\left(a^2 + b^2 + c^2\right)$$

Si consultas con el tabulador los métodos de las expresiones simbólicas, verás que hay métodos específicos para expresiones con funciones trigonométricas, exponenciales, con radicales o fracciones (es decir, con funciones racionales), ...




```
p = sin(3*a)
show(p.expand_trig())
```

$$-\sin(a)^3 + 3\,\sin(a)\cos(a)^2$$

```
p = sin(a)^2 - cos(a)^2
show(p.simplify_trig())
```

$$-2\,\cos(a)^2 + 1$$

```
p = 2^a * 4^(2*a)
show(p.simplify_exp())
```

$$2^{5\,a}$$

```
p = 1/a - 1/(a+1)
show(p.simplify_rational())
```

$$\frac{1}{a^2 + a}$$

2.3.2. Variables booleanas

Un tipo especial de variables es el *booleano* o lógico. Una tal variable toma valores `True` (verdadero) o `False` (falso). [6]

El doble signo igual (`==`) sirve para comparar, y devuelve **True** o **False** según los objetos comparados sean iguales o no para el intérprete. De la misma manera se pueden usar, siempre que tengan sentido, las comparaciones a<b y a<=b. Como veremos en el capítulo 4, estas comparaciones aparecen en los bucles **while** y al bifurcar la ejecución de código mediante un **if**.

Operaciones básicas con variables booleanas son la *conjunción* (**and**), la *disyunción* (**or**) y la *negación* (**not**):

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

	not
True	False
False	True

[6] Aunque, como veremos algo más adelante, también los valores 1, en lugar de `True`,y 0 en lugar de `False`.

2.4. Gráficas

2.4.1. Gráficas en 2D

Utilizaremos comandos como los que siguen para obtener objetos gráficos:

1. **point**(punto o lista), **points**(lista), **point2d**(lista), **point3d**(lista) dibujan los puntos de la lista que se pasa como argumento. Usaremos este tipo de gráficas, en particular, al estudiar la paradoja de Bertrand en el capítulo **??**.
2. **line**(lista), **line2d**(lista), **line3d**(lista) dibujan líneas entre los puntos de la lista que se pasa como argumento. Usaremos este tipo de gráficas, en particular, al estudiar el problema conocido como la ruina del jugador en el capítulo **??**.
3. **plot**(f,x,xmin=a,xmax=b) esboza una gráfica de la función de una variable f en el intervalo $[a, b]$. La escala en el eje OY la elige Sage automáticamente. En ocasiones, debido a esa elección automática, la gráfica mostrada consiste en un trozo encima del eje OX y otro prácticamente vertical, lo que no resulta muy útil. Podemos centrarnos en la parte que nos interesa del plano mediante **plot**(f,x,xmin=a,xmax=b,ymin=c,ymax=d), que muestra la parte de la gráfica contenida en el rectángulo $[a, b] \times [c, d]$.
4. **parametric_plot**([f(t),g(t)],(t,0,2)) esboza una gráfica de la curva dada en paramétricas mediante dos funciones de t, f y g, con la variable t variando en el intervalo $[0, 2]$.
5. Mediante **implicit_plot**(f,(x,−5,5),(y,−5,5)) obtenemos una representación de la parte de la curva $f(x, y) = 0$ contenida en el cuadrado $[-5, 5] \times [-5, 5]$. Decimos que se trata de una representación de una curva *"en implícitas"*.

Puedes ver algunos ejemplos de gráficas en el plano en la hoja de Sage `21-CAVAN-graficas.ipynb`.

2.4.2. Gráficas en 3D

El sistema para representaciones gráficas en 3D que tiene Sage no funciona en `jupyterhub`, es decir no muestra los gráficos cuando se ejecuta en el servidor del Departamento. En cambio, desde la versión **8.0** de Sage, funciona bien cuando se ejecuta en la versión local de `Jupyter`. Entonces, y hasta que se resuelva este problema, en el servidor usaremos `matplotlib`.

1. En general, `matplotlib` produce gráficos 3D a partir de listas de coordenadas de puntos, de la forma $(x, y, f(x, y))$, en $\mathbb{R}^3$. Así por ejemplo, para representar la gráfica de una función de dos variables $f(x, y)$ primero necesitamos una lista con coordenadas de puntos en el rectángulo $[a, b] \times [c, d]$ sobre el que deseamos obtener la gráfica.

```
x = np.arange(−2,2,0.05)
y = np.arange(−2,2,0.05)
X,Y = np.meshgrid(x, y)
```

   Esto nos dará $80 = 4/0.05$ puntos en el intervalo $[-2, 2]$ para la variable x, otros 80 para la y, y un retículo de 6400 puntos, uniformemente distribuidos, en el cuadrado $[-2, 2] \times [2, 2]$. Los comandos que empiezan por `np` son de `numpy`, y permiten la manipulación eficiente de listas y matrices. Veremos más sobre esto más adelante.

2. Ahora definimos la función que vamos a representar

```
def f(x,y):
    return np.cos(x**2+y**2)
```

Como usamos listas y matrices de `numpy`, las funciones matemáticas como el coseno también deben ser de `numpy`, y por eso usamos `np.cos`.

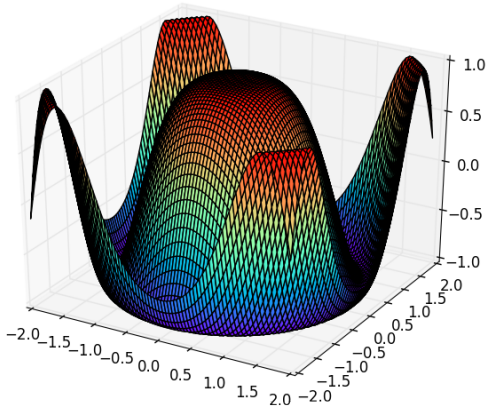
3. Finalmente, representamos la superficie mediante, por ejemplo,

```
fig  = plt.figure()
ax = fig.add_subplot(111, projection='3d')
zs = np.array([f(x,y) for x,y in zip(np.ravel(X), np.ravel(Y))])
Z = zs.reshape(X.shape)
ax.plot_surface(X,Y,Z, cmap=plt.cm.rainbow,rstride=1, cstride=1)
```

La tercera línea es la que crea las coordenadas de los puntos en $\mathbb{R}^3$ que usamos para construir la superficie, y la quinta es la que finalmente genera el gráfico interactivo.

4. Para obtener el gráfico primero, es decir en la primera celda que ejecutamos, hay que importar una serie de paquetes

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from math import *
%matplotlib notebook
```



Puedes ver un resumen del uso de `matplotlib` en este tutorial[7] y algunos ejemplos en la hoja

`21-CAVAN-graficas-matplotlib.ipynb`.

2.4.3. Gráficas 3D en el `notebook` antiguo

Las tres funciones principales para representaciones gráficas 3D son

1. **plot3d**(g,(x,−10,10),(y,−10,10)) esboza una gráfica de la función de dos variables g sobre el cuadrado $[-10, 10] \times [-10, 10]$.

[7] `IPython` es el nombre antiguo de `Jupyter`, pero la mayor parte de lo que se dice en este tutorial funciona en `Jupyter` sin problema.

2. **parametric_plot3d**([f(u,v),g(u,v),h(u,v)],(u,0,2),(v,0,2)) esboza una gráfica de la superficie dada en paramétricas mediante tres funciones de u y v, f, g y h, con las variables u y v en el intervalo $[0, 2]$.

   Los puntos de la superficie son entonces los que se obtienen mediante $x = f(u, v), y = g(u, v), z = h(u, v)$.

3. Con **implicit_plot3d**(f,(x,−5,5),(y,−5,5),(z,−5,5)) se representa una parte de la superficie $f(x, y, z) = 0$ contenida en el cubo $[-5, 5] \times [-5, 5] \times [-5, 5]$. Decimos que se trata de una representación de una superficie *"en implícitas"*.

Si se ejecutan en `Jupyter` se obtiene un gráfico vacío, pero en el `notebook` antiguo de Sage (1.1) funcionan perfectamente.

2.5. Cálculo

En esta sección usaremos la hoja de Sage `22-CAVAN-calculo-s1.ipynb`.

Cuando usamos el ordenador para estudiar problemas de cálculo diferencial o integral operamos frecuentemente con valores aproximados de los números reales implicados, es decir truncamos los números reales después de un número prefijado de decimales.

Sin embargo, en Sage podemos realizar gran cantidad de operaciones de manera "simbólica", es decir, sin evaluar de manera aproximada los números reales o complejos implicados. Así, por ejemplo, **sqrt**(2) es una representación simbólica de $\sqrt{2}$, una expresión cuyo cuadrado es exactamente 2, mientras que **sqrt**(2).**n**() (o bien **n**(**sqrt**(2))) es un valor aproximado con 20 decimales y su cuadrado no es exactamente 2. Para obtener una representación decimal de una expresión simbólica, podemos usar los comandos **n**(), **N**(), o los métodos homónimos .**n**(), .**N**() (n de numérico).[8]

El cálculo con valores aproximados de números reales necesariamente introduce errores, y suele llamarse *Cálculo numérico* a la materia que estudia cómo controlar esos errores, de manera que conocemos el grado de validez del resultado final y tratamos de que sea lo más alto posible. Volveremos en el capítulo **??** sobre este asunto.

La diferencia entre el *cálculo simbólico* y el *cálculo numérico* es importante: el cálculo simbólico es exacto pero más limitado que el numérico. Un ejemplo típico podría ser el cálculo de una integral definida

$$\int_a^b f(x)\ dx, \tag{2.1}$$

que simbólicamente requiere el cálculo de una primitiva $F(x)$ de $f(x)$ de forma que el valor de la integral definida es $F(b) - F(a)$. El cálculo de primitivas no es fácil, y para algunas funciones, como por ejemplo $\sin(x)/x$, la primitiva no se puede expresar usando las funciones habituales y, si fuera necesario, deberíamos considerarla como una nueva función elemental semejante a las trigonométricas o la exponencial.

En cambio, para cada función continua $f(x)$ en un intervalo $[a, b]$ podemos calcular fácilmente un valor aproximado para la integral definida (2.1). Veremos alguno de estos métodos en la sección **??**.

[8] Es muy habitual la asignación '$n =$ ', o '$N =$ ', cuando pensamos en dar nombre a una variable entera. Si la utilizamos en una hoja de Sage, el sistema no nos avisa de que estamos *tapando las funciones* **n**(), **N**(). Desde ese momento, y hasta que no reiniciemos la hoja, ya no funcionará la función. Se recomienda el uso de los métodos, .**n**(), .**N**(), para evitar innecesarios dolores de cabeza.

2.5.1. Ecuaciones

1. El comando **solve**() permite resolver ecuaciones (al menos lo intenta): para resolver una ecuación llamamos a **solve**() con la ecuación como primer argumento y la variable a despejar como segundo argumento, y recibimos una lista con las soluciones.

```
#Las soluciones de esta ecuación cúbica son números complejos
solve(x^3 - 3*x + 5, x)
```

2. El algoritmo resuelve también un sistema de ecuaciones. Basta pasar la *lista* de igualdades o expresiones.

```
var('x y')
solve([x^2+y^2==1,x-y+1],x,y)
```

3. El comando **solve**() intenta obtener soluciones exactas en forma de expresión simbólica, de la ecuación o sistema, y frecuentemente no encuentra ninguna. También es posible buscar soluciones aproximadas mediante métodos numéricos: el comando **find_root**(f,a,b) busca una solución de la ecuación $f(x) = 0$ perteneciente al intervalo $[a, b]$. Volveremos sobre este asunto en la sección **??**.

2.5.2. Límites

1. El método .**limit**() (o la función **limit**()) permite calcular límites de funciones. Para calcular el límite de `f` en un punto:

```
f1=x^2
print f1.limit(x=1)
```

2. También se puede calcular el límite cuando la variable tiende a infinito

```
f2=(x+1)*sin(x)
f3=f2/f1
f3.limit(x=+oo)
```

3. Si Sage sabe que el límite no existe, por ejemplo para la función $f(x) = 1/(x * sin(x))$, la función **limit**() devuelve el valor **und**, abreviatura de *undefined*, o el valor **ind**, que indica que no existe límite pero la función permanece acotada, por ejemplo $\sin(x)$ cuando x tiende a infinito, cerca del valor límite de la variable.

   Por último, en algunos casos Sage no sabe cómo determinar el límite o no sabe que no existe, y entonces devuelve la definición de la función. Esto ocurre, por ejemplo, con la función $f(x) := x/\sin(1/x)$ cuando se quiere calcular el límite cuando x tiende a cero.

4. También podemos calcular así límites de sucesiones. Al fin y al cabo, si la expresión simbólica admite valores reales, el límite de la función que define en infinito, si existe, es el mismo que el límite de la sucesión de naturales definido por la misma expresión.

2.5.3. Series

Una serie es un tipo particular de sucesión: dada una sucesión $a_0, a_1, a_2, \ldots$ queremos dar sentido a la suma infinita

$$S = a_0 + a_1 + a_2 + \cdots + a_n + \ldots$$

Definimos S como el límite, si existe, de la sucesión de "sumas parciales" $S_n := a_0 + a_1 + a_2 + \cdots + a_n$ cuando n tiende a infinito y decimos que S es la suma de la serie.

Para que pueda existir un límite para S_n tiene que ocurrir que el límite de a_n, cuando n tiende a infinito, sea cero. Es una condición necesaria pero NO SUFICIENTE para la existencia de suma de la serie. Por ejemplo, la sucesión $a_n := 1/n$ tiende a cero cuando n tiende a infinito, pero el límite de la correspondiente sucesión de sumas parciales es infinito.

El comando **sum**(), tomando ∞ como límite superior, permite, a veces, calcular la suma de una serie infinita:

sum(expresion, variable, limite_inferior, limite_superior)

```
sum(1/k^2, k, 1, oo).show()
```

$$\frac{1}{6}\pi^2$$

En casos como el anterior, Sage no realiza ningún cálculo para devolvernos su respuesta. Lo único que hace es identificar la serie que queremos sumar y nos devuelve el valor de la suma si lo conoce. Por ejemplo, si le pedimos la suma de los inversos de los cubos de enteros nos dice que vale zeta(3), que no es sino el nombre que se usa en Matemáticas para esa suma. Estudiaremos en detalle el cálculo con series en el capítulo **??**.

2.5.4. Cálculo diferencial

1. El método **.derivative**(), o la función **derivative**(), permite calcular derivadas de funciones simbólicas. Las **derivadas** se obtienen siguiendo metódicamente las reglas de derivación y no suponen ningún problema al ordenador:

   ```
   f=1/(x*sin(x))
   g=f.derivative()
   show([g,g.simplify_trig()])
   ```

2. Se pueden calcular derivadas de órdenes superiores

   ```
   m=5
   [derivative(x^m,x,j) for j in range(m+1)]
   ```

   Para derivar funciones de varias variables, es decir calcular derivadas parciales (i.e. derivar una función de varias variables respecto a una de ellas manteniendo las otras variables en valores constantes), basta especificar la variable con respecto a la que derivamos y el orden hasta el que derivamos:

   ```
   F(x,y)=x^2*sin(y)+y^2*cos(x)
   show([F.derivative(y,2),derivative(F,x,1).derivative(y,1)])
   ```

2.5.5. Desarrollo de Taylor

Así como la derivada de una función en un punto x_0 nos permite escribir una aproximación lineal de la función CERCA DEL PUNTO

$$f(x) \sim f(x_0) + f'(x_0)(x - x_0),$$

el polinomio de Taylor nos da una aproximación polinomial, de un cierto grado prefijado, válida CERCA DEL PUNTO. Así por ejemplo:

```
f(x)=exp(x)
taylor(f,x,0,20)
```

nos devuelve un polinomio de grado 20 que cerca del origen aproxima la función exponencial. Volveremos sobre este asunto en el capítulo **??**.

2.5.6. Cálculo integral

El cálculo de *primitivas* es mucho más complicado que la derivación, ya que no existe ningún método que pueda calcular la primitiva de cualquier función. En realidad, hay muchas funciones elementales (construídas a partir de funciones trigonométricas, exponenciales y algebraicas mediante sumas, productos y composición de funciones) cuyas primitivas, aunque estén bien definidas, no se pueden expresar en términos de estas mismas funciones. Los ejemplos $f(x) = e^{-x^2}$ y $f(x) = \frac{\sin(x)}{x}$ son bien conocidos.

Aunque en teoría existe un algoritmo (el algoritmo de Risch) capaz de decidir si la primitiva de una función elemental es elemental, dificultades prácticas imposibilitan llevarlo a la práctica, y el resultado es que incluso en casos en los que integramos una función cuya primitiva es una función elemental, nuestro algoritmo de integración simbólica puede no darse cuenta. Si Sage no puede calcular una primitiva de $f(x)$ explícita devuelve

$$\int f(x)\ dx.$$

1. Los métodos (funciones) para el cálculo de primitivas a nuestra disposición son: **.integral**() e **.integrate**().

   ```
   f=1/sin(x)
   show(f.integrate(x))
   ```

2. Siempre podemos obtener una aproximación numérica de una integral definida:

   ```
   f=tan(x)/x
   [numerical_integral(f,pi/5,pi/4),N(f.integrate(x,pi/5,pi/4))]
   ```

   Obsérvese que, con **numerical_integral**(), el intérprete devuelve una tupla, con el valor aproximado de la integral y una cota para el error.

3. También es posible integrar numéricamente funciones de varias variables, aunque, de momento, no vamos a tratar el tema. Así como las integrales de funciones de una variable corresponden al cálculo de áreas, las integrales de funciones de varias variables permiten calcular volúmenes e hipervolúmenes.

   En el capítulo **??** veremos un método, conocido como integración de Monte Carlo que permite calcular valores aproximados de integrales de funciones de una o varias variables.

2.6. Álgebra lineal

Podemos decir que el Álgebra Lineal, al menos en espacios de dimensión finita, trata de la *resolución de todos aquellos problemas que pueden reducirse a encontrar las soluciones de un sistema de ecuaciones de primer grado en todas las incógnitas,* es decir, un sistema lineal de ecuaciones. Es posible realizar todas las operaciones necesarias para resolver tales sistemas mediante cálculo con matrices, y esa es la forma preferida para resolver problemas de Álgebra Lineal mediante ordenador.

Los sistemas de ecuaciones lineales con coeficientes en un cuerpo, por ejemplo el de los números racionales, siempre se pueden resolver y encontrar explícitamente todas las soluciones del sistema en ese cuerpo o en uno que lo contenga. Esto no es cierto para ecuaciones polinomiales de grado más alto, y, por ejemplo, no hay métodos generales para resolver una única ecuación polinomial, de grado arbitrario n, en una única variable

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = 0.$$

Veremos en la parte final del curso algo de los que se puede decir acerca de la resolución de los sistemas de ecuaciones polinomiales.

2.6.1. Construcción de matrices

El constructor básico de matrices en Sage es la función **matrix**(), que, en su forma más simple, tiene como argumentos

1. El conjunto de números, enteros, racionales, etc., al que pertenecen las entradas de la matriz.
2. Una lista cuyos elementos son listas de la misma longitud, y que serán las *filas* de la matriz.

```
A=matrix(ZZ,[[1,2,3],[4,5,6]]);A;show(A)
```

Una variante útil contruye la misma matriz con

```
A=matrix(ZZ,2,[1,2,3,4,5,6]);show(A)
```

que trocea la lista dada como tercer argumento en el número de filas dado por el segundo, en este caso 2 filas. Por supuesto, el número de elementos de la lista debe ser múltiplo del segundo argumento o aparece un error.

Para más ejemplos e información sobre **matrix**(), pulsar el `tabulador` del teclado tras escribir **matrix**(en una celda.

2.6.2. Submatrices

Sobre el acceso a las entradas de una matriz se ha de tener en cuenta que tanto filas como columnas empiezan su numeración en 0. Así, en la matriz 3×2 definida por A=**matrix**(3,**range**(6)), es decir

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{pmatrix}$$

el extremo superior izquierdo tiene *índices* '0,0', y el inferior derecho, '2,1'. Una vez claros los índices de un elemento, Sage usa la notación *slice* (por cortes) de Python para acceder a él: A[0][0] y A[2][1] serían los elementos recién mencionados. Para el caso especial de matrices, se ha adaptado también una notación más habitual: A[0,0] y A[2,1].

En la notación *slice*, dos índices separados por dos puntos, i:j, indican el corte entre el primer índice, incluido, hasta el segundo, que se excluye. En el ejemplo, se muestran las filas de índices 2 y 3 de la matriz. Si se excluye alguno de los índices, se toma, por defecto, el valor más extremo.

```
A=matrix(ZZ,10,[1..100])
A[7:]
```

```
[ 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]
```

La matriz A[7:] consiste en las filas desde la octava, que tiene índice 7 porque hemos empezado a contar en 0, hasta la décima.

```
A=matrix(10,[1..100])
A[:2]
```

```
[ 1 2  3  4  5  6  7  8  9 10]
[11 12 13 14 15 16 17 18 19 20]
```

La matriz A[:2] consiste en las filas primera y segunda, es decir, las de índice menor estrictamente que 2. Utilizando el doble índice, podemos recortar recuadros más concretos:

```
A=matrix(10,[1..100])
A[3:5,4:7]
```

```
[35 36 37]
[45 46 47]
```

Unas últimas virguerías gracias a la notación slice: saltos e índices negativos. La matriz A es, como en el ejemplo anterior, la matriz 10×10 con los primeros cien enteros colocados en orden en las filas de A.

```
A[2:5:2,4:7:2]
```

produce

```
[25 27]
[45 47]
```

se queda con las filas con índice del 2 al 4, saltando de dos en dos debido al 2 que aparece en tercer lugar en 2:5:2, y con las columnas con índices del 4 al 6 también saltando de dos en dos por el que aparece en 4:7:2. Siempre hay que recordar que las filas y columnas se numeran empezando en el cero.

Por otra parte, también podemos usar índices negativos lo que conduce a quedarnos con filas o columnas que contamos hacia atrás, de derecha a izquierda, empezando por las últimas:

```
A[-1];A.column(-2)
```

```
(91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
(9, 19, 29, 39, 49, 59, 69, 79, 89, 99)
```

La primera fila del resultado, que corresponde a A[−1], es la última fila de la matriz, y la segunda es la novena columna (que tiene índice 8). La última columna se obtendría con A.**column**(−1).

2.6.3. Operaciones con matrices

Las operaciones entre matrices, suma, diferencia, multiplicación, potencia e inversa, se denotan con los mismos símbolos que las correspondientes operaciones entre números. En el caso de operaciones entre matrices es posible que la operación no sea posible porque los tamaños de las matrices implicadas no sean compatibles, o en el caso de la inversa porque la matriz no sea cuadrada de rango máximo.

De entre todas las funciones y métodos que se aplican a matrices hemos seleccionado las que parecen más útiles:

- **identity_matrix**(**n**).- matriz identidad $n \times n$
- A.**det**(), **det**(A).- determinante de A
- A.**rank**(), **rank**(A).- rango de A
- A.**trace**().- traza de A
- A.**inverse**().- inversa de A
- A.**transpose**().- traspuesta de A
- A.**adjoint**().- matriz adjunta de A
- A.**echelonize**(), A.**echelon_form**().- matriz escalonada de A, en el menor anillo en que vivan sus entradas
- A.**rref**().- matriz escalonada de A, en el menor cuerpo en que coincidan sus entradas

Dado que resolver un sistema lineal de ecuaciones consiste, desde un punto de vista matricial, en obtener la forma escalonada de la matriz (reducción gaussiana), la instrucción quizá más importante en la lista es A.**echelon_form**().

Aunque A.**echelonize**() y A.**echelon_form**() hacen esencialmente lo mismo, la primera deja la forma escalonada en la variable A, por tanto machaca el valor antiguo de A, y no es posible asignar el resultado a otra variable, es decir B = A.**echelonize**() no funciona, mientras que la segunda forma funciona como esperamos B=A.**echelon_form**() hace que la forma escalonada quede en B y la matriz A todavía existe con su valor original. Puedes comprobar este comportamiento en la hoja `23-CAVAN-reduccion-gaussiana.ipynb`.

2.6.4. Espacios vectoriales

Operar con matrices en el ordenador *siempre* ha sido posible debido a que los lenguajes de programación disponen de la estructura de datos **array**, que esencialmente es lo mismo que una matriz. En los sistemas de cálculo algebraico, como Sage, existe la posibilidad de definir objetos matemáticos mucho más abstractos como, por ejemplo, grupos, espacios vectoriales, grafos, anillos, etc.

Nos fijamos en esta subseccción en el caso de los espacios vectoriales. ¿Qué significa definir, como objeto de Sage, un espacio vectorial E de dimensión 3 sobre el cuerpo de los números racionales?

Lo definimos mediante E = **VectorSpace**(**QQ**,3) y sus elementos, vectores, mediante, por ejemplo, v = **vector**([1,2,3]) o bien v = E([1,2,3]).

Una vez definido el espacio vectorial y vectores en él, podemos realizar cálculos con vectores siguiendo las reglas que definen, en matemáticas, los espacios vectoriales. Así por ejemplo, el sistema sabe que $\mathbf{v} + (-1) * \mathbf{v} = \mathbf{0}$.

También es posible construir espacios de matrices con una instrucción como M = MatrixSpace(**QQ**, 4, 4), que define M como el conjunto de matrices con coeficientes racionales 4×4, con operaciones de suma, producto por escalares, producto de matrices y matriz inversa.

2.6.5. Subespacios vectoriales

1. Podemos definir fácilmente el subespacio engendrado por un conjunto de vectores, y después realizar operaciones como intersección o suma de subespacios, o comprobaciones como igualdad o inclusión de subespacios.

```
V1 = VectorSpace(QQ,3)
v1 = V1([1,1,1])
v2 = V1([1,1,0])
L1 = V1.subspace([v1,v2])
print L1
```

2. Definido un subespacio, podemos averiguar información sobre él:

```
print dim(L1); L1.degree(); L1.ambient_vector_space()
```

   Sage llama *degree* de un subespacio a la dimensión de su espacio ambiente.

3. Muchos operadores actúan sobre subespacios vectoriales, con los significados habituales.

```
# Pertenencia a un subespacio
v3, v4 = vector([1,0,1]),  vector([4,4,3])
print v3 in L1; v4 in L1
```

```
#Comprobación de igualdad
print L1 == V1
print L1 == V1.subspace([v1,v1+v2])
```

```
#Comprobación de inclusión
print L1 <= V1; L1 >= V1; L1 >= V1.subspace([v1])
```

```
#Intersección y suma de subespacios
L1 = V1.subspace([(1,1,0),(0,0,1)])
L2 = V1.subspace([(1,0,1),(0,1,0)])
L3 = L1.intersection(L2)
print '* Intersección de subespacios: '
print L3
L4 = L1+L2
print '* Suma de subespacios: ';L4
```

2.6.6. Bases y coordenadas

Como hemos visto, se define un subespacio de un espacio vectorial en Sage mediante un conjunto de generadores del subespacio, pero internamente se guarda mediante una base del subespacio. Esta base, en principio no es un subconjunto del conjunto generador utilizado para definir el subespacio, pero podemos imponer que lo sea con el método **subspace_with_basis**:

```
L1 = V1.subspace_with_basis([v1,v2,v3])
print L1.basis(); L1.basis_matrix()
```

El método **.coordinates**() nos da las coordenadas de un vector en la base del espacio

```
print (L1.coordinates(v1), L2.coordinates(v1))
```

2.6.7. Producto escalar

Podemos definir un espacio vectorial con una forma bilineal mediante

```
V = VectorSpace(QQ,2, inner_product_matrix=[[1,2],[2,1]])
```

Acabamos con una lista, incompleta, de funciones y métodos relacionados con el producto escalar de vectores.

- u.**dot_product**(v).- producto escalar, $u \cdot v$
- u.**cross_product**(v).- producto vectorial, $u \times v$
- u.**pairwise_product**(v).- producto elemento a elemento
- **norm**(u), u.**norm**(), u.**norm**(2).- norma Euclídea
- u.**norm**(1).- suma de coordenadas en valor absoluto
- u.**norm**(Infinity).- coordenada con mayor valor absoluto
- u.**inner_product**(v).- producto escalar utilizando la matriz del producto escalar

2.7. Ejercicios

2.7.1. Inducción y sucesiones

Ejercicio 1. *Demuestra por inducción sobre $n \in \mathbb{N}$ las afirmaciones siguientes:*

1. $1^2 + 2^2 + \cdots + n^2 = \dfrac{n(n+1)(2n+1)}{6}$.
2. $\dfrac{1}{1\cdot 2} + \dfrac{1}{2\cdot 3} + \cdots + \dfrac{1}{n\cdot(n+1)} = \dfrac{n}{n+1}$, *si* $n \geq 1$.

3. $1 \cdot 1! + 2 \cdot 2! + \cdots + n \cdot n! = (n+1)! - 1$.

4. $\dfrac{1}{2} + \dfrac{2}{2^2} + \dfrac{3}{2^3} + \cdots + \dfrac{n}{2^n} = 2 - \dfrac{n+2}{2^n}$.

5. $(1+q)(1+q^2)(1+q^4)\cdots(1+q^{2^n}) = \dfrac{1-q^{2^{n+1}}}{1-q}$.

En este ejercicio se pide una "demostración matemática" por inducción. Más adelante veremos cómo programar la comprobación, hasta un n prefijado, de fórmulas copmo estas.

Ejercicio 2. *La sucesión de Fibonacci, $\{F_n\}$, está definida por medio de la ley de recurrencia:*

$$F_1 = 1, \quad F_2 = 1, \quad F_{n+2} = F_n + F_{n+1}.$$

Calcular los diez primeros términos de la sucesión y comprobar, para $1 \le n \le 10$, la siguiente identidad:

$$F_n = \frac{\left[\frac{1+\sqrt{5}}{2}\right]^n - \left[\frac{1-\sqrt{5}}{2}\right]^n}{\sqrt{5}}.$$

La sucesión de Fibonacci vuelve a aparecer varias veces a lo largo del curso, sobre todo en la página **??** *y siguientes.*

Ejercicio 3. *Demostrar por inducción la fórmula para la suma de los primeros n cubos*

$$1^3 + 2^3 + \ldots + n^3 = \frac{(n+1)^2\, n^2}{4}\,.$$

En el ejercicio 13 se plantea un método para encontrar esta fórmula.

Ejercicio 4. *Estudiar el límite de las siguientes sucesiones*

(a) $\left\{\frac{n^2}{n+2}\right\}$; (b) $\left\{\frac{n^3}{n^3+2n+1}\right\}$; (c) $\left\{\frac{n}{n^2-n-4}\right\}$; (d) $\left\{\frac{\sqrt{2n^2-1}}{n+2}\right\}$;

(e) $\left\{\frac{\sqrt{n^3+2n}+n}{n^2+2}\right\}$; (f) $\left\{\frac{\sqrt{n+1}+n^2}{\sqrt{n+2}}\right\}$; (g) $\left\{\frac{(-1)^n n^2}{n^2+2}\right\}$; (h) $\left\{\frac{n+(-1)^n}{n}\right\}$;

(i) $\left\{\left(\frac{2}{3}\right)^n\right\}$; (j) $\left\{\left(\frac{5}{3}\right)^n\right\}$; (k) $\left\{\frac{2^n}{4^n+1}\right\}$; (l) $\left\{\frac{3^n+(-2)^n}{3^{n+1}+(-2)^{n+1}}\right\}$;

(m) $\left\{\frac{n}{n+1} - \frac{n+1}{n}\right\}$; (n) $\left\{\sqrt{n+1} - \sqrt{n}\right\}$; (ñ) $\left\{\frac{1}{n^2} + \frac{2}{n^2} + \cdots + \frac{n}{n^2}\right\}$.

Atención: la sucesión $\frac{3^n+(-2)^n}{3^{n+1}+(-2)^{n+1}}$ tiene límite $\frac{1}{3}$. El siguiente código

```
var('m')
l(m)=3^m+(-2)^m
ll=l(m)/l(m+1)
ll.limit(m=oo)
```

devuelve, incorrectamente, 0, pero es fácil ayudar al intérprete:

```
var('m')
l(m)=3^m+(-2.0)^m
ll=l(m)/l(m+1)
ll.limit(m=oo)
```

```
1/3
```

¿Cuál puede ser la explicación?

Ejercicio 5. *Calcular, si existen, los límites de las sucesiones que tienen como término general*

$$a_n = \left(\frac{n^2+1}{n^2}\right)^{2n^2-3}, \quad b_n = \left(\frac{n^2-1}{n^2}\right)^{2n^2+3}, \quad c_n = a_n + \frac{1}{b_n}\,.$$

2.7.2. Un ejercicio de cálculo

Consideramos el polinomio en la variable x y dependiente de dos parámetros reales, a y b, dado por

$$p(x, a, b) := x^4 - 6x^2 + ax + b,$$

y queremos estudiar el número de raíces reales que tiene dependiendo del valor de los parámetros.

1. Es un teorema importante, llamado en ocasiones el *teorema fundamental del álgebra*, el hecho cierto de que todo polinomio de grado n con coeficientes complejos tiene exactamente n raíces complejas si se cuentan con sus multiplicidades.
2. Si un polinomio tiene coeficientes reales entonces tiene un número par de raíces complejas no reales, ya que si un complejo no real es raíz su complejo conjugado también lo es.
3. En consecuencia, el número de raíces reales de un polinomio de grado cuatro con coeficientes reales es siempre par.
4. Supongamos para empezar que $a = 0$, y podemos empezar dibujando la gráfica para diversos valores de b. Si no estaba claro desde el principio, vemos que todas las gráficas son iguales, la joroba doble de un dromedario invertida, y que al crecer b la gráfica "sube" respecto a los ejes.

   Entonces, para valores suficientemente grandes de b no habrá ninguna raíz real y para valores muy negativos de b habrá dos raíces reales. Para valores intermedios debería haber cuatro raíces reales ya que el eje OX puede cortar a las dos jorobas.
5. Cuando $a = 0$ podemos tratar el problema usando que podemos escribir

   $$p(x, 0, b) = (x^2 - 3)^2 - 9 + b,$$

   y vemos inmediatamente que si $9 - b < 0$ no puede haber raíces reales y las cuatro raíces son números complejos no reales. Los casos restantes ($b \leq 9$) se pueden tratar de forma similar, resolviendo explícitamente la ecuación $p(x, 0, b) = 0$.
6. Supongamos ahora que $a \neq 0$ y vamos a tratar el caso particular en que $a = 8$. Si observamos cuidadosamente la gráfica para $a = 8$ y $b = 0$ vemos que una de las jorobas ha desaparecido y la parte de abajo de la gráfica se ha aplanado. ¿Por qué? Cuando estamos cerca del origen los dos primeros sumandos de $p(x, a, b)$ toman valores muy pequeños, si x es pequeño x^4 es mucho más pequeño, y el término $ax + b$ domina.
7. Para ver el motivo de la desaparición de una de las jorobas debemos calcular los máximos y mínimos de la función, es decir, debemos derivar e igualar a cero. Vemos entonces que lo que tiene de especial el valor $a = 8$ es que para ese valor y $x = 1$ se anulan la primera y la segunda derivadas. En la gráfica eso se ve como una zona casi horizontal cerca de $x = 1$.
8. En la gráfica también parece verse que la función $p(x, 8, 0)$ es creciente para $x \geq 0$. ¿Es esto verdad?
9. Debemos pensar que la desaparición de una de las jorobas implica que ahora sólo puede haber 2 raíces reales o bien ninguna. ¿Cómo se puede probar esta afirmación y para qué valores de b se obtendría cada uno de los casos?
10. ¿Puedes decir algo, acerca del número de raíces reales en función de a y b, si $a \neq 0$ y $a \neq \pm 8$?

Puedes ver una solución en la hoja `24-CAVAN-raices-reales.ipynb`, y de la parte más matemática en este archivo.

2.7.3. Algunos ejercicios más

Ejercicio 6. *Comenzamos con el semicírculo de radio uno y centro el origen. Elegimos un ángulo* $0 < \theta < \pi/2$ *y representamos las rectas* $x = sen(\theta)$ *y* $x = -cos(\theta)$ *de forma que junto con el eje* $y = 0$ *y la circunferencia encierran una región que llamamos* A.

Llamamos B *al complemento de* A *en el semicírculo.* DETERMINAR *el valor máximo, cuando* θ *varía, del cociente* $F(\theta) := Area(A)/Area(B)$.

Ejercicio 7. *Consideramos la parábola de ecuación* $y = x^2$ *y una circunferencia de centro* $(0, a)$ *y tal que es tangente*[9] *a la parábola en dos puntos distintos.* DETERMINAR *los valores de* a *para los que tal circunferencia existe.*

Ejercicio 8. *Determinar el valor de* m *que hace que la ecuación*

$$x^4 - (3m + 2)x^2 + m^2 = 0$$

tenga cuatro raíces reales en progresión aritmética (i.e. las raíces serían $x_0, x_0 + a, x_0 + 2a, x_0 + 3a$*).*

Ejercicio 9. *Sea* M *un real muy grande. Demuestra que la ecuación* $Mx = e^x$ *tiene una solución, digamos* w*, única con* $w > 1$*. Estima el valor de* w*, usando como guía representaciones gráficas de* $f(x) := e^x - Mx$*, después de darle a* M *un valor suficientemente grande. ¿Podrías estimar el valor de* M *a partir del que se obtiene la solución* w *única?¿Qué ocurre para valores de* M *más pequeños?*

Ejercicio 10. *Determina el valor mínimo de la constante* $a > 1$ *tal que siempre que* $x \leq y$ *se verifica*

$$\frac{a + sen(x)}{a + sen(y)} \leq e^{y-x}.$$

Ejercicio 11. *¿Es mayor* e^π *que* π^e*? A una pregunta así podríamos responder calculando los dos números reales, con cierta aproximación, y resulta que ciertamente es mayor* e^π*. Sin embargo, se trata de demostrarlo en la forma más convincente posible, sin usar el valor aproximado. Dicho de otra manera, nuestro argumento nos debe conducir a una demostración puramente matemática, para la que podemos usar como apoyo el ordenador (gráficas, derivadas, etc.).*

Puedes ver algo de ayuda para los primeros dos ejercicios en este archivo. Además, en la hoja `27-CAVAN-tangencias.ipynb` puedes ver una animación de la solución del segundo.

2.7.4. Ejercicios de Álgebra Lineal

Antes de plantear una lista de ejercicios resolvemos, a modo de ejemplo, un problema de interpolación. Si bien la solución propuesta dista de ser la más eficiente, nos sirve como motivación a la manera de hacer que se pretende en este curso. En una hoja de Sage, tanto el enunciado como los comentarios en la solución, deberían aparecer en cuadros de texto.

Ejercicio 12. Método de coeficientes indeterminados. *Encontrar el polinomio de menor grado cuya gráfica pasa por los puntos*

$$(-2, 26), (-1, 4), (1, 8), (2, -2)\,.$$

SOLUCIÓN.- Puesto que se tienen 4 puntos, se considera un polinomio general, de grado ≤ 3: $P(x) = a_0 + a_1x + a_2x^2 + a_3x^3$.

[9] Dos curvas son tangentes en un punto en que se cortan si tienen la misma recta tangente en ese punto.

Con las coordenadas de los 4 puntos dados, sustituyendo las abcisas e igualando a las respectivas ordenadas, se obtiene un sistema lineal de 4 ecuaciones con 4 incógnitas (los coeficientes, a_0, a_1, a_2, a_3, del polinomio):

$$\begin{aligned} 1*a_0 - 2*a_1 + 4*a_2 - 8*a_3 &= 26 && \text{punto } (-2, 26)\\ 1*a_0 - 1*a_1 + 1*a_2 - 1*a_3 &= 4 && \text{punto } (-1, 4)\\ 1*a_0 + 1*a_1 + 1*a_2 + 1*a_3 &= 8 && \text{punto } (1, 8)\\ 1*a_0 + 2*a_1 + 4*a_2 + 8*a_3 &= -2 && \text{punto } (2, -2) \end{aligned}$$

El hecho de que los 4 puntos se encuentren en distintas verticales, asegura que el sistema es compatible determinado. Su matriz es

$$\begin{pmatrix} 1 & -2 & 4 & -8\\ 1 & -1 & 1 & -1\\ 1 & 1 & 1 & 1\\ 1 & 2 & 4 & 8 \end{pmatrix}$$

con inversa

$$\begin{pmatrix} -\frac{1}{6} & \frac{2}{3} & \frac{2}{3} & -\frac{1}{6}\\ \frac{1}{12} & -\frac{2}{3} & \frac{2}{3} & -\frac{1}{12}\\ \frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} & \frac{1}{6}\\ -\frac{1}{12} & \frac{1}{6} & -\frac{1}{6} & \frac{1}{12} \end{pmatrix}$$

y la solución del sistema, obtenida multiplicando la matriz inversa por el vector columna de términos independientes, es el vector $(4, 5, 2, -3)$, es decir, el polinomio $4 + 5x + 2x^2 - 3x^3$. ■

Ejercicio 13. *La suma de los n primeros enteros positivos, $1 + 2 + \cdots + n$, es un polinomio en n de grado 2: $\frac{1}{2}n^2 + \frac{1}{2}n$. La suma de sus cuadrados, $1^2 + 2^2 + \cdots + n^2$, es un polinomio de grado 3. En general, fijada la potencia, k, la suma $1^k + 2^k + 3^k + \cdots + n^k$ es un polinomio en n de grado $k + 1$ (¿*POR QUÉ*?).*

Encontrar, usando interpolación como en el ejercicio anterior, una fórmula para la suma de los cubos de los primeros n enteros positivos.[10]

Podemos comprobar el resultado obtenido mediante las instrucciones

```
var('m k');sum(k^3,k,1,m)
```

que devuelve directamente el polinomio buscado, en m.

Ejercicio 14. *Dada una matriz $\mathbf{A}$ de tamaño $m \times n$ con entradas racionales, podemos definir una función $\Phi_{\mathbf{A}}$, entre espacios vectoriales de matrices, mediante*

$$\begin{array}{ccc} \mathcal{M}_{n\times k} & \xrightarrow{\Phi_{\mathbf{A}}} & \mathcal{M}_{m\times k}\\ \mathbf{B} & \mapsto & \mathbf{A}\cdot\mathbf{B}, \end{array}$$

con $\mathcal{M}_{n\times k}$ ($\mathcal{M}_{m\times k}$) los espacios de matrices con n filas y k columnas (m filas y k columnas).

Cuando $k = 1$, la aplicación lineal $\Phi_{\mathbf{A}}$ es bien conocida: se trata de la obtenida al multiplicar vectores columna de n filas, colocados a la derecha de $\mathbf{A}$, por la propia matriz, y nos referimos a ella como la **aplicación lineal asociada** *a la matriz $\mathbf{A}$. Sabemos que toda aplicación lineal $u : \mathbb{Q}^n \to \mathbb{Q}^m$ es la aplicación asociada a una matriz $\mathbf{U}$, a la que llamamos* LA MATRIZ *de la aplicación lineal.*

1. *Demostrar (completamente) que $\Phi_{\mathbf{A}}$ es, de hecho, una aplicación lineal.*
2. *Supongamos que la matriz $\mathbf{A}$ es invertible, y, por tanto, cuadrada. ¿Será cierto que $\Phi_{\mathbf{A}}$ es necesariamente biyectiva (es decir, invertible)?*
3. *Supongamos que la aplicación lineal que corresponde a $\mathbf{A}$* NO *es inyectiva. ¿Es cierto que $\Phi_{\mathbf{A}}$ no puede ser inyectiva?*

[10] En el ejercicio 3, se pide una demostración, para este caso, de la afirmación de que estas sumas son polinomios en el número de sumandos.

4. *Supongamos que $m < n$, ¿es cierto que si la aplicación lineal que corresponde a $\mathbf{A}$ es suprayectiva (es decir, de rango m) también será suprayectiva la aplicación lineal $\Phi_{\mathbf{A}}$?¿Es cierta la afirmación recíproca?*

5. *Supongamos ahora que $\mathbf{A}$ es 2×2. En este caso, la aplicación lineal $\Phi_{\mathbf{A}}$ va del espacio de matrices $2 \times n$ en sí mismo.*

   a) *¿Podrías calcular la matriz de $\Phi_{\mathbf{A}}$ en las bases estándar de los espacios de matrices (las matrices de esas bases tienen todas sus entradas 0 menos una que vale 1)?*

   b) *Determina el núcleo de $\Phi_{\mathbf{A}}$, y discute los casos $rango(\mathbf{A}) = 0, 1, 2$.*

   c) *Calcula una base del núcleo de $\Phi_{\mathbf{A}}$ en cada uno de los tres casos.*

Capítulo 3

Estructuras de datos

Muchos de los programas que elaboramos sirven para transformar y analizar datos que creamos o recibimos. En este resumen se revisan las estructuras de datos que llamamos `lista, tupla, cadena de caracteres, conjunto y diccionario.` Una característica común a todos estos tipos es que son ITERABLES, es decir, que para todos ellos tiene sentido hacer un bucle **for** como

```
for item in <estructura_de_datos>:
    ........
```

Nos referimos a estas estructura de datos con el nombre genérico de *contenedor*, o más precisamente *contenedor iterable.*

Otra cualidad común de los contenedores iterables es la de poder comprobar fácilmente si un dato está o no en ellas. Basta utilizar la partícula **in**, de manera que

```
dato in <estructura_de_datos>
```

adquiere valor **True** o **False**.

Por último, la función **len**() nos informa del número de datos en cualquiera de estas estructuras.

Existen, en Python, otros tipos de estructuras de datos más sofisticados pero en este curso nos bastará con los mencionados. Puedes consultar esta página para obtener más información.

3.1. Datos básicos: tipos

Algunos lenguajes de programación, por ejemplo C, obligan a declarar explícitamente de qué tipo es cada variable. Así por ejemplo, en C hay tipos como

1. `long` para almacenar enteros, que deben pertenecer al intervalo[1]

$$[-2147483648, 2147483648]$$

. Cada entero, de tipo `long`, utiliza 32 bits en la memoria del ordenador.

2. `float` para números decimales, de hasta 8 cifras decimales, pertenecientes al intervalo

$$[1.175494351 \cdot 10^{-38}, 3.402823466 \cdot 10^{38}].$$

Cada decimal, de tipo `float`, utiliza 32 bits en la memoria del ordenador.

[1] Este rango depende del compilador de C utilizado, pero el indicado es el habitual.

3. `double` para números decimales de mayor precisión, dieciseis cifras decimales, que los dados por `float`, concretamente, pertenecientes al intervalo

$$[2.2250738585072014 \cdot 10^{-308}, 1.7976931348623158 \cdot 10^{308}].$$

   Cada decimal, de tipo `double`, utiliza 64 bits en la memoria del ordenador.

4. `char` en los que se almacenan los caracteres del código ASCII, las letras y símbolos de puntuación.

   Cada carácter, tipo `char`, utiliza 8 bits en la memoria.

La declaración de tipos es importante porque hace posible la compilación del programa, es decir la conversión del código en un ejecutable que puede funcionar de manera totalmente independiente. Para ese funcionamiento independiente, en otro ordenador en el que no están necesariamente ni el código ni el compilador de C, es esencial que el ejecutable contenga toda la información sobre la gestión de la memoria RAM, que es lo que explica la necesidad de la declaración de tipos. El compilador traduce el código C a código máquina del procesador que tenga el ordenador que usamos, de forma que el ejecutable funcionará en cualquier otro ordenador con ese procesador.

En Sage, y en Python, no es necesario declarar los tipos de las variables debido a que el código no se compila, sino que se *interpreta*: el código se va traduciendo, por el *intérprete de Python*, a lenguaje máquina *sobre la marcha*, y es el intérprete el que se ocupa de la gestión de la memoria RAM. La ejecución de código en lenguajes interpretados es, en general, mucho más lenta que en lenguajes compilados, pero en cambio es mucho más cómodo programar en Python (Sage) que en C.

En Sage los datos tienen tipos, aunque no hay que declararlos, y se pueden producir *errores de tipo (Type error)*. Por ejemplo, si intentamos factorizar un número decimal, se producirá un tal error.

Los objetos básicos en Sage, a partir de los que se construyen objetos complejos, son similares a los de C: enteros, decimales, caracteres. Dado el enfoque de Sage hacia las matemáticas, hay otros tipos de datos, números racionales o complejos, que también podemos considerar básicos.

Cuando nos encontramos con un error de tipo, al aplicar una función de Sage a un cierto objeto A, podemos evaluar A.**type**() o **type**(A), lo que nos informará del tipo que tiene el objeto A. Analizando la documentación sobre la función podremos darnos cuenta de que esa función concreta no se puede aplicar a los objetos del tipo de A, y quizá deberemos cambiar la definición de A.

Hay dos tipos diferentes de enteros: los enteros de Python y los enteros de Sage. La única diferencia, en la práctica, es que no es posible aplicar a los enteros de Python los métodos disponibles para los enteros de Sage. Por ejemplo, los enteros de Sage tienen el método .**digits**(), que nos devuelve una lista ordenada de los dígitos del entero, mientras que si intentamos aplicar este método a un entero de Python obtenemos un error de tipo.

En este capítulo veremos la forma de crear estructuras de datos complejas, que contienen objetos básicos de alguno de los tres tipos, o bien otras estructuras de datos. Así, por ejemplo, podemos pensar una matriz como una lista que contiene listas de la misma longitud, que serán las filas de la matriz. Estas construcciones tienen, como veremos, algunas restricciones, y así, por ejemplo, no podemos crear un conjunto cuyos elementos sean listas.

3.2. Listas

En este curso USAREMOS SOBRE TODO LISTAS Y MATRICES como estructuras de datos, y las otras estructuras de datos servirán sobre todo en situaciones particulares en las que necesitaremos un contenedor más especializado.

1. Definimos una lista separando sus elementos por comas y delimitando el principio y el final de la lista con corchetes cuadrados:

```
L = [1,2,3,4,5]
type(L), len(L), L
```

```
(<type 'list'>,  5,  [1,  2,  3,  4,  5])
```

   El nombre L queda asignado a la lista [1,2,3,4,5].

2. Podemos cambiar, en una lista L existente, el elemento con índice i reasignando su valor mediante L[i]=\dots. Por ejemplo L[1]=7 transforma la lista L del apartado anterior en $[1, 7, 3, 4, 5]$.

3. Usaremos la notación *slice* para el acceso a sublistas. Si L es una lista que ya ha sido definida,

   - seleccionamos el elemento de índice `j` (o lugar `j+1`) en L con L[j];[2]
   - seleccionamos con L[j:k:d], la sublista formada por los elementos de índices $j, j + d, j + 2d, ..., j + \ell d < k$. Si no se especifica el tercer argumento, los *saltos* son de uno en uno. La ausencia de alguno de los dos primeros argumentos, manteniendo los dos puntos de separación, indica que su valor es el más extremo (primero o último).

```
L=[0,1,2,3,4,5,6,7,8,9,10,11,12]
print (L[3],  L[1:5:2],   L[6:10],   L[11:],   L[:4],   L[::3])
```

```
(3,[1,  3],[6,  7,  8,  9],[11,  12],[0,  1,  2,  3],[0,  3,  6,  9,  12])
```

4. El signo + concatena listas

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

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

5. Se abrevia la concatenación k veces de una misma lista L con k∗L.

3.2.1. Métodos y funciones con listas

1. Al aplicar un método a un contenedor tipo lista, este cambia. Esta es una característica distintiva de una lista: es un contenedor de datos *mutable*.

   - El constructor **list**() nos permite crear una copia exacta de una lista.

```
L=[1,2,3,4,5]
LL=list(L)
L.append(12)
L,LL
```

```
([1,  2,  3,  4,  5,  12],  [1,  2,  3,  4,  5])
```

   - Por ejemplo, L.**sort**() cambia la lista L de números enteros por la lista ordenada pero mantiene el nombre L. Si más adelante, en nuestro programa, necesitamos usar otra vez la lista L original es preciso generar una copia mediante LL=**list**(L), o bien primero generar la copia LL y ordenar LL manteniendo L con su valor original.
     En particular, una asignación como LL = L.**sort**() no crea la lista LL aunque sí ordena L.

[2] Esto se debe a que `Sage`, y `Python`, llaman elemento `0` de las listas al primero.

- En general, si se *alimenta* de cualquier otro contenedor iterable, se crea una lista con sus elementos. El siguiente ejemplo, que usa una cadena de caracteres, se entenderá mejor al leer la siguiente sección. Lo incluimos para ilustrar el uso de otro tipo de datos iterable:

```
Letras=list('ABCDE')
Letras
```

```
['A', 'B', 'C', 'D', 'E']
```

- La función **srange**() permite crear listas de enteros de Sage. En python se usa la función **range**(), que devuelve enteros de Python, a los que no son aplicables los métodos de Sage.
  - **srange**(j,k,d): devuelve los números entre j (inclusive) y k (exclusive), pero contando de d en d elementos. A pesar de que el uso más extendido es con números enteros, los números j, k y d pueden ser enteros o no.

  Abreviaturas:
  - **srange**(k): devuelve **srange**(0,k,1). Si, en particular, k es un natural, devuelve los naturales entre 0 (inclusive) y k (exclusive); y si k es negativo, devuelve una lista vacía.
  - **srange**(j,k): devuelve la lista **srange**(j,k,1). Si, en particular, j y k son enteros, devuelve los enteros entre j (inclusive) hasta el anterior a k.
  - [a..b] es equivalente a **srange**(a,b+1).
- El método .**reverse**() cambia la lista por la resultante de reordenar sus elementos dándoles completamente la vuelta.

```
L=[2,3,5,7,11]
LL=list(L)
L.reverse()
L, LL
```

```
([11, 7, 5, 3, 2], [2, 3, 5, 7, 11])
```

- Con .**append**() se añade un nuevo elemento, y solo uno, al final de la lista. Como método, cambia la lista. Así, L.**append**(5) equivale a L=L+[5] (o L+=[5]). Para ampliar con más de un elemento, existe el método .**extend**().
- Si los elementos de la lista son comparables, se pueden ordenar, de menor a mayor, con el método .**sort**().
- **sum**(`<lista_numérica>`) [3] calcula la suma de los elementos de la lista, que deben ser números.
- Listamos, para finalizar, métodos referentes a un elemento concreto y una lista: el número de apariciones se averigua con el método .**count**(); información sobre la primera aparición la ofrece .**index**(); se puede borrar con métodos como .**pop**() o .**remove**(); y se añade en una posición determinada con .**insert**(). La ayuda interactiva, (tras el nombre del método y `tabulador`, nos amplia esta información.

[3] Esta notación, que usaremos frecuentemente, indica el tipo de objetos a los que podemos aplicar la función **sum**(), en este caso a los objetos de tipo `lista de números` y solo a tales listas. Cuando la aplicamos, por ejemplo **sum**([1,2,3]), no escribimos los angulitos sino solo una lista, [1,2,3] o el nombre, por ejemplo L, de una lista de números.

3.2.2. Otras listas

Muchos métodos y funciones de sage devuelven listas como producto final de su evaluación, o, al menos, información fácilmente utilizable para generar una lista. Terminamos esta sección con varios de estos ejemplos.

- *List comprehension.* Más que un método o función, es una manera abreviada de creación de listas. Por su especial sintaxis, en ocasiones es difícil de entender en una primera lectura. En general, se utiliza para generar listas aplicando cierta transformación sobre otra lista, o con los elementos de otro contenedor iterable. Así, por ejemplo

```
[ j^2 for j in [1..5]  ]
```

tomará, uno a uno, los elementos de la lista [1,2,3,4,5] y generará la lista de sus cuadrados:

```
[1, 4, 9, 16, 25]
```

Se puede, también, filtrar los elementos del contenedor original. De nuevo nos adelantamos a un material posterior, aunque es fácil entender el siguiente ejemplo.

```
[ j^2 for j in [1..20]  if j.is_prime() ]
```

```
[4, 9, 25, 49, 121, 169, 289, 361]
```

Volveremos sobre este asunto en la subsección 4.2.3.

- Dado un entero de Sage, k, el método k.**digits**() devuelve una lista cuyos elementos son sus dígitos. Por ejemplo, 123.**digits**() devuelve la lista [3,2,1] (atención al orden). Por defecto los dígitos se consideran en la base 10, y una lista *sin ceros a la izquierda*. Otra base de numeración se indica como primer argumento, y un número de dígitos fijo, m, se indica asignando al parámetro padto dicho valor: padto=m.

```
L=123.digits(padto=5)
LL=111.digits(2,padto=7)
L.reverse(),LL.reverse()
L, LL
```

```
([0, 0, 1, 2, 3], [0, 1, 1, 0, 1, 1, 1, 1])
```

- Los enteros de Sage se pueden factorizar en primos con el método .**factor**(). El resultado no se muestra como una lista, pero en ocasiones es útil la lista de factores y potencias: el constructor **list**() es apropiado para este objetivo.

```
k=3888
print k.factor()
L=list(k.factor())
L
```

```
2^4 * 3^5
[(2, 4), (3, 5)]
```

3.2.3. Ejercicios con listas

Ejercicio 1. *Dada la lista L=[3,5,6,8,10,12], se pide:*
(a) *averiguar la posición del número* 8*;*
(b) *cambiar el valor* 8 *por* 9*, sin reescribir toda la lista;*
(c) *intercambiar* 5 *y* 9*;*
(d) *intercambiar cada valor por el que ocupa su posición* simétrica*, es decir, el primero con el último, el segundo con el penúltimo, ...*
(e) *crear la lista resultante de concatenar a la original, el resultado del apartado anterior.*

Ejercicio 2. *La orden* ***prime_range****(100,1001) genera la lista de los números primos entre* 100 *y* 1000.
Se pide, siendo `primos`=***prime_range****(100,1001):*
(a) *averiguar el primo que ocupa la posición central en* `primos`*;*
(b) *averiguar la posición, en* `primos`*, de* 331 *y* 631*;*
(c) *extraer, conociendo las posiciones anteriores, la sublista de primos entre* 331 *y* 631 *(ambos incluidos);*
(d) *extraer una sublista de primos que olvide los dos centrales de cada cuatro;*
(e) *(**) extraer una sublista de primos que olvide el tercero de cada tres.*

3.3. Tuplas

Las "tuplas" son similares a las listas, la mayor diferencia es que no podemos asignar nuevo valor a los elementos de una tupla. Decimos que las tuplas son "inmutables". Las listas no lo son porque podemos reasignar el valor de una entrada de la lista, L[1] = 7, y también un método puede modificar el valor de una lista, por ejemplo, L.**reverse**() cambia el valor de L.

- Definimos una tupla mediante t = (1,2,3,4,5,6).
- Nos referimos a los elementos de una tupla con la notación *slice*, de la misma forma que a los de una lista, así: t[0] es el elemento en la primera posición de la tupla, 1; t[1:5] es la tupla (2,3,4,5); etc.
- Con el símbolo de la suma se concatenan tuplas: t + (1,5) es la tupla (1,2,3,4,5,6,1,5). Y con el del producto, se repite: 2∗(1,5) devuelve la tupla (1,5,1,5).
- A diferencia de las listas, no se puede cambiar un elemento de una tupla intentando reasignar su valor:

```
t=(1,2,3,4,5,6,7)
t[1]=3
```

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

Si queremos cambiar algún elemento de una tupla, tenemos que reasignar toda la tupla.

En ocasiones, esta será una tarea ingrata y preferiremos un camino más directo. Pensemos en una tupla con algunas decenas de elementos. Para el intento anterior, podríamos hacerlo con una sintaxis algo enrevesada:

```
t1,t2=t[:1],t[2:]    ##cortamos la tupla en dos,
##evitando el elemento a sustituir
t1+(3,)+t2  ## concatenamos intercalando la tupla (3,)
```

```
(1, 3, 3, 4, 5, 6, 7)
```

Es fácil comprobar que (3) no se interpreta como una tupla, pero (3,) sí.

Otra opción es cambiar a un contenedor tipo lista, más manipulable, realizar los cambios y asignar el resultado a una tupla. Los constructores **list**() y **tuple**() son idóneos para este camino que se deja como ejercicio al lector.

- La función **len**(), aplicada a una tupla, nos devuelve su tamaño.
- Los métodos **.index**() y **.count**() se aplican de manera análoga al caso de listas.

3.3.1. Ejemplos con tuplas

Ejemplo 1. La composición **list**(**factor**()), aplicada a un natural, devuelve una lista de pares

```
factores=list(factor(600)); factores
```

```
[(2, 3), (3, 1), (5, 2)]
```

Se puede iterar sobre esta lista para extraer los pares

```
for k in factores: print k,
```

```
(2, 3) (3, 1) (5, 2)
```

o *desempaquetar* cada par, es decir extraer simultáneamente cada miembro de la pareja al iterar la lista

```
for a,b in factores: print a^b,
```

```
8 3 25
```

Ejemplo 2. La función **xgcd**() aplicada a dos enteros, a,b, devuelve una tupla con 3 valores: (d,u,v). El primero es el máximo común divisor[4], y se verifica la *identidad de Bézout*: $d = a \cdot u + b \cdot v$.

```
a,b=200,120
d,u,v=xgcd(a,b)
d,u,v,d-(a*u+b*v)
```

```
(40, -1, 2, 0)
```

En este curso usaremos mucho más listas que tuplas, ya que, esencialmente, realizan la misma función y las listas son más flexibles.

3.4. Cadenas de caracteres

Una cadena de caracteres es otro contenedor de datos inmutable, como las tuplas. Lo que contiene son caracteres de un alfabeto, por ejemplo el nuestro (*alfabeto latino*).

Dado que la CRIPTOGRAFÍA trata del enmascaramiento sistemático de un texto hasta hacerlo ilegible, pero recuperable conociendo la clave, es claro que en el capítulo **??**, dedicado a ella usaremos sistemáticamente las funciones y métodos que se aplican a cadenas de caracteres.

- Una `cadena de caracteres` se forma concatenando caracteres de un alfabeto, habitualmente formado por letras, dígitos, y otros símbolos como los de puntuación.
- En el siguiente ejemplo se asigna a C una cadena

[4] *gcd* son las siglas de *greatest common divisor*. La función **gcd**() devuelve, sin más, el máximo común divisor; **xgcd**() es una versión e**x**tendida.

```
C='Esta es nuestra casa.'
```

Las comillas delimitan el inicio y el final de la cadena. En este caso es suficiente con comillas simples. Si se quieren incluir comillas simples entre los caracteres, se usarán comillas dobles[5] o triples (tres sencillas consecutivas) para delimitar; forzosamente triples si han de incluir simples y dobles.

- **str**(k) convierte el entero k en una cadena de caracteres, de forma que **str**(123) da como resultado la cadena `'123'`.
- Las operaciones + (concatenación) y ∗ (repetición) se utilizan y comportan como en el caso de listas.
- **len**(<cadena>) es el número de caracteres de la cadena.
- Para el acceso a subcadenas, se utiliza la notación *slice*.

```
frase='Recortando letras de una frase'
print (frase[3],frase[1:5:2],frase[6:10],frase[11:],frase[:4],frase[::3])
```

```
('o', 'eo', 'ando', 'letras de una frase', 'Reco', 'Roaoeadu a')
```

- <cadena>.**count**(<cadena1>) devuelve el número de veces que la cadena1 aparece "textualmente" dentro de la cadena.

```
frase='Contando letras de una frase'
frase.count('e')
```

```
3
```

- <cadena>.**index**(<cadena1>) devuelve el lugar en que la cadena1 aparece POR PRIMERA VEZ dentro de la cadena.

```
frase='Buscando letras en una frase'
frase.index('a')
```

```
4
```

- El método .**split**() trocea una cadena de caracteres, devolviendo una lista con las subcadenas resultantes. Si no hay argumento, se utiliza, por defecto, el espacio en blanco para trocear, eliminándose de la lista de subcadenas resultante. Si se indica al método una subcadena, se utiliza esta para recortar.

```
frase='Una␣frase␣para␣trocear.
Por␣defecto,␣se␣recorta␣por␣el␣espacio␣en␣blanco.'
print frase.split('␣')==frase.split()
palabras=frase.split('␣')
subfrases=frase.split('.')
palabras; subfrases
```

```
True
['Una', 'frase', 'para', 'trocear.', 'Por', 'defecto,', 'se',
'recorta',
'por', 'el', 'espacio', 'en', 'blanco.']
['Una frase para trocear', ' Por defecto, se recorta por el espacio en
blanco', '']
```

[5] Comillas dobles no son dos simples consecutivas, sino las que suelen estar en la tecla 2 de un teclado.

- El efecto inverso al uso de .**split**() viene dado por el método .**join**(), que actúa sobre cadenas y espera un contenedor de cadenas; o la función **join**(), que actúa sobre un contenedor de cadenas. Un ejemplo, conectado con el anterior, nos sirve para entender el uso

```
'-'.join(palabras); join(palabras,'|')
```

```
'Una-frase-para-trocear.-Por-defecto,
-se-recorta-por-el-espacio-en-blanco.'
'Una|frase|para|trocear.|Por|defecto,
|se|recorta|por|el|espacio|en|blanco.'
```

De no especificarse el segundo argumento de la función **join**(), se utilizará un espacio en blanco simple como *pegamento*. Para concatenar cadenas, aparte del signo +, podemos usar **join** con la cadena *vacía* `''`

```
palabras[0]+palabras[1]; ''.join(palabras); join(palabras,'')
```

```
'Unafrase'
'Unafraseparatrocear.Pordefecto,serecortaporelespacioenblanco.'
'Unafraseparatrocear.Pordefecto,serecortaporelespacioenblanco.'
```

- El método .**find**() devuelve el primer índice en que aparece una subcadena en la cadena a que se aplica; si no aparece, devuelve −1. Se puede delimitar el inicio de la búsqueda, o el inicio y el final (de omitirse este, es el último índice de la cadena). En la cadena `frase` anterior aparece la subcadena `'or'` en 3 ocasiones. El siguiente código nos encuentra dónde:

```
L=len(frase)
a=frase.find('or')
b=frase.find('or',a+1,-2)
c=frase.find('or',b+1)
a, b, c, frase.find('or',c+1)
```

```
(25, 43, 49, -1)
```

Sugerencia: Pulsar el `tabulador` tras escribir un punto detrás del nombre de una variable que contenga una cadena de caracteres, `<cadena>. + tabulador`, para encontrar los métodos aplicables a cadenas de caracteres.

3.4.1. Ejercicios

Ejercicio 3. *Considérese el número* 1000! *(**factorial**(1000)):*

- *¿en cuántos ceros acaba?*
- *¿Se encuentra el número* 666 *entre sus subcadenas? En caso afirmativo, localizar (encontrar los índices de) todas las apariciones.*
- *Encontrar la subcadena más larga de doses consecutivos. Mostrarla con los dos dígitos que la rodean.*

Ejercicio 4. *Si se aplica la función **sum**() a una lista numérica, nos devuelve la suma de todos sus elementos. En particular, la composición **sum**(k.**digits**()), para k un variable entera, nos devuelve la suma de sus dígitos (en base* 10*).*

- *Calcular, con la composición **sum**(k.**digits**()), la suma de los dígitos del número k=**factorial**(1000).*
- *Calcular la misma suma sin utilizar el método .**digits**().*

  Sugerencia: *considerar la cadena digitos=`'0123456789'`, en la que digitos[0]=`'0'`, digitos[1]=`'1'`, ..., digitos[9]=`'9'`. Sumar los elementos de la lista*

$$\left[j * (\text{ veces que aparece } j \text{ en } 1000!) : \text{ con } j = 1, 2, 3, 4, 5, 6, 7, 8, 9\right].$$

3.5. Conjuntos

Los conjuntos en SAGE se comportan y tienen, esencialmente, las mismas operaciones que los conjuntos en Matemáticas. La mayor ventaja que tienen sobre las listas es que, por su propia definición como conjunto, suprimen las repeticiones de elementos. Así [1, 2, 1] tiene 3 elementos (y están indexados), pero {1, 2, 1} es lo mismo que {1,2} y tiene solo 2 elementos.

Así podemos usar conjuntos para suprimir repeticiones en una lista: transformamos la lista en conjunto y el conjunto resultante en lista.

- Se utilizan las llaves para delimitar un conjunto:

```
A = {1,2,1}
type(A), A
```

```
(<type 'set'>, set([1, 2]))
```

- Aunque los conjuntos son mutables, sus elementos han de ser objetos inmutables. Así, no podemos crear conjuntos con conjuntos o listas entre sus elementos.

```
A={{1,2},{3}}
```

```
TypeError: unhashable type: 'set'
```

```
A={[1,2]}
```

```
TypeError: unhashable type: 'list'
```

- Otra manera de crear conjuntos es con el constructor **set**(), que toma los elementos de cualquier contenedor

```
l,t,s=[1,2,1],(3,2,3),'Hola gente'
set(l); set(t); set(s)
```

```
set([1, 2])
set([2, 3])
set(['a', ' ', 'e', 'g', 'H', 'l', 'o', 'n', 't'])
```

`Nota:` La orden A=**set**() crea un conjunto sin elementos, el *conjunto vacío*.

- Los elementos de un conjunto no están ordenados ni indexados: si A es un conjunto A[0] no es nada.
- La instrucción 1 **in** A devuelve **True** si el 1 es uno de los elementos del conjunto A.
- Si A y B son dos conjuntos, la unión de dos conjuntos se obtiene con A | B, la intersección con A & B y la diferencia con A−B. La comparación A<=B devuelve **True** si todos los elementos de A están en B.
- Como para los contenederos ya estudiados, **len**(A) es el número de elementos de un conjunto.
- Añadimos un elemento, por ejemplo el 1, a un conjunto mediante A.**add**(1), y lo suprimimos con A.**remove**(1). Si se quieren añadir más elementos, contenidos en cualquier otro contenedor, basta aplicar el método .**update**()

```
A={-1,-2}
l=srange(100)
A.update(l)
len(A)
```

```
102
```

Sugerencia: Averiguar el uso de otros métodos como: .**pop**(), .**union**(), .**intersection**(), ...

3.5.1. Ejercicios

Ejercicio 5. *A partir de la cadena de caracteres*

```
texto='Omnes homines, qui sese student praestare ceteris animalibus,
summa ope niti decet ne uitam silentio transeant ueluti pecora, quae
natura prona atque uentri oboedientia finxit.'
```

extraer la lista de los caracteres del alfabeto utilizados, sin repeticiones, sin distinguir mayúsculas de minúsculas y ordenada alfabéticamente.
Sugerencia: *La composición* ***list****(****set****()) aplicada a una lista, genera una lista con los elementos de la original sin repeticiones, ¿por qué?*

Ejercicio 6. Máximo común divisor.
Sin utilizar los métodos ***.divisors****() ni la función* ***max****(), elabora código que, a partir de dos números* a *y* b*, calcule:*

- *El conjunto* $\mathrm{Div}_a = \{k \in \mathbb{N} : k|a\}$
- *El conjunto* $\mathrm{Div}_b = \{k \in \mathbb{N} : k|b\}$
- *El conjunto* $\mathrm{Div}_{a,b} = \{k \in \mathbb{N} : k|a \text{ y } k|b\}$.

Una vez se tenga el conjunto de los divisores comunes, *encontrar el mayor de ellos.*

Ejercicio 7. Mínimo común múltiplo.
El mínimo común múltiplo, m*, de dos números,* a *y* b*, es menor o igual que su producto:* $m \leq a \cdot b$*. Sin utilizar la función* ***min****(), elabora código que, a partir de dos números* a *y* b*, calcule:*

- *El conjunto* $\mathrm{Mult}_{a(b)} = \{k \in \mathbb{N} : a|k \text{ y } k < a \cdot b\}$
- *El conjunto* $\mathrm{Mult}_{b(a)} = \{k \in \mathbb{N} : b|k \text{ y } k < a \cdot b\}$
- *El conjunto* $\mathrm{Mult}_{a,b} = \{k \in \mathbb{N} : a|k\,,\ b|k \text{ y } k < a \cdot b\}$

Una vez se tenga este subconjunto de los múltiplos comunes, *encontrar el menor de ellos.*

Ejercicio 8. *Dada una lista de números enteros, construye un conjunto con los factores primos de todos los números de la lista.*
Indicación: Usa ***list****(k.****factor****()) para obtener los factores primos.*

Estos ejercicios, y los de las páginas 42 y 45, son importantes porque vamos a estar usando manipulaciones de estructuras de datos, sobre todo listas y cadenas de caracteres, durante todo el curso. Soluciones en la hoja de Sage `31-ESTRD-ejercicios-sol.ipynb`.

3.6. Diccionarios

Los elementos de una lista L están indexados por los enteros entre 0 y **len**(L)−1 y podemos localizar cualquier elemento si conocemos su orden en la lista. Los diccionarios son parecidos a las listas en que los elementos están indexados, pero el conjunto de índices es arbitrario y adaptado al problema que queremos resolver.

- Definimos un diccionario mediante

  ```
  diccionario = {clave1:valor1, clave2:valor2, ...}
  ```

  Las claves son los identificadores de las entradas del diccionario y, por tanto, han de ser todas diferentes. En general, la información que contiene el diccionario reside en los pares clave:valor. Por ejemplo, un diccionario puede contener la información `usuario:contraseña` para cada uno de los usuarios de una red de ordenadores.

- Una vez creado el diccionario, recuperamos la información sobre el valor correspondiente a una clave con la instrucción diccionario[clave].

```
granja={'vacas':3, 'gallinas':10, 'ovejas':112}
granja['ovejas']
```

```
112
```

- Definimos una nueva entrada, o cambiamos su valor, mediante

  diccionario[clave]=valor,

  y suprimimos una entrada con **del** diccionario[clave].

```
granja['vacas']+=1 ## Se adquieren una vaca...
granja['caballos']=2 ## ... y dos caballos.
del granja['gallinas'] ## Se venden todas las gallinas.
show(granja)
```

```
{vacas : 4, ovejas : 112, caballos : 2}
```

- Podemos crear un diccionario vacío con el constructor **dict**(). Si pasamos al constructor una *lista de pares*, este creará un diccionario con claves los primeros objetos de cada par, y valores los segundos.

```
Cuadrados_y_cubos=dict([(j,(j^2,j^3)) for j in [1..6]])
show(Cuadrados_y_cubos)
```

```
{1 : (1, 1) , 2 : (4, 8) , 3 : (9, 27) , 4 : (16, 64) , 5 : (25, 125) , 6 : (36, 216)}
```

  `Nota:` Un buen constructor de listas de pares es **zip**(). Si se tienen dos listas, L1 y L2, la lista **zip**(L1,L2) está formada por las parejas (L1[j],L2[j]) para j=0,1,..., el menor índice final posible:

```
L1, L2=[1..15],  [10..16]
DD=dict(zip(L1,L2))
print DD
```

```
{1: 10,  2: 11,  3: 12,  4: 13,  5: 14,  6: 15,  7: 16}
```

- Los métodos .**keys**() y .**values**() producen listas con las claves y los valores, respectivamente, en el diccionario. El método .**items**() devuelve la lista de pares (clave,valor).

- La instrucción x **in** diccionario devuelve **True** si x es una de las claves del diccionario.

Los diccionarios sirven para ESTRUCTURAR LA INFORMACIÓN, haciéndola mucho más accesible. En el capítulo **??**, dedicado a la criptografía, veremos un ejemplo muy próximo a la noción de "diccionario como un libro con palabras ordenadas alfabéticamente": queremos decidir de forma automática si un texto dado pertenece a un idioma y disponemos de un archivo que contiene unas 120.000 palabras en ese idioma, una en cada línea.

Podríamos crear una lista con entradas las palabras del archivo, pero es mucho más eficiente estructurar la información como un diccionario. Buscar una palabra en una tal lista requiere recorrer la lista hasta que la encontremos, y sería equivalente a buscar una palabra en un diccionario de papel comparándola con cada palabra del diccionario empezando por la primera.

En lugar de usar una lista, creamos un diccionario con claves tripletas de caracteres y cada tripleta toma como valor la lista de todas las palabras en el

archivo que comienzan exactamente por esa tripleta. De esta forma conseguimos que los valores en el diccionario sean listas de longitud moderada, y resulta mucho más fácil averiguar si una cierta palabra está o no en el diccionario.

Puedes consultar la hoja de Sage `41-PROGR-diccionario.ipynb` para ver una implementación de esta idea, aunque ES NECESARIO HABER VISTO EL CAPÍTULO SIGUIENTE, dedicado a la programación, para entenderla.

Además, se discute en esa hoja la ventaja que se obtiene al usar diccionarios, siempre que las longitudes de los valores estén equilibradas, frente a listas. Este tipo de discusión se repetirá a lo largo del curso, ya que repetidas veces deberemos comparar, mediante *experimentos bien elegidos* los méritos o deméritos de diversos métodos para realizar los mismos cálculos.

3.7. Conversiones

En ocasiones necesitamos convertir los datos contenidos en una cierta estructura a otra, debido, esencialmente, a que la nueva estructura admite métodos que se adaptan mejor a las manipulaciones con los datos que pretendemos hacer.

1. TUPLA A LISTA O CONJUNTO: Para una tupla `T`

   ```
   list(T)
   set(T)
   ```

2. LISTA A TUPLA: **tuple**(L).

3. CADENA A LISTA:

   ```
   C = 'abc'
   list(C)
   ```

   ```
   ['a',  'b',  'c']
   ```

4. ¿LISTA A CADENA?:

   ```
   C = 'abc'
   str(list(C))
   ```

   ```
   "['a',  'b',  'c']"
   ```

   En general esto NO es lo que queremos.

5. LISTA A CADENA:

   ```
   C = 'abc'
   join(list(C),sep="")
   ```

   ```
   'abc'
   ```

6. LISTA A CONJUNTO: para una lista L, **set**(L). Suprime repeticiones en la lista.

7. CONJUNTO A LISTA: para un conjunto A, **list**(conjunto).

8. DICCIONARIO A LISTA DE PARES: Para un diccionerio D, D.**items**().

9. LISTA DE PARES A DICCIONARIO: Este pequeño programa, con un bucle **for**, realiza la conversión.

   ```
   def convert_list_dict(L):
           dict = {}
           for item in L:
               dict[item[0]]=item[1]
           return dict
   ```

10. Dadas dos listas, `L1` y `L2`, de la misma longitud podemos formar una lista de pares mediante **zip**(L1,L2), y transformar esta lista en diccionario mediante la funciṕn del apartado anterior.

Capítulo 4

Técnicas de programación

El contenido de este capítulo es fácil de describir: un programa consiste esencialmente en bloques **for**, que repiten un número de veces un grupo de instrucciones, y bloques **if**, que bifurcan la ejecución del código según se cumplan o no determinadas condiciones booleanas: si se cumple una condición ejecutan un bloque de instrucciones y si no se cumple otro distinto.

Se describe la sintaxis de ambos tipos de bloque y se muestra mediante ejemplos sencillos su uso, junto con otros métodos como la recursión y los bloques **while**.

Como se indicó en el prólogo, y nunca comnseguiremos repetirlo un número suficiente de veces, la única forma conocida de aprender a programar es *programando*, y no es algo que se pueda asimilar en la última semana antes del examen final.

Por otra parte, la programación en algún lenguaje de alto nivel forma ya parte importante de la formación que se espera de un graduado en matemáticas, y, les guste mucho o no, es en lo que terminan trabajando muchos de nuestros graduados.

4.1. Funciones

En esta sección, y en otros lugares a lo largo de este texto, la palabra *función* designa un programa que recibe unos argumentos y devuelve, después de algunos cálculos, un resultado. Es claro que una función en este sentido define, si el conjunto de argumentos no es vacío, una *función matemática* del conjunto de todos los posibles argumentos al de todos los posibles resultados. Según el contexto se puede decidir si la palabra *función* se refiere a un programa o a una función matemática.

La sintaxis para definir funciones es:

```
def nombre_funcion(arg_1,arg_2,...,arg_n):
    '''Comentarios sobre la funcion'''
    instruccion_1
    instruccion_2
    etc.
    return res_1,res_2,...,res_m
```

1. El código escrito en *Python* utiliza el sangrado de las líneas para indicar los distintos bloques de código. Otros lenguajes de programación, por ejemplo C, utilizan llaves para obtener el mismo resultado. Esto hace que, en general, sea más fácil leer código escrito en Python.

   No es difícil acostumbrase a "ver" los errores de sangrado, y el intérprete ayuda produciendo un *Indentation Error* cuando intentamos ejecutar código mal sangrado. Además, pinchando con el cursor a la izquierda del

mensaje de error nos muestra el lugar aproximado donde ha encontrado el error mediante un angulito en la línea de debajo.

2. Los nombres `arg_j` y `res_i`, así como el de la función, `nombre_funcion`, son genéricos. Es una buena costumbre el utilizar nombres que sean lo más descriptivos posible. Así por ejemplo, si un resultado va a ser el cociente de una división es bueno utilizar como nombre `cociente` o `coct` en lugar de `res_3`.

3. Los comentarios son opcionales, pero ayudan a entender mejor el código cuando se relee pasado un tiempo, y también cuando lo leen personas diferentes a su autor. Es por tanto una buena práctica, aunque por pereza muchas veces no los escribimos.

4. Las instrucciones que forman el programa indican cómo calcular `res_i` utilizando los argumentos de la función `arg_j` y otras funciones ya definidas dentro de Sage o por nosotros.

   En general, debe haber líneas en el código el tipo `res_i=...` que definan todos los valores que debe devolver `return`. Recuérdese (ver página 17) que una línea como estas asigna un valor, lo que está a la derecha del igual, a una *variable* de nombre `res_i`.

   Si la función debe devolver un valor que no ha sido calculado se produce necesariamente un error.

5. Dentro de otra función, por ejemplo `nf2(arg_1,arg_2,...,arg_k)` podemos *llamar* a la función `nombre_funcion` mediante una línea como

   ```
   res_1,res_2,...,res_m=nombre_funcion(arg_1,arg_2,...,arg_n)
   ```

   convenientemente sangrada. En las líneas anteriores, incluyendo la posibilidad de que algunos de esos valores sean argumentos de la propia función `nf2`, debe estar definido el valor de todos los argumentos de la función, y si no es así se producirá un error al interpretar el código.

   Esta línea, si llega a ejecutarse sin errores, asigna un valor a todas las variables `res_i`.

6. De esta manera podemos dividir nuestra tarea en varias tareas simples, definidas cada una de ellas en una función propia, y ejecutar la tarea principal en una función que va *llamando* a las funciones auxiliares.

   Se llama a esta técnica *programación estructurada*, y la usaremos ampliamente a lo largo del curso. Usándola es mucho más fácil escribir, leer o depurar el código.

4.2. Control del flujo

4.2.1. Bucles `for`

1. Un bucle **for** es una manera de repetir ("iterar") la ejecución de un bloque de código un NÚMERO DE VECES DADO.

2. La sintaxis de un **for** es

   ```
   for <elemento> in <contenedor>:
           instruccion 1
           instruccion 2
           etc ...
   ```

3. Este bucle empieza asignando a <elemento> el valor que ocupa la primera posición[1] en el <contenedor>, y termina para el valor en la última posición, **len**(contenedor)−1. Por tanto, repite el bloque de instrucciones **len**(contenedor) veces. Así, por ejemplo, un bucle con línea de entrada **for** j **in srange**(10): asignará a j los enteros de sage $0, 1, 2, ..., 9$, y repetirá su bloque de instrucciones 10 veces.

4. Las instrucciones del bloque DEBEN ESTAR TODAS ALINEADAS y "sangradas" respecto al **for**.

5. Un bloque de código puede tener un subbloque, por ejemplo una de las instrucciones de un bloque puede ser un bloque `for` con una serie de instrucciones que forman el subbloque. En ese caso, el `for` que determina el subbloque está alineado con todas las instrucciones del bloque `for` inicial, y todas las instrucciones del subbloque están alineadas entre sí y sangradas respecto a las del primer bloque. La estructura sería:

```
for <elemento> in <contenedor>:
        instruccion 1
        instruccion 2
        for <elemento2> in <contenedor2>:
              instruccion 3
              instruccion 4
              ..........
        instruccion m
         ..........
```

   Este bucle doble se ejecuta en la forma natural: para cada valor de <elemento> se ejecutan en orden instruccion 1, instruccion 2, y se entra en el segundo bucle, que se ejecuta completo, antes de poder continuar con la instruccion m y las que vengan a continuación.

   Un ejemplo típico de un bucle doble es el que nos sirve para recorrer los elementos de una matriz $m \times m$:

```
def matriz_hilbert(m):
  A = matrix(QQ,m,m,[0]*m^2)
  for i in srange(m):
        for j in  srange(m):
              A[i,j]  = 1/(i+j+1)
  return A
```

6. En ocasiones, cada valor sobre el que itera el **for** es una variable que se usa explícitamente en las instrucciones del bloque:

```
cuadrados=[] ## iniciamos una lista vacía
for j in srange(20): ## 20 iteraciones
      cuadrados.append(j^2) ## actualizamos la lista cuadrados
cuadrados[1::2]  ## listamos los de los impares
```

```
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

   Pero no es obligatorio, como vemos en el siguiente código[2] que sirve para calcular la suma de los $K = 100\,000$ primeros términos de una progresion aritmética:

[1] En los conjuntos y diccionarios los elementos no están explícitamente ordenados, y el bucle los recorre de acuerdo a la forma en que están almacenados en la memoria de la máquina. Las listas, tuplas y cadenas de caracteres están explícitamente ordenadas y el bucle las recorre en el orden en que las hemos definido.

[2] En ocasiones el rango aparece como **xsrange**() en lugar de **srange**(). La diferencia fundamental es que el segundo rango genera una lista de enteros y luego la recorre, y el primero no genera la lista y va aumentando el valor del contador en cada vuelta. Para rangos grandes la segunda forma de la instrucción genera una lista enorme que puede saturar la memoria RAM de la máquina.

```
a,d,K=3,5,10^5
suma=a
for j in xsrange(K):
    a=a+d
    suma+=a
print suma
```

```
25000550003
```

7. EJEMPLO:

   Programamos mediante un **for** (*iterativamente*) el cálculo del término n-ésimo de la sucesión de Fibonacci, definida mediante $F_0 := 0$, $F_1 := 1$, $F_n := F_{n-1} + F_{n-2}$ para $n \geq 2$:

```
def fibon(m):
    p,q = 1,0
    for j in xsrange(m):
        p,q = q,p+q
    return q
```

   Cuando empezamos el bucle, el par (p, q) vale $(1, 0)$, y en cada vuelta sus valores se sustituyen simultáneamente por $(q, p + q)$. Al final, la función devuelve el último valor calculado, es decir, el valor de q cuando ya el **for** ha dado todas las vueltas que debe. En particular, fibon(0) devuelve 0, ya que **xsrange**(0) es una lista vacía, de manera que no se entra ni una sola vez al bloque de instrucciones.

   El procedimiento es muy eficiente, en el uso de la RAM, porque ni se genera una lista que recorrer ni se almacenan los términos de la sucesión. Como indica la definición de la sucesión de Fibonacci, en cada vuelta del bucle, no necesita acordarse de todos los valores anteriores de q, solo de los dos últimos.

   Sage cuenta con una instrucción **fibonacci**() que realiza el mismo cálculo que la que acabamos de definir. ¿Es más eficiente que la nuestra?

4.2.2. Contadores y acumuladores

Supongamos que queremos calcular la suma de una gran cantidad de números enteros definidos mediante una cierta propiedad, por ejemplo *la suma de los primeros cien mil primos*.

Una posibilidad es generar una lista L que los contenga a todos y después aplicar la función **sum**(L), pero si no nos interesa saber cuáles son los primos sino únicamente su suma es claro que esta forma de calcularla no es muy eficiente: estamos creando una estructura de datos enorme, la lista L, en la memoria de la máquina que en realidad no necesitamos.

En lugar del enfoque anterior podemos usar un ACUMULADOR, que simplemente es una variable que según vamos encontrando enteros que satisfacen el criterio los sumamos al valor del acumulador. Ya hemos visto un ejemplo al sumar los términos de una progresión aritmética un poco más arriba.

El código para la suma de los primeros N primos sería

```
def suma_primos(N):
  suma = 2
  primo = 2
  for j in xsrange(N-1):
      primo = next_prime(primo)
      suma += primo ##Equivale a suma = suma+primo
  return suma
```

y conviene definir otra función (EJERCICIO) que obtenga la suma creando la lista de todos los primos y luego sumando. Podemos registrar el tiempo que dura el cálculo comenzando la línea con **time**, como en **time** suma_primos(10^5), y, entonces, comparar el tiempo que tarda el código con acumulador con el que tarda el código que genera la lista.

La idea de un *contador* es similar: queremos *contar el número de veces que "algo" ocurre*, y en lugar de generar un contenedor C con todos los elementos que cumplen las condiciones requeridas para luego usar **len**(C), podemos usar una variable `cont` que inicialmente vale 0 e incrementa su valor en una unidad cada vez que encontramos un elemento que cumple las condiciones. Usaremos contadores profusamente en el capítulo **??**.

Para implementar un contador podemos usar una estructura como

```
cont = 0
for int in srange(N):
    ....
    if condicion:
        ....
        cont += 1
return cont
```

Para poder escribir un programa así debemos conocer *a priori* el número N, el número de vueltas del bucle **for**, y si no lo conocemos debemos intentar usar un bucle **while**. La `condicion` que aparece en el **if** expresa la definición de los objetos que estamos contando, ya que el contador únicamente se incrementa cuando se cumple la condición.

Es claro que, si es posible, es más eficiente usar contadores y acumuladores que generar grandes estructuras de datos que en muchos casos no necesitamos. Como veremos en la subsección sobre *iteradores*, esa eficiencia, sobre todo en el consumo de RAM, mejora cuando usamos *iteradores* en lugar listas (**xsrange**(**N**) en lugar de **srange**(**N**)).

4.2.3. Otra sintaxis para los bucles

En Python existe otra manera, muy concisa, de ejecutar bucles `for`: la instrucción

```
[f(x) for x in L]
```

produce una lista cuyas entradas son los valores de la función, previamente definida, f(x) obtenidos al recorrer x la lista L. Además se puede filtrar la lista resultante con un **if**:

```
[f(x) for x in L if Condicion]
```

que solo calcula y se queda con los f(x) cuando se cumple la condición booleana Condicion.

Un ejemplo típico sería

```
[m for m in srange(2,10^6) if is_prime(m)]
```

Es fácil traducir esta sintaxis a la estándar, y la única ventaja de ésta es la concisión. Si nos acostumbramos a esta sintaxis concisa corremos el peligro de usarla en situaciones en las que no necesitamos la lista resultante sino únicamente su longitud, su suma u otra característica. En esos casos es mucho más eficiente, como se indicó en la subsección anterior, usar contadores o acumuladores dentro de bucles estándar.

También es posible, como se indica al tratar de los iteradores, convertir esta sintaxis en un *generador* que va produciendo los elementos de la lista sin crear en memoria la lista completa.

4.2.4. Otra más

Otra forma concisa de escribir cierta clase de bucles es mediante **map**(f,L), con f una función cualquiera y L una lista de sus posibles argumentos. El resultado es la lista [f(item) **for** item **in** L], es decir, **map** aplica la función f a cada uno de los elementos de la lista L.

La función f puede ser una función predefinida de Sage a la que llamamos por su nombre (`sin, cos, exp, log, sqrt`, etc.) o una función definida por nosotros mediante

```
def f(x):
    return .....
```

Es claro que el resultado de **map**(f,L) es equivalente al programa

```
def map2(f,L):
    LL = []
    for item in L:
      LL.append(f(item))
    return LL
```

de forma que esencialmente se trata de una sintaxis cómoda para un bucle `for`.

Usaremos esta forma del bucle `for` en el capítulo **??**, dedicado a la criptografía, para modificar textos aplicándoles una función que cambia unos caracteres por otros.

4.2.5. Los bloques if <condicion>:

1. Un **if** sirve para *ramificar la ejecución de un programa*: cada uno de las partes del **if** define un camino alternativo y el programa entra o no según se verifique o no una condición.

2. La sintaxis del **if** es:

```
if <condicion 1>:
        instruccion 1
        instruccion 2
        etc ...
elif <condicion 2>:
        instruccion 3
        instruccion 4
        etc ...
        etc ...
else:
        instruccion 5
        instruccion 6
        etc ...
```

3. Decimos que una *expresión o función es booleana* si cuando se ejecuta devuelve un valor de *verdadero o falso* (`True` o `False`) (ver la subsección 2.3.2). Ejemplos típicos son los operadores de comparación

   a) A == B, es decir, A es idéntica a B que se puede aplicar tanto a números como a estructuras de datos.

   b) A < B, A <= B, A > B, A>=B que se aplica a objetos para los que hay un orden natural[3]

```
[3<2, 'a'<'b', 'casa'>'abeto', set([1,2])>=set([1]) ]
```

```
[False, True, True, True]
```

[3] Averiguar el comportamiento con los números complejos.

c) Muchas funciones predefinidas en Sage son booleanas, y en particular todas las que tienen un nombre de la forma `is_...`, como por ejemplo **is_prime**(m) que devuelve `True` si m es primo y `False` si no lo es. Podemos ver todas las funciones predefinidas cuyo nombre empieza por `is` escribiendo `is` en una celda y pulsando la tecla del tabulador.

4. Las diversas condiciones deben ser *booleanas* y puede haber tantas líneas **elif** como queramos.

5. La ejecución del programa *entra a ejecutar el bloque situado debajo del **if**, **elif** o **else*** la primera vez que se cumple una de las condiciones, consideradas en orden descendente en el código. Cada vez que se ejecuta un **if** únicamente se ejecuta uno de los bloques.

6. No es estrictamente necesario que las condiciones sean *disjuntas*, y si no lo son se aplica lo indicado en el apartado anterior. Sin embargo, es conveniente, porque es más claro cómo se ejecuta todo el bloque **if**, intentar que lo sean.

7. La última parte, el **else** es opcional y lo usamos si queremos indicar lo que debe hacer el programa en el caso en que no se cumpla ninguna de las condiciones.

4.2.6. Bucles while

1. Los bucles **while** son similares a los **for**, pero los usamos cuando no sabemos, a priori, cuantas vueltas debe dar el bucle, aunque estamos seguros de que acabará.

2. Su sintaxis es:

```
while <condición>:
        instrucción 1
        instrucción 2
        etc ...
        actualización de la condición
```

   No es necesario que la `'actualización de la condición'` esté al final, pero sí es imperativo que aparezca.

3. El bucle se sigue ejecutando mientras la condición, que debe ser una expresión booleana, es cierta (valor de verdad `True`).

4. Un bucle como **while** 2>1: ... es, en principio, un bucle infinito que no puede terminar ni producir un resultado salvo que haya un **break** que lo pare. Cuando, dentro de la ejecución de un bucle, el programa llega a una línea con un **break** el bucle termina y el programa sigue ejecutándose a continuación del bucle.

   Siempre hay que tener mucho cuidado con los bucles **while** infinitos ya que, aparte de no producir ningún resultado, frecuentemente cuelgan la máquina.

5. En la condición debe haber una variable cuyo valor actualizamos, normalmente al final, dentro del bloque de instrucciones. Habitualmente, cada vuelta del bucle nos acerca más al momento en que la condición se hace falsa y el bucle se para.

6. Ejemplos:

   a) *Encontrar los dígitos cuya posición en la expresión decimal del factorial de un entero m (calculado mediante la instrucción **factorial**(m)) es un número de Fibonacci.*

```
def gen_subcadena(m):
    C = str(factorial(m)) #Convertimos el factorial  de m en una cadena
    K = len(C)
    C1 = "" #Contendrá la solución
    j = 1
    while fibonacci(j) < K:
        C1 = C1+C[fibonacci(j)-1] #Añadimos a C1 un dígito cuya
posición es un número de Fibonacci
        j += 1 #Incrementamos j para que el bucle NO sea infinito
    return C1
```

Usar un **while** es muy conveniente porque no sabemos *a priori* cuál va a ser el primer número en la sucesión de Fibonacci que supere K. Podríamos intentar resolver primero el problema (matemático) consistente en calcular, dado un entero K, el mayor número de Fibonacci menor que K.

b) El programa que sigue es EQUIVALENTE al anterior, aunque tiene más líneas, y es igual de eficiente gracias a que usa **xsrange**(): en cuanto fibon(j) supera a K, el programa entra en el **else** y la instrucción **break** para el bucle **for**. Entonces, no importa que inicialmente el bucle esté definido para un j que puede llegar a $K-1$ porque, de hecho, nunca llega y se para el bucle mucho antes. Acerca de la instrucción **xsrange**() puede verse la nota a pie de página en el punto 5 de la subsección 4.2.1, o, en mucho mayor detalle, en la sección 5.5.2.1.

```
def gen_subcadena2(m):
    C = str(factorial(m))
    K = len(C)
    C1 = ""
    for j  in xsrange(1,K):
      Fj = fibonacci(j)
        if  Fj < N:
            C1 = C1+C[Fj-1]
        else:
            break
    return C1
```

c) Lee cuidadosamente los dos programas anteriores, y trata de entender que la respuesta que producen es realmente una solución del problema. Para eso puede ser conveniente hacer que la respuesta contenga más información, por ejemplo, que al lado de cada dígito calculado aparezca, entre paréntesis, el lugar que ocupa en la cadena C.

d) Este ejemplo muestra que la combinación **for**+**if**+**break** puede ser equivalente a **while**. ¿Es siempre así? ¿De qué depende el que sean equivalentes?

e) En ocasiones queremos usar un contador incrementado en un bucle del que no sabemos *a priori* cuantas veces se va a ejecutar. Intentaremos usar un bucle **while** en una estructura como

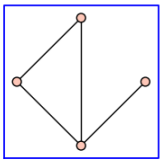
```
cont = 0
while condicion1:
      ....
      if  condicion2:
            ....
            cont += 1
      <actualizamos parametros de condicion1>
return cont
```

La actualización de los parámetros de los que depende la `condicion1` es lo que debe acercarnos al momento en que al no cumplirse la condición el bucle **while** se para, mientras que la `condicion2` expresa la definición de los objetos que estamos contando.

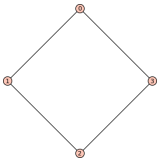
4.3. Recursión

1. La recursión es un procedimiento similar, en muchos aspectos, a la inducción matemática.
2. Decimos que un programa es `recursivo` cuando dentro del bloque de instrucciones del programa hay una llamada al mismo programa con otros argumentos, en general más pequeños.
3. En la definición `recursiva` de una función debe estar previsto un caso inicial que PUEDA PARAR la recursión y tiene que ocurrir que LA RECURSIÓN EFECTIVAMENTE LLEGUE AL CASO INICIAL.
4. Los programas recursivos suelen utilizar una cantidad grande de RAM, ya que, por ejemplo, para calcular una función de n necesitan guardar en memoria la relación entre el valor para n y el valor para $n-1$, la relación entre el valor para $n-1$ y el valor para $n-2$, etc. y no calculan nada hasta que llegan hasta el caso inicial, y entonces empiezan a sustituir para valores crecientes de n.
5. Necesitamos ahora algunas definiciones básicas en la teoría de grafos:

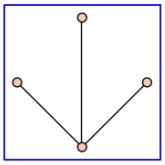
   a) Un *grafo* consiste en un conjunto de *vértices* V y un conjunto de *aristas* $E \subset V(2)$, con $V(2)$ que denota el conjunto de todos los subconjuntos de V que tienen dos elementos. Representamos geométricamente un grafo pintando los vértices como puntos y las aristas como líneas que unen vértices.

   b) Un *ciclo* en un grafo es una sucesión ordenada de vértices $v_0, v_1, v_2, \ldots, v_n$ tales que $v_n = v_0$ y cada vértice v_i, con $i < n$, está unido por una arista a v_{i+1}.

   c) Un *árbol* es un grafo sin ciclos. En ocasiones elegimos un vértice al que llamamos *raíz* y representamos el árbol con la raíz como el punto más bajo y las aristas "hacia arriba" sin cortarse.

   d) Llamamos *hojas* de un árbol con raíz a los vértices, distintos de la raíz, a los que sólo llega un eje.

e) Llamamos *profundidad* de un árbol con raíz al máximo del número de aristas entre la raíz y una hoja.

f) La profundidad y el número de hojas son una medida de lo *frondoso* (=complejo) que es un árbol y por eso nos serán útiles. Dependen de la elección del vértice raíz, pero en nuestro caso tendremos una elección *natural* de raíz.

6. La ejecución de un programa recursivo tiene asociado un *árbol de llamadas* recursivas, con raíz natural la llamada que arranca el programa y vértices cada una de las llamadas a sí mismo, cada una con los valores de los parámetros con los que se llama. Las hojas son todas las llamadas a los casos iniciales de la recursión.

7. Ejemplos:

   a) Fibonacci recursivamente:

```
def fibon2(m):
    if m in [0,1]:
        return m
    else:
        return fibon2(m−1)+fibon2(m−2)
```

   Si comparamos el tiempo, usando `time` al comienzo de la línea en la celda de cálculo, que tarda el programa recursivo con el que tarda el iterativo vemos que el iterativo es muchísimo más eficiente.

```
def fibon(m):
    a,b=1,0
    for j in xrange(m):
        a,b=b,a+b
    return b
```

   La única ventaja del recursivo es que el programa es prácticamente igual a la definición de la sucesión de Fibonacci.

   ¿Qué aspecto tiene el árbol de este programa?

   b) Factorial:

```
def fact(m):
    if m == 0:
        return 1
    else:
        return factorial(m−1)∗m
```

   ¿Qué aspecto tiene el árbol de este programa? ¿Puedes explicar por qué `fibon` recursivo es muy ineficiente mientras que `fact` recursivo es prácticamente tan eficiente como el iterativo?

8. Entonces, los programas recursivos pueden ser muy poco eficientes, aunque no siempre, comparados con un programa similar iterativo, pero tienen la ventaja de que suelen ser más cortos, y (quizá) más fáciles de entender.

9. En ocasiones podemos escribir programas recursivos muy eficientes. Por ejemplo, supongamos un programa que debe manipular una lista de enteros, digamos ordenarla. Un enfoque posible consiste en dividir la lista por la mitad y ordenar las dos mitades (llamada recursiva), y, por último debemos intercalar los elementos de la segunda lista entre los de la primera para obtener la lista original ordenada.

   Esto es factible y muy eficiente, cada vez que se incrementa la profundidad en el árbol de llamadas en una unidad el tamaño de la lista se divide por dos. Podemos ver un ejemplo de esta forma de resolver el problema en la hoja `43-PROGR-ordenacion-listas.ipynb`, y en mayor detalle en la sección 4.5.2.

10. Todo programa recursivo se puede traducir en un programa iterativo y viceversa, pero en ocasiones la traducción es de lectura difícil y el resultado puede ser ineficiente. En este curso no queremos entrar en esas profundidades, que, propiamente, pertenecen a la teoría de la computación.

11. Se pueden ver varios ejemplos más de recursiones en la hoja de Sage `42-PROGR-recursiones.ipynb`.

4.4. Depurando código

1. En primer lugar un código puede tener *errores sintácticos*, es decir, puede ocurrir que el intérprete de Python no lo entienda, no sepa que hacer con él, y produzca mensajes de error:

   a) Los mensajes de error son, en ocasiones, bastante crípticos pero, al menos, suelen mostrar mediante un angulito debajo de una zona del código el lugar aproximado en el que se ha detectado el error.

   b) Los errores más simples, y fáciles de arreglar, son los errores de sangrado ("Indentation Error", "error de alineamiento del código").

   c) También es fácil detectar errores como la falta de *los dos puntos* al final de líneas que deben llevarlos (**def**, **for**, **if**, **while**), paréntesis o corchetes no balanceados (i.e., mismo número de abiertos y cerrados en una expresión), errores en el nombre o el número de argumentos de una función, etc.

   d) En la hoja de Sage `44-PROGR-mensajes-error.ipynb` pueden verse algunos ejemplos de errores sintácticos típicos y de los mensajes de error que generan.

2. Una vez que es posible ejecutar, sin errores, el código todavía cabe la posibilidad de *errores semánticos*, es decir, el código produce resultados erróneos:

   a) Conviene usar NOMBRES DIFERENTES para todos las variables que deben ser distintas en un cálculo, y lo mismo se debe aplicar dentro de una hoja de Sage.

   b) Ocurre frecuentemente que podemos ver si los resultados son correctos directamente: si la función ordena listas el resultado debe estar ordenado, si estamos calculando π el resultado debe comenzar como $3.14\ldots$, etc.

   Siempre hay que tratar de ver si los resultados obtenidos son razonables comparando con lo que esperaríamos *a priori*.

   c) En muchos casos podemos detectar errores semánticos calculando *a mano*, para valores pequeños de sus argumentos, el valor o valores devueltos por la función.

   En ocasiones estamos programando una función que ya existe en Sage, y lo más simple es comparar nuestros resultados con los de Sage.

   d) Si encontramos errores semánticos puede deberse, y suele ser el caso, a que no hemos entendido bien el algoritmo que estamos tratando de implementar. En ese caso es muy conveniente ejecutar *en papel*, y para valores pequeños de sus argumentos, nuestro programa incorrecto y comparar con la ejecución, también *en papel*, del algoritmo.

   e) Una de las teécnicas más útiles para corregir errores semánticos consiste en mostrar el valor de algunas de las variables al paso del programa por algún punto en su ejecución. Habitualmente hacemos esto intercalando una línea como

```
print var_1,var_2,...,var_n
```

en el lugar adecuado del código.

f) Un programa puede tener un sangrado sintácticamente correcto pero semánticamente incorrecto: el sangrado indica el alcance del bloque sobre el que se itera o el de las instrucciones que se ejecutan sólo si se cumple la condición, y es posible que ese alcance de un bloque sea incorrecto.

También puede ser incorrecto el orden de las líneas dentro de un bloque (ver página 64).

g) En criptografía, capítulo **??**, manejamos dos funciones: la primera pasa de un texto legible al correspondiente texto encriptado y la segunda debe recuperar el texto legible a partir del encriptado. Cuando programamos estas dos funciones, para un método criptográfico cualquiera, debemos siempre comprobar que se cumple esta condición.

h) En ocasiones el número de soluciones es conocido *a priori*, de forma que si nuestro programa las calcula es fácil comprobar si las ha encontrado todas. Por ejemplo, supongamos que queremos determinar todas las funciones biyectivas de un conjunto finito con n elementos en sí mismo: sabemos que el número de tales biyecciones es $n!$ de forma que es fácil comprobar, incluso para n grande, si están todas.

i) Es muy conveniente, si es posible, dividir el código de una función entre varias subfunciones que son llamadas desde la principal (programación estructurada). Si hacemos esto es mucho más fácil encontrar y corregir errores semánticos ya que podemos trabajar de manera independiente con cada una de las subfunciones.

3. Por último, un programa puede ser sintácticamente y semánticamente correcto pero muy ineficiente, es decir, puede tardar mucho más tiempo y usar mucha más memoria RAM de la estrictamente necesaria. De cómo mejorar esta situación se tratará en el siguiente capítulo.

4.5. Ejemplos

4.5.1. Órbitas

A lo largo del curso nos vamos a encontrar, en más de una ocasión, con esta situación:

> *Tenemos una función $f : X \to X$ de un conjunto X en sí mismo y un valor inicial $x_0 \in X$ y queremos estudiar la sucesión de iterados de x_0 mediante la función f*
>
> $$o(x_0) := \{x_0, f(x_0), f^2(x_0), f^3(x_0), \dots, f^n(x_0), \dots\} \subset X$$
>
> *La notación $f^n(x_0)$ indica que hemos aplicado sucesivamente n veces la función f a x_0, es decir,*
>
> $$f^n(x_0) := f(f(f(\dots)))(x_0),$$
>
> *y la función f aparece n veces.*

Cuando queremos pensar geométricamente llamamos al conjunto $o(x_0)$ de todos los iterados de x_0 la **órbita** de x_0 mediante f. Otras veces pensamos de "manera física" y entonces el exponente n es el tiempo variando de forma discreta, es decir a saltos, la función f representa la evolución temporal de un proceso físico, mientras que los elementos del conjunto X son los "estados posibles" del sistema que estamos estudiando.

Por poner un ejemplo concreto, un punto (x, v) en $X = \mathbb{R} \times \mathbb{R}$ podría representar la posición y velocidad de una partícula que se mueve en una única dimensión espacial, y $f(x, v) =: (x_1, v_1)$ sería el punto representando la posición y velocidad de la partícula, calculada de acuerdo a las leyes de Newton, después de un segundo. Para cada estado inicial (x_0, v_0) los iterados de la función f son instantáneas, tomadas de segundo en segundo, del estado de movimiento de la partícula.

Cuando trabajamos con el ordenador las órbitas deben ser finitas, lo que está asegurado si el conjunto X es finito. En Si X es finito aparece necesariamente un punto $x_{i_0} = f^{i_0}(x_0)$ de la órbita de x_0 al que se vuelve cuando seguimos iterando, es decir, tal que $x_{i_0} = f^{i_0}(x_0) = f^{i_1}(x_0) = x_{i_1}$. Decimos que se ha producido un **ciclo**, y cuando X es finito todas las órbitas terminan en un ciclo, que puede consitir en un único punto fijo por f ($f(x) = x$).

Observa que el ciclo no contiene necesariamente al punto inicial x_0, sin embargo

Ejercicio 1. *Demuestra que si X es finito y f es biyectiva, el conjunto X es la unión disjunta de ciclos de f.*

Frecuentemente queremos calcular la órbita de un elemento $x_0 \in X$ o bien su longitud, i.e. el número de sus elementos distintos, o incluso los ciclos de una cierta función f.

Entonces vamos a programar una función de SAGE genérica que luego podremos aplicar en casos particulares.

1. Debemos definir la función f que vamos a iterar:

```
def f(?):
    return ------------
```

   Hay que precisar el argumento (argumentos) de la función y los valores que asignamos a cada elección de los argumentos.

2. La función que calcula la órbita tendrá un bucle `while`, ya que no sabemos *cuanto tenemos que iterar* hasta volver a un punto por el que ya hemos pasado.

```
def orbita(ini,f):
    L = []
    while not(ini in L):
        L.append(ini)
        ini = f(ini)  #Actualiza el valor de ini
    return L
```

3. Esta función nos devuelve una lista con primer elemento **ini**, el x_0 de la órbita, y último elemento x_n con la propiedad de que $f(x_n)$ ya está en la lista y eso no ocurre para un n más pequeño.

   Si sólo nos interesa la longitud de la órbita, ¿cómo hay que modificar la función?

4. Una generalización del programa anterior trataría de parar el bucle `while` no cuando se cerrara un ciclo sino cuando dejara de cumplirse una condición booleana cualquiera. En primer lugar, deberíamos definir una función

```
def condparada(??):
    if -----:
        return True
    else:
        return False
```

con argumento (argumentos) y condición para que devuelva `True` adecuados a nuestro problema. Usando ésta, y modificando adecuadamente `orbita`, podemos definir otra, por ejemplo **iterahasta**(**ini**,f,condparada), que devuelva la lista de todos los valores que se obtienen iterando f a partir de **ini** hasta que se cumple la condición de parada.

5. ¿Qué pasa si cambiamos, en el programa del apartado 2) el orden de las dos líneas a continuación del **while**? ¿Por qué?

Ejercicio 2. *Estudia las órbitas de la función $f(n) := n^2$ en el conjunto finito de clases de restos módulo m, para diversos valores de m primos y compuestos. Trata de enunciar principios generales sobre la estructura de las órbitas, su longitud, etc. y de demostrar tus afirmaciones.*

4.5.1.1. Collatz

Comenzamos definiendo una función $f(n)$ de un entero positivo n mediante

$$\begin{cases} n/2 \text{ si } n \text{ es par} \\ 3 \cdot n + 1 \text{ si } n \text{ es impar} \end{cases}$$

El problema de Collatz consiste en probar que para todo entero positivo n la órbita de n contiene al 1. Como el 1 produce un ciclo

$$1, f(1) = 4, f(4) = 2, f(2) = 1,$$

si fuera verdad querría decir que todas las órbitas "mueren" en ese ciclo y no antes.

A pesar de que tiene una apariencia bastante inocente, NO SE SABE si la respuesta al problema de Collatz es afirmativa o negativa.

Puedes ver un par de artículos sobre el problema de Collatz en este enlace y en este otro. El segundo es más elemental, y, por tanto, un poco más asequible. En cualquier caso se trata de un problema difícil que se ha intentado atacar con una gran variedad de métodos y, hasta ahora, sin resultado.

Ejercicio 3.

1. *Escribe un programa para comprobar si el problema tiene solución afirmativa para los enteros positivos n menores que un entero n_0 dado.*
2. *Escribe un programa que, dado un entero n_0, produzca una lista que tenga en la posición n, $n \leq n_0$, el mínimo entero N tal que $f^N(n) = 1$.*
3. *¿Por qué crees que el problema de Collatz es tan difícil?*

4.5.2. Ordenación

Ordenar una lista de enteros desordenados es una de las operaciones básicas que se pueden hacer con un ordenador, y, cuando la lista es enorme, puede exigir demasiado tiempo de cálculo. En este ejercicio hay que programar tres de los métodos existentes para ordenar listas, vamos a ordenarlas en orden creciente, y compararemos los resultados

1. PRIMER ALGORITMO (INSERTION SORT):

   La forma más sencilla, pero no la más eficiente, de ordenar una lista consiste en comparar cada par de elementos e invertirlos si el mayor aparece antes en la lista.

   Ejercicio 4.

a) *Define una función de Sage* `intercambiar(L,i,j)`*, con argumentos una lista y un par de enteros, y tal que devuelva la lista que se obtiene intercambiando el elemento* `i`*-ésimo de la lista con el* `j`*-ésimo si* $i < j$ *pero el* `i`*-ésimo elemento de la lista es mayor que el* `j`*-ésimo.*

b) *Define una segunda función* `ordenar1(L)` *que, usando la del apartado anterior, devuelva la lista* `L` *ordenada. Observa que para que la lista quede ordenada hay que comparar cada elemento de la lista con* TODOS *los siguientes, es decir, necesitamos un bucle doble para recorrer los pares* (i, j) *con* $i < j$*.*

2. SEGUNDO ALGORITMO (MERGESORT):

Ejercicio 5.

a) *Este es un algoritmo recursivo que también utiliza una función auxiliar.*
Define una función de Sage `intercalar(L,L1,L2)` *de tres listas que haga lo siguiente: si el primer elemento de* `L1` *es menor o igual que el primero de* `L2` *debe quitar el primer elemento de* `L1`*, añadirlo a* `L` *y volver a llamar a la función* `intercalar(L,L1,L2)`*; en caso contrario debe quitar el primer elemento de* `L2`*, añadirlo a* `L` *y volver a llamar a la función* `intercalar(L,L1,L2)`*. Las recursiones deben parar cuando una de las dos listas (*`L1` *o* `L2`*) sea vacía y devolver la suma de las tres listas* `L+L1+L2`*.*

b) *Define una función de Sage* `ordenar2(L)` *que en los casos* `len(L)=0` *o* `len(L)=1` *devuelva la lista* `L`*, y devuelva* `intercalar([],L1,L2)` *con* `L1` *y* `L2` *las listas que se obtienen aplicando* `ordenar2` *a las dos "mitades" de* `L` *(si el número de elementos de* `L` *es impar una de las "mitades" tiene un elemento más que la otra). Es importante observar que cuando llamamos a* `intercalar` *las listas* `L1` *y* `L2` *ya están ordenadas.*

c) *Compara la eficiencia de los dos métodos anteriores generando una lista de* 800 *enteros aleatorios comprendidos entre* -1000 *y* 1000

```
L = [randint(-1000,1000) for muda in srange(800)]
```

y midiendo el tiempo que el ordenador tarda en ordenarlas

```
time ordenar1(L); time ordenar2(L)
```

3. TERCER ALGORITMO (QUICKSORT):

a) Si la longitud de la lista L es menor o igual que 1 la devolvemos.

b) Si la longitud en $n > 1$ creamos tres listas: $L1$ contiene todos los elementos de L que son menores que $L[0]$, $L2$ contiene todos los elementos de L que son mayores que $L[0]$, y $L3$ contiene todos los elementos de L que son iguales a $L[0]$.

c) Aplicamos recursivamente el procedimiento a $L1$ y $L2$, para obtener $M1$ y $M2$, y devolvemos $M1 + L3 + M2$.

Ejercicio 6.

a) *Define una función ordenar(L) que devuelva la lista L ordenada y utilice el algoritmo propuesto. Comprueba que realmente ordena una lista de longitud* 20 *formada por enteros aleatorios del intervalo* $[-100, 100]$.

b) *Sage dispone del método L.***sort***() para ordenar listas. Genera una lista de* 10^5 *números aleatorios del intervalo* $[-10^6, 10^6]$ *y compara los resultados y tiempos usando tu función y la de Sage.*

c) *Una variante de este algoritmo consiste en utilizar en el paso 2 la comparación con un elemento aleatorio de la lista L en lugar de comparar con el primero. Implementa esta variante y estudia si es más eficiente o no que el original.*

d) *Queremos encontrar la menor diferencia entre dos enteros cualesquiera pertenecientes a una lista L de enteros. Una idea consiste en ordenar la lista y luego buscar la menor diferencia entre enteros consecutivos de la lista ordenada. Define una función que haga esto.*

4. Cuarto algoritmo (Bucket sort):

   a) La función que vamos a definir tiene el formato $ordenar(L, i, j)$ con L la lista a ordenar e i, j enteros del intervalo $[0, len(L) - 1]$.

   b) Comparamos el elemento $L[i]$ con $L[j]$, y si este último es menor los intercambiamos.

   c) Si $j-i+1 > 2$ aplicamos la función (*ordenar*) a L con $j-(j-i+1)//3$ en lugar de j y sin cambiar el valor de i, luego con $i+(j-i+1)//3$ en lugar de i y sin cambiar el valor de j, y finalmente, otra vez con $j-(j-i+1)//3$ en lugar de j sin cambiar el valor de i.

   d) Devolvemos L.

   Ejercicio 7.

   a) *Define la función $ordenar(L, i, j)$ propuesta. Cuando se invoca en la forma $ordenar(L, 0, n)$, con $n = len(L) - 1$ debe devolver la lista L ordenada.*

   b) *Comprueba que realmente ordena una lista de longitud 20 formada por enteros aleatorios del intervalo $[-100, 100]$.*

   c) *Sage dispone del método $L.sort()$ para ordenar listas.Genera una lista de 10^3 números aleatorios del intervalo $[-10^6, 10^6]$ y compara los resultados y tiempos usando tu función y la de Sage.*

   d) *En el apartado anterior obtendrás tiempos grandes, del orden de un minuto, y queremos estudiar el motivo. Modifica tu función para incluir un contador que nos informe del número de veces que se ha producido el intercambio del punto b en la descripción del algoritmo.*

En esta página de la Wikipedia puedes encontrar descripciones de diversos algoritmos para ordenar listas, y es un buen ejercicio programar algunos y comparar resultados.

4.6. Ejercicios

Cuando en los ejercicios que siguen se utilice un bucle **for** hay que intentar usar las dos modalidades mostradas en las páginas 52 y 55.

1. Mejorar los programas `gen_subcadena` para que la salida $C1$ contenga información acerca de la posición que ocupa cada dígito de los que añadimos a $C1$.

2. Un par de enteros primos se dicen "gemelos" si difieren en 2 unidades. Definir una función de n que devuelve una lista de todos los pares de primos gemelos menores que n.

3. Define una función, de dos enteros k y a, que cuente el número de primos gemelos en cada uno de los subintervalos $[kt, k(t+1)]$ de longitud k dentro del intervalo $[0, ka]$.

4. Define una función que cuente el número de enteros primos menores que N de la forma $n^2 + 1$ para algún n.

5. Dado un entero N define una función de N que devuelva el subintervalo $[a, b]$ de $[1, N]$ más largo tal que no contenga ningún número primo.

6. Dado un entero, que escribimos en la forma $N = 10 \cdot x + y$ con y la cifra de las unidades, definimos $F(N) := x - 2 \cdot y$.

   a) Demuestra que N es múltiplo de 7 si y sólo si $F(N)$ lo es.

   b) Estudia las órbitas de F en el conjunto de los enteros positivos, tratando de enunciar un criterio de divisibilidad de N entre 7 en términos de la órbita de N mediante F.

7. Consideramos la función $F : \mathbb{N}^* \to \mathbb{N}^*$ que a cada entero positivo le asocia la suma de los cuadrados de sus dígitos.

   a) Estudia las órbitas de F, tratando de determinar el *comportamiento a largo plazo* de las iteraciones de F.

   b) Después de obtener tus conclusiones acerca de las órbitas, demuéstralas formalmente. Para eso es importante entender cómo decrece un entero N, al aplicarle F, dependiendo de su número de dígitos.

8. Queremos estudiar las órbitas de la función $f(n) := n^2$ en el conjunto finito de clases de restos módulo m. En particular, queremos determinar las órbitas cerradas (también llamadas *ciclos*), es decir, órbitas de elementos n tales que aplicando repetidas veces f volvemos a obtener el mismo n. Puede haber ciclos que consistan en un único elemento n tal que $f(n) = n$, y decimos en este caso que n es un punto fijo de f.

   a) Define una función de Sage $ciclos(f, m)$ que devuelva una lista de todos los ciclos de f en el conjunto de clases de restos módulo m, cada ciclo una lista de los enteros que forman parte del ciclo. En principio, no hay problema en que haya ciclos repetidos en la lista que devuelve tu función.

   b) Experimenta con los resultados de tu función, para diversos valores de m primos y compuestos, y trata de obtener conjeturas razonables sobre los ciclos de f.

9. Encontrar la condición necesaria y suficiente sobre el entero n para que $1 + 2 + 3 + \cdots + n$ divida exactamente a $n!$. Una vez encontrada hay que intentar demostrar la equivalencia.

10. Encontrar la condición necesaria y suficiante para que existan infinitos múltiplos de un entero n que se escriban únicamente con unos en base 10. Una vez encontrada hay que intentar demostrar la equivalencia. ¿Ocurrirá algo similar si queremos que infinitos múltiplos se escriban usando únicamente otro dígito (2 , 3, etc.)?

11. La cifra dominante de un entero es la primera por la izquierda que no es nula. Encontrar los dígitos que pueden ser la cifra dominante al mismo tiempo de 2^n y 5^n, con el mismo exponente n. Una vez encontrados, hay que intentar demostrar que esos dígitos son los únicos posibles.

12. Demuestra que para $n > 1$ el entero $n^4 + 4$ es siempre compuesto.

13. En el sitio *web* `projecteuler` pueden encontrarse los enunciados de casi 500 problemas de programación. Dándose de alta en el sitio es posible, pero no es lo recomendable, ver las soluciones propuestas para cada uno de ellos.

    Tal como están enunciados muchos de ellos son difíciles, ya que no se trata únicamente de escribir código que, en principio, calcule lo que se pide, sino que debe ejecutar el código en un ordenador normal de sobremesa, en

un tiempo razonable, para valores de los parámetros muy altos. Es decir, lo que piden esos ejercicios es que se resuelva el problema mediante un código correcto y muy eficiente. Sin embargo, en primera aproximación, podemos intentar producir código sintáctica y semánticamente correcto, que no es poco, y dejar el asunto de la eficiencia para más adelante.

Muchos de estos problemas de programación tienen una base matemática fuerte, y eso es lo que sobre todo nos interesa.

a) Goldbach conjeturó que todo entero compuesto e impar es la suma de un primo y el doble de un cuadrado. Así, por ejemplo, $9 = 7 + 2 \cdot 1^2$, $15 = 7 + 2 \cdot 2^2$, $21 = 3 + 2 \cdot 3^2$, etc. Esta conjetura resultó ser falsa. Determina el menor entero que no cumple lo conjeturado por Goldbach.

b) Existen enteros, por ejemplo 145, que son iguales a la suma de los factoriales de sus dígitos. Determina todos los enteros con esta propiedad.

c) Determina todas las tripletas de enteros primos de 4 cifras tales que cumplen las dos condiciones siguientes:

1) Los 3 enteros están en progresión aritmética, es decir, el segundo menos el primero es igual al tercero menos el segundo.
2) Los tres enteros de la tripleta tienen las mismas cifras y cada cifra aparece el mismo número de veces en cada uno de ellos.

Por ejemplo, $(1487, 4817, 8147)$ es una de las soluciones.

d) Sea (a, b, c) una tripleta de enteros positivos tal que existe un triángulo rectángulo con la longitud de los lados igual a los enteros de la tripleta. Podemos decir, por ejemplo, que una tal tripleta es rectangular. Sea p el perímetro de un tal triángulo. Para $p \leq 1000$ determina el perímetro p_m para el que existe el mayor número de tripletas rectangulares distintas con ese perímetro.

e) Un **primo de Mersenne** es un entero primo de la forma $2^p - 1$ con p primo. No es difícil ver que si $2^n - 1$ es primo, entonces el exponente n debe ser también primo, pero el recíproco es falso. Los primos de Mersenne son los mayores conocidos porque hay criterios de primalidad bastante eficientes para candidatos a ser primo de Mersenne.

En 2004 se descubrió un primo muy grande que no es de Mersenne, concretamente se trata de $P := 28433 \times 2^{7830457} + 1$.

1) Determina el número de cifras decimales de P.
2) Determina las últimas, por la derecha, diez cifras decimales de P.
3) Determina las primeras, por la izquierda, diez cifras decimales de P.

Se entiende que, aunque fuera posible calcular completamente P, debe hacerse este ejercicio sin pasar por ese cálculo.

f) Hay polinomios de grado 2 que, como $p(x) := x^2+x+41$, toman valores primos para los primeros valores enteros consecutivos de x. El polinomio $p(x)$ indicado toma valores primos para $x = 0, 1, 2, \ldots, 39$, pero el valor es compuesto para $x = 40$. Entonces, para $p(x)$ se obtiene una sucesión de 40 primos al evaluarlo en los enteros consecutivos del intervalo $[0, 39]$.

Determina, de entre todos los polinomios de la forma $x^2 + ax + b$, con a y b de valor absoluto menor o igual a 1000, el polinomio que produce el mayor número de valores primos al evaluarlo en enteros consecutivos $x = 0, 1, 2, \ldots$. Indica también el número de valores primos obtenido, que será mayor o igual a 40.

g) Otros ejercicios del proyecto Euler que podemos recomendar son los números

$$2, 4, 8, 10, 12, 21, 24, 23, 26, 28, 29, 30, 35, 36, 37, 41, 45, 50, 57.$$

Son ejercicios del comienzo de la lista y, en general, son m'as sencillos que los del final. Otros pocos, situados alrededor del número 400 serían

h) Ejercicio #401: Pide que se evalúe la suma de todos los cuadrados de los divisores de un número n, y luego que se sumen esas sumas para n entre 1 y N. Puede ser difícil evaluar esas sumas eficientemente para N muy grande.

i) Ejercicio #413: Es un ejercicio en el manejo de cadenas de caracteres.

j) Ejercicio #414: En este se utilizan listas y el cálculo de órbitas.

k) Ejercicio #421: Factorización de enteros y de polinomios. Sage será de gran ayuda para resolverlo.

l) Ejercicio #429: Otro ejercicio sobre divisores de un entero.

14. Algunos programas recursivos son muy parecidos a una demostración *por inducción*. Consideremos, por ejemplo, el primer ejercicio en la lista de ejercicios del capítulo 2:

Se trata de comprobar, por inducción, que

$$1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}. \tag{4.1}$$

En la demostración por inducción debemos ver que:

a) *Caso inicial:* $1^2 = \dfrac{1 \cdot 2 \cdot 3}{6}$, que es claro.

b) Que si suponemos,*hipótesis inductiva*, que

$$1^2 + 2^2 + \cdots + (n-1)^2 = \frac{(n-1)n(2n-1)}{6},$$

entonces también debe ser cierta (4.1).

Define recursivamente una función de Sage, por ejemplo **def** comprueba(**N**):..., que sirva para asegurarnos de que la fórmula (4.1) es cierta desde $n = 1$ hasta $n = N$,

a) Debemos usar el *caso inicial* como condición de parada de la recursión. Entonces, debemos tener un bloque **if N** == 1: ...

b) En lugar de la *hipótesis de inducción* tenemos que incluir en algún lugar del programa una llamada recursiva a la propia función comprueba(**N**−1).

Escribe programas recursivos para los ejercicios de demostración por inducción de la lista mencionada.

Apéndice C

Máquina virtual

En computación una *máquina virtual* es un programa que se ejecuta en otra máquina, virtual o física, y que simula el funcionamiento de un ordenador. Al proceso de ejecutar aplicaciones dentro de máquinas virtuales se le llama *virtualización* de aplicaciones, y ciertamente es una tecnología que está actualmente bastante de moda.

Por ejemplo, en las máquinas de nuestro Laboratorio no está instalado *MS Windows*©, pero como hay algunos cursos en los que se usa es necesario que, al menos, se pueda ejecutar en una máquina virtual. En nuestro caso usamos el programa *VirtualBox* que nos permite ejecutar MS Windows©, y sus programas, dentro de Linux[1].

En esta sección vamos a estudiar el funcionamiento de la máquina virtual [2] más simple que existe, que aún así es capaz de ejecutar cualquier algoritmo finito.

Comenzamos describiendo la máquina:

1. Un ordenador necesita *memoria RAM* para almacenar los datos sobre los que trabaja. Normalmente, también tiene un *disco duro (HD)* o *disco de estado sólido (SSD)* en el que almacenar permanentemente esos datos.

   Nuestra máquina virtual va a almacenar sus datos en una lista de enteros no negativos de longitud arbitraria. Esta lista, a la que llamaremos `datos`, va a simular al mismo tiempo la memoria RAM y el disco duro.

2. Los ordenadores tienen *procesadores (CPU)* en los que se ejecutan las *instrucciones* contenidas en los *programas (software)*. Los datos que el procesador lee de la memoria RAM o del disco son modificados, procesados, y devueltos a la memoria o al disco.

   Nuestros programas serán *listas de instrucciones* elementales y describiremos el procesador de nuestra máquina virtual detallando las instrucciones que admite:

   a) El primer tipo de instrucciones, lo llamaremos `'A'`, va a sumar una unidad a un cierto elemento de la lista `datos` y a continuación va a saltar a otra instrucción del programa que estamos ejecutando.

   Una instrucción de tipo `'A'` consistirá en una lista de longitud 3 de la forma [`'A'`,i,j], con i un entero no negativo que determina un elemento, `datos[i]` de la lista `datos` y `j` otro entero no negativo que nos indica la instrucción del programa que se va a ejecutar a continuación.

[1] Aunque pueda parecer increíble, es posible ejecutar software de Windows© dentro de Linux, usando por ejemplo *Wine*, de manera bastante más rápida que directamente en Windows©. Puedes leer una breve explicación el los párrafos 2 y 3 de esta página.

[2] Técnicamente, esta máquina es un ejemplo de *máquina de registros* conocida como "modelo ábaco" de Lambek.

b) Una instrucción de tipo `'S'` va a restar una unidad a un cierto elemento de la lista `datos` y a continuación va a saltar a otra instrucción del programa que estamos ejecutando.

Sin embargo, y esto es MUY IMPORTANTE, como los elementos de la lista `datos` no pueden ser negativos, cuando nos encontremos que el elemento al que habría que restarle una unidad ya es cero no restamos y saltamos a otra instrucción directamente.

En consecuencia, las instrucciones de tipo `'S'` son listas de longitud 4 de la forma `['S',i,j,k]` con `i,j` y `k` enteros no negativos: `i` un índice de la lista `datos`, `j` la instrucción a la que saltamos si `datos[i]`$\neq 0$ y k la instrucción a la que saltamos si `datos[i]=0`.

c) Finalmente, una instrucción de tipo `'E'` es una instrucción de fin del programa y consiste de una lista de longitud 1 que únicamente contiene el carácter `'E'`. Cuando el procesador se encuentra una instrucción de tipo `'E'` para la ejecución del programa.

3. Por último, los ordenadores necesitan un *sistema operativo (OS)* que gestione la asignación de recursos físicos para la ejecución de los programas.

   En nuestra máquina virtual el sistema operativo consistirá en una función que tendrá como argumentos un programa y una lista de datos, modificará la lista de datos ejecutando en sucesión las instrucciones del programa y devolverá el estado final de la lista de datos.

Implementamos esta máquina virtual como un programa en Python, con dos funciones una que describe el procesador y la otra el sistema operativo:

```
def procesador(instruccion,datos,end):
    if instruccion[0] == 'A':
        datos[instruccion[1]] += 1
        estado = instruccion[2]
    elif instruccion[0] == 'S':
        if datos[instruccion[1]] != 0:
            datos[instruccion[1]] -= 1
            estado = instruccion[2]
        else:
            estado = instruccion[3]
    elif instruccion[0] == 'E':
            estado = end
    return datos,estado

def sistema_op(programa,datos):
    estado = 0
    end = len(programa)
    while estado != end:
        datos,estado = procesador(programa[estado],datos,end)
    return datos
```

Ya podemos comenzar a programar nuestra máquina:

```
programa1=[['S',0,1,2],['A',1,0],['E']]
programa2=[['S',0,1,3],['A',1,2],['A',2,0],['S',2,4,5],['A',0,3],['E']]
```

¿Qué hacen estos dos programas? Lo primero que debemos determinar, para cada uno de ellos, es la longitud de la lista `datos` que procesan. Para el primer programa vemos que el máximo del segundo elemento en cada instrucción es 1 luego la lista de datos debe ser de longitud dos (puede ser de longitud mayor pero entonces el programa no modificará los elementos a partir del tercero), mientras que para el segundo vemos que la longitud de la lista de datos es tres.

A continuación, ejecuta en papel cada uno de los dos programas, usando enteros pequeños como elementos de la lista de datos, y trata de entender su funcionamiento y uso.

Comentarios:

1. Nuestra máquina virtual tiene un conjunto de instrucciones minimal, lo que hace que los programas sean simples pero muy largos. Hablando muy en general, nuestra máquina virtual tiene un procesador de tipo RISC (Reduced Instruction Set Computer) mientras que los ordenadores personales tienen casi todos procesadores CISC (Complex Instruction Set Computer).

   Un programa que se va a ejecutar en un procesador RISC consistirá en muchas más instrucciones elementales que uno para CISC, pero estas instrucciones son mucho más sencillas y se pueden ejecutar más rápido. Típicamente, los procesadores RISC ejecutan más instrucciones por ciclo de reloj que los CISC lo que se compensa con tener que ejecutar más instrucciones.

2. Por otra parte, el tipo de programas que utilizamos en nuestra máquina virtual se conoce como de *bajo nivel*, lo que significa que se utilizan directamente las instrucciones que admite el procesador.

   En los lenguajes de *alto nivel* que usamos habitualmente para programar es necesario utilizar un *compilador o intérprete* que traduce cada una de las instrucciones de nuestro programa de alto nivel a un montón de instrucciones del procesador.

   El lenguaje de programación C es *compilado*, lo que significa que el código pasa por un programa compilador que produce un ejecutable en el que las instrucciones ya son las del procesador. En cambio Python es un lenguaje *interpretado* que usa un programa, llamado intérprete, que va traduciendo *sobre la marcha* las instrucciones del programa a instrucciones del procesador.

   Los lenguajes de programación interpretados son mucho más flexibles que los compilados, pero también son más lentos.

3. Debe ser claro que con las instrucciones `'A'` y `'S'` podemos ejecutar bucles **for** y crear ramificaciones en la ejecución del programa similares a los **if**. Es un poco limitado porque el único caso en que la ejecución del programa se bifurca es en las instrucciones de tipo `'S'`: si la instrucci'on va a actuar sobre `datos[i]` hace algo distinto si ese valor es cero o si no lo es.

   Disponiendo de **for** e **if** la máquina virtual puede, de hecho, ejecutar cualquier algoritmo finito, aunque los programas, para casi cualquier algoritmo, serían inmensos.

4. Puede parecer que la máquina virtual simula una calculadora y no un ordenador. Sin embargo, puede realizar manipulaciones arbitrarias sobre cadenas de bits y, por tanto, es un ordenador.

Ejercicios:

Programar esta máquina virtual es un buen ejercicio cuando se comienza a programar sin experiencia previa. En particular, se ven bastante bien los errores como bucles infinitos, programas sintácticamente correctos pero semánticamente incorrectos, etc.

1. Programa, usando como ayuda los dos ejemplos suministrados, el producto de dos enteros positivos.
2. Programa la resta de dos enteros n y m con $n \geq m$.
3. Programa la suma y resta con enteros positivos o no.
4. Programa la división con resto de enteros positivos.
5. Programa las potencias de exponente entero de enteros positivos.

Apéndice D

Tortuga

El lenguaje de programación LOGO fué desarrollado en el MIT, comezando en los años 60, como un lenguaje bien adaptado a la enseñanza de los conceptos básicos de la programación de ordenadores. Aunque se trata de un lenguaje bastante completo, la parte que se hizo más popular fué su módulo gráfico, *la tortuga,* que ya desde finales de los 60 se ha utilizado en algunas escuelas para enseñar a programar a niños pequeños.

Originalmente la tortuga era un robot móvil controlado por un programa escrito en LOGO y que podía dibujar en el suelo figuras porque disponía de un bolígrafo. Pronto se pasó de dibujar en el suelo con un robot a dibujar en la pantalla del ordenador.

La programación de gráficas es una buena manera de comenzar a programar porque vemos inmediatamente el resultado del programa y, si no corresponde a lo que queremos, vamos corrigiéndolo y estudiando en cada modificación cómo cambia el resultado. En el caso en que no hayas programado antes, es muy recomendable que dediques un tiempo a practicar con la tortuga.

Python dispone de un módulo, *turtle*, que permite programar la tortuga usándolo en lugar de LOGO, y que funciona desde dentro de Sage. Desgraciadamente, es imposible que funcione en el servidor de Sage y se debe ejecutar siempre en la máquina local, bien en el Laboratorio o en tu propia máquina.

Para que funcione la tortuga en Sage hay que instalar un paquete. En Ubuntu o Debian se hace mediante la instrucción

```
sudo apt-get install tk tk-dev python-tk.
```

Podemos comprobar que funciona correctamente evaluando en una celda de Sage

```
import turtle as tt
tt.forward(1500)
```

La primera línea carga en la hoja el módulo `turtle` de Python, y la segunda debe abrir una nueva ventana en la que hay dibujada una flecha horizontal cuya longitud depende del argumento 1500.

Todas las intrucciones de control de la tortuga deben tener el formato `tt.instruccion(parámetros).` Con la segunda línes la tortuga avanza al frente 150 unidades y se para. La flecha es horizontal porque, por defecto, la tortuga está inicialmente *mirando al Este.*

La tortuga tiene un eje de simetría, y un sentido dentro de ese eje, y su orientación viene determinada por el ángulo que el eje forma con la horizontal medido en sentido contrario a las agujas del reloj.

Instrucciones básicas del paquete `turtle:`

1. Las instrucciones tt.window_width(x) y tt.**window_height**(y) fijan el tamaño,en píxeles, de la ventana en la que se va a mover la tortuga. Un pixel es el tamaño de un punto, en realidad un cuadradito, en la pantalla

que usemos, y por tanto, depende de ella. En el laboratorio la resolución de las pantallas es *****.

2. Movimiento:

   a) La instrucción tt.**forward**(x) hace avanzar la tortuga x píxeles al frente, mientras que tt.**back**(x) la hace retroceder.

   b) La instrucción tt.**left**(a) gira el eje de la tortuga a grados en sentido contrario a las agujas del reloj, mientras que tt.**right**(a) lo hace en sentido de las agujas.

   Si queremos cambiar a radianes basta ejecutar tt.**radians**(), y todas las medidas de ángulos, a partir de ese momento, serán en radianes.

   c) La velocidad con la que se mueve la tortuga se controla con tt.**speed**(arg), con `arg` igual a *"fast", "normal", "slow", o "slowest".*

3. Posición:

   a) tt.**pos**() devuelve las coordenadas de la posición de la tortuga en el momento en que se ejecuta.

   b) tt.**heading**() devuelve el ángulo que la tortuga, su eje, forma con la horizontal, medido en grados y en el sentido contrario a las agujas del reloj. Si se ha ejecutado en la hoja tt.**radians**() la medida en es radianes. ¿Cómo se volverá a medidas es grados?

   c) tt.**setposition**(x,y) lleva la tortuga al punto de coordenadas (x, y) en línea recta y dibujando salvo que se haya levantado el bolígrafo mediante tt.**penup**(), y no se haya bajado posteriormente.

   d) tt.**setheading**(a) deja el eje de la tortuga formando un ángulo a con la horizontal.

4. Bolígrafo:

   a) La instrucción tt.**penup**() levanta el bolígrafo del papel, deja de dibujar, mientras que tt.**pendown**() lo vuelve a bajar.

   b) tt.**pencolor**(color) sirve para cambiar el color de tinta del bolígrafo. El color puede ser un nombre, como *"red", "blue","green", etc.* o, más en general, el código hexadecimal de un color RGB. Puedes ver algunos de esos códigos en esta tabla.

   c) tt.**pensize**(ancho) determina el ancho en píxeles del trazo del bolígrafo.

En caso de necesidad, puedes consultar el conjunto completo de instrucciones del módulo `turtle`.

Es importante entender que el movimiento de la tortuga se determina siempre *localmente* respecto a la posición que ocupa en cada momento. Es como si la tortuga llevara encima un sistema de referencia con respecto al que se interpretan en cada momento las instrucciones que vienen a continuación.

Ejemplos y ejercicios

1.
```
def poligono(n,R):
    '''Poligono regular de n lados inscrito
       en una circunferencia de radio R'''
    tt.radians()
    A = 2*pi.n()/n
    L = 2*sin(A/2)*R ##Lado del pol\'{\i}gono
    tt.penup()
    tt.forward(R)
    tt.left((pi.n()+A)/2)
    tt.pendown()
```

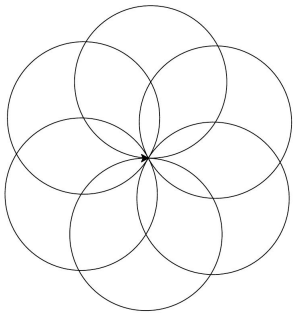
```
    for j in srange(n):
      tt.forward(L)
    tt.left(A)
    tt.penup()
    tt.home()
```

Si tomamos n grande la *gráfica parece una circunferencia.*

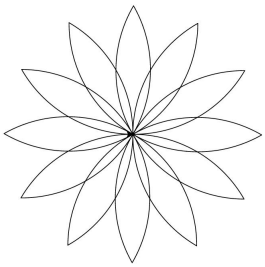
2. Modifica el programa anterior para que, partiendo de la posición en que esté la tortuga, dibuje un arco de α radianes en una circunferencia de radio R.

   ¿Hace falta dar como dato el centro de la circunferencia? No hace falta si entendemos que la posición original de la tortuga determina la recta tangente, en el punto inicial, a la circunferencia que queremos dibujar. Llamemos L a esa recta, que es el eje de simetría de la propia tortuga en su posición inicial. Entonces el centro de la circunferencia es un punto en la recta perpendicular a L que dista R del punto inicial. Hay dos de esos puntos, uno en cada semiplano de los determinados por L. Entonces hay dos arcos que resuelven el problema, y podemos definir dos programas llamados, por ejemplo, arcoL y arcoR, uno dibuja el arco girando hacia la izquierda y el otro hacia la derecha.

3. Escribe un programa que produzca esta figura:



4. Escribe otro programa que produzca esta otra figura:



5. Escribe un programa que produzca una casa como las que dibujan los niños pequeños, hecha de segmentos y arcos de circunferencia. Intenta producir el mayor número posible de los detalles que suelen incluir esas casas (el árbol, el camino que lleva a la puerta, ¿el humo de la chimenea?, etc.)

6. ¿Continuará?

Capítulo 5

Complementos

En este capítulo se recogen pequeñas introducciones a algunos temas, relativos a la programación, más avanzados. Algunos de ellos los usaremos alguna vez a lo largo del curso, y otros no los usaremos pero nos parece conveniente que se sepa de su existencia. Concretamente,

1. Se discuten brevemente los cambios entre bases de los SISTEMAS DE NUMERACIÓN. Puede ser innecesario si se ha visto el tema en el Bachillerato.
2. TRUCOS: en esta sección se recogen algunas ideas para programar que, posiblemente, son reutilizables. Esperamos que pueda ir creciendo a lo largo del curso.
3. FUERZA BRUTA: cuando no se nos ocurre algo mejor siempre podemos intentar un método de *fuerza bruta*. En problemas finitos, o que podamos aproximar mediante un problema finito, será posible este tipo de tratamiento, aunque seguramente muy poco eficiente.
4. CÁLCULO EN PARALELO: Sage dispone de un procedimiento básico para ejecutar en paralelo bucles **for** en máquinas con varios núcleos. No es verdadero paralelismo porque no hay *comunicación* entre los núcleos, pero en condiciones favorables nos puede ahorrar tiempo de espera.
5. EFICIENCIA: se discute la forma de medir tiempos de ejecución y uso de la memoria RAM, junto con métodos, Cython y numpy, para mejorar dramáticamente la eficiencia del cálculo numérico en Sage. Hay que tener en cuenta que los tiempos de cálculo que obtenemos al evaluar la eficiencia dependen mucho de la máquina que estemos usando.

   Esta sección intenta resumir el contenido de parte del Bloque II en las notas de Pablo Angulo mencionadas en el Prólogo.
6. ITERADORES: por último, dentro de la subsección sobre la memoria RAM, se mencionan los *iteradores* o *generadores* como métodos para ahorrar RAM en la ejecución de bucles.

5.1. Sistemas de numeración

Elegida una base $b \geq 2$, podemos representar los números enteros como "polinomios" en b con coeficientes los dígitos en base b, es decir, símbolos que representan los enteros entre cero y $b-1$. Si $b \leq 10$ se utilizan los dígitos decimales habituales, y si $b > 10$ se utilizan a continuación letras. Así por ejemplo, si la base b es 16 los dígitos son

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F,$$

con A representando el dígito en base 16 que corresponde al decimal 10, B el que corresponde al 11, etc.

Un número entero escrito en base 16 puede ser $7C041F$ que corresponde al número en base 10

$$15 + 1 \cdot 16^1 + 4 \cdot 16^2 + 0 \cdot 16^3 + 12 \cdot 16^4 + 7 \cdot 16^5 = 8127519.$$

Es claro que usando una base más grande se consigue una notación mucho más compacta.

Las expresiones que representan un entero en una base dada no son verdaderos polinomios porque la variable ha sido sustituida por la base del sistema de numeración, y los coeficientes no son arbitrarios sino *dígitos* menores que la base.

Sin embargo, las operaciones entre polinomios, suma, multiplicación y división con resto, y entre enteros en una base de numeración b son esencialmente las mismas. Así, por ejemplo, el algoritmo para efectuar la división con resto es prácticamente el mismo para polinomios y para números enteros.

Una explicación más detallada, con algunos ejercicios, de este basunto puede encontrarse en este documento.

5.1.1. Cambios de base

Habitualmente usamos el sistema de numeración decimal, con base $b = 10$, pero podríamos usar cualquier otro, y en computación se usan, sobre todo, los sistemas binario ($b = 2$), octal ($b = 8$) y hexadecimal ($b = 16$).

Los cambios de base de numeración son simples, y los hacemos pasando a través de la base 10:

1. Un entero escrito en base b, $a_n a_{n-1} a_{n-2} \dots a_1 a_0$ se pasa a base 10 evaluando en base 10 su correspondiente polinomio en la variable b

   $$a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + a_{n-2} \cdot b^{n-2} + \cdots + a_1 \cdot b + a_0.$$

   Hemos visto un ejemplo con $b = 16$ al comienzo de esta sección.

2. Al revés, si tenemos un entero N en base 10 y lo queremos pasar a base b, es decir, escribirlo en la forma

   $$a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + a_{n-2} \cdot b^{n-2} + \cdots + a_1 \cdot b + a_0,$$

   debemos, en primer lugar dividir N entre b y el resto es el dígito de las unidades a_0 en base b. Esto se debe a que podemos escribir

   $$a_n \cdot b^n + \cdots + a_1 \cdot b + a_0 = b \cdot (a_n \cdot b^{n-1} + \cdots + a_1) + a_0.$$

   Para calcular el segundo dígito a_1 debemos, dividir el primer cociente $a_n \cdot b^{n-1} + \cdots + a_1$ entre la base b y el resto es a_1. Continuamos en la misma forma hasta llegar a un cociente menor que b, y en ese momento el proceso necesariamente para y hemos obtenido la expresión en base b de N.

5.1.2. Sistemas de numeración en Sage

En Sage podemos obtener los dígitos en una base b de un entero decimal D mediante la instrucción (D).**digits**(base=b) que devuelve una lista de los dígitos ordenados según potencias decrecientes de b, es decir, el primer elemento de la lista es el dígito con exponente de b más alto.

Las instrucciones **bin**(D), **oct**(D) y **hex**(D) devuelven cadenas de caracteres con la expresión de D en las bases 2, 8 y 16.

Para convertir un entero en base b a decimal podemos usar

ZZ('expresion en base b',b),

con el número en base b entre comillas porque debe ser una cadena de caracteres. Entonces, por ejemplo, ZZ(**hex**(D),16) debe devolver D.

5.1.3. La prueba del 9 y otros trucos similares

En el sistema de numeración decimal es fácil calcular el resto de la división de un entero entre 9 sin hacer la división. Ocurre que todas las potencias de 10 tienen resto uno al dividir entre 9 (comprobarlo). Entonces, el resto de dividir entre 9 el entero D es el mismo que el resto de dividir entre 9 la suma de las cifras decimales de D. Podemos así calcular ese resto sumando las cifras y cada vez que superamos 9, volver a sumar las cifras de esa suma, hasta que dejemos de superar 9:

```
k=2312348912498
suma=sum(k.digits())
while suma>9:
    print suma,
    suma=sum(suma.digits())
suma, suma==k %9
```

```
56 11
(2, True)
```

Cualquier operación aritmética entre enteros, por ejemplo una división con resto $D = d \cdot c + r$, se puede comprobar rápidamente cambiando cada entero z por su resto $[z]$ al dividir entre 9 ("tomando clases de restos módulo 9") para obtener que debe ser $[D] = [d] \cdot [c] + [r]$. Si esta última igualdad no se cumple podemos asegurar que la división está mal hecha, mientras que el recíproco no es siempre cierto.

El mismo argumento nos permite calcular el resto de la división de un entero D entre 3 (o 9) como el resto de la división entre 3 (o 9) de la suma de sus cifras.

5.2. Trucos

En esta sección se recogen algunos *trucos* de programación útiles, es decir reutilizables, y, también, hay enlaces a otras zonas del documento en las que se pueden encontrar otros.

En la hoja de Sage `51-COMPL-trucos.ipynb` puedes ver algunos ejemplos relacionados con esta sección.

5.2.1. Potencias

Queremos calcular, en la forma más eficiente posible, una cierta potencia a^n de un entero a. Multiplicando sin más efectuaríamos $n-1$ productos, y querríamos reducir el número de productos para tratar de acortar el tiempo de cálculo.

Una forma de hacerlo es expresando el exponente en "base 2", es decir, como una suma de potencias de 2, para obtener

$$a^n = a^{2^{n_0}+2^{n_1}+\cdots+2^{n_k}} = a^{2^{n_0}} \cdot a^{2^{n_1}} \dots a^{2^{n_k}}.$$

Una vez que hayamos calculado las potencias $a^{2^{n_i}}$ bastará efectuar $k-1$ productos para terminar el cálculo.

¿Cómo calcular de manera eficiente potencias de la forma a^{2^k}?

Basta observar que

$$a^{2^k} = ((\dots(((a^2)^2)^2)^{\cdots})^2,$$

donde el número de veces que se eleva al cuadrado es exactamente k. En consecuencia, podemos calcular a^{2^k} con sólo k productos: el primero para elevar a al cuadrado, el segundo para elevar a^2 al cuadrado, etc.

El número total de multiplicaciones para calcular a^n sería $(\sum_{i=0}^{k} n_i)+k-1$. El número máximo de multiplicaciones que tenemos que hacer, suponiendo que el exponente n es un número de N bits (i.e. se expresa en base dos con N ceros o unos), es el que corresponde al del n que se expresa en base 2 como N unos ($n = 2^N - 1$), y resulta ser

$$0 + 1 + 2 + 3 + \cdots + (N-1) + N - 1 = \frac{N(N-1)}{2} + N - 1.$$

Esta idea puede expresarse mediante un programa recursivo bastante sencillo

```
def potencia(a,k):
    if k==0:
        return 1
    elif k %2 == 0:
        b = potencia(a,k/2)
        return (b*b)
    else:
        b = potencia(a,(k-1)/2)
        return (a*b*b)
```

Si el exponenente k es una potencia de 2 el programa nunca entra en el **else**, y vemos que este programa recursivo está implementando las ideas de la discusión anterior.

Es fácil comprobar que este programa es muchísimo más eficiente que uno que fuera acumulando los productos parciales en una variable, y en cada vuelta del bucle multiplicara el valor acumulado por a. Por ejemplo, este programa puede, en un ordenador de sobremesa estándar, elevar 77 al exponente 2^{30} en 152 segundos, mientras que el que usa el acumulador no terminó el cálculo en 12 horas. El motivo tiene que ser que el algoritmo para multiplicar números grandes (¿cuál es?) es muy eficiente de forma que es muchísimo mejor hacer pocas multiplicaciones, aunque sean de números muy grandes, que hacer muchas multiplicaciones, del orden de mil millones, de un número grande, el que está en la variable que va acumulando los resultados parciales, por uno pequeño como 77.

El algoritmo es especialmente eficiente cuando lo que nos interesa es calcular, sin calcular a^k, el resto (a^k %m) de la división de a^k entre un entero m. En ese caso podemos efectuar todas las operaciones *módulo* m, es decir, cada vez que al operar superamos m podemos quedarnos con el resto de dividir entre m.

```
def potencia_mod(a,k,m):
    if k==0:
        return 1
    elif k%2 == 0:
        b = potencia_mod(a,k/2,m)
        return (b*b)%m
    else:
        b = potencia_mod(a,(k-1)/2,m)
        return (a*b*b)%m
```

Esta *aritmética módulo* m no sobrecarga tanto los recursos del ordenador como la aritmética entera con números grandes, y así es posible calcular en 176 segundos

```
potencia_mod(7777^(1234),2^(157),10991^(987654)+1)
```

en el que elevamos un entero que tiene unas 4800 cifras decimales a otro que tiene 47 módulo un entero que tiene unos cuatro millones de cifras. ¿Cómo se

calcula el número de cifras decimales que tiene un número de la forma a^n? ¿Cuántas cifras decimales tiene el entero

$$(7777^{1234})^{2^{157}}?$$

En Sage podemos usar la instrucción **power_mod**(a,**n**,m) para calcular, por un procedimiento que debe ser[1] el mismo que usa potencia_mod(a,**n**,m), el resto de dividir a^n entre m.

Esta operación, elevar un entero grande a un exponente grande módulo otro entero también grande, es, como veremos en el capítulo **??**, usado en criptografía para encriptar mensajes, mientras que la operación inversa, el *logaritmo discreto*, es decir el cálculo del exponente n tal que a^n módulo m es un entero prefijado M, es clave para romper el sistema que encripta mediante potencias.
.

5.2.2. Listas binarias

Supongamos que queremos iterar sobre todas las listas binarias, ceros o unos, de longitud k. Sabemos que el número de tales listas es 2^k. Una manera es *anidar* k bucles **for** con la variable en cada uno de ellos tomando valor 0 ó 1. Ésto no es práctico si k es grande, y no es factible si queremos que k sea un parámetro en una función de Sage.

Una alternativa es generar la lista de todas las listas binarias de longitud k e iterar sobre ella, lo que se puede hacer fácilmente usando el método **digits**(base=2).

Concretamente, se puede usar un código como el siguiente

```
def funcion(k):
    L = []
    for m in srange(2^k):
        L.append(m.digits(base=2,padto=k))
    for L1 in L:
        ....
        ....
```

donde el primer bucle genera la lista de todas las listas binarias de longitud k y el segundo itera sobre los elementos de esa lista un bloque de instrucciones que no se ha escrito.

¿Para qué nos puede servir este truco? Intenta escribir, sistemáticamente, todos los polinomios de grado $< k$ con coeficientes en $\mathbb{Z}_j = \{0, 1, 2, \ldots, j-1\}$ sin usar el truco y usándolo.

5.2.3. Archivos binarios

Un archivo de texto con N caracteres ocupa $8N$ bits, ceros o unos, o, lo que es lo mismo, N bytes. Cada caracter se codifica como una cadena de 8 bits usando el código ASCII.

Entonces, si generamos una cadena binaria de N bits, N ceros o unos, y la guardamos en un archivo obtendremos uno de $8N$ bits (N bytes), ya que los ceros o unos los trata como un caracter cualquiera, el cero se representa en binario como 00110000 y el uno como 00110001.

Para comprimir archivos, con métodos como `.zip`, necesitamos que un archivo que contenga N bits ocupe N bits, y no $8N$ bits. El truco para conseguir esto consiste en dividir el archivo en trozos de 8 bits y escribir al archivo NO LOS 8 CEROS O UNOS sino el caracter que corresponde a los 8 bits en ASCII. De esta forma lo que realmente se escribe al archivo es la cadena de bits original y el archivo resultante ocupa N bits o $N/8$ bytes.

[1] Tarda prácticamente lo mismo en hacer el cálculo anterior.

5.2.4. La Enciclopedia de sucesiones de enteros

La *Enciclopedia de sucesiones de enteros* es un *lugar web*[2] que permite acceder y buscar dentro de una inmensa base de datos de sucesiones de números enteros.

En particular, permite, en ocasiones, identificar números reales de los que hemos calculado las primeras cifras decimales usando el ordenador, sin más que escribir las cifras decimales que creemos seguras del número real buscado, separadas por comas, en la ventana que encontramos en la página de acceso.

La Enciclopedia permite identificar sucesiones de enteros, no sólo de dígitos, como por ejemplo, la sucesión de Fibonacci y es enormemente útil al realizar *experimentos matemáticos.*

5.2.5. Enlaces a otras zonas del documento

1. Usando conjuntos podemos eliminar elementos repetidos en una lista: primero convertimos la lista L en conjunto mediante A=**set**(L), y a continuación el conjunto en lista mediante L1=**list**(A).

2. Veremos ejemplos en los que es rentable calcular un número entero muy grande pasando a través de los números reales. Uno típico se encuentra al calcular un número de Fibonacci muy grande sin calcular los anteriores (ver la sección **??**).

3. Usando logaritmos podemos calcular, ver la sección **??**, el número de cifras que tendrá en una base de numeración dada b un número de la forma a^k, sin necesidad de calcular a^k. También usando logaritmos es posible calcular la *cifra dominante* de un entero de la forma a^k, es decir, su primer dígito por la izquierda, sin necesidad de calcular a^k.

4. Por supuesto, podemos calcular de manera eficiente la cifra de las unidades de a^k, en una base cualquiera b, mediante la instrucción potencia_mod(a,k,b) que hemos visto en esta sección. ¿Cómo podríamos calcular la segunda cifra por la derecha o la tercera?

5. Hemos visto al menos dos ejemplos, `mergesort`, y en la definición de la función potencia en esta misma sección, en los que el problema se presta a un tratamiento recursivo especialmente eficiente.

   Esto ocurre porque en cada llamada recursiva a la propia función se divide el tamaño de los datos por dos, y se consigue así que la profundidad del árbol recursivo de llamadas sea del orden de **log**(n,base=2) con n el entero que "mide el tamaño de los datos", por ejemplo, si el dato es una lista, como en mergesort, n sería su longitud, y en la función potencia n sería el exponente que, en este caso, también hemos llamado n.

6. En la sección del capítulo 3 dedicada a los diccionarios se explica cómo se pueden usar los diccionarios de Sage para estructurar la información de forma que las búsquedas sean rápidas. El procedimiento es similar a la ordenación *lexicográfica* de las palabras en un diccionario de papel, que es lo que nos permite encontrar una palabra concreta rápidamente.

7. ¿Continuará?

5.3. Fuerza bruta

En muchos problemas matemáticos tenemos que encontrar, en un cierto conjunto X, un elemento x que cumple una cierta propiedad P.

Si X es finito, podemos construirlo en el ordenador, y resolver el problema recorriendo el conjunto hasta que encontremos un elemento que cumple P. Decimos que este método para resolver el problema es de *fuerza bruta.*

[2] Originalmente se publicó en papel.

Por ejemplo, en criptografía podemos haber interceptado un mensaje que sabemos que ha sido encriptado con un cierto método que conocemos. Entonces "conocemos" el conjunto finito de todas las posibles claves para desencriptarlo y, en teoría, podemos ir probando todas hasta que encontremos una clave tal que al desencriptar con ella se obtenga un mensaje legible. Es claro que cualquier sistema criptográfico debe tener un conjunto de claves posibles tan grande que sea imposible desencriptar mediante fuerza bruta. Este método puede funcionar en algunos casos, pero, como veremos hay sistemas criptográficos que son inmunes a él.

Como método es claro que es bastante "bruto", no tenemos que pensar mucho para aplicarlo, y, por otra parte, está limitado por el tamaño máximo de los X que podamos manejar en nuestro ordenador. Puede ser que el X que nos interese sea demasiado grande para que pueda caber en la memoria RAM, o bien que el proceso de comprobar si x cumple la propiedad P requiera demasiado tiempo, de forma que el tiempo total de cálculo pudiera ser demasiado grande, meses o años, para que nos interese estudiar el problema así.

El problema de la RAM se resuelve en parte usando **iteradores**, que recorren los elementos del conjunto X "uno a uno" sin construir en RAM el conjunto completo. Si es posible construir los elementos de X usando iteradores nos quedaría, en muchos casos, sólo el segundo problema cuya solución es armarnos de paciencia o bien tratar de *paralelizar* el problema.

En los ordenadores que estamos usando, mi experiencia es que suele ser viable tratar estructuras de datos, por ejemplo listas, de un tamaño del orden de 10^6, pero suelen aparecer problemas cuando se llega a 10^7.

5.4. Cálculo en paralelo

Las máquinas que tenemos actualmente en el Laboratorio disponen de un procesador con dos núcleos (2 *cores*), lo que significa que *en teoría* pueden realizar cálculos usando los dos al mismo tiempo en la mitad del tiempo. De hecho, gracias a una técnica llamada *hyperthreading*, el sistema operativo de la máquina cree que tiene cuatro núcleos con cada núcleo físico soportando dos virtuales, pero, como debemos esperar, los tiempos que se obtienen calculando con cuatro son prácticamente iguales a los que se obtienen con dos.

Veremos al menos un par de ejemplos de cálculo en paralelo, uno en un ejemplo de ataque mediante fuerza bruta de un sistema criptográfico (capítulo **??**) y el otro el cálculo aproximado de áreas planas usando puntos aleatorios (capítulo **??**).

Es claro que la mayor parte de los programas pasan casi todo su tiempo ejecutando bucles y podemos paralelizar un bucle dividiendo su rango en tantos trozos como núcleos y mandando un trozo a cada núcleo. Sin embargo:

1. Al dividir el trabajo entre los núcleos es posible, dependiendo del problema, que varios necesiten los mismos datos, o bien que un núcleo necesite en algún momento resultados que calcula otro. Aparece entonces un problema *logístico* serio: los núcleos pueden tener que comunicarse entre ellos, y si se organiza mal es posible que haya muchos núcleos esperando que les llegue información que necesitan para continuar su trozo del cálculo.

2. En máquinas *multicore* el problema de la comunicación se resuelve usando zonas de memoria RAM compartida por todos los núcleos (*shared memory*). El *software* estándar para controlar el acceso a la memoria compartida se llama **OpenMP**.

3. También se puede hacer cálculo paralelo en *clusters* de ordenadores que se comunican enviándose mensajes a través de la red local. Esto es bastante más complicado que el caso anterior, y se hace usando un protocolo llamado **MPI** del que existen varias implementaciones.

4. Hay bucles completamente *imparalelizables*, por ejemplo, si cada vuelta del bucle necesita el resultado de la anterior. ¿Se te ocurre algún ejemplo?

   Si dividimos el trabajo entre los núcleos, en uno de esos casos imparalelizables, puede ocurrir que sólo uno pueda calcular en cada momento y los otros van a estar esperando a que les llegue su turno. El tiempo de ejecución va a ser igual o mayor que si ejecutamos el bucle en un sólo núcleo.

5. Algoritmos matemáticos complejos pueden ser muy difíciles de paralelizar, de manera que sistemas como Sage utilizan casi siempre un único núcleo. Si el cálculo es muy largo podemos observar, usando el *System monitor*[3], que el núcleo en uso va cambiando de tiempo en tiempo. Eso lo hace el sistema operativo, no Sage, para evitar el sobrecalentamiento del procesador.

6. Bucles que pueden ser divididos en trozos completamente independientes pueden siempre ejecutarse en paralelo, pero es conocido como *paralelismo vergonzante* para indicar que para practicarlo no es necesario pensar mucho.

   Sage dispone de una implementación de esta clase de paralelismo y es lo único que veremos en este curso sobre el asunto.

7. Incluso el paralelismo vergonzante tiene un problema: el tiempo que tarda uno de los núcleos en ejecutar una vuelta del bucle puede depender del tamaño de los datos, y en tales casos el número de vueltas que asignamos a cada núcleo puede ser distinto ya que tenemos que *equilibrar la carga que soporta cada núcleo*. Veremos algún ejemplo de esta situación a continuación, junto con un truco que permite, en ocasiones, equilibrar la carga fácilmente.

Consideremos, por ejemplo un problema en el que debemos elevar al cuadrado un gran número de matrices, de tamaño creciente m, con entradas enteros aleatoriamente elegidos. Haciendo el cálculo en un único núcleo obtenemos, por ejemplo, un tiempo

```
def elevar2_m(m):
    L = [randint(0,1000) for i in srange(m*m)]
    M = matrix(ZZ,m,m,L)
    return M*M

time LL = map(elevar2_m,srange(1,200))
```

```
Time: CPU 22.98 s, Wall: 23.27 s
```

En este ejemplo, el tamaño de las matrices varía entre 1×1 y 199×199. En una máquina con 4 núcleos podemos utilizarlos todos mediante

```
@parallel(4)
def cuadrado(L):
    map(elevar2_m,L)
time LL =
list(cuadrado([srange(1,50),srange(50,100),srange(100,150),srange(150,200)]))
```

```
Time: CPU 0.13 s, Wall: 14.79 s
```

En este reparto de la carga el primer núcleo tiene que tratar con matrices de tamaño mucho menor y, por tanto, acabará antes su tarea. La carga está muy desequilibrada. El tiempo de cálculo es el que se obtiene para el núcleo que más tarda, es decir el que calcula con tamaños entre 150 y 200.

[3] `Aplicaciones/Herramientas del sistema/Monitor del sistema`

```
@parallel(4)
def cuadrado(L):
        map(elevar2_m,L)
L1 = [m for m in xrange(1,200) if m%4 == 0]
L2 = [m for m in xrange(1,200) if m%4 == 1]
L3 = [m for m in xrange(1,200) if m%4 == 2]
L4 = [m for m in xrange(1,200) if m%4 == 3]
time LL = list(cuadrado([L1,L2,L3,L4]))
```

```
Time: CPU 0.02 s, Wall: 5.82 s
```

Mediante este truco conseguimos que la carga se equilibre, los cuatro núcleos tardan más o menos lo mismo y ese es el tiempo total resultante.

Puedes comprobar lo anterior, los tiempos por supuesto dependen del ordenador que estemoa usando, en la hoja `52-COMPL-paralelo.ipynb`.

5.5. Eficiencia

Debemos tratar de escribir programas SINTÁCTICAMENTE CORRECTOS, es decir, que el intérprete de Python acepte como válidos y no muestre mensajes de error, y SEMÁNTICAMENTE CORRECTOS, es decir, que calculen lo que pretendemos calcular. Es claro que un programa que no cumple estas dos condiciones no nos sirve.

Pero además, nuestros programas deben ser EFICIENTES, no deben tardar más tiempo del necesario ni utilizar más memoria RAM de la necesaria. En esta sección discutimos algunos aspectos de la posible mejora en la eficiencia de nuestros programas.

Una de las reglas básicas, que ya discutimos en la sección 4.2.2, es que DEBEMOS CALCULAR EL MÍNIMO NECESARIO PARA PODER RESPONDER a la pregunta que nos hacemos: si queremos calcular la longitud de una lista en lugar de calcular la lista completa y, una vez está en memoria, calcular su longitud debemos usar un "contador" que se incrementa según vamos calculando elementos que deberíamos añadir a la lista pero sin generar la lista.

De la misma forma, si queremos calcular la suma (producto) de una serie de números es mejor irlos sumando (multiplicando) en un "acumulador" que generar la lista completa y luego sumar (multiplicar).

5.5.1. Control del tiempo

Para empezar, podemos usar la instrucción **time** al comienzo de una línea de código en la que se ejecute una función propia de Sage o definida por nosotros. Al mostrarnos la respuesta también muestra dos tiempos: el tiempo de CPU y el llamado *Wall time* que corresponde al tiempo transcurrido desde que empezó el cálculo hasta que terminó. No coinciden necesariamente, aunque si somos el único usuario y no estamos ejecutando otros programas aparte de Sage prácticamente coincidirán, porque el ordenador puede dedicar algo del tiempo de CPU a otras labores no relacionadas con nuestro cálculo.

En segundo lugar tenemos la instrucción **timeit**, que se ejecuta en la forma **timeit**(`'instruccion'`), y difiere de **time** en que ejecuta la instrucción varias veces y devuelve un promedio. La instrucción admite varios parámetros (averiguar con la ayuda interactiva para **timeit**) que controlan el número de repeticiones, etc.

Por último, **cProfile** produce un resultado mucho más detallado que nos permite saber cómo se distribuye el tiempo total de cálculo entre las distintas funciones que se ejecutan dentro de nuestro programa.

Se invoca el `profiler` mediante

```
import cProfile, pstats
cProfile.runctx("funcion(n,m,...)", globals(), locals(), DATA + "Profile.prof")
s = pstats.Stats(DATA + "Profile.prof")
s.strip_dirs().sort_stats("time").print_stats()
```

y hay un par de ejemplos de su uso en la hoja de Sage `53-COMPL-eficiencia-tiempo.sws` que acompaña esta subsección.

5.5.2. Control de la RAM

Programas que construyen grandes estructuras de datos, por ejemplo listas enormes, pueden llegar a saturar la memoria RAM de la máquina en la que estemos trabajando. Se pueden ver los incrementos en el uso de RAM mediante la instrucción **get_memory_usage**() que nos devuelve la cantidad de memoria RAM, en megabytes (MB), que está en uso en el momento en que se invoca.

Usándola, como en la hoja `54-COMPL-eficiencia-ram.sws` que acompaña esta subsección, podemos ver los incrementos en memoria RAM al generar grandes estructuras de datos.

5.5.2.1. Iteradores

1. Muchas estructuras de datos son *iterables*, es decir, podemos crear un bucle **for** que recorra uno por uno los elementos de la estructura de datos y para cada uno de esos elementos ejecute un bloque de instrucciones. Todas las estructuras de datos básicas, que vimos en el capítulo 3, listas, tuplas, cadenas de caracteres, conjuntos y diccionarios, son iterables.

   Sin embargo, esta forma de iterar funciona creando en memoria la estructura de datos completa y, a continuación, recorriéndola. Si, por ejemplo, la lista es enorme va a ocupar una gran cantidad de memoria RAM, y crearla y recorrerla puede ser un procedimiento muy ineficiente.

2. Estos inconvenientes se resuelven en parte con los *iteradores* o *generadores*, que en lugar de crear la estructura de datos en memoria para luego iterar sobre ella, van generando elementos de uno en uno y cada vez que tienen uno ejecutan el bloque de instrucciones del bucle sobre él.

   Es claro que esta forma de iterar debe ser mucho más eficiente, al menos en términos de memoria RAM.

3. El iterador básico es **xsrange**(k) que genera enteros de la lista **srange**(k) de uno en uno. Si por ejemplo definimos gen = **xsrange**(10^8), no se crea la lista sino únicamente el generador gen, y ejecutando varias veces gen.**next**() podemos ir viendo los enteros sucesivos `0,1,2,...`.

   Iteramos sobre un generador de la misma manera que sobre una lista: **for** j **in xsrange**(10^7):....

4. Es posible crear nuevos generadores usando la sintaxis breve para bucles:

   ```
   gen2 = (f(x) for x in xsrange(10^8) if Condition)
   ```

   que produce un generador nuevo, a partir del básico **xsrange**(10^8), y filtrado por la condición booleana `Condition`. Nótese que el generador va delimitado por paréntesis en lugar de corchetes.

5.5.3. Cython

Python no se compila al lenguaje de máquina, sino que se interpreta (i.e. se va traduciendo a lenguaje de máquina sobre la marcha al irse ejecutando), lo que hace que, en general, código escrito en Python se ejecute mucho más lentamente que código escrito en lenguajes que se compilan.

Sin embargo, modificando muy poco código escrito en Python es posible compilarlo automáticamente a través de C, con lo que se consiguen mejoras impresionantes en su eficiencia. Cuando un programa se compila, por ejemplo en C, es necesario que el código reserve explícitamente la memoria que se va a utilizar durante el cálculo. Para eso existen *tipos de datos* predefinidos que ocupan cantidades prefijadas de memoria RAM.

cython traduce código escrito en Python a C y, una vez que se declaran los tipos de las variables, consigue mejoras importantes en el rendimiento. Para usarlo dentro de Sage basta escribir en la primera línea de la celda %**cython** y declarar los tipos de las variables. Por ejemplo

```
%cython
def cuadrado(double x):
    cdef int a=1
    return x*x+a
```

define una función de **cython** en la que x está declarada como un real de doble precisión y a como un entero. Los tipos más usados son **int** para enteros, **long** para enteros grandes, **float** para decimales y **double** para decimales de precisión doble.

Cuando el programa incluye además listas o matrices grandes es conveniente, y se consiguen enormes mejoras adicionales, usar los correspondientes objetos de **numpy** en lugar de los propios de Sage.

Pueden verse algunos ejemplos de estas mejoras en las hojas `55-COMPL-cython.sws` y `56-COMPL-planeta-cython.sws`. En la primera se discute la manera de conseguir grandes cantidades de números aleatorios de manera eficiente, es decir, usando librerías de funciones compiladas en *C*, y volveremos sobre este asunto en el capítulo **??**, Probabilidad, en el que será crucial.

En la segunda se simula, usando métodos elementales, el movimiento de un planeta alrededor del Sol de acuerdo a las Leyes de Newton. Si te interesan los detalles de este asunto puedes leer, y es absolutamente recomendable, este capítulo en el libro de Física de Feynman.

5.5.4. Numpy

Python dispone de un número grande de módulos adicionales que amplían sus capacidades y resuelven algunos de sus problemas. **numpy** está diseñado para permitir operaciones muy rápidas con listas y matrices, de hecho con *arrays* de cualquier dimensión, y permite, en conjunción con Cython, utilizar Python (y Sage), en Cálculo Numérico.

En este curso veremos muy poco Cálculo Numérico ya que hay una asignatura específica en el segundo cuatrimestre, y además en ella se utiliza Matlab en lugar de Sage, pero sería perfectamente factible utilizar la combinación Sage+cython+**numpy** para conseguir los mismos resultados que con Matlab[4] en cuanto a eficiencia.

Para poder usar **numpy** dentro de Sage debemos evaluar una celda con el contenido **import numpy as** np, y una vez hecho esto una línea como A = np.**array**([[1,2],[1,1]]) define la matriz A como una matriz de **numpy**. Una vez que definimos una lista o matriz como de **numpy**, las operaciones que hagamos con ella serán operaciones de **numpy** y, por tanto, muy optimizadas.

La sintaxis de **numpy** es algo diferente a la de Sage, por ejemplo A∗A para matrices de **numpy** es el producto elemento a elemento mientras que el producto de matrices se obtiene mediante A.**dot**(A), y, en este curso no usaremos sistemáticamente, aunque quizá sí puntualmente, **numpy** para el cálculo con matrices.

En la hoja `57-COMPL-numpy.sws` se pueden ver un par de ejemplos que comparan los tiempos de ejecución de códigos que utilizan **numpy** con códigos similares que no lo utilizan.

Un ejemplo más se encuentra en la hoja `58-COMPL-fractal-cython-numpy.sws`, que contiene un programa escrito por Pablo Angulo, dedicada a la obtención de gráficas de conjuntos fractales, y en particular del conjunto de Mandelbrot.

[4] Matlab, al contrario que Sage, no es un programa gratuito sino bastante caro y es la Universidad quien paga las licencias que nos permiten usarlo en el Laboratorio. Además, el motor de cálculo de Matlab consiste en librerías para Álgebra Lineal, muy optimizadas, que están en el dominio público hace mucho tiempo.

Puedes leer la página de la Wikipedia sobre el conjunto de Mandelbrot y esta otra sobre la noción general de conjunto fractal. Además en la wiki de Sagemath puede encontrarse esta página, en la que hay varios ejemplos de construcción de fractales con Sage, junto con ejemplos sobre cómo hacer las gráficas interactivas.

Apéndice E

Recursos

E.1. Bibliografía

1. Documentación producida por sagemath: Los desarrolladores de Sage producen y mantienen actualizada una cantidad inmensa de documentación sobre el sistema. Puede verse completa en este enlace.

   Hemos seleccionado y la que nos parece más interesante, de forma que no es necesario decargarla. Hay zonas de estos documentos que todavía no han sido escritas, es decir, la documentación parece estar siempre en desarrollo.

   Por otra parte, parte de estos documentos se refieren a partes de las matemáticas mucho más avanzadas, y que, por supuesto, no nos conciernen en este curso.

   a) En primer lugar tenemos el *tutorial* de Sage, del que los tres primeros capítulos son lectura muy recomendable. Su contenido se puede solapar en parte con el del comienzo de estas notas, pero encontraréis allí multitud de ejemplos cortos que ayudan a comenzar con Sage.

   b) El documento `sage-power.pdf` contiene información más avanzada sobre el uso de Sage.

   c) Este otro documento contiene información sobre el uso de Sage en campos concretos de las matemáticas. En particular, pueden ser útiles la secciones `6.1.7` (teoría de números y criptografía), `6.1.1` (teoría de grafos) y el capítulo 5 (técnicas de programación).

   d) En este documento se explica cómo funcionan muchos de los objetos abstractos, grupos, anillos,etc., que se pueden construir y manipular en Sage.

2. Otros:

   a) Un libro interesante de introducción a Sage. "Desgraciadamente" está en francés, pero los ejemplos de código sirven.

   b) Otro texto con aplicaciones, sobre todo, a las matemáticas de la etapa preuniversitaria.

   c) Este texto se centra en las matemáticas de los primeros cursos de Universidad.

3. Python:

   Sage incluye varios sistemas preexistentes de cálculo simbólico y el *pegamento* que los hace funcionar juntos es Python. Cuando queremos sacarle partido a Sage necesitamos utilizar Python para definir nuestras propias

funciones, que frecuentemente incluyen funciones de Sage dentro. Python es entonces también el *pegamento* que nos permite obtener de Sage respuestas a problemas que no estaban ya preprogramados.

   a) En primer lugar tenemos aquí un *tutorial* de Python. Utiliza la versión 3 del lenguaje, mientras que Sage todavía usa la 2.7.
   b) El *tutorial* escrito por el autor, Guido van Rossum, del lenguaje. También para la versión 3 y bastante avanzado.
   c) Una muy buena introducción al lenguaje. Hay una versión del texto que utiliza Python 3. Allen B. Downey es también el autor de este libro, en el que se aplica Python a diversos problemas, como los autómatas celulares, relacionados con la *teoría de la complejidad*. Este último texto contiene muchas ideas y ejercicios interesantes, y lo hemos usado para preparar la parte final del curso.

4. Bibliografía por materias:

   a) Un estupendo resumen del uso de Sage en matemáticas, sobre todo en cálculo y representaciones gráficas.
   b) Un curso de cálculo diferencial que utiliza Sage masivamente. No contiene cálculo integral.
   c) En este sitio *web* (parte de la documentación oficial de Sage) puede verse otra introducción al estudio del cálculo diferencial usando Sage.
   d) Un curso *tradicional* de Álgebra lineal que contiene una extensión sobre el uso de Sage para resolver problemas.
   e) Un texto bastante completo de criptografía que utiliza Sage para realizar los cálculos.
   f) Un curso muy popular, llamado CHANCE, de introducción a la teoría de de probabilidades. Los primeros capítulos pueden ser de alguna utilidad cuando lleguemos al capítulo **??**.

E.2. Otros

1. Ya se mencionaron los chuletarios de Sage como forma rápida de acceder a la sintaxis de las principales instrucciones preprogramadas.

2. También se indicó en el prólogo de las notas que este curso se basa, en gran medida, en sus primeras versiones que montó Pablo Angulo. Puedes ver el curso original en este enlace.

3. El sitio *web* `rosettacode.org` es especialmente útil cuando se sabe ya programar en algún lenguaje pero se pretende aprender uno nuevo. Consiste en una serie grande de problemas resueltos en casi todos los lenguajes disponibles, en particular, casi todos están resueltos en los más populares como *C* y Python.

   A fin de cuentas, casi todos los lenguajes tienen bucles **for** y **while**, bloques **if** y recursión. Hay entre ellos pequeñas diferencias en la sintaxis y en la forma en que se manejan las estructuras de datos.

4. En el sitio *web* `projecteuler` pueden encontrarse los enunciados de casi 500 problemas de programación. Dándose de alta en el sitio es posible ver las soluciones propuestas para cada uno de ellos.

   Tal como están enunciados muchos de ellos son difíciles, ya que no se trata únicamente de escribir código que, en principio, calcule lo que se pide, sino que debe ejecutar el código en un ordenador normal de sobremesa, en un tiempo razonable, para valores de los parámetros muy altos. Es decir, lo que piden esos ejercicios es que se resuelva el problema mediante un código correcto y muy eficiente.